Table of Contents

Conexión a API y Uso de Web APIs con Requests

Curso gráfico MIRO

Contenido repositorio

Python venv y módulos:

En el directorio raiz del repositorio encontramos archivos con prefijo requirements-X.XX.txt, los mismos corresponden a versiones compiladores python, de esta manera cuando se quieran ejecutar los ejemplos expuestos en los talleres los mismos contarán con todas las dependencias a librerias necesarias.

pip install -r requirements-X.XX.txt

pip install flask

pip install connexion[flask]

pip install connexion[uvicorn]

pip install connexion[swagger-ui]

pip install requests

pip install flask_restx

pip install flask_restful

pip install flask_sqlalchemy

pip install flask_swagger_ui

pip install flask_marshmallow

pip install marshmallow-sqlalchemy

directorio introduccion:

Este directorio contiene ejemplos request metodos: GET,POST,PUT y DELETE

ejemplo-01: Request método GET

ejemplo-02: Request método POST

ejemplo-03: Request método PUT

ejemplo-04: Request método DELETE

ejemplo-05: Request método GET con key

ejemplo-06: Esqueleto base implementacion FLASK

ejemplo-07: Flask con clase interna implementando método GET

Nota: Para la ejecución de estos ejemplos es necesario ejecutar desde flask-swagger-01/app-02.y

directorio flask-swagger-01:

app-01.py:

Implementación Flask, OpenApi/Swagger, a partir de modelo JSON se generan endpoints donde se obtiene una lista de usuarios.

app-02.py:

Implementación Flask, OpenApi/Swagger, a partir de modelo YAML se generan endpoints para manejo de personas, GET, POST, PUT, DELETE, a su vez determina la entidad People con sus atributos.

directorio flask-swagger-02:

app.py:

Implementación Flask, OpenApi/Swagger, a partir de modelo YAML se generan endpoints manejo de personas y notas relacionadas, sustentado sobre base de datos.

Web API

¿Qué es una Web API?

Una Web API, o Application Programming Interface, es un conjunto de métodos y datos que se exponen a través de Internet para que los desarrolladores puedan interactuar con ellos. Las Web APIs se utilizan para una amplia gama de propósitos, como proporcionar acceso a datos, realizar tareas automatizadas o crear nuevas aplicaciones.

¿Cómo funcionan las Web APIs?

Las Web APIs se basan en el protocolo HTTP, que es el protocolo de comunicación estándar para la World Wide Web. Las solicitudes a una Web API se realizan utilizando el método HTTP GET o POST. El método GET se utiliza para recuperar datos de la API, mientras que el método POST se utiliza para enviar datos a la API.

¿Cuáles son los tipos de Web APIs?

Hay muchos tipos diferentes de Web APIs, pero algunas de las más comunes incluyen:

APIs RESTful: Las APIs RESTful son las más comunes y se basan en el modelo REST. El modelo REST define una serie de convenciones para el diseño de APIs, como el uso de URIs para identificar recursos, el uso de verbos HTTP para indicar acciones y el uso de formatos de datos estándar, como JSON o XML.
APIs SOAP: Las APIs SOAP son un tipo de API más antiguo que utiliza el protocolo SOAP. SOAP es un protocolo más complejo que REST, pero ofrece algunas ventajas, como el soporte para seguridad y transacciones.
APIs GraphQL: Las APIs GraphQL son un tipo de API más reciente que ofrece una forma más flexible de interactuar con los datos. GraphQL permite a los desarrolladores

solicitar solo los datos que necesitan, lo que puede mejorar el rendimiento y la eficiencia.

¿Cuál es la importancia de las Web APIs?

Las Web APIs son una herramienta esencial para el desarrollo web moderno. Las APIs se utilizan para una amplia gama de propósitos, como:

Acceso a datos: Las APIs se pueden utilizar para acceder a datos de una variedad de fuentes, como bases de datos, sistemas de archivos o servicios en la nube.

* Automatización de tareas: Las APIs se pueden utilizar para automatizar tareas, como enviar correos electrónicos, procesar pagos o actualizar datos.
Creación de nuevas aplicaciones: Las APIs se pueden utilizar para crear nuevas aplicaciones, como aplicaciones móviles, aplicaciones de escritorio o aplicaciones web.

Además de GET y POST, los métodos HTTP más utilizados en las Web APIs son:

PUT: Se utiliza para actualizar datos en una API.
DELETE: Se utiliza para eliminar datos de una API.
PATCH: Se utiliza para actualizar parte de los datos en una API.
HEAD: Se utiliza para obtener el encabezado de una respuesta HTTP.
OPTIONS: Se utiliza para obtener información sobre los métodos HTTP admitidos por una API.

Estos métodos se utilizan para realizar diferentes acciones en una API, como:

GET: Recuperar datos de una API.
POST: Enviar datos a una API.
PUT: Actualizar datos en una API.
DELETE: Eliminar datos de una API.
PATCH: Actualizar parte de los datos en una API.
HEAD: Obtener el encabezado de una respuesta HTTP.
OPTIONS: Obtener información sobre los métodos HTTP admitidos por una API.

En Python, estos métodos se pueden utilizar de la siguiente manera:

Ejemplo 01:

```
    # Importar la biblioteca requests
    import requests
    import json

    # Realizar una solicitud GET
    headers = {"accept": "*/*"}
    response = requests.get('http://localhost:8000/api/people',
headers=headers)

    # Obtener el contenido de la respuesta
    data = response.json()
```

Ejemplo 02:

```
    # Importar la biblioteca requests
    import requests
    import json

    # Realizar una solicitud POST
```

```
    headers = {"accept": "*/*", "Content-Type" : "application/json"}
    data = {'fname': 'John', 'lname': "Doe"}
    response = requests.post('http://localhost:8000/api/people',
json=data, headers=headers)

    # Obtener el código de estado de la respuesta
    status_code = response.status_code
    data_response = response.json()

    # Imprimir el código de estado de la respuesta
    print()
    print(f"Status code => { status_code}")
    print()
    print(json.dumps(data_response, indent=2))
```

Ejemplo 03:

```
    # Importar la biblioteca requests
    import requests
    import json

    # Realizar una solicitud PUT
    headers = {"accept": "*/*", "Content-Type" : "application/json"}
    data = {'fname': 'Estela', 'lname': "Quiroga"}
    response = requests.put('http://localhost:8000/api/people/Rodriguez',
json=data, headers=headers)

    # Obtener el código de estado de la respuesta
    status_code = response.status_code
    data_response = response.json()

    # Imprimir el código de estado de la respuesta
    print()
    print(f"Status code => { status_code}")
    print()
    print(json.dumps(data_response, indent=2))
```

Ejemplo 04:

```
    # Importar la biblioteca requests
    import requests

    # Realizar una solicitud DELETE
    headers = {"accept": "*/*", "Content-Type" : "application/json"}
    response =
requests.delete('http://localhost:8000/api/people/Rodriguez',
headers=headers)

```

```
    # Obtener el código de estado de la respuesta
    status_code = response.status_code

    # Imprimir el código de estado de la respuesta
    print()
    print(f"Status code => { status_code}")
    print()
    print(response.text)
```

Ejemplo head function:

```
    # Importar la biblioteca requests
    import requests

    # Realizar una solicitud HEAD
    response = requests.head('https://api.example.com/')

    # Obtener el encabezado de la respuesta
    headers = response.headers

    # Imprimir el encabezado de la respuesta
    print(headers)
```

Ejemplo options function:

```
    # Importar la biblioteca requests
    import requests

    # Realizar una solicitud OPTIONS
    response = requests.options('https://api.example.com/')

    # Obtener la información sobre los métodos HTTP admitidos
    allowed_methods = response.headers['Allow']

    # Imprimir la información sobre los métodos HTTP admitidos
    print(allowed_methods)
```

Estos son solo algunos de los métodos HTTP que se pueden utilizar en las Web APIs. Otros métodos HTTP menos comunes incluyen:

TRACE: Se utiliza para realizar un seguimiento de una solicitud HTTP.
CONNECT: Se utiliza para crear una conexión TCP/IP a un host remoto.
OPTIONS: Se utiliza para obtener información sobre los métodos HTTP admitidos por una API.

La elección del método HTTP correcto para una API depende de la acción que se desee realizar.

Conclusión

Las Web APIs son una herramienta poderosa que puede ser utilizada por desarrolladores de todo nivel de experiencia. Las APIs se utilizan para una amplia gama de propósitos y pueden ser una gran manera de mejorar la funcionalidad y la eficiencia de sus aplicaciones web.

Flask

`Introducción a Flask, el framework de desarrollo web de Python`

¿Qué es Flask?

* Es un microframework ligero y flexible para crear aplicaciones web en Python.
* Es conocido por su simplicidad, naturaleza no opinante y facilidad de uso.
* Proporciona las herramientas esenciales para el desarrollo web, lo que le permite agregar funcionalidad según sea necesario a través de extensiones.

Características clave:

Minimalista: No impone mucha estructura, lo que le brinda libertad en las elecciones de diseño.
Enrutamiento: Define patrones de URL para mapear funciones (vistas) que manejan diferentes solicitudes.
Plantillas Jinja2: Utiliza el motor de plantillas Jinja2 para generar contenido HTML dinámico.
Biblioteca de utilidades WSGI de Werkzeug: Construido sobre Werkzeug, que proporciona utilidades WSGI para el manejo de solicitudes y respuestas.
Extenso: Ofrece un rico ecosistema de extensiones (Flask-SQLAlchemy, Flask-Login, etc.) para tareas comunes de desarrollo web.

Estructura básica:

1. Importar Flask:

.. code:: python

```
from flask import Flask
```

2. Crear una instancia de la aplicación:

.. code:: python

```
app = Flask(__name__)
```

3. Definir rutas:

.. code:: python

```
@app.route('/')
def index():
    return "Hola, mundo!"
```

4. Ejecutar la aplicación:

.. code:: python

```
flask run
```

Ventajas de Flask:

* Simple y fácil de aprender, incluso para principiantes.
* Flexible y adaptable a diversas necesidades de proyectos.
* Gran comunidad y documentación extensa.
* Ideal para prototipado, aplicaciones pequeñas a medianas y API.

Casos de uso comunes:

* Sitios web y blogs personales
* API RESTful

* Servicios web
* Plataformas de comercio electrónico
* Paneles de visualización de datos
* Aplicaciones web personalizadas

Ejemplo básico

El siguiente ejemplo muestra cómo crear una aplicación web simple con Flask:

Ejemplo 06:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hola, mundo!"

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

Este código crea una aplicación web con una sola ruta, `/`, que devuelve la cadena "Hola, mundo!"

Esto abrirá una instancia de la aplicación en el puerto 5000. Puede acceder a la aplicación en su navegador web en la siguiente URL:

http://localhost:5000

Conclusiones

Flask es un framework de desarrollo web flexible y poderoso que es ideal para una amplia gama de proyectos. Es una buena opción para principiantes y desarrolladores experimentados por igual.

Python OpenAPI y Swagger

Integración en Python de OpenAPI y Swagger:

Conceptos clave:

* Especificación OpenAPI (OAS): Un formato estandarizado para describir APIs REST, que permite una documentación, generación de clientes y herramientas consistentes.
* Swagger: Un conjunto de herramientas para implementar OpenAPI, que incluye una interfaz de usuario para exploración visual y generación de código.

Bibliotecas y marcos de Python:

* Generación de documentación OpenAPI:
  * Flask-RESTX: Se integra a la perfección con Flask para definir puntos finales de API y generar automáticamente la interfaz de usuario Swagger.
  * apispec: Independiente del marco, admite varios formatos de serialización para generar documentación OpenAPI.
  * connexion: Construye APIs REST a partir de especificaciones Swagger/OpenAPI, asegurando el cumplimiento de la especificación.
* Consumo de APIs descritas por OpenAPI:
  * requests: Biblioteca base para realizar solicitudes HTTP, pero requiere el manejo manual de las estructuras de solicitud/respuesta.
  * pyswagger: Genera código cliente Python a partir de especificaciones OpenAPI, ofreciendo una interacción con las API segura de tipos.

Casos de uso comunes:

* Documentar API existentes: Generar una interfaz de usuario Swagger interactiva para un uso claro de la API y su exploración.
* Construir API con Swagger: Diseñar las especificaciones de la API primero, luego generar el código del lado del servidor y las SDK de clientes.
* Interactuar con API externas: Utilizar las especificaciones OpenAPI para comprender la estructura de la API y automatizar las interacciones.

Pasos para integrar:

1. Elija un marco/biblioteca de Python: Seleccione según las necesidades y preferencias del proyecto.
2. Instale los paquetes necesarios: Utilice `pip` para instalar las bibliotecas elegidas.
3. Defina los puntos finales de la API y los modelos: Estructurar la API utilizando las convenciones del marco respectivo.
4. Generar la especificación OpenAPI (si corresponde): Utilice las funciones integradas del marco o herramientas externas.
5. Integre la interfaz de usuario Swagger (opcional): Incluya una interfaz para la exploración e implementación interactivas.
6. Consumir APIs descritas por OpenAPI: Utilice el código cliente generado o bibliotecas como pyswagger.

Consideraciones adicionales:

* Versionado: OAS admite el versionado para la evolución de la API.
* Validación: Bibliotecas como `connexion` garantizan el cumplimiento de la especificación.
* Pruebas: La interfaz de usuario Swagger facilita las pruebas manuales, mientras que herramientas como `tavern` automatizan las pruebas de la API.

Ejemplo (Flask-RESTX):

Ejemplo 07:

```
1       from flask import Flask
 2      from flask_restx import Api, Resource
 3
 4      app = Flask(__name__)
 5      api = Api(app, title="Mi API", description="Un ejemplo simple de API")
 6
 7      @api.route("/hello")
 8      class HelloWorld(Resource):
 9          def get(self):
10              return {"message": "¡Hola, mundo!"}
11
12      if __name__ == "__main__":
13          app.run(debug=True)
```

Recuerde: Elija herramientas y enfoques que se adapten mejor a los requisitos específicos de su proyecto.

Explicación adicional:

* OpenAPI Specification (OAS): La especificación OpenAPI es un lenguaje de descripción de API que define un conjunto de términos y reglas para describir APIs REST. Se utiliza para documentar APIs existentes, diseñar APIs nuevas y generar código cliente.
* Swagger: Swagger es un conjunto de herramientas y recursos para implementar OpenAPI. Incluye una interfaz de usuario para exploración visual, generación de código y documentación.

Bibliotecas y marcos de Python para OpenAPI:

* Flask-RESTX: Una biblioteca para crear APIs REST con Flask. Se integra con Swagger para generar automáticamente la interfaz de usuario Swagger.
* apispec: Una biblioteca independiente del marco para generar documentación OpenAPI. Admite varios formatos de serialización, incluidos JSON, YAML y XML.
* connexion: Un marco para crear APIs REST a partir de especificaciones Swagger/OpenAPI. Asegura el cumplimiento de la especificación y proporciona características adicionales, como la seguridad y la autenticación.

Casos de uso:

∗ Documentación: La especificación OpenAPI se puede utilizar para documentar APIs existentes o diseñar APIs nuevas. La interfaz de usuario Swagger permite a los usuarios explorar e interactuar con las APIs de forma interactiva.

Web APIs final (desarrollo final)

Teniendo código template implementar dos apps, en ambas endpoints GET,POST,PUT,DELETE para manejo de vuelos, en la primera de ellas utilizando decoradores propios de swagger para exponerlos y en la segunda solo el modelo presentado en la parte inferior como archivo YAML.

```python
import flask
from flask_restful import Api
from flasgger import Swagger

app = flask.Flask(__name__)
api = Api(app)
swagger = Swagger(app)

@app.route("/flights")  # GET: Obtener todos los vuelos
@swagger.doc(tags=["Flights"], description="Obtiene todos los usuarios")
def get_usuarios():
    return {}

@app.route("/flights", methods=["POST"])  # POST: Crear un vuelo
@swagger.doc(tags=["Vuelos"], description="Crea un nuevo vuelo")
def create_usuario():
    data = flask.request.get_json()
    return {}

@app.route("/flights/<int:id>", methods=["PUT"])  # PUT: Actualizar un vuelo
@swagger.doc(tags=["Flights"], description="Actualiza un vuelo")
def update_usuario(id):
    data = flask.request.get_json()
    return {"usuario": data["nombre"]}

@app.route("/flights/<int:id>", methods=["DELETE"])  # DELETE: Eliminar un vuelo
@swagger.doc(tags=["Flighs"], description="Elimina un vuelo")
def delete_usuario(id):
    return {"id": id}

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

```yaml
openapi: 3.0.0
```

```yaml
info:
  title: Vuelos API Ceste
  description: Una API para consultar vuelos
  version: 1.0.0

paths:
  /flights:
    get:
      operationId: get_flights
      parameters:
      - name: origin
        in: query
        schema:
          type: string
      - name: destination
        in: query
        schema:
          type: string
      responses:
        '200':
          description: Lista de vuelos
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Flight'

components:
  schemas:
    Flight:
      type: object
      properties:
        origin:
          type: string
        destination:
          type: string
        departure_date:
          type: string
          format: date
        arrival_date:
          type: string
          format: date
        price:
          type: number
```