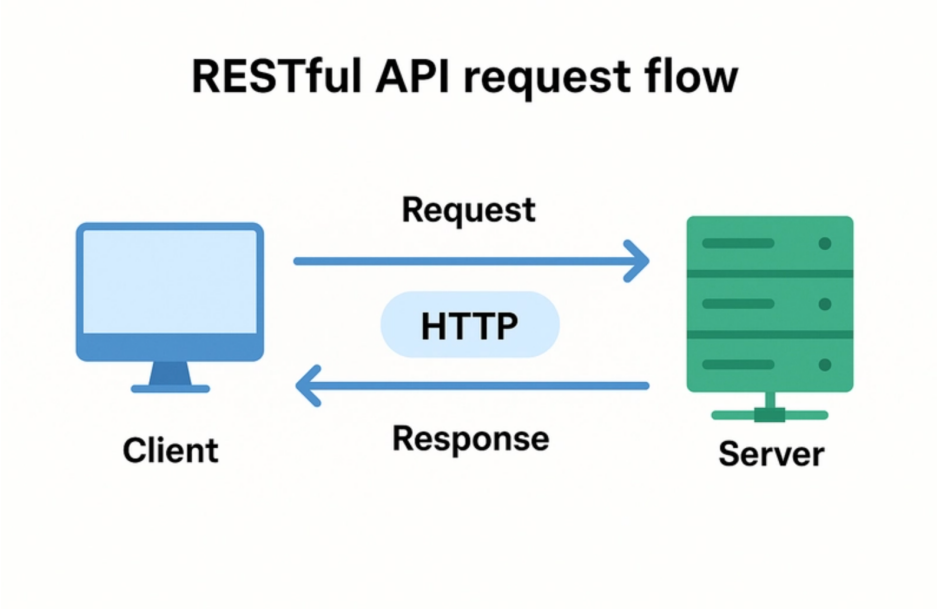
API REST con

1. Que es una API Rest

Una API Rest podemos decir que es una interfaz para que dos sistemas se comuniquen usando peticiones HTTP simples como GET, POST, PUT o DELETE.

Con este gráfico quizá se entienda mejor.



En esta práctica, lo que vas a desarrollar va a ser el Server. Crearás una interfaz con cierta información accesible para el resto de la clase.

2. Que es Fast API

FastAPI es un framework de Python para crear APIs web rápidas. Puede ser que hayáis escuchado otros frameworks como Flask o Django. Os voy a dar unos losm tips necesarios para escoger bien el framework dependiendo de la ocasión.

FastAPI destaca cuando quieres rendimiento, tipado fuerte y APIs puras sin tanta estructura pesada. Además, genera docs automáticas (Swagger), y es asíncrono desde el inicio, lo que lo vuelve muy rápido.

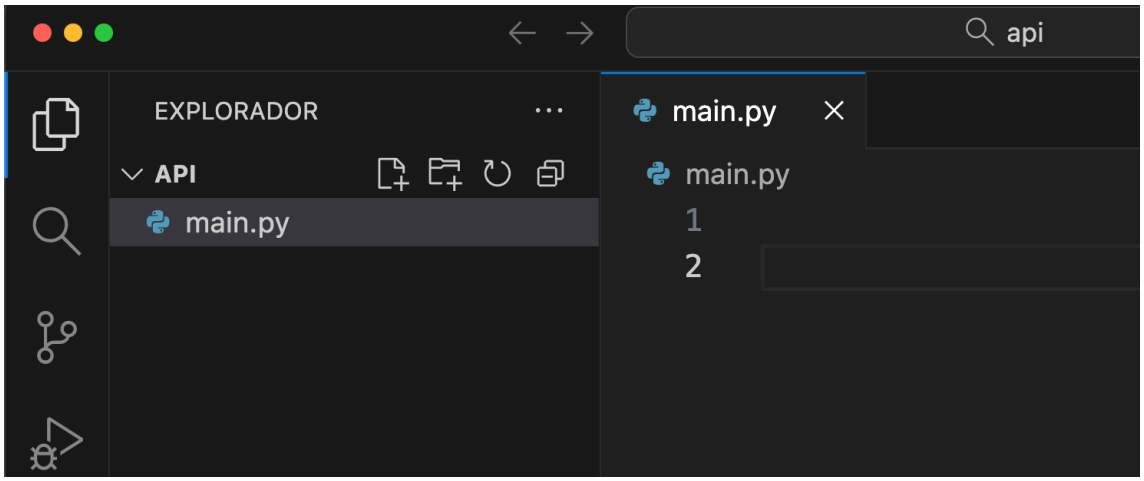
Django es un “todo en uno”: ORM, plantillas, autenticación... perfecto para webs completas, pero sobrado para solo APIs. Si solo quieres levantar una pequeña API es matar moscas a cañonazos. Podemos decir que es el simil a Spring en Java o .NET a C#. Vamos, es un framework FullStack.

Flask es muy flexible, pero minimalista. Terminas armando muchas cosas a mano. De las 3 opciones es la que menos me gusta, porque no es tan rápida como FastAPI y ya cuando el proyecto es más grande, es mejor tirar por Django.

En proyectos donde lo importante es una API limpia, rápida y bien tipada, FastAPI suele ser la opción más cómoda.

3. Arrancamos el proyecto

Crea la carpeta del proyecto y ábrela con VSCode (puedes utilizar PyCharm también)
Crea un archivo que se llame **main.py**



Testea el python usando un pequeño print("Hola Mundo")

4. Creando nuestro primer endPoint

Un endpoint es una URL específica de una API donde un cliente puede realizar una petición concreta. Es el "punto de entrada" o una "puerta" para acceder a un recurso determinado. Una API Rest puede estar formada por uno o decenas de endpoints.

Primero vamos a importar fastAPI:

```
from fastapi import FastAPI
```

Si no resuelve el módulo, quiere decir que no tenemos instalada la librería. Abre una terminal y ejecuta:

pip install fastapi uvicorn

Ahora vamos a escribir la línea donde se especifica que nuestra aplicación va a ser una aplicación FastAPI:

```
app = FastAPI()
```

Si te fijas en esa línea, lo que hace es crea un Objecto FastAPI() y lo añade a la variable app.

Ahora vamos a crear nuestro primer endPoint. Será el endPoint de saludo o de inicio o raíz como quieras llamarlo. Cada vez que llamen a tu API REST sin especificar la URL les aparecerá este saludo:

```
@app.get("/")
def saludo():
    return "Hola mundo"
```

Debería de quedarte un archivo así:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def saludo():
    return "Hola mundo"
```

5 líneas de código (que maravilla) ☺

Ahora solo nos falta arrancarlo. Para arrancarlo vamos a tener que llamar al servidor web “uvicorn”. La linea es la siguiente:

uvicorn main:app --host 0.0.0.0 --reload

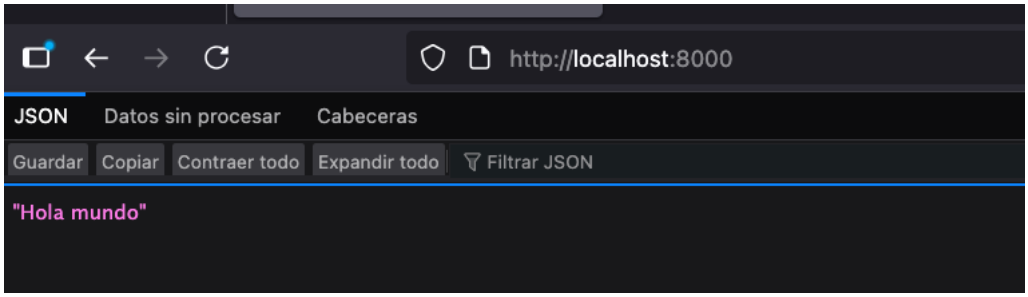
Vamos a explicar la línea.

- Uvicorn es el servidor web con el que trabaja FastAPI (aunque no es igual podemos decir que es como un Apache o nginx)
- main:app. Le está diciendo a uvicorn que busque en un archivo llamado main la variable app. (Donde tenemos nuestra clase fastAPI)
- Por último, la opción --reload hace que cuando hagas cambios en el proyecto vuelva a relanzar la API REST cargando los nuevos cambios (Como SpringBoot)

Al ejecutar te habrá aparecido algo así.

```
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [6951] using StatReload
INFO:     Started server process [6953]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Por defecto, fastAPI arranca en el puerto 8000. Así que uvicorn ha levantando un servidor web en tu localhost:8000. Si accedes a un navegador y vas a esa URL, deberías de ver algo como:



Ahora, el resto de la clase, si accede a tu IP:8000 podrá ver tu saludo.

---

1. Creando un EndPoint método POST

Si te has fijado, hemos usado un método get para ese saludo inicial. A continuación, te explico los métodos HTTP más utilizados y cuál es su utilidad o su estándar en la industria.

- **GET**: El cliente le pide datos al servidor.
- **POST**: El cliente quiere enviarle datos al servidor.
- **PUT**: El cliente le envía datos al servidor para que haga algún cambio.
- **DELETE**: El cliente le envía datos al servidor para borrar algo.

Típica pregunta ¿Se puede pedir datos al servidor con métodos PUT POST o DELETE?

Técnicamente si se puede, pero es muy mala práctica.

Sobre el clásico “yo he visto páginas PHP que mandan parámetros por GET”, la explicación es simple: en la web tradicional el navegador solo usaba GET y POST, y GET era muy cómodo para enviar pequeños parámetros en la URL. Eso no hace que esté bien usar GET para borrar cosas o cambiar datos; solo refleja que PHP clásico venía del mundo de las páginas y no de las APIs bien diseñadas.

Conclusión:

Cada método HTTP es una intención, no una obligación técnica. Los navegadores y los frameworks pueden enviar lo que quieras por donde quieras, pero la industria espera una serie de buenas prácticas.

Una vez explicado esto, vamos a crear nuestro primer endpoint con el método POST.

```
@app.post("/message")
async def get_message(mensaje: str = Body(
     ..., media_type="text/plain")):
    with open("frases.txt", "a", encoding="utf-8") as f:
            f.write(mensaje + "\n")

    return "Frase guardada correctamente"
```

Que hace esto:

- Espera una petición HTTP con método POST
- Espera un parámetro tipo texto plano en el body de la petición http. (Normalmente se utiliza JSON pero no quiero meterme ahora con eso. Lo veremos en siguientes prácticas)
- La frase que nos envían al endPoint la recogemos en la variable mensaje.
- Abrimos en archivo “frases.txt” con el parámetro “a” de append y se escribirá la frase recibida en ese archivo.
- Por último, contestamos a nuestro cliente con un “Frase correctamente guardada”

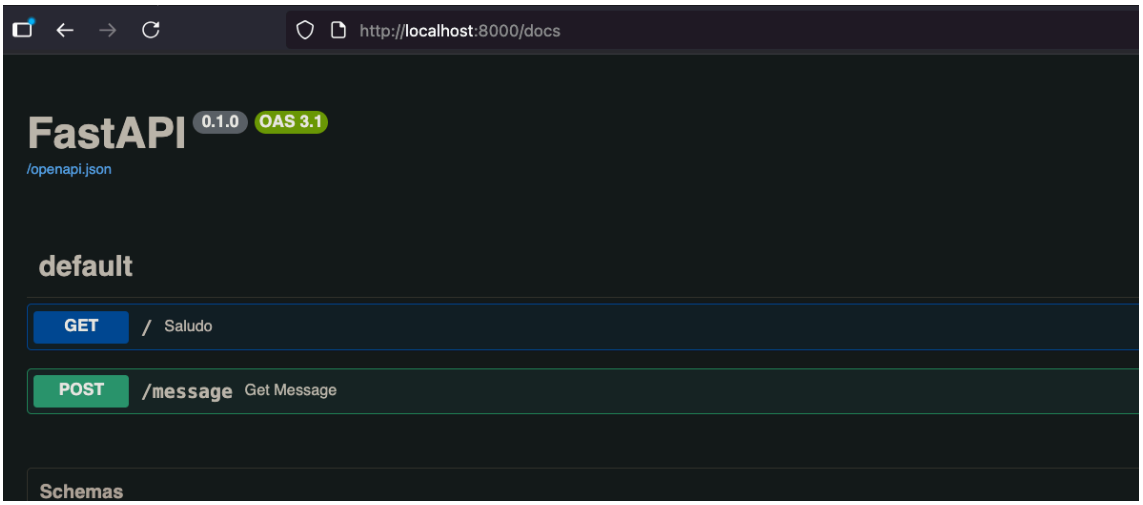
Pues vamos a probarlo.
Entra en el navegador… Uy espera…
.
.
.
.
esto es un método POST. Me da que no podemos testarlo desde el navegador de manera convencional. **El navegador solo envía peticiones HTTP GET**. ¿Y entonces? ¿Qué hacemos?

Tenemos varias alternativas. La primera y la más utilizada en entornos de producción es POSTMAN. ¡Si quieres puedes utilizarla, pero en este caso tenemos otra alternativa más rápida!

FASTAPI LEVANTA POR DEFECTO UN SWAGGER EN /docs y para testear en desarrollo es genial.

Accede mediante URL a *localhost:8000/docs*

Deberías de ver algo como:



Fíjate. Fast API te brinda un Swagger e incluso te separa por colores los métodos get, post, put, delete.... Lo mejor de todo es que puedes probar los métodos desde ahí.
Pero, antes de nada, ¿Qué es un swagger?

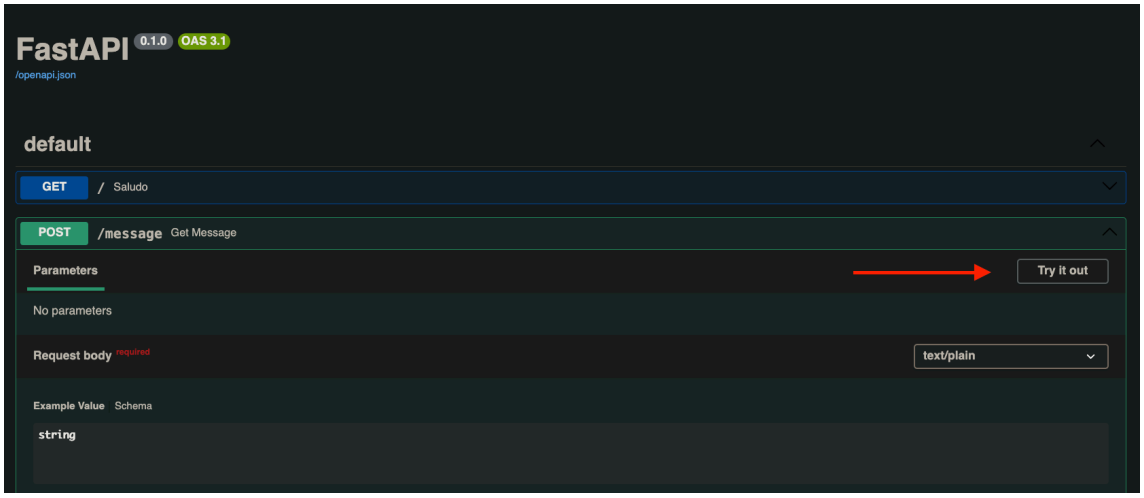
Un swagger es una herramienta para documentar y probar APIs. Te permite:

- Ver todos los endpoints disponibles en tu API.

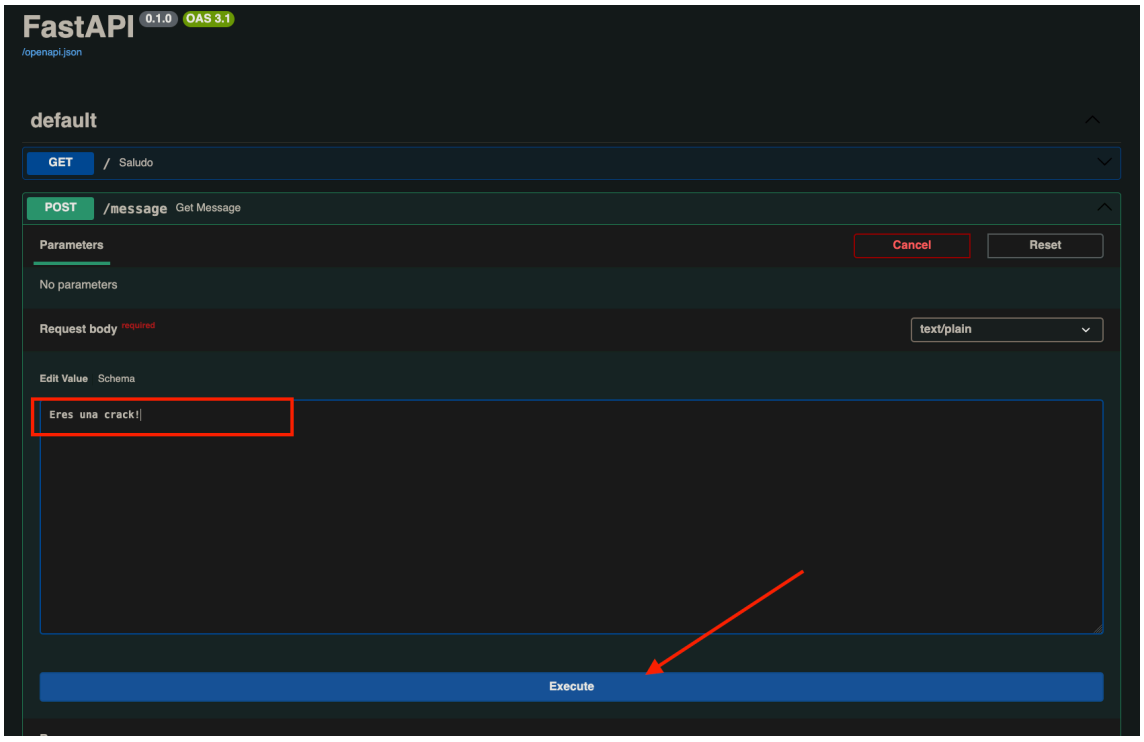
- Saber qué parámetros esperan y qué devuelve cada endpoint.
- Probar los endpoints directamente desde el navegador, enviando GET, POST, etc., sin necesidad de Postman u otras herramientas externas.

Esto es super útil cuando tienes que consultar una API externa. Puedes ver que contrato tienen (que parámetros necesita y que devuelve)

Pues vamos a probar la petición POST. Haz click en “Try it Out”



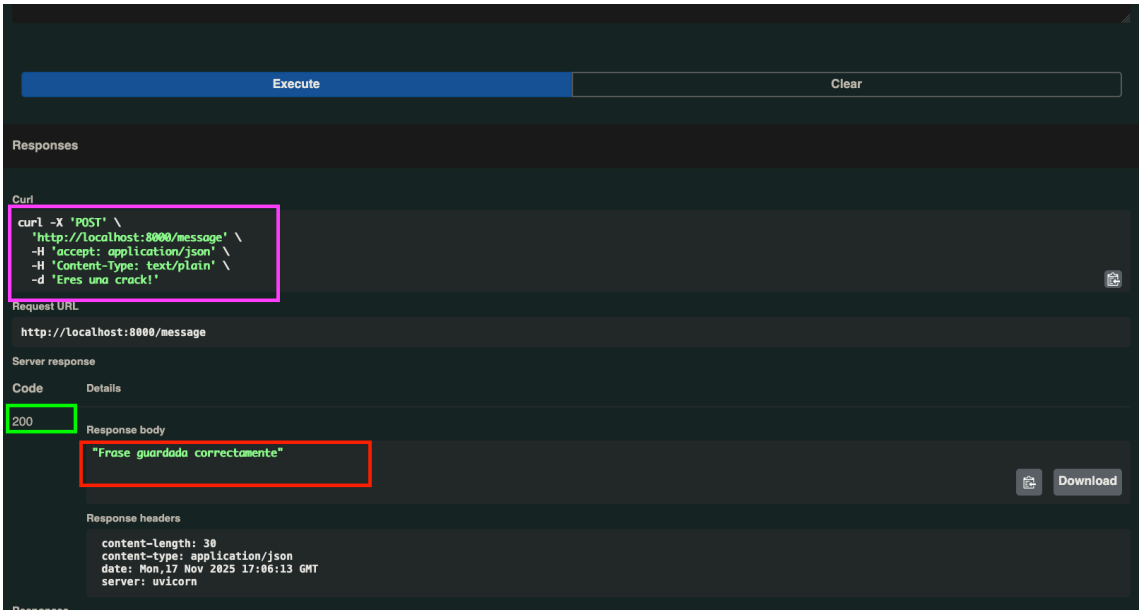
Escribe el mensaje que quieras enviar y dale a “Execute”:



Puedes ver el response de tu propio servidor:

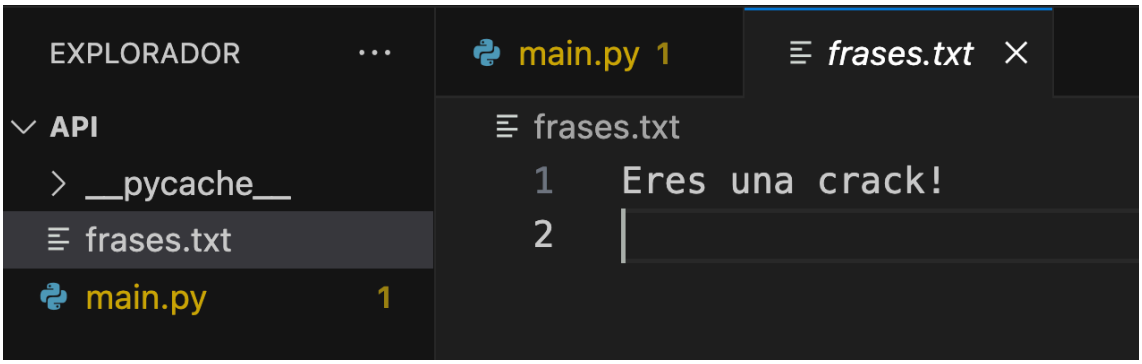
- En el recuadro lila se puede ver la petición que se ha hecho a tu server con la herramienta “curl”

- En el recuadro verde se puede ver el código que ha devuelto. Un 200 este que todo ha salido bien. ☺
- En el recuadro rojo, se puede ver lo que ha respondido tu servidor.



Vete a tu proyecto en VSCode o PyCharm y mira a ver que ha pasado.

Se debería de haber creado un archivo “frases.txt” con el mensaje:



Y además mirar el log:

Esto si os fijáis, podemos identificar la IP origen, el puerto, el EndPoint que nos ha consultado y la respuesta que le hemos dado de StatusCode (200 OK).

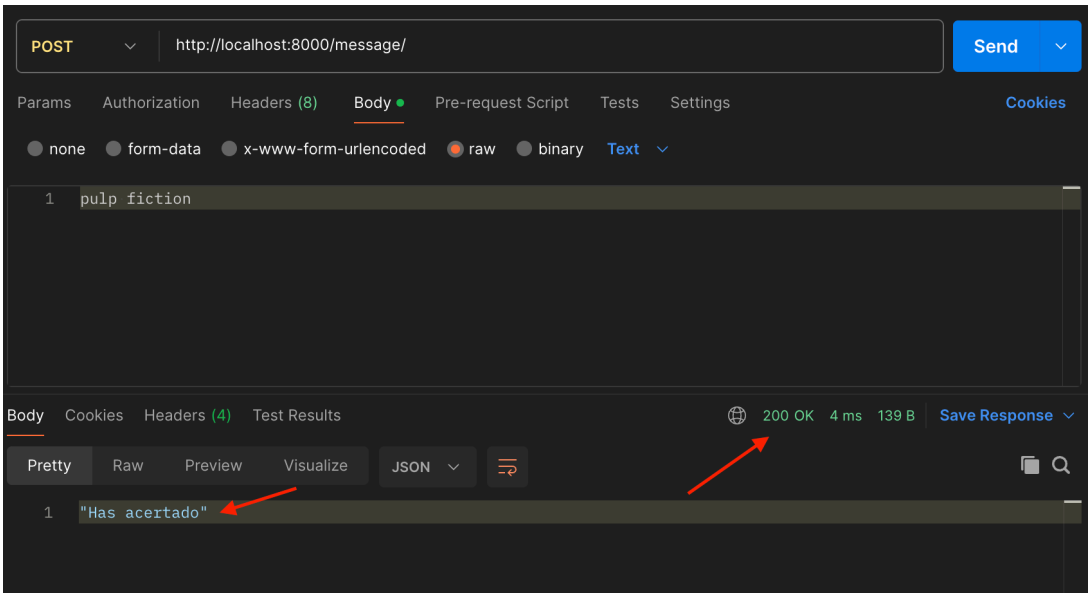
1. Propuesta de ejercicio

Ahora vamos a hacer un pequeño juego. Voy a proyectar las IPs de todos vosotros para que cada uno pueda dejarle mensajes al compañero que quiera. La idea es que la siguiente:

Cada uno de vosotros escogerá una de sus películas favoritas y publicará en el endPoint raiz "/" una pequeña sinopsis o algo muy reseñable para poder adivinarla. El resto de la clase intentarán adivinarla. En caso de adivinarla, deberán de poder dejar una opinión sobre esa película.

Todo esto lo haremos con la herramienta Postman
Descargaos postman.
Cuando lo tengáis descargado, instalarlo y darle a skip a lo de crear cuenta.

Cuando estéis en el cuadro principal, probadlo con vosotros mismos. Deberéis poner el método POST, la URL y dentro de Body darle a "raw" para enviar los datos en crudo. Cuando le des a "send" deberías de ver el statusCode 200 y el response. También debería de añadirse a tu frases.txt. Compruébalo.



¿Qué tienes que hacer?

- Modifica el endPoint "/" para que diga la pista.
    - Puedes ser muy creativo aquí. La pista puede ser una frase, una sinopsis, pero también una imagen de la película, GIF o HTML. Tú decides. Te dejo el código para devolver una imagen.

```python
@app.get("/")
def image():
    file = "miImagen.png"
    return FileResponse(file, media_type="image/png")
```

- Modifica tu “/message” por “/respuesta” para que puedan responder. Cambia también frases.txt por respuestas.txt. Además, este endpoint debe responder con “Has acertado” o “Has fallado”. Evidentemente deberás de crear alguna cosa para comprobarlo.
- Crea un endPoint “/opinion” para que puedan valorar tu peli. Guarda las repuestas en un fichero llamado opiniones.txt
- Crea un endPoint “/opiniones” para poder visualizar todas las opiniones de tu película.

Extra:

- Crea el endPoint “/estadisticas” para dar esa estadística. Lo mismo, deberías de crear algunas variables para poder dar este dato.

- **¡Por último, pástelo bien adivinando las películas favoritas de tus compis!**