We will learn to create a simple and basic flask based webapp. Flask is one of the best and light-weight web application framework in python. Fast api is good for asynchronous tasks and to serve rest api quick. Django is all in one web framework that has built-in admin dashboard, user authentication and so on. Flask, provides relatively simpler interface and easy to learn for beginners so we will learning that. It should not be very complicated to move to either fastapi or django after learning the basics of flask.

Change the directory to Workspace and create new folder name flask and change to that directory. Crate a new virtual environment inside that directory. Source the environment to activate it. Then install flask and psycopg2-binary library inside that virtual environment.

```
ardent@ardent:~$ cd Workspace/
ardent@ardent:~/Workspace$ mkdir flask
ardent@ardent:~/Workspace$ cd flask
ardent@ardent:~/Workspace/flask$ python3 -m venv venv
ardent@ardent:~/Workspace/flask$ source venv/bin/activate
(venv) ardent@ardent:~/Workspace/flask$ pip install flask psycopg2-binary
Collecting flask
  Downloading flask-3.1.1-py3-none-any.whl.metadata (3.0 kB)
Collecting psycopg2-binary
  Using cached psycopg2_binary-2.9.10-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.9 kB)
```

With this we have our initial setup and can get started developing flask app. Open this directory in vscode

```
Using cached click-8.2.1-py3-none-any.whl (102 kB)
Using cached itsdangerous-2.2.0-py3-none-any.whl (16 kB)
Using cached jinja2-3.1.6-py3-none-any.whl (134 kB)
Using cached MarkupSafe-3.0.2-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (23 kB)
Using cached werkzeug-3.1.3-py3-none-any.whl (224 kB)
Installing collected packages: psycopg2-binary, markupsafe, itsdangerous, click, blinker, werkzeug, jinja2, flask
Successfully installed blinker-1.9.0 click-8.2.1 flask-3.1.1 itsdangerous-2.2.0 jinja2-3.1.6 markupsafe-3.0.2 psycopg2-b
inary-2.9.10 werkzeug-3.1.3
(venv) ardent@ardent:~/Workspace/flask$ code .
(venv) ardent@ardent:~/Workspace/flask$
```

Create a new file app.py and add the following code:

```
EXPLORER
FLASK
> venv
app.py
Welcome
app.py
app.py > ...
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/')
6  def index():
7      return "Hello World"
8
9  if __name__ == '__main__':
10     app.run(debug=True)
```

Initially, we import Flask which will be used to create an instance of app (Flask). __name__ is used to tell flask where to perform its path related setups

@app.route decorator is used to route each function to a url. / is the default/root route and it is being mapped to index function. Initially we are just returning “Hello World” as text.

Finally we have a __name__ check to verify whether this file is being run or not.

Let us run this code

```
if __name__ == '__main__':
    app.run(debug=True)
```

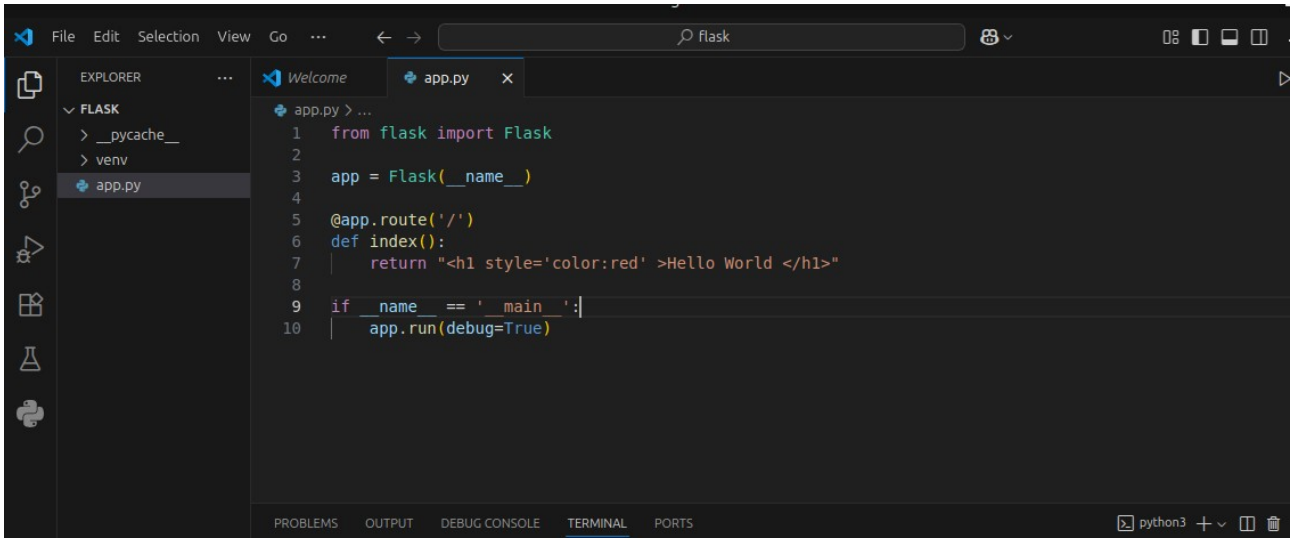
```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                python3

(venv) ardent@ardent:~/Workspace/flask$ python3 app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 387-772-040
```

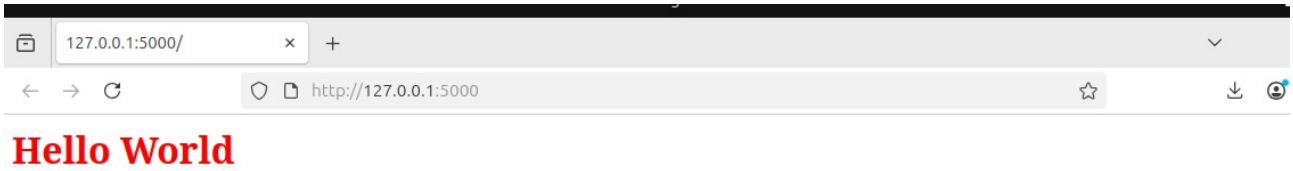
Run it the same way we will execute any python file. Open the url mentioned on brower i.e., http://127.0.0.1:5000

This should be your output

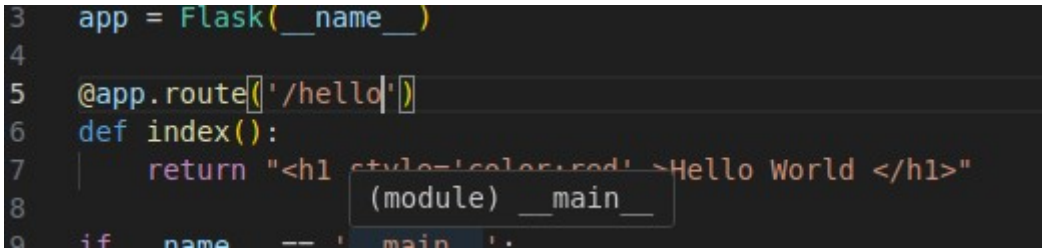
Let us change the code a little bit. Return an h1 tag with color set to red. It is assumed that you understand html and css



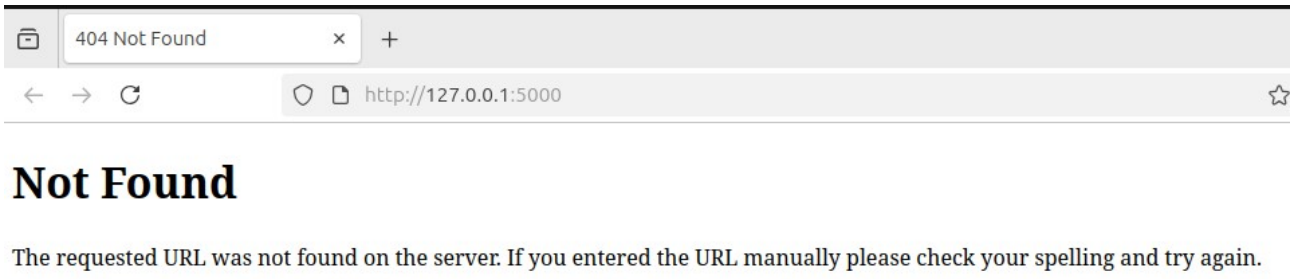
Save the file and reload the web browser. You should get below output.



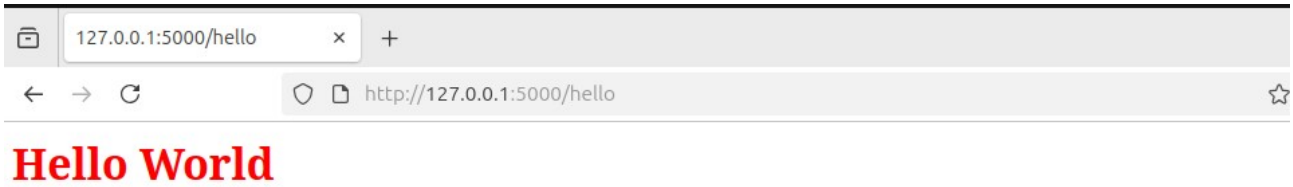
Now let us change the route of the function



Save the file and reload the browser



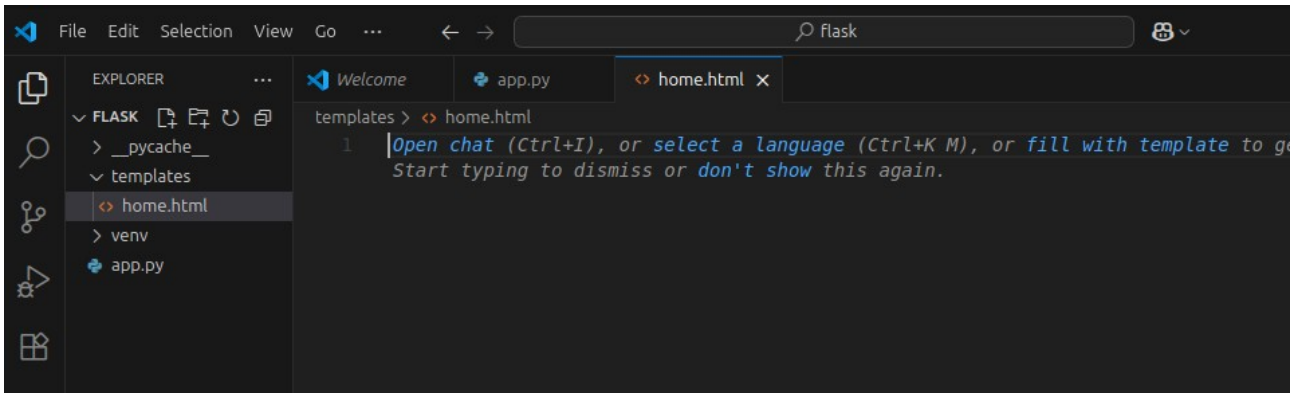
You will get the below error. This is because the route has change from / to /hello. Let us change our url as well.



Now, you should be able to view the content again. This is the most basics of setup. We can create as many routes as we want and assign a separate function for it.

Also, it can get tedious to write html code directly in python string format so flask provides templating engine which we will learn next.

Create a new folder ‘templates’. Inside it create an html file home.html



Add a basic html code and our hello world header

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Home Page</title>
</head>
<body>
    <h1 style='color: red' >Hello World from template</h1>
</body>
</html>
```

Create new function home and route it to / and return an render_template instance with the name of html file

```python
@app.route('/')
def home():
    return render_template("home.html")
```

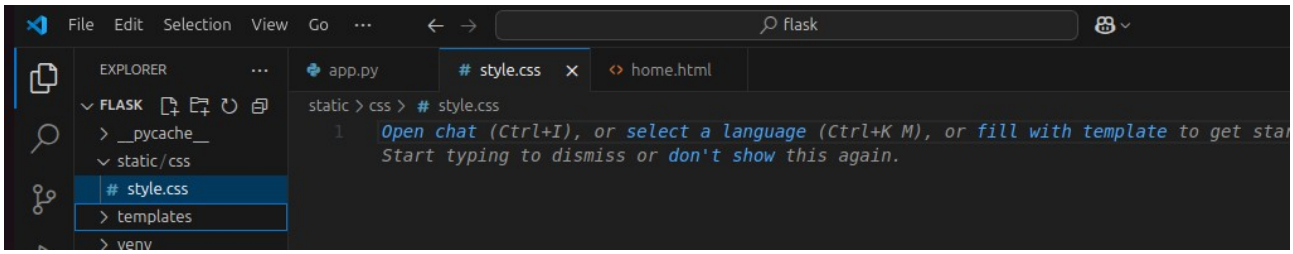
Make sure you import render_template function at the top

```python
from flask import Flask, render_template
```

Save and rerun the python code

You should be able to see above output. Now since we might have several elements and it can get tedious to add style components in lined. Let us create CSS file as well

Create an style directory in the root folder. Then create css directory inside and inside css direcoty create an style.css file



Add the css content for h1 inside it

```
Go ···                                  Fla
app.py   # style.css ×   home.html
static > css > # style.css > h1
1  h1 {
2      color: burlywood;
3  }
```

In our html file we need to make changes to include this file. Add the link tag. Make sure you follow the pattern as shown for href. Remove the previous style tag inlined in h1

```
Go ···                  flask
app.py   # style.css   home.html ×
templates > home.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Home Page</title>
6      <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
7  </head>
8  <body>
9      <h1>Hello World from template</h1>
10 </body>
11 </html>
```

Save the changes and refresh the webpage in browser

```
Home Page ×   +
http://127.0.0.1:5000

Hello World from template
```

Now we have the basic setup working. Let us create a new function that connects to postgres database and retrieves result and displays it

Let us create a new route and function db

```
app.py
@app.route('/')
def home():
    return render_template("home.html")

@app.route('/db')
def db():
    return "For db"

if __name__ == '__main__':
    app.run(debug=True)
```

Let us add the link to this route from our / root route html i.e., in home.html add the following:

```
ome.html
v
.py
    <title>Home Page</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.c
</head>
<body>
    <h1>Hello World from template</h1>
    <a href="/db">View db data</a>
</body>
</html>
```

Save the files with changes. Go to browser and refresh

Click on the View Artist Dashboard link. You should be re-routed to the proper route/function

Let us connect to db and retrieve result for the db function. You can also get the below code from my github repository

https://github.com/neotheobserver/flask-intro

Crate an dictionary of all the parameters required for connection and pass it to pyscopg2 to connect to the database.
Create cursor out of that connection and execute a query.
Retrieve results of the query and save it in a variable named tracks. Close the cursor and connection.
Return render_template function with db.html and also the tracks variable

```
    return render_template("home.html")

@app.route('/db')
def db():
    DB_CONFIG = {
    'dbname': 'postgres',
    'user': 'postgres',
    'password': 'postgres',
    'host': 'localhost',
    'port': '5432'
}
    connection = psycopg2.connect(**DB_CONFIG)
    cursor = connection.cursor()
    cursor.execute("SELECT id, name, danceability, energy, tempo FROM track_metadata LIMIT 10")
    tracks = cursor.fetchall()
    cursor.close()
    connection.close()
    return render_template('db.html', tracks=tracks)


if __name__ == '__main__':
    app.run(debug=True)
```

Modify the db.html template to incorporate the changes we have made

```html
<head>
    <meta charset="UTF-8">
    <title>Track Data</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
    <h1>Track Data</h1>
    <table border="1">
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Danceability</th>
            <th>Energy</th>
            <th>Tempo</th>
        </tr>
        {% for track in tracks %}
        <tr>
            <td>{{ track[0] }}</td>
            <td>{{ track[1] }}</td>
            <td>{{ track[2] }}</td>
            <td>{{ track[3] }}</td>
            <td>{{ track[4] }}</td>
        </tr>
        {% endfor %}
    </table>
    <a href="/">Back to Home</a>
</body>
</html>
```

Notice the use of {% %} {% endfor %} syntax. This is the standard jinja2 template syntax. It is used bu django as well. You can run python code in the html using this syntax

Let us now modify the style.css a little as well

app.py | db.html | style.css | home.html

static > css > style.css > a:hover

```
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 20px;
    background-color: #f4f4f4;
}
h1 {
    color: #333;
}
a {
    color: #007bff;
    text-decoration: none;
}
a:hover {
    text-decoration: underline;
}
```

Save the changes made. Go to web browser and refresh

http://127.0.0.1:5000

Hello World from template

View db data

Click the View db data link

http://127.0.0.1:5000/db

Track Data

ID	Name	Danceability	Energy	Tempo
000Npgk5e2SgwGaIsN3ztv	Mere Hamsafar	0.277	0.145	75.644
000TXa2oEZLYfQGPCiv23U	Casanova	0.61	0.91	129.639
000d0lQMYaRR5ZXS9nTeiN	Paris sous les bombes	0.826	0.827	92.137
000xYdQfIZ4pDmBGzQalKU	Eu, Você, O Mar e Ela	0.509	0.803	166.018
0017A6SJgTbfQVU2EtsPNo	Pangarap	0.682	0.401	97.091
001IcYypSE1ryXKY5KNIin	Baby Sleep Waves	0.111	0.376	95.28
001YQlnDSduXd5LgBd66gT	El Tiempo Es Dinero - Remasterizado 2007	0.554	0.921	183.571
001eyxfoYptAWzvF4ewLrR	"Die Meistersinger von Nürnberg, WWV 96 / Act 3: ""Morgenlich leuchtend"""	0.326	0.181	135.032
0029TH4cSnQ12KKfHaq11C	L.A. Goodbye	0.616	0.819	126.816
002DkDzzQ7IrgaqWBF2o1M	Snabbköpskassörskan	0.509	0.715	178.144

Back to Home

Congratulation! You have successfully queried the database server and shown the result in web page to user. Everything from here on is just addition to the basic concepts covered before this.

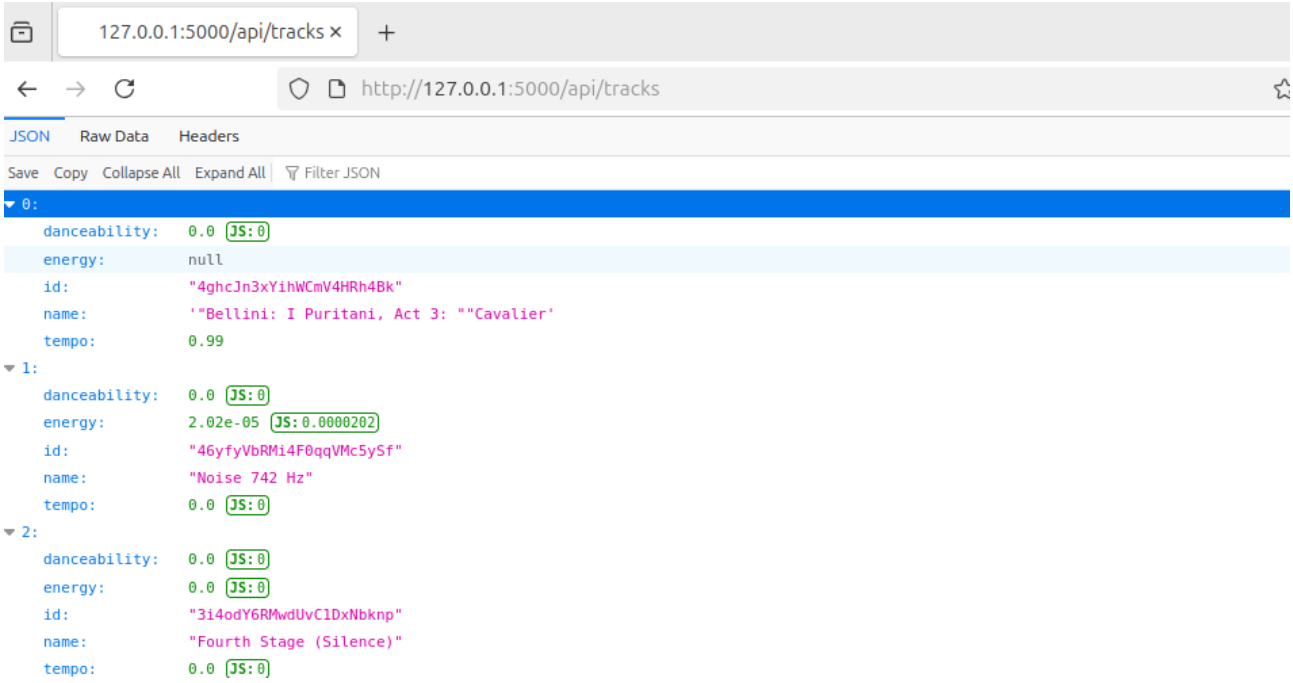
Let us try to create and display charts. First lets create an api endpoint that returns json of the track data that can be used by any chart library

```
@app.route('/api/tracks')
def get_artists_data():
    DB_CONFIG = {
    'dbname': 'postgres',
    'user': 'postgres',
    'password': 'postgres',
    'host': 'localhost',
    'port': '5432'
}
    connection = psycopg2.connect(**DB_CONFIG)
    cursor = connection.cursor()
    cursor.execute("SELECT id, name, danceability, energy, tempo FROM track_metadata order by danceability LIMIT 100")
    tracks = cursor.fetchall()
    cursor.close()
    connection.close()
    # Return JSON for Chart.js
    return jsonify([
        {'id': row[0], 'name': row[1], 'danceability': row[2], 'energy': row[3], 'tempo': row[4]}
        for row in tracks
    ])
```

Make sure you import jsonify

```
app.py > get_artists_data
from flask import Flask, render_template, jsonify
import psycopg2
```

If we save our file and refresh our browser and visit /api/tracks route we should be able to see the json being returned



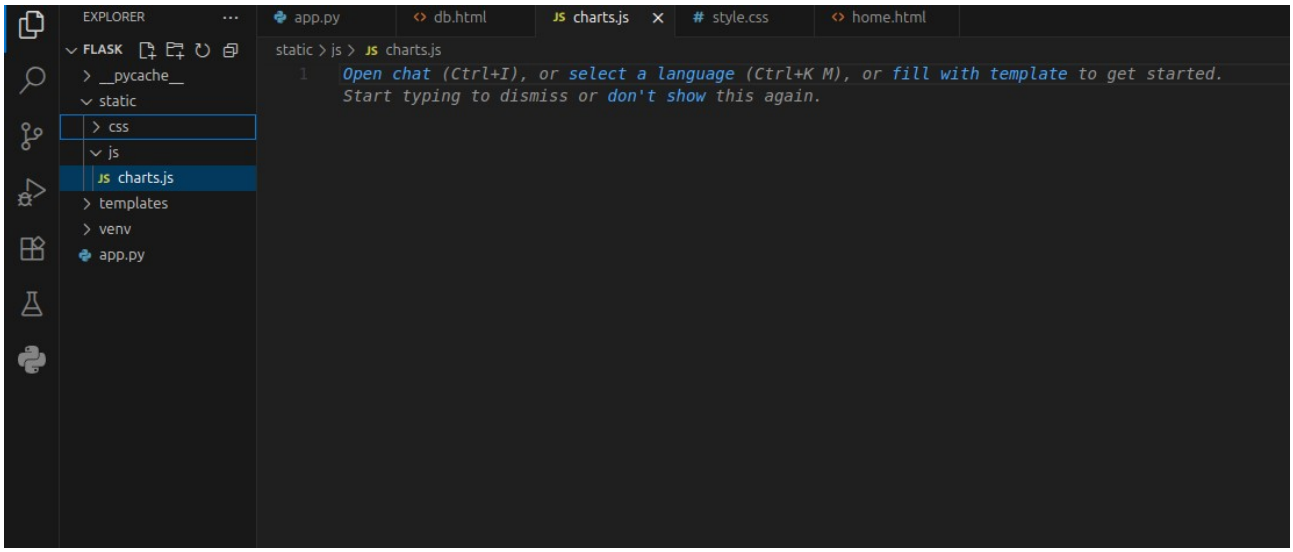
We have created our first api endpoint. Which can be used by other application (generally frontend) to get the required data. In our case we won't be running a separate frontend application but rather use javascript as we will be needing to incorporate java script to be display dynamic chart

Add a script tag with link to chart.js. Also add a script tag to charts.js file which we will be creating

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Track Data</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js@4.4.3/dist/chart.umd.min.js"></script>
    <script src="{{ url_for('static', filename='js/charts.js') }}"></script>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
```

Follow link (ctrl + click)

In the static folder. Create a new folder js and then new file charts.js



Add the relevant content. It is expected that you understand javascript. I will not be explaining any of it as this is not a javascript course.

app.py | db.html | charts.js | style.css | home.html

tic > js > charts.js > document.addEventListener('DOMContentLoaded') callback > then() callback

```javascript
document.addEventListener('DOMContentLoaded', function () {

    fetch('/api/tracks')
        .then(response => response.json())
        .then(data => {

            const scatterData = {
                datasets: [{
                    label: 'Danceability vs Energy',
                    data: data.map(artist => ({
                        x: artist.danceability,
                        y: artist.energy,
                        name: artist.name,
                        tempo: artist.tempo
                    })),
                    backgroundColor: 'rgba(0, 123, 255, 0.5)',
                    pointRadius: 5
                }]
            };
```

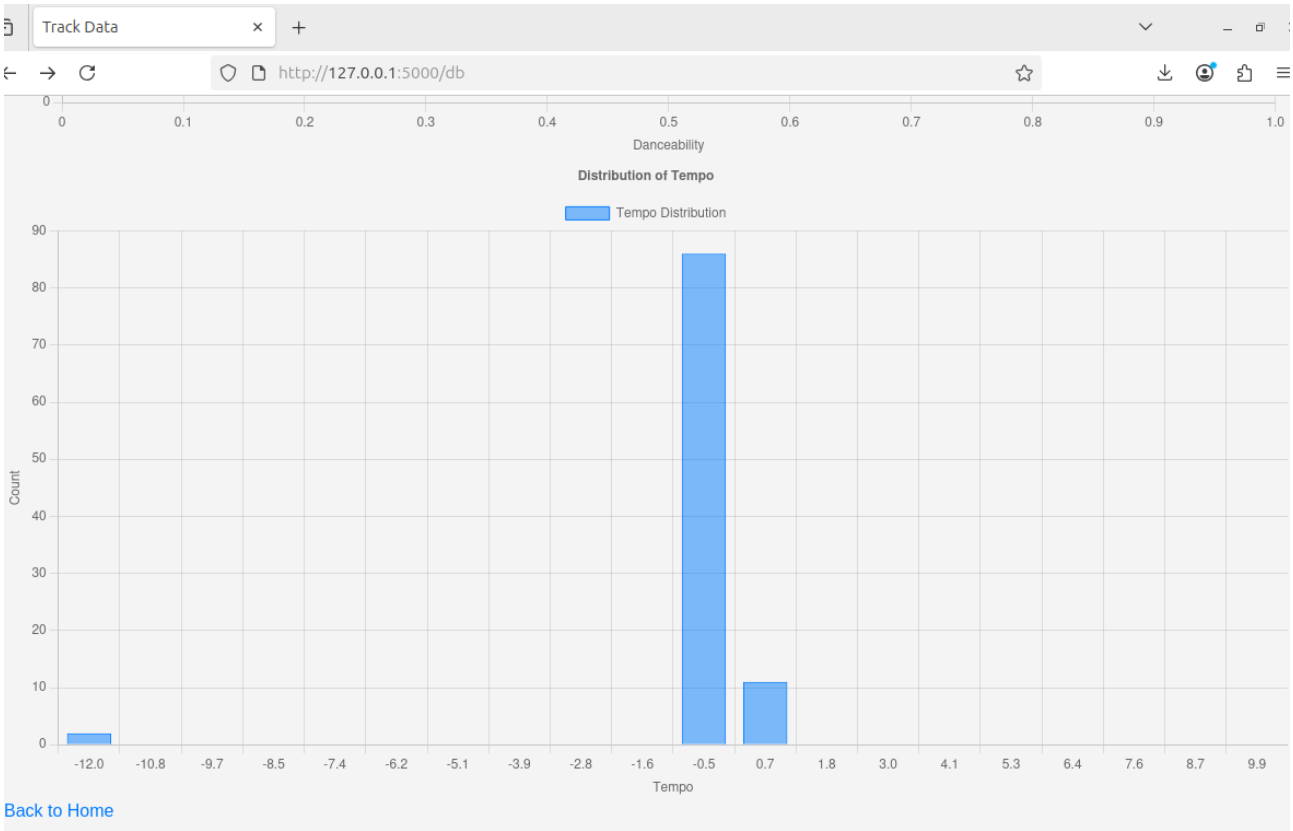
In our db.html file right after the content of table finishes. Lets add the visualization part. Bascailly, We will be creating two empty canvases which will then be loaded with charts by above javascript code. The js code first waits for content to load. Then calls the api we created earlier and using the data it responds with uses the Chart class of chart.js script we have included in our db.html file. They all work together to create one complete web page

```html
            <td>{{ track[2] }}</td>
            <td>{{ track[3] }}</td>
            <td>{{ track[4] }}</td>
        </tr>
        {% endfor %}
    </table>

    <h2>Visualizations</h2>
    <div>
        <canvas id="scatterChart" width="400" height="200"></canvas>
    </div>
    <div>
        <canvas id="histogramChart" width="400" height="200"></canvas>
    </div>

    <a href="/">Back to Home</a>
</body>
</html>
```

If you save the files. Refresh the browser. You should now be able to see the charts. The purpose of adding the js part is to demonstrate that things like this can also be done. However, much changes and work needs to be done before we can make it good and presentable. Queries must be modified. The styling must be changed and so on.



With this you have to basic idea of how to create a new route, whether it servers a webpage or json for an api, connect sytlesheets and javascripts files, how to connect to database and so on. This should get you started on the journey of creating webapp. Best of luck!