

# Presentación Lab 1

Desarrollo de una API

Integrantes: Francisco Villan, Jeremías Almeira, Máximo Agüero, Pedro Pedernera

# Parte 1:

Configuración del entorno e instalación de librerías

# RESTful API

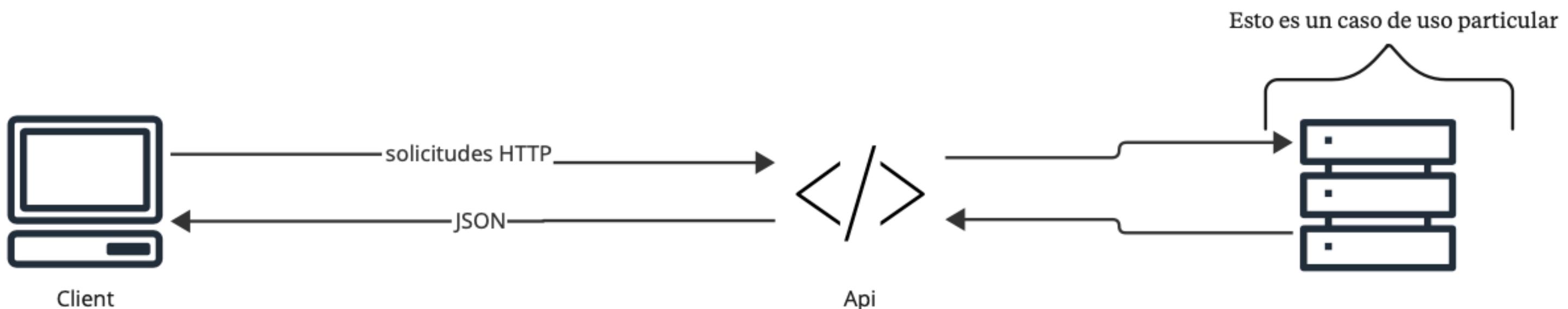
Una API RESTful (API de Transferencia de Estado Representacional)

Es una interfaz de programación de aplicaciones (**API**) que sigue los principios de diseño de la arquitectura **REST**, permitiendo que diferentes sistemas de software se comuniquen y comparten datos de forma flexible y eficiente a través de Internet, utilizando el protocolo HTTP

¿Qué es REST?

**REST** es una arquitectura de desarrollo web que puede ser utilizada en cualquier cliente HTTP

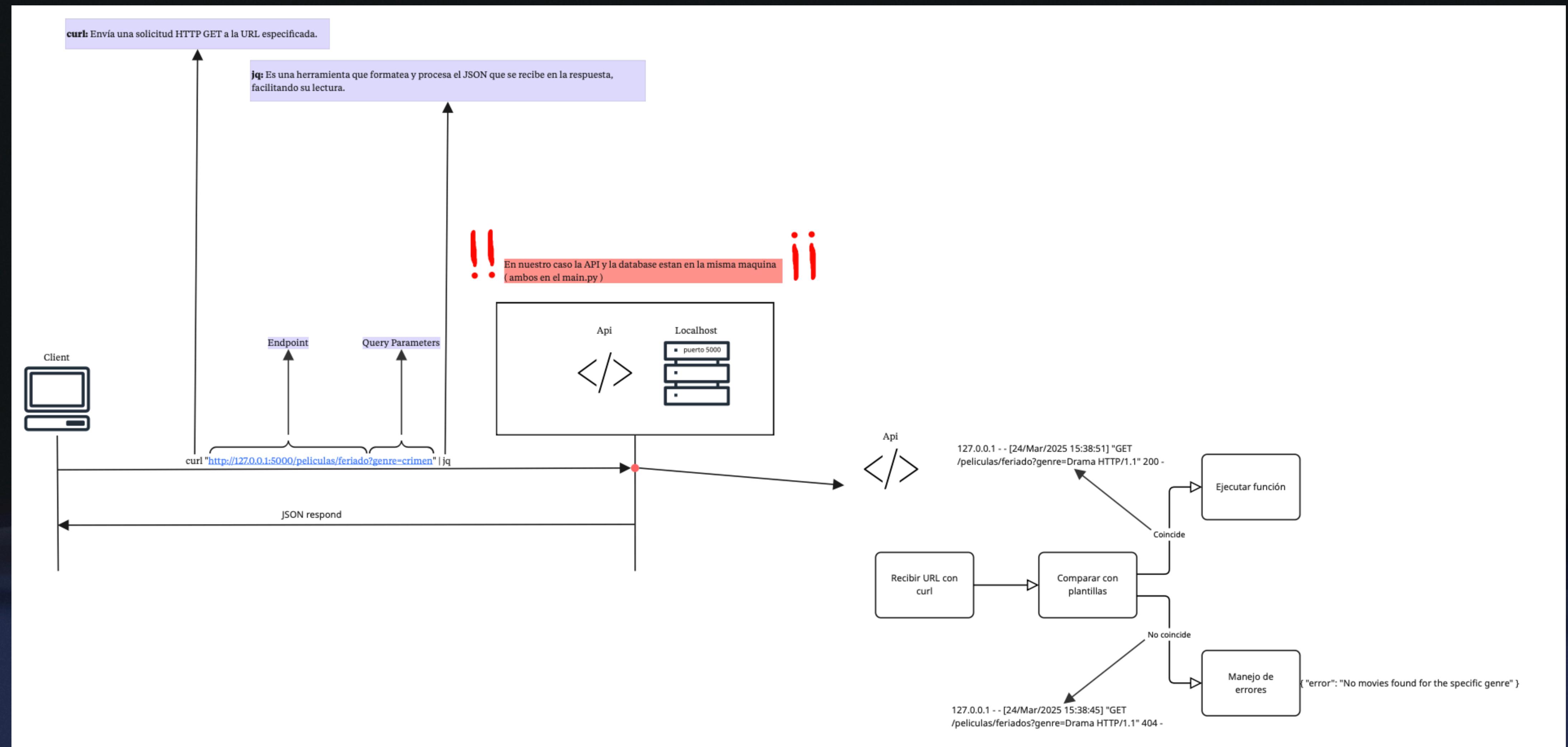
```
curl -X POST http://localhost:5000/peliculas \  
-H "Content-Type: application/json" \  
-d '{"titulo": "Nuevo Titulo", "genero":  
"Acción"}'
```



La principal ventaja de esta arquitectura es que ha aportado a la web una mayor **escalabilidad**.

Esta separación de responsabilidades tiene varias ventajas, como la **seguridad** (el cliente no tiene acceso directo a la base de datos), la **modularidad** y la **fácilidad para escalar o realizar cambios** en cualquiera de las capas sin afectar a las otras.

# RESTful API



# Entorno Virtual De Programación

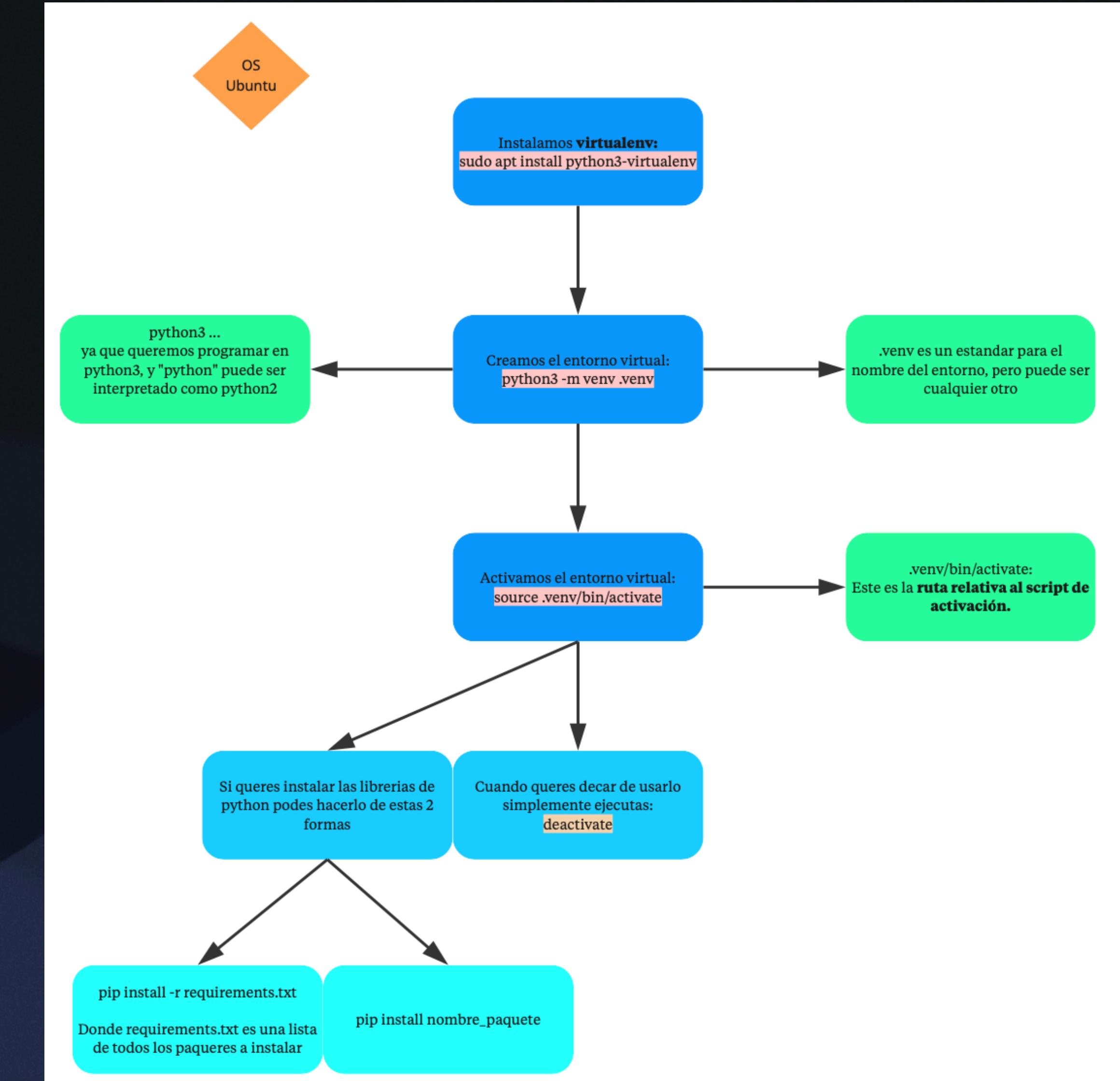
**Aislamiento de dependencias:** Permite que cada proyecto tenga su propio conjunto de bibliotecas y paquetes, evitando conflictos entre diferentes versiones de dependencias.

Por ejemplo, un proyecto podría necesitar la versión 2.0 de una librería, mientras que otro podría requerir la 3.0. El entorno virtual garantiza que ambos puedan coexistir sin problemas.

**Reproducibilidad:** Facilita que otras personas puedan replicar exactamente las mismas condiciones de desarrollo en sus máquinas, lo cual es crucial para la colaboración y la depuración de problemas. Esto se logra porque el entorno virtual registra y administra las dependencias usadas en el proyecto.

**Evita problemas a nivel del sistema:** Instalar paquetes globalmente (a nivel del sistema) puede causar desorden o incluso romper configuraciones existentes. Con un entorno virtual, todo queda contenido y no afecta al sistema operativo o a otros proyectos.

**Portabilidad:** Si necesitas mover un proyecto de una máquina a otra, puedes recrear el mismo entorno en pocos pasos, asegurándote de que todo funcione como esperas sin importar dónde se ejecute.



# Parte 2:

Creación de una API con Flask

# Flask:

¿Qué es y cómo funciona?

- Flask es un microframework para Python, diseñado para crear aplicaciones web y APIs de forma sencilla y flexible.
- Permite definir rutas y manejar métodos HTTP (GET, POST, PUT, DELETE) de manera intuitiva.

```
1  from flask import Flask, jsonify, request
2
3  app = Flask(__name__)
4
5  // lista de peliculas ...
6
7  // funciones ...
8
9  app.add_url_rule('/peliculas', 'obtener_peliculas', obtener_peliculas, methods=['GET'])
10 app.add_url_rule('/peliculas<int:id>', 'obtener_pelicula', obtener_pelicula, methods=['GET'])
11 app.add_url_rule('/peliculas', 'agregar_pelicula', agregar_pelicula, methods=['POST'])
12 app.add_url_rule('/peliculas<int:id>', 'actualizar_pelicula', actualizar_pelicula, methods=['PUT'])
13 app.add_url_rule('/peliculas<int:id>', 'eliminar_pelicula', eliminar_pelicula, methods=['DELETE'])
14
15 if __name__ == '__main__':
16     app.run()
17
```

# Funcionalidades Básicas de Nuestra API

Se busca el manejo eficiente de una base de datos de películas, facilitando operaciones como la búsqueda, actualización, eliminación y listado por criterios.

## Gestión de Películas:

- Búsqueda por ID: Permite obtener los detalles de una película específica.
- Actualización y Eliminación: Facilita la modificación o eliminación de registros existentes.

## Consultas Avanzadas:

- Listado por Género: Filtra las películas según el género solicitado.
- Búsqueda por Título: Encuentra películas que contienen un texto específico en su título.

## Sugerencias

- Aleatoria: Recomienda una película de manera aleatoria.
- Por Género: Sugiere una película basada en el género proporcionado.

# cURL:

¿Qué es y cómo funciona?

- Es una herramienta de línea de comandos para realizar solicitudes HTTP a servidores web.
- Permite enviar solicitudes a la API y recibir respuestas en formato JSON.
- Beneficios:
  - Rápida verificación de endpoints sin necesidad de una interfaz gráfica.
  - Ideal para pruebas y depuración de la API en entornos de desarrollo.

Ejemplos Prácticos:

HTTP Method	Descripción	Comando cURL
GET	Retrieve information about a resource.	<code>curl -X GET http://localhost:5000/api/resource</code>
POST	Create a new resource on the server	<code>curl -X POST -H "Content-Type: application/json" -d '{"key": "value"}' http://localhost:5000/api/resource</code>
PUT	Completely update an existing resource.	<code>curl -X PUT -H "Content-Type: application/json" -d '{"key": "new value"}'</code>
PATCH	Partially update an existing resource.	<code>curl -X PATCH -H "Content-Type: application/json" -d '{"key": "partial value"}'</code>
DELETE	Delete a resource.	<code>curl -X DELETE http://localhost:5000/api/resource/1</code>
HEAD	Retrieve only the headers from a resource response.	<code>curl -I http://localhost:5000/api/resource/1</code>
OPTIONS	Retrieve communication options available for a resource.	<code>curl -X OPTIONS http://localhost:5000/api/resource</code>

## Estructura de la API main.py

```
1  from flask import Flask, jsonify, request
2  import random
3  from proximo_feriado import NextHoliday
4
5  app = Flask(__name__)
6  peliculas =
7      {'id': 1, 'titulo': 'Indiana Jones', 'genero': 'Acción'},
8      {'id': 2, 'titulo': 'Star Wars', 'genero': 'Acción'},
9      {'id': 3, 'titulo': 'Interstellar', 'genero': 'Ciencia ficción'},
10     {'id': 4, 'titulo': 'Jurassic Park', 'genero': 'Aventura'},
11     {'id': 5, 'titulo': 'The Avengers', 'genero': 'Acción'},
12     {'id': 6, 'titulo': 'Back to the Future', 'genero': 'Ciencia ficción'},
13     {'id': 7, 'titulo': 'The Lord of the Rings', 'genero': 'Fantasía'},
14     {'id': 8, 'titulo': 'The Dark Knight', 'genero': 'Acción'},
15     {'id': 9, 'titulo': 'Inception', 'genero': 'Ciencia ficción'},
16     {'id': 10, 'titulo': 'The Shawshank Redemption', 'genero': 'Drama'},
17     {'id': 11, 'titulo': 'Pulp Fiction', 'genero': 'Crimen'},
18     {'id': 12, 'titulo': 'Fight Club', 'genero': 'Drama'}
19 ]
```



Base de datos de nuestra API de películas

# Algunas funciones de la API main.py

## Método PUT

```
79 def update_movie(id):
80     """
81     Update the title and genre of an existing movie by ID.
82
83     Expects a JSON body with 'titulo' (str) and 'genero' (str).
84     If the given ID is invalid (out of range), returns a 404
85     error message. Otherwise, updates the movie in place.
86
87     Args:
88         id (int): The unique ID of the movie to update.
89
90     Returns:
91         Response: A JSON response with the updated movie or an
92         error message if not found.
93     """
94
95     if id <= 0 or id > len(peliculas):
96         return jsonify({"error": "Movie could not be found"}), 404
97     updated_movie = {
98         'id': id,
99         'titulo': request.json['titulo'],
100        'genero': request.json['genero']
101    }
102    peliculas[id - 1].update(updated_movie)
103    return jsonify(updated_movie)
```

```
192 def get_movie_by_title_keyword():
193     """
194     Return movies whose titles contain a given keyword.
195
196     Reads the 'kw' (keyword) query parameter from the request.
197     Searches each movie title (case-insensitive) for the keyword.
198     If none is provided or no matches are found, returns an error.
199
200     Query Parameters:
201         kw (str): The keyword to look for in movie titles.
202
203     Returns:
204         Response: A JSON list of matching movies or an error
205         message if none match.
206     """
207     kw = request.args.get("kw")
208     if not kw:
209         return jsonify({'error': 'Missing keyword'}), 404
210
211     match_list = []
212     new_id = 1
213     for movie in peliculas:
214         if kw.lower() in movie['titulo'].lower():
215             copy_aux = movie.copy()
216             copy_aux['id'] = new_id
217             match_list.append(copy_aux)
218             new_id += 1
219
220     if not match_list:
221         return jsonify({'error': f'There is no movie with keyword: "{kw}"'}), 404
222
223     return jsonify(match_list)
```

## Método GET

# Parte 3 :

Consumo de una API externa

“¿Cómo podemos transformar  
una API básica en una  
experiencia dinámica  
integrando datos externos?”

# Integración con la API externa: Feriados

## Paso 1: Construcción de la URL

Mediante la función get\_url(year), generamos la dirección exacta de la API según el año que nos interese.

```
def get_url(year):  
    return f"https://nolaborables.com.ar/api/v2/feriados/{year}"
```

# Integración con la API externa: Feriados

## Paso 2: Solicitud HTTP con `requests.get`

En el método que maneja la obtención de feriados (por ejemplo, `next_holiday`), realizamos una llamada a la API

```
response = requests.get(get_url(self.year))
data = response.json()
```

Aquí, `data` contendrá la lista de feriados en formato JSON que nos devuelve la API externa

# Integración con la API externa: Feriados

## Paso 3: Filtrado y Procesamiento de Datos

Una vez que tenemos data, lo recorremos para filtrar los feriados según el criterio deseado (por ejemplo, buscar el primero que ocurra después de la fecha actual o filtrar por tipo de feriado). Al encontrar el feriado que cumpla las condiciones, lo almacenamos en nuestro objeto:

```
for holiday in data:  
    # Lógica de filtrado, por ejemplo, si coincide con el tipo solicitado  
    # o si la fecha es posterior a hoy  
    # ...  
    self.holiday = holiday  
    break
```

# Integración con la API externa: Feriados

## Paso 4: Mostrar los Resultados

Por último, al momento de renderizar o imprimir la información (por ejemplo, en render), usamos los datos almacenados en self.holiday para mostrar la fecha, el motivo o el tipo de feriado, según nos interese:

```
def render(self):
    if self.loading:
        print("Buscando...")
    else:
        print("Próximo feriado")
        print(self.holiday['motivo'])
        print("Fecha:")
        print(day_of_week(self.holiday['dia'], self.holiday['mes'], self.year))
        print(self.holiday['dia'])
        print(months[self.holiday['mes'] - 1])
        print("Tipo:")
        print(self.holiday['tipo'])
```

# Integración con la API externa: Feriados

## Agregando nuevas funcionalidades

```
class NextHoliday:
    def __init__(self):
        self.loading = True
        self.year = date.today().year
        self.holiday = None

    def set_next(self, holidays, type=None):
        now = date.today()
        today = {
            'day': now.day,
            'month': now.month
        }

        if type is None:
            holiday = next(
                (h for h in holidays if
                 h['mes'] == today['month'] and
                 h['dia'] > today['day'] or
                 h['mes'] > today['month']),
                holidays[0]
            )
        else:
            holiday = next(
                (h for h in holidays if
                 (h['mes'] == today['month'] and
                  h['dia'] > today['day'] or
                  h['mes'] > today['month']) and
                 h['tipo'] == type),
                holidays[0]
            )

        self.loading = False
        self.holiday = holiday
```

La modificación principal en la función **set\_next** radica en incorporar el parámetro opcional **type=None**, lo que permite filtrar el próximo feriado según su tipo. A grandes rasgos, esto funciona así:

- 1. Nuevo parámetro type:** Permite filtrar por tipo de feriado o ignorarlo si es None.
- 2. Filtrado condicional:** Busca el primer feriado posterior a hoy, con o sin criterio de tipo.
- 3. Generalización en NextHoliday:** Amplía la flexibilidad para futuras extensiones de filtrado y búsqueda.

A futuro, se podrían agregar más parámetros (por ejemplo, filtrar por mes, o por día festivo) y seguir la misma estructura de filtrado condicional, haciendo que **set\_next** maneje múltiples tipos de consultas.

# Integración con la API externa: Feriados

Activamos el entorno y corremos main.py

```
[pedropedernera@Pedros-MacBook-Pro Laboratorios % source .venv/bin/activate
(.venv) pedropedernera@Pedros-MacBook-Pro Laboratorios % cd redes25lab1g09
(.venv) pedropedernera@Pedros-MacBook-Pro redes25lab1g09 % python3 main.py
 * Serving Flask app 'main'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
127.0.0.1 - - [26/Mar/2025 20:48:48] "GET /peliculas/holiday_type?genre=Drama&h_type=trasladable HTTP/1.1" 200 -
```

Una vez activado el servidor mandamos el comando curl:

**curl "http://127.0.0.1:5000/peliculas/holiday\_type?genre=Drama&h\_type=trasladable" | jq**

```
pedropedernera@Pedros-MacBook-Pro redes25lab1g09 % curl "http://127.0.0.1:5000/peliculas/holiday_type?genre=Drama&h_type=trasladable" | jq
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload   Total Spent   Left  Speed
100  296  100  296    0     0  478      0 --:--:-- --:--:--:--:--  478
{
  "feriado": {
    "dia": 17,
    "id": "martin-guemes",
    "info": "https://es.wikipedia.org/wiki/Mart%C3%ADn_Miguel_de_G%C3%BCemes",
    "mes": 6,
    "motivo": "Paso a la Inmortalidad del Gral. Don Martín Güemes",
    "original": "17-06",
    "tipo": "trasladable"
  },
  "pelicula": {
    "genero": "Drama",
    "id": 2,
    "titulo": "Fight Club"
  }
}
```

Conexión a la API de Feriados:

Éxito 100%

# Parte 4:

## Evaluación de la API

# test\_pytest.py

```
def test_agregar_pelicula(mock_response):
    nueva_pelicula = {'titulo': 'Película de prueba', 'genero': 'Acción'}
    response = requests.post('http://localhost:5000/peliculas',
                             json=nueva_pelicula)
    assert response.status_code == 201
    assert response.json()['id'] == 3

def test_obtener_detalle_pelicula(mock_response):
    response = requests.get('http://localhost:5000/peliculas/1')
    assert response.status_code == 200
    assert response.json()['titulo'] == 'Indiana Jones'

def test_actualizar_detalle_pelicula(mock_response):
    datos_actualizados = {'titulo': 'Nuevo título', 'genero': 'Comedia'}
    response = requests.put('http://localhost:5000/peliculas/1',
                           json=datos_actualizados)
    assert response.status_code == 200
    assert response.json()['titulo'] == 'Nuevo título'

def test_eliminar_pelicula(mock_response):
    response = requests.delete('http://localhost:5000/peliculas/1')
    assert response.status_code == 200
```

```
@pytest.fixture
def mock_response():
    with requests_mock.Mocker() as m:
        # Simulamos la respuesta para obtener todas las películas
        m.get('http://localhost:5000/peliculas', json=[
            {'id': 1, 'titulo': 'Indiana Jones', 'genero': 'Acción'},
            {'id': 2, 'titulo': 'Star Wars', 'genero': 'Acción'}
        ])

        # Simulamos la respuesta para agregar una nueva película
        m.post(
            'http://localhost:5000/peliculas',
            status_code=201,
            json={'id': 3, 'titulo': 'Película de prueba', 'genero': 'Acción'}
        )

        # Simulamos la respuesta para obtener detalles de una película específica
        m.get(
            'http://localhost:5000/peliculas/1',
            json={'id': 1, 'titulo': 'Indiana Jones', 'genero': 'Acción'}
        )

        # Simulamos la respuesta para actualizar los detalles de una película
        m.put(
            'http://localhost:5000/peliculas/1',
            status_code=200,
            json={'id': 1, 'titulo': 'Nuevo título', 'genero': 'Comedia'}
        )

        # Simulamos la respuesta para eliminar una película
        m.delete('http://localhost:5000/peliculas/1', status_code=200)

    yield m
```

# test.py

```
# Obtener peliculas por genero
genre_list = ['Drama', 'Acción', 'Ciencia ficción',
              'Aventura', 'Fantasía', 'Crimen']
for index in genre_list:
    response = requests.get(
        f'http://localhost:5000/peliculas/get_movie_by_genre?genre={index}')
    if response.status_code == 200:
        print(f"Película de {index} obtenida correctamente\n")
    else:
        print(f"Error al buscar películas del género {index}\n")

# Obtener las películas que en el título contenga determinada letra o palabra
keyword = random.choice(string.ascii_lowercase)
response = requests.get(
    f'http://localhost:5000/peliculas/get_movie_by_title_keyword?kw={keyword}')
if response.status_code == 200:
    for index in response.json():
        print(f"Películas obtenidas con el keyword {keyword}")
        print(
            f"ID: {index['id']}, "
            f"Título: {index['titulo']}, "
            f"Género: {index['genero']}\n"
        )
    else:
        print(f"No se obtuvieron películas con el keyword {keyword}\n")
```

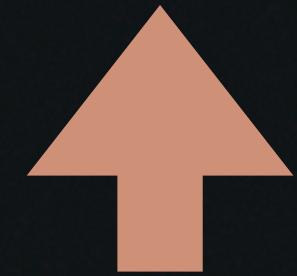
```
# Actualizar los detalles de una película
id_pelicula = 1 # ID de la película a actualizar
datos_actualizados = {
    'titulo': 'Nuevo título',
    'genero': 'Comedia'
}
response = requests.put(
    f'http://localhost:5000/peliculas/{id_pelicula}',
    json=datos_actualizados
)
if response.status_code == 200:
    pelicula_actualizada = response.json()
    print("Película actualizada:")
    print(
        f"ID: {pelicula_actualizada['id']}, "
        f"Título: {pelicula_actualizada['titulo']}, "
        f"Género: {pelicula_actualizada['genero']} "
    )
else:
    print("Error al actualizar la película.")
print()

# Eliminar una película
id_pelicula = 1 # ID de la película a eliminar
response = requests.delete(
    f'http://localhost:5000/peliculas/{id_pelicula}'
)
if response.status_code == 200:
    print("Película eliminada correctamente.\n")
else:
    print("Error al eliminar la película.\n")
```

# test\_pytest\_main.py

```
import pytest
from main import app

@pytest.fixture
def client():
    app.testing = True # activa el modo de testing de Flask
    with app.test_client() as client:
        yield client
```



Cliente de pruebas para Flask

```
def test_eliminar_pelicula(client):
    response = client.delete('/peliculas/1')
    assert response.status_code == 200
    data = response.get_json()
    assert 'mensaje' in data
    response = client.get('/peliculas/1')
    assert 'mensaje' in data
    assert response.status_code == 404

def test_get_movie_by_genre(client):
    genre_list = ['Drama', 'Acción', 'Ciencia ficción',
                  'Aventura', 'Fantasía', 'Crimen']
    for index in genre_list:
        response = client.get(f'/peliculas/get_movie_by_genre?genre={index}')
        assert response.status_code == 200
    # Verificar que sea una lista de películas solo del género solicitado
    data = response.get_json()
    for element in data:
        assert index == element['genero']

def test_get_movie_by_title_keyword(client):
    keyword = "Th"
    response = client.get(f'/peliculas/get_movie_by_title_keyword?kw={keyword}')
    assert response.status_code == 200
    data = response.get_json()
    # Verificar que keyword efectivamente está en el título de las películas encontradas
    for element in data:
        assert keyword.lower() in element['titulo'].lower()

def test_random_movie(client):
    response = client.get('/peliculas/random')
    assert response.status_code == 200
```

# Diferencias Clave entre *test.py*, *test\_pytest.py* y *test\_pytest\_main.py*

## 1. Tipo de pruebas que hacen:

*test\_pytest.py*: Simula las respuestas de la API.

*test.py*: Consume la API real, pero no realiza pruebas unitarias.

*test\_pytest\_main.py*: Interactua con un servidor Flask real.

## 2. El propósito de cada test:

*test\_pytest.py*: Pruebas unitarias y validación de una API simulada.

*test.py*: Ejemplo de uso de la API con interacción por consola

*test\_pytest\_main.py*: Realización de pruebas a nivel de servidor Flask

## 3. Interacción con el servidor:

Tanto *test\_pytest.py* como *test.py* interactúan con endpoints específicos de una API.

Mientras que *test\_pytest\_main.py* valida directamente la lógica interna de la aplicación web.

# Postman

Es una herramienta de desarrollo muy popular que se utiliza para probar APIs.

Permite enviar solicitudes HTTP a servidores web y ver las respuestas de manera visual y organizada, sin necesidad de escribir código ni usar la terminal.

## **¿Para qué lo usamos en este laboratorio?**

- Para probar los endpoints de nuestra API de películas.
- Para verificar que las funcionalidades como obtener, actualizar o eliminar funcionen correctamente.
- Para reproducir las pruebas que también se hacen con curl o pytest, pero con una interfaz visual amigable.
- Para simular solicitudes que incluyen datos en el cuerpo (por ejemplo, POST o PUT con JSON).

# Ejemplos de uso de Postman

The screenshot shows the Postman application interface. On the left, there's a sidebar with icons for Collections, Environments, Flows, and History. The main area displays a collection named "API peliculas" with various endpoints like "get\_all\_movies", "get\_movie", etc. The selected endpoint is "GET get\_movie\_by\_title\_keyword". The top bar shows the URL "http://127.0.0.1:5000/peliculas/get\_movie\_by\_title\_keyword?kw=th" and a "Send" button. Below the URL, tabs for Params, Authorization, Headers (6), Body, Scripts, and Settings are visible, with "Params" being the active tab. Under "Query Params", there's a table with one row: "kw" with value "th". The "Headers (5)" tab shows the following headers:

- Content-Type: application/json
- Accept: \*/\*
- Cache-Control: no-cache
- Postman-Token: 1234567890abcdef1234567890abcdef
- Host: 127.0.0.1:5000

The "Body" tab shows a JSON response:

```
1 [  
2 {  
3   "genero": "Acción",  
4   "id": 1,  
5   "titulo": "The Avengers"  
6 },  
7 {  
8   "genero": "Ciencia ficción",  
9   "id": 2,  
10  "titulo": "Back to the Future"  
11 },  
12 {  
13   "genero": "Fantasía",  
14   "id": 3,  
15   "titulo": "The Lord of the Rings"  
16 },  
17 {  
18   "genero": "Acción",  
19   "id": 4,  
20   "titulo": "The Dark Knight"  
21 }]
```

The status bar at the bottom indicates a "200 OK" response with 3 ms latency and 483 B size.

# Ejemplos de uso de Postman

The screenshot shows the Postman application interface. On the left, there's a sidebar with icons for Collections, Environments, Flows, and History. The main area displays a collection named "API peliculas". A specific endpoint, "PUT update\_movie", is selected. The request details show a PUT method and the URL `http://127.0.0.1:5000/peliculas/2`. The "Body" tab is active, showing raw JSON input:

```
1 {
2   "titulo": "Zodiac",
3   "genero": "Crimen"
4 }
```

Below the request details, the response section shows a green "200 OK" status with a response time of 4 ms and a body size of 210 B. The response body is also displayed in JSON format:

```
1 {
2   "genero": "Crimen",
3   "id": 2,
4   "titulo": "Zodiac"
5 }
```

# Conclusión

# Resultados y Mejoras

## **Evaluación de la API:**

- Validación exitosa mediante pruebas automatizadas y manuales (Postman, curl, pytest).
- La API cumple con los criterios de funcionalidad, integración y buenas prácticas.

## **Posibles Mejoras o Ampliaciones:**

- Optimización del rendimiento y escalabilidad.
- Incorporar nuevos filtros de búsqueda o funcionalidades adicionales.
- Ampliar la documentación y casos de prueba para mayor cobertura.

¡Gracias por su atención!