Flask

Introducción: Un marco web minimalista de Python

Flask es un micro marco web de Python que permite construir aplicaciones web de manera rápida y sencilla. Es conocido por su enfoque minimalista y su facilidad de uso. Flask está diseñado para ser ligero y flexible, lo que lo convierte en una excelente opción para crear aplicaciones web desde pequeños proyectos hasta aplicaciones más complejas.

Arquitectura Cliente-Servidor y el Protocolo HTTP

Antes de sumergirnos en Flask, es importante comprender la arquitectura cliente-servidor y el Protocolo de Transferencia de Hipertexto (HTTP). En esta arquitectura, los clientes (navegadores web u otras aplicaciones) hacen solicitudes a través de Internet a un servidor, que responde proporcionando datos o recursos solicitados. HTTP es el protocolo de comunicación que permite esta interacción, y consta de métodos de solicitud (GET, POST, PUT, DELETE, etc.) y códigos de estado (200, 404, 500, etc.) que indican el resultado de la solicitud.

¿Qué es Flask y cómo funciona?

Flask es una biblioteca de Python que se basa en el patrón de diseño WSGI (Web Server Gateway Interface). Este patrón permite que las aplicaciones web de Python se comuniquen con servidores web, como el popular servidor WSGI, Gunicorn o uWSGI.

En términos más simples, Flask recibe las solicitudes HTTP del cliente, procesa la lógica de la aplicación y devuelve una respuesta al cliente. La lógica de la aplicación puede incluir consultar bases de datos, procesar datos, autenticar usuarios y más. Flask es capaz de hacer esto gracias a su enrutador y sus vistas.

Enrutamiento en Flask

El enrutamiento en Flask es una parte esencial para manejar las solicitudes entrantes. Utilizando decoradores, podemos asignar funciones de Python a rutas específicas en la aplicación. Por ejemplo, si un cliente hace una solicitud HTTP GET a "/inicio", podemos asignar una función que maneje esa ruta y devuelva una página de inicio.

```
from flask import Flask
```

```
app = Flask(__name__)

@app.route('/')
def inicio():
    return '¡Bienvenido a la página de inicio de mi aplicación!'

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

1. `from flask import Flask`: Importamos la clase `Flask` del paquete `flask`. `Flask` es la clase principal de Flask y se utiliza para crear una instancia de la aplicación web.
2. `app = Flask(__name__)`: Creamos una instancia de la clase `Flask` y la asignamos a la variable `app`. `__name__` es una variable especial de Python que representa el nombre del módulo actual. Cuando ejecutamos este archivo directamente, `__name__` será igual a `"__main__"`, lo que indica que es el punto de entrada principal del programa. Al pasar `__name__` como argumento a `Flask`, le estamos diciendo a Flask que esta aplicación es el punto de entrada principal.
3. `@app.route('/')`: Esto es un decorador de Flask, que indica que la función `inicio()` será la manejadora de las solicitudes HTTP que llegan a la ruta `'/'`, es decir, la página de inicio de nuestra aplicación.
4. `def inicio():`: Definimos la función `inicio()`, que manejará las solicitudes HTTP en la ruta `'/'`. Cuando un cliente (como un navegador web) solicite la página de inicio de nuestra aplicación, Flask ejecutará esta función.
5. `return '¡Bienvenido a la página de inicio de mi aplicación!'`: En esta línea, simplemente devolvemos una cadena que será la respuesta que Flask enviará al cliente cuando se acceda a la página de inicio. En este caso, será el mensaje "¡Bienvenido a la página de inicio de mi aplicación!".
6. `if __name__ == '__main__':`: Esta es una construcción condicional de Python que verifica si el módulo se está ejecutando como programa principal (es decir, no se está importando como un módulo en otro programa). Si es el programa principal, el siguiente bloque de código se ejecutará.
7. `app.run()`: Cuando llamamos a `app.run()`, Flask ejecutará el servidor de desarrollo incorporado y la aplicación comenzará a escuchar las solicitudes entrantes en el puerto 5000 de forma predeterminada (puedes especificar otro puerto si lo deseas).

Esto significa que nuestra aplicación ahora está en funcionamiento y puede manejar las solicitudes HTTP.

En resumen, este código crea una aplicación web Flask muy simple que tiene una sola ruta ("/") que devuelve un mensaje de bienvenida cuando se accede a ella. Cuando ejecutas este archivo directamente, Flask activa el servidor de desarrollo y pone en funcionamiento la aplicación, permitiéndote acceder a la página de inicio a través de tu navegador web en `http://127.0.0.1:5000/` (o `http://localhost:5000/`), donde verás el mensaje "¡Bienvenido a la página de inicio de mi aplicación!".

Vistas y Plantillas

En Flask, las vistas son funciones que se asocian con rutas y se utilizan para generar una respuesta para el cliente. Esta respuesta puede ser HTML, JSON u otros formatos, dependiendo de lo que la aplicación necesite devolver.

Las plantillas son archivos que permiten separar el contenido estático (HTML) del contenido dinámico (datos de la aplicación). Flask utiliza el motor de plantillas Jinja2, que permite generar páginas web dinámicas utilizando datos de Python.

Interacción con bases de datos

Las aplicaciones web a menudo requieren interactuar con bases de datos para almacenar y recuperar información. Flask es compatible con una variedad de bases de datos, como SQLite, MySQL, PostgreSQL y más. Para interactuar con bases de datos, se utilizan extensiones de Flask o bibliotecas de terceros.

Middleware y Extensiones

Flask también admite el uso de middleware y extensiones, que son componentes adicionales que agregan funcionalidades específicas a la aplicación. El middleware puede utilizarse para realizar tareas como autenticación, registro de solicitudes o compresión de respuestas. Las extensiones, por otro lado, pueden ofrecer soporte para bases de datos, autenticación de usuarios, API RESTful, entre otros.

Despliegue de aplicaciones Flask

Una vez que hemos creado nuestra aplicación Flask, necesitamos desplegarla para que pueda ser accesible en la web. Hay varias opciones de despliegue disponibles, como utilizar servidores WSGI, servicios de alojamiento en la nube, contenedores Docker, entre otros.

Utilización:

Paso 1: Configurar el entorno

Antes de comenzar, asegúrate de tener Python instalado en tu sistema. Si no lo tienes, descárgalo e instálalo desde el sitio web oficial de Python.

Luego, crea un nuevo directorio para el proyecto y accede a él a través de la línea de comandos. Es recomendable utilizar un entorno virtual para mantener las dependencias del proyecto aisladas. Puedes crear un entorno virtual usando `venv` (disponible en Python 3) o `virtualenv`.

```
# Crea y activa un entorno virtual con venv
python3 -m venv myenv
source myenv/bin/activate

# O con virtualenv
virtualenv myenv
source env/Scripts/activate
```

Paso 2: Instalar Flask y Requests

En la instalación del proyecto, solo necesitas instalar Flask y las dependencias específicas de tu proyecto.

Para corregir, simplemente instala Flask:

```
pip install Flask
```

Si en tu proyecto deseas utilizar `requests` para interactuar con la API de OpenWeatherMap u otras APIs, puedes instalarlo por separado en tu entorno virtual:

```

pip install requests
```

Recuerda que cada vez que agregues una nueva biblioteca o dependencia a tu proyecto, es recomendable mantener un archivo `requirements.txt` donde se enumeren todas las dependencias junto con sus versiones para facilitar la reproducción del entorno en diferentes máquinas y colaboración con otros desarrolladores. Puedes generar el archivo `requirements.txt` con el siguiente comando:

```
pip freeze > requirements.txt
```

Y luego, al configurar un nuevo entorno virtual, puedes instalar todas las dependencias desde el archivo `requirements.txt` con el siguiente comando:

```
pip install -r requirements.txt
```

Paso 3: Obtener una clave de API de OpenWeatherMap

Para utilizar la API de OpenWeatherMap, necesitaremos obtener una clave de API gratuita. Ve al sitio web de OpenWeatherMap, crea una cuenta (si aún no tienes una) y sigue las instrucciones para obtener tu clave de API.

Paso 4: Estructura del proyecto

Antes de continuar, echemos un vistazo a la estructura del proyecto que crearemos:

```
weather_app/
|-- app.py
|-- templates/
|   |-- index.html
|-- static/
|   |-- styles.css
```

- `app.py`: Este será el archivo principal donde escribiremos nuestra aplicación Flask.
- `templates`: Aquí almacenaremos nuestras plantillas HTML.
- `static`: En este directorio, colocaremos nuestros archivos estáticos, como hojas de estilo CSS.

Paso 5: Crear la aplicación Flask

Comencemos a escribir el código para nuestra aplicación Flask en el archivo `app.py`.

```
# app.py
import os
import requests
from flask import Flask, render_template, request

app = Flask(__name__)
```

```
# Configuración de la clave de API
api_key = os.environ.get('OPENWEATHERMAP_API_KEY')
if not api_key:
    raise ValueError("No se encontró la clave de API de OpenWeatherMap. "
                     "Asegúrate de configurar la variable de entorno 'OPENWEATHERMAP_API_K
EY'.")

# Ruta de inicio
@app.route('/')
def index():
    return render_template('index.html')

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

1. **Configuración de la clave de API**: En esta parte del código, se está intentando obtener la clave de API de OpenWeatherMap desde una variable de entorno llamada `'OPENWEATHERMAP_API_KEY'`. La función `os.environ.get()` se utiliza para acceder a las variables de entorno del sistema operativo. Si la variable de entorno `'OPENWEATHERMAP_API_KEY'` está configurada en el sistema, se asignará el valor de la clave a la variable `api_key`. Si la variable de entorno no está configurada o no se encuentra, la función `os.environ.get()` devolverá `None`, y la condición `if not api_key:` será verdadera.
2. **Excepción si no se encuentra la clave de API**: Si la clave de API de OpenWeatherMap no se encuentra (es decir, `api_key` es `None`), se genera una excepción utilizando `raise ValueError`. Esto mostrará un mensaje de error indicando que la clave de API no ha sido configurada correctamente en las variables de entorno del sistema.
3. **Ruta de inicio**: En esta parte del código, se define una ruta para la página de inicio de la aplicación. Cuando un cliente acceda a la ruta raíz `'/'`, Flask ejecutará la función de vista `index()`. La función de vista simplemente devuelve el resultado de la función `render_template()`, que renderizará la plantilla `'index.html'`.

En resumen, este fragmento de código realiza lo siguiente:

- Intenta obtener la clave de API de OpenWeatherMap desde las variables de entorno del sistema.

- Si la clave de API no se encuentra, se generará una excepción para indicar que la configuración es incorrecta.
- Define una ruta para la página de inicio de la aplicación, que muestra la plantilla `'index.html'`.

Paso 6: Crear la plantilla HTML

En la carpeta `templates`, crearemos nuestro archivo HTML para mostrar la interfaz web.

```
<!-- templates/index.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Weather App</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body>
    <div class="container">
        <h1>Weather App</h1>
        <form action="/" method="post">
            <input type="text" name="city" placeholder="Ingrese una ciudad" required>
            <button type="submit">Buscar</button>
        </form>
        {% if weather_data %}
        <div class="weather-info">
            <h2>{{ weather_data['city'] }}</h2>
            <p>{{ weather_data['description'] }}</p>
            <p>Temperatura: {{ weather_data['temperature'] }}°C</p>
            <p>Humedad: {{ weather_data['humidity'] }}%</p>
            <p>Viento: {{ weather_data['wind_speed'] }} km/h</p>
        </div>
        {% endif %}
    </div>
</body>
</html>
```

```
{% if weather_data %}
```

- Esta línea utiliza la estructura de control `{% if %}` de Jinja2 para verificar si la variable `weather_data` tiene algún valor. La variable `weather_data` debe estar disponible en el contexto de la plantilla, y se espera que contenga datos climáticos

proporcionados por la API de OpenWeatherMap. Si `weather_data` tiene un valor, se ejecutará el bloque de código dentro del `{% if %}`, de lo contrario, se ignorará.

```
<div class="weather-info">
```

- Aquí, se define un elemento `<div>` con la clase de CSS "weather-info". Este elemento contiene la información del clima que se mostrará en la página web.

```
<h2>{{ weather_data['city'] }}</h2>
```

- Esta línea muestra el nombre de la ciudad en un encabezado (`<h2>`) utilizando la variable `weather_data['city']`, que debe contener el nombre de la ciudad proporcionado por la API de OpenWeatherMap.

```
<p>{{ weather_data['description'] }}</p>
```

- Aquí se muestra la descripción del clima en un párrafo (`<p>`) utilizando la variable `weather_data['description']`, que debe contener la descripción del clima proporcionada por la API de OpenWeatherMap.

```
<p>Temperatura: {{ weather_data['temperature'] }}°C</p>
```

- Esta línea muestra la temperatura en grados Celsius en un párrafo (`<p>`) utilizando la variable `weather_data['temperature']`, que debe contener la temperatura proporcionada por la API de OpenWeatherMap.

```
<p>Humedad: {{ weather_data['humidity'] }}%</p>
```

- Aquí se muestra el porcentaje de humedad en un párrafo (`<p>`) utilizando la variable `weather_data['humidity']`, que debe contener la humedad proporcionada por

la API de OpenWeatherMap.

```
<p>Viento: {{ weather_data['wind_speed'] }} km/h</p>
```

- Esta línea muestra la velocidad del viento en kilómetros por hora en un párrafo (`<p>`) utilizando la variable `weather_data['wind_speed']`, que debe contener la velocidad del viento proporcionada por la API de OpenWeatherMap.

```
{% endif %}
```

- Esta línea cierra el bloque `{% if %}`. Si `weather_data` tenía un valor y se ejecutó el bloque de código anterior, aquí se cierra ese bloque.

Paso 7: Estilo CSS

En la carpeta `static`, crearemos un archivo `styles.css` para dar estilo a nuestra aplicación.

```
/* static/styles.css */
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #f1f1f1;
}

.container {
    max-width: 600px;
    margin: 0 auto;
    padding: 20px;
    background-color: #fff;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}

h1 {
    text-align: center;
    margin-bottom: 20px;
}

form {
    display: flex;
    justify-content: center;
```

```
    align-items: center;
    margin-bottom: 20px;
}

input[type="text"] {
    padding: 10px;
    width: 60%;
    border: 1px solid #ccc;
    border-radius: 4px;
}

button {
    padding: 10px 20px;
    background-color: #4CAF50;
    color: white;
    border: none;
    border-radius: 4px;
    cursor: pointer;
}

.weather-info {
    padding: 10px;
    background-color: #f9f9f9;
    border: 1px solid #ddd;
    border-radius: 4px;
}

.weather-info h2 {
    margin-top: 0;
}
```

Paso 8: Obtener datos climáticos de OpenWeatherMap

Ahora que tenemos nuestra interfaz web, debemos obtener los datos climáticos de la API de OpenWeatherMap cuando el usuario realice una búsqueda. Modificaremos la función `index` en `app.py` para que maneje las solicitudes POST y realice la solicitud a la API.

```
# app.py
# ... (código anterior)

# Ruta de inicio
@app.route('/', methods=['GET', 'POST'])
def index():
    weather_data = None

    if request.method == 'POST':
```

```
        city = request.form['city']
        weather_data = get_weather_data(city)

    return render_template('index.html', weather_data=weather_data)

# Función para obtener datos climáticos de OpenWeatherMap
def get_weather_data(city):
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&units=metric&appid={ap
i_key}"
    response = requests.get(url)
    data = response.json()

    if data['cod'] == 200:
        weather = {
            'city': data['name'],
            'description': data['weather'][0]['description'],
            'temperature': data['main']['temp'],
            'humidity': data['main']['humidity'],
            'wind_speed': data['wind']['speed'],
        }
        return weather

    return None

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

Paso 9: Ejecutar la aplicación

¡Es hora de probar nuestra aplicación web! Ejecuta el servidor de desarrollo de Flask para ver la aplicación en acción.

```
python app.py
```

Abre tu navegador web e ingresa `http://127.0.0.1:5000/`. Verás la página de inicio de la aplicación, donde puedes ingresar el nombre de una ciudad y obtener los datos climáticos en tiempo real utilizando la API de OpenWeatherMap.

Utilizacion para crear una REST Api:

Paso 1: Configuración del entorno

Si aún no has creado el entorno virtual y configurado Flask y Requests, sigue los pasos mencionados anteriormente en la parte teórica.

Paso 2: Estructura del proyecto

Vamos a crear una nueva estructura de proyecto para nuestra REST API.

- `app.py`: Este archivo será el punto de entrada de nuestra aplicación Flask y contendrá la configuración y la inicialización de la aplicación.
- `api.py`: Aquí definiremos las rutas y las funciones que manejarán las solicitudes HTTP y proporcionarán los datos climáticos a través de la API RESTful.
- `utils.py`: En este archivo, escribiremos funciones de utilidad que interactuarán con la API de OpenWeatherMap.

Paso 3: Configurar la aplicación Flask

En el archivo `app.py`, configuraremos y configuraremos nuestra aplicación Flask.

```
# app.py
from flask import Flask

app = Flask(__name__)

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

Paso 4: Definir rutas en la API

En el archivo `api.py`, definiremos las rutas de nuestra REST API y las funciones que manejarán las solicitudes HTTP.

```
# api.py
from flask import jsonify, request
from app import app
from utils import get_weather_data

@app.route('/api/weather', methods=['GET'])
def get_weather():
    city = request.args.get('city')

    if not city:
        return jsonify({'error': 'Debe proporcionar el nombre de una ciudad.'}), 400

    weather_data = get_weather_data(city)

    if not weather_data:
        return jsonify({'error': 'No se pudo obtener los datos climáticos para la ciudad p
```

```
roporcionada.'}), 404

    return jsonify(weather_data), 200

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

Paso 5: Funciones de utilidad

En el archivo `utils.py`, escribiremos la función de utilidad que interactuará con la API de OpenWeatherMap y obtendrá los datos climáticos.

```
# utils.py
import os
import requests

def get_weather_data(city):
    api_key = os.environ.get('OPENWEATHERMAP_API_KEY')

    if not api_key:
        raise ValueError("No se encontró la clave de API de OpenWeatherMap. "
                         "Asegúrate de configurar la variable de entorno 'OPENWEATHERMAP_A
PI_KEY'.")

    url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&units=metric&appid={ap
i_key}"
    response = requests.get(url)
    data = response.json()

    if data['cod'] == 200:
        weather = {
            'city': data['name'],
            'description': data['weather'][0]['description'],
            'temperature': data['main']['temp'],
            'humidity': data['main']['humidity'],
            'wind_speed': data['wind']['speed'],
        }
        return weather

    return None
```

Paso 6: Ejecutar la API

¡Nuestra REST API está lista! Ahora podemos ejecutarla y probarla.

```
python app.py
```

Paso 7: Prueba de la API

Podemos probar nuestra REST API utilizando herramientas como `curl`, `httpie` o un cliente REST como Postman. Aquí hay un ejemplo de cómo probar la API con `httpie`:

```
# Obtener datos climáticos para una ciudad específica (por ejemplo, Londres)
http GET http://127.0.0.1:5000/api/weather?city=London
```

Deberías recibir una respuesta JSON con los datos climáticos para la ciudad de Londres o cualquier otra ciudad que hayas proporcionado.

Ejemplos de uso:

1. https://github.com/Jcibernet/flaskclass
2. https://github.com/Jcibernet/meta-connection