Project report
Web Engineering 2019
Group 14

E. Waterink (s3417611), T. de Vries (s2367610), I. Oralin (s4171446)

October 2019
v2.0

Introduction

This document contains information regarding the architectural choices of the project and our experiences implementing these.

Technology Stack

For the backend we decided to use the programming language Python as two members of the team have experience with it, and the third member was able to learn it easily due to its simplicity. As a framework we decided to use the Flask microframework as it is easy to use and it is suitable for a small projects like this one.

As a database management system we decided to use MySQL. The reason we chose this over a no-SQL database is due to the (relatively) small amount of data we have to deal with (it is in the thousands). In other words, we are able to get very fast query responses.

We also made use of the 3^{rd} party API QuickChart. With this we can easily make graphs, and in our case histograms. To use this we can create a URL containing a JSON/JavaScript object that includes all data and display options and get the histogram we want to display.

Database Design

The CSV file with songs has a lot of duplicate information. For example, artists that are listed for multiple songs always have the same data. So we decided to split data into three tables: Artist, Song, Release (the last one does not contain any useful data for current API but it was kept just in case). To make things easier, some field names were changed, e.g. 'terms' to 'genre'.

In terms of relations, Songs contain a foreign key referencing Artist (id of the Artist) and Release (id of the Release) (see Figure 1).

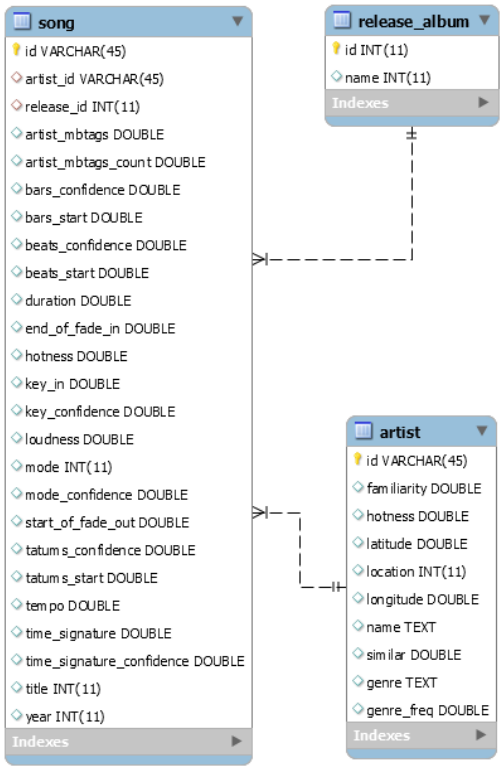

Figure 1: Diagram of the song database

Architecture of the project

The Flask application plays the role of the server which receives HTTP requests from the API user. After a request is received, the server forms an SQL query and sends it to the MySQL DBMS which responds with with

some data. In turn, the server encodes this data (in case of a GET request) in JSON or CSV format and sends it to the user in a HTTP response.

API design & implementation

To implement the API we used Flask's Blueprint object to seperate endpoints in their own file. In practice, implementation of an endpoint went through 3 steps: input checking, building queries from valid arguments (as well as executing these) and finally formatting the result into a valid HTTP response.

As we were unable to quickly find a good standard method for checking input we defined a Constraint class with several inheriting classes such as LengthConstraint, TypeConstraint or CustomConstraint. After specifying constraints on each input field/argument we would feed our input and a set of constraints to a function that would check for each field/argument if constraints were met. In hindsight using the re (regular expression) library to check for this would probably have been cleaner and more efficient than using a solution of our own.

When implementing building queries from often many (conditional) arguments that required a specific order, we often ended up with long functions with many conditionals. It would probably be possible to define an abstraction that deals with this more general case, but the way we dealt with this was more time-efficient for us. In order to execute queries, we defined a context manager function which allowed for easy GET querying and cleanup, however for our POST/PATCH/UPDATE queries we did not use this solution as obtaining the rowcount for queries was not compatible with our solution at the time.

Formatting our response came down to two things: formatting the content to JSON or CSV as well as adding appropriate headers and status codes to our HTTP responses. When formatting text, the f-string functionality Python offers really helped to keep the formatting process easily readable. In addition to the usual response statuses and headers, we added custom headers with links to indicate if another page was available. While including these in a JSON response instead would have been possible, we chose not to do this as CSV does not have an easy way to include additional metadata.

Front-end Design & implementation

Overview

For the front-end the following tools were used:

- HTML
- JavaScript
- Fetch API

We added a home page from which the following pages can be accessed:

- artists
- songs
- CRUD song
- keys (additional)
- genres (additional)
- statistics

Logic and Implmentation

We designed a simple fronted for our API. In the home page for artists, term values can be specified. When submitting, a request is sent to the server. This is done by using the Fetch API. With the data in the response (converted to JSON) we make a table and display it. The same approach is used for songs. The pages for keys and genres work in a similar way, but instead of making a table, a JSON object is constructed used for QuickChart, which will produce a histogram.

For searching/adding/modifying/removing single songs, we have another page. Here all the song fields are the inputs for a form. After providing an Id, we can get the values of that song by pressing the 'search'-button, which are filled in the input fields. We can also add/modify/remove the song with the given Id by using, respectively, the 'Add'/'Modify'/'Remove'-button.