Web Engineering
Songregator

Dan Plămădeală (s3436624)
Alexander Fyodorov (s3274349)

October 25, 2019

1 Introduction

The purpose of the project is to develop a Web application, the main purpose of which is to provide users with information related to songs and their affiliated artists. The data required is taken from the Million Song Dataset prepared by CORGIS Dataset Project.

Users should be provided with the following minimum functionality:

1. Obtain information about all artists in the dataset.
   Optional filtering:
   (a) By artist name
   (b) By genre
2. Obtain all information about a specific song (identified by unique ID).
3. Obtain a list of songs by a specific artist and/or in a specific year.
4. Obtain a list of songs by artists in a specific genre.
5. Obtain a list of artists ranked by their popularity, with the possibility to subset this order, e.g. the top 50 artists.
6. Obtain a list of songs ranked by popularity.
7. Obtain descriptive statistics (mean, median, standard deviation) for the popularity of the songs for a particular artist.
   Optional filtering:
   (a) By year

2 How to run

Backend

1. Install dependencies:
   ```
   pip install -r requirements.txt
   ```
   where **requirements.txt** is the file located in the root of the project directory.
2. Run the following command from the root of the project (where file **manage.py** is located):
   ```
   python manage.py runserver
   ```
   It will run a local server located at http://127.0.0.1:8000/ (this address will be printed to the console as well). The port may be different if 8000 is busy.

Frontend

1. Navigate to the **frontend** folder, located in the root of the project directory, and run:
   `npm install`
   This will install all modules listed as dependencies in **package.json**.

2. In the same directory, run:
   `npm start`
   It will run a local server located at http://127.0.0.1:3000/ (this address will be printed to the console as well). The port may be different if 3000 is busy.

3 API Design

For every API endpoint, the following **header field** holds:

Path parameter	Value
Content-Type	`application\json` || `text\csv`

The default representation of resources is JSON.

A parameter is optional if it is typeset italic, for example the *Content-Type* parameter described earlier.

Used response codes:
200 - **OK**; this means that the request was successfully fulfilled.
400 - **Bad Request**; this means that there is a mistake in the syntax of the request.
403 - **Forbidden**; this means that the given Authorization header is not a valid one.
404 - **Not Found**; this means that the given resource was not found.
405 - **Method Not Allowed**; using an unsupported method, for example POST where PUT is expected.

Response content types:

1. JSON **(default)**
2. CSV

Requests and responses:

1. All the artists in the dataset.

   Objective:
   Get all artists available in the dataset. Optionally, the user may ask for the artists to be filtered by the name and/or by their genre.
   The user may also request the results to be ordered ranked by their popularity and limit the size of the result to a given size.
   Calling this endpoint with no parameters yields a response containing all the artists in the dataset.

   Endpoint:

   ```
   GET /artists/?name={name}
               &genre={genre}
               &ordered={ordered}
               &subset={size}
   ```

Query parameters:

Query parameter	Value
name <string>	The name of the artist.
genre <string>	The genre of the artist.
ordered <boolean>	Whether the results should be ordered.
size <integer>	The size of the response.

HTTP status codes:

(a) 200 (SUCCESS)

Response:

```
{
    "count": integer,
    "next": URL,
    "previous": URL,
    "results": [
        {
            "artist_familiarity": float,
            "artist_hotttnesss": float,
            "artist_id": string,
            "artist_latitude": float,
            "artist_location": integer,
            "artist_longitude": float,
            "artist_name": string,
            "artist_similar": float,
            "artist_terms": string,
            "artist_terms_freq": float
        },
        ...
    ]
}
```

2. Information about a song.

Objective:
Get all the available information about a song given a unique ID.
Calling this endpoint with no path parameter yields a response containing all the songs in the dataset (as described in the next point).

Endpoint:

```
GET /songs/{song_id}/
```

Path parameters:

Path parameter	Value
song_id <string>	The unique ID of the song.

HTTP status codes:

(a) 200 (SUCCESS)

(b) 404 (NOT FOUND) - if a song with the specified **song_id** does not exist.

Response:

```
{
    "song_artist_mbtags": float,
    "song_artist_mbtags_count": float,
    "song_bars_confidence": float,
    "song_bars_start": float,
    "song_beats_confidence": float,
    "song_beats_start": float,
    "song_duration": float,
    "song_end_of_fade_in": float,
    "song_hotttnesss": float,
    "song_id": string,
    "song_key": float,
    "song_key_confidence": float,
    "song_loudness": float,
    "song_mode": integer,
    "song_mode_confidence": float,
    "song_start_of_fade_out": float,
    "song_tatums_confidence": float,
    "song_tatums_start": float,
    "song_tempo": float,
    "song_time_signature": float,
    "song_time_signature_confidence": float,
    "song_title": integer,
    "song_year": integer,
    "artist_terms": string
}
```

3. & 4. All songs in the dataset.

Objective:
Get all the songs in the dataset with optional parameters: artist, year of release and genre of the song.
The user may also request the results to be ordered ranked by their popularity and limit the size of the result to a given size.
Calling this endpoint with no query parameters yields a response containing all the songs in the dataset.

Endpoint:

```
GET /songs/?artist={artist}
          &year={year}
          &genre={genre}
          &ordered={ordered}
          &subset={size}
```

Query parameters:

Query parameter	Value
artist <string>	The artist of the song.
year <integer>	The year of release of the song.
genre <string>	The genre of the song.
ordered <boolean>	Whether the results should be ordered.
size <integer>	The size of the response.

HTTP status codes:

(a) 200 (SUCCESS)

Response:

```
{
    "count": integer,
    "next": URL,
    "previous": URL,
    "results": [
        {
            "artist_mbtags": float,
            "artist_mbtags_count": float,
            "bars_confidence": float,
            "bars_start": float,
            "beats_confidence": float,
            "beats_start": float,
            "duration": float,
            "end_of_fade_in": float,
            "hotttnesss": float,
            "id": string,
            "key": float,
            "key_confidence": float,
            "loudness": float,
            "mode": integer,
            "mode_confidence": float,
            "start_of_fade_out": float,
            "tatums_confidence": float,
            "tatums_start": float,
            "tempo": float,
            "time_signature": float,
            "time_signature_confidence": float,
            "title": integer,
            "year": integer
        },
        ...
    ]
}
```

5. Descriptive statistics of an artist.

Objective:
Get descriptive statistics (mean, median, standard deviation) for the popularity of the songs for a particular artist with an optional filter by year.
Calling this endpoint with no query parameters yields an empty response.

Endpoint:

```
GET /popularity/?artist={artist}&year={year}
```

Query parameters:

Query parameter	Value
artist <string>	Artist name.
year <integer>	Song release year.

HTTP status codes:

(a) 200 (SUCCESS)

(b) 404 (NOT FOUND) - if **artist** is not specified or does not exist.

Response:

```
{
    "mean": float,
    "median": float,
    "std": float,
    "links": [
        {
            "artist": "GET /artists/{string}"
        }
    ]
}
```

6. Delete all the songs of a given artist.

Objective:
Deletes all the entries (songs) in the dataset by the given artist.
Calling this endpoint with no path parameter yields a list of all artists without deletion.

Endpoint:

```
DELETE /artists/{artist}/
```

Path parameters:

Path parameter	Value
artist <string>	Artist name.

HTTP status codes:

(a) 200 (SUCCESS)

(b) 404 (NOT FOUND) - if an artist with specified name does not exist.

Response (deleted songs):

```
[
    {
```

```
        "song_artist_mbtags": float,
        "song_artist_mbtags_count": float,
        "song_bars_confidence": float,
        "song_bars_start": float,
        "song_beats_confidence": float,
        "song_beats_start": float,
        "song_duration": float,
        "song_end_of_fade_in": float,
        "song_hotttnesss": float,
        "song_id": string,
        "song_key": float,
        "song_key_confidence": float,
        "song_loudness": float,
        "song_mode": integer,
        "song_mode_confidence": float,
        "song_start_of_fade_out": float,
        "song_tatums_confidence": float,
        "song_tatums_start": float,
        "song_tempo": float,
        "song_time_signature": float,
        "song_time_signature_confidence": float,
        "song_title": integer,
        "song_year": integer,
        "artist_terms": string
    },
```

7. Update the location of the artist.

   Objective:
   Updates the location of the artist given the logitude and latitude coordinates.
   Calling this endpoint with no or one query parameter yields an information about artist's name, longitude and latitude without any changes.

   Endpoint:

```
PATCH /artists/{artist}/
           &longitude={longitude}
           &latitude={latitude}
```

   Path parameters:

Path parameter	Value
artist <string>	Artist name.

   Query parameters:

Query parameter	Value
longitude <float>	Longitude coordinate.
latitude <float>	Latitude coordinate.

   HTTP status codes:

   (a) 200 (SUCCESS)

(b) 405 (METHOD NOT ALLOWED) - if artist is not specified.

Response:

```
{
    "artist_name": string,
    "artist_longitude": float,
    "artist_latitude": float,
    "links": [
        {
            "artist": "GET /artists/{string}"
        }
    ]
}
```

8. **BONUS:** Add an artist to the database.

Objective:
Add a new artist to the database with information specified as query parameters.
Calling this endpoint with some missing information parameters will "default" them (integer - to 0, float - to 0.0, string - to "").
artist_name field is required.

Endpoint:

```
POST /artists/?familiarity={familiarity}
            &hotttnesss={hotttnesss}
            &id={id}
            &latitude={latitude}
            &location={location}
            &longitude={longitude}
            &name={name}
            &similar={similar}
            &terms={terms}
            &terms_freq={terms_freq}
```

Query parameters:

Path parameter	Value
familiarity <float>	Artist familiarity.
hotttnesss <float>	Artist hotttnesss.
id <string>	Artist id.
latitude <float>	Artist latitude.
location <int>	Artist location.
longitude <float>	Artist longitude.
name <string>	Artist name.
similar <float>	Artist similar.
terms <string>	Artist terms.
terms <float>	Artist terms frequency.

HTTP status codes:

(a) 200 (SUCCESS)

(b) 403 (FORBIDDEN) - if artist's name is not specified or such artist already exists.

Response:

```
{
    "artist_familiarity": float,
    "artist_hotttnesss": float,
    "artist_id": string,
    "artist_latitude": float,
    "artist_location": int,
    "artist_longitude": float,
    "artist_name": string,
    "artist_similar": float,
    "artist_terms": string,
    "artist_terms_freq": float
}
```

4 Architecture

Backend

Django is selected as the backend framework. Reasons:

1. **It is flexible**. Since the app's functionality may be expanded, it is necessary to minimize any possible problems as much as possible.
2. **Database queries**. The database can be accessed right in the Python code, no SQL queries are used. It leads to an easier database access and an enhanced security (for example, by eliminating SQL injections).
3. **DRY**. By using Django, Web application and API backend are in the same codebase, making the development process easier and more reliable.
4. **Great support**. Django has a good documentation and supportive community.

SQLite was selected as a database management systems due to its compactness, portability and simplicity.

All the interaction with the backend is happening with a help of Django REST framework - a powerful and flexible toolkit that provides an extensive functionality out-of-the-box. It provides a great Serialization feature that translates complex datasets (e.g. queryset) to native Python datatypes, making it easy to render the data into JSON and XML. It is capable to generate a highly customizable view, providing a rich CRUD API to manipulate data with a minimum code. It allows to parse HTTP query and path parameters in a fast and convenient way.

Also, the way Django parses query parameters provides some flexibility in URL construction. For example, `GET /artists/?id=ABCDE` and `GET /artists/id/ABCDE/` are the same requests from the perspective of Django.

Pagination: Django has a pagination support built-in. A page contains 100 elements.
To obtain a page **n**, append the `offset=N` query parameter, where $N = 100 * (n - 1)$.

Linking implemented:

1. `PATCH /artists/{artist}/<params>` - response contains a request to retrieve an information about the artist modified: `GET /artists/<artist_name>`

2. `GET /popularity?artist=<artist_name>` - response contains a request to retrieve an information about the artist: `GET /artists/<artist_name>`

Frontend

React is selected as the frontend framework.

CREATE is not implemented yet.

Alexander

1. Backend
   (a) Implemented Views (request handling).
2. Frontend
   (a) Added pagination for Song list.
   (b) Added "Previous"/"Next" buttons for pagination above table for Song and Artist lists.
3. Documentation
   (a) Wrote **Introduction**.
   (b) Extended **API Design** and described **8**th (bonus) request.
   (c) Wrote **Architecture**.

Dan

1. Backend
   (a) Implemented models.
   (b) Loaded dataset into database.
2. Frontend
   (a) Wrote React client.
3. Documentation
   (a) Wrote **API Design**.