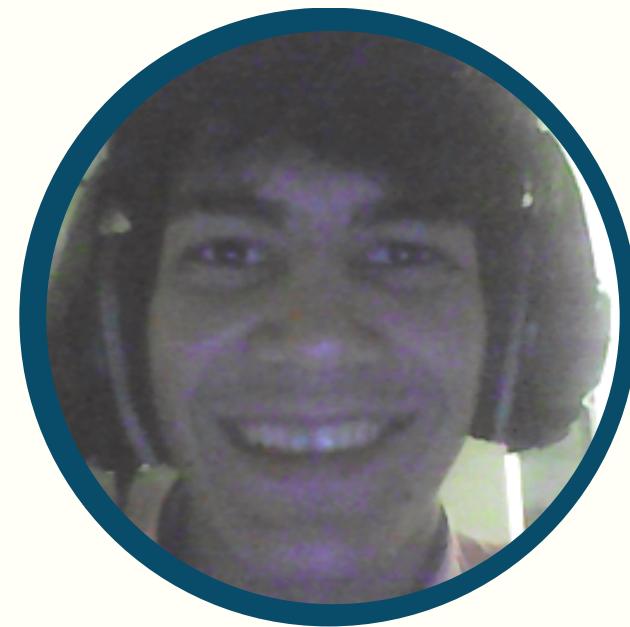
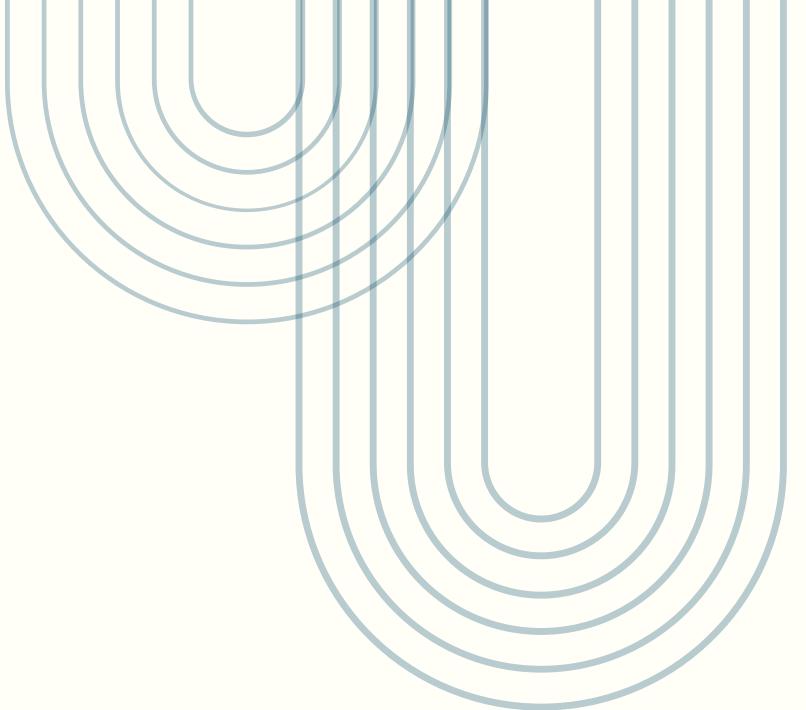


# LABORATORIO REDES

## DESARROLLO DE UNA API



**Luca Irrazabal**

Flask



**Santino Ponchiardi**

Entorno



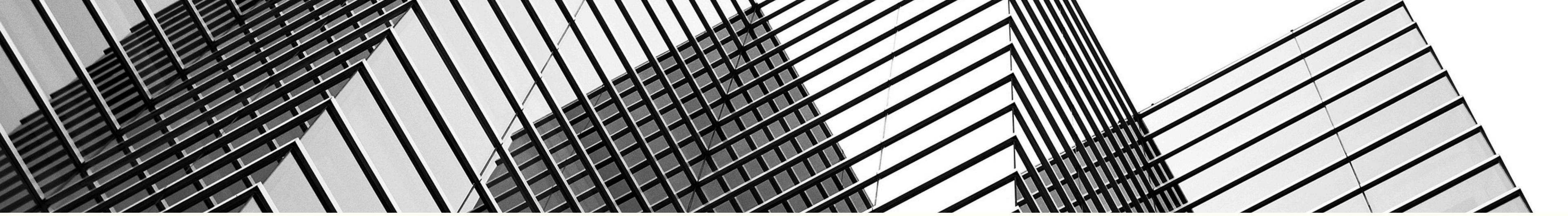
**Santos Farias**

API externa



**Brandon Michel**

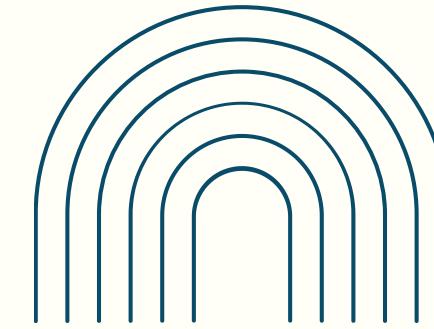
Test



# INTRODUCCION

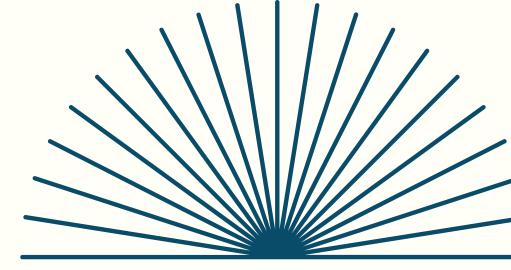
En este laboratorio **creamos una API** en el lenguaje de programacion *Python* y con el framework *Flask*. Esta dividido en *cuatro partes*.

1. La **configuracion del entorno virtual** de programacion en el que se desarrolla este laboratorio.
2. La **creacion e implementacion de una API** cinematografica con Flasks.
3. La **integracion de nuestra API** cinematografica **con una API externa** de feriados.
4. La **evaluacion** de la API.



# PERO QUE ES UNA API???

Una API REST o API RESTful es una Interfaz de Programacion de Aplicacion, esto permite que dos sistemas intercambien información a través de internet. que se ajusta a los principios de diseño del estilo arquitectónico de transferencia de estado representacional. Proporcionan una forma flexible y ligera de integrar aplicaciones y conectar componentes en arquitecturas de microservicios.



# CONFIGURACION DEL ENTORNO E INSTALACION DE LIBRERIAS

Las aplicaciones de Python usualmente usan paquetes y modulos que no son parte de la libreria estandar. Tambien estas aplicaciones normalmente fueron diseñadas para usarse con una version especifica de una libreria.

Lo cual nos lleva a un problema, que pasa si diferentes aplicaciones necesitan de diferentes versiones de una misma libreria?. Ya sea porque no se decidió actualizar una de las aplicaciones a la nueva versión de la librería o porque se aprovecha de una funcionalidad que en librerías nuevas ya no existe.

La solución es usar un entorno virtual, un directorio que contiene una instalación de Python de una versión en particular, además de unos paquetes adicionales. Que simula justamente un entorno artificial con todas las versiones específicas de librerías con las que fue pensada y hecha la aplicación.

# PASOS A SEGUIR

Fuente: [Python docs](#)

## 0. Requisito Previo

- Tener actualizado Python, para tener los modulos respectivos como venv y pip.
- Situarse en el directorio de la aplicacion,
- En el caso de estar trabajando en un repositorio con git para el control de versiones, recordar agregar un .gitignore con el directorio del entorno virtual.

## 1. Crearlo con venv

**python3 -m venv .nombre-venv**

Donde el modulo venv instalara la version de Python con la que fue llamada, en este caso Python 3. Y creara el directorio “.nombre-venv” si no existe, con una copia del interprete de Python y archivos de soporte. Un nombre estandar para el directorio es “.venv”.

## 2. Activar el Entorno

**source .nombre-venv/bin/activate**

Activar el entorno virtual cambiará el prompt de la consola para mostrar que entorno virtual está usando, y modificará el entorno para que al ejecutar python sea con esa versión e instalación en particular.

# PASOS A SEGUIR

Fuente: [Python docs](#)

## 2bis. Desactivar el Entorno

### `deactivate`

Este comando desactivara el entorno virtual para seguir usando la consola para otras aplicaciones o usos.

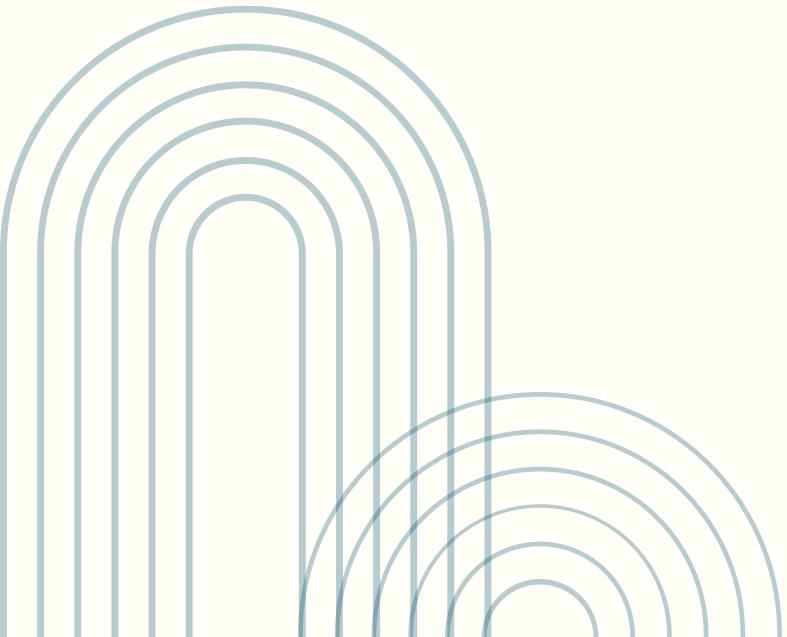
## 3. Instalar las librerias

### `python3 -m pip install -r requirements.txt`

Alternativamente ya estando con el entorno activado, hacer directamente:

### `pip install -r requirements.txt`

El programa pip (package installer for Python) instala, actualiza o elimina paquetes de Python. En nuestro caso como ya se “freezeo” un entorno con las diferentes librerias y sus versiones y nos las dieron en el archivo requirements.txt, solo necesitamos que pip lea el archivo. Pero tambien se podrian instalar manualmente una por una especificando la libreria y su version.





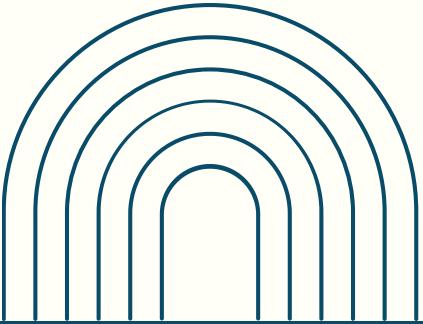
# FLASK

Flask fue el framework utilizado  
manejar el servicio básico del servidor:

- Realizar conexión con cliente
- Leer y responder solicitudes http



"ME GUSTA MUCHO  
FLASK."  
-OPINIONES DE  
USUARIOS RECIENTES



# Implementación

Dentro del proyecto, se importó de este framework, su clase "Flask".

Se utilizaron las funciones otorgadas por dicha clase para:

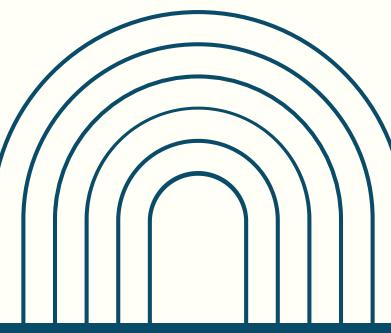
- Mapear urls a funciones del programa
- Leer datos entrantes en formato JSON
- Responder utilizando dicho formato.



# 4

# CONSUMO DE UNA API EXTERNA

- 
- Como modificar el codigo para agregar funcionalidades
  - Integracion de una API Externa
  - como utilizar las mismas para recomendar peliculas

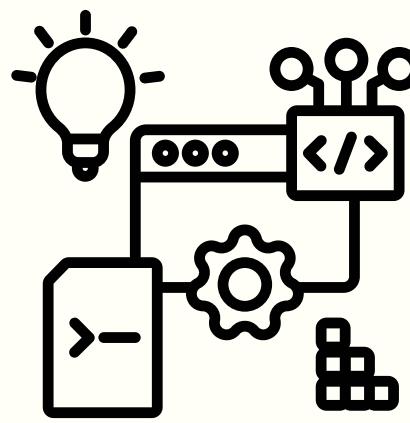


# Proximo Feriado

- Funciones Auxiliares:
  - get\_url
  - day\_of\_week
- buscar\_feriado(self, type)

- Clase NextHoliday
  - Atributos → 1.loading  
2.year  
3.holiday
  - Metodos
    - 1.\_\_init\_\_(self)
    - 2.set\_next(self, holidays)
    - 3.fetch\_holidays(self)
    - 4.render(self)

```
def buscar_feriado(self, type):  
    response = requests.get(get_url(self.year))  
    data = response.json()  
    now = date.today()  
    today = {  
        'day': now.day,  
        'month': now.month  
    }  
  
    data = next(  
        (h for h in data if h['mes'] == today['month'] and h['dia'] > today['day'] or h['mes'] > today['month'] and h['tipo'] == type),  
        data[0]  
    )  
  
    self.loading = False  
    self.holiday = data
```



# Integración en main.py

## 1. recomendar\_pelicula\_feriado

1. Crear una instancia de la clase `NextHoliday`
2. Obtener el proximo feriado
3. Seleccionar una película aleatoria
4. Devolver un JSON

```
def recomendar_pelicula_feriado(genero):
    """
        Returns a movie given a 'genero'. Also indicating next holiday's date.
    """
    # Crea instancia de la clase
    next_holiday = NextHoliday()
    # Llama la funcion que devuelve los feriados para guardarla en
    # next_holiday.holiday
    next_holiday.fetch_holidays()

    # Guarda el proximo feriado
    proximo_feriado = next_holiday.holiday
    day = proximo_feriado['dia']
    month = proximo_feriado['mes']
    year = next_holiday.year
    motivo = proximo_feriado['motivo']
    # Busca una pelicula aleatoria del genero pedido
    pelicula_encontrada = _pelicula_aleatoria_segun_genero(genero)

    return jsonify({'feriado': {'fecha': f'{day}/{month}/{year}',
                                'motivo': motivo},
                    'pelicula sugerida': pelicula_encontrada}), 200
```



# Integración en main.py

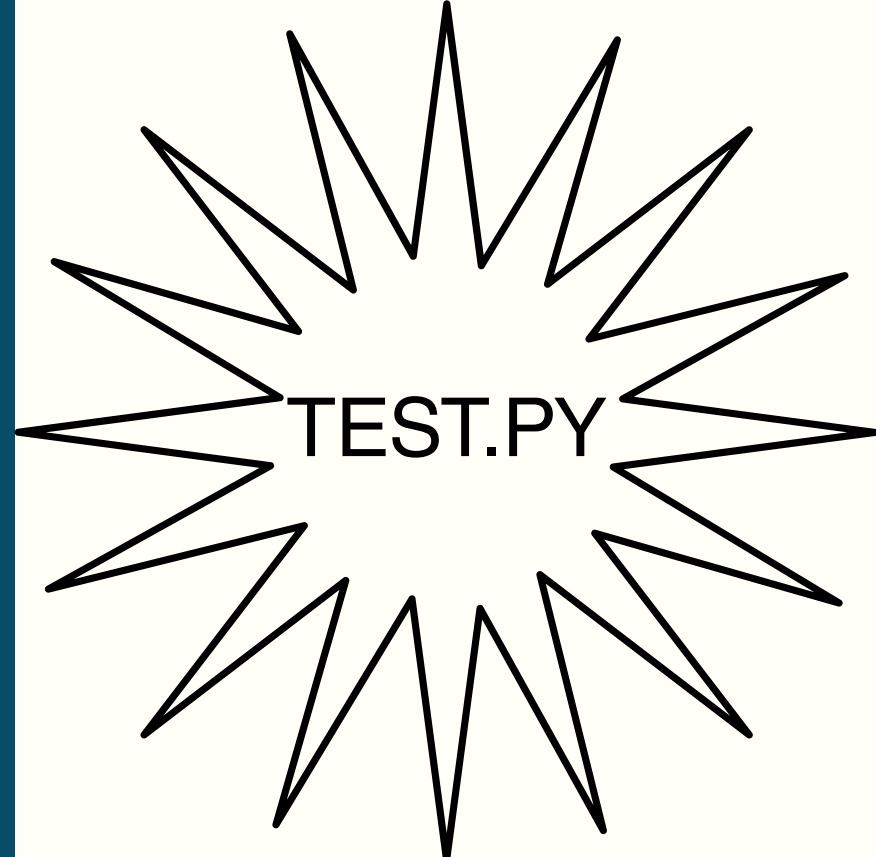
2. recomendar\_feriado\_por\_tipo

```
def recomendar_feriado_por_tipo(genero, tipo):
    next_holiday = NextHoliday()
    next_holiday.buscar_feriado(tipo)

    pelicula_encontrada = _pelicula_aleatoria_segun_genero(genero)

    proximo_feriado = next_holiday.holiday
    day = proximo_feriado['dia']
    month = proximo_feriado['mes']
    year = next_holiday.year
    tipo_feriado = proximo_feriado['tipo']

    return jsonify({'feriado': {'fecha': f'{day}/{month}/{year}',
                                'tipo': tipo_feriado},
                  'pelicula_sugerida': pelicula_encontrada}), 200
```



Lo hacemos con una  
comunicacion  
directa con nuestra  
API, devolviendo  
en pantalla las  
respuestas que se le  
dan al cliente

- test.py lo usamos para realizar pruebas de la API que hemos creado viendo como un cliente se puede comunicar con ella usando las funcionalidades que nosotros hemos creado

```
[master] [~/Documentos/Redes/redes25lab1g27]$ python3 test.py
Películas existentes:
ID: 1, Título: Indiana Jones, Género: Acción
ID: 2, Título: Star Wars, Género: Acción
ID: 3, Título: Interstellar, Género: Ciencia ficción
ID: 4, Título: Jurassic Park, Género: Aventura
ID: 5, Título: The Avengers, Género: Acción
ID: 6, Título: Back to the Future, Género: Ciencia ficción
ID: 7, Título: The Lord of the Rings, Género: Fantasía
ID: 8, Título: The Dark Knight, Género: Acción
ID: 9, Título: Inception, Género: Ciencia ficción
ID: 10, Título: The Shawshank Redemption, Género: Drama
ID: 11, Título: Pulp Fiction, Género: Crimen
ID: 12, Título: Fight Club, Género: Drama
ID: 13, Título: Cars 1, Género: Binario

Película agregada:
ID: 14, Título: Pelicula de prueba, Género: Acción

Detalles de la película:
ID: 1, Título: Indiana Jones, Género: Acción

Película actualizada:
ID: 1, Título: Nuevo título, Género: Comedia

Busqueda de películas que tengan 'the' en el título:
{'genero': 'Acción', 'id': 5, 'titulo': 'The Avengers'}
{'genero': 'Ciencia ficción', 'id': 6, 'titulo': 'Back to the Future'}
{'genero': 'Fantasía', 'id': 7, 'titulo': 'The Lord of the Rings'}
{'genero': 'Acción', 'id': 8, 'titulo': 'The Dark Knight'}
{'genero': 'Drama', 'id': 10, 'titulo': 'The Shawshank Redemption'}
Buscar usando espacios:

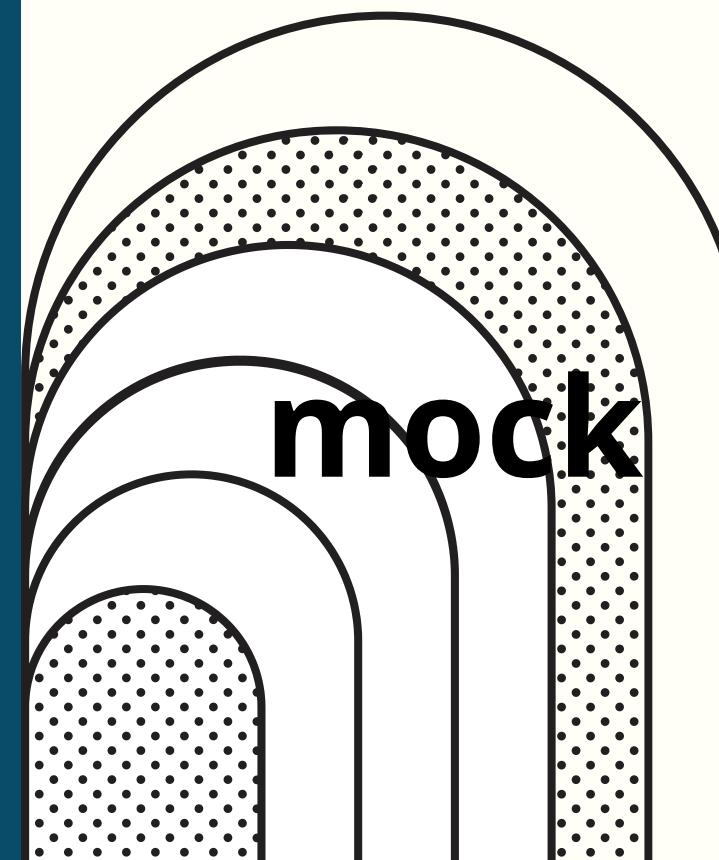
Película eliminada correctamente.
Verificar si película con ID = 1 fue reemplazada correctamente
Película con id 1 encontrada!

Película según género para ver en el próximo feriado:
Para feriado del 2/4/2025, con Motivo Día del Veterano y de los Caídos en la Guerra de Malvinas
Película encontrada según el género Binario
Ver película Cars 1
{'feriado': {'fecha': '1/1/2025', 'tipo': 'inamovible'}, 'pelicula_sugerida': {'genero': 'Binario', 'id': 13, 'titulo': 'Cars 1'}}

[master] [~/Documentos/Redes/redes25lab1g27]$ █
```

# **test\_pytest.py**

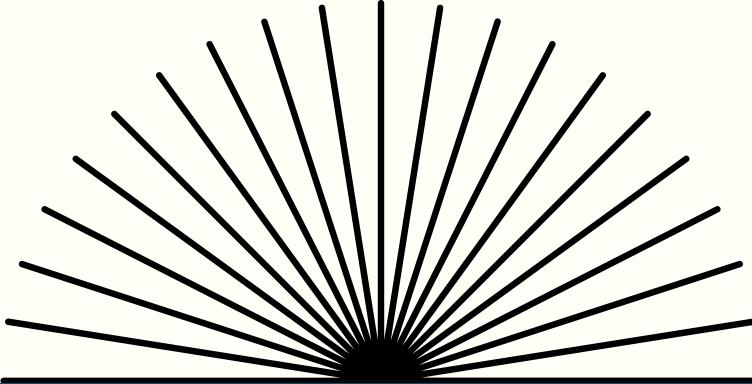
**pytest**



En este archivo lo que hacemos para ejecutar las pruebas es simular las respuestas de nuestra API, teniendo un entorno mas controlado y mas comodo para el uso y la prueba de nuestra API a pesar de no haber una verdadera comunicacion con ella, esto lo hacemos con el uso de pytest y mock.

es un framework de pruebas de python, automatiza la ejecucion de pruebas, es sencillo para su uso y te muestra de forma mas detallada los errores que hubo en las pruebas lo que facilita solucionarlos

en nuestro archivo usamos mock para simular las respuestas que podria llegar a dar nuestra API para cada endpoint, no hace falta que haya una comunicacion real con nuestra API



# Agregar pruebas en test\_pytest.py

Para agregar tests para las nuevas funcionalidades debemos agregar la respuesta

simulada que podria recibir de parte de nuestra API

luego definir el test

ejecutando el endpoint y comprobando que los resultados sean los deseados

por medio de asserts.

```
# Simulamos la respuesta para buscar una pelicula por el titulo
m.get('http://localhost:5000/peliculas/the', json=[{'id': 5, 'titulo': 'The Avengers', 'genero': 'Acción'}, {'id': 6, 'titulo': 'Back to the Future', 'genero': 'Ciencia ficción'}])
```

```
def test_buscar_pelicula(mock_response):
    titulo = "the"
    response = requests.get(f'http://localhost:5000/peliculas/{titulo}')
    assert response.status_code == 200
    assert len(response.json()) == 2
```

# POSTMAN

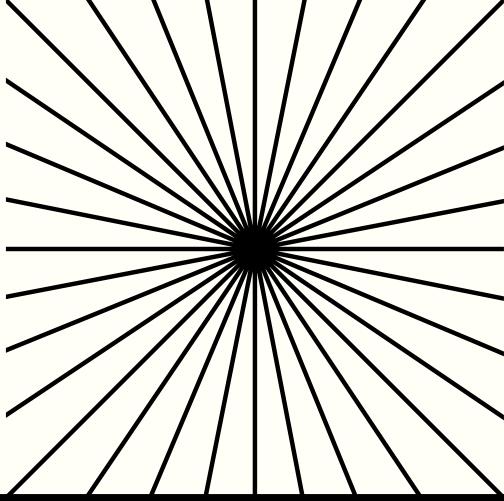
La prueba con postman es de las 3 pruebas lo que se podria considerar mas cercano a lo que ve el cliente, usamos los endpoint y metodos de nuestra api y te devuelve en pantalla los resultados lo que vuelve la prueba mas visual.

The screenshot shows the Postman application interface. On the left, there's a sidebar with a tree view under 'Lab1 Redes' containing various API endpoints. The main area shows a request for 'GET todas las peliculas' with the URL 'http://localhost:5000/peliculas'. The response status is '200 OK' with a response time of '44 ms' and a size of '923 B'. The response body is displayed as JSON, showing a list of movies:

```
1 [  
2 {  
3     "genero": "Acción",  
4     "id": 1,  
5     "titulo": "Indiana Jones"  
6 },  
7 {  
8     "genero": "Acción",  
9     "id": 2,  
10    "titulo": "Star Wars"  
11 },  
12 {  
13     "genero": "Ciencia ficción",  
14     "id": 3,  
15     "titulo": "Interstellar"  
16 },  
17 {  
18     "genero": "Aventura",  
19     "id": 4,  
20     "titulo": "Jurassic Park"  
21 }]
```

At the bottom, there are navigation links for 'Postbot', 'Runner', 'Capture requests', 'Cookies', 'Vault', and 'Trash'.

# Diferencias en los tests



## **test.py**

permite comunicacion directa con la API, las pruebas se hacen a mano y se presentan en pantalla las respuestas tambien a mano.

## **test\_pytest.py**

un entorno mucho mas controlado sin la necesidad de estar comunicandose con la API, pruebas mucho mas especificas y gracias al pytest mucho mas rapido, amigable a la vista mostrando los errores y facilitando su resolucion.

## **postman**

postman es la prueba mas cercana al cliente de la API, las otras 2 pruebas son mas convenientes ya que podemos probar situaciones mas especificas que pueden surgir y ver como reacciona nuestra api a ellas.



# Contact Us

[www.reallygreatsite.com](http://www.reallygreatsite.com)

hello@mi.unc.edu.ar

Phone number: 351 681-7101

---

Follow us:

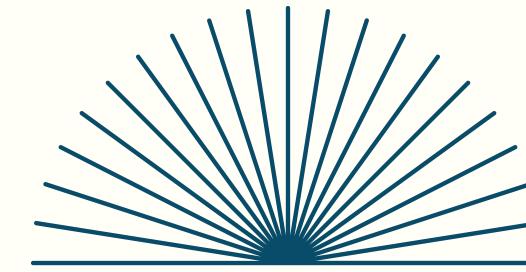
@famaf.posting



—¿Cómo se despide una API en invierno?  
—"¡Nos freeze-amos hasta la próxima thaw request!"

"¡Adiós! Si no me encuentras, revisa el cache... o espera mi reload."

—¿Por qué la API canceló su fiesta de despedida?  
\*—Porque solo recibió "DELETE /party"... y nadie vino.



"Por qué PUT terminó con su novia?"  
-Porque en lugar de arreglar los problemas, solo las reemplazaba

## -¿Por qué la API se despidió con un status code 200?

**-"Porque todo salió OKey,  
¡hasta la próxima request!"**

"Me voy, pero no te preocupes... ¡soy stateless! Volveré como si nada hubiera pasado."

"¿Por qué la API se puso a llorar?  
Porque recibió un 400 Bad Request... y era su cumpleaños." 🎂😢