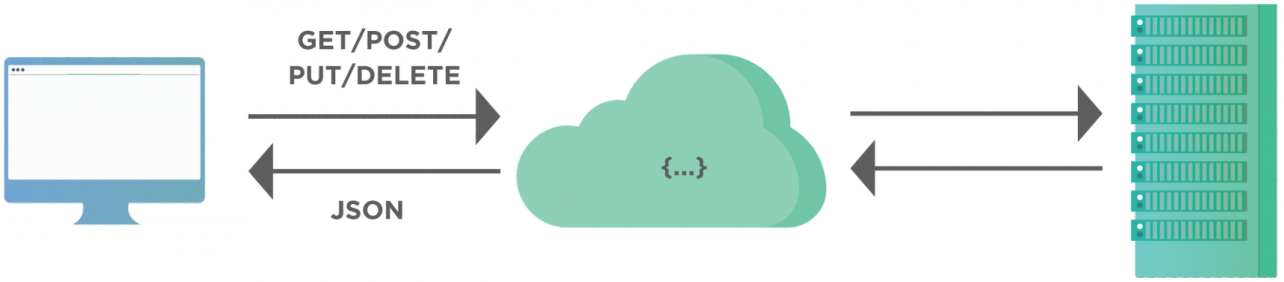
Fachhochschule
Nordwestschweiz

REST API for Retrieving (Lists of) Similar Songs



Source: TOOLSQA

Authors: **Bruno Baptista Kreiner, Katarina Fatur**

Mentor: **Fernando Benites**

Data Science BSc Study

Module: **Module Web Data Acquisition (*Web Datenbeschaffung*)**

Autumn Semester 2022

Motivation

Why another Spotify API?

Spotify's related tracks algorithm is user-centered. It recommends songs based on the user's listening activities (collaborative-filtering, content-based recommendation). The suggested tracks match our taste, but might not be acoustically similar to the initial track at all. This is not useful when all we really want is to get a list of tracks whose sound qualities are similar or the same.

We wanted to get a track (list) that matches the chosen (acoustic) profile. This is why we decided to make an API that is track-centered. It allows us to obtain a track or create simple playlists of songs that are related only by a certain feature (e.g. their key), or it recommends more advanced playlists, where similarity is determined by multiple features.

Why REST?

The main (non-technical) reason for our interest in REST over GraphQL was that it is still the preferred API architecture in the industry (as per State of API 2022 Report), reliably used by some very important players in the field like Google, NASA, Wikipedia, Spotify, Twitter and many others.

While we sincerely appreciate the coolness of GraphQL, in cases where either of the technologies can be implemented with success, we follow the motto "REST first, GraphQL second". We stipulate that the current prevalence of the REST architecture might prove a valuable skill to our future employer. The technical reasons for our choice are discussed in the chapter *REST vs GraphQL - why REST?*.

Introduction

We undertook this project as a partial fulfillment of the requirements to complete the WDB Module (web data acquisition). The task was to make a cool API service that allows CRUD. We upgraded the options provided by the basic HTTPS verbs (GET, PUT, DELETE, PATCH) with URL parameters. This allows us to implement sorting, filtering and a recommendation matrix. We implement an API to retrieve track(s) information and recommendations for single tracks. The recommendation algorithm can be later refined to implement a weighted recommendation based on various track attributes.

Project

The project can be run inside a Docker or by creating a virtual environment manually (from the requirements.txt file). A concise summary of the project is provided below, but for a more comprehensive overview refer to the README.md.

Architecture

The REST API is built on top of Flask and uses Flask-RESTful for creating REST endpoints. The communication with the database is done via Flask-SQLAlchemy ORM tool in order to avoid writing SQL queries manually. To communicate with the server you can use the provided Python client. It uses the requests module for generating HTTP requests and implements all the endpoints the server provides.

API Description

Our API is divided into 3 endpoints for accessing individual tracks and list of tracks. The first endpoint implements CRUD functions to create, retrieve, update and delete a single track with the track model attribute "track_id". The second endpoint is to retrieve a paginated sorted list of all tracks in the

database filtered by field. The third endpoint allows you to request song recommendations based on the provided track.

TrackList endpoint

/api/tracks/: Get a list of songs based on some criteria
GET /api/tracks/ - Return the requested list of Tracks

Track endpoint

/api/track/<int:track_id>
GET /api/track/<int:track_id> - Retrieve single Track by id
PUT /api/track/<int:track_id> - Create new Track record
PATCH /api/track/<int:track_id> - Update existing track by id
DELETE /api/track/<int:track_id> - Delete existing track by id

Recommender Endpoint

/api/recommendation/<int:track_id>/
GET /api/recommendation/<int:track_id>/ - Get recommended song(s) based on a track id

More detailed information and examples are provided in the README.md.

Storage

A table in an sqlite database, which is created by the script import_data.py, is called track_model and contains the following columns:

Column name	Type	Description
id	INTEGER	Primary index of records. This field is auto-incremented by the database.
track	TEXT	Name of the track.
artist	TEXT	Name of the artist performing the track.
danceability	REAL	Danceability score assigned by Spotify.
key	INTEGER	Key the track is performed in. E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on.
instrumentalness	REAL	Instrumentalness score assigned by Spotify.
tempo	REAL	Tempo of the track.
duration_ms	INTEGER	Track duration in milliseconds.
popularity	INTEGER	Track popularity on Spotify.
decade	TEXT	Decade in which the track was created.

REST vs GraphQL - why REST?

GraphQL Key Features

- A Query Language for APIs, developed by Facebook in 2012. Works best on mobile-responsive platforms.
- Objects are represented by nodes, edges represent connections between objects. A client has the power to specify the (exact type of the) data required from the API. This results in improved connectivity and enables the client to retrieve what it needs in a single request, whether this is a portion of data from one resource, or several pieces of data from multiple resources. Client requests data with queries.

- Only supports JSON format, error identification is difficult, initial set up takes more time.

REST Key Features

- A Software Architectural Style introduced by Roy Fielding in 2000.
- It defines a set of architectural constraints of a distributed system. Since there are no standardized guidelines, implementation is left to the individual. Request object is defined on the server (and not by client like in GraphQL).
- Client sends an HTTP request and receives an HTTP response.
- Supports different data formats.
- REST is deployed over a set of endpoints, each request is sent to a distinct endpoint.
- The server does not monitor the client, who uses the API, REST is stateless.
- HTTP Status Codes allow REST APIs to identify errors easily.
- An important disadvantage is the over- / under-fetching of data.

The Reasoning Behind our Choice

The client does not need to control the type and amount of data it needs, no bandwidth concerns, no changes in the amount or type of the required data. We chose REST, not because it would not be possible to implement the project in GraphQL, but simply because there was no need to. REST API is very easy to build and adapt. Hence, the decision was based not only on the reasons in favor of REST, but also on the lack of the reasons for GraphQL.

Just one data source and no nested data structures. We query just one table and our data structure is flat, so the only advantage of GraphQL over REST in our case - the fact that GraphQL handles nested data structures better - is not applicable to our situation. However, if there were a lot of nested queries, we might want to consider GraphQL.

No need for data fetching control. Our user-specific needs are clear and unchanging, so we could satisfy our data searching needs by implementing sorting and filtering. We do not encounter the problem of over- and under-fetching. If this was a concern, we could introduce query parameters that control how many fields are returned.

In short: REST is the industry standard and can do much of what GraphQL does. You can specify only the information you need and receive exactly that by passing the name of the fields you want to use in the URL. Since implementing the project in GraphQL would not introduce any benefits to us, it might actually be an overkill for our small application, we implemented REST.

Other Sources (not Hyperlinked in the Text)

API Design Principles
CRUD API Design & CRUD Recommendations
REST Resources
Flask-SQLAlchemy
How to Query Tables and Paginate Data in Fflask SQLAlchemy
GraphQL vs REST: 4 Critical Differences
GraphQL vs. REST APIs: Why you shouldn't use GraphQL
GraphQL is the better REST