



Python: Introducción

FLASK

Esta clase

- Más operaciones en flask



Recordemos

Entonces, un servidor **web** es simplemente un proceso ejecutado en un computador, que se encuentra escuchando a un puerto correspondiente a la IP del equipo dentro de la red actual, y que se enfoca específicamente en proveer acceso a **páginas** o **servicios web REst**. Un servicio **REst** es un servicio que utiliza el protocolo **HTTP** a manera de transmitir información mediante una estructura estandarizada.

```
GET https://miservicio.com/datos/frutas
```

```
Respuesta:
```

```
200 OK
```

```
[  
  {  
    "nombre": "manzana",  
    "color": "verde"  
  },  
  {  
    "nombre": "banana",  
    "color": "amarilla"  
  },  
]
```

```
POST https://miservicio.com/datos/crear_fruta
```

```
Cuerpo:
```

```
{  
  "nombre": "mango",  
  "color": "rojo"  
}
```

```
Respuesta:
```

```
201 Creado
```



[Welcome to Flask — Flask Documentation \(2.1.x\)](https://palletsprojects.com/en/2.1.x/)
[\(palletsprojects.com\)](https://palletsprojects.com/)

Un mini-framework para servicios y aplicaciones web en Python!

Enfocado en la simpleza, muy fácil de utilizar y de entender. ¡Permite por defecto ofrecer aplicaciones web muy rápidamente!



Recordemos que podemos declarar puntos de acceso de nuestro servicio que apliquen a los diferentes verbos de HTTP.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/enviar_json', methods=['POST'])
def procesar_json():
    tipo = request.headers.get('Content-Type')
    json = request.json
    # Hacer algo con el objeto JSON
    return json

@app.route("/", methods=['GET'])
def hola_mundo():
    return "<p>Hola a todos!</p>"
```



Flask utiliza decoradores para describir como acceder a un recurso solicitado por un cliente HTTP. En este caso, permite por ejemplo crear configuraciones muy interesantes.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
@app.route('/inicio')
def procesar_informacion():
    return "Hola"
```



Reglas variables permiten definir rutas dinámicas en el contexto de las operaciones del servicio.

Por ejemplo, la captura de variables como parte de la URL.

```
from flask import Flask
app = Flask(__name__)

@app.route('/hola/<nombre>')
def hola_nombre(nombre):
    return 'Hello %s!' % nombre
```

En este caso podemos definir la sintaxis con <nombre>, y nombre será inyectado a la función como parámetro.



La captura de variables puede incluir cierta validación por defecto para los tipos esperados, por ejemplo, cuando queramos un identificador que siempre sea un número.

```
from flask import Flask
app = Flask(__name__)

@app.route('/blog/<int:publicacionId>')
def ver_publicacion(publicacionId):
    return 'Blog Publicacion %d' % publicacionId
```

El framework acepta otros tipos de conversiones, tales como string, int, float, path, uuid.

[Quickstart — Flask Documentation \(2.0.x\) \(palletsprojects.com\)](https://palletsprojects.com/en/2.0.x/quickstart/)



Flask define reglas de especificidad al responder a una solicitud hecha por un cliente. En caso de que dos o más reglas sean aplicables a una solicitud, el framework aplicará la más específica a tal solicitud.

```
from flask import Flask
app = Flask(__name__)

@app.route('/blog')
def ver_blog():
    return 'Blog'

@app.route('/blog/<int:publicacionId>')
def ver_publicacion(publicacionId):
    return 'Blog Publicacion %d' % publicacionId
```



En muchos escenarios, es necesario responder al cliente con la dirección de otro recurso dentro del mismo servicio. La utilidad `url_for` permite realizar esto de manera sencilla. La utilidad `redirect`, responde con un código de redirección HTTP para que el cliente redirija al recurso automáticamente.

```
from flask import Flask, redirect, url_for
app = Flask(__name__)

@app.route('/admin')
def hola_admin():
    return 'hola Admin'

@app.route('/invitado/<invitado>')
def hola_invitado(invitado):
    return 'hola %s invitado' % invitado

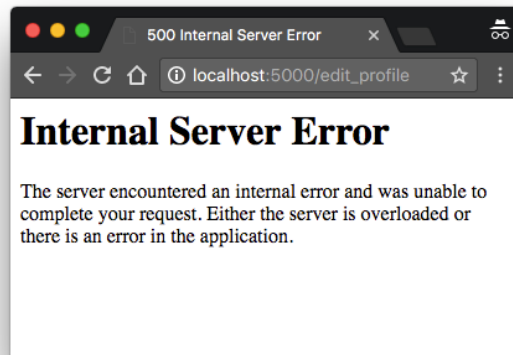
@app.route('/usuario/<nombre>')
def hola_usuario(nombre):
    if nombre == 'admin':
        return redirect(url_for('hola_admin'))
    else:
        return redirect(url_for('hola_invitado', invitado=nombre))
```

Gestión de Errores



Cuando un error ocurre en una aplicación **flask**, en caso de no ser controlado, lo que la hace es fallar y la conexión HTTP produce un error 5xx que existe un problema con el servidor.

Por ejemplo, si producimos una excepción intencionalmente, lo que veremos es:



```
(venv) $ python -m flask run
* Serving Flask app "blog"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
[2021-06-14 22:40:02,027] ERROR in app: Exception on /edit_profile [GET]
BaseException: Error forzado
```

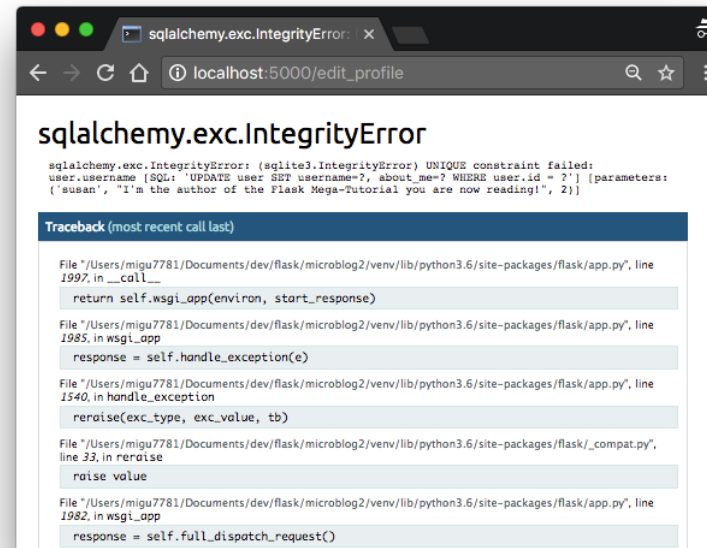


Sin embargo, lo presentado en la pantalla no es tan detallado y existen errores muy complejos que suceden en el contexto de aplicaciones reales.

Flask ofrece la flexibilidad de activar un modo para análisis de errores mediante la bandera `--debug`, que en modo **development** activa el debugger.

```
(venv) proyecto $ python -m flask run --debug
* Serving Flask app 'app.py' (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 118-204-854
```

En este caso, **flask** devuelve una página más detallada con el rastro del error. El cuál puede ser muy útil para determinar que problemas ocurren.



```
sqlalchemy.exc.IntegrityError: (sqlite3.IntegrityError) UNIQUE constraint failed:
user.username [SQL: 'UPDATE user SET username=?, about_me=? WHERE user.id = ?'] [parameters:
{'susan', 'I'm the author of the Flask Mega-Tutorial you are now reading!', 2}]

Traceback (most recent call last)
File "/Users/migu7781/Documents/dev/flask/microblog2/venv/lib/python3.6/site-packages/flask/app.py", line
1997, in __call__
    return self.wsgi_app(environ, start_response)
File "/Users/migu7781/Documents/dev/flask/microblog2/venv/lib/python3.6/site-packages/flask/app.py", line
1985, in wsgi_app
    response = self.handle_exception(e)
File "/Users/migu7781/Documents/dev/flask/microblog2/venv/lib/python3.6/site-packages/flask/app.py", line
1540, in handle_exception
    reraise(exc_type, exc_value, tb)
File "/Users/migu7781/Documents/dev/flask/microblog2/venv/lib/python3.6/site-packages/flask/_compat.py",
line 33, in reraise
    raise value
File "/Users/migu7781/Documents/dev/flask/microblog2/venv/lib/python3.6/site-packages/flask/app.py", line
1982, in wsgi_app
    response = self.full_dispatch_request()
```



Sin embargo, es importante recalcar que este modo solo es útil al desarrollar una aplicación, al publicarla uno debe usar el modo de **producción**.

Si uno despliega una aplicación a usuarios finales en modo de desarrollo, es muy posible que información que debe mantenerse escondida sea descubrible por el resto del mundo.

Por ejemplo, estos rastros de errores pueden incluir secretos o claves que no queremos que se descubran por accidente.



Recordemos que al final del día flask implementa un servicio HTTP. Y nuestros puntos de acceso pueden responder con errores HTTP de manera dinámica.

Adicionalmente el framework responde con estos errores automáticamente en caso de solicitarse recursos inexistentes, o cuando hay un error que no queremos.

Por defecto, se utilizan plantillas básicas que no den mayor información.



Mediante el uso del decorador `errorhandler`, uno puede definir funciones que devuelvan errores a gusto del usuario.

Por ejemplo, podemos usar el sistema de plantillas, o hacer referencia a una página HTML que esté guardada en el disco, o simplemente devolver mensajes personalizados.

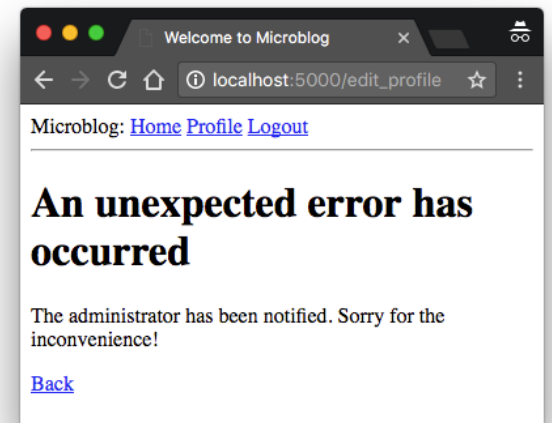
```
from flask import render_template
from app import app

@app.errorhandler(404)
def no_se_encontro(error):
    return render_template('404.html'), 404

@app.errorhandler(500)
def error_interno(error):
    return render_template('500.html'), 500
```

500.html

```
{% block content %}
    <h1>An unexpected error has
occurred</h1>
    <p>The administrator has been
notified. Sorry for the
inconvenience!</p>
    <p><a href="{{ url_for('index')
}}">Back</a></p>
{% endblock %}
```





El protocolo HTTP define decenas de códigos con sus significados específicos.

Lo recomendado es usarlos claramente para asegurarnos que nuestro servicio funcione de manera adecuada dentro de un ecosistema integrado que asuma la implementación correcta de la especificación del protocolo.

Por ejemplo, nuestro servicio web puede devolver códigos 200 en todo momento, inclusive cuando haya errores, pero habrá aplicaciones que asuman que el código 401 va a ser recibido cuando haya un error de acceso con el fin de mostrar un mensaje útil.

En este caso es nuestro servicio el que está siendo inconsistente y no el cliente.

[HTTP response status codes - HTTP | MDN \(mozilla.org\)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)



Recordemos que Python permite generar excepciones personalizadas durante el flujo de trabajo.

Usando las rutinas comunes de lenguaje es posible capturarlos y responder con un código HTTP correspondiente.

```
from miproyecto.errores import AccesoRestringidoAUsuarios
import miproyecto.acceso_usuarios as usuarios
from flask import Flask, jsonify
app = Flask(__name__)

@app.route('/perfiles/<usuario>')
def obtener_perfil(usuario):
    try:
        perfil = usuarios.perfil(usuario)
        return jsonify(perfil), 200
    except AccesoRestringidoAUsuarios:
        return 'No tienes permiso para ver perfiles', 401
```

Recibiendo y
devolviendo
archivos



Recordemos que **flask** no es nada más que un mecanismo para proveer servicios **HTTP** que permitan:

1. Proveer un servidor de páginas web, que funcione como back-end para un sitio cualquiera.
2. Proveer un servicio web, cuyo objetivo es ejecutar rutinas de código remotamente y devolver o recibir datos de acuerdo a los requisitos.
3. Una combinación de ambas.



En cualquier caso de uso, es posible que sea necesario tener la habilidad de recibir y devolver archivos.

En una página web, devolver archivos estáticos que representen la página solicitada por el usuario. Por ejemplo, ficheros HTML, Javascript, CSS, etc.

En un servicio web, devolver archivos de datos, por ejemplo, un subconjunto de columnas de una base de datos en formato CSV de acuerdo a los parámetros especificados por la solicitud.

Archivos de imágenes dinámicos generados por librerías, etc.



El ejemplo más básico, es servir archivos estáticos que se encuentran en algún sitio de nuestro computador. La función `send_file` se encarga de realizar este proceso de manera integrada.

```
from flask import send_file
app = Flask(__name__)

@app.route('/descarga_imagen')
def descarga_imagen():
    return send_file(
        '/mi_directorio/python.jpg',
        download_name='python.jpg'
    )
```



Sin embargo, recordemos que podemos encontrar archivos de otras partes de la red y descargarlos mediante la utilidad **requests**, el siguiente ejemplo combina ambos conceptos para adquirir un archivo externo y devolverlo mediante nuestro servicio web.

```
import requests
from flask import Flask, send_file
from io import BytesIO
app = Flask(__name__)
@app.route('/descarga_imagen_externa')
def descarga_imagen_externa():
    resultado = requests.get("https://wwwnc.cdc.gov/travel/images/map-ecuador.png")
    contenidos_binarios = BytesIO(resultado.content)
    # Aqui podriamos procesarla y transformarla con otras librerias como OPENCV
    return send_file(
        contenidos_binarios,
        download_name='ecuador.png'
    )
```




send_file puede ser utilizada para devolver archivos estáticos que se utilicen para una página web, sin embargo, en este contexto queremos más dinamismo y no declarar una función que provea cada uno de los archivos de nuestra página.

¡Hay páginas que tienen cientos de archivos estáticos!

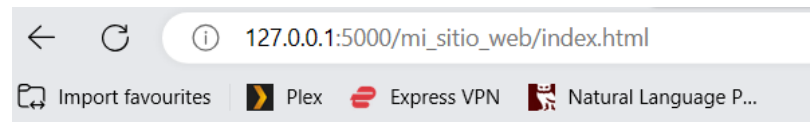
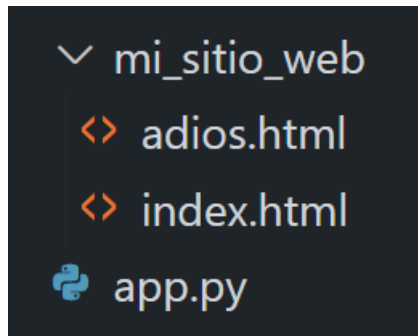
Flask ofrece automáticamente la capacidad de servir archivos estáticos ubicados en directorios de la aplicación.



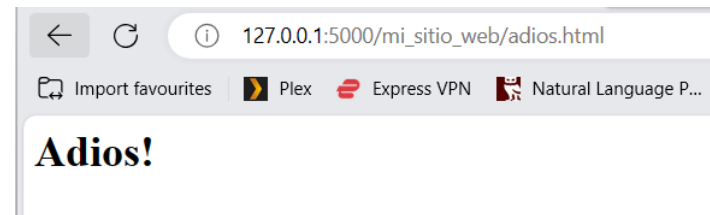
Al inicializar la aplicación uno puede especificar de qué directorio recoger los archivos estáticos, sin necesidad de especificar rutas para cada uno de ellos.

```
from flask import Flask

app = Flask(__name__, static_url_path="/mi_sitio_web", static_folder="mi_sitio_web")
```



Hola!

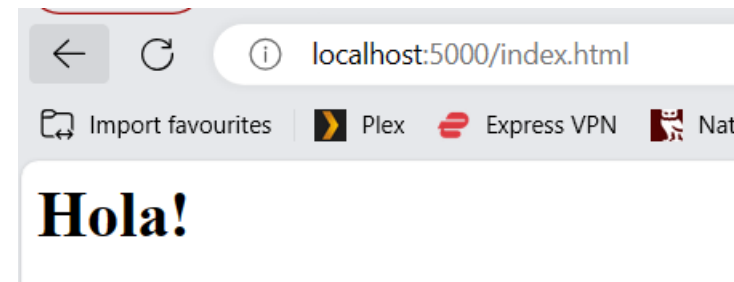
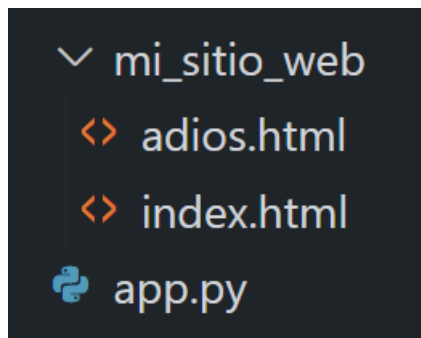




En caso de que querramos enteramente un sitio estático, que use flask como servidor, podemos mapear la carpeta a la ruta base.

```
from flask import Flask

app = Flask(__name__, static_url_path="/", static_folder="mi_sitio_web")
```





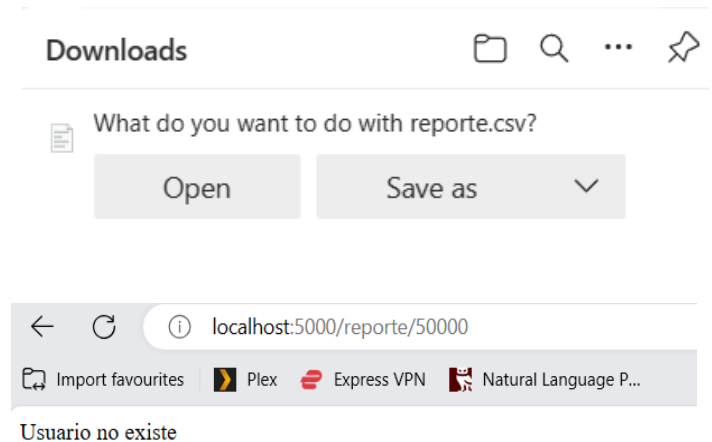
En el contexto de un servicio web, es posible que deseemos generar archivos dinámicos de datos, por ejemplo, un CSV a parte de un resultado de una base datos.

```
from flask import Flask, Response

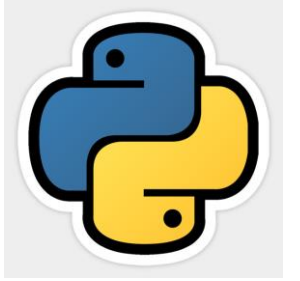
datos_usuarios = {
    "10000": ["Daniel", "Ortiz", "10101010"],
    "20000": ["Juan", "Morales", "1566010"],
    "30000": ["Ana", "Costa", "10144010"],
}

app = Flask(__name__)

@app.route("/reporte/<usuario>")
def generar_reporte(usuario):
    if usuario not in datos_usuarios:
        return "Usuario no existe", 404
    datos = datos_usuarios[usuario]
    tabla = {"nombre": datos[0], "cedula": datos[1], "cedula": datos[2]}
    csv = ",".join(tabla.keys()) + "\n" + ",".join(tabla.values())
    return Response(
        csv,
        mimetype="text/csv",
        headers={"Content-disposition": "attachment; filename=reporte.csv"},
    )
```



Siguiente clase



- Entendiendo Github y git
- Ejercicio interactivo de todos los conceptos de la clase usando Github Spaces