

# MUSIC MANAGEMENT SYSTEM

Suprajasai Konegari, Supriya Konegari

## INTRODUCTION

Music Management System is a dedicated platform crafted to meet the diverse needs of artists, record labels, and music enthusiasts alike. This robust system seamlessly integrates powerful features, providing users with a dynamic space to explore, manage, and elevate their musical journeys. Users can seamlessly log in to our system, unlocking a world of musical possibilities tailored to their tastes and preferences. Users have the power to curate their own musical experience by adding songs to favorites and creating personalized playlists. We have simplified the music discovery process with a user-friendly search button. Users have the ability to run various queries on the database through the dedicated "Queries" tab. Additionally, users can be up-to-date on the musical trends with our system's observation features.

## DATABASE DESIGN

Our project consists of the following tables:

Tables	Schemas
Artist	<u>Artist ID</u> , Name, Birthdate, Nationality, Gender, Height
Label	<u>Label ID</u> , Label_Name, Country, Founded_Year, Headquarters
Album	<u>Album ID</u> , Album_Title, Release_Date, <u>Artist ID</u> , <u>Label ID</u>
Genre	<u>Genre ID</u> , Genre_Name
Song	<u>Song ID</u> , Song_Title, Time_Period, Musical_Key, <u>Album ID</u> , <u>Artist ID</u> , <u>Genre ID</u>
Users	<u>User ID</u> , User_Name, Email, User_Password, Date_Of_Birth
Playlists	<u>Playlist ID</u> , Playlist_Name, Created_Date, <u>User ID</u>
User_Song_Rating	<u>User Song Rating ID</u> , Rating, <u>User ID</u> , <u>Song ID</u>
User_Favourite_Song_and_Album	<u>ID</u> , <u>Song ID</u> , <u>Album ID</u>
Billboard_Chart_Data	<u>Chart ID</u> , <u>Song ID</u> , Daily_Rank, Daily_Movement, Weekly_Movement
Song_Features	<u>Feature ID</u> , Danceability, Energy, Loudness, Mode, Speechiness, Acousticness, Instrumentalness, Liveness, Valence, Tempo, Time_Signature, <u>Song ID</u>
Awards	<u>Award ID</u> , Award_Name, <u>Song ID</u>

## **DESIGN DECISIONS**

### **1.Many-to-Many Relations:**

In our database model, we've identified a many-to-many relationship between users and their favorite albums. A user can have multiple favorite albums, and conversely, an album can be present in the favorite lists of many users. To handle this, we introduced an intermediary relation, "Album\_UserFavouriteSong&Album," effectively maintaining the association between users and their favorite albums.

### **2.One-to-Many Relations:**

- a. Between Artist and Song: An artist can produce multiple songs, establishing a one-to-many relationship between artists and songs. This allows for a comprehensive representation of an artist's body of work within the database.
- b. Between User and User Song Rating: Users can rate multiple songs, while each song is rated by only one user, creating a one-to-many relationship. This facilitates the tracking of individual user preferences and ratings.
- c. Between User and Playlist: A user can create multiple playlists, forming a one-to-many relationship between users and playlists. This design accommodates the diverse playlists users may curate for different occasions.
- d. Between Genre and Song: A genre can encompass multiple songs, constituting a one-to-many relationship. This allows for the classification of songs into distinct genres for better organization and categorization.
- e. Between Song and Song Features: Each song has distinct features, resulting in a one-to-many relationship between songs and their features. This separation enables a more granular analysis of individual song characteristics.
- f. Between Song and UserFavouriteSong&Album: Songs can be present in multiple users' favorite lists or albums, establishing a one-to-many relationship. This connection ensures that users can collectively appreciate and engage with popular songs.

### **3.Many-to-One Relations:**

- a. Between Label and Album: A label can produce many albums, but each album is associated with only one label, forming a many-to-one relationship. This delineation facilitates the organization of albums under specific record labels.
- b. Between Album and Artist: An album is created by one artist, establishing a many-to-one relationship between albums and artists. This ensures a clear attribution of each album to a specific artist.

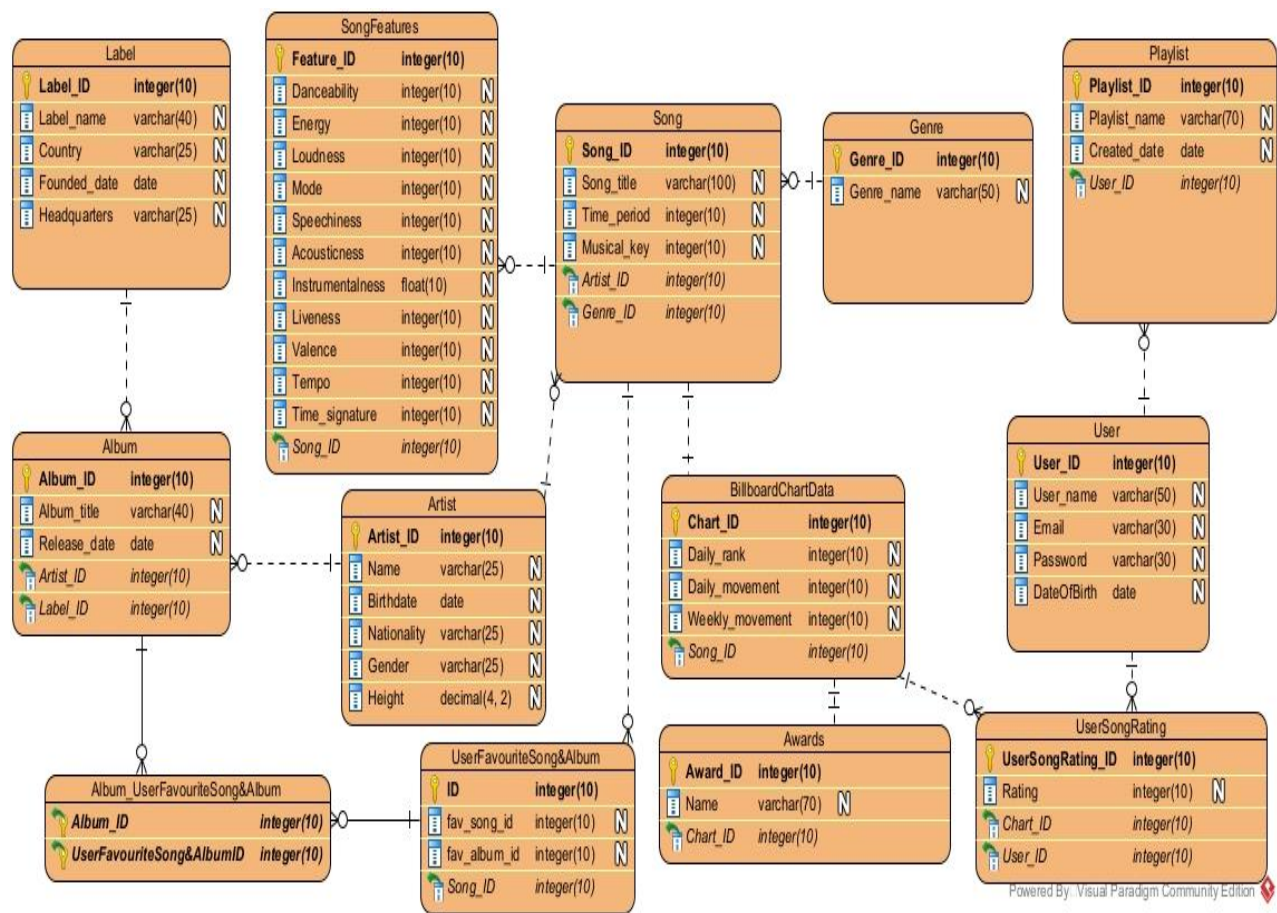
### **4.One-to-One Relations:**

- a. Between Song and Billboard Chart Data: Each song corresponds to specific Billboard chart data, creating a one-to-one relationship. This connection allows for a detailed examination of a song's chart performance.
- b. Between Awards and Billboard Chart Data: Similarly, there is a one-to-one relationship between awards and Billboard chart data. This ensures a direct association between a song's awards and its chart performance.

## 5.Normalization in the 3rd Normal Form:

In the pursuit of a well-structured database, we ensured that there are no partial dependencies and no transitive dependencies. By adhering to the principles of the third normal form, we minimized redundancy and enhanced data integrity, contributing to a robust and efficient database design.

### FINAL ERD



## USE OF TRIGGERS, FUNCTIONS, STORED PROCEDURES AND VIEWS:

### STORED PROCEDURES

In our project, there are 11 stored procedures.

1. When a new user registers, this procedure adds the users details in the users table.

```

DELIMITER $$
CREATE PROCEDURE AddNewUser(
    IN p_user_name VARCHAR(255),
    IN p_email VARCHAR(255),
    IN p_user_password VARCHAR(512),
    IN p_date_of_birth DATE,
    OUT p_user_id INT
)
BEGIN
    INSERT INTO users (User_Name, Email, User_Password, Date_Of_Birth)
    VALUES (p_user_name, p_email, p_user_password, p_date_of_birth);
    SELECT LAST_INSERT_ID() INTO p_user_id;
END $$
DELIMITER ;

```

2. This stored procedure searches for song titles based on a provided search term.

```

DELIMITER //
CREATE PROCEDURE SearchSongs(IN search_term VARCHAR(100))
BEGIN
    SELECT * FROM song
    WHERE song_title LIKE CONCAT('%', search_term, '%');
END //
DELIMITER ;

```

3. This procedure will retrieve information of a specific artist given their Artist\_ID.

```

DELIMITER $$
CREATE PROCEDURE GetArtistInformation(
    IN artistID INT,
    OUT artistName VARCHAR(255),
    OUT artistBirthdate DATE,
    OUT artistGender VARCHAR(10),
    OUT artistHeight DECIMAL(3,2),
    OUT artistNationality VARCHAR(255)
)
BEGIN
    SELECT Full_Name, Birthdate, Gender, Height, Nationality
    INTO artistName, artistBirthdate, artistGender, artistHeight, artistNationality
    FROM Artist
    WHERE Artist_ID = artistID;
END $$
DELIMITER ;

```

4. This procedure gives a list of top rated songs based on user ratings.

```

DELIMITER $$
CREATE PROCEDURE GetTopRatedSongs(IN topCount INT)
BEGIN
SELECT s.Song_ID,s.Song_Title,s.Time_Period,s.Musical_Key,s.Album_ID,s.Artist_ID,s.Genre_ID,
      AVG(us.Rating) AS Average_Rating
FROM Song s
JOIN User_Song_Rating us ON s.Song_ID = us.Song_ID
GROUP BY s.Song_ID, s.Song_Title, s.Time_Period, s.Musical_Key, s.Album_ID, s.Artist_ID, s.Genre_ID
ORDER BY Average_Rating DESC
LIMIT topCount;
END $$
DELIMITER ;

```

5. This procedure updates information about a specific artist.

```

DELIMITER $$
CREATE PROCEDURE UpdateArtistInformation(
IN artistID INT,
IN newFullName VARCHAR(255),
IN newBirthdate DATE,
IN newGender CHAR(1),
IN newHeight DECIMAL(5,2),
IN newNationality VARCHAR(255))
BEGIN
UPDATE Artist
SET Full_Name = newFullName, Birthdate = newBirthdate, Gender = newGender, Height = newHeight,
    Nationality = newNationality
WHERE Artist_ID = artistID;
END $$
DELIMITER ;

```

6. This procedure will retrieve billboard chart data for a specific song.

```

DELIMITER $$
CREATE PROCEDURE GetBillboardChartData(
IN songID INT)
BEGIN
SELECT Chart_ID, Daily_Rank, Daily_Movement, Weekly_Movement
FROM Billboard_Chart_Data
WHERE Song_ID = songID;
END $$
DELIMITER ;

```

7. This procedure calculates the average duration of songs for each genre in the database and store the results in a temporary table called AvgDurationsTable.

```

DELIMITER $$
CREATE PROCEDURE GetAverageSongDurationForEachGenre()
BEGIN DECLARE done INT DEFAULT FALSE;
    DECLARE genreID INT;
    DECLARE genreName VARCHAR(255);
    DECLARE avgSongDuration DECIMAL(10,2);
    DECLARE cur CURSOR FOR
        SELECT g.Genre_ID, g.Genre_Name, AVG(s.Time_Period) AS Avg_Song_Duration
        FROM Genre g
        JOIN Song s ON g.Genre_ID = s.Genre_ID
        GROUP BY g.Genre_ID, g.Genre_Name;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    CREATE TEMPORARY TABLE IF NOT EXISTS AvgDurationsTable ( Genre_ID INT, Genre_Name VARCHAR(255),
        Avg_Song_Duration DECIMAL(10,2));
    OPEN cur;
    FETCH cur INTO genreID, genreName, avgSongDuration;
    WHILE NOT done DO
        INSERT INTO AvgDurationsTable VALUES (genreID, genreName, avgSongDuration);
        FETCH cur INTO genreID, genreName, avgSongDuration;
    END WHILE;
    CLOSE cur;
    SELECT * FROM AvgDurationsTable;
END $$
DELIMITER ;

```

8. This procedure allows users to add songs to their Favorites.

```

DELIMITER $$
CREATE PROCEDURE AddToFavorites(IN p_user_id INT, IN p_song_id INT)
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM UserFavorites
        WHERE User_ID = p_user_id AND Song_ID = p_song_id
    ) THEN
        INSERT INTO UserFavorites (User_ID, Song_ID)
        VALUES (p_user_id, p_song_id);
    END IF;
END $$
DELIMITER ;

```

9. This procedure allows users to add songs to their playlist.

```

DELIMITER $$
CREATE PROCEDURE AddToPlaylist(IN p_playlist_id INT, IN p_song_id INT)
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM PlaylistSongs
        WHERE Playlist_ID = p_playlist_id AND Song_ID = p_song_id
    ) THEN
        INSERT INTO PlaylistSongs (Playlist_ID, Song_ID)
        VALUES (p_playlist_id, p_song_id);
    END IF;
END $$
DELIMITER ;

```

10. This procedure allows users to delete songs from their playlist.

```

DELIMITER $$
CREATE PROCEDURE DeleteFromPlaylist(
    IN p_playlist_id INT,
    IN p_song_id INT
)
BEGIN
    DELETE FROM playlist
    WHERE Playlist_ID = p_playlist_id AND Song_ID = p_song_id;
END $$
DELIMITER ;

```

11. This procedure allows users to delete songs from their favourites.

```

DELIMITER $$
CREATE PROCEDURE DeleteFromFavorites(
    IN p_user_id INT,
    IN p_song_id INT
)
BEGIN
    DELETE FROM UserFavorites
    WHERE User_ID = p_user_id AND Song_ID = p_song_id;
END $$
DELIMITER ;

```

## FUNCTIONS

In our project, there are 3 functions.

1. This function called GetTotalAlbumsByArtist accepts an Artist\_ID and returns the total number of albums released by that artist.

```

DELIMITER $$
CREATE FUNCTION GetTotalAlbumsByArtist(
    artistID INT)
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE totalAlbums INT;
    SELECT COUNT(*) INTO totalAlbums
    FROM Album
    WHERE Artist_ID = artistID;
    RETURN totalAlbums;
END $$
DELIMITER ;

```

2. This function named GetGenreForSong takes a Song\_ID and returns the genre of the corresponding song.

```

DELIMITER $$
CREATE FUNCTION GetGenreForSong(
    songID INT)
RETURNS VARCHAR(255)
DETERMINISTIC
BEGIN
    DECLARE genreName VARCHAR(255);
    SELECT g.Genre_Name INTO genreName
    FROM Song s
    JOIN Genre g ON s.Genre_ID = g.Genre_ID
    WHERE s.Song_ID = songID;
    RETURN genreName;
END $$
DELIMITER ;

```

3. This function called GetMostRecentAlbumReleaseDate takes an Artist\_ID and returns the most recent release date of any album by that artist.

```

DELIMITER $$
CREATE FUNCTION GetMostRecentAlbumReleaseDate(
    artistID INT)
RETURNS DATE
DETERMINISTIC
BEGIN
    DECLARE mostRecentDate DATE;
    SELECT MAX(a.Release_Date) INTO mostRecentDate
    FROM Album a
    WHERE a.Artist_ID = artistID;
    RETURN mostRecentDate;
END $$
DELIMITER ;

```

## TRIGGERS

We have one trigger in our project:

It prevents the deletion of a song if it has received awards.

```

CREATE TRIGGER PreventAlbumDeletion
BEFORE DELETE ON Song
FOR EACH ROW
BEGIN
    DECLARE awardCount INT;
    SELECT COUNT(*) INTO awardCount
    FROM Awards
    WHERE Song_ID = OLD.Song_ID;
    IF awardCount > 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot delete album with associated awards';
    END IF;
END $$
DELIMITER ;

```



## VIEWS

We have 3 views in our project.

1. This view (TopRatedSongs) joins the Song, User\_Song\_Rating, and Artist tables. It calculates the average user rating for each song and includes the song ID, title, artist name, and average rating in the view.

```
CREATE VIEW TopRatedSongs AS
SELECT
    s.Song_ID,
    s.Song_Title,
    a.Full_Name AS Artist_Name,
    AVG(usr.Rating) AS Average_Rating
FROM
    Song s
JOIN
    User_Song_Rating usr ON s.Song_ID = usr.Song_ID
JOIN
    Artist a ON s.Artist_ID = a.Artist_ID
GROUP BY
    s.Song_ID, s.Song_Title, a.Full_Name;
```

2. This view named LabelAlbumStatistics that includes information about labels, such as label ID, label name, label country, and the total number of albums released by each label, excluding labels with no albums.

```
CREATE VIEW LabelAlbumStatistics AS
SELECT
    l.Label_ID,
    l.Label_Name,
    l.Country AS Label_Country,
    COUNT(DISTINCT a.Album_ID) AS Total_Albums
FROM
    Label l
LEFT JOIN
    Album a ON l.Label_ID = a.Label_ID
WHERE
    a.Album_ID IS NOT NULL
GROUP BY
    l.Label_ID, l.Label_Name, l.Country;
```

3. This view includes statistics for each genre, including the number of songs, average song duration, and the most common musical key.

```

CREATE VIEW GenreStatistics AS
SELECT g.Genre_ID, g.Genre_Name, COUNT(s.Song_ID) AS Number_of_Songs,
      AVG(s.Time_Period) AS Average_Song_Duration,
      SUBSTRING_INDEX(GROUP_CONCAT(s.Musical_Key ORDER BY keyCounts.KeyCount DESC), ',', 1)
      AS Most_Common_Musical_Key
FROM Genre g
JOIN Song s ON g.Genre_ID = s.Genre_ID
JOIN (SELECT Song_ID, Musical_Key, COUNT(Musical_Key) AS KeyCount
      FROM Song
      GROUP BY Song_ID, Musical_Key) AS keyCounts ON s.Song_ID = keyCounts.Song_ID
GROUP BY g.Genre_ID, g.Genre_Name;

```

The number\_of\_songs column denotes the number of songs belonging to that genre while the average\_song\_duration column denotes the average song duration in seconds. The last column denotes the most\_common\_musical\_key used in the songs of that particular genre.

Genre_ID	Genre_Name	Number_of_songs	Average_song_duration	Most_common_musical_key
111111	Art Punk	1	167303	6
111116	Crust Punk (thanks Haug)	1	391888	1
111119	Folk Punk	1	172797	7
111120	Goth / Gothic Rock	2	278641.5	5
111121	Grunge	1	212953	2
111122	Hardcore Punk	1	165582	1
111127	New Wave	1	189901	4
111129	Punk	1	244684	8
111134	Blues Rock	1	191959	6
111136	British Blues	1	319369	10
111142	Contemporary R&B	1	178426	9
111146	Detroit Blues	1	260111	8

## DATA SOURCES

The pertinent information extracted for analysis includes Artist name, Song Title, Time period, Musical key, Album title, Danceability, Energy, Loudness, Mode, Speechiness, Acousticness, Instrumentalness, Liveness, Valence, Tempo, and Time\_signature. This data was acquired from the Spotify Datasets available on Kaggle, specifically the "artists.csv" and "tracks.csv" files, constituting a comprehensive repository of information about artists, songs, and albums.

It is noteworthy that certain data refinement processes have been undertaken to manually populate additional tables such as Genre, Label, Playlist, User, User Song rating, Users' favorite songs and albums, Billboard Chart data, and Awards. Furthermore, meticulous manual efforts

have been invested in completing the remaining attributes within the Artist, Album, and Song tables.

Links:

<https://www.kaggle.com/datasets/lehaknarnauli/spotify-datasets?select=artists.csv>

<https://www.kaggle.com/datasets/lehaknarnauli/spotify-datasets?select=tracks.csv>

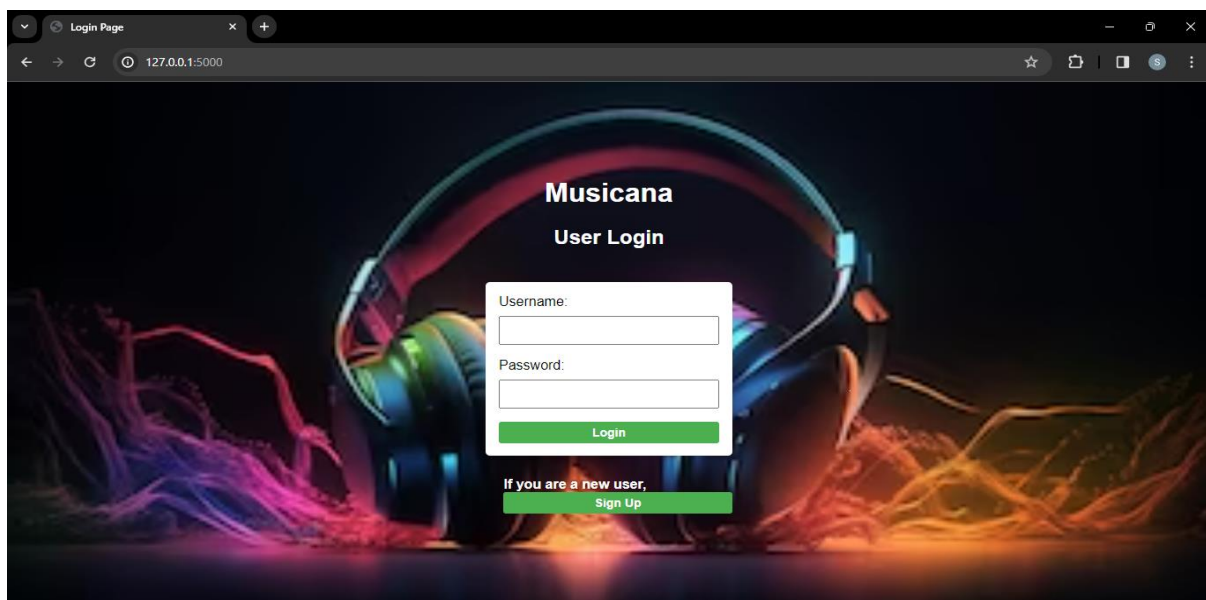
In the data refinement process for the music dataset, our initial step involved gathering raw data from diverse sources. Subsequently, during the Data Processing phase, we implemented a series of critical steps to enhance data quality and structure. Duplicate entries were systematically removed, and careful scrutiny was applied to rectify inaccuracies, especially within the artists' information. Given instances where songs featured multiple artists, we executed a meticulous breakdown of these entries into individual tuples. This approach not only maintained the atomic value property but also ensured uniformity across other key attributes.

In the normalization of the genres table, each tuple underwent a transformative process, being split into multiple rows. This normalization effort aimed to achieve a standardized form, particularly relevant as each song could be associated with multiple genres.

## **APPLICATION DESCRIPTION**

### **Log-in:**

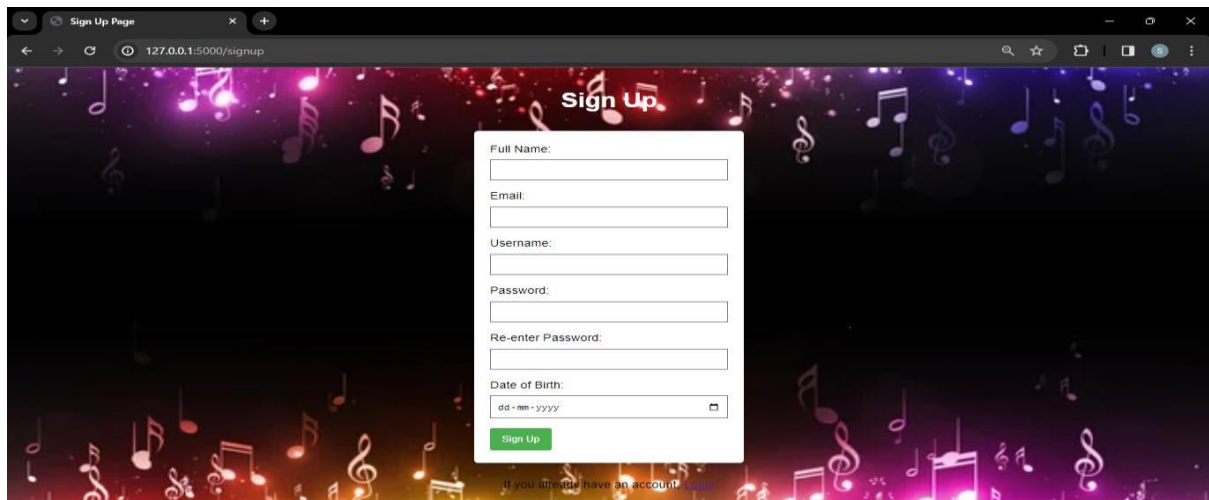
The first thing the user must do to access the website is to log in with their username and password.



### **Sign-up:**

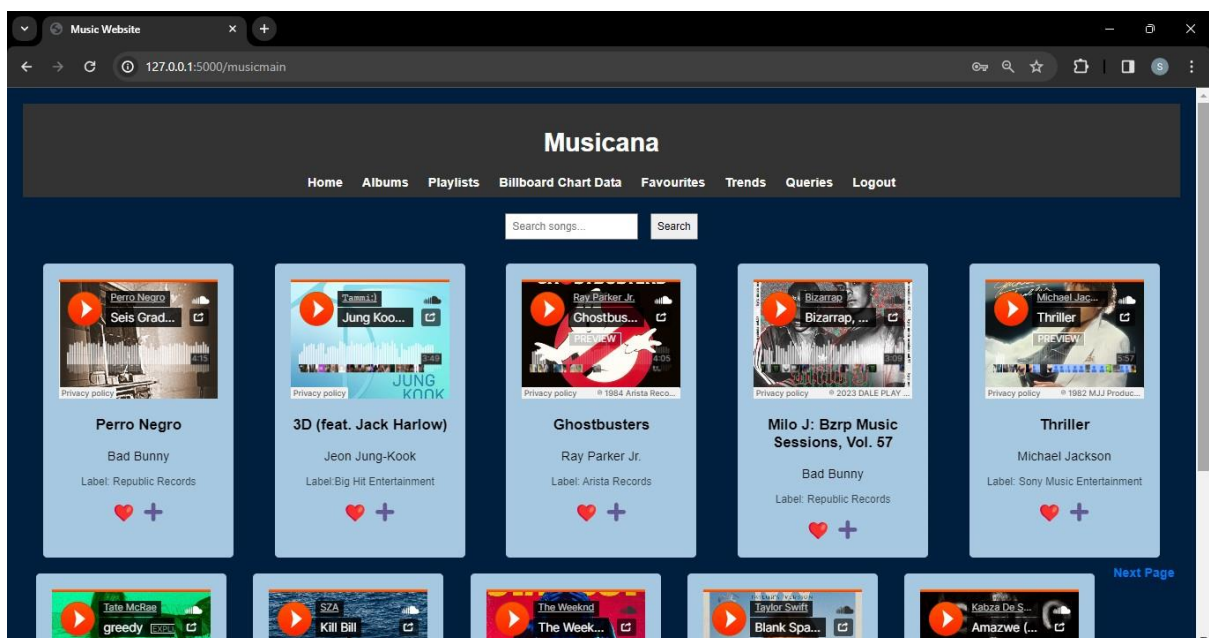
If the user is not registered and is new, then the user can click on the “Sign Up” button to get registered. We created a sign-up page for this. The users have to enter their details and sign

up. After signing up, the users must go back to the login page and enter their new credentials. After login is successful, they will be redirected to the home page.

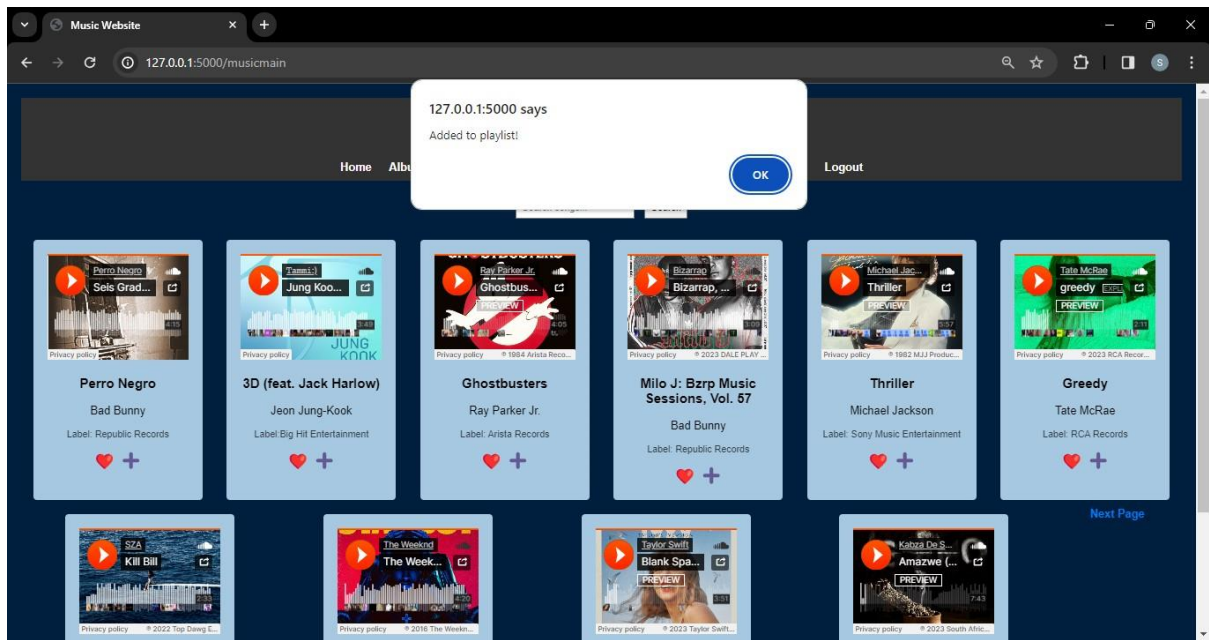
A screenshot of a web browser showing a 'Sign Up' page. The page has a dark background with floating musical notes. A white sign-up form is centered, containing fields for 'Full Name', 'Email', 'Username', 'Password', 'Re-enter Password', and 'Date of Birth' (with a dropdown for format). A green 'Sign Up' button is at the bottom of the form. Below the form, a link says 'If you already have an account, login'. The browser's address bar shows '127.0.0.1:5000/signup'.

## Homepage:

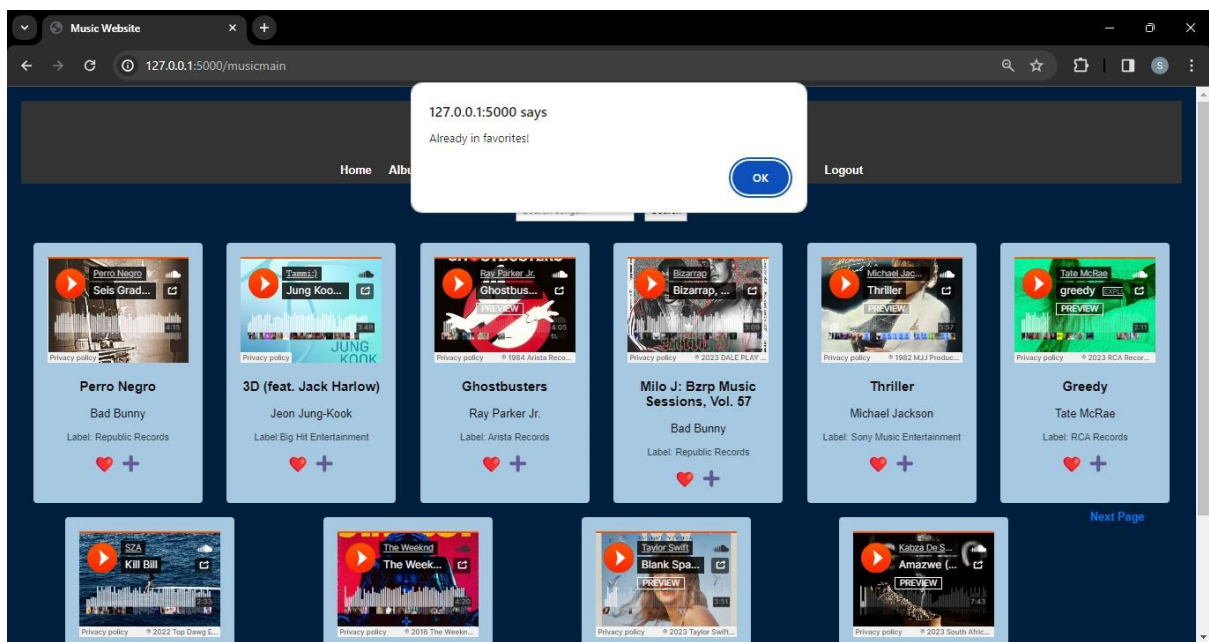
The homepage contains list of all songs and the users can click on the play button to play any song. Each song has the details of the song name, artist and the label name. The header consists of links to all other pages, a search bar and the log-out button. This page also contains a link to the next page where there are more songs. Each song has a heart icon and a plus icon. When the users click on the heart icon and the plus icon, that particular song gets added to the favourites tab and the playlist tab respectively.



Whenever a user tries to add a song to favourites or playlist by clicking on the heart icon and the plus icon, a pop-up will appear stating that the song is added.



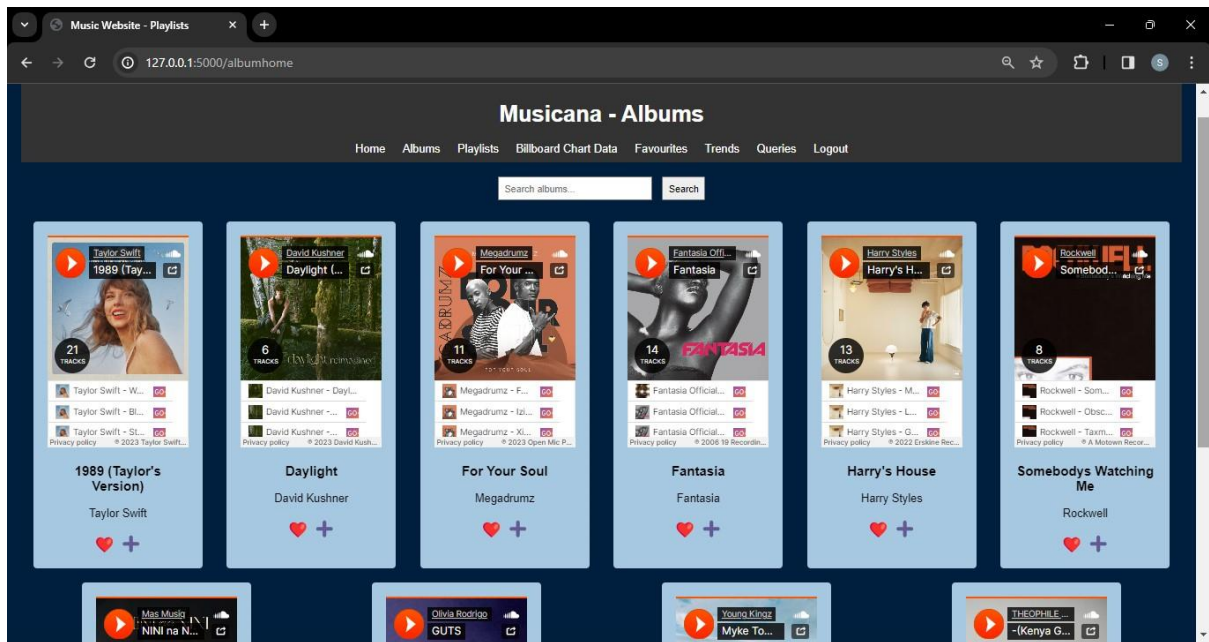
Whenever a user tries to add a song to favourites or playlist that is already existing in the tabs, a pop up will appear stating that that particular song already exists.



### Album page:

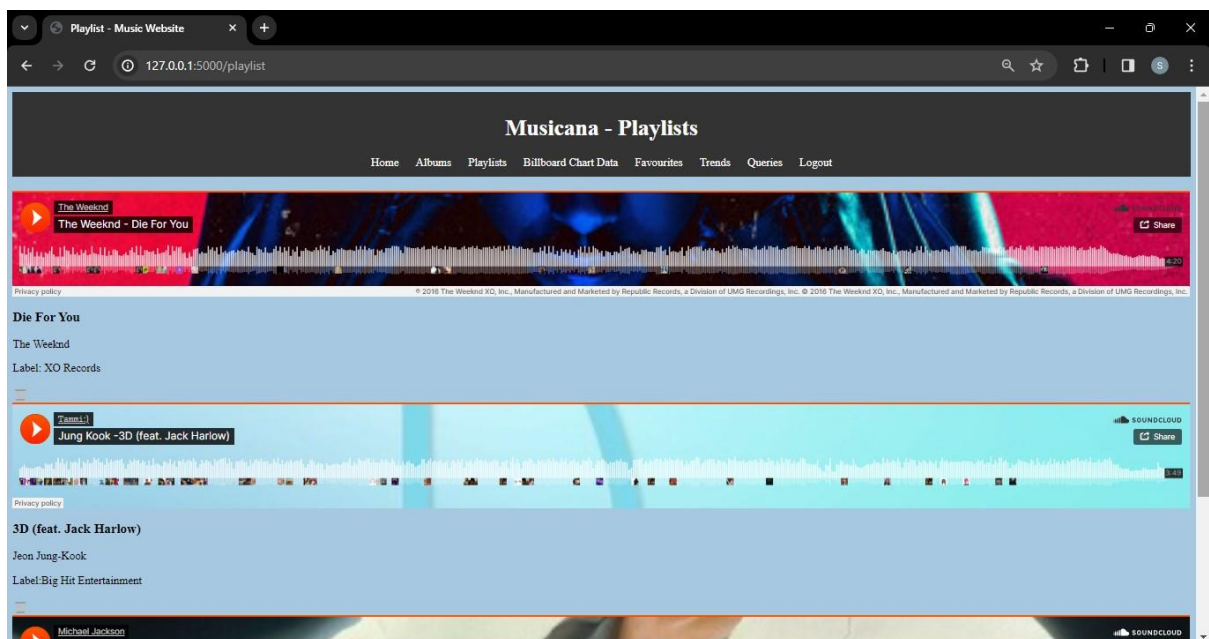
The album page contains all the albums of different artists. The user can click on the play button and play any song which they like. As in the home page, this page also contains the heart and the plus icons which perform the same actions. It has the same header as the home page.



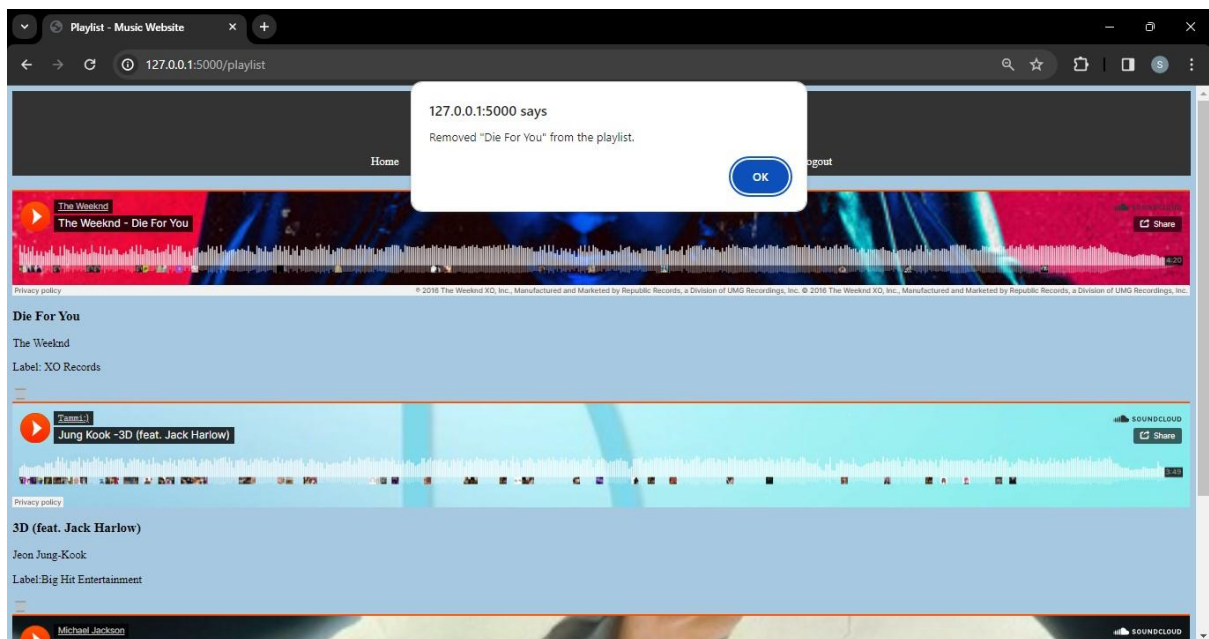


### Playlists:

This page contains the songs that the user added to his playlist. The user can click on the play button to play the song. There is a “bin” icon which when clicked, removes that particular song from the playlist.



After the user clicks on the bin icon, a pop-up will appear that the song has been removed.



## Billboard Chart Data:

This page contains the rankings of the songs. Each song card has the song name, artist name, chart position and the weekly movement of the song.

# Musicana - Billboard Chart Data

[Home](#)
[Albums](#)
[Playlists](#)
[Billboard Chart Data](#)
[Favourites](#)
[Trends](#)

[Queries](#)
[Logout](#)

1	<p><b>Is It Over Now? (Taylor's Version) (From The Vault)</b> Taylor Swift Chart Position: #1   Weekly movement: 49</p>	1	<p><b>iPlan</b> Dlala Thukzin Chart Position: #1   Weekly movement: -23</p>
2	<p><b>Greedy</b> Tate McRae Chart Position: #2   Weekly movement: 0</p>	3	<p><b>Exchange</b> Bryson Tiller Chart Position: #3   Weekly movement: -3</p>
3	<p><b>Dalie (feat. Baby S.O.N)</b> Kamo Mphela Chart Position: #3   Weekly movement: 2</p>	4	<p><b>Hit Me Up (feat. Nomovodka)</b> Taylor Swift Chart Position: #4   Weekly movement: 4</p>

### Favourites:

This page is similar to the playlists page. It consists of the songs that the user added to his favourites by clicking on the heart icon. The “bin” icon is used to remove the song from the favourites. The same pop-up will be shown as in the playlists that the song is removed.



### Trends:

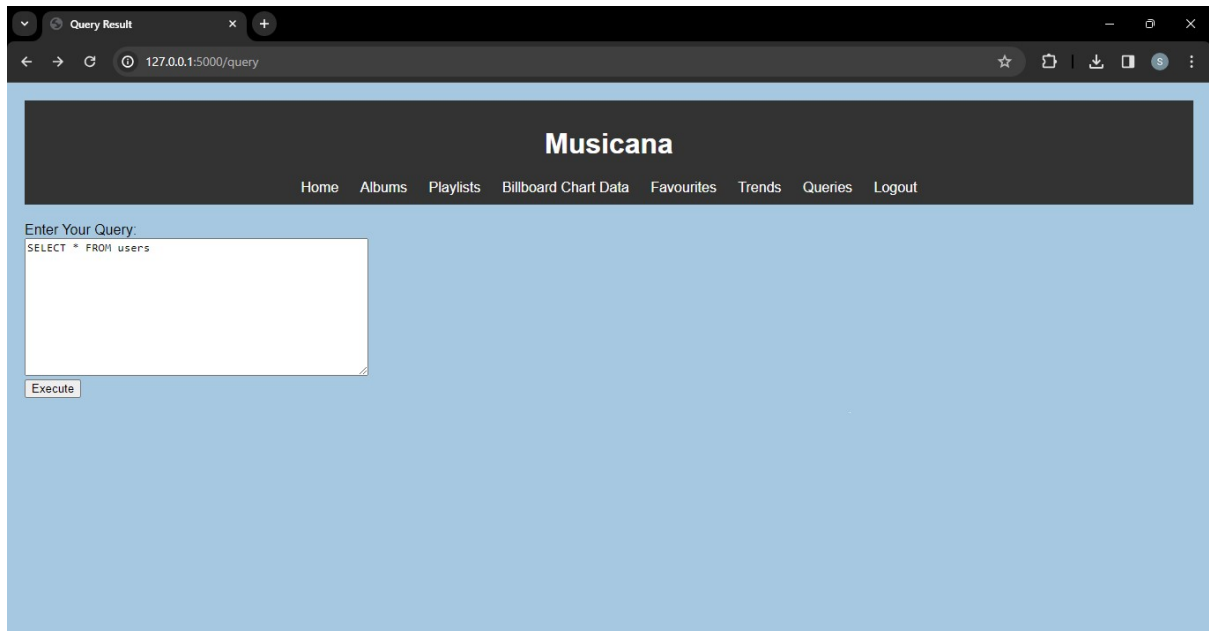
This page shows the analysis of our data. There is a drop down button from which the user can select which trend to be viewed. More about the trends is discussed under the Analysis heading.



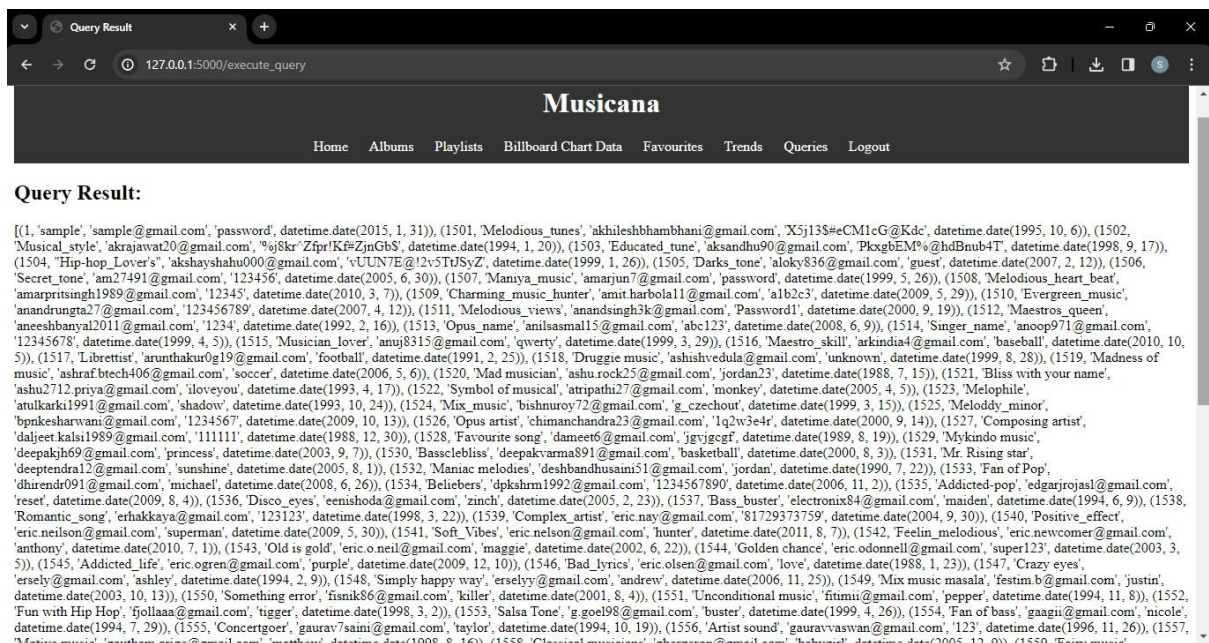


## Queries:

In this page, users are free to run SQL queries and experiment with the database. The query is to be written in the text box and the user must click on the “Execute” button to submit the query.



The result is shown in the query result window.



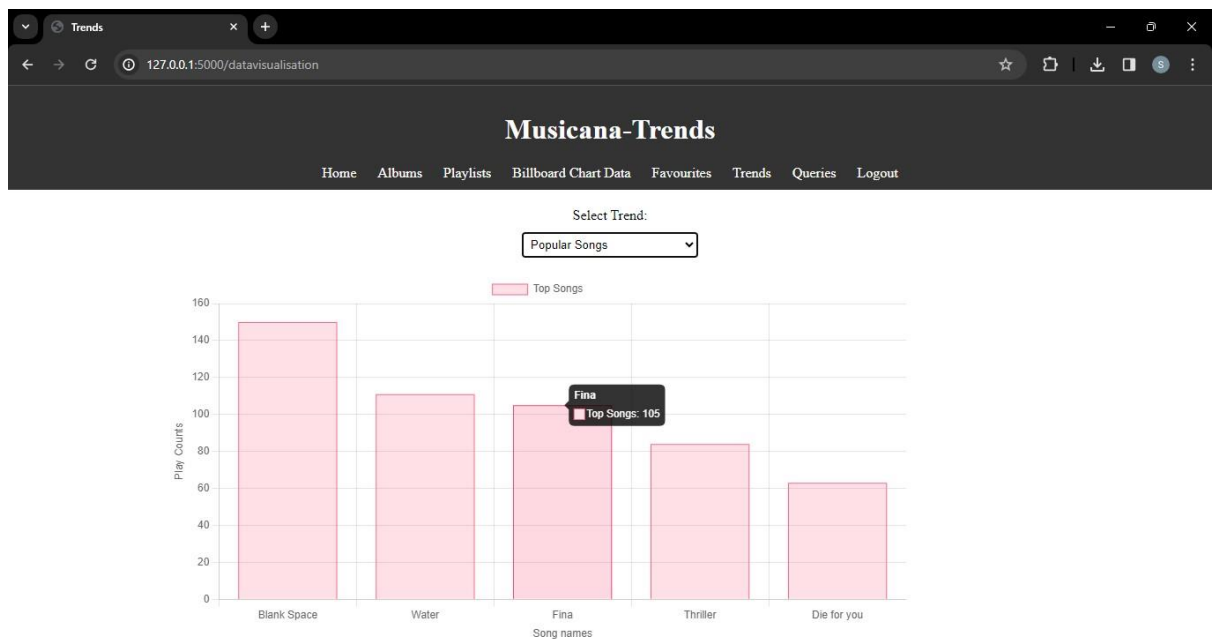
If the user wants to run more queries, he can click on the queries link in the header of the page which will redirect to the queries page.

In order to exit our application, the user can click on the logout button which will redirect to the login page.

# ANALYSIS

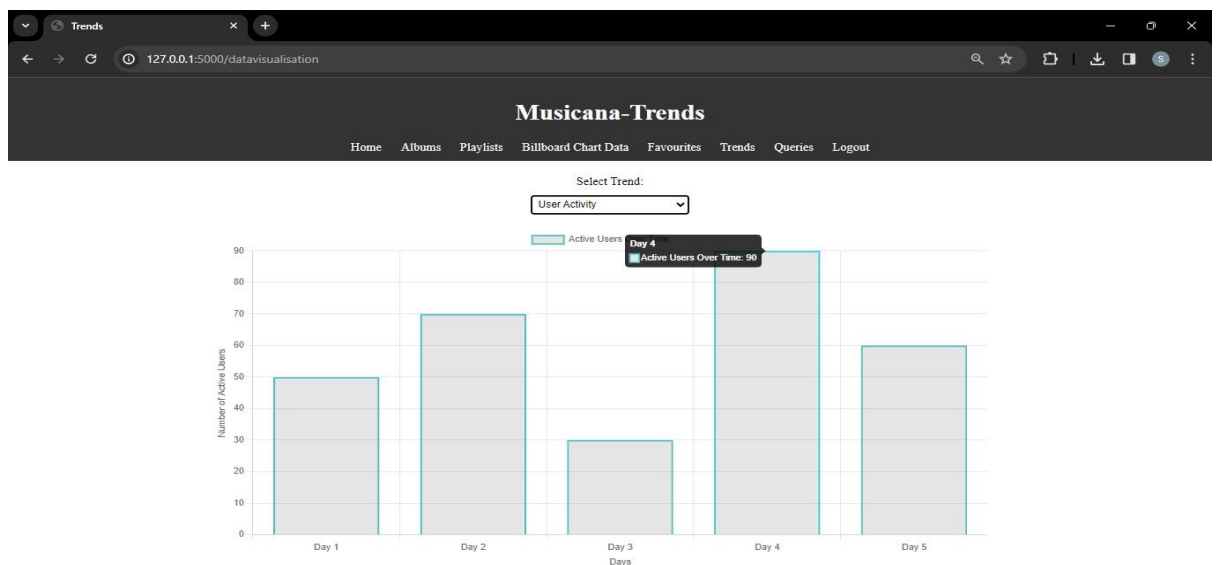
## 1. Popular songs

This bar chart shows the top 5 songs based on play counts. The x-axis contains the song names while the y-axis contains the number of times that song has been played.



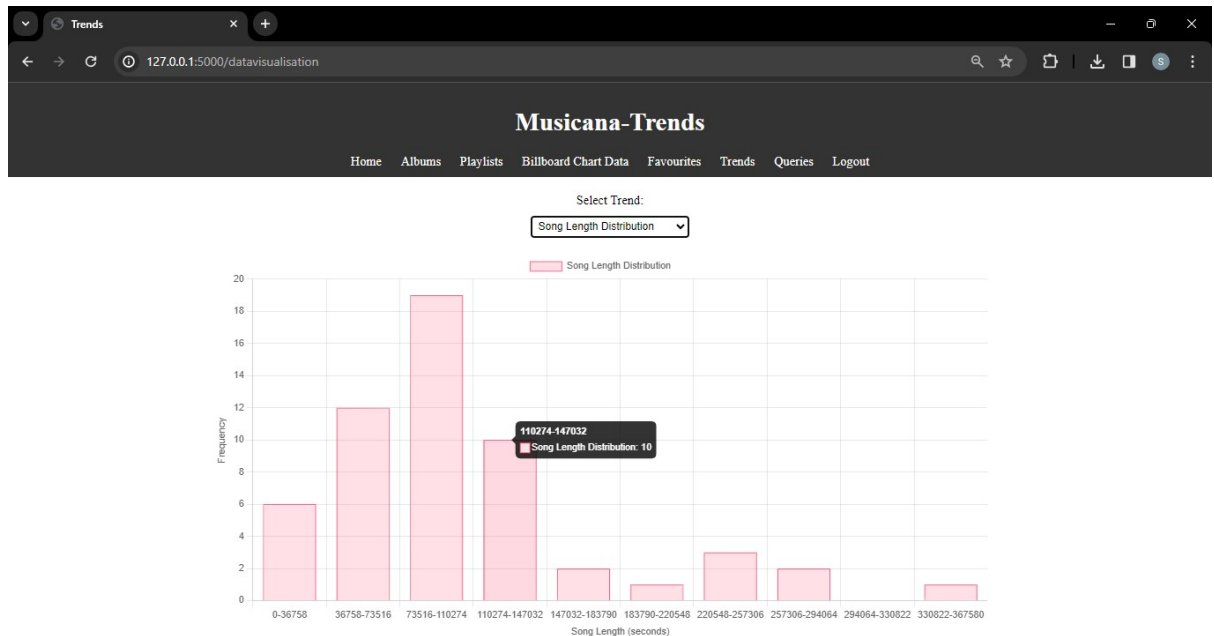
## 2. User Activity

This bar chart shows the user activity in the website on a daily basis. The x-axis contains the day and the y-axis contains the number of active users.



### 3. Song Length Distribution

This bar chart shows the song length distribution of the songs. The x-axis consists the range of song length values and the y-axis consists of the frequency of the songs. Eg. 10 songs have their length between a particular range. The song length is measured in seconds.



## CONCLUSION

In the course of our project journey, we not only curated our dataset but also harnessed the power of SQL queries to meticulously shape a comprehensive music database. This database became the backbone for the development of Musicana, an innovative music website designed to cater to diverse user preferences. Musicana provides users with a dynamic and immersive experience, allowing them to seamlessly listen to a vast library of songs while empowering them with features to personalize their music journey. Users on Musicana can actively engage with their music by adding songs to their favorites and creating personalized playlists, ensuring a tailored and enjoyable listening experience. In addition to these features, we introduced a tool within Musicana that enables users to run SQL queries, providing a unique and interactive dimension to the platform.

While our vision for Musicana initially encompassed the integration of a sophisticated recommendation system, regrettably, time constraints limited its inclusion in the current project iteration. However, this serves as a testament to the ambitious scope of our project, and the foundation laid by Musicana is poised for future enhancements. The prospect of implementing a recommendation system to deliver personalized song suggestions based on user preferences remains an exciting avenue for exploration and development as we continue to refine and expand the Musicana platform.

For future DS5110 students, we advise early planning, prioritizing tasks, adopting a modular approach, fostering collaboration, and considering flexible architecture for future developments.