Web Engineering 2019

This repository contains our code for the music API for web engineering 2019.

Usage

```
make
./server -h # help
./server    # listen on port 8080 (default)
./seeddb    # todo, but seed database later when we have one
```

Structural organisation

Much like most go projects, this one has a `cmd` directory that contains all of the application entry points. The most commonly used one will be `cmd/server` which is the one that will start the server. The code for the server itself is contained in `api`. At some point documentation will end up in the `doc` directory.

Currently this projects contains a single package (which might also change) that is the main package. In the main function in `cmd/server` we limit ourselves to some flag parsing, and immediately go into the `run()` function from `api/server.go`. This is because main cannot return an error, but `run()` can (which `main` will then format and print).

Throughout the code you will find structs, such as the `Config` struct and the `server` struct. These are the way they are in the interest of reducing global variables.

The `server` struct holds global resources, such as database connections. This struct is accessible by every route handler function.

Routes

In spite of Golang's generous standard library, we outsourced the routing to gorilla mux. This decision was made because Go's router from the standard library does not filter by request method very nicely, while this one does.

The routes can be found, as you may have guessed, in `routes.go`. This gives a nice overview of all the API endpoints which is convenient for debugging (and grading). Each route has an associated `HandlerFunc`, which are created by functions in the `server` struct. We do this so that every `HandlerFunc` has its own closure environment, and thereby void global variables.

API doc

`GET /api/songs` – returns a list of all songs. Query parameters:

You are able to filter the list of songs according to the following parameters:

- artist_id

- year
- genre

In addition to these filters, you are able to paginate the response using the following parameters:

- limit
- page

You are also able to sort the response using the *sort* query parameter. This can be used to sort in *hotttnesss*, which in combination with *limit* can yield a top 50 list.

Example query: `GET /api/songs?year=2010&sort=hotttnesss&limit=3`

–

Response code: 200 OK


```
{
    "content": [
        {
            "song": { song1 },
            "links": []
        },
        {
            "id": "SO12456",
            "name": "Never gonna give you up",
            "links": [ { "self": "/api/songs/SO12456" } ]
        },
        { song3 },
    ],

    "links": {
        "self": "/api/songs?year=2010&sort=hotttnesss&limit=3",
        "next_page": "/api/songs?year=2010&sort=hotttnesss&limit=3&page=2",
        "last_page": "/api/songs?year=2010&sort=hotttnesss&limit=3&page=6",
    }
}
```


`GET /api/songs/{id}` **Query parameters**: none

Returns all information about a particular song.

Example query:

`GET /api/songs/SO12456`

–

Response code: 200 OK

```
{
    "song_id": "SO12456",
    "title": "Never gonna give you up",
    "artist_id": "AR1CKASTL3Y",
    "release_id": "213918492",
    "length": "212",
    ...
    "links" [
        "self": "/api/songs/SO12456",
        "artist": "/api/artists/AR1CKASTL3Y",
    ]
}
```

Questions to Sofie

1. Which way are we supposed to represent pieces of data in the paginated responses? (song1, 2, or 3)
2. How are we supposed to include links in the single data response e.g. `/api/songs/SO12456`. Options are `{ "id": "some id", "name": "some name", "links": [ .. ] }` or `{ "data": { "id": "some_id", "name": "some name" }, "links": [ ..  ] }`.