universidade de aveiro

BeatBase

Plataforma de streaming digital de música

Base de Dados
2023/2024

Diogo Fernandes 114137
Raquel Vinagre 113736

P4G6

Índice

Introdução

Este projeto consiste numa base de dados para uma plataforma de streaming digital de música, de modo a gerir utilizadores, playlists, músicas, artistas e álbuns. A nossa escolha foi motivada pelo nosso interesse mútuo na área da música e na versatilidade que disponibiliza para um projeto deste tipo.

Como em qualquer plataforma de streaming de música, os utilizadores podem ouvir músicas adicionadas pelos artistas. É possível, ainda, criar/editar playlists personalizadas, criar/editar álbuns, verificar as leaderboards globais e ainda algumas estatísticas.

A interface foi desenvolvida para o uso administrativo (todas as permissões), de modo a demonstrar todas as features implementadas, quer sejam de artista ou utilizador.

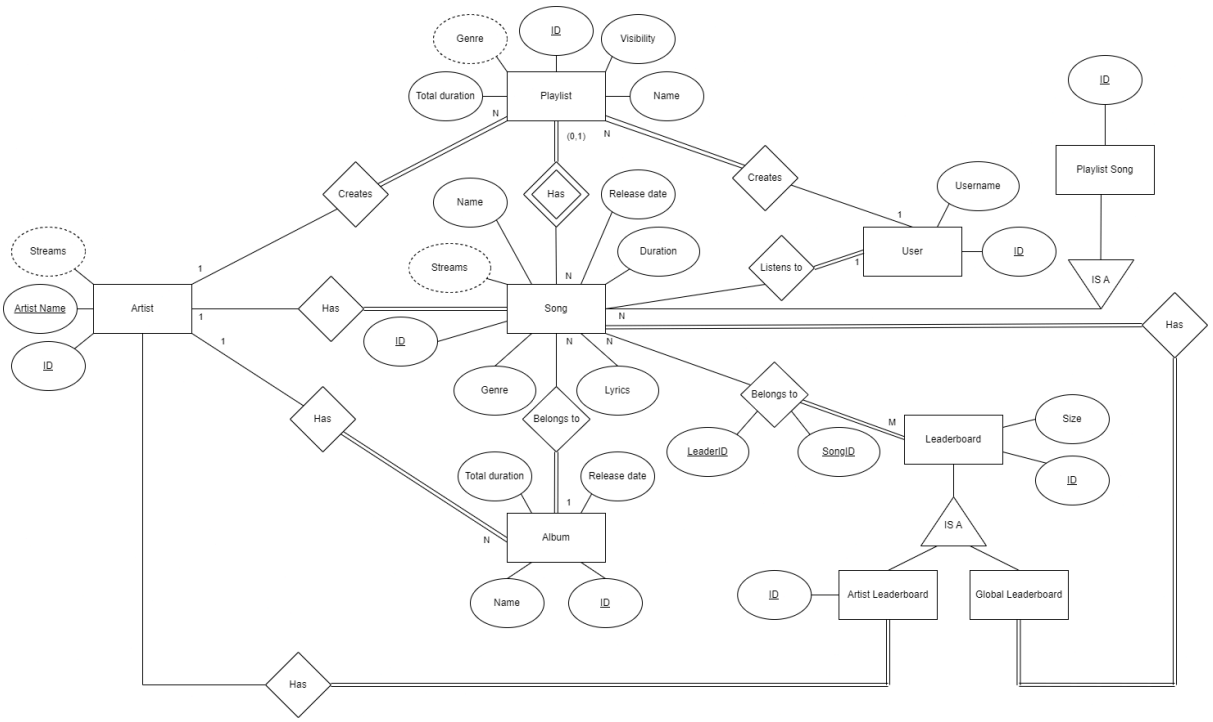
Requisitos Funcionais

Entidades	Música - pode ou não estar num álbum
	Álbum - conjunto de músicas lançadas pelo mesmo artista na mesma data
	Playlist - lista de músicas personalizada
	Música de Playlist - música que pertence a uma playlist
	Artista - pessoa que pode adicionar músicas e álbuns à plataforma
	Utilizador - pessoa que pode ouvir as músicas e criar playlists
	Leaderboard - lista de músicas mais populares

	Administrador
Funcionalidades	Adicionar/editar/remover músicas
	Adicionar/editar/remover álbuns
	Criar/editar/remover artistas
	Criar/editar/remover playlists
	Procurar/filtrar info sobre artista/álbum/música/playlists
	Ouvir e favoritar músicas
	Leaderboards
	Estatísticas

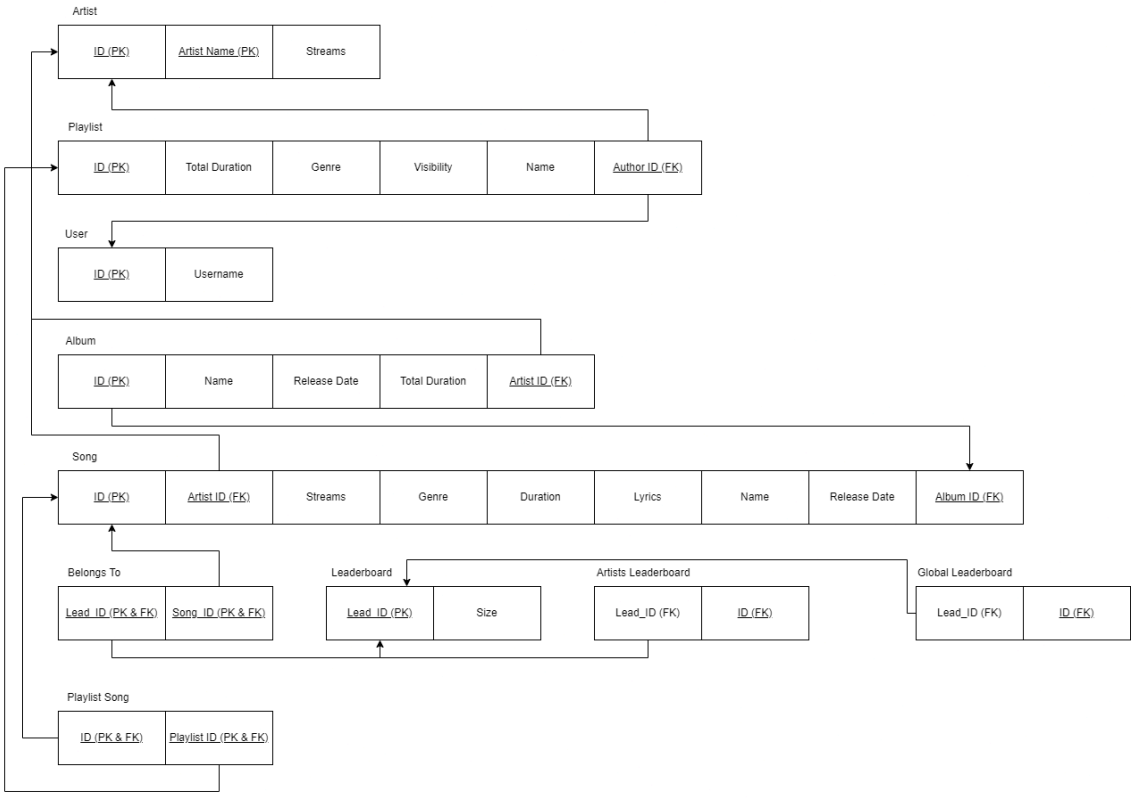
Diagrama Entidade-Relacionamento

Em comparação com o DER apresentado na proposta, este tem algumas alterações, nomeadamente, a nova entidade, Playlist Song. Define-se, ainda, que a global leaderboard é específica para músicas.



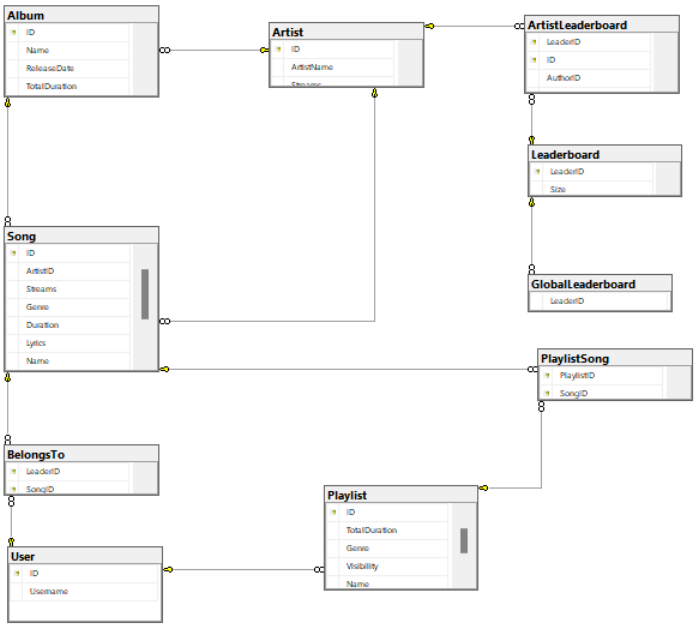
Esquema Relacional

O ER apresenta as alterações referidas acima.



Esquema Relacional (SGDB)

Este ER foi gerado automaticamente pela base de dados atualizada.



DDL

A Data Definition Language (DDL) permitiu definir a estrutura dos objetos a ser guardados na base de dados. O código que gera estas tabelas é o seguinte apresentado:

```
-- Drop referencing tables
DROP TABLE IF EXISTS PlaylistSong;
DROP TABLE IF EXISTS BelongsTo;
DROP TABLE IF EXISTS ArtistLeaderboard;
DROP TABLE IF EXISTS GlobalLeaderboard;

-- Drop referenced tables
DROP TABLE IF EXISTS Song;
DROP TABLE IF EXISTS Album;
DROP TABLE IF EXISTS Playlist;
DROP TABLE IF EXISTS Artist;
DROP TABLE IF EXISTS [User];
DROP TABLE IF EXISTS Leaderboard;
```

```
-- Create tables
CREATE TABLE Artist (
  ID INT NOT NULL IDENTITY(1,1),
  ArtistName VARCHAR(255) NOT NULL,
  Streams INT DEFAULT 0,
  PRIMARY KEY(ID)
);

CREATE TABLE [User] (
  ID INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Username VARCHAR(255)
);

CREATE TABLE Playlist (
  ID INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  TotalDuration INT DEFAULT 0,
  Genre VARCHAR(255),
  Visibility BIT,
  Name VARCHAR(255),
  AuthorID INT NOT NULL,
  FOREIGN KEY (AuthorID) REFERENCES [User](ID)
);

CREATE TABLE Album (
  ID INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Name VARCHAR(255),
  ReleaseDate DATE,
  TotalDuration INT DEFAULT 0,
  ArtistID INT NOT NULL,
  CoverImage VARBINARY(MAX),
  FOREIGN KEY (ArtistID) REFERENCES Artist(ID) ON DELETE CASCADE ON
UPDATE CASCADE
);

CREATE TABLE Song (
  ID INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  ArtistID INT NOT NULL,
  Streams INT,
  Genre VARCHAR(255),
  Duration INT,
  Lyrics TEXT,
  Name VARCHAR(255),
  ReleaseDate DATE,
```

```
  AlbumID INT,
  FOREIGN KEY (ArtistID) REFERENCES Artist(ID) ON DELETE CASCADE ON
UPDATE CASCADE,
  FOREIGN KEY (AlbumID) REFERENCES Album(ID) --ON DELETE NO ACTION ON
UPDATE NO ACTION
);

CREATE TABLE PlaylistSong (
    PlaylistID INT NOT NULL,
    SongID INT NOT NULL,
    PRIMARY KEY (PlaylistID, SongID),
    FOREIGN KEY (PlaylistID) REFERENCES Playlist(ID) ON DELETE CASCADE
ON UPDATE CASCADE,
    FOREIGN KEY (SongID) REFERENCES Song(ID) ON DELETE CASCADE ON
UPDATE CASCADE
);

CREATE TABLE BelongsTo (
  LeaderID INT NOT NULL,
  SongID INT NOT NULL,
  PRIMARY KEY (LeaderID, SongID),
  FOREIGN KEY (LeaderID) REFERENCES [User](ID),
  FOREIGN KEY (SongID) REFERENCES Song(ID) ON DELETE CASCADE ON UPDATE
CASCADE
);

CREATE TABLE Leaderboard (
  LeaderID INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Size INT
);

CREATE TABLE ArtistLeaderboard (
  LeaderID INT NOT NULL,
  ID INT NOT NULL,
  AuthorID INT NOT NULL,
  ArtistName VARCHAR(255) NOT NULL,
  PRIMARY KEY (LeaderID, ID),
  FOREIGN KEY (LeaderID) REFERENCES Leaderboard(LeaderID),
  FOREIGN KEY (AuthorID) REFERENCES Artist(ID) ON DELETE CASCADE ON
UPDATE CASCADE
);

CREATE TABLE GlobalLeaderboard ( --this is the songs leaderboard
```

```
  LeaderID INT NOT NULL,
  FOREIGN KEY (LeaderID) REFERENCES Leaderboard(LeaderID)
);;
```

DML

A Data Manipulation Language (DML) permitiu inserir dados nas tabelas da base de dados. O código de inserção de dados é o seguinte apresentado (não contém o código total):

```
-- Clean the values from the tables before inserting new ones
DELETE FROM ArtistLeaderboard;
DELETE FROM BelongsTo;
DELETE FROM Song;
DELETE FROM Album;
DELETE FROM Playlist;
DELETE FROM [User];
DELETE FROM Artist;
DELETE FROM Leaderboard;
DELETE FROM GlobalLeaderboard;


INSERT INTO Artist (ArtistName, Streams) VALUES
('The Weeknd', 2839132),
('Drake', 483922),


INSERT INTO [User] (Username) VALUES
('user1'),
('user2'),

('user10');

INSERT INTO Playlist (Genre, Visibility, Name, AuthorID) VALUES
('Pop', 1, 'Liked Songs', 1),
('Hip-Hop', 0, 'Workout Playlist', 2),
```

```sql
INSERT INTO Album (Name, ReleaseDate, ArtistID) VALUES
('After Hours', '2020-03-20',  1),
('Scorpion', '2018-06-29',  2),

INSERT INTO Song (ArtistID, Streams, Genre, Duration, Lyrics, Name,
ReleaseDate, AlbumID) VALUES
(1, 500000, 'R&B', 240, '...', 'Blinding Lights', '2020-01-21', 1),
(1, 300000, 'R&B', 200, '...', 'Heartless', '2019-11-27', 1),


INSERT INTO PlaylistSong (PlaylistID, SongID) VALUES
(1, 1),
(1, 3),


INSERT INTO BelongsTo (LeaderID, SongID) VALUES
(1, 1),
(1, 3),


INSERT INTO Leaderboard (Size) VALUES
(10),
(5),


INSERT INTO ArtistLeaderboard (LeaderID, ID, AuthorID, ArtistName)
VALUES
(1, 4, 4, 'Kendrick Lamar'),
(1, 5, 5, 'Billie Eilish'),


INSERT INTO GlobalLeaderboard (LeaderID) VALUES
(1),
(2),
```

Stored Procedures

As stored procedures permitiram armazenar e executar blocos de código SQL, de maneira reutilizável. De maneira geral, fizemos uso das mesmas para selecionar entidades como artistas e músicas ou atributos como género, e ainda para apagar álbuns com músicas. O código das Stored Procedures é o seguinte apresentado (não contém o código total):

```
DROP PROCEDURE GetAllArtists;
DROP PROCEDURE GetSongsByAlbumID;
DROP PROCEDURE DeleteAlbumWithSongs;

go
CREATE PROCEDURE GetAllArtists
AS
BEGIN
    SELECT ID, ArtistName, Streams
    FROM Artist
END
go

CREATE PROCEDURE GetSongsByAlbumID
    @AlbumID INT
AS
BEGIN
    SELECT
        ID,
        ArtistID,
        Streams,
        Genre,
        Duration,
        Lyrics,
        Name,
        ReleaseDate,
        AlbumID
    FROM
        Song
    WHERE
        AlbumID = @AlbumID;
END
go
```

```
CREATE PROCEDURE DeleteAlbumWithSongs
    @AlbumID INT
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRANSACTION;

        -- Update songs associated with the album to set AlbumID to
NULL
        UPDATE Song
        SET AlbumID = NULL
        WHERE AlbumID = @AlbumID;

        -- Now delete the album
        DELETE FROM Album
        WHERE ID = @AlbumID;

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;

        -- Raise error or handle as per your application's requirement
        THROW;
    END CATCH;
END;

go
```

Triggers

Os triggers são úteis pois são objetos que são acionados automaticamente, em resposta a determinados eventos. Neste caso, usamo-los para atualizar streams em álbuns e playlists, também foram usados para calcular a duração total dos mesmos. O código dos Triggers é o seguinte apresentado (não contém o código total):

```
-- Drop triggers
IF EXISTS (SELECT name FROM sys.triggers WHERE name =
'UpdateArtistStreamsOnInsert')
BEGIN
  DROP TRIGGER UpdateArtistStreamsOnInsert;
END;
IF EXISTS (SELECT name FROM sys.triggers WHERE name =
'UpdateArtistStreamsOnUpdate')
BEGIN
  DROP TRIGGER UpdateArtistStreamsOnUpdate;
END;
IF EXISTS (SELECT name FROM sys.triggers WHERE name =
'UpdateArtistStreamsOnDelete')
BEGIN
  DROP TRIGGER UpdateArtistStreamsOnDelete;
END;


-- Create triggers to update Artist streams
CREATE TRIGGER UpdateArtistStreamsOnInsert
ON Song
AFTER INSERT
AS
BEGIN
  UPDATE Artist
  SET Streams = (
    SELECT SUM(Streams)
    FROM Song
    WHERE ArtistID = inserted.ArtistID
  )
  FROM inserted
  WHERE Artist.ID = inserted.ArtistID;
END;
GO

CREATE TRIGGER UpdateArtistStreamsOnUpdate
ON Song
AFTER UPDATE
AS
BEGIN
  UPDATE Artist
  SET Streams = (
```

```
    SELECT SUM(Streams)
    FROM Song
    WHERE ArtistID = inserted.ArtistID
  )
  FROM inserted
  WHERE Artist.ID = inserted.ArtistID;
END;
GO

CREATE TRIGGER UpdateArtistStreamsOnDelete
ON Song
AFTER DELETE
AS
BEGIN
  UPDATE Artist
  SET Streams = (
    SELECT SUM(Streams)
    FROM Song
    WHERE ArtistID = deleted.ArtistID
  )
  FROM deleted
  WHERE Artist.ID = deleted.ArtistID;
END;
GO
```

Cursores

Os cursores permitem selecionar informação em várias linhas de uma vez. No entanto, apenas é vantajoso utilizá-los para operações complexas. No nosso caso, utilizar cursores iria diminuir a performance, pelo que decidimos não implementar os mesmos no projeto.

Views

As views permitem fazer consultas personalizadas abstraindo a complexidade das tabelas adjacentes. O código das views é o seguinte apresentado:

```
DROP VIEW IF EXISTS SongsWithoutAlbums;

go
CREATE VIEW SongsWithoutAlbums AS
SELECT
  ID,
  ArtistID,
  Streams,
  Genre,
  Duration,
  Lyrics,
  Name,
  ReleaseDate
FROM
Song
WHERE
  AlbumID IS NULL;
go
```

Indexes

Os índices são uma ferramenta útil no que toca ao aumento do desempenho da procura de dados. Utilizamos índices para procurar músicas e álbuns.
O código dos Índices é o seguinte apresentado:

```
-- Drop and create index on Song table for ArtistID
IF EXISTS (SELECT name FROM sys.indexes WHERE name =
'idx_Song_ArtistID')
BEGIN
    DROP INDEX idx_Song_ArtistID ON Song;
END
CREATE INDEX idx_Song_ArtistID ON Song (ArtistID);

-- Drop and create index on Album table for ArtistID
IF EXISTS (SELECT name FROM sys.indexes WHERE name =
'idx_Album_ArtistID')
BEGIN
    DROP INDEX idx_Album_ArtistID ON Album;
```

```
END
CREATE INDEX idx_Album_ArtistID ON Album (ArtistID);
```

UDFs

As User-Defined Functions (UDF) criam funções personalizadas, pelo que simplificam a consulta complexa de informação. Estas foram utilizadas principalmente para criar filtros, calcular estatísticas e pesquisar por nome. O código das UDFs é o seguinte apresentado (não contém o código total):

```
DROP FUNCTION IF EXISTS FilterSongsByArtistID;
GO
CREATE FUNCTION dbo.FilterSongsByArtistID (@ArtistID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT *
    FROM Song
    WHERE ArtistID = @ArtistID
);
GO

DROP FUNCTION IF EXISTS GetTopSongs;
GO
CREATE FUNCTION dbo.GetTopSongs (@TopN INT)
RETURNS TABLE
AS
RETURN (
  SELECT TOP(@TopN) ID, Name, Streams
  FROM Song
  ORDER BY Streams DESC
);
GO

DROP FUNCTION IF EXISTS GetSongsInPlaylist;
go
CREATE FUNCTION dbo.GetSongsInPlaylist (@PlaylistID INT)
RETURNS TABLE
```

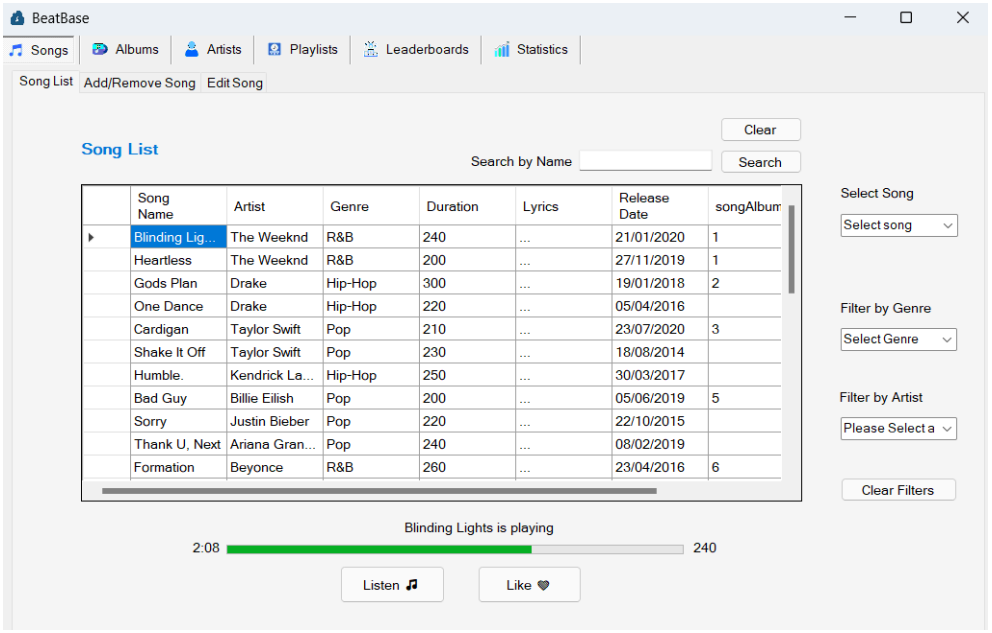
```
AS
RETURN
(
    SELECT ps.SongID
    FROM PlaylistSong ps
    WHERE ps.PlaylistID = @PlaylistID
);
go


drop FUNCTION IF EXISTS AverageSongDuration;
go
CREATE FUNCTION dbo.AverageSongDuration()
RETURNS FLOAT
AS
BEGIN
    DECLARE @AvgSongDuration FLOAT;
    SELECT @AvgSongDuration = AVG(Duration) FROM Song;
    RETURN @AvgSongDuration;
END;
GO

drop FUNCTION IF EXISTS MostPopularSongGenre;
go
CREATE FUNCTION dbo.MostPopularSongGenre()
RETURNS TABLE
AS
RETURN
(
    SELECT TOP 1 Genre, COUNT(*) AS GenreCount
    FROM Song
    GROUP BY Genre
    ORDER BY GenreCount DESC
);
GO

CREATE FUNCTION dbo.AverageNumSongsPerArtist()
RETURNS FLOAT
AS
BEGIN
    DECLARE @AvgNumSongsPerArtist FLOAT;
    SELECT @AvgNumSongsPerArtist = AVG(SongCount)
```

```
    FROM (SELECT COUNT(*) AS SongCount FROM Song GROUP BY ArtistID) AS
ArtistSongs;
    RETURN @AvgNumSongsPerArtist;
END;
GO

drop FUNCTION IF EXISTS TotalSongs;
go

CREATE FUNCTION dbo.TotalSongs ()
RETURNS INT
AS
BEGIN
    DECLARE @TotalSongsCount INT;
    SELECT @TotalSongsCount = COUNT(*)
    FROM Song;
    RETURN @TotalSongsCount;
END;
GO

drop function if exists SearchSongByName;
go
CREATE FUNCTION dbo.SearchSongByName(@songName VARCHAR(255))
RETURNS TABLE
AS
RETURN(
    SELECT
        s.ID ,
        s.Name ,
        s.ArtistID ,
        s.Genre ,
        s.Duration ,
        s.Lyrics ,
        s.ReleaseDate ,
        s.AlbumID,
        s.Streams
    FROM
        Song s
    JOIN
        Artist a ON s.ArtistID = a.ID
    WHERE
        s.Name LIKE '%' + @songName + '%'
);
```

Interface

Como referido anteriormente, a interface foi desenvolvida do ponto de vista de um administrador, que basicamente tem acesso a todas as funcionalidades dos artistas e utilizadores.

De maneira geral, a interface divide-se em 6 tabs principais: Songs, Albums, Artists, Playlists, LeaderBoards e Statistics. A maior parte destas divide-se noutras sub-tabs, que permitem ver listas de dados e alterá-los. Há, ainda, filtros e barras de pesquisa para ajudar na consulta de informação.



Conclusão

O desenvolvimento deste projeto permitiu a criação de uma base de dados e uma interface para a respectiva manipulação da mesma. A maior parte dos objetivos iniciais foi cumprida, não sendo possível apenas a interação entre utilizadores e a visualização da sua página de perfil.

Realça-se a importância de um bom planeamento e estruturação da base de dados, de modo a facilitar a interação com a mesma, de forma eficiente e segura.