```
Desarrollo API REST {

    [HTTP/1.1 200 & Framework]

        < Flask Python Curl postman >

}
```

Introducción {

El proyecto se centra en el desarrollo de una API (Application Programming Interface) robusta utilizando Python y el framework Flask

}

Objetivos principales {

- Utilizar y diseñar APIs, aplicando la programación en la capa de aplicación de redes según el modelo OSI.
- Desarrollar fluidez y buenas prácticas en la programación con Python.
- Familiarizarse con herramientas relevantes de la industria como Git, Python, Flask, Postman y Curl.

}

¿Por qué utilizar entornos virtuales?

1. Nos permiten aislar dependencias, evitando conflictos en versiones de librerías y paquetes.
2. Creando un archivo `requirements.txt` podemos replicar las librerías de nuestro entorno virtual en otra computadora.
3. Cada aplicación cuenta con sus versiones de librerías sin afectar las de otras aplicaciones.

¿Cómo configuramos un entorno virtual?

1. Para instalar venv, usamos el siguiente comando,
   sudo apt install python3-venv
2. Luego, *Python3 -m venv .venv*
   Para activar el entorno virtual:
3. *source .venv/bin/activate* #Linux/Mac
   env\Scripts\activate #Windows
4. *pip install -r requirements.txt*
5. Y para desactivarlo:
   desactivate

Que es API REST?

Una API (Application Programming Interface) es un conjunto de reglas y protocolos que permiten que diferentes aplicaciones de software se comuniquen e intercambien datos entre sí.

REST (Representational State Transfer)

Es un estilo arquitectónico para diseñar APIs que se basa en los siguientes principios:

- Cliente servidor: La comunicación se divide entre un cliente que realiza las peticiones y un servidor que las responde.
- Sin estado (Stateless): Cada petición del cliente al servidor debe contener toda la información necesaria para ser procesada. El servidor no almacena información sobre las peticiones anteriores.
- Interfase uniforme: Se utilizan métodos HTTP() para realizar operaciones sobre los recursos (como películas en este caso) identificados por URLs.

Funcionamiento de una ‘API REST’;

* El cliente envía una petición HTTP al servidor a una URL específica (endpoint) que representa un recurso.

* La petición incluye el método HTTP que indica la acción a realizar (GET para obtener datos, POST para crear, PUT para actualizar, DELETE para eliminar)

* El servidor procesa la petición y devuelve una respuesta HTTP que incluye un código de estado (ej 200 ok, 404 Not Found) y, opcionalmente, un cuerpo de datos solicitado (en formato JSON o XML)

Creación API con Flask {

- Flask es un framework ligero de Python que permite la creación de APIs sencillas
- Nuestra API cuenta con las siguientes funcionalidades:

- Obtener todas las películas *(GET /peliculas)*
- Obtener una película específica *(GET /peliculas/<id>)*
- Agregar una película *(POST /peliculas)*
- Actualizar una película *(PUT /peliculas/<id>)*
- Eliminar una película *(DELETE /peliculas/<id>)*
- Buscar películas por título *(GET /peliculas/buscar?titulo=<termino>)*
- Sugerir una película aleatoria por género *(GET/peliculas/sugerir/<genero>)*
- Recomendar una película para un feriado *(GET /recomendar/<genero>)*

}

- El código principal de la API está en main.py, donde tenemos por ejemplo:

```python
@app.route('/peliculas/buscar', methods=['GET'])
def buscar_por_titulo():
    termino_busqueda = request.args.get('titulo', '').lower()
    resultado = [p for p in peliculas if termino_busqueda in p['titulo'].lower()]
    return jsonify(resultado)
```

- Para recibir datos con POST o PUT usamos *request.json*, que permite acceder a la información enviada por el cliente :

```python
def agregar_pelicula():
    nueva_pelicula = {
        'id': obtener_nuevo_id(),
        'titulo': request.json['titulo'],
        'genero': request.json['genero']
    }
    peliculas.append(nueva_pelicula)
    print(peliculas)
    return jsonify(nueva_pelicula), 201
```

}

Manejo de errores

- Es importante asegurarnos de que los datos recibidos sean válidos y poder manejar los errores de manera adecuada.
- Por ejemplo, al actualizar una película, verificamos que el campo id esté presente:

```python
def actualizar_pelicula(id):
    for pelicula in peliculas:
        if pelicula['id']==id:
            pelicula['titulo'] = request.json.get('titulo', pelicula['titulo'])
            pelicula['genero'] = request.json.get('genero', pelicula['genero'])
            return jsonify(pelicula)
    return jsonify({"mensaje" : "Pelicula no encontrada"}), 404
```

- Si no encuentra el id, devolvemos mensaje de error

Pruebas con pytest

- Para garantizar que nuestra API funcione correctamente, implementamos pruebas automatizadas con Pytest.

```python
def test_obtener_peliculas(mock_response):
    response = requests.get('http://localhost:5000/peliculas')
    assert response.status_code == 200
    assert len(response.json()) == 2
```

- Verificamos que la solicitud GET / peliculas responde con el código de estado 200

Usando cURL (client URL) {

- Herramienta de linea de comandos que permite hacer peticiones HTTP directamente desde la terminal
- Permite enviar datos y recibir respuestas sin necesidad de una interfaz gráfica o navegador
- Principales métodos HTTP que usamos en nuestra API:

1. GET (Obtener información)

```
curl -X GET http://localhost:5000/peliculas
```

2. POST (Agregar nueva información)

```
curl -X POST http://localhost:5000/peliculas \
```

3. PUT (Actualizar información existente)

```
curl -X PUT http://localhost:5000/peliculas/1 \
```

4. DELETE (Eliminar información)

```
curl -X DELETE http://localhost:5000/peliculas/1
```

CONSUMO DE UNA API EXTERNA {

Una API externa es una interfaz de programación de aplicaciones (API) proporcionada por un servicio o plataforma de terceros, que permite a otras aplicaciones acceder a sus funcionalidades y datos.

Integrar una API externa es útil porque permite extender las capacidades de una aplicación sin tener que desarrollar toda la funcionalidad desde cero. En este caso, la API de feriados proporciona información sobre los días festivos en Argentina, lo que permite a la API de películas ofrecer recomendaciones basadas en el contexto de los feriados.

}

API ‘feriados’ {

}

Interactuar con la API de feriados {

```python
def get_url(year, tipo=None):
    base_url = f"https://nolaborables.com.ar/api/v2/feriados/{year}"
    return f"{base_url}?tipo={tipo}" if tipo else base_url
```

}

Petición a la API de feriados {

```python
def fetch_holidays(self, tipo=None):
    try:
        response = requests.get(get_url(self.year, tipo))
        response.raise_for_status()  # Lanza error si HTTP != 200
        data = response.json()
        self.set_next(data)
    except requests.exceptions.RequestException as e:
        print(f"Error al obtener feriados: {e}")
        self.holiday = None
```

}

Ejemplos de peticiones a la API de feriados{

```
  Para obtener el próximo feriado sin filtrar por tipo:

  response =
  requests.get("https://nolaborables.com.ar/api/v2/feriados/2025")


  Para obtener el próximo feriado de tipo "inamovible":

  response =
  requests.get("https://nolaborables.com.ar/api/v2/feriados/2025?tipo=inamovible")


}
```

Integración de la API de feriados con la API de películas

Se creó un nuevo endpoint en la API de películas:

```python
@app.route('/recomendar/<string:genero>', methods=['GET'])
def recomendar_feriado(genero):

    try:
        tipo_feriado = request.args.get('tipo', None)  # Obtener tipo desde query params
        next_holiday = NextHoliday()
        next_holiday.fetch_holidays(tipo=tipo_feriado)  # Pasar el tipo

        if not next_holiday.holiday:
            return jsonify({'error': 'No se encontraron feriados'}), 404

        opciones = [p for p in peliculas if p['genero'].lower() == genero.lower()]
        if not opciones:
            return jsonify({'error': 'No hay películas de este género'}), 404

        pelicula = random.choice(opciones)
        return jsonify({
            'feriado': next_holiday.holiday['motivo'],
            'fecha': f"{next_holiday.holiday['dia']}/{next_holiday.holiday['mes']}",
            'tipo': next_holiday.holiday['tipo'],  # Incluir tipo en respuesta
            'pelicula': pelicula
        }), 200
    except Exception as e:
        return jsonify({'error': f'Error interno: {str(e)}'}), 500
```

test.py {

El archivo test.py es un script de prueba que utiliza la librería requests para interactuar con una API REST es un script manual que realiza solicitudes HTTP a un servidor local (http://localhost:5000) y muestra las respuestas.

El script implementa una serie de pruebas manuales para comprobar el funcionamiento de los endpoints de la API. Está estructurado para probar las operaciones CRUD y algunas funcionalidades adicionales

}

Cómo se están comprobando los resultados? {

Código de estado HTTP

El script evalúa los códigos de respuesta HTTP para determinar si la solicitud fue exitosa o fallida:

200 OK → La operación fue exitosa.
201 Created → La creación fue exitosa.
Cualquier otro código → La operación falló.

Contenido de la respuesta

Si el estado es exitoso, intenta convertir el cuerpo de la respuesta a JSON e imprimir los valores clave (como id, titulo, genero). Si no puede convertirlo, imprime el texto en bruto.

}

✅ ¿Qué es pytest?{

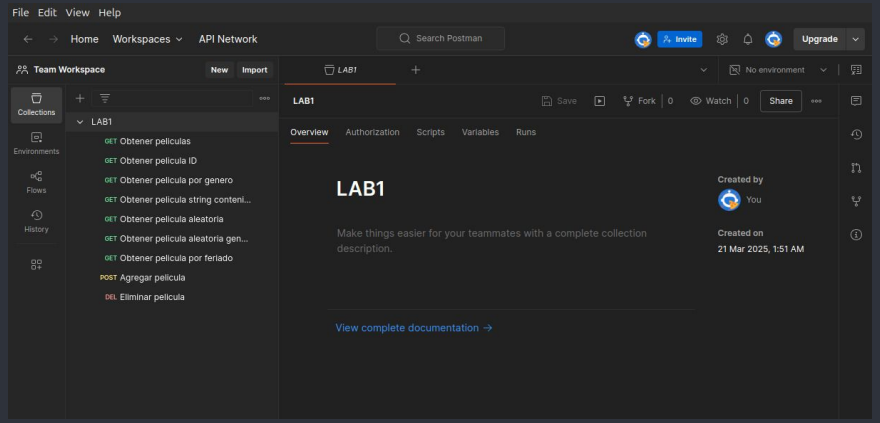
Pytest es un framework de testing para Python que permite escribir tests de manera simple y eficiente.
Está diseñado para que las pruebas sean fáciles de leer y mantener, y ofrece herramientas para manejar fixtures, mocks, y parametrización.

✅ ¿Qué es requests_mock?

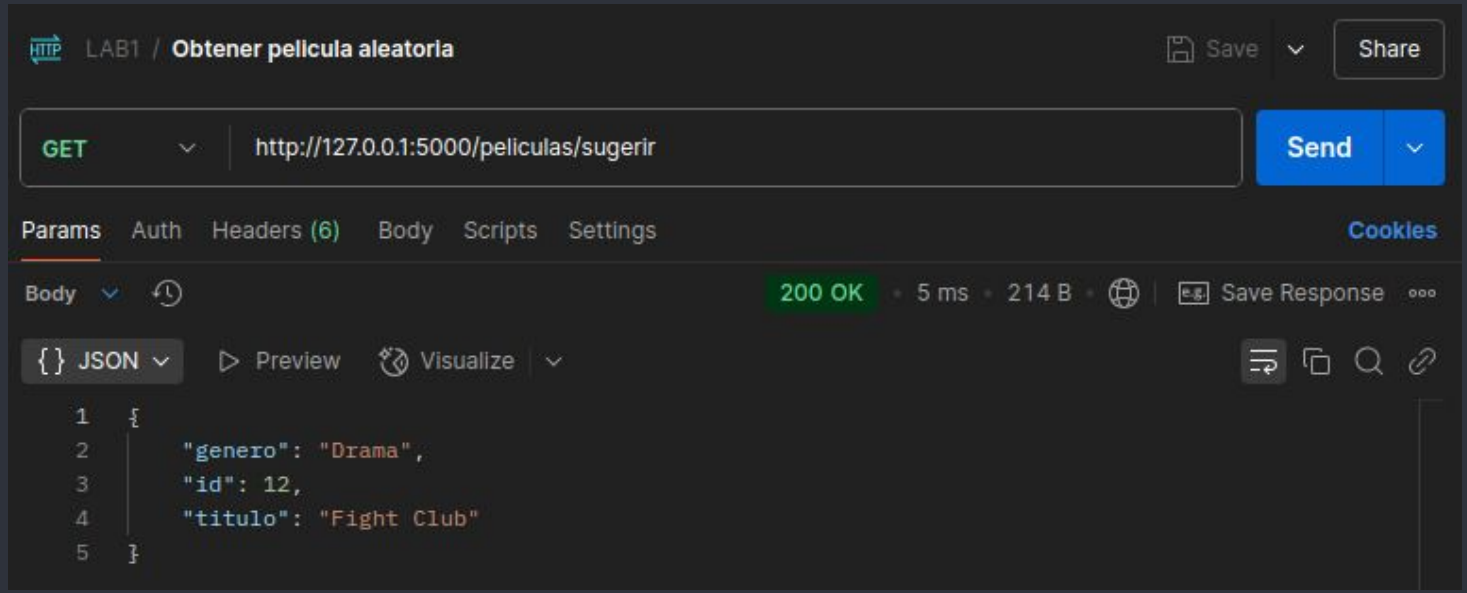
requests_mock es una librería que extiende requests para que puedas simular respuestas HTTP en tus pruebas, sin necesidad de tener un servidor real corriendo.
Simula respuestas de API REST.
Permite probar el comportamiento de una función o endpoint sin hacer llamadas reales a un servidor.
Acelera las pruebas y las hace predecibles, ya que las respuestas están controladas.
}

¿Qué es POSTMAN?

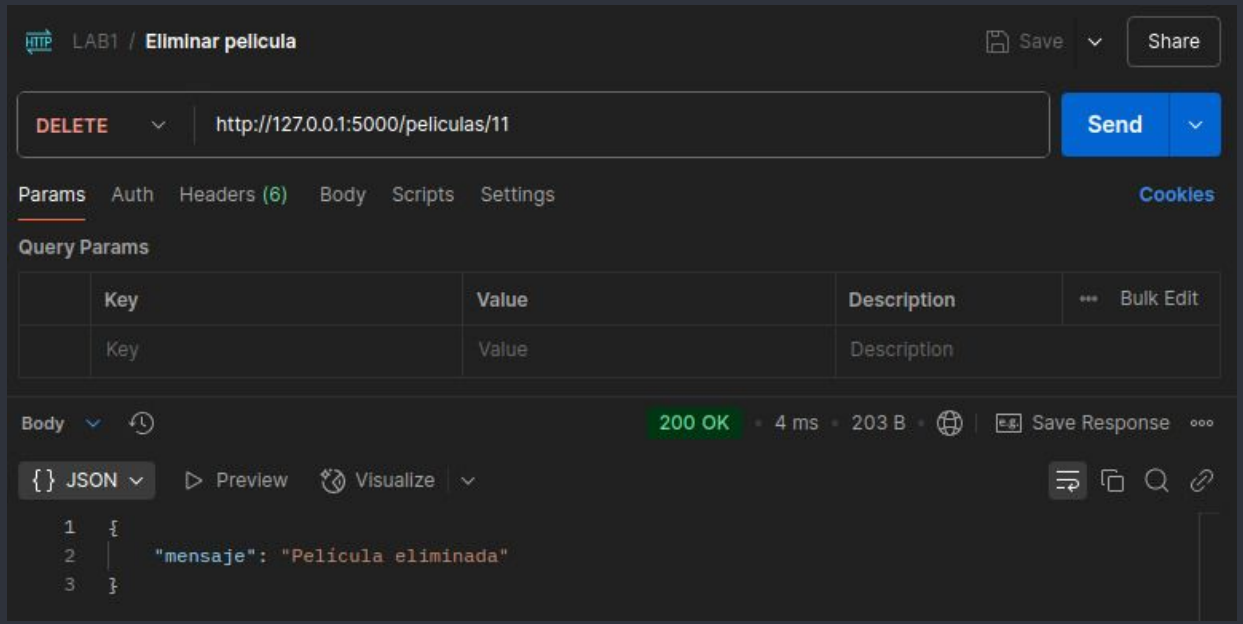
Postman es una herramienta que nos simplifica mediante una interfaz gráfica, probar y documentar APIs REST.



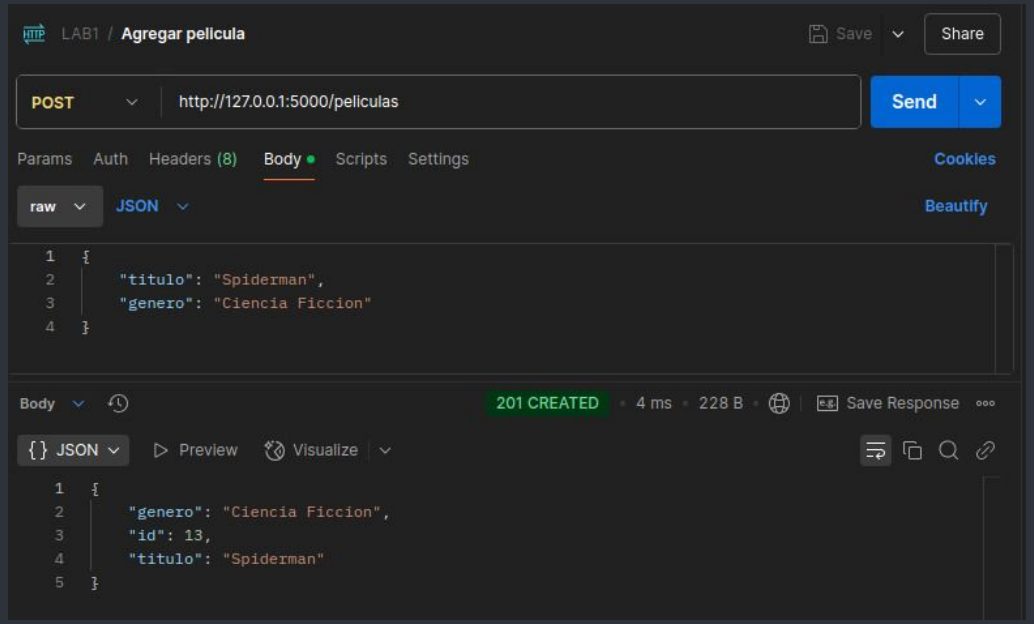
Algunas pruebas en POSTMAN



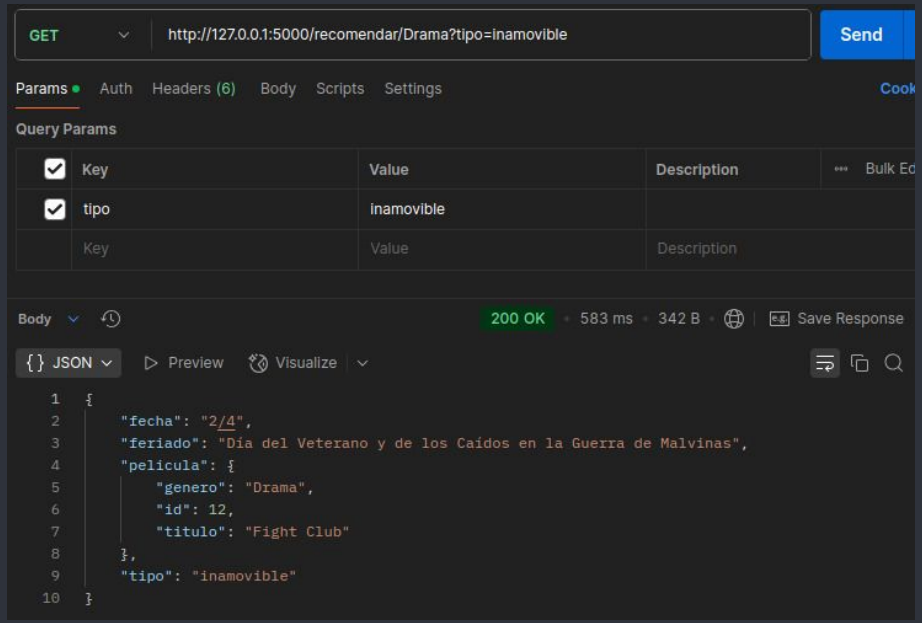
Algunas pruebas en POSTMAN



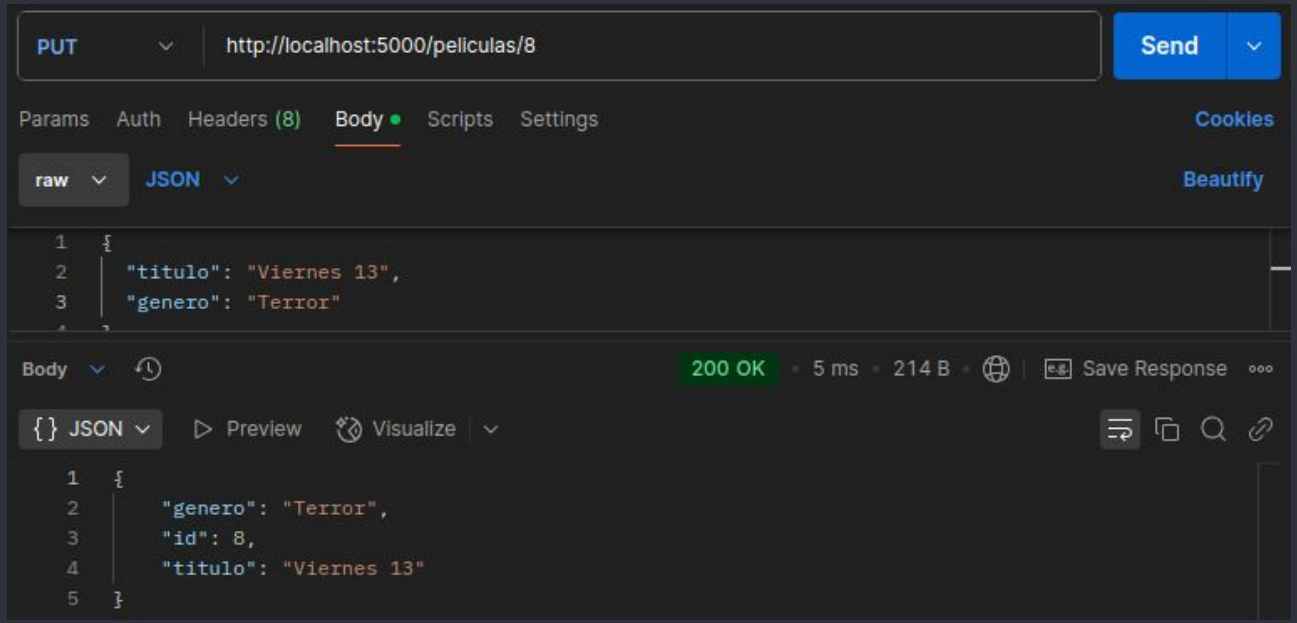
Algunas pruebas en POSTMAN



Algunas pruebas en POSTMAN



Algunas pruebas en POSTMAN



Diferencias entre las distintas pruebas

Las diferencias están en el enfoque y el nivel de automatización que maneja cada prueba.

POSTMAN

- Nos permite hacer pruebas manuales de solicitudes HTTP como GET, PUT, DELETE, etc.
- Es ideal para hacer pruebas simples y ver cómo responde la API.
- Responde en formato JSON.
- Cada prueba es manual e individual.

TEST.py

- Es un script manual.
- Funciona más rápido que Postman.
- Nos permite verificar que responde la API.
- En caso de fallo, hay que revisar manualmente los errores.

PYTEST

- Nos permite testear de manera más automatizada.
- Podemos realizar las pruebas sin levantar el servidor.
- Nos ofrece información detallada de los errores del código.

Conclusiones {

01 Uso de Apis

Permiten la comunicación de forma eficiente entre distintos sistemas.

02 Framework Flask

Ligero y sencillo que facilita el desarrollo de APIS Rest

03 Evaluación de la API

Desarrollamos una API que responde a solicitudes HTTP para la gestión de peliculas.

}

Conclusiones {

04 Posible mejora futura

Utilizar un sistema de base de datos en lugar de listas.

05 Sistema de versionado

Permite un flujo de trabajo en equipo organizado mediante la utilización de ramas de trabajo.

}

Gracias por la atención {

Integrantes:

Fonseca Gonzalo
Fernandez Bodereau Constanza
Aravena Lihuel

}