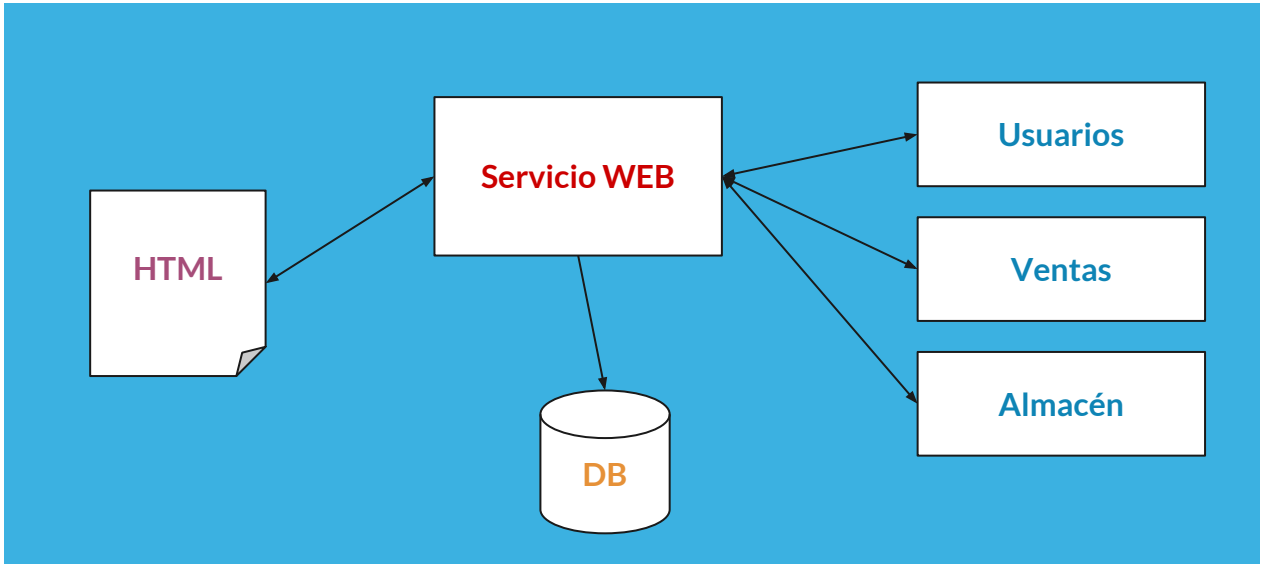
Desarrollo de una API REST en Python

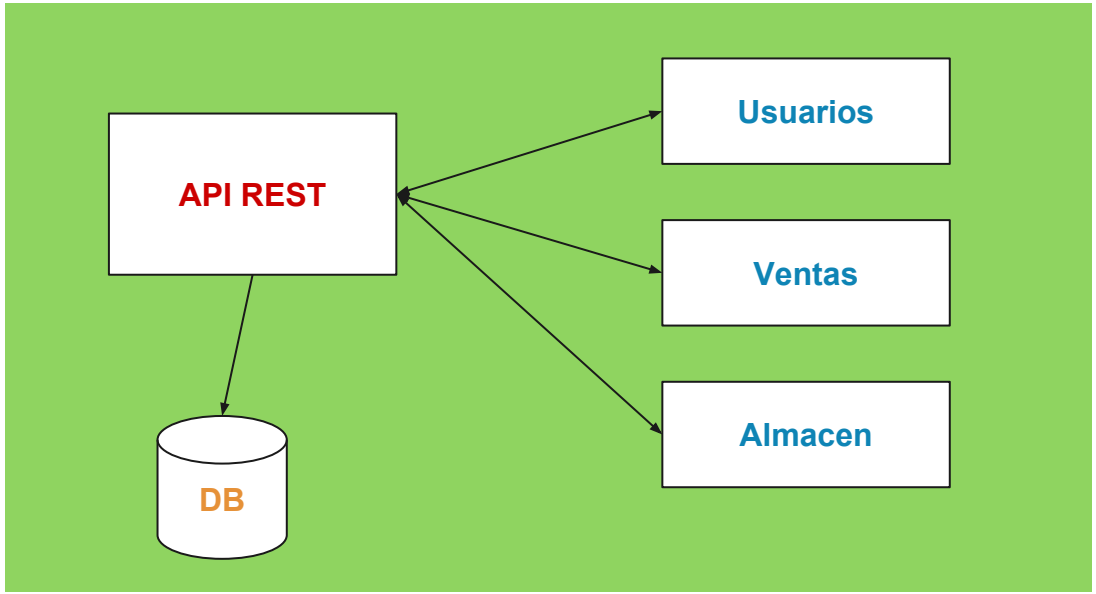
Flask Framework

Entendiendo una aplicación REST

Aplicación tradicional



Aplicación REST



Beneficios de usar microservicios

- Separación de responsabilidades
  - Proyectos independientes y separados por equipo de trabajo.
  - Podemos usar cualquier lenguaje o base de datos.
  - Fuerte documentación usando el protocolo HTTP
- Proyectos pequeños para mantenerlos fácil
  - Rompe complejidad al proyecto.
  - Te enfoca en un solo objetivo.
  - Reduce el riesgo de cambios.
- Más escalabilidad y opciones de **despliegues**
  - Al dividirse en componentes es más fácil actualizar.

Dificultades al usar microservicios

- Separación ilógica.
  - Mal diseño de la arquitectura
- Más interacción de la red.
  - Dependencias de un servicio para usar otro.
  - Falla de conexión o lag en el servicio
- Compartir el guardado de datos.
  - Separar o no los datos dependiendo el caso de uso
- Temas de compatibilidad
  - Aplicar cambios nuevos que en otros servicios no están preparados
  - Versionado de la API
- Pruebas
  - Hacer una revisión de los endpoints del servicio que se consume como el que se usa.

Implementando API REST con Python

Python es un lenguaje asombroso y versátil

Python

Algunos desarrolladores critican a Python por ser un lenguaje lento (interpretado) para construir servicios web de manera eficiente.

Python es lento (en comparación de otros lenguajes), pero es un lenguaje que se escoge (casi siempre) para construir microservicios, y la mayoría de las empresas y compañías están felizmente utilizandolo.

El Estándar WSGI

Introducción

Lo que sorprende a la mayoría de los desarrolladores de Python es lo fácil que es levantar y correr una aplicación web.

La comunidad web de python creó un estándar (inspirado en CGI) llamado **Web Server Gateway Interface** (WSGI). Permite simplificar la escritura de código en Python para poder crear una aplicación que sirva web mediante HTTP.

Cuando el código usa el estándar, tu proyecto puede ser ejecutado en servidores web como **Apache** o **nginx** usando la extensión wsgi como *uwsgi* o *mod_wsgi*.

Tu aplicación solo deberá de capturar las peticiones y enviar de vuelta las peticiones en JSON.

Características

El protocolo WSGI se convirtió en un estándar esencial y la comunidad de python lo adoptó ampliamente.

El mayor problema del estándar WSGI es la naturaleza sincrónica. Se tiene que esperar hasta que la función que recibió la petición regrese la respuesta. Escribiendo microservicios significa que tu código tiene que esperar la respuesta de varios recursos (Si es el caso).

Esto es un comportamiento totalmente válido de un API HTTP. Los servidores wsgi permiten ejecutar un grupo de subprocesos para atender simultáneamente las solicitudes, pero no se puede tener miles de ellos por qué pool (piscina) se agota, bloqueando el acceso al cliente, incluso si el microservicio no está haciendo nada.

No-WSGI

Es una de las razones para que frameworks como no-wsgi como **Twisted** y **Tornado**, en el lado de javascript **Node.js** hayan tenido mucho existo. Por qué son tecnologías totalmente asíncronas.

Con frameworks como twisted puedes usar callbacks para pausar y seguir con el proceso habitual y enviar una respuesta. Para que de este modo se aceptan más solicitudes concurrentes.

No hay una simple manera con el estándar WSGI para hacer algo similar, la comunidad debate por mucho tiempo este problema, dando como consecuencia que la comunidad termine de abandonar el estándar wsgi

Flask

Flask

Flask es un microframework fue lanzado en el 2010. construido sobre Werkzeug WSGI toolkit, que proporciona las bases para interactuar con solicitudes HTTP a través del protocolo WSGI, y varias herramientas como un sistema de enrutamiento.

El término micro-framework puede ser engañoso y no significa que sirvan solo para hacer micro aplicaciones, usando estas herramientas puedes construir cualquier aplicación con un base de código reducida, permite organizar libremente tu código como quieras y toma poca decisiones.

NO IMPONE NINGÚN PARADIGMA

Componentes de Flask

\+

=

Werkzeug
WSGI

Motor de templates

Flask

Como Flask maneja peticiones

El punto de entrada del Framework es la clase **Flask** del módulo **flask.app**.

Ejecutar Flask significa correr una única instancia de esta clase, que se encargará de gestionar las solicitudes del WSGI entrantes, enviarlas al código correcto y después devolver una respuesta.

La clase ofrece un método de ruta que puede decorar sus funciones. Cuando se decora una función, se convierte en una vista y se registra en el sistema de enrutamiento de Werkzeug. Este sistema es un pequeño motor de reglas para coincidir con las solicitudes y las respuestas.

Ejemplo: Aplicación en Flask

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api')
def index():
    return jsonify({'Hello': 'World!'})

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

Única instancia de Flask

Decorador de función

Ejecutar el código (Terminal)

```
$ python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Llamar al endpoint /api devolverá una respuesta JSON válida con los encabezados correctos, gracias a la función jsonify(), que se encarga de convertir los diccionarios de python en respuesta JSON válida con el encabezado adecuado de Content-Type.

La función jsonify() crea un objeto de respuesta.

Peticiones en Flask

Mientras que muchos frameworks web pasan explícitamente un objeto de petición a su código, Flask proporciona una variable de petición global implícita, que apunta al objeto request actual construido con la petición entrante.

Esta decisión de diseño hace que el código de vistas sea más simple y conciso.

Ejemplo: Objeto request

```
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/api')
  def index():
    print(request)
    print(request.environ)
    response = jsonify({'Hello': 'World!'})
    print(response)
    print(response.data)
    return response

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

Routing

El enrutamiento ocurre en el app.url_map, que es una instancia de la clase Map de **Werkzeug**. Esta clase usa expresiones regulares para determinar si una función decorada por @app.route coincide con la petición entrante. El enrutamiento sólo examina la ruta que se proporciona en la llamada para saber si coinciden.

En el decorador de la ruta tu puedes especificar el método en el que se podrá acceder a esa ruta mediante una lista con los verbos (en ingles y mayúsculas)

Ejemplo: Rutas con verbos

```
@app.route('/api', methods=['POST', 'DELETE', 'GET'])
  def index():
    return jsonify({'Hello': 'World!'})
```

Resumen

- **Routing**: Flask create una clase Map
- **Request**: Flask pasa un objeto request en la vista. (En otro frameworks el controlador)
- **Response**: Un objeto respuesta se envía de vuelta retornando el contenido

Variables

Una característica importante en el sistema de rutas son las variables.

Tu puedes usar variables usando la sintaxis **<NOMBRE_VARIABLE>**. Esta notación es un estándar (otros micro-frameworks como Bottle la usa), y permite describir en los endpoint valores dinámicos.

Por ejemplo, si tu quieres crear una función que maneje la respuesta de una persona determinada se usa **/person/N,** donde **N** podría ser un identificador único de la persona, y esto se podría usar: **/person/<person_id>**

Tu puedes forzar que una variable se le pase un tipo de dato en específico: **/person/<int:person_id>**, puedes usar los siguientes convertidores: int, float, path, string (default), uuid, si no coinciden retornará 404

Ejemplo: Variables

```
@app.route('/api/person/<person_id>')
  def person(person_id):
    response = jsonify({'id': person_id})
    return response
```

Convertidores personalizados

Flask permite la creación de convertidores personalizados que sigan ciertas reglas, para hacer esto necesitarás crear una clase que derive de **BaseConverter**, el cual implementa dos métodos: **to_python()** que convierte el valor a un objeto de python para la vista y el **to_url()** que va a otro camino.

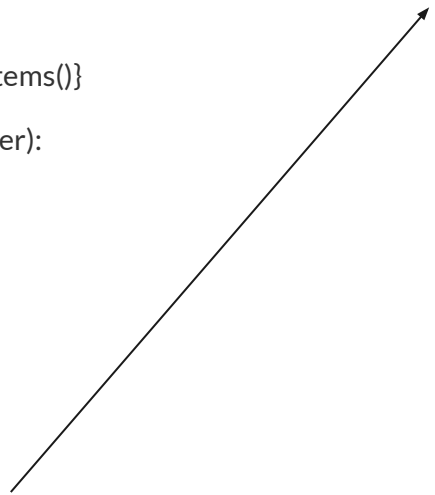
Ejemplo: Convertidor personalizado

```
from flask import Flask, jsonify, request
from werkzeug.routing import BaseConverter, ValidationError

_USERS = {'1': 'Tarek', '2': 'Freya'}
_IDS = {val: id for id, val in _USERS.items()}

class RegisteredUser(BaseConverter):
    def to_python(self, value):
        if value in _USERS:
            return _USERS[value]
        raise ValidationError()

    def to_url(self, value):
        return _IDS[value]
```



```
app = Flask(__name__)
app.url_map.converters['registered'] =
RegisteredUser


@app.route('/api/person/<registered:name>')
    def person(name):
        response = jsonify({'Hello hey': name})
        return response
```

La función url_for

La última característica interesante del sistema de rutas de Flask es la función url_for(). Dato una vista, regresará su URL actual

```
from flask_converter import app
from flask import url_for

with app.test_request_context():
   print(url_for('person', name='Tarek'))
```

Request

Cuando un “Request” o petición llega, Flask llama a la vista dentro de un código en un hilo seguro y usa el helper **local** de Wezeug, y eso hace que cada hilo este en un entorno aislado, para cada petición en específico.

En otras palabras cuando tu accedes al objeto global request en tu vista, tu garantiza que ese objeto es único para tu hilo y no llama fuga de datos en otro hilo en un entorno multihilo.

Como se mencionó Flask utiliza los datos del entorno WSGI entrantes para crear el objeto request. Este objeto es una instancia de la clase Request, que fusiona varias clases por medio de mixin para analizar los encabezados específicos del entorno entrante.

Request

La deducción es que cada vista puede intro inspeccionar la petición entrante a través de los atributos del objeto de solicitud.

El trabajo que hace Flask para este propósito es muy inteligente.

Ejemplo: Request

```
from flask import Flask, request

  app = Flask(__name__)

  @app.route("/")
  def auth():
    print("The raw Authorization header")
    print(request.environ["HTTP_AUTHORIZATION"])
    print("Flask's Authorization header")
    print(request.authorization)
    return ""
```

Response

En los ejemplos anteriores, usamos la función jsonify(), el cual crea un objeto Response que mapea el retorno de una vista. (controlador)

El objeto de Response es técnicamente un estándar de una aplicación WSGI que tu podrías usar directamente, este es envuelto por Flask, llamando a una función con parámetros environ y start_response recibidos del servidor web.

Cuando Flask toma una vista vía URL, se espera un que se retorne un objeto llamable que pueda recibir los argumentos environ y start_response.

Response

En caso de que el valor no sea llamable, Flask intentará convertirlo en un objeto Response si cumple con uno de los siguientes caso:

- **str:** Los datos se codifican como UTF-8 y se asignan al cuerpo de la respuesta HTTP.
- **bytes/bytesarray:** Utilizado como el cuerpo
- **A(response, status, headers) tupla:** Donde el response puede ser un objeto Response o uno de los anteriores, status es un entero que sobreescribe el estado de una respuesta y el header es un mapeado que extiende de los headers response.
- **A(response, status) tupla:**
- **A(response, headers) tupla:**

Response

En la mayoría de los casos, cuando construimos microservicios, utilizaremos la función jsonify() incorporada, pero en caso de necesitar otro tipo de contenido en los endpoints, crear una función que genere dato con la clase Response es muy simple y fácil.

Example: Response personalizado

```
from flask import Flask
import yaml     # require PyYAML

app = Flask(__name__)

def yamlify(data, status=200, headers=None):
    _headers = {'Content-Type': 'application/x-yaml'}
    if headers is not None:
        _headers.update(headers)
    return (yaml.safe_dump(data), status, _headers)

@app.route('/api')
def index():
    return yamlify(['Hello', 'YAML', 'World!'])
```

Características incorporadas en Flask

Características incorporadas

Flask viene con muchas más características a la hora de trabajar un desarrollo web.

- **Objeto de sesión:** Datos basados en cookies.
- **Globals:** Almacenar datos en el contexto de la petición.
- **Señales:** Envío e intercepción de eventos.
- **Extensiones y middlewares:** Agregar funciones.
- **Templates:** Construir contenido basado en texto (html).
- **Configuración:** Agrupar las opciones de ejecución en un archivo de configuración.
- **Blueprints:** Organizar el código en espacios de nombres
- **Manejo de errores y depuración:** Como tratar errores en la aplicación.

Globales

Flask incorpora un mecanismo para almacenar variables globales que son exclusivas de un hilo en particular y del contexto de la petición. Que utiliza para la petición y la sesión, pero también está disponible para almacenar cualquier tipo de objeto personalizado

La variable flask.g contiene todos los globales y pueden establecerse atributos que se desee en él.

En Flask el decorador @app.before_request puede utilizarse para señalar a una función y que esta se realice antes de enviar una petición a la vista (controlador).

Es un patrón común utilizar en Flask usar before_request para establecer valores en el objeto global, para que todas las peticiones puedan interactuar con el global y obtener datos

Ejemplo: Globales

```
from flask import Flask, jsonify, g, request

  app = Flask(__name__)

  @app.before_request
  def authenticate():
    if request.authorization:
      g.user = request.authorization['username']
    else:
      g.user = 'Anonymous'

  @app.route('/api')
  def index():
    return jsonify({'Hello': g.user})
```

Señales

Flask se integra con Blinker (https://pythonhosted.org/blinker/) , que es una biblioteca de señales que le permite suscribir una función a un evento.

Los eventos son instancias de la clase blinker.signal creada con una etiqueta única y Flask instancia 10 de ellos en la versión 0.12. (http://flask.pocoo.org/docs/0.12/api/#core-signals-list).

Registrarse a un evento particular se realiza llamando al método connect. Las señales se activan cuando algún código llama al método send. El método send acepta argumentos adicionales para pasar datos a todas las funciones registradas.

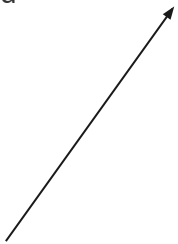
Ejemplo: Señales

```
from flask import Flask, jsonify, g, request_finished
  from flask.signals import signals_available

  if not signals_available:
    raise RuntimeError("pip install blinker")

  app = Flask(__name__)

  def finished(sender, response, **extra):
    print('About to send a Response')
    print(response)
```



```
request_finished.connect(finished)

@app.route('/api')
def index():
  return jsonify({'Hello': 'World'})
```

Señales

Se debe de tomar en cuenta que la función de señal solo función de señales sólo funcionará si se instala Blinker.

Algunas señales implementadas en Flask no son útiles en las API REST o en microservicios. Pero algunas señales que Flask dispara a lo largo de la vida de la petición pueden ser realmente útiles.

Por ejemplo, la señal got_request_exception se activa cuando se produce una excepción antes de que Flask sepa qué hacer con ella. Así se podría utilizar Python Raven de Sentry para engancharse y poder registrar las excepciones

Extensiones o middlewares

Las extensiones en Flask son simplemente proyectos, que una vez instalados, proporcionan un paquete o un módulo denominado flask_LOQUESEA. En versiones anteriores era flask.ext.LOQUESEA

Para crear un proyecto se tiene que seguir ciertos procedimientos como se describen aquí: (http://flask.pocoo.org/docs/latest/extensiondev.). Estas guías o directrices son prácticas más o menos que podrían aplicarse a cualquier proyecto en python. Flask tiene una lista de extensiones mantenidas en http://flask.pocoo.org/extensions/ , que es un buen punto de referencia cuando buscas características adicionales. Lo que proporciona una extensión depende del desarrollador.

Otro mecanismo para extender a Flask es usar los WSGI middlewares. Un WSGI middleware es un patrón para extender aplicaciones WSGI envolviendo las peticiones realizadas.

Ejemplo: Middleware

```
from flask import Flask, jsonify, request
  import json

  class XFFMiddleware(object):
    def __init__(self, app, real_ip='10.1.1.1'):
      self.app = app
      self.real_ip = real_ip

    def __call__(self, environ, start_response):
      if 'HTTP_X_FORWARDED_FOR' not in environ:
        values = '%s, 10.3.4.5, 127.0.0.1' % self.real_ip
        environ['HTTP_X_FORWARDED_FOR'] = values
      return self.app(environ, start_response)
```

```
app = Flask(__name__)
  app.wsgi_app = XFFMiddleware(app.wsgi_app)

  @app.route('/api')
  def index():
    if "X-Forwarded-For" in request.headers:
      ips = [ip.strip() for ip in
          request.headers['X-Forwarded-For'].split(',')]
      ip = ips[0]
    else:
      ip = request.remote_addr
    return jsonify({'Hello': ip})
```

Middleware

La manipulación del entorno WSGI antes de que su aplicación lo obtenga la información está bien. Pero si se desea implementar cualquier cosa que pueda afectar a la respuesta, hacerlo dentro de un middleware WSGI puede ser un trabajo muy doloroso.

Configuración

Al crear aplicaciones, se tendrá que exponer opciones para ser ejecutadas, como la información para conectarse a una base de datos o cualquier otra variable que sea específica para su implementación (entornos).

Flask utiliza un mecanismo similar a Django para realizar una configuración. El objeto Flask viene con un objeto denominado config, que contiene algunas variables incorporadas, y que puede actualizarse cuando se instancie el objeto Flask a través de algún objeto personalizado que se haga.

Ejemplo: Configuración

```
class Config:
    DEBUG = False
    SQLURI = 'postgres://tarek:xxx@localhost/db'
from flask import Flask
app = Flask(__name__)
app.config.from_object('Config')
```

Configuración

Existen convenientes al usar módulos o clases en Python como archivos de configuración.

En primer lugar puede ser tentador poder usar la configuración en el código, esto puede traer problemas porque esos archivos serán tratados como archivos de aplicación adicionales. Esto no es lo práctico se realiza una aplicación en forma.

En segundo lugar, si otro equipo o personas se encargan de administrar el archivo de configuración en su aplicación, tendrá que editar el código en python para hacerlo.

Es mas conveniente cargar las configuraciones en un archivo basado en texto como JSON, YAML o cualquier otro

Configuración

El formato INI es el formato más usado en la comunidad de Python, porque hay un analizador INI incluído en la biblioteca estándar y porque es bastante universal.

Existen muchas extensiones en Flask para cargar la configuración desde un archivo INI, pero con la biblioteca estándar ConfigParse se podrá realizar

Blueprints

Cuando se escriben APIs o microservicios que tienen más de un endpoint. Se tendría que hacer una lista larga de funciones con decoradores que trabajen sobre el objeto Flask, el primer paso lógico para un proyecto de este tipo es organizar el código y separar los endpoints en módulos, y crear una instancia de la aplicación (app = Flask(__name__)), para asegurarse de tener todo lo necesario para que Flask registre las vistas.

Los Blueprints agrupa las vistas en espacios de nombres. El crear un objeto Blueprint es muy parecido a un objeto Flask y se puede utilizar de la misma manera organizar las vistas.

Para poder registrar un Blueprint dentro de nuestra instancia de Flask se realiza: app.register_blueprint

Ejemplo: Blueprint

```
from flask import Blueprint, jsonify

teams = Blueprint('teams', __name__)

_DEVS = ['Tarek', 'Bob']
_OPS = ['Bill']
_TEAMS = {1: _DEVS, 2: _OPS}

@teams.route('/teams')
def get_all():
  return jsonify(_TEAMS)

@teams.route('/teams/<int:team_id>')
def get_team(team_id):
  return jsonify(_TEAMS[team_id])
```

Blueprints

El módulo principal (app.py), puede importar este archivo y registrar el blueprint con app.register_blueprint(teams).

Este mecanismo es interesante cuando se requiere reutilizar un conjunto genérico de vistas en otra aplicación o varias veces en la misma aplicación.

Manejo de errores y depuración

Cuando algo va mal en la aplicación, es importante controlar que respuestas recibirán los clientes. En aplicaciones web HTML, normalmente se obtienen páginas HTML específicas cuando se encuentra un error 404 o 50x y así funciona Flask.

Pero al construir una api o un microservicio, se necesita tener más control sobre lo que debe de ser enviado de vuelta al cliente, es donde los manejadores de errores personalizados nos ayudarán.

La otra característica importante es la capacidad de depurar lo que está mal con el código cuando se produce un error inesperado. Flask viene con un depurador incorporado, que se puede activar cuando la aplicación se ejecuta en modo de depuración.

Errores personalizados

Cuando el código no puede controlar una excepción, Flask devuelve un HTTP 500 sin proporcionar ninguna información específica, como el traceback. Producir un error genérico es un comportamiento predeterminado seguro para evitar que se escape cualquier información privada a los usuarios en el cuerpo del error.

Al implementar un API REST usando JSON, es una buena práctica asegurarse que cada respuesta enviada a los clientes, incluyendo cualquier excepción, esté formateada con JSON.

Flask permite personalizar el manejo de errores de la aplicación a través de un par de funciones. El primero es usar el decorador @app.errorhandler, que funciona como @app.route, pero en lugar de proporcionar un endpoint el decorador víncula una función a un código de error en específico.

Ejemplo: Manejo de errores

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.errorhandler(500)
def error_handling(error):
    return jsonify({'Error': str(error)}, 500)

@app.route('/api')
def index():
    raise TypeError("Algun error")
```