Project Proposal

Team Members:

- Tamer Boutros
- Abed Karim Nasreddine
- Karl Skaff
- Charbel El Haddad

Stored Information:

- User - this entity will store information about each user of the app, including their name, email address, password, and subscription status.
- Artist - this entity will store information about each music artist, including their name and popularity score.
- Album - this entity will store information about each music album, including its title, release date, and the artist who created it.
- Track - this entity will store information about each individual track, including its title, length, and the album it belongs to.
- Playlist - this entity will store information about each playlist, including its name, description, and the tracks it contains.
- Genre - this entity will store information about each music genre, including its name and description.

Relationships between entities:

- User-Playlist - Each user can have multiple playlists. This relationship will be represented by a foreign key in the Playlist entity, which references the primary key in the User entity.
- Album-Artist - Each album is created by a specific artist. This relationship will be represented by a foreign key in the Album entity, which references the primary key in the Artist entity.
- Album-Track - Each album contains multiple tracks. This relationship will be represented by a foreign key in the Track entity, which references the primary key in the Album entity.

- Track-Playlist - Each playlist can contain multiple tracks. This relationship will be represented by a junction table that contains a foreign key to both the Track and Playlist entities.
- Track-Genre - Each track can belong to one genre. This relationship will be represented by a foreign key in the Track entity, which references the primary key in the Genre entity.

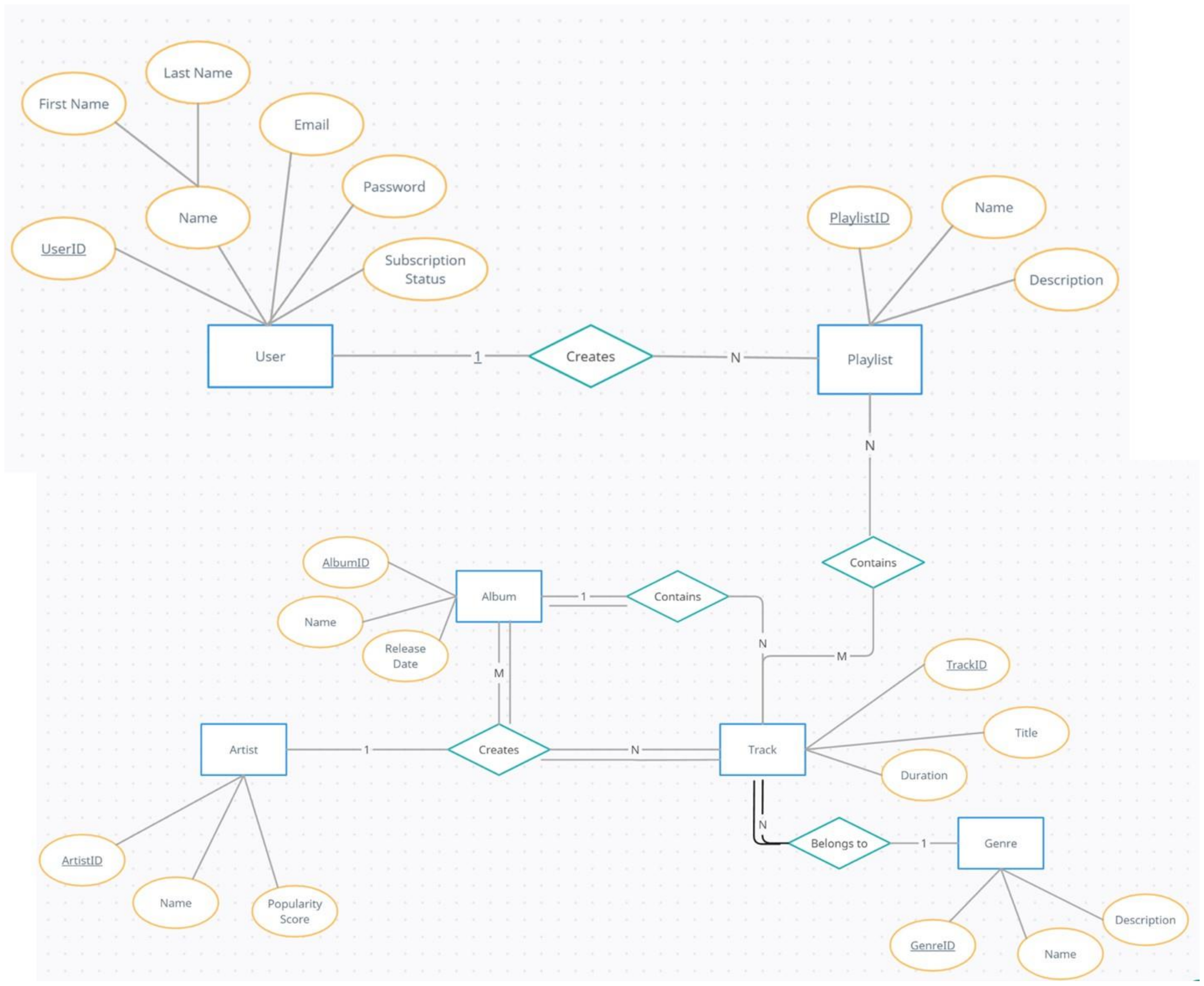
Constraints:

Constraints for these entities and relationships can be added as necessary to ensure the accuracy and consistency of the data stored in the database. For example, the User ID, Artist ID, Album ID, Track ID, Playlist ID, and Genre ID can be enforced as unique IDs by the primary key constraints. The foreign key constraints can be used to enforce the relationships between the entities. The NOT NULL constraint can be used to ensure that each playlist belongs to a user, each track belongs to an album and a genre, and each album is created by a specific artist.

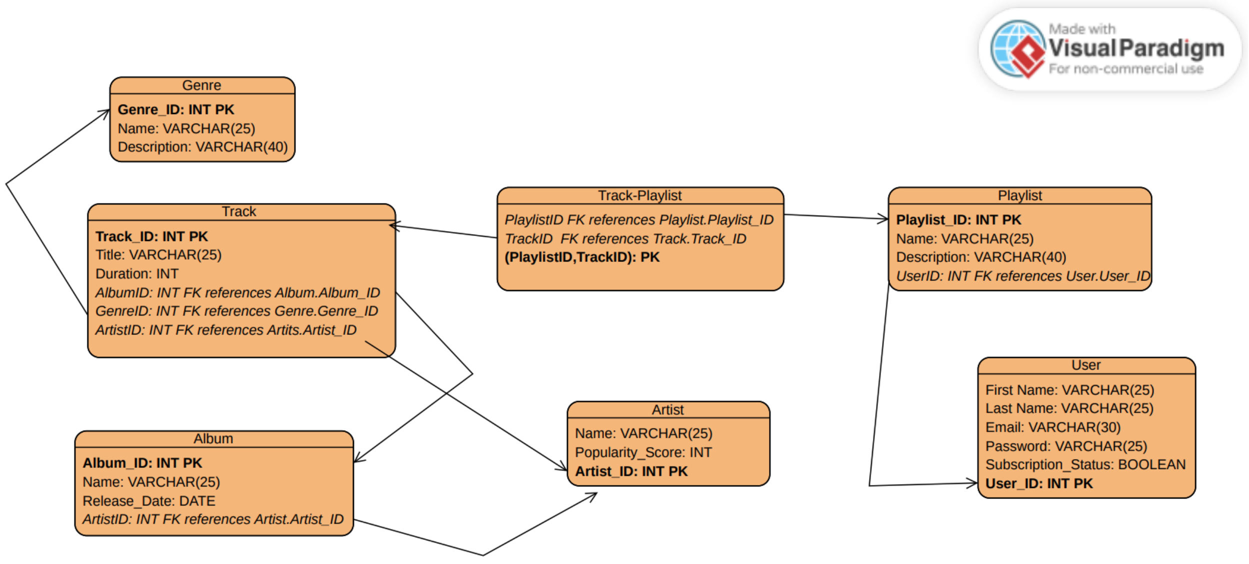
Typical Information:

Typical information that will likely be retrieved from this database includes the list of all tracks and albums by a specific artist, the list of all playlists created by a user, the tracks contained in a specific playlist, and all tracks belonging to a specific genre. Queries can be written to retrieve this information, such as "SELECT * FROM Track WHERE ArtistID = [Artist ID]" or "SELECT * FROM Track WHERE GenreID = [Genre ID]".

Enhanced Entity Relationship Model:



Relational Database Schema Diagram:



Indexes:

The following indexes were used:

- track.track_id: This is the primary key of the track table and is used in all views. Indexing this field will speed up queries that involve joining the track table with other tables.
- artist.artist_id: This is the primary key of the artist table and is used in all views. Indexing this field will speed up queries that involve joining the artist table with other tables.
- album.album_id: This is the primary key of the album table and is used in two views. Indexing this field will speed up queries that involve joining the album table with other tables.
- genre.genre_id: This is the primary key of the genre table and is used in two views. Indexing this field will speed up queries that involve joining the genre table with other tables.
- artist.genre_id: This is a foreign key in the artist table, which is used in one view. Indexing this field will speed up queries that involve joining the artist table with the genre table.
- user.user_id: This is the primary key in the user table, that I used in views and in all authentication operations. An index on the user_id column can speed up the authentication process by allowing the database to quickly locate the user record based on the user_id.
- track_playlist.track_id and track_playlist.playlist_id: The index playlist_track_index created on the track_playlist table will help in optimizing the performance of queries that involve searching, filtering or joining records based on the playlist_id and track_id fields, as they are also used in a view. Since playlist_id and track_id are used as foreign keys, the index will also help in joining records with their corresponding records in the parent tables (Playlist and Track tables) faster. Having an index on the primary key (playlist_id, track_id) is also a good practice as it ensures the uniqueness of the combination of the two fields and enforces data integrity.

  This is the code used for the indexes:

  ```
  CREATE INDEX user_index ON user (user_id);
  CREATE INDEX playlist_track_index ON track_playlist (playlist_id, track_id);
  CREATE INDEX track_index ON track (track_id);
  CREATE INDEX album_index ON album (album_id);
  CREATE INDEX genre_index ON genre (genre_id);
  CREATE INDEX artist_index ON artist (artist_id);
  ```

<u>Views:</u>

Views were created for each of the Album, artists, genre, playlist, track.

1. **genre_view**: This view combines the **genre** and **artist** tables to display the **genre_id**, **name**, **description** fields from the **genre** table and **name** field from the **artist** table. This view can be used to get a list of all genres and their associated artists.
2. **artist_view**: This view combines the **album** and **artist** tables to display the **artist_id**, **name**, **popularity_score**, **genre_id**, **album_name**, **release_date**, **album_id** fields. This view can be used to get a list of all artists, their associated albums and their popularity score.
3. **playlist_view**: This view combines the **playlist**, **track_playlist**, **track**, **album**, **artist** and **genre** tables to display the **playlist_id**, **name**, **track_title**, **duration**, **album_name**, **artist_name** and **genre_name** fields. This view can be used to get a list of all playlists and their associated tracks, along with details about the album, artist, and genre of each track.
4. **album_view**: This view combines the **album** and **artist** tables to display the **album_id**, **name**, **release_date**, and **artist_name** fields. This view can be used to get a list of all albums and their associated artist.
5. **track_view**: This view combines the **track**, **artist**, **album**, and **genre** tables to display the **track_id**, **title**, **duration**, **album**, **genre**, and **artist** fields. This view can be used to get a list of all tracks and their associated album, artist, and genre.

Each of these views is linked with an API in the backend server. These APIs are get apis used to display the output of a certain view based on the request. For example, if a user call the GET "/genres" api, the response will contain the output of the genre_view.

Here is the code for the views: CREATE

```
    ALGORITHM = UNDEFINED
    DEFINER = `root`@`localhost`
    SQL SECURITY DEFINER
VIEW `genre_view` AS
    SELECT
        `genre`.`genre_id` AS `genre_id`,
        `genre`.`name` AS `genre_name`,
        `genre`.`description` AS `description`,
        `artist`.`name` AS `artist_name`
    FROM
        (`genre`
        JOIN `artist` ON ((`genre`.`genre_id` = `artist`.`genre_id`)))
```

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = `root`@`localhost`
    SQL SECURITY DEFINER
VIEW `artist_view` AS
    SELECT
        `artist`.`artist_id` AS `artist_id`,
        `artist`.`name` AS `artist_name`,
        `artist`.`popularity_score` AS `popularity_score`,
        `artist`.`genre_id` AS `genre_id`,
        `album`.`name` AS `album_name`,
        `album`.`release_date` AS `release_date`,
        `album`.`album_id` AS `album_id`
    FROM
        (`album`
        JOIN `artist` ON ((`album`.`artist_id` = `artist`.`artist_id`)))
```

```
CREATE
    ALGORITHM = UNDEFINED
    DEFINER = `root`@`localhost`
    SQL SECURITY DEFINER
VIEW `playlist_view` AS
    SELECT
        `playlist`.`playlist_id` AS `playlist_id`,
        `playlist`.`name` AS `playlist_name`,
        `track`.`title` AS `track_title`,
        `track`.`duration` AS `duration`,
        `album`.`name` AS `album_name`,
        `artist`.`name` AS `artist_name`,
        `genre`.`name` AS `genre_name`
    FROM
        (((((`playlist`
        JOIN `track_playlist` ON ((`playlist`.`playlist_id` = `track_playlist`.`playlist_id`)))
        JOIN `track` ON ((`track_playlist`.`track_id` = `track`.`track_id`)))
        JOIN `album` ON ((`track`.`album_id` = `album`.`album_id`)))
        JOIN `artist` ON ((`track`.`artist_id` = `artist`.`artist_id`)))
        JOIN `genre` ON ((`track`.`genre_id` = `genre`.`genre_id`)))
```

```
CREATE
   ALGORITHM = UNDEFINED
   DEFINER = `root`@`localhost`
   SQL SECURITY DEFINER
VIEW `album_view` AS
   SELECT
      `album`.`album_id` AS `album_id`,
      `album`.`name` AS `album_name`,
      `album`.`release_date` AS `release_date`,
      `artist`.`name` AS `artist_name`
   FROM
      (`album`
      JOIN `artist` ON ((`album`.`artist_id` = `artist`.`artist_id`)))


CREATE
   ALGORITHM = UNDEFINED
   DEFINER = `root`@`localhost`
   SQL SECURITY DEFINER
VIEW `track_view` AS
   SELECT
      `track`.`track_id` AS `track_id`,
      `track`.`title` AS `track_name`,
      `track`.`duration` AS `duration`,
      `album`.`name` AS `album`,
      `genre`.`name` AS `genre`,
      `artist`.`name` AS `artist`
   FROM
      (((`track`
      JOIN `artist` ON ((`track`.`artist_id` = `artist`.`artist_id`)))
      JOIN `album` ON ((`track`.`album_id` = `album`.`artist_id`)))
      JOIN `genre` ON ((`track`.`genre_id` = `genre`.`genre_id`)))
```

The interface and the backend:

Our project is mainly composed of a Python Flask backend with a react javascript frontend. Although the backend with all the corresponding functions is complete and tested using Postman, the frontend unfortunately could not be completed on time as the deadline cut us off before we could complete it. However, we will continue working on the frontend for a day or two maximum so we have a complete interface at the presentation. When it comes to the backend, we provided a link with the documentation for all the functions and API routes we created. As a brief description, there are 3 different user types that are the admin, the subscribed non-admin user, and the non-subscribed non-admin user. The admin user has the create, edit, and delete privileges for the albums, artists, tracks, playlists… basically everything. A user is made admin by manually setting the Boolean field of is_admin =1 in the user table. The subscribed user has read-only rights to everything except playlists, as he can create playlists and add tracks to it, however he does not have the right create, delete, or edit anything but his own account. The unsubscribed user can only view things.

Now when it comes to running the APIs, most of them require authentication, which is handled by a token that is given to the user after logging in. This bearer token is then passed as authorization to the operations, which fail in case there is a problem with the token or the user is not signed in… The case for is_admin is also handled in all the corresponding functions in the backend, as through this authentication, the user_id can be decoded which allows the accessing of the user info. This will then allow us to check whether user is an admin or not to determine whether the api should work or fail.

As to what functionalities are included in the backend, please check out the link below which shows all the APIs with a description of what they do, what request they take, and what response they generate.

https://documenter.getpostman.com/view/25583565/2s93Xu3Rgk

this is for the postman. The following also explains each api:

API to get a user by user_id:

Description: Retrieves user information by user_id

Method: ‘GET’

Parameters:

- ‘user_id’ (integer): Required. The ID of the user to retrieve details for

Returns:

- If the user exists:
  - ‘user_id’ (integer): The ID of the user
  - ‘first_name’ (string): The first name of the user

    - ‘last_name’ (string): The last name of the user
    - ‘email’ (string): The email address of the user
    - ‘subscription_status’ (boolean): The subscription status of the user
    - ‘is_admin’ (boolean): The admin status of the user

- If the user does not exist:
    - ‘message’ (string): A message indicating that the user was not found

<u>Success Response:</u>
- Code: 200
- Content:

```
{
   "user_id": 1,
   "first_name": "John",
   "last_name": "Doe",
   "email": "johndoe@example.com",
   "subscription_status": "active",
   "is_admin": true
}
```

<u>Error Response:</u>
- Code: 404
- Content:

```
{
   "message": "User not found"
}
```

<u>Data Types:</u>
- ‘user_id’: integer
- ‘first_name’: string
- ‘last_name’: string
- ‘email’: string
- ‘subscription_status’: boolean
- ‘is_admin’: boolean

API to create a new user:

<u>Endpoint:</u> /users/create

Description: Creates a new user

Method: “POST”

Parameters:
- ‘first_name’ (string): First name of the user. Required
- ‘last_name’ (string): Last name of the user. Required
- ‘email’ (string): Email address of the user. Required and must be unique
- ‘password’ (string): Password of the user. Required

- ‘subscription_status’ (boolean): Subscription status of the user. Required

Returns:

- If the user is created successfully
  - ‘message’ (string): A message indicating that the user was created
  - ‘user_id’ (int): ID of the user created
- If the user creation fails
  - ‘message’ (string): A message indicating that the user creation failed

Success Response:

- Code: 201
- Content:

```
{
“message”: “User created”,
“user_id: 1
}
```

Error Response:

- Code: 400
- Content:

```
{
“message”: “Bad request. Please provide all required fields”
}
```

Data Types:

- ‘first_name’: string
- ‘last_name’: string
- ‘email’: string
- ‘password’: string
- ‘subscription_status’: boolean

<u>API to update an existing user:</u>

<u>Endpoint:</u> /users/edit/int:user_id

<u>Method:</u> PUT

<u>Description:</u> This endpoint updates user details for the given user_id

<u>Parameters:</u>

- user_id(integer): Required. The ID of the user to update details for.

<u>Request Body:</u>

- The request body must be a JSON object containing the following keys:
  - First_name (string): The first name of the user
  - Last_name (string): The last name of the user
  - Email (string): The email address of the user
  - Password (string): The password of the user
  - Subscription_status (boolean): The subscription status of the user
  - Is_admin (boolean): the admin status of the user

Returns:

- If the user exists:
    - Message(string): A message indicating that the user was updated
- If the user does not exist:
    - Message (string): A message indicating that the user was not found

Success Response:

- Code: 200
- Content:

```
{
"message": "User updated"
}
```

Error Response:

- Code: 404
- Content:

```
{
"message": User not found"
}
```

Data Types:

- user_id: integer
- first_name: string
- last_name: string
- email: string
- password: string
- subscription_status: boolean
- is_admin: boolean

Retrieves User Information by User ID

Endpoint: /users/int:user_id

Method: GET

Description: Retrieves user information for the given user ID

Parameters:

- 'user_id' (integer): Required. The ID of the user to retrieve details for

Returns:

- If the user exists:
    - 'user_id' (integer): The ID of the user
    - 'first_name' (string): the first name of the user
    - 'last_name' (string): the last name of the user
    - 'email' (string): the email address of the user
    - 'subscription_status' (boolean): The subscription status of the user
    - 'is_admin' (boolean): The admin status of the user
- If the user does not exist:

- ‘message’ (string) A message indicating that the user was not found.

Success Response:

- Code: 200
- Content:

```
{
“user_id”: 1,
“first_name”: “John”,
“last_name”: “Doe”,
“email” : “johndoe@example.com”,
“subscription_status” : true,
“is_admin”: false
}
```

Error Response:

- Code: 404
- Content:

```
{
“message”: “User not found”
}
```

<u>Data Types:</u>

- ‘user_id’: integer
- ‘first_name’: string
- ‘last_name’: stirng
- ‘email’: string
- ‘subscription_status’ : boolean
- ‘is_admin’ : boolean

<u>**Delete User**</u>

Endpoint: /users/delete/int:user_id
Method: DELETE
Description: Deletes a user with the given user ID
Parameters:

- ‘user_id’ (integer): Required. The ID of the user to delete

Returns:

- ‘message’ (string): A message indicating that the user was deleted.

Success Response:

- Code:200
- Content:
  - {
  - “message” : “User deleted”
  - }

Error Response:
- Code: 404
- Content:

  {
  “message” : “User not found”
  }

Data Types:
- ‘user_id’ : integer

<u>**Get All Tracks:**</u>
<u>Endpoint:</u> /tracks
<u>Method:</u> GET
<u>Description:</u> Retrieves all tracks from the track_view table
<u>Parameters:</u>
- None

<u>Returns:</u>
- If tracks are found:
  - ‘track_id’ (integer): The ID of the track
  - ‘title’ (string): The title of the track
  - ‘duration’ (integer): the duration of the track in seconds
  - ‘album’ (string): The name of the album the track belongs to
  - ‘genre’ (string): The genre of the track
  - ‘artist’ (string): the name of the artist who created the track
- If the track does not exist
  - ‘message’ (string): A message indicating that the track could not be found

<u>Success Response:</u>
- Code: 200
- <u>Content:</u>

```
[
{
“track_id” : 1,
“title” : “Track 1”,
“duration”: 180,
“album” : “Album 1”,
“genre” : “Pop”,
“artist”: “Artist 1”
},
{
“track_id”: 2,
“title”: “Track 2”,
“duration” : 210,
“album”: “Album 2”,
“genre”: “Rock”,
```

“artist”: Artist 2”
}
]

Error Response:
- Code: 404
- Content:
  {
  “message”: “No tracks found”
  }

Data Types:
- ‘track_id’: integer
- ‘title’: string
- ‘duration’ : integer
- ‘album’ : string
- ‘genre’ : string
- ‘artist’ : string

Get a single track by ID:
Endpoint: /tracks/int:track_id
Description: Retrieves track information by track_id
Method: GET
Parameters:
- ‘track_id’ (integer): Required. The ID of the track to retrieve details for

Returns:
- If the track exists
  - ‘track_id’ (integer): The ID of the track
  - ‘title’ (string): The title of the track
  - ‘duration’ (integer): the duration of the track in seconds
  - ‘album’ (string): The name of the album the track belongs to
  - ‘genre’ (string): The genre of the track
  - ‘artist’ (string): the name of the artist who created the track
- If the track does not exist
  - ‘message’ (string): A message indicating that the track could not be found

Success Response:
- Code: 200
- Content: The JSON object containing the track details

Error Responses:
- Code: 404
- Content: The JSON object containing the error message

Data Types:

- ‘track_id’: integer
- ‘title’: string
- ‘duration’: integer
- ‘album_id’: integer
- ‘genre_id’: integer
- ‘artist_id’: integer

Adding a new track

Endpoint: /tracks/create

Method: POST

Description: Adds a new track to the database

Headers:

- ‘Authorization’ (string): Required. The token obtaind by logging in with valid user credentials

Parameters:

- ‘title’ (string): Required. The title of the new track
- ‘duration’ (integer): Required. The duration of the new track in seconds
- ‘album_id’ (integer): Required. The ID of the album that the new track belongs to
- ‘genre_id’ (integer): Required. The ID of the genre that the new track belongs to
- ‘artists_id’ (integer): Required. The ID of the artists that the new track belongs to

Returns:

- If track was added successfully
  - ‘message’ (string): A message indicating that the track was added successfully

Success Response:

- Code: 201
- Content:

```
{
'message' : 'Track added successfully',
'track_id': <new_track.track_id>'
}
```

Error Response:

- Code: 400
- Content:

```
{
'error': 'Invalid JSON format'
}
```

- Code: 401

- Content:

  {
  'error': 'Authorization header is missing'
  }

  Or

  {
  'error': 'Invalid Token'
  }

- Code: 403
- Content:

  {
  'error': 'Only admins can add tracks'
  }

<u>Data Types:</u>

- 'title' : string
- 'duration': integer
- 'album_id' : integer
- 'genre_id': integer
- 'artist_id': integer
- 'track_id': integer

<u>Updating an existing track:</u>

<u>Endpoint:</u> /tracks/track_id
<u>Method:</u> PUT
<u>Description:</u> Update an existing track
<u>Parameters:</u>

- 'title' (string): Required. The title of the new track
- 'duration' (integer): Required. The duration of the new track in seconds
- 'album_id' (integer): Required. The ID of the album that the new track belongs to
- 'genre_id' (integer): Required. The ID of the genre that the new track belongs to
- 'artists_id' (integer): Required. The ID of the artists that the new track belongs to

<u>Success Response:</u>

- Code: 200
- Content:

  {
  'message' : 'Track updated successfully'
  }

<u>Error Response:</u>

- Code: 400

- Content:

  {
  'error': 'Invalid JSON format'
  }

- Code: 401
- Content:

  {
  'error': 'Authorization header is missing'
  }

  Or

  {
  'error': 'Invalid Token'
  }

- Code: 403
- Content:

  {
  'error': 'Only admins can add tracks'
  }

- Code: 404
- Content: None

Data Types:

- 'title' : string
- 'duration': integer
- 'album_id' : integer
- 'genre_id': integer
- 'artist_id': integer
- 'track_id': integer

Deleting a Track:

Endpoint: /tracks/track_id

Method: DELETE

Description: Deletes an existing track

Parameters:

- None

Success Response:

- Code: 200
- Content:

  {

‘message’ : ‘Track deleted successfully’
}

Error Response:

- Code: 400
- Content:

    {
    ‘error’: ‘Invalid JSON format’
    }

- Code: 401
- Content:

    {
    ‘error’: ‘Authorization header is missing’
    }

    Or

    {
    ‘error’: ‘Invalid Token’
    }

- Code: 403
- Content:

    {
    ‘error’: ‘Only admins can add tracks’
    }

- Code: 404
- Content: None
-

Data Types:

- None

API to create a playlist:

Endpoint: /playlists/create

Method: POST

Description: This endpoint creates a new playlist with the provided name and description for the authenticated user

Parameters:

- ‘name’ (string): Required. The name of the playlist
- ‘description’ (string): Optional. The description of the playlist

Success Response:

- Code 200
- Content:

  {
  'message' : 'Playlist created successfully!'
  }

Error Response:
- Code 401
- Content:

  {
  'error': Authorization header is missing'
  }
- Code 400
- Content:

  {
  'error': Invalid JSON format'
  }

API for editing playlist:

Endpoint: /playlists/edit/playlist_id

Method: PUT

Description: This endpoint allows a user to update the name and description of an existing playlist, provided they are the owner of the playlist

Parameters:
- 'playlist_id' (integer): The ID of the playlist to be edited

Request Headers:
- 'Authorization' (string): A valid JWT token for authentication

Request Body:
- 'name' (string): Optional. The new name for the playlist
- 'description' (string): Optional. The new description of the playlist

Success Response:
- Code: 200
- Content:

  {
  'message' : 'Playlist updated successfully!'
  }

Error Responses:
- Code 401
- Content:

  {
  'error' : 'Unauthorized access!'

}

Or
{
'error' : 'No authorization token provided!'
}

- Code 404
- Content: None

<u>API to delete a playlist:</u>

Endpoint: /playlists/delete/<int:playlist_id>
Method: DELETE
Description: Deletes a specific playlist from the database, if authorized.
Parameters:

- playlist_id (integer) - Required. ID of the playlist to be deleted.

Request Headers:

- Authorization (string) - Required. Token for user authorization.

Success Response:

- Code 200
- Content:

```
{
'message' : 'Playlist deleted successfully!'
}
```

Error Responses:

- Code 401
- Content:

```
{
'error' : 'Unauthorized access!'
}

Or
{
'error' : 'No authorization token provided!'
}
```

- Code 404
- Content: None

<u>Add Track to Playlist</u>
<u>Endpoint:</u> /playlists/add_track
<u>Method:</u> POST
<u>Description:</u> This API route allows users to add a track to a playlist
<u>Parameters:</u>

- 'playlist_id' (integer): Required. The ID of the playlist to which the track should be added
- 'track_id' (integer): Required. The ID of the track to be added to the playlist

<u>Success Response:</u>

- Code 200
- Content:

```
{
'message' : 'Track was added successfully'
}
```

<u>Error Response:</u>

- Code 400
- Content:

```
{
'error' : 'Track already in playlist'
}
```

- Code 401
- Content:

```
{
'error' : 'You are not authorized to add tracks to this playlist'
}
```

- Code 404
- Content:

```
{
'error' : 'Playlist not Found'
}
```

<u>Remove Track from Playlist</u>
<u>Endpoint:</u> /playlists/remove_track
Method: DELETE
Description: This API is used to remove a track from a playlist. It requires the playlist_id and track_id as parameters in the request body. The user must be authorized to remove tracks from the playlist, and the playlist and track must exist in the database. The API returns a JSON response with a message indicating whether the operation was successful or not.
Parameters:

- playlist_id (integer): Required. The ID of the playlist from which the track should be removed.
- track_id (integer): Required. The ID of the track to be removed from the playlist.

Headers:

- Authorization: Required for authentication. The token should be sent in the format Bearer <token>.

Success Response:

- 200 OK: If the track is removed from the playlist successfully.

Error Response:

- 401 Unauthorized: If the user is not authorized to remove tracks from the playlist.
- 404 Not Found: If the playlist or the track is not found in the database.

<u>**Get Track from Playlist**</u>

Endpoint: /playlist/tracks
Method: GET
Description: This endpoint retrieves a list of all tracks in the playlists belonging to the authenticated user. It first checks if the request is authorized and if so, retrieves the user's playlists from the database. It then executes a SELECT query on the playlist_view table to fetch all records of tracks in all playlists. It filters the results to only include tracks in the user's playlists and serializes the result set into a list of dictionaries with the following keys: playlist_id, playlist_name, track_title, duration, album_name, artist_name, and genre_name. It returns the serialized list of playlists in JSON format.
Headers:

- Authorization: Required for authentication. The token should be sent in the format Bearer <token>.

Success Response:

- 200 OK: The request was successful, and the list of playlists is returned in the response body.

Error Response:

- 401 Unauthorized: The user is not authenticated.

<u>**Get Playlist**</u>

Endpoint: /playlist/int:user_id
Method: GET
Description: Returns a list of playlists associated with a specific user, or all playlists if ‘user_id’ is not provided
Parameters:

- ‘user_id’ (integer): Optional. The ID of the user whose playlists to retrieve. If not provided, all playlists will be returned.

<u>Response:</u>

- If playlist was found
    - ‘playlist_id’ (integer): The unique identifier of the playlist
    - ‘name’ (string): The name of the playlist
    - ‘description’(string): The description of the playlist

    - ‘user_id’ (integer): The ID of the user who created the playlist

Success Response:

- 200 OK: The request was successful. Returns a JSON object containing a list of playlist objects

Error Response:

- 404 Not found: The requested user ID was not found

Add Artist:

Endpoint: /artists/create

Method: POST

Parameters:

- ‘name’ (string): Required. The name of the artist
- ‘popularity_score’ (integer): Optional. The popularity score of the artist (between 0 and 100)
- ‘genre_id’ (integer): Required. The ID of the genre that the artist belongs to.

Success Response:

- 201 Created: The artists was added successfully

Error Responses:

- 400 Bad Request: The request body is not valid JSON or is missing required fields
- 401 Unauthorized: The authorization header is missing or invalid
- 403 Forbidden: The authenticated user is not an admin

Edit Artist Details:

Endpoint: /artists/{artist_id}

Method: PUT

Description: Updates the details of an existing artist. Only admins can edit artists.

Method: PUT

Parameters:

- artist_id(integer): Required. The ID of the artist to update

Request Body

The request body must be in JSON format, and must include the following fields:

- name (string) the name of the artist
- popularity_score (float) The popularity score of the artist
- genre_id(integer) The ID of the genre of the artist

Success Response

- 200 OK: A message indicating the success of the operation

Error Responses

- 400 Bad Request A message indicating that the request body is not in JSON format
- 401 Unauthorized A message indicating that the authorization header is missing or a message indicating that the token is invalid
- 403 Forbidden A message indicating that only admins can edit artists

- 404 Not Found: A message indicating that the artist with the given ID was not found

Delete an Artist:
Endpoint: /artists/<int:artist_id>
Method: DELETE
Description: Delete an existing artist from the database.
Request Parameters:

- artist_id (integer, required): The ID of the artist to be deleted.

Request Headers:

- Authorization (string, required): The JWT token for authentication.

Request Body:

- None

Response:

- 200 OK if artist is successfully deleted.
- 400 Bad Request if the request body is not in JSON format.
- 401 Unauthorized if the Authorization header is missing or invalid.
- 403 Forbidden if the user is not an admin.
- 404 Not Found if the artist with the given ID is not found.

Get All Artists:
Endpoint: /artists
Method: GET
Description: Returns a list of all artists in the database, along with their information.
Parameters:

- None

Response

- 200 OK on success, along with a JSON object containing the list of artists.
- 500 Internal Server Error on failure.

Add a new Genre:
Endpoint: /genres/create
Method: POST
Description: Adds a new Genre

Request Body:

- name (required): A string representing the name of the genre.
- description (optional): A string representing a description of the genre.

Responses:

- 201 Created: If the genre was created successfully.
- 400 Bad Request: If the request body is missing or not in JSON format.

- 401 Unauthorized: If the authentication token is missing or invalid.
- 403 Forbidden: If the user is not an admin.

Edit a new Genre:

Endpoint: /genres/<int:genre_id>
Method: PUT
Description: Update the details of a genre
Request Headers:

- Authorization : Token <access_token>

Request Body:
JSON object containing the updated details of the genre

- name (optional): New name of the genre
- description (optional): New description of the genre

Response:

- 200 OK: Genre updated successfully

JSON object containing the message "Genre updated successfully"

- 400 Bad Request: Invalid JSON format

JSON object containing the error message "Invalid JSON format"

- 401 Unauthorized: Authorization header is missing or invalid token

JSON object containing the error message "Authorization header is missing" or "Invalid token"

- 403 Forbidden: Only admins can edit genres

JSON object containing the error message "Only admins can edit genres"

- 404 Not Found: Genre not found

JSON object containing the error message "Genre not found"

Deleting an existing Genre:

Endpoint: /genres/int:genre_id
Method: DELETE
Description: Delete a genre from the database

Request Headers:

- Authorization: Bearer token

Request Parameters:

- genre_id (int, required): The ID of the genre to be deleted

Response:

- 200 OK: Genre deleted successfully

JSON Object containing the message "Genre deleted successfully"

- 401 Unauthorized: Authorization header is missing or token is invalid

JSON Object containing the error message "Authorization header is missing" or "Invalid token"

- 403 Forbidden: Only admins can delete genres

JSON Object containing the error message "Only admins can delete genres"

- 404 Not Found: Genre not found in the database

JSON Object containing the error message "Genre not found"

Display All Genres:
Endpoint: /genres
Method: GET
Description: Retrieve a list of all genres
Response:

- 200 OK: Successfully retrieved the list of genres

JSON Array of genre objects:

- genre_id: ID of the genre (integer)
- genre: Name of the genre (string)
- description: Description of the genre (string)
- artist_name: Name of the artist associated with the genre (string)

Add New Album:
Endpoint: /albums/create
Method: POST
Description: Adds a new album to the database.
Request Body:

- name (string, required): The name of the album.
- release_date (string, required): The release date of the album in ISO format (e.g. "2023-04-11").
- artist_id (integer, required): The ID of the artist who released the album.

Response:
If successful, returns a JSON object containing a success message and the ID of the newly added album.
If the request is not in JSON format, returns a JSON object with an error message and a 400 status code.
If the user is not authorized, returns a JSON object with an error message and a 401 status code.

Edit Album:
Method: PUT
Endpoint: /albums/<album_id>
Description: Edits an album with the given album_id.
Request Header:

- Authorization: Bearer <auth_token>

Request Body:

- "name" (string):
- "release_date": "string",
- "artist_id": integer

Response:

- Returns 200 OK if the album was successfully updated.
- Returns 401 Unauthorized if the authorization header is missing or the token is invalid.
- Returns 404 Not Found if the album with the given album_id does not exist.
- Returns 400 Bad Request if the request body is not in JSON format.

Delete Album:

Method: DELETE
Endpoint: /albums/<album_id>
Description: Deletes an album with the given album_id.
Request Header:

- Authorization: Bearer <auth_token>

Response:

- Returns 200 OK if the album was successfully deleted.
- Returns 401 Unauthorized if the authorization header is missing or the token is invalid.
- Returns 403 Forbidden if the user is not an admin.
- Returns 404 Not Found if the album with the given album_id does not exist.

Get All Albums:

Method: GET
Endpoint: /albums
Description: Retrieves a list of all albums.
Request Header:

- None

Request Body:

- None

Response:

- Returns 200 OK with a list of all albums in JSON format.
- Returns 500 Internal Server Error if there was an issue with the database connection.