Project 2 – Flask init

In this project you are going to be able to connect to your website via a web server. In other words, you should have all the basics routes working already, but no logic.

Task 1

The first task is to create the right skeleton. While this seems to be an easy task, it is not. Use the following structure:

-/app.py (should be your entry point)
-/static/css/ (should contain your CSS)
-/static/music/ (should contain some mp3 files)
-/frontend/ (should contain your HTML files)
-/routes/ frontend/ (should contain the routes used to display your HTML)
-/routes/api/ (should contain the API that your website WILL use. Empty for now)

In app.py you should initialize your webserver using Flask and listening on port 5000
Remember: use "python3 -m venv venv" to create a virtual environment. Every time, you can load your virtual env via "source venv/bin/activate". You can deactivate using "deactivate". Once it is activated, you can install packages (as Flask). Every time you install a package, remember to do "pip freeze > requirements.txt"
Theoretical question: why venv? Why requirements.txt is needed?

The goal of this task is to have the basic structure, and to have a first route that is listening on localhost:5000 saying "hello world"

Tip: remember "__init__.py"
Theoretical question: why __init__.py? What is it used for? Why is here necessary?

Task 2

There is an easy way to serve the static files in Flask. Can you find it? You should serve the static file under the route /static. For example, if you have a file "/static/css/index.css" you should navigate to "localhost:5000/static/css/index.css" and you should see the CSS file. Same for the mp3 and all the rest of the content should apply.

Task 3

The goal is to serve your HTML pages as they are. In /routes/frontend/ create frontend.py. Define one route per page:

- "/" for index.html
- /album
- /artist
- /tracks
- (other pages?)

Please, use "Blueprints" to define the routes:
https://flask.palletsprojects.com/en/2.2.x/blueprints/

For each route, you need to “serve” the relative HTML page. In other words, you need to load the correct HTML file in “/frontend/” and send the text to the user.

Now, if you go to “localhost:5000/album” you should see your album page.

Tip: the CSS is not loaded? Remember that now you serve it under “/static/css/….”. When you load the CSS in the HTML, do not include “localhost:5000”, just the absolute path is enough.

Theoretical question: if you had many, many pages for your website, having a single “/routes/frontend/frontend.py” file would not be suitable. Why? What would be a better solution in case you had tens or even hundreds of different HTML pages?

Project 3 – A simple Database

There is no application without a database. There are many different DBs, all different among each other.

Theoretical question: what are the two main categories of DBs? When is used what? What are the PROs and the CONs?

In this project you are going to use the simplest real DB out there: SQLite. The cool thing is that SQLite works in a file, so there is no need of a complex DB. Good for us!

Tip: spend 10 minutes to familiarize yourself with SQLite. Understand what it is, what tools you can use to can view a DB, …

In Flask, there is a cool library called SQLAlchemy.

Theoretical question: when interacting with a DB, there are three ways you can do that:

- *Raw: you write the SQL queries yourself*
- *Query builder*
- *ORM*

What are the advantages and disadvantages that you see among these methods? What would you use in a simple application? And what if performance is the most important thing? What is SQLAlchemy (yes, this is a tricky question)?

Task 1

In the folder "/models" (that you need to create), create the following models:

Artist:

- IdArtist: autoincrement, int
- name: string (64 char)
- surname: string (64 char)
- birthDate: DATE
- popularity: int

Album:

- idAlbum: autoincrement, int
- name: string (64 char)
- year: int
- popularity: int
- create a relationship many-to-many (an album can be authored by multiple people)

Song:

- idSong: autoincrement, int
- name: string (64 char)
- duration: int
- file: string (64 char)
- popularity: int

- create a relationship many-to-many artist (one song can be written by multiple artists), can be Null
- create a relationship one-to-many with album (one song can be only in one album)

Note: Album must have at least one artist, while a song can have one or more artists. Idea is that you could publish an album, and a song can be done by you and somebody else. If a song does not have an author (null), then you will infer that the author(s) is the same as the album
Note: all fields are required but where noted.

SQLAlchemy comes with a Flask extension. Initialize an SQLite DB as a file called db.sqlite in the root of you project (where app.py is). If you now use a visualizer, you should be able to see your DB and the correct table created.

Theoretical question: How do you see the one-to-many relationship represented? What about the many-to-many?
Theoretical question: What is varchar? It was used a lot in the past. Do you see anything better (or worse) than text?

Task 2

While testing an application, it is good to have a seed for the DB. In other word, you drop all the existing rows, and you add some pre-defined row. Let's do the same!
Create a new file /seed.py which should delete all rows in all tables, and create new pre-defined entries in your DB. You should have a bit of everything, so try to create about 7 albums, 7 artists and 10 songs. Mix and match all the relationship (for example, create an album with no tracks, create a track with no artist, and one with artist, create an album with all songs without artists, one all with artists, and one with a bit with and a bit without, …)
Tip: be realistic with Artist, Album and songs, so the debugging will be much easier (you will know when something is not working).
Tip: you can create more entries if you feel it is needed
Tip: filenames should only be the name of the file, that you will have in /static/mp3.

Task 3

Create a new route in app.py "/test" which should return the first track from the DB as a JSON. This task is just to make the connection between Flask and SQLAlchemy. Leave the route in.

Project 4 – MVC

The concept of MVC is very complex, but in a nutshell, you want to keep your logic, your views and your data all in separated parts of the application. You view and you data is already separated: one is a route serving HTML, the other are the models representing your tables. We need to work on the logic.

Create a new folder "/interfaces" and create a new file per model (for example, create "/interfaces/track.py". In each file you will put the logic to get, modify, delete, …, each model. In our simple app, we will just get things.

Create the following functions:

Track.py

- get the top "n" (n is given in input) track ordered by popularity
- get all the tracks
- get all albums which contain a given word in any of the fields (used for search. Use the LIKE function in the DB)

Album.py:

- get the top "n" by popularity
- get all the albums
- get all the tracks in a given album (by idAlbum)
- get all albums which contain a given word in any of the fields (used for search. Use the LIKE function in the DB)

Artist.py:

- get the top "n" by popularity
- get all the artists
- get all albums from given artists (by idArtist, note, multiple IDs can be passed. You should use an AND. If artist A and B are given, you should return all the albums created by A AND B)
- get all the tracks from given artists (by idArtist. Again many artist can be passed to the function.)
  Note well! Some tracks will not have an artist, as the album has it. If a track does not have an artist (null), you should check if the idArtist is owing the album where the track is. For example:
  - album A is from artist F and has track:
    - 1 with artist null
    - 2 with artist F and K
    - 3 with artist J

  If I giv you artist K, you should return song 2, if I send you artist F, you should return song 1 and 2 (two joins are needed)
- get all artists which contain a given word in any of the fields (used for search. Use the LIKE function in the DB)

Project 5 – Templates: Jinja

An old technique (but still very used) to serve webpages is by using template languages. Instead of using fancy frontend like Vue or React, you dynamically generate the HTML, and then serve it to the client.

Why serving HTML directly to the client is sometime better than using fancy frontend tools like Vue or React? (This was true especially years ago)

Play a bit with Jinja (https://jinja.palletsprojects.com/en/3.1.x/), see how it works and play a bit around. Jinja is also very used in scripting (I personally used it multiple times in the last 2 years).

Why is Jinja used in scripting? Can you think of some script that might take advantage of Jinja?

Task 1

Port your HTML files into Jinja templates into /templates folder. Try to reuse the common components as much as possible. For example, you could reuse the side bar in all the page and define it only once

Task 2

Using Project 4 and 2, you should be able to create all the pages. If needed feel free to create new functions in /interfaces. Do not add any JS (I do not expect you player to work).

If I go to /, I should see the top 5 artists, 5 tracks, and 5 albums. In /album I should see ALL the albums. In /album/{idAlbum} I should see the tracks in the album. Similarly, in /artist I should see all the artists, and in /artist/{idArtist}/alum the list of albums, while in /artist/{idArtist}/track the list of tracks. In /track I should see the list of all the track. Around, links should be clickable (if in the track you write the album and the artists, then the album should bring you to the /album/{idAblum}, and the artist to /artist/{idArtist}/track. Add functionalities as you see them fit.

Task 3

Finally, the search form should in the navbar should be send a request to /search, and it should return the correct info divided into tracks, artist, album. You will have to implement this in HTML and the route, but the interfaces already support the search function.

Project 6 – Simple JS

The goal of this project is to actually play the track's audio. From the DB you know the filename, and as the files are saved under /static/mp3, you can access the files.

Use vanilla JS, not JQuery or any library. Use "audio" to manage the audio.

Task 1

When I am in an album, and I click on a track, I should be able to start the music. If I click on next, I should go to the next song in the album. With prev in the previous song. If I am in any other list of tracks (like when I go to /track) a similar concept should apply. For search, the same yet again.

Task 2

Make the volume, the time bar of the song, and the play/pause button work.

Project 7 – API

Go back to project 4 (MVC) and create an API endpoint for each function. The idea is that with GET requests you should have the correct result.
This project is very small, but very important for the next project
You API should be in /routes/api/v1/….

Theoretical question: why putting the API under v1? What is the advantage?

Project 8 – React and Vue

At this point, you know the drill, you know the app, and you know the functionalities probably even better than me. This will be a multi-day project.

Choose React or Vue first. What do you prefer.
Theoretical question: why do we use tools like react and vue? What is the advantage over the previously used method?

Task 0

Based on the chosed library, your url for the frontend should be something like: localhost:5000/react or /vue (and localhost:5000/ should display the website using Jinja as before)

Task 1

Create the components for your app. They should represent your original graphic 100%. Do not use pre-defined components. The reason is simple: pre-defined components do not allow you to fully understand how components work and can be very painful sometimes to debug.

Task 2

Recreate your app with the whole logic, but do not use Jinja, use the API endpoints to display the various items. Also, while you click the app, you should change the page but not reload. In other words, by clicking around, the music should not stop. The URL should change based on the page you click on, and the history of your browser should be pushed.

Task 3

You Now do it with the other library

Theoretical question: what is better Vue or React? What are the differences? What do you prefer and why?