Project Report

Group 23

October 2019

1 Introduction

The presented web application was designed through the course of Web Engineering. It provides users with a simple functionality over the Million Song Dataset by the CORGIS dataset project (https://think.cs.vt.edu/corgis/csv/music/).

Through this project we designed a RESTful API, which we implemented in the back-end of our web application and we developed a simple front-end user interface for the application.

We tried our web-app to support the basic CRUD (Create, read, update and delete) functions and to supply users with some features based on the dataset.

2 Instructions on the delivered software

- **RESTful API Documentation**
  For the viewing of our API's Documentation, you need to generate it from our .js files . That can be done by extracting the apidocgroup23.zip that is provided, opening the terminal into the main folder ('RESTAPI') and typing the command " apidoc -i myapp/ -o apidoc/ ". After that, the 'index.html 'file inside the apidoc file is the application of our documentation.

- **Back-end**
  For running the back-end, python's `dataclasses` functionality is required, which means you need Python 3.7 or later. By default it runs on localhost, with port 5000 (`http://127.0.0.1:5000`).

- **Front-end**
  The front-end is a plain `html` file, with some local and remote javascript and css sources (the remote sources require an internet connection). By default it loads the first 20 entries from the database, and retrieves their `song_` prefixed attributes, such as `song_beats_start` and `song_duration`. Not all of these are displayed. The songs' included links are used to give access to related resources available in the API, such as the artist that created the song and the data about the song itself. The API also

provides links with associated methods, such as `DELETE`, which can be used for creating a `Delete this song`-button, for instance.

3 Architecture

The whole design was created following the REST principles. Both back and front end implements those principles in an accurate way.

- **RESTful API Documentation**
  For the documentation of our API we used apiDoc, a Node.js module, that creates a Documentation HTML-Page based on API-Descriptions from our source code in Javascript.
  We chose apiDoc because of its ease of use, as it comes up with a straightforward environment which can be used with any programming language, thus it helped us to build the documentation in a flexible way. The output of this module helps the user understand the principles of the web app, as it delivers a very comprehensible presentation.

- **Back-end**
  For the creation of back-end we used Flask, a web framework written in Python. The supported database functions under a SQLite database which we created from the provide music.csv file, using a small Python script.
  The backend uses classes `Artist`, `Release` and `Song` to represent their respective values in a database entry. Note however that the classes `Release` and `Song` have uplinks to their parent resources (A song is a 'child' of a release). These values are used for creating the links to related resources when creating API responses.
  We were somewhat reluctant to merge these links into our data, as it felt like we were 'polluting' the data with information that, in many cases, would only be relevant as long as the users would be communicating with our API. It also meant that, to support users `POST`-ing a song they just had `GET` from our endpoints, would not be accepted, as the added `links` field was undesired. This was fixed relatively easily by filtering out any fields that were not required for a `POST`. The API provides feedback using a simple `Exception` class, which allows us to communicate about wrongly formatted JSON payloads, incorrect parameter values, or that a resource already existed when a user tries to `POST`. These explain what went wrong, and in certain cases provide a list of accepted values, or the exception thrown by the JSON converter, which can help users with debugging. The use of Flask was chosen because of its simple environment which helps users with basic knowledge of Python to get started, so it wasn't required for us to learn a new programming language. Python provides its users with a variety of libraries that can be helpful in several cases, without the need of creating new functions. SQLite is a part of Python Library, so was very convenient to convert .csv file to a SQLite database

and implement it to our application using Python, as it provides a clear interface for SQLite databases. We were familiar with SQLite, and have known it as a database system that has caused little problems in previous projects. Flask was less familiar to us, but the size of the community around it, in combination with the lifetime of the project, gave us confidence that it would be a mature framework, that has a large infrastructure of supporting modules, informations sources and stack-overflow answers around it, which is what we needed for this project, as we dove in with little experience.

- **Front-end**
  Front-end of our app was created with Vue.js model-view framework and provides the basic user interface for the web application.
  The advantages of Vue that led us chose it for our front-end page include its small size, simple structure and usage, as the environment doesn't require advanced knowledge, therefore it helped us in an instant setup of the page without getting any trouble. Also, use of Vue was presented in the tutorials that happened during the classes, so we were provided with some basic knowledge of it. In the front-end we also include a button with every song entry, which allows users to retrieve some additional data about the artist, in the form of a single line description. These are retrieved using the Wikipedia API found at `https://en.wikipedia.org/w/api.php`. We require a button click before retrieving this, as to not overload the Wikipedia API. The retrieved data includes a link to the Wikipedia page, which is displayed along the summary.

4 Potential issues

Not all endpoints have the `CUD` functionality implemented. Under these: `/artists`, `/artists/:artist_id/songs`. Also, there is no endpoints for Releases, nor stand-alone, nor as a sub-source of `/artists/:artist_id`. Also, against some conventions, we have underscores in our set of accepted parameter values, namely for the columns. We wanted the internal workings of our API to mirror the format of the CORGIS Music data-set, which uses underscores over dashes (hyphens) to join multiple words into a single column name. We have opted to not accept hyphenated versions of these column names, as they would usually be provided by a program, a setting where the human-readability benefit of hyphens over underscores is low. This might not be an issue, but it is also not expected by experienced Flask users: we barely use the flask SQLite 3 integration, as this was unfamiliar to us, and we had some familiarity with using SQLite 3 disjoint from Flask. Right now there is no pagination implemented for endpoints. Endpoints which provide access to a large collection, such as `/artists` and `/songs` have a `count` parameter, which limits the amount of entries returned.