Web Engineering Documentation

Robert Patrona (S3322211) and Yona Moreda (S3324338)

Group 20

24 October 2019

API Design

Define data model

The data model consists of several songs and their corresponding attributes such as the year of the song, the hotness index which rates the popularity of the songs, duration etc. In addition, the data model also contains artists which consists of their id, name, location, familiarity, etc.

Split data into resources

The data is split into two major resources: the artists and the songs.

Name resources with URIs

Resource: The artists **URL**: `/artists` **URN**: `artist-name`
Resource: The songs **URL**: `/songs` **URN**: `song-id`

Design representations

The representation simply consists of the resource represented into its corresponding `json` or `csv` format. The `json` format will enable a representation of name/value pairs for the requested resources while `csv` provides the data with comma separated values.

Link between resources

The resources are organized into "artists" and "songs" collections. Also, the artists and songs are linked with each other. For instance, they can have the same genre and artist name.

Model typical sequence of events

When querying a request, the user goes through a few steps. The first step is specifying the type of request, followed by the collection name, and then the query parameters. For instance, in the query `GET /songs?id=<song-id>`, `GET` denotes the type of request, `/songs` refers to the collection of songs, and `?id=<song-id>` is the query parameter. In addition, when doing a PATCH query, like in query 6, the user also needs to provide his own data, such as `{ song.title : <song-title> }`, which the API will use to update the entry in the dataset.

Headers

The `Content-Length` and `Last Modified` headers are handled by the API when processing requests. These are both entity headers which give information about the message body. The `Content-Length` header is used to specify the size of the information retrieved, while the `Last Modified` records the date of the last modification of a resource. This is used in caching, for instance, a browser will retrieve a resource from the cache if the `Last Modified` date is relatively old, which means that the cached resource has not expired and will be provided to the user. Otherwise if it has been modified recently, the browser will request the resource directly from the server.

Response bodies

The responses will be in JSON format. They will be of the form:


```
{
   "Artists": [
     {
       "Artist": {
         "artist.familiarity": "0.581793766",
         "artist.hotttnesss": "0.401997543",
         "artist.id": "ARD7TVE1187B99BFB1",
         "artist.latitude": "0.0",
         "artist.location": "0",
         "artist.longitude": "0.0",
         "artist.name": "Casual",
         "artist.similar": "0.0",
         "artist.terms": "hip hop",
         "artist.terms_freq": "1.0"
         }
     },
       {
       "Artist": {
         "artist.familiarity": "0.581793766",
```

```
            "artist.hotttnesss": "0.401997543",
            "artist.id": "ARD7TVE1187B99BFB1",
            "artist.latitude": "0.0",
            "artist.location": "0",
            "artist.longitude": "0.0",
            "artist.name": "Casual",
            "artist.similar": "0.0",
            "artist.terms": "hip hop",
            "artist.terms_freq": "1.0"
            }
        }
    ]
}
```


This is a sample query result. There is an array `Artists`, which contains `Artist` elements in JSON format. Each `Artist` element has the requested fields, which have the same data type as their corresponding fields in the CSV. For instance, artist.name is a string, while `artist.hotttnesss` is an integer.

Update Request Body

The PATCH query is an Update operation, which updates the name of a song. It has the following request body:

```
{
    song.title : <song-title>
}
```

The request will have the headers:

```
PATCH /songs/<song-id>/rename HTTP/1.1
Host: localhost
Content-Type: application/json
Content-Length: <length>
```

The request body consists of a `application/json` format. It contains the attribute name `song.title` and its corresponding value `<song-title>`. Based on the provided `<song-title>`, the title of the song with the matching id will be updated.

Define error conditions

A likely error condition might be requesting a PATCH query to update a song entry in the dataset. The error might occur if the provided song ID does correspond to any song.ID in the dataset. Another error condition might be requesting GET queries while providing an artist name which is not found in the dataset. In these cases, the API will respond with an error message.

Pagination

The pagination for the retrieved list of songs or artists are specified using two numeric query fields. The first field, `pageSize`, is used to specify the number of listed entities on a page and the second field `pageStartIndex` to indicate the starting index of the list of entities for the page. The pagination will follow the following format:

```
GET /<some-collection>?pageSize=<page-size>&pageStartIndex=<start-index>
```

Query Design

The following headers will be used in all requests.

```
Content-Type: application/json or text/csv;
Content-Length: <length>
Last-Modified: <day-name>, <day> <month> <year> <hour>:<minute> GMT
Cache-Control: max-age=300
```

For the cache control, we decided to use to use data up to 5 minutes of age. This is based on the assumption that the relevant data will not change within 5 minutes intervals.

1. Get all artists from the dataset, optionally filtered by artist name or music genre.
`GET /artists?name=<artist-name>&terms=<artist-terms>`

2. Get all information about a song from the database, identified by its unique song ID.
`GET /songs?id=<song-id>`

3. Get all songs by an artist given the artist name, or all songs in a specific year.
`GET /songs?artist=<artist-name>&year=<song-year>`

4. Get all songs by artists in a specific genre.
`GET /songs?terms=<artist-terms>`

5. Get the mean, median and standard deviation statistics of the popularity of the songs for a specific artist, optionally filtered by year.
`GET /artists/<artist-name>?year=<song-year>`

6. Update the song title (rename) of a song in the dataset.
`PATCH /songs/<song-id>/rename { song.title : <song-title> }`

7. Delete a song using its song id.

`DELETE /songs/<song_id>/delete`

8. Add a song for an artist.
`POST /artists/<artist-name>/addsong`

The first 5 queries support the Read CRUD method, since they all retrieve information from to the client, without modifying it. The 6th query supports the Update CUD method, since it allows the user to modify a resource, namely the song title field of a song entry in the data set.

We will have 3 endpoints. One endpoint deals with the `songs` collection and supports the GET and PATCH queries and by extension the CRUD methods Read and Update. The other two endpoints deal with `artists` and `stats` collection, and they both support GET queries, or Read methods.

Architecture, Technology Selection & API Implementation

For the back-end implementation of our API, we use Python Flask. Flask is a micro framework, which means that it does not depend on external libraries. It is very lightweight and offers its users a lot of freedom in implementing API, being able to be transformed into more advanced frameworks by using a few extensions if needed. It is one of the most flexible framework, while also being ORM-agnostic, and having a built-in development server. Moreover, Flask documentation is abundant, and well-structured which really helps us in our learning process. For the front end, we will use HTML templating with CSS. HTML and CSS are the backbones of front end development, and are easy to write, edit and maintain, are compatible with all browsers and are user-friendly. We will make use of libraries like Bootstrap to help us with implementing the view.

In our API, we have four routes which deal with each of our endpoints, and one which deals with a value-added feature. The `artists` route facilitates the first query, which returns all the artists in the database filtered by name or genre. The `songs` route returns all information about a song based on its ID, or a list of songs based on artists name and year, or the genre of the artist. This fulfils the second, third and fourth queries. The `songs/stats` route calculates the mean, median and standard deviation statistics of the popularity of songs filtered by artist name or year, the fifth query. Finally, the `songs/<id>/rename` route is a value-added feature which deals with the sixth query, enabling the user to update the name of a song based on a specific ID.

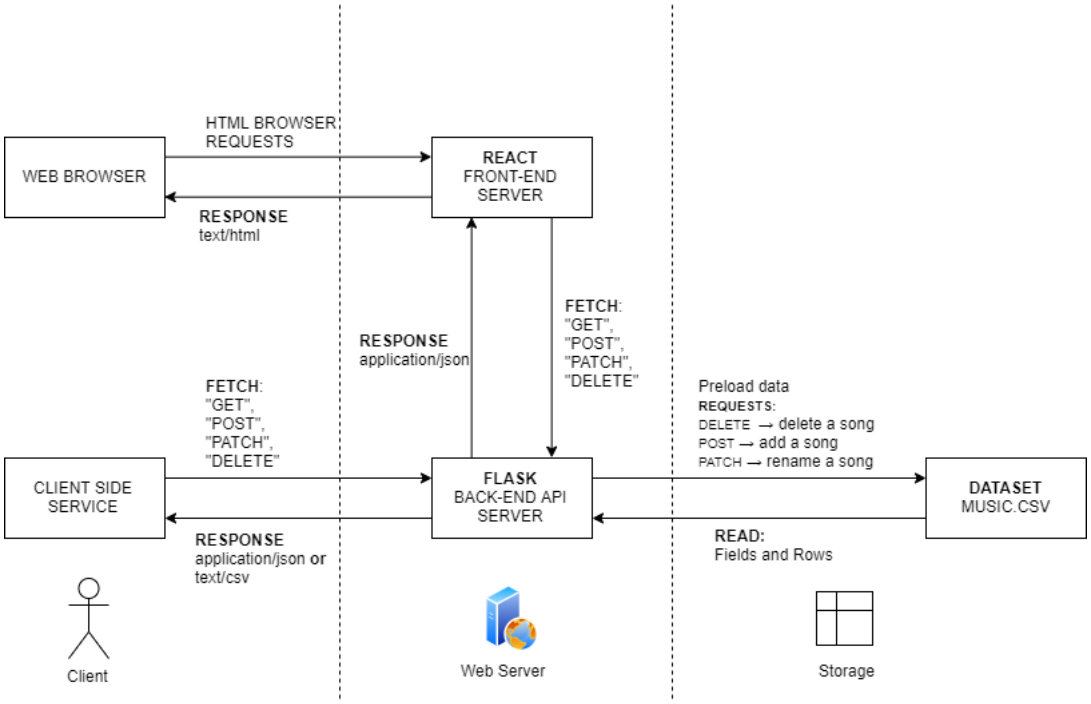

Figure 1: Architecture diagram

Architecture

Front-End/Back-End Structure

For the front-end implementation, we use React JavaScript library to create different UI components for the back-end API service. For each task, a query is created and is sent to back-end and the back-end gives an appropriate response for each requested query. All the different ends points provided from the API server are utilized in front-end server. The full front-end to back-end interaction of the system is shown in Figure 1 and the front-end component structure is shown below in Figure 2.

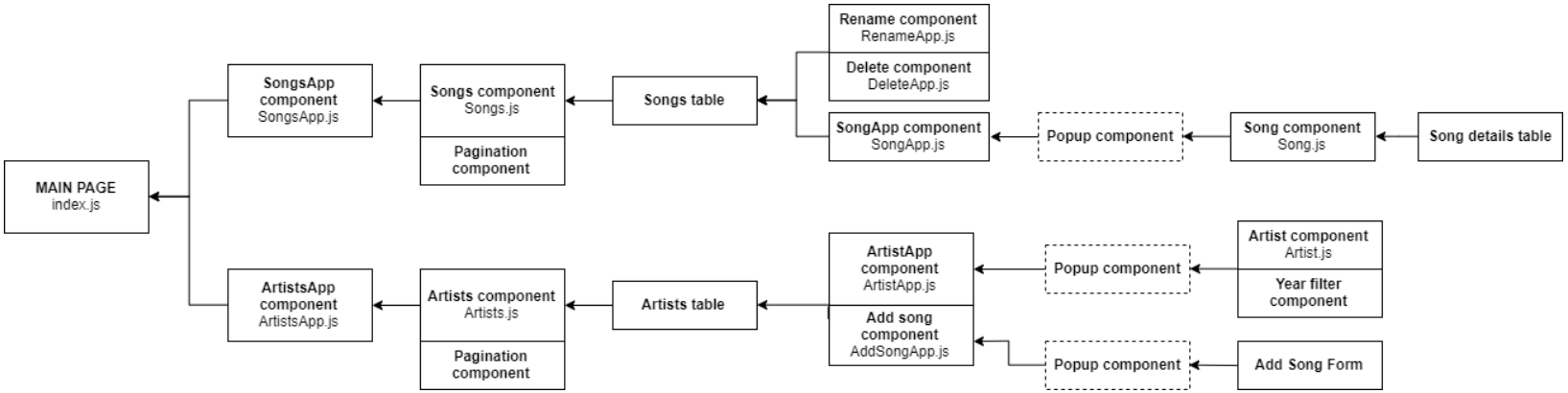

Figure 2: Front-End Components Structure