

# *Apollofy Music Project*

What is Apollofy?	3
<b>Architecture Diagram</b>	<b>4</b>
<b>Why Spring Boot + JPA?</b>	<b>5</b>
<b>Resources to implement the Backend</b>	<b>5</b>
1. The online Swagger documentation of the API:	5
2. Jar file to execute the backend locally in your computer:	5
3. Open source (MIT License) Android application that interacts with the REST API:	7
4. Demos interaction between the Android application and the Spring Boot backend:	9
Sample Music uploaded to the platform:	9
Why the name Apollofy?	9
<b>Epics</b>	<b>9</b>
<b>Sample User stories</b>	<b>10</b>
1.- Account Management	10
2.- Song genres	10
3.- Publication, visualization and management of songs (tracks)	11
4.- Publication, display and management of playlists	11
5.- Search in the platform	12
6.- Optional Front-end	12
<b>User's level</b>	<b>12</b>
<b>Beyond the conventional... you will reach the glory!</b>	<b>13</b>
<b>Sprint 1</b>	<b>13</b>
Genres CRUD	13
Tracks CRUD	14
<b>Sprint 2</b>	<b>15</b>
Playlist CRUD	15
POST example:	16
PUT example:	17
PUT example to Add Tracks:	19
GET example with Id:	21
Official Spotify API	22
Delete example with Id:	23
Search Users, Tracks and Playlists	24
Search Architecture Diagram	24
SearchDiagrama.drawio.png	24
Search Playlists	25

Search Tracks	31
<b>Professional API Search Examples:</b>	<b>33</b>
<b>Sprint 3</b>	<b>34</b>
Users, Roles and JWT Authentication	34
API endpoints: User activity related to tracks, playlists and artists	34
Important advice for third and fourth Sprints	35
Association Class JPA LikeTrack	36
Professional API Like and Follow Examples:	37
Postman Collections: Organize API Development and Testing	38
<b>Sprint 4</b>	<b>39</b>
Top Likes and Follow Endpoint	39
How to create and use the DTO class	40
Project Lombok very nice for DTOs ;-)	41
Add collaborators to playlists	42
Sample project:	42
Bonus feature:	42
Bring lyrics to your application!	43
Musixmatch Developer API Architecture Diagram	43
Other Music APIs you can explore	43
Spring Reference Documentation	44
Webclient Example	44

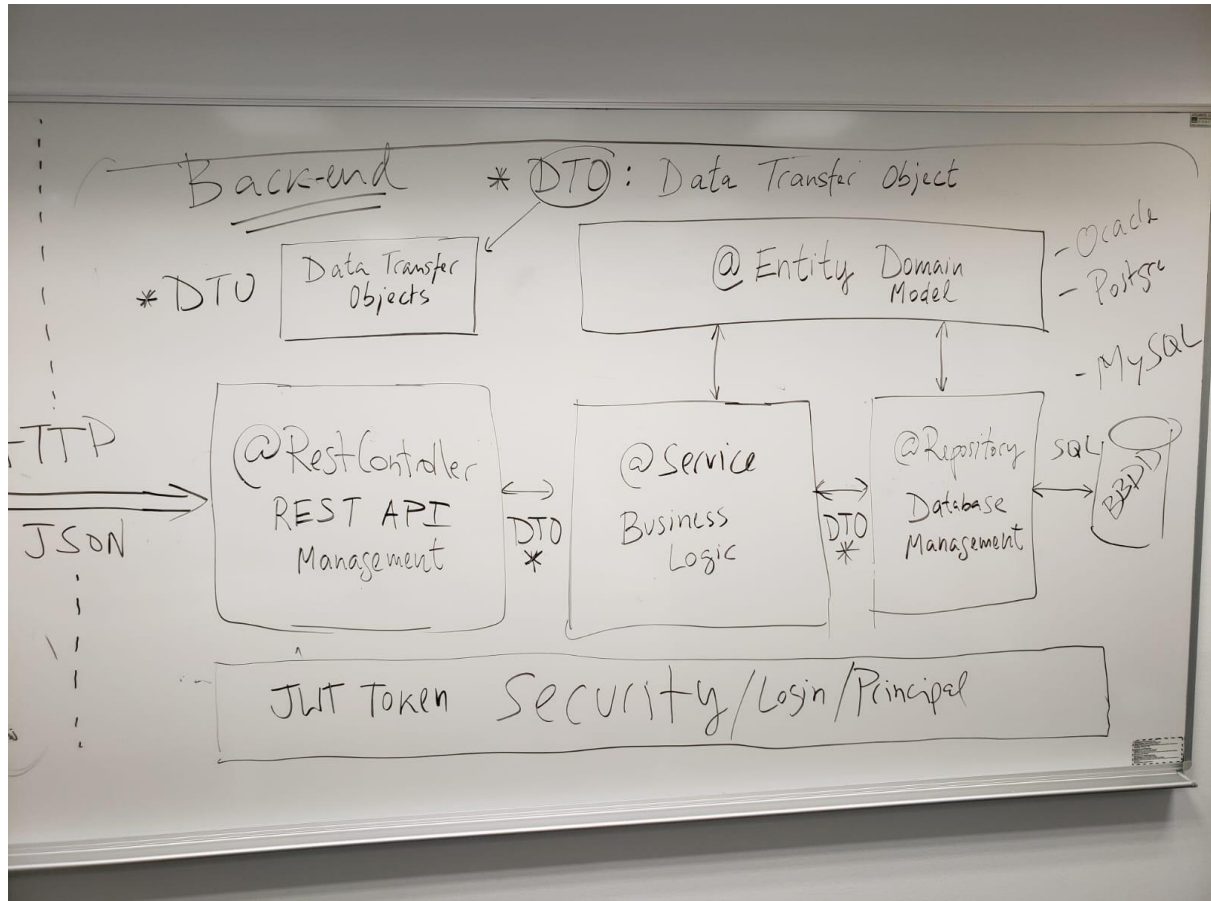
# What is Apollofy?

The objective of this project is to create a REST API based Backend using Java Spring Boot, Spring MVC, JPA and MySQL for the management of songs among a community of users.

Apollofy is going to be a music platform inspired on real applications like Spotify or SoundCloud.

Users will be able to access the application to play songs uploaded by themselves or songs uploaded by others. They can also create playlists and can “like” both songs and playlists. Song and image files will be stored on a cloud platform called Cloudinary <https://cloudinary.com/>

# Architecture Diagram



# Why Spring Boot + JPA?

[Java data access technology survey results - Vlad Mihalcea](#)

<https://www.jrebel.com/success/java-developer-productivity-report-2022>

[Java Programming - The State of Developer Ecosystem in 2021 Infographic | JetBrains](#)

## Resources to implement the Backend

You will have at your disposal several resources to implement your REST API:

### 1. The online Swagger documentation of the API:

<http://apollo.eu-west-3.elasticbeanstalk.com/developer/api>

User: test

Password: test

You can also create your own user.

The online API is provided exclusively to facilitate demonstrations during the first week. Having the backend published on AWS implies an important economic cost, so you will run the API directly on your computer, as specified [here](#).

You only need a JVM 11 or higher and a MySQL 8 or higher database server. Both products are free of charge.

### 2. Jar file to execute the backend locally in your computer:

[sallefy-0.2.4-BETA.jar](#)

```
C:\Users\Alfredo\Downloads\sallefy-back\target>java -jar sallefy-0.2.4-BETA.jar
```

The Spring Boot server expects a MySQL Database with the following credentials:

`datasource:`

`type: com.zaxxer.hikari.HikariDataSource`

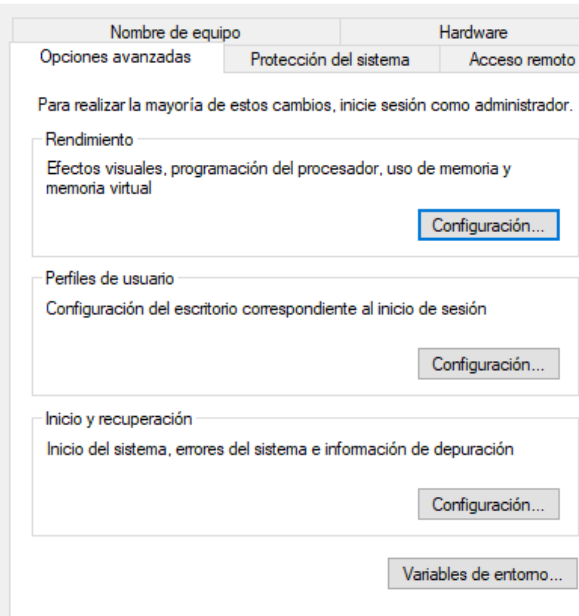
`url:`

`jdbc:mysql://localhost:3306/sallefy?useUnicode=true&characterEncoding=utf8&allowPublicKeyRetrieval=true&useSSL=false&useLegacyDatetimeCode=false&serverTimezone=UTC&createDatabaseIfNotExist=true`

```
username: ${SALLEFY_DB_USER}
password: ${SALLEFY_DB_PASSWORD}
```

Set environment variables via Windows UI:

Propiedades del sistema



Variables del sistema	
Variable	Valor
PROCESSOR_REVISION	3c03
PSModulePath	%Program
SALLEFY_DB_PASSWORD	root
SALLEFY_DB_USER	root

[Environment Variables for Java Applications - PATH, CLASSPATH, JAVA\\_HOME](#)

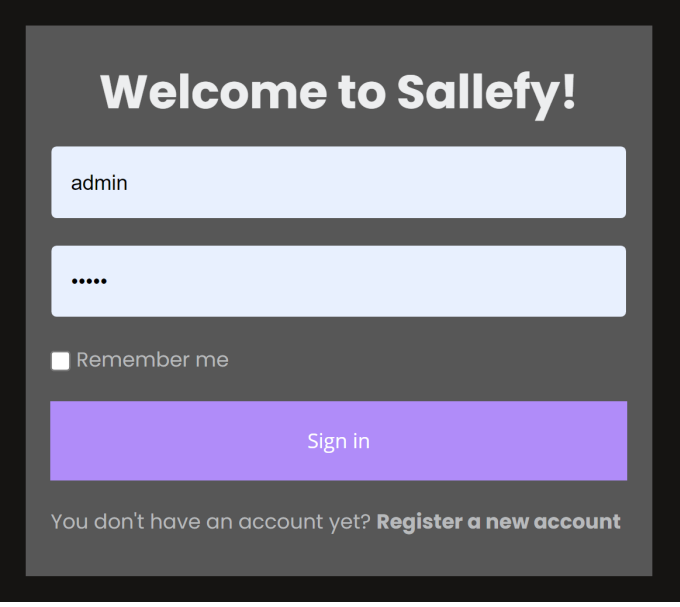
The Spring Boot server takes care of creating the database and the corresponding tables!

```
2022-10-17 00:46:23.066 INFO 4764 --- [main] com.sallefy.SallefyApp
n 40.724 seconds (JVM running for 42.231)
2022-10-17 00:46:23.127 INFO 4764 --- [main] com.sallefy.SallefyApp
-----
Application 'Sallefy' is running! Access URLs:
Local:      http://localhost:8080/
External:   http://192.168.1.150:8080/
Profile(s): [dev, swagger]
-----
```

Once the server is up and running, we can log in through the following address:

<http://localhost:8080/>

User: admin  
Password: admin

A screenshot of a login interface for an application named 'Sallefy'. The interface has a dark gray background. At the top, it says 'Welcome to Sallefy!' in white bold text. Below this, there are two light blue input fields. The first field contains the text 'admin'. The second field contains five dots, representing a password. Below the password field, there is a checkbox labeled 'Remember me'. At the bottom of the login area, there is a purple button with the text 'Sign in'. Below the button, there is a link that says 'You don't have an account yet? Register a new account'.

We can login with the admin user, or create a new user.  
<https://vimeo.com/760886290/9e0ebb491c>

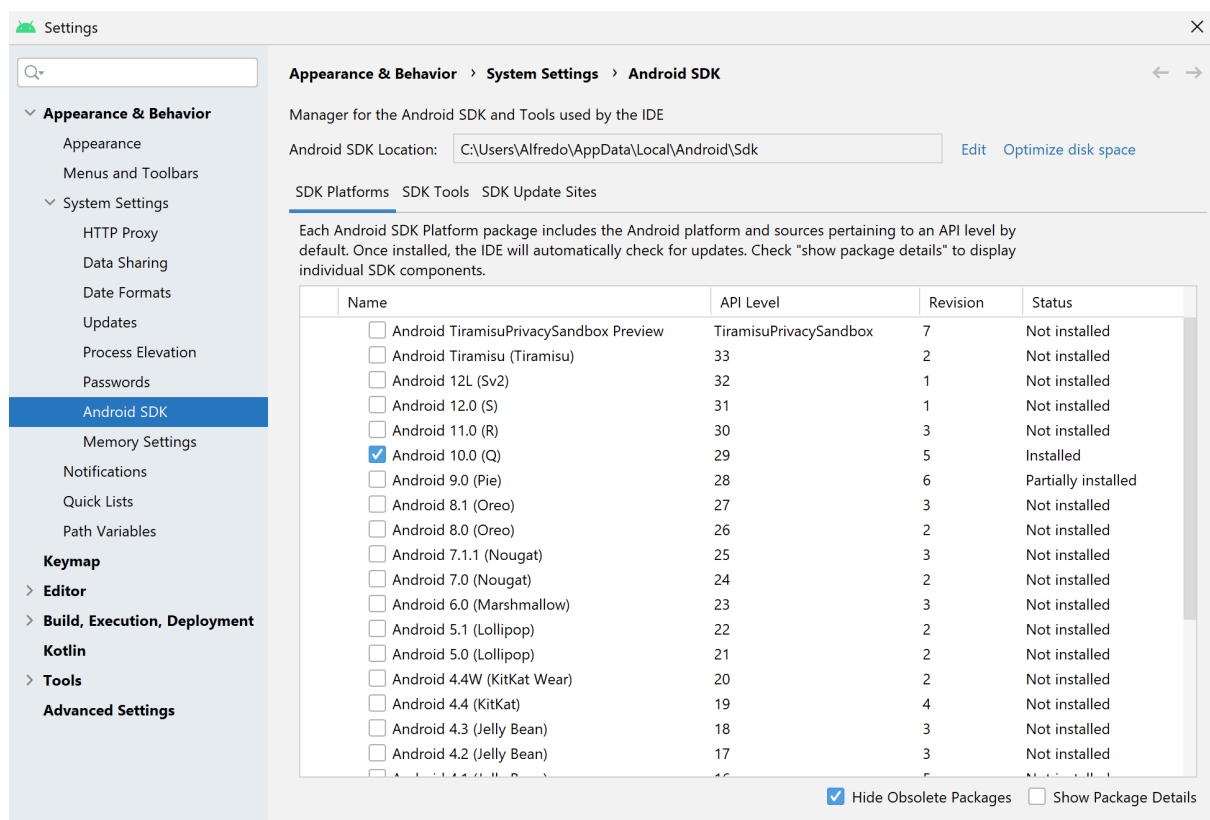
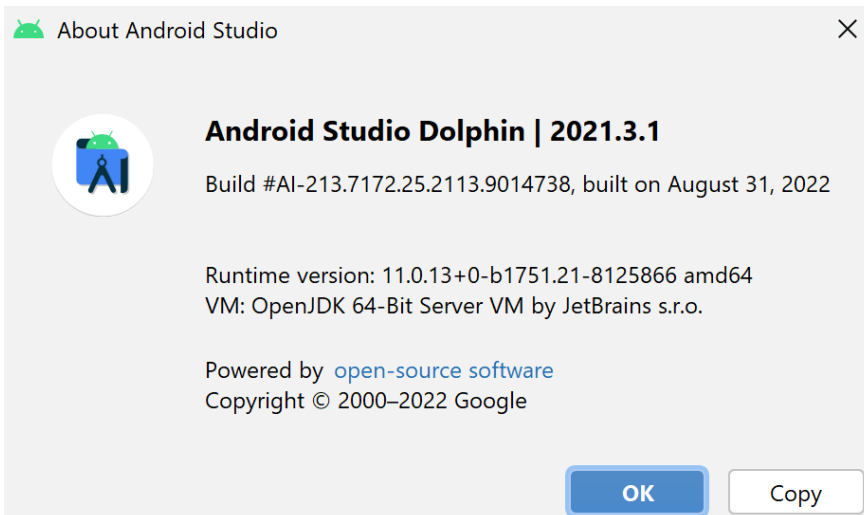
### 3. Open source (MIT License) Android application that interacts with the REST API:

<https://github.com/alfredorueda/Sallefy-AndroidApp>

To configure the Base URL that points to the API:

<https://github.com/alfredorueda/Sallefy-AndroidApp/blob/master/app/src/main/java/com/example/sallefy/SallefyApplication.java#L9>

The application can be compiled directly with Android Studio. It has been tested and works correctly with Android API 26 or higher.





#### 4. Demos interaction between the Android application and the Spring Boot backend:

<https://vimeo.com/759914348/87dae94440>

<https://vimeo.com/759924764/f999de8a93>

<https://vimeo.com/759930734/0bb1d6ea93>

<https://vimeo.com/759935219/813d736227>


<https://vimeo.com/759939989/c15a17d779>

<https://vimeo.com/759948205/93dacbeba8>

<https://vimeo.com/760821317/4d4779a04f>

<https://vimeo.com/760856771/a289461239>

#### Sample Music uploaded to the platform:

 Native American Eastern Woodlands Flute in Low E - Daniela Riojas - Quetzalcoatl Flu...

## Why the name Apollofy?

Apollo is the Greek god of the arts, truth, beauty, harmony and reason. How many times will we need the favor of the god Apollo to achieve order and balance in the development of the project! [https://en.wikipedia.org/wiki/Apollo#God\\_of\\_music](https://en.wikipedia.org/wiki/Apollo#God_of_music)

## Epics

1	Account management	User account management, photograph, account data
2	Song genres	Management and creation of new genres to be related to songs
3	Publication, visualization	Section of "My Songs" in the main menu where

	and management of songs	published songs appear and with "I like", publication of songs and visualization of songs
4	Publication, display and management of playlists	Section of "My Playlists" in my main menu with own playlists and playlists with "me gusta", creation, management and visualization of playlists of other users
5	Search in the platform	The user will be able to search songs, playlists and users/artists
6	<b>Optional</b> Web Front-end	Do a very nice front-end

## Sample User stories

User stories are a fine grain specification of the epics.

Be aware that the following user stories must fulfil the limitations explained at Table 1 depending on the user's level

### 1.- Account Management

1.1	User account visualization	Users will be able to view their own information, that is, full name, username, email, profile photo and user level
1.2	User account update	Users will be able to edit their own information but the user level. The only user information that can be empty is the photo
1.3	User level modification	Platform administrators can change the user level. There will be 3 different levels. Table 1 shows the rights for each level. When a user is downgraded she has to select the "I likes", playlists and songs in the playlist she needs to lose in order to fulfil the level limits

### 2.- Song genres

2.1	Create a new genre	Administrators can create new genres. A genre has a name and a description. None of them can be empty
2.2	List all genres	Users can list genres
2.3	Delete a genre	Administrators can delete genres that have no songs belonging to them

### 3.- Publication, visualization and management of songs (tracks)

3.1.1	Song upload (part I)	A user will be able to upload a song together with its information: name, title, genre(s), image. The user will become the owner of the song. Name, title and genres cannot be empty. Genres must be chosen from one existing in the platform.
3.1.2	Song upload (part II)	The song itself (the file) will be uploaded to the Cloudinary platform ( <a href="https://cloudinary.com/">https://cloudinary.com/</a> ). Once you upload the song you need to store the url to access it in the database.
3.2	List "My songs"	The user will be able to access all the songs uploaded within the "My songs" section, in addition, this section will show the songs that the user has given to "I like"
3.3	Give "I like" to a song	The user will be able to "I Like" a song. Also to remove the "I like" to a song
3.4	Edit song	The user will be able to edit the information of a song she owns
3.5	Delete song	The user will be able to delete a song she owns. All "I like" from other users should be deleted
3.6	Play a song	The user will be able to play songs by selecting the desired song, the song will download along with its information

### 4.- Publication, display and management of playlists

4.1	List user's playlists	The user is able to list all her playlists and the playlist she has marked as "I like"
4.2	Playlist creation	The user is able to create a new playlist with name, description, photo, and whether it is private. She may also include a list of songs
4.3	Adding and removing songs to a playlist	The user is able to add and remove songs from her playlists. She only can add songs she owns or she likes ("I like")
4.4	Mark and unmark a playlist as "I like"	The user is able to mark and unmark a playlist as "I like"
4.5	Delete a playlist	Delete a playlist and all the "I like" form other users

## 5.- Search in the platform

5.1	Search by text	Users, songs and playlists that have a coincidence with the text are listed. Playlists are only shown if the user owns them or are set as public
5.2	Search songs by genre	All songs of the given genre are listed
5.3	Search songs by owner	List all the songs of a given artist (or equivalently owner)

## 6.- Optional Front-end

As an extra, your team may create a nice frontend using React

Example UI: <https://the-wave-app.netlify.app/>

## User's level

A user can have 3 different levels depending on her profile. The first level is free and the second and third have a monthly fee.

User Level	# "I like" songs	# uploaded songs	# playlists and songs in them	# "I like" playlist
1 FREE	10	10	2 - 5	0
2 PREMIUM	100	100	5 - 7	6
3 PROFESSIONAL	No limit	No limit	No limit	No limit

**Table 1:** User level and their associated capabilities

# Beyond the conventional... you will reach the glory!

Wouldn't it be great to enrich your platform with fantastic functionalities offered by external APIs?

For example, you could provide lyrics to the songs on the platform. For this, you could integrate with <https://www.musixmatch.com/es> via <https://developer.musixmatch.com/>.

Or you could offer recommendations based on the songs or artists your users listen to. For this, you could integrate with Spotify or Souncloud APIs:

<https://developer.spotify.com/documentation/web-api/reference/#/>

<https://developers.soundcloud.com/>

Let your imagination run wild! We invite you to explore ways to extend the platform with wonderful and attractive features for your users.

## Sprint 1

### Genres CRUD

genre-resource Genre Resource		
GET	/api/genres	getAllGenres
POST	/api/genres	Creates a Genre
PUT	/api/genres	updateGenre
GET	/api/genres/{id}	getGenre
DELETE	/api/genres/{id}	deleteGenre
GET	/api/genres/{id}/tracks	getTracksByGenreId

```
@Entity
@Table(name = "genre")
@Cache(usage = READ_ONLY)
public class Genre implements Serializable {
```

2022-10-24 03:17:05.487 ERROR 11668 --- [ XNIO-1 task-10]  
o.h.i.ExceptionMapperStandardImpl : HHH000346: Error during managed flush [Can't update readonly object]

Our implementation has configured the cache in read mode, for efficiency reasons. That is why the PUT does not allow updating the genre of the song.

You can implement without this efficiency restriction.

## Tracks CRUD

track-resource Track Resource		
GET	/api/tracks	Shows tracks
POST	/api/tracks	Creates a track
PUT	/api/tracks	Updates a track
GET	/api/tracks/{id}	Shows the desired track by id param
DELETE	/api/tracks/{id}	Deletes a track by id

### Steps to start:

- Clone the GitHub Repo generated by GitHub Classroom at your file system (directory X)
- Generate new Spring Boot project with IntelliJ IDEA using Spring Initializr wizard, selecting the location for your new project to the previous location in your file system (directory X)

## Sprint 2

### Playlist CRUD

playlist-resource Playlist Resource		
GET	/api/playlists	Shows playlists
POST	/api/playlists	createPlaylist
PUT	/api/playlists	updatePlaylist
GET	/api/playlists/{id}	getPlaylist
DELETE	/api/playlists/{id}	deletePlaylist

## POST example:

To create a new playlist it is as simple as specifying only the name of the Playlist with the JSON format.

Your backend must work with DTO objects to facilitate the communication of the REST API with the service layer.

POST

/api/playlists createPlaylist

Parameters

Name	Description
<b>playlistRequest</b> * required object (body)	playlistRequest <a href="#">Edit Value</a> <a href="#">Model</a>

```
{  
  "name": "Demo Playlist"  
}
```





## PUT example:

To perform a PUT operation on the Playlist, you can first call a GET by providing the new id generated by the backend.

The image shows a screenshot of an API client interface. At the top, there is a blue button labeled 'GET' followed by the endpoint `/api/playlists/{id}` and the function name `getPlaylist`. Below this is a section titled 'Parameters' with a light gray background. Inside this section, there is a table with two columns: 'Name' and 'Description'. The first row in the table has the name `id` with a red asterisk and the word 'required' next to it. The description for `id` is `integer($int64)`. Below the table, there is a text input field with the value '4'. At the bottom of the interface, there is a large blue button labeled 'Execute'.

Name	Description
<code>id</code> * required	<code>integer(\$int64)</code>

(path)

4

Execute

The GET request will give you the Playlist information. Then, with the object data in JSON format provided by the GET request, you are ready to make a PUT request to update the Playlist with the desired changes.

To do this, it is necessary to make the modifications in the desired attributes. Let's see some examples:

Add a description, to the newly created playlist with Id=4

PUT

/api/playlists updatePlaylist

Parameters

Name	Description
<b>playlistRequest</b> * required	playlistRequest
object (body)	<div>Edit Value Model</div> <pre>{   "id": 4,   "name": "Demo Playlist",   "description": "Description Demo Playlist",   "cover": null,   "thumbnail": null,   "publicAccessible": null,   "followed": false,   "followers": 0,   "owner": {     "id": 3,     "login": "admin",     "firstName": "Administrator",     "lastName": "Administrator",     "email": "admin@localhost",     "imageUrl": "",     "langKey": "en"   },   "tracks": [] }</pre>

## PUT example to Add Tracks:

We are following SoundCloud API design:

<https://developers.soundcloud.com/docs/api/guide#uploading>

## Adding Tracks to a Playlist

Once a playlist has been created, you can continue to add tracks to it by updating the `tracks` property. You can also update the playlist's metadata.

### Curl

```
$ curl -X PUT "https://api.soundcloud.com/playlists/PLAYLIST_ID" \
-H "accept: application/json; charset=utf-8" \
-H "Content-Type: application/json" \
-H "Authorization: OAuth ACCESS_TOKEN" \
-d '{"playlist": {"tracks":[{"id":1}, {"id":2}, {"id":3}]}}'
```

## Example:

Add track with id=1 to the playlist

Note that it is not necessary to enter all the attributes of the track, to add that track to the playlist. It is enough to specify the id of the track.

The Playlist PUT has been designed in such a way that it is not recursive. That is to say, it is not intended to modify the internal attributes of the track, through the PUT of a playlist.

PUT

/api/playlists updatePlaylist

Parameters

Name	Description
<b>playlistRequest</b> * required	playlistRequest
object (body)	<a href="#">Edit Value</a> <a href="#">Model</a>

```
{
  "id": 4,
  "name": "Demo Playlist",
  "description": "Description Demo Playlist",
  "cover": null,
  "thumbnail": null,
  "publicAccessible": null,
  "followed": false,
  "followers": 0,
  "owner": {
    "id": 3,
    "login": "admin",
    "firstName": "Administrator",
    "lastName": "Administrator",
    "email": "admin@localhost",
    "imageUrl": "",
    "langKey": "en"
  },
  "tracks": [{"id": 1}]
}
```

GET example with Id:

GET

/api/playlists/{id} getPlaylist

Parameters

Name	Description
<b>id</b> * required	
integer(\$int64)id	
(path)	<input type="text" value="4"/>

CodeDetails

200

Response body

```
{
  "id": 4,
  "name": "Demo Playlist",
  "description": "Description Demo Playlist",
  "cover": null,
  "thumbnail": null,
  "publicAccessible": null,
  "followed": false,
  "followers": 0,
  "owner": {
    "id": 3,
    "login": "admin",
    "firstName": "Administrator",
    "lastName": "Administrator",
    "email": "admin@localhost",
    "imageUrl": "",
    "langKey": "en"
  },
  "tracks": [
    {
      "id": 1,
      "name": "krishna",
      "url": "http://res.cloudinary.com/zallefy/video/upload/v1665400611/sallefy/songs/krishna.mp3",
      "thumbnail": "http://res.cloudinary.com/zallefy/image/upload/v1665400599/sallefy/thumbnails/image:3122.jpg",
    }
  ]
}
```

## Official Spotify API

The official Spotify API is designed in such a way that it offers two distinct endpoints to clearly separate **changes to simple playlist attributes** from **changes to the playlist's collection of tracks**:

<https://developer.spotify.com/documentation/web-api/reference/#/operations/change-playlist-details>

<https://developer.spotify.com/documentation/web-api/reference/#/operations/add-tracks-to-playlist>

It is certainly a more purist design of the REST API.

You can implement the endpoint to add songs following Spotify's strategy, as an extra functionality.

Delete example with Id:

**DELETE** `/api/playlists/{id}` deletePlaylist

Parameters

Name	Description
<b>id</b> * required	
<code>integer(\$int64)</code>	id
<code>(path)</code>	

After deleting the playlist, if we get the playlist with the Id=4

Request URL

`http://192.168.1.150:8080/api/playlists/4`

Server response

Code	Details
404 <i>Undocumented</i>	Error: Not Found

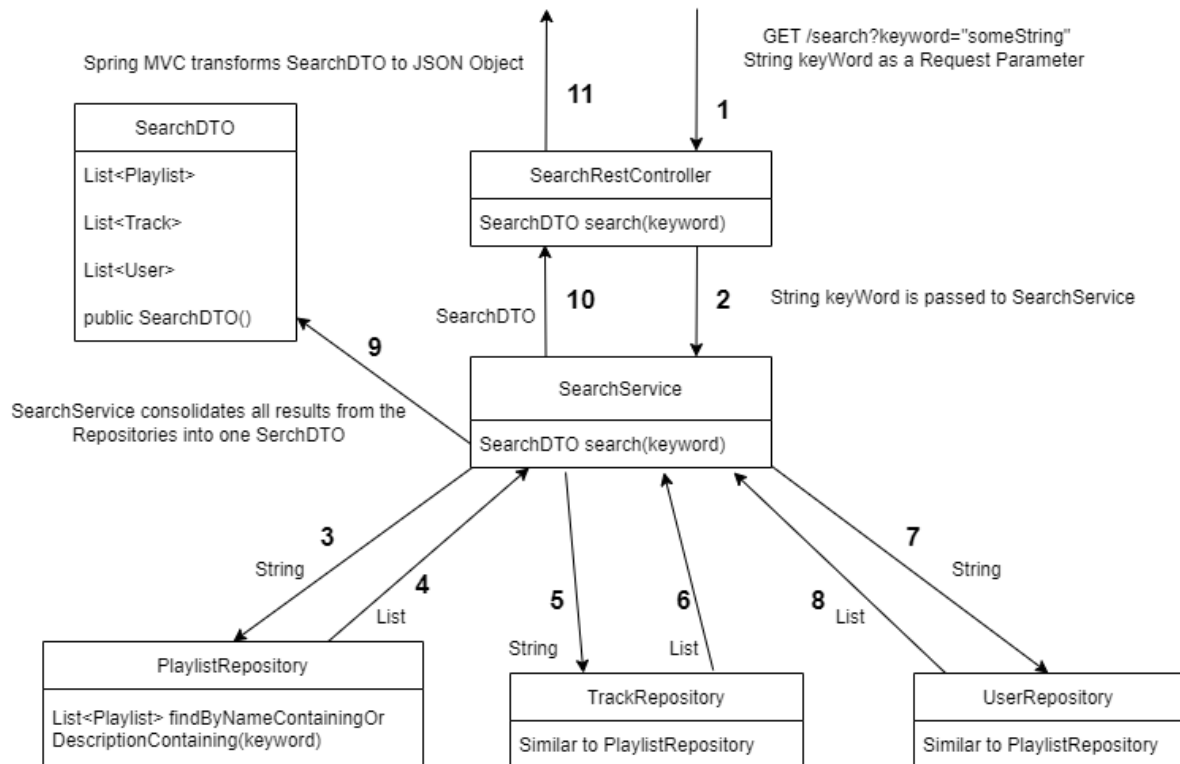
Response body

```
{
  "type": "http://sallefy.eu-west-3.elasticbeanstalk.com/problem/not-found",
  "status": 404,
  "path": "/api/playlists/4",
  "message": "No playlist was found",
  "code": "error.playlist.notFound"
}
```

The response is 404, Not Found as expected.

# Search Users, Tracks and Playlists

## Search Architecture Diagram



[SearchDiagrama.drawio.png](#)

Sample query related to Playlists:

```
@Query("""
    select p from Playlist p
    where upper(p.name) like upper(concat('%', :keyword, '%')) and
    upper(p.description) like upper(concat('%', :keyword, '%')) and
    p.publicAccessible = true""")
List<Playlist> findByKeyword(@Param("keyword") String keyword);
```



# Search Playlists

Next we are going to create a playlist, in order to demonstrate the search functionality.

POST

/api/playlists createPlaylist

Parameters

Name	Description
<b>playlistRequest</b> * required object (body)	playlistRequest <div>Edit Value Model</div> <div><pre>{   "name": "Demo Playlist to Test Search" }</pre></div>

**Code****Details**

201

**Response body**

```
{
  "id": 5,
  "name": "Demo Playlist to Test Search",
  "description": null,
  "cover": null,
  "thumbnail": null,
  "publicAccessible": null,
  "followed": false,
  "followers": 0,
  "owner": {
    "id": 3,
    "login": "admin",
    "firstName": "Administrator",
    "lastName": "Administrator",
    "email": "admin@localhost",
    "imageUrl": "",
    "langKey": "en"
  },
  "tracks": []
}
```

We are going to perform the search by entering a part of the Playlist name.

**search-resource** Search Resource

**GET** `/api/search` Search globally

Search in the whole application. Users, Tracks and Playlists.

Parameters

Name	Description
<b>keyword</b> * required string (query)	keyword

Demo playlist

**Request URL**

`http://192.168.1.150:8080/api/search?keyword=Demo%20playlist`

**Server response**

Code	Details
200	<div><b>Response body</b></div> <pre>{   "playlists": [],   "users": [],   "tracks": [] }</pre>

We notice that the search result is empty. Why did this happen?

The reason why the search result is empty is that the attribute "publicAccessible": true is missing.

PUT

/api/playlists updatePlaylist

Parameters

Name	Description
<b>playlistRequest</b> * required	playlistRequest
object (body)	<div>Edit Value Model</div> <pre>{   "id": 5,   "name": "Demo Playlist to Test Search",   "description": null,   "cover": null,   "thumbnail": null,   "publicAccessible": true,   "followed": false,   "followers": 0,   "owner": {     "id": 3,     "login": "admin",     "firstName": "Administrator",     "lastName": "Administrator",     "email": "admin@localhost",     "imageUrl": "",     "langKey": "en"   },   "tracks": [] }</pre>

Once the attribute has been correctly updated to make the Playlist public, we proceed to perform the search again.

GET

/api/search Search globally

Search in the whole application. Users, Tracks and Playlists.

### Parameters

Name	Description
------	-------------

<b>keyword</b> * required	
---------------------------	--

string	keyword
--------	---------

(query)	
---------	--

Execute

### Request URL

http://192.168.1.150:8080/api/search?keyword=Demo%20playlist

### Server response

#### Code

#### Details

200

#### Response body

```
{
  "playlists": [
    {
      "id": 5,
      "name": "Demo Playlist to Test Search",
      "description": null,
      "cover": null,
      "thumbnail": null,
      "publicAccessible": true,
      "followed": false,
      "followers": 0,
      "owner": {
        "id": 3,
        "login": "admin",
        "firstName": "Administrator",
        "lastName": "Administrator",
        "email": "admin@localhost",
        "imageUrl": "",
        "langKey": "en"
      },
      "tracks": []
    }
  ],
  "users": [],
  "tracks": []
}
```

Now we can see that the search result is indeed correct and includes the expected Playlist

In the next step, we will proceed to search for a song whose name is Krishna.

## Search Tracks

**GET** `/api/tracks/{id}` Shows the desired track by id param

**Parameters**

Name	Description
<b>id</b> * required	
<code>integer(\$int64)</code>	id
<code>(path)</code>	

**Request URL**

`http://192.168.1.150:8080/api/tracks/1`

**Server response**

Code	Details
200	<div><b>Response body</b></div> <pre>{   "id": 1,   "name": "krishna",   "url": "http://res.cloudinary.com/zallefy/video/upload/v1665400611/sallefy/songs/krishna.mp3",   "thumbnail": "http://res.cloudinary.com/zallefy/image/upload/v1665400599/sallefy/thumbnails/image:3122.jpg",   "released": null,   "duration": null,   "liked": true,   "likes": 1,   "plays": 0,   "color": null,   "owner": {     "id": 3,     "login": "admin",     "firstName": "Administrator",     "lastName": "Administrator",     "email": "admin@localhost",     "imageUrl": "",     "langKey": "en"   },   "genres": [     {       "id": 1,       "name": "kirtan"     }   ] }</pre>

# search-resource Search Resource

GET /api/search Search globally

Search in the whole application. Users, Tracks and Playlists.

### Parameters

Name	Description
------	-------------

**keyword** \* required

string  
(query)

keyword

Krishna

Execute

### Request URL

http://192.168.1.150:8080/api/search?keyword=Krishna

### Server response

Code	Details
------	---------

200

### Response body

```
{
  "playlists": [],
  "users": [],
  "tracks": [
    {
      "id": 1,
      "name": "krishna",
      "url": "http://res.cloudinary.com/zallefy/video/upload/v1665400611/sallefy/songs/krishna.mp3",
      "thumbnail": "http://res.cloudinary.com/zallefy/image/upload/v1665400599/sallefy/thumbnails/image:3122.jpg",
      "released": null,
      "duration": null,
      "liked": true,
      "likes": 1,
      "plays": 0,
      "color": null,
      "owner": {
        "id": 3,
        "login": "admin",
        "firstName": "Administrator",
        "lastName": "Administrator",
        "email": "admin@localhost",
        "imageUrl": "",
        "langKey": "en"
      }
    }
  ],
}
```



When implementing searches in the various entities, it is important that you take into account the various attributes of the songs, users and playlists. For example, the name and description.

## Professional API Search Examples:

<https://developers.soundcloud.com/docs/api/explorer/open-api#/search>

<https://developer.spotify.com/documentation/web-api/reference/#/operations/search>

<https://developer.musixmatch.com/documentation/api-reference/track-search>

# Sprint 3

## Users, Roles and JWT Authentication

The users will be created manually, **using the same strategy as the third individual exercise**.

Users will be registered with one of the following roles: FREE, PREMIUM or PROFESSIONAL.

We will also proceed to include API authentication through JWT, in the same way as in the third individual exercise.

Once we have included the concept of users and roles, we will establish the relevant associations between the system entities, e.g. tracks and playlists, and their corresponding users.

The user's capabilities will be taken into account according to the corresponding role, as specified in this [table](#).

## API endpoints: User activity related to tracks, playlists and artists

Once users and roles have been successfully implemented in your system, the next step will focus on developing the following API endpoints:

GET	/api/tracks/{id}/like	Check if current user liked a track
PUT	/api/tracks/{id}/like	Likes the track by id

GET	/api/playlists/{id}/follow	Check if current user follows the playlist
PUT	/api/playlists/{id}/follow	Follows the playlist by id

GET	/api/users/{login}/follow	Check if following
PUT	/api/users/{login}/follow	Follow the desired user

GET	/api/me/followers	Shows all the current user followers
GET	/api/me/followings	Shows all the following users by the current user

GET	/api/me/playlists	Shows own playlists
GET	/api/me/playlists/following	Shows all the following playlists by the current user

GET	/api/me/tracks	Shows own tracks
GET	/api/me/tracks/liked	Shows al the liked tracks by the current user

## Important advice for third and fourth Sprints

If you implement the likes, follows, etc. with Association Classes, saving the attribute of the date on which the event occurs, you would be able to offer very interesting endpoints with high added value from a statistical point of view. For example, the top 5 songs that have received the most likes in the last week.

More information: [Association classes in UML diagrams](#)

The Association Class will be a JPA Entity with ManyToOne to the associated entities. For instance, LikeTrack will have ManyToOne to a User and ManyToOne to the liked Track.

Super useful resource to perform queries on Spring Data Repositories:  
[Ultimate Guide: Derived Queries with Spring Data JPA](#)

## Association Class JPA LikeTrack

```
@Entity
public class LikeTrack {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "liked")
    private Boolean liked;

    //https://thorben-janssen.com/persist-creation-update-timestamps-hibernate/
    @UpdateTimestamp
    private ZonedDateTime date;

    @ManyToOne(optional = false)
    @NotNull
    @JsonIgnoreProperties("likeTracks")
    private User user;

    @ManyToOne(optional = false)
    @NotNull
    @JsonIgnoreProperties("likeTracks")
    private Track track;
```

Notes: <https://thorben-janssen.com/persist-creation-update-timestamps-hibernate/>

Professional strategy to implement Association Class with JPA:

[The best way to map a many-to-many association with extra columns when using JPA and Hibernate - Vlad Mihalcea](#)

Thanks [roure@tecnocampus.cat](mailto:roure@tecnocampus.cat) for the tip! 😊

## Professional API Like and Follow Examples:

<https://developer.spotify.com/documentation/web-api/reference/#/operations/save-tracks-user>

<https://developer.spotify.com/documentation/web-api/reference/#/operations/follow-artists-users>

<https://developer.spotify.com/documentation/web-api/reference/#/operations/follow-playlist>

<https://developer.spotify.com/documentation/web-api/reference/#/operations/check-current-user-follows>

<https://developers.soundcloud.com/docs/api/explorer/open-api#likes>

## [Postman Collections: Organize API Development and Testing](#)

For Sprint 4, **it will be imperative to organize all your API calls using a Postman collection**. You will also write tests to verify that your backend REST API is working properly.

<https://apitransform.com/how-to-share-collection-in-postman/>

We provide you with the following resources to implement this task successfully:

[pm.test | Test examples in Postman | Postman API Network](#)

[Postman Collections: Organize API Development and Testing](#)

[Intro to writing tests - with examples | Postman Team Collections](#)

[Test examples in Postman | Postman API Network](#)

[Generate Spotify Playlists using a Postman Collection](#)

[Spotify | Music Discovery with Postman](#)

<https://github.com/cinexin/spotify-api-postman>

# Sprint 4

Thanks to the endpoints that you have implemented in the third sprint, the music platform will be able to collect information about track likes, playlist and user follows, etc.

Considering all this rich information, you will implement endpoints that offer great added value from a statistical point of view.

## Top Likes and Follow Endpoint

*GET /top/{type}*

Type valid values: users, playlists, tracks or genres \*

### **Request params:**

- count: from 0 to 100 (mandatory) (use javax.validation)
- start date (optional) \*
- final date (optional) \*

Request example: ***GET /top/tracks?count=3***

Response example:

```
[
  { "id" : 67 , "name" : "Vuela con el viento", "artist": "Ayla Schafer", "likes" : 85459302 },
  { "id" : 543 , "name" : "Travelling One", "artist": "Susie Ro", "likes" : 45459302 },
  { "id" : 236 , "name" : "Temple of Silence", "artist": "Deuter", "likes" : 25459302 }
]
```

*\* Note: To count the likes related to a genre, we will add up the number of likes that all the songs of a certain genre have received.*

*\* <https://www.baeldung.com/spring-request-param#1-using-java-8-optional>*

*Loading all the database information into memory and filtering with manual Java code is unacceptable, due to performance reasons. You should use some efficient approach such as, for example, the power of JPQL.*

**Resources:**

[JPQL - How to Define Queries in JPA and Hibernate](#)

Para limitar el Count: [What is the LIMIT clause alternative in JPQL? - Stack Overflow](#)

[Limiting Query Results with JPA and Spring Data JPA | Baeldung](#)

## How to create and use the DTO class

### DTO Projections

When using a DTO projection, you tell your persistence provider to map each record of your query result to an unmanaged object. As shown in a [previous article](#), this performs much better than entities if you don't need to change the selected data. And, in contrast to scalar value projections, they are also very easy to use. This is because the DTO objects are named and strongly typed.

### JPA's DTOs

The goal of a DTO class is to provide an efficient and strongly typed representation of the data returned by your query. To achieve that, a DTO class typically only defines a set of attributes, getter and setter methods for each of them, and a constructor that sets all attributes.

```
1      public class AuthorSummaryDTO {
2
3          private String firstName;
4          private String lastName;
5
6          public AuthorSummaryDTO(String firstName, String lastName) {
7              this.firstName = firstName;
8              this.lastName = lastName;
9          }
10
11         public String getFirstName() {
12             return firstName;
13         }
14         public void setFirstName(String firstName) {
15             this.firstName = firstName;
16         }
17         public String getLastName() {
18             return lastName;
19         }
20         public void setLastName(String lastName) {
21             this.lastName = lastName;
22         }
23     }
```

To use this class as a projection with plain JPA, you need to use a [constructor expression in your query](#). It describes a call of the constructor. It starts with the keyword *new*, followed by the DTO class's fully-qualified class name and a list of constructor parameters.



```
1  @Repository
2  public interface AuthorRepository extends CrudRepository<Author, Long>
3  {
4
5      @Query("SELECT new
6  com.thorben.janssen.spring.jpa.projections.dto.AuthorSummaryDTO(a.firstName, a.lastName) FROM Author a WHERE a.firstName = :firstName")
7      List<AuthorSummaryDTO> findByFirstName(String firstName);
8  }
```

As you can see in the code snippet, you can use this approach in [Spring Data JPA's @Query annotation](#). Your persistence provider then executes a query that selects the columns mapped by the referenced entity attributes and executes the described constructor call.

*From: [Spring Data JPA: Query Projections](#) and  
[Why, When and How to Use DTO Projections with JPA and Hibernate](#)*

[Project Lombok](#) very nice for DTOs ;-)

<https://projectlombok.org/setup/maven>

## Add collaborators to playlists

Extend the Backend API to allow defining collaborators that will be able to add songs to a playlist. The owner of the playlist must have the PREMIUM or PROFESSIONAL role.

A collaborator will be a regular user with special permission to collaborate on a playlist. The approach may be similar to the second individual exercise. You can add to the *Playlist entity* an attribute *Set<User> collaborators* with the association type *ManyToMany*.

The system will also offer an endpoint to remove collaborating users from a playlist.

To be able to add a track to a playlist, the user must be the owner of the playlist or one of the collaborators.

Choose the most suitable API endpoint design, inspired by professional APIs like Spotify, Google, GitHub, etc.

[Web API design best practices - Azure Architecture Center | Microsoft Learn](#)

[Google API Design Guide](#)

[HTTP API design guide](#)

[Web API | Spotify for Developers](#)

### Sample project:

[REST API Notes TecnoCampus-master.zip](#)

Simple project to showcase similar behavior: Add editors to Notes

This project provides a guide to implement the feature “Add collaborators to playlists”.

This is a minimalist implementation that does not use DTOs.

### Bonus feature:

Improve the design and implementation so that the system records which user has added a track to a playlist, as well as the exact time at which the event occurred. For this, it is recommended to use an Association Class, as we have done with the implementation of Track Likes.

[How to Make a Collaborative Playlist on Spotify](#)

# Bring lyrics to your application!

To extend your Backend with advanced functionalities, beyond a basic CRUD, it is proposed to interact with the [Musixmatch Developer](#) API.

There are several ways to do this, for example, when searching for a track, your API may include the lyrics of the track in a new “*lyrics*” attribute.

Integrate the musixmatch service with your website or application

In the most common scenario you only need to implement two **API** calls:

The first call is to match your catalog to ours using the search function (details on the [track.search](#) page) and the second is to get the lyrics (see the page [track.lyrics.get](#)). That's it!

<https://developer.musixmatch.com/documentation/api-reference/track-search>

<https://developer.musixmatch.com/documentation/api-reference/track-lyrics-get>

You can also explore this API call [Matcher.lyrics.get](#)

This resource is recommended to add the query parameters to your request 😊

[https://www.amitph.com/spring-webclient-request-parameters/#query\\_parameters](https://www.amitph.com/spring-webclient-request-parameters/#query_parameters)

## Musixmatch Developer API Architecture Diagram

📄 Diagrama api externa.drawio.png

There are other advanced functionalities that can be explored, to enrich your backend.

Get the mood list (and raw value that generated it) of a lyric

<https://developer.musixmatch.com/documentation/api-reference/track-lyrics-mood-get>

## Other Music APIs you can explore

<https://rapidapi.com/blog/top-free-music-data-apis/>

<https://www.programmableweb.com/news/7-top-apis-lyrics/brief/2019/10/27>

## Spring Reference Documentation

### [34. Calling REST Services with WebClient](#)

#### Webclient Example

<https://github.com/alfredorueda/demo-webclient>

<https://github.com/alfredorueda/demo-webclient/blob/master/src/main/java/edu/webclient/demowebclient/RunPokeApi.java>