



# Python: Conceptos de Aplicaciones en Producción

---

REST Y SERVICIOS WEB



# Recordando

---

Mediante Python podemos ofrecer una multitud de diferentes servicios en muchos contextos.

Entre ellos, la habilidad de utilizar servidores y proveer servicios web.



# Concepto: Servidores

---

Con el pasar de los años, el mundo se ha convertido en una red de interconexiones entre múltiples entidades, personas y servicios.

Mediante el internet por ejemplo, podemos acceder a páginas web alojadas en múltiples lugares del mundo, o utilizar aplicaciones que internamente dependan de servicios externos para ofrecer distintas funcionalidades, como por ejemplo una aplicación de videollamadas.



# Concepto: Servidores

---

Un servidor es simplemente un programa de computadora que provee un servicio a otro programa de computadora, conocido como el cliente.

Estos programas pueden encontrarse en el mismo contexto, o como suele ser más común, en distintos nodos o máquinas.

Una aplicación puede funcionar como un servidor para ciertas aplicaciones, y a su vez como un cliente para otras.



# Servidores: Generalización

---

Un servidor de manera más general, es un proceso en un computador o máquina virtual, el cuál mediante protocolos de redes, se comunica con otros procesos.



# Concepto: Dirección IP

---

Simplemente una cadena de caracteres que identifica un computador usando el protocolo de internet (IP) en una red.

Esto permite que cada unidad o equipo en una red, y en el mundo pueda ser encontrado e identificado por otras máquinas en la misma red.



# Concepto: Puertos

---

Un puerto es un punto de acceso virtual en donde empiezan y terminan las conexiones de red dentro de un sistema operativo.

Por ejemplo, si nuestro computador expone la IP 1.2.3.4, pueden existir muchos procesos distintos que escuchan tráfico de red, por lo qué mediante puertos podemos diferenciarlos.

Un servidor ejecutándose en un computador suele escuchar uno o más puertos específicos.



# Ejemplo: Puertos

---

- David tiene un computador con IP **1.2.3.4** asignada por la red
- Juan tiene un computador dentro de la misma red, cuya IP es **4.5.6.7**
- David creó un servidor en su lenguaje favorito con el fin de proveer su fecha de cumpleaños a cualquier persona que tenga acceso a la red
- David inició el servidor escuchando al puerto **5000**

Juan puede ahora solicitar mediante la red la información provista por la IP **1.2.3.4**, en el puerto **5000**. Esto quiere decir que su computador mediante protocolos de red obtendrá la fecha de cumpleaños de David.





# Servidor web

---

Recordemos que mediante el internet uno puede hacer muchas cosas más aparte de navegar en páginas web, por ejemplo usar una aplicación de correo, jugar juegos en línea, etc.

Entonces es importante entender que un servidor puede tener muchas clases.

Un **servidor web** se enfoca enteramente en ofrecer acceso a páginas y servicios web, comúnmente mediante el protocolo **HTTP**.



# Página web

---

Recordemos que una página web no es nada más que una serie de archivos HTML, JavaScript y CSS que se encuentran ubicados en algún lugar del mundo.

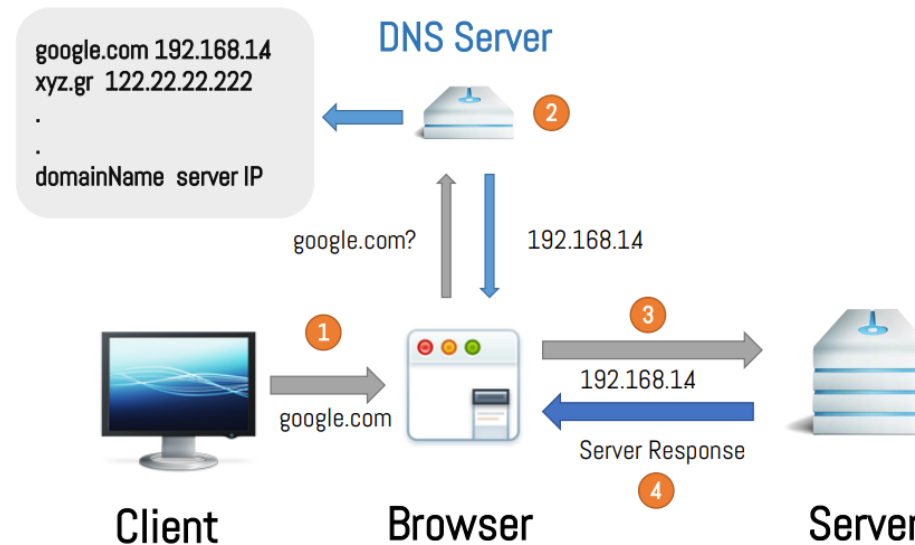
La manera en que estos archivos llegan a nuestro explorador web es mediante un servidor web.



# ¿Cómo accedemos a una página?

*Implícitamente nuestro explorador solicita el puerto 80 o 443 en la IP del servidor*

*El servidor DNS es una herramienta que traduce nombres de dominio a IPs.*



[Entendiendo qué es un servidor DNS - Blog \(godaddy.com\)](http://godaddy.com)



# Servicio web

---

Otra manera en la que podemos usar servidores es para exponer servicios web.

Los servicios web esencialmente permiten acceder a funciones de software mediante la red, usando los mismos protocolos que utilizamos para resolver páginas web.



# Servicio web

---

Cuando programamos, usamos funciones para operaciones repetitivas, o para obtener datos que permitan que nuestra aplicación cumpla su objetivo.

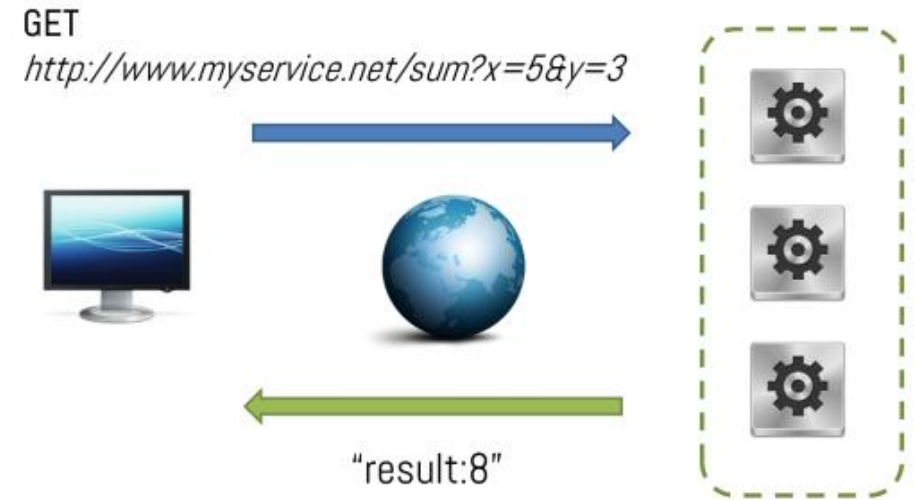
```
sumar(1, 2) > 3
```

```
obtener_cedula("Daniel") > 11000045678
```

Un servicio web es cualquier tipo de software disponible sobre la red y que utiliza un formato estandarizado de transmisión de mensajes (JSON, XML, etc.) con el fin de ejecutar funciones remotas.

# Servicio web

---



- ✓ Un servicio web se accede mediante una red
- ✓ Tiene una interfaz bien definida de funciones disponibles
- ✓ No requieren un lenguaje de programación específico

# Servicio web

---



Por lo tanto, es posible crear servicios web con nuestro lenguaje favorito, ya que casi todos proveen utilidades de red que permiten ofrecer las funcionalidades requeridas.

# Interfaces de programación de aplicaciones (API)

---



Una **API** es simplemente la especificación que describe como los componentes de software deben interactuar.

Permite que el software intercambie mensajes con otros productos sin necesidad de conocer su implementación.

En un servicio web es necesario definir una **API** clara que permita acceder a sus funcionalidades.

<https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>





# Servicios REST

---

Un estilo arquitectónico que permite definir servicios web.

REpresentational

State

Transfer

- Aprovecha la tecnología y protocolos de comunicación web para proveer acceso a funcionalidades de software en lugar de páginas.
- Por lo tanto existe interoperabilidad en sistemas conectados por el internet.



# Servicios REST

---

- Utiliza enteramente el protocolo HTTP como método de comunicación entre clientes y servidores.
- No retiene el estado de las operaciones, es decir cada operación es independiente y debe incluir toda la información requerida para la misma.
- Mediante el protocolo HTTP podemos transmitir información usando JSON, XML o cualquier estructura estandarizada, y recibir JSON, XML o cualquier estructura estandarizada.



# ¿Protocolos? ¿HTTP?

---

REST funciona enteramente aprovechando el protocolo HTTP como forma de comunicación entre procesos.

Un protocolo de red es simplemente una forma estandarizada mediante la cuál uno o más procesos pueden hablar entre sí.

*HTTP fue ideado originalmente para acceder a recursos web*

# HTTP

---



Funciona mediante una serie de métodos enviados a través de la red, que son interpretados de acuerdo a su significado estandarizado.

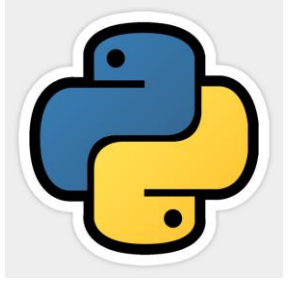
GET:

- Instruye al servidor que transmita la información identificada por la URL pasada
- Normalmente es de solo lectura, es decir, que una operación GET no debería modificar ninguna clase de información

Por ejemplo, GET se usa para solicitar una página web, esto no debería en ningún caso cambiar alguna información.

# HTTP

---



Funciona mediante una serie de métodos enviados a través de la red, que son interpretados de acuerdo a su significado estandarizado.

POST:

- Instruye al servidor que cree o actualice el recurso identificado por la URL pasada
- Las solicitudes adicionalmente incluyen un cuerpo de información que puede ser utilizado en el proceso de actualización o creación
- Por ejemplo, POST se puede usar para enviar los datos de una ficha llenada por un usuario, lo que permitirá crear un registro en alguna base de datos.

# HTTP

---



Funciona mediante una serie de métodos enviados a través de la red, que son interpretados de acuerdo a su significado estandarizado.

PUT:

- Instruye al servidor que cree o actualice el recurso identificado por la URL pasada
- Similar a POST pero representa una acción idempotente.



# HTTP

---

Funciona mediante una serie de métodos enviados a través de la red, que son interpretados de acuerdo a su significado estandarizado.

## DELETE:

- Instruye al servidor que elimine el recurso identificado por la URL pasada



# REST y HTTP

---

Entonces, REst como modelo arquitectónico de servicios web permite aprovechar HTTP para definir el significado de las operaciones provistas por la API del servicio.

Por ejemplo

- Se expone un punto de acceso GET para obtener una lista de nombres
- Se expone un punto de acceso POST para crear un estudiante en una base de datos
- Se expone un punto de acceso DELETE para eliminar una persona de un registro





# HTTP: Códigos

---

Un servidor web responde con códigos de error de acuerdo a la respuesta del mismo, estos códigos han sido estandarizados en la documentación del protocolo HTTP.

Estos se pueden usar por el cliente para determinar la acción correcta a seguir.

- 200 OK
- 201 Recurso creado
- 401 Credenciales requeridas para acceder al servicio
- 500 Error en el servidor
- Entre otros...

Es la responsabilidad del creador de la API emitir los códigos más acertados

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



# Alternativas: SOAP

---

SOAP es una alternativa bastante común para servicios web, no utiliza HTTP y provee un protocolo mucho más complejo.

Al no ser nativa en muchos contextos web no provee la misma interoperabilidad, por ejemplo, no es sencillo llamar a estos servicios dentro de un explorador de internet.

[REST vs. SOAP \(redhat.com\)](http://redhat.com)



# REst

---

Aprendiendo un poco más

[¿Qué es API RESTful? | Guía sobre API RESTful para principiantes | AWS \(amazon.com\)](#)



# Resumen

---

Entonces, un servidor 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.



# Resumen

---

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**.



# Resumen

---

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
```

# Conclusión

---



Preguntas?



# Servicios en Python

---

Podemos recordar que Python posee un ecosistema muy extendido de librerías y utilidades.

Esto también aplica a herramientas para acceder a servicios web, o exponer servidores!





# requests

---

A pesar de existir muchas librerías útiles, requests es una de las más sencillas y mejor documentadas.

Permite acceder a puntos HTTP, utilizar proxies, procesar códigos de respuesta, y mucho más.

```
py -m pip install requests
```



# Usando requests

---

Podemos fácilmente utilizar los verbos HTTP mediante las funciones expuestas por la librería

```
import requests
respuesta = requests.get('https://xkcd.com/1906/')
codigo = respuesta.status_code
print(codigo)
contenido = respuesta.text
```

```
Out[17]: 200
```

Mediante el parámetro **text** podemos obtener el cuerpo de la respuesta, por ejemplo si esta es una página HTML, podemos posteriormente procesarla con ayuda de otras librerías.



# Usando requests: descargas

---

Es muy fácil guardar el resultado de una solicitud, en caso de que por ejemplo sea un archivo.

```
import requests
resultado = requests.get("https://wwwnc.cdc.gov/travel/images/map-ecuador.png")
with open("alguna/ubicacion", 'wb') as f:
    f.write(resultado.content)
```

Recordemos que *open* nos permite trabajar con archivos de manera muy sencilla.



# Usando requests: parámetros

---

Dentro del protocolo HTTP, las operaciones permiten proveer diferentes parámetros en la solicitud.

```
import requests
argumentos = {'argumento1':1, 'argumento2':2}
r = requests.get('https://httpbin.org/get', params=argumentos)
print(r.text)
```

En el caso de una solicitud GET, esto simplemente agregará los mismos a la URL

```
https://httpbin.org/get?argumento1=1&argumento2=2
```



# Usando requests: POST

---

Como aprendimos anteriormente, existen diferentes verbos que podemos usar en HTTP, y cuando un servicio web soporta puntos de acceso POST, PUT o DELETE, los podemos acceder de manera similar.

```
import requests
argumentos = {'usuario':1,'clave':2}
r = requests.post('https://httpbin.org/post',params=argumentos)
```

A diferencia de GET, los parámetros serán enviados en el cuerpo de la solicitud, no en la URL.



# Usando requests: JSON

---

Es posible que un servicio web responda directamente con información estructurada como objetos JSON y no texto plano. La siguiente función crea un diccionario a partir de ese JSON.

```
import requests
argumentos = {'username': 'olivia', 'password': 123:}
respuesta = requests.post('https://httpbin.org/post',params=argumentos)

print(respuesta.json())
```

```
{'args': {}, 'data': '', 'files': {}, 'form': {'password': '123', 'username': 'olivia'}, 'headers': {'Accept': '*/*', 'Accept-encoding': 'gzip, deflate', 'Content-Length': '28', 'Content-Type': 'application/x-www-form-urlencoded', 'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.18.4'}, 'json': None, 'origin': '103.10.31.17, 103.10.31.17', 'url': 'https://httpbin.org/post'}
```



# Usando requests

---

Entonces, mediante **requests** podemos comunicarnos de manera sencilla con servidores HTTP, ya sea para interactuar con páginas web, o para vincularnos con servicios.



# Servidores en Python

---

De la misma manera que podemos interactuar con servidores externos mediante nuestro código de Python, podemos crear aplicaciones o servicios web de manera muy sencilla mediante un ecosistema de librerías y frameworks con este propósito.





---

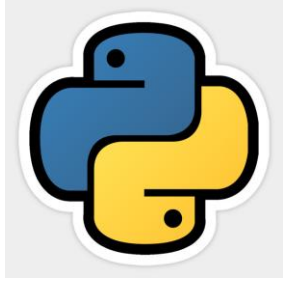
[Welcome to Flask — Flask Documentation \(2.1.x\)](https://palletsprojects.com/en/2.1.x/)  
[\(palletsprojects.com\)](https://palletsprojects.com/)

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!



---

## ¿Qué es un framework web?

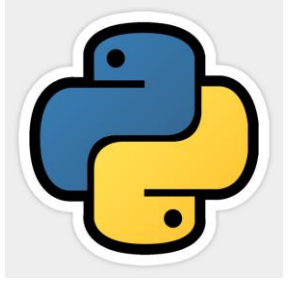
Una serie de utilidades y módulos que le permite a un desarrollador crear aplicaciones web sin preocuparse por gestionar código de nivel bajo como los protocolos, gestión de hilos, y muchas otras consideraciones.



---

Flask require simplemente que instalemos su dependencia base

```
py -m pip install flask
```



Un ejemplo básico!

```
from flask import Flask

app = Flask(__name__)

# Usando la sintaxis de @ podemos definir las rutas y su comportamiento

@app.route("/")
def hola_mundo():
    return "<p>Hola a todos!</p>"

@app.route("/adios")
def adios():
    return "<p>Adios a todos!</p>"
```



---

Podemos definir la variable `FLASK_APP` con el nombre del módulo inicial

```
FLASK_APP=ejemplo.py python -m flask run
```

O alternativamente por defecto flask buscará el archivo **app.py**



---

Por defecto flask inicia el servidor en el puerto 5000

```
FLASK_APP=ejemplo.py python -m flask run -p OTROPUERTO
```



Flask entonces, opera bajo el concepto de rutas, cada función se puede asignar a una o más rutas dependiendo de su comportamiento. Por defecto, el método es GET.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hola_mundo():
    return "<p>Hola a todos!</p>"
```





En el contexto de páginas web, no es eficiente escribir el código HTML en el código. Existe el concepto de plantillas o **templates**.

[Template Designer Documentation — Jinja Documentation \(3.1.x\) \(palletsprojects.com\)](#)

Por defecto, flask busca plantillas **jinja** en la carpeta **templates**.

```
from flask import render_template, Flask

app = Flask(__name__)

@app.route('/hola')
def hola():
    return render_template('hola.html', nombre="Daniel")
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Mi pagina</title>
</head>
<body>
    Hola {{ nombre }}
</body>
</html>
```



Sin embargo, la función puede definirse para operar con otros métodos, por ejemplo POST.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/enviar_json', methods=['POST'])
def procesar_json():
    tipo = request.headers.get('Content-Type')
    if (tipo == 'application/json'):
        json = request.json
        # Hacer algo con el objeto JSON
        return json
    else:
        return 'Tipo de contenido en el objeto no soportado!'
```



Las rutas, no son recursos específicos, de hecho la cadena de texto es una “regla”.

```
@app.route('/ruta/')
```

La misma puede ser dinámica e incluir parámetros.

```
@app.route('/ruta/<variable>')  
def procesar_ruta(variable):  
    return f"Recibi el parametro {variable}"
```

[127.0.0.1:8000/ruta/daniel](http://127.0.0.1:8000/ruta/daniel)

[127.0.0.1:8000/ruta/juan](http://127.0.0.1:8000/ruta/juan)



Así como es posible devolver texto plano, archivos o manejar plantillas. Es posible de igual manera devolver objetos formateados, por ejemplo JSON, algo muy útil en contextos de servicios web!

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/obtener_datos')
def procesar_informacion():
    informacion = {
        "nombre": "Daniel",
        "cedula": "123456"
    }
    #La función jsonify convierte un diccionario en una respuesta JSON!
    return jsonify(informacion)
```



Lo importante es que tenemos el poder entero de Python! Los datos que devolvamos pueden venir de una base de datos, un archivo, o inclusive otro servidor web (por ejemplo si usamos requests como intermediario!).

```
from flask import Flask, jsonify

app = Flask(__name__)

def leer_archivo():
    archivo = None
    with open("archivo.csv") as a:
        archivo = a.readlines()
    return archivo

@app.route('/obtener_datos')
def procesar_informacion():
    contenido = leer_archivo()
    #Procesamiento de la línea del archivo!
    partes = contenido[0].split(',')
    #La función jsonify convierte un diccionario en una respuesta JSON!
    return jsonify({"nombre": partes[0], "cedula": partes[1]})
```

archivo.csv

daniel,12345

resultado

```
{
  "cedula": "12345",
  "nombre": "daniel"
}
```



---

¡Pero eso es solo la superficie! Mediante flask podemos gestionar sesiones, aplicar contextos separados, manejar integraciones con cookies, y muchas cosas más.

Mejor aún, podemos integrarlo con las herramientas de validación de entidades y conexión eficiente a bases de datos.

**¡Revisar la documentación!**



# 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))
```

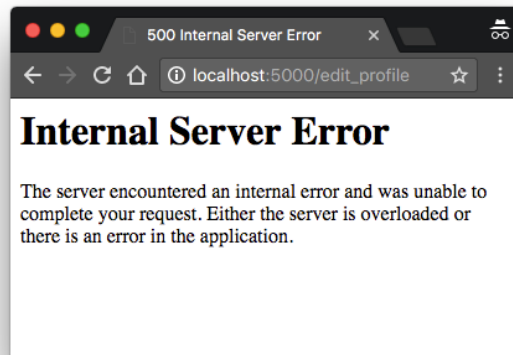
# Gestión de Errores

---



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
```



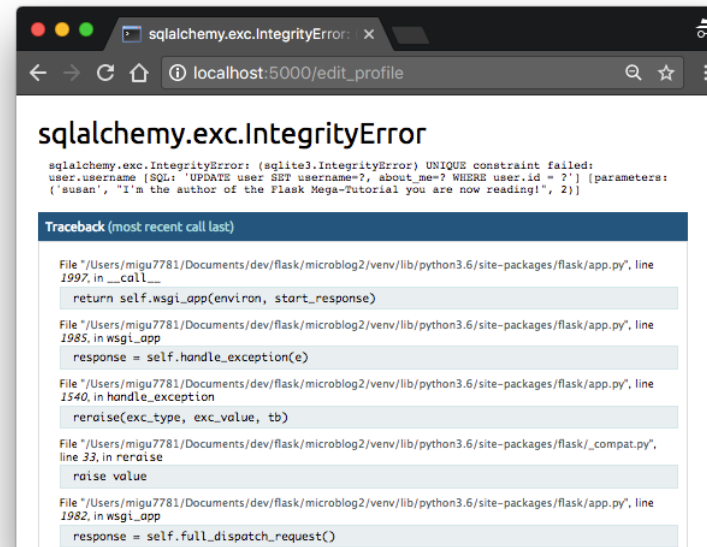


Sin embargo, lo presentado en la pantalla no es tan detallado y existen errores muy complejos que suceden en el contexto de aplicaciones reales.

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
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.

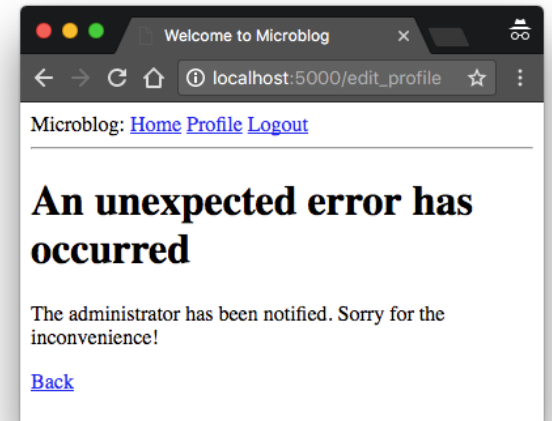
```
from flask import render_template
from app import app

@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.

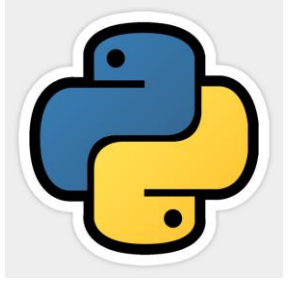
```
from flask import send_file
app = Flask(__name__)

@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!

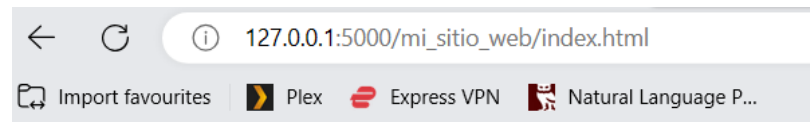
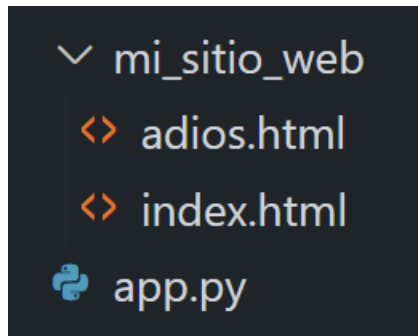
Flask ofrece automáticamente la capacidad de servir archivos estáticos ubicados en directorios de la aplicación.



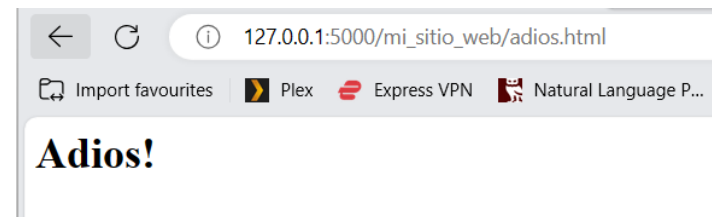
Al inicializar la aplicación uno puede especificar de qué directorio recoger los archivos estáticos, sin necesidad de especificar rutas para cada uno de ellos.

```
from flask import Flask

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



**Hola!**

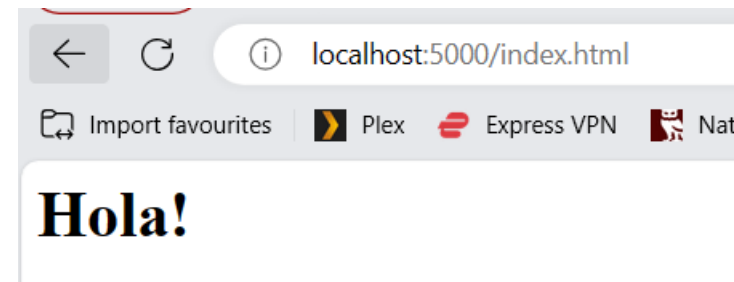
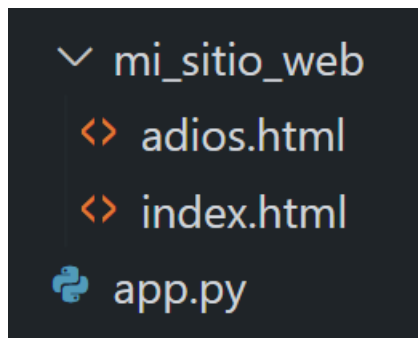




En caso de que querramos enteramente un sitio estático, que use flask como servidor, podemos mapear la carpeta a la ruta base.

```
from flask import Flask

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





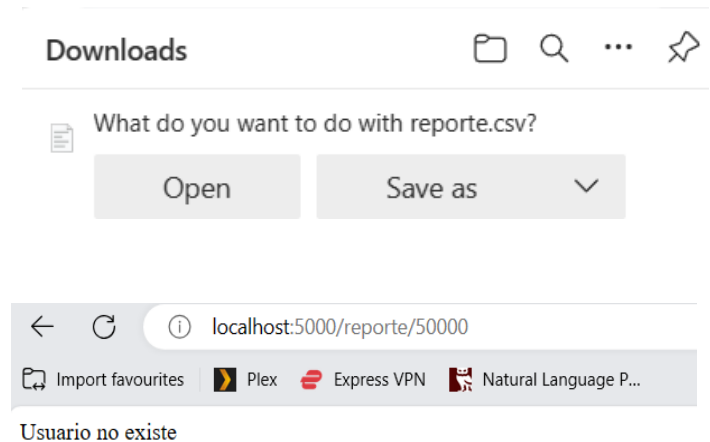
En el contexto de un servicio web, es posible que deseemos generar archivos dinámicos de datos, por ejemplo, un CSV a parte de un resultado de una base datos.

```
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"},
    )
```







---

A medida que una aplicación o servicio crece, es necesario implementar funcionalidades que permitan ofrecer un entorno más completo en cuanto a capacidades de usuario.

Por ejemplo, seguridad vía autenticación o funciones de mantenimiento de sesiones (en el contexto de aplicaciones web).

**Es importante explorar el ecosistema de extensiones.**

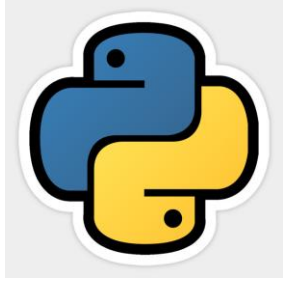


---

## Flask-SQLAlchemy

Extensión de flask que agrega soporte para SQLAlchemy, el set de utilidades ORM que permite trabajar con bases de datos directamente con objetos del lenguaje y no SQL.

Esta extensión permite acceder a los patrones comunes a nivel de empresa para acceso eficiente a bases de datos.



---

## Flask-Mail

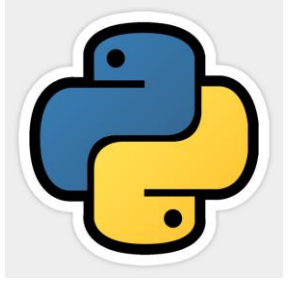
Extensión que permite enviar correos dentro del servidor. Por ejemplo, a los usuarios durante el proceso de registro, o errores del sistema a los administradores del mismo.



---

## Flask-RESTful

Extensión que permite diseñar APIs de acuerdo a la especificación REST. Recordemos que flask al ser un framework no opinionado da a lugar a que muchas veces se utilicen patrones que no siempre se alienen a las especificaciones de ciertos estándares.



---

## Flask-Login

Extensión que permite gestionar la autenticación y sesiones de usuario. Maneja el flujo de usuarios para un servicio o aplicación de manera simple, y ofrece inclusive gestión de cookies.



---

A final del día, existen muchas alternativas. Lo importante es conocer que existen diversas extensiones para los propósitos diferentes de los desarrolladores.

[Awesome Flask](#) | [Curated list of awesome lists](#) | [Project-Awesome.org](#)

# django



---

Una alternativa a flask.

Django es un proyecto más complejo y mucho más opinionado.

Pero al definir una estructura más establecida permite que las aplicaciones grandes sean más fáciles de gestionar en ciertos contextos.

[Flask vs Django: What's the Difference Between Flask & Django? \(guru99.com\)](https://guru99.com/flask-vs-django-what-s-the-difference-between-flask-django/)

[The web framework for perfectionists with deadlines | Django \(djangoproject.com\)](https://www.djangoproject.com/)

# django



---

No se considera un mini framework

- Incluye muchas características integradas como un ORM (Mapeo Objeto-Relacional), autenticación y una interfaz de administración
- Estructura impuesta, ventajas y desventajas.

Varias “**aplicaciones**” dentro de un “**proyecto**”.



# django



---

## •Proyecto:

- Representa todo el sitio o servicio web.
- Contiene configuraciones globales.
- Puede tener múltiples aplicaciones.

## •Aplicación:

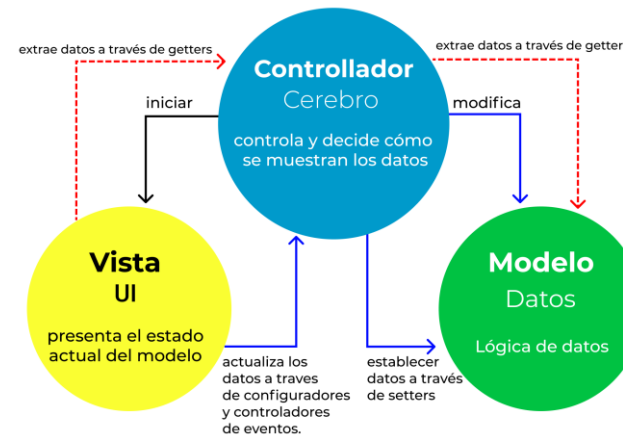
- Representa una funcionalidad específica.
- Puede ser reutilizada en diferentes proyectos.
- Contiene modelos, vistas, y plantillas específicas.



## Modelo – Vista – Controlador

Django primariamente utiliza MVC: Arquitectura de software que separa una aplicación en tres componentes principales

### Patrones de Arquitectura MVC





---

## Modelo

- Representa los datos y la lógica de negocio de la aplicación.
- Gestiona el acceso y la manipulación de los datos.
- Notifica a la vista sobre cualquier cambio en los datos.

```
# miapp/models.py
from django.db import models

class Producto(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    descripcion = models.TextField()

    def __str__(self):
        return self.nombre
```



## Vista – conocida como plantilla

- Es la presentación de los datos del modelo en un formato específico.
- Se encarga de mostrar la información al usuario.
- Actualiza la interfaz de usuario cuando el modelo cambia.

```
<!-- miapp/templates/miapp/lista_productos.html -->
<!DOCTYPE html>
<html>
<head>
  <title>Lista de Productos</title>
</head>
<body>
  <h1>Lista de Productos</h1>
  <ul>
    {% for producto in productos %}
      <li>{{ producto.nombre }} - {{ producto.precio }}</li>
    {% endfor %}
  </ul>
</body>
</html>
```



---

## Controlador (llamada view en django)

- Actúa como intermediario entre el modelo y la vista.
- Recibe la entrada del usuario y la traduce en acciones para el modelo o la vista.
- Puede actualizar el modelo y cambiar la vista en respuesta a las interacciones del usuario.

```
# miapp/views.py
from django.shortcuts import render
from .models import Producto

def lista_productos(request):
    productos = Producto.objects.all()
    return render(request,
        'miapp/lista_productos.html', {'productos':
        productos})
```

# django

---



```
pip install django
django-admin startproject miproyecto
cd miproyecto
python manage.py startapp miapp
```

OpenAPI

---



---

La especificación OpenAPI es una especificación para las API HTTP que proporciona un medio estandarizado para definir su API en cualquier lenguaje

Al implementar esta especificación, una API permite que sea sencillo describir la funcionalidad de la misma a los clientes, así como comprender el ciclo de vida completo de la misma.

[Inicio - OpenAPI Initiative \(openapis.org\)](https://openapis.org)





---

Originalmente conocido como Swagger. Es una especificación abierta que permite definir patrones claros para escribir y documentar nuestros servicios.

En el contexto de flask, la extensión flask-smorest ofrece generación automática de documentación que se alinea a esta especificación.

Esta extensión permite que, mediante comentarios del lenguaje, el desarrollador describa a las funciones del servicio, y luego éstos sean extraídos para construir un explorador interactivo.

[marshmallow-code/flask-smorest: DB agnostic framework to build auto-documented REST APIs with Flask and marshmallow \(github.com\)](https://marshmallow-code.com/flask-smorest)



## Simple Example ¶

Here is a basic "Petstore example", where The `Pet` class is an imaginary ORM.

First instantiate an `Api` with a `Flask` application.

```
from flask import Flask
from flask.views import MethodView
import marshmallow as ma
from flask_smorest import Api, Blueprint, abort

from .model import Pet

app = Flask(__name__)
app.config["API_TITLE"] = "My API"
app.config["API_VERSION"] = "v1"
app.config["OPENAPI_VERSION"] = "3.0.2"
api = Api(app)
```

Define a marshmallow `Schema` to expose the model.

```
class PetSchema(ma.Schema):
    id = ma.fields.Int(dump_only=True)
    name = ma.fields.String()
```

Define a marshmallow `Schema` to validate the query arguments.

```
class PetQueryArgsSchema(ma.Schema):
    name = ma.fields.String()
```

Instantiate a `Blueprint`.

```
blp = Blueprint("pets", "pets", url_prefix="/pets", description="Operations on pets")
```

Use `MethodView` classes to organize resources, and decorate view methods with `Blueprint.arguments` and `Blueprint.response` to specify request deserialization and response serialization respectively.

Use `abort` to return errors, passing kwargs used by the error handler (`handle_http_exception`) to build the error response.

```
@blp.route("/")
class Pets(MethodView):
    @blp.arguments(PetQueryArgsSchema, location="query")
    @blp.response(200, PetSchema(many=True))
    def get(self, args):
        """list pets"""
        return Pet.get(filters=args)

    @blp.arguments(PetSchema)
    @blp.response(201, PetSchema)
    def post(self, new_data):
        """Add a new pet"""
        item = Pet.create(**new_data)
```

[Quickstart — flask-smorest 0.42.0 documentation](#)



SMARTBEAR  
SwaggerHub

PetStoreSwagger 1.0.0

Export

## Swagger Petstore

1.0.0 OAS3

This is a sample Petstore server. You can find out more about Swagger at <http://swagger.io> or on [#swagger](irc://freenode.net).

[Terms of service](#)  
[Contact the developer](#)  
[Apache 2.0](#)  
[Find out more about Swagger](#)

Servers

Authorize

pet Everything about your Pets [Find out more](#)

|        |                   |   |       |
|--------|-------------------|---|-------|
| POST   | /pet              | Add a new pet to the store                | ⌵ 🔒 🔑 |
| PUT    | /pet              | Update an existing pet                    | ⌵ 🔒 🔑 |
| GET    | /pet/findByStatus | Finds Pets by status                      | ⌵ 🔒 🔑 |
| GET    | /pet/findByTags   | Finds Pets by tags                        | ⌵ 🔒 🔑 |
| GET    | /pet/{petId}      | Find pet by ID                            | ⌵ 🔒 🔑 |
| POST   | /pet/{petId}      | Updates a pet in the store with form data | ⌵ 🔒 🔑 |
| DELETE | /pet/{petId}      | Deletes a pet                             | ⌵ 🔒 🔑 |