FLASK CON PYTHON

Universidad de Python en UDEMY

Se utiliza para hacer aplicaciones con Python. Framework aplicaciones de tipo rest o web. DJANGO ya trae muchos componentes ya integrados no como flask que todo lo maneja a través de extensiones.

1. INSTALACION

Proceso de instalación Página web de FLASK. Vamos a ver para instalar Flask en MAC en un entorno virtualizado por consola:

```
mkdir myProject
cd myProyect
python3 -m venv venv

source venv/bin/activate

pip install flask
```

Pero aquí lo vamos a hacer con Visual Studio Code.

2. COMENZANDO CON FLASK

Miramos la documentación de Flask y lo primero que nos explica es para ponerlo en funcionamiento.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

Importamos la clase de Flask, el primer argumento es el nombre de la aplicación que en mi caso lo estoy haciendo en el archivo app.py, creamos una instacia de la clase Flask siendo (__name) que se necesita para que flask sepa donde buscar recursos y templates. app.route es el decorador que le dice a flask que url debería usar en cada función. Mientras en nuestra consola:

```
$ python -m flask
$ export FLASK_APP=app.py
$ export FLASK_ENV=development
$ flask run
 * Running on http://127.0.0.1:5000/http://127.0.0.1:5000/
```

Si vamos a la aplicación en la web pordemos ver nuestro Hello World.

3. LOGGING

Algunas veces puedes querer comprobar si todo va correctamente, puede que tengas algún código del lado del cliente que envíe una solicitud HTTP al servidor, pero obviamente está malformada. Esto puede deberse a que un usuario manipule los datos o a que falle el código del cliente. La mayoría de las veces está bien responder con 400 malas solicitudes en esa situación, etc para eso estan los loggers, que los mensajes salen por la terminal.

```
@app.route("/")
def inicio():
    app.logger.debug("Mensaje a nivel debug")
    app.logger.warning("Mensaje a nivel warning")
    app.logger.error("Mensaje a nivel error")
    return "<h1>Olee tu polla bro</h1>"
```

4. MODULO REQUEST

Podemos realizar peticiones por ejemplo:

```
app.logger.info(f"Entramos al paht {request.path}")
```

la respuesta del servidor será la siguiente

```
[2022-04-06 13:03:55,648] INFO in app: Entramos al paht /
127.0.0.1 - - [06/Apr/2022 13:03:55] "GET / HTTP/1.1" 200 -
```

una respuesta de tipo GET a la ruta elegida en este caso, / .

4. ROUTING EN FLASK

Vamos a seguir poniendo más paths, todos comienzan siempre con /loQueSea @app.route("/user") definimos debajo una nueva función para que haga lo que sea ese path.

```
@app.route("/saludar")
def saludar():
    return "<h1>Salidos!!</h1>"
``
También podemos recibir parámetros y estos tienen que especificar de que tipo son:
1. String
2. Int
3. Float
4. path
5. uuid
para ello quitando el String que es por defecto hay que especificarlo.
```python
@app.route("/saludar/<nombre>")
def saludar(nombre):
    return f"<h1>Salidos!!{nombre}</h1>"
```

le pasamos el parámetro nombre y nos vamos a la url si ponemos http://127.0.0.1:5000/saludar/Victor la respuesta que veremos será SALUDOS! VICTOR Si le pasamos un Int como parámetro tendremos esto:

```
@app.route('/edad/int:edad<int:edad>')
def muestra_edad(edad):
    return f"<h1>Hola, tienes {edad} años, gilipollas</h1>"
```

si le ponemos /edad/40 en el navegador nos devolvera: Hola, tienes 40 años, gilipollas Con el valor de las variables que le pasamos podemos hacer de todo como cualquier variable en python nombre.upper para mayúsculas, edad + 1 para hacer operaciones aritméticas etc.

5. PETICIONES GET Y POST CON INSOMNIA REST O POSTMAN

1. GET El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.
2. HEAD El método HEAD pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
3. POST El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.
4. PUT El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
5. DELETE El método DELETE borra un recurso en específico.
6. CONNECT El método CONNECT establece un túnel hacia el servidor identificado por el recurso.
7. OPTIONS El método OPTIONS es utilizado para describir las opciones de comunicación para el recurso de destino.
8. TRACE El método TRACE realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino.
9. PATCH El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.

Vamos a descargar insomnia rest Client y elegimos nuestra instalación, yo mac en este caso. Arrancamos nuestro servidor en la aplicación, como siempre con, export FLASK_ENV=development y flask run. Creamos un new request en insomnia, le ponemos el nombre que queramos, podemos tener todo tipo de peticiones GET POST etc. Hacemos una petición de tipo GET a nuestra web htts://localhost:5000 y nos da varios datos en headers: Lo que resgresa: Server: Werkzeug/2.1.1 Python/3.8.9. Date: Sat, 09 Apr 2022 18:11:34 GMT. Content-Type: text/html; charset=utf-8. Content-Length: 26 //número de caracteres que estamos pidiendo
En Timeline es el que más información te ofrece.
```

GET / HTTP/1.1.

Host: 127.0.0.1:5000. User-Agent: insomnia/2022.2.1. Accept: /. El tipo de petición en este caso GET, a donde va dirigido en este caso a la URL de / de tipo http y la petición se ha hecho al localhost puerto 5000, usando como cliente (User-Agent): insomnia vemos la respuesta con el código de estado 200: HTTP/1.1 200 OK Método POST Si por ejemplo tenemos este código en una route del path:

```
@app.route("/mostrar/<nombre>")
def mostrar_nombre(nombre):
    return f"Su nombre es {nombre}"
```

Si hacemos una petición por el método POST, nos saldrá error ya que hay que programar que pueda recibir tanto GET como POST

```
@app.route("/mostrar/<nombre>", methods["GET", "POST"])
def mostrar_nombre(nombre):
    return f"Su nombre es {nombre}"
```

Ahora si vamos a insomnia y le ponemos el método POST nos devuelve la página: POST /mostrar/victor HTTP/1.1 Host: 127.0.0.1:5000 User-Agent: insomnia/2022.2.1 Accept: / Content-Length: 0 ya que le hemos implementado que contenga los dos methods. GET para recuperar info del servidor POST para enviar info al servidor Es lo que usaremos más adelante para enviar y recibir peticiones de nuestra aplicación.

6. TEMPLATES

Los templates en flask se usan con Jinja para realizar plantillas de html para saber más sobre jinja PULSA AQUÍ. Para ello vamos a importar rendering_templates que en vez de lleavr una cadena como hasta ahora, traeremos un documento html. En la carpeta de nuestro proyecto crearemos una carpeta llamada templates. dentro de esta carpeta creamos nuestro archivo html en este caso lo llamamos mostrar.html como la función anterior. Por el otro lado en la función lo vamos a dejar de esta manera:

```
@app.route('/mostrar/<nombre>', methods=['GET', 'POST'])
def mostrar_nombre(nombre):
    return render_template('mostrar.html', nombre_llave=nombre)
```

render_template = es para que flask busque en la carpeta templates el archivo html en este caso 'mostrar.html' render lleva dos parámetros el archivo y la variable quye queremos mostrar nombre_llave es la variable del archivo html y el otro nombre es el del archivo app.py, que por convención sería el mismo nombre pero para aclararnos ahora le ponemos nombrellave el html quedaría tal que así:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>mostrar nombre</title>
</head>
<body>
    <h1>MOSTRAR VALORES </h1>
    <p>Tu nombre es: {{nombre_llave}}</p>
    <h1>{{nombre_llave}}</h1>
</body>
</html>
```

como vemos le pasamos la variable entre dos llaves {{nombre_llave}}, así que en el navegador le pasamos la url localhost:5000/mostrar/victor y nos mostrará el resultado.

7. URL BUILDING

Redireccionamientos de urls.

```
@app.route('/redireccionar')
def redireccionar():
    return redirect(url_for('inicio'))
```

Ponemos una nueva route llamada /redireccionar, para ello definimos la función con un return que nos devuelve el método redirect para ello importamos

```
from werkzeug.utils import redirect
```

y luego con el método url_for de la biblio de flask le damos el nombre de la función a la que queremos redireccionar 'inicio' solo el nombre sin parétesis. Volvemos a nuestro html y le ponemos un link para redireccionar a / inicio:

```
 <div>
        <a href="/">INICIO</a>
    </div>

 <div>
        <a href="{{url_for('inicio')}}">INICIO</a>
    </div>

<div>
        <a href="{{url_for('mostrar', edad=30)}}">INICIO</a>
    </div>
```

De esta manera ya tendríamos el redireccionamiento hecho, una vez que pulsamos el link nos lleva a donde le digamos. Podríamos realizar la misma función con url_for redireccionando directamente a la función inicio.

8. MANEJO DE ERRORES

Para esto en Flask tenemos el método abort:

```
@app.route('/salir')
def salir():
    return abort(404)
```

Si vamos a /salir nos da not found, página no encontrada. La podemos personalizar con un template en html.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ERROR 404</title>
</head>
<body>
    error 404
    <div>
        {{error}}
    </div>
    <div>
        <a href="{{url_for('redireccionar')}}">INICIO</a>
    </div>
</body>
</html>

@app.errorhandler(404)
def pagina_no_encontrada(error):
    return render_template('error404.html', error=error), error
```

En python podemos regresar varios valores sin paréntesis de ahi el , error. Si accedemos a /error nos da como ya es sabido nuestro error 404 not found, podemos poner el enlace al inicio debajo.

9. USO DE JSON Y FLASK

Sobre las respuestas en Flask podemos responder con un objeto, con un String o con un diccionario. Con un dic en Flask directamente se hace con jsonify, esto es responder con respuestas de tipo json, que es lo más común. Esto es lo que se conoce con REST representation state transfer, esta respuesta de tipo json la van a consumir otros sistemas, y en particular frameworks de tipo front end como angular, React o VueJs, cualquiera de estos puede consumir esta info de tipo json. Por lo que vamos a crear desde Flask es nuestro back-end, por lo que hay que definir nuestros paths de una manera, /api (aplication normal interface)

```
@app.route("/api/mostrar/<nombre>", ,methods=['GET', POST])
def mostrar_json(nombre):
```

```
  valores = {'nombre': nombre, 'metodo_http': request.method}
  return valores
```

JSON (JavaScrip Object Notation) es texto escrito en json y debe de seguir ciertas reglas para responder en json y no en html. llaves clave:valor = {"name" : "John"} // los objetos en JavaScript usan llaves. Podemos usar todo tipo de Values:

1. String
2. number
3. objet (JSON objets)
4. arrays
5. boolean
6. null

Ejemplo:

```
var person = {name : "John", age: 31, city: "EEUU"};
```

Del route /api/mostrar/ nos da el siguiente json:

```
{
  "metodo_http": "GET",
  "nombre": "victor"
}
```

Le podemos dar methods de GET o POST o lo que veamos.

10. **SESIONES EN FLASK

COOKIES es un archivo que se almacena en el servidor del lado del cliente y que contiene cierta información en forma de llave-valor y nos va dejar almacenar esa información durante dirferentes peticiones. Concepto de sesiones en flask:ç vamos a crear una variable secreta a través de app.secret_key por lo que será más seguro ya que la cookie podrá ser leida pero no modificada. vamos a app.py:

```
app.secret_key = 'mi_llave_secreta'

def inicio():
    if 'username' in session:
        return 'El user ya hizo loggin'
    return 'No ha hecho loggin'
```

Así cuando vamos al path de inicio nos dice que el user no ha hecho loggin por lo que vamos a agregar el path de loggin y realizar un formulario en un nuevo template de html

```
@app.route('login')
def login():
    return render_template(login.html)

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login</title>
</head>
<body>
    <h1>LOGIN</h1>
    <form action=""></form>
</body>
</html>
```

Nos quedaría un formulario para hacer login. Vamos a procesar esas peticiones de tipo POST ya que solicita al servidor que valide el usuario y contraseña:

```
@app.route("/", methods=['GET', 'POST'])
def inicio():
    if 'username' in session:
          return f'El user {session["username"]} ya hizo loggin'
```

```
    return 'No ha hecho loggin'


@app.route('/login', methods=['GET', 'POST'])
def login():
    # no se va a validar el user
    if request.method == 'POST':
        session['username'] = request.form['username']
        # agregamos el usuario a la seession, sería como poner
        # usuario = request.form['username'] (form es nuestro formulario)
        # session['username'] = usuario
        return redirect(url_for('inicio'))
    return render_template('login.html')
```

Aquí saldría el usuario ya ha hecho login, ya que se ha rellenado el form del login.html, y se ha validado por la funcion login, remirtiendote a la pagina inicio con redirect, o donde quieras. Arriba con el session["username"] nos da el nombre que pongamso en el formulario de login.

11. ELIMINAR USUARIO DE LA SESIÓN.

Es muy facil con session.pop():

```
@app.route('/logout')
def logout():
    session.pop('username')
    return redirect(url_for('inicio'))
```

Con pop lo saca de la session y con redirect lo vuelve a mandar inicio o donde quieras.