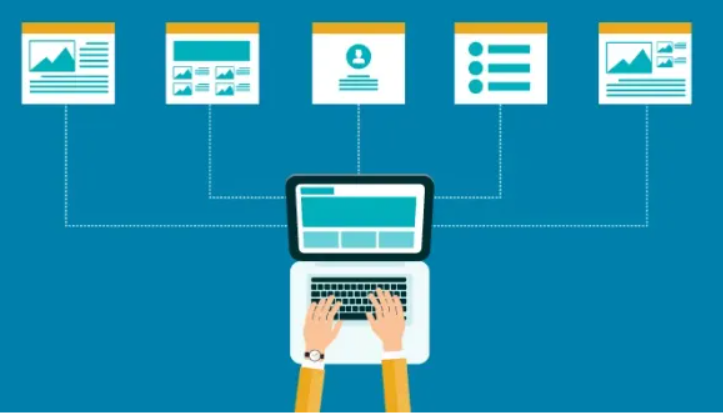
**FASTAPI – PYTHON**

# Estudio de Python – FastAPI

<https://fastapi.tiangolo.com/es/>

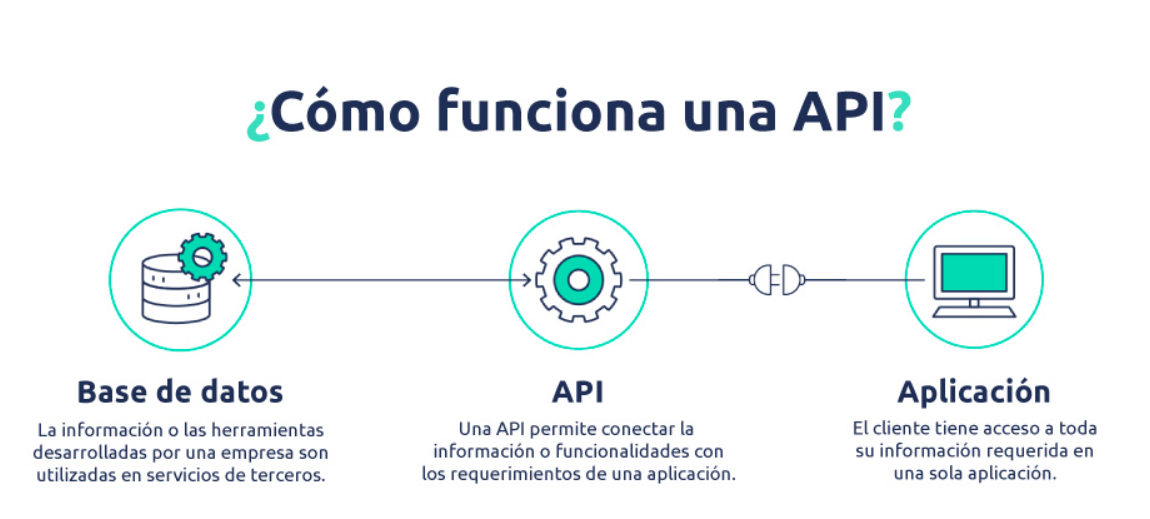
<https://fastapi.tiangolo.com/es/tutorial/>

**Framework**



Un framework es un conjunto de reglas y convenciones que se usan para desarrollar software de manera más eficiente y rápida. Estos marcos de trabajo se emplean para ahorrar tiempo y esfuerzo en el desarrollo de aplicaciones, ya que proporcionan una estructura básica que se puede utilizar como punto de partida.

**API: application programming interface (interfaz de programación de aplicaciones)**



Capa de comunicación que utiliza ciertos mecanismos (protocolos y estándares) que permiten hablar con el Backend.

**FastAPI**



FastAPI es un web framework moderno y rápido (de alto rendimiento) para construir APIs con Python 3.8+ basado en las anotaciones de tipos estándar de Python.

* **Rapidez**: Alto rendimiento, a la par con **NodeJS** y **Go** (gracias a Starlette y Pydantic). [Uno de los frameworks de Python más rápidos](https://fastapi.tiangolo.com/es/#rendimiento).
* **Rápido de programar**: Incrementa la velocidad de desarrollo entre 200% y 300%. \*
* **Menos errores**: Reduce los errores humanos (de programador) aproximadamente un 40%. \*
* **Intuitivo**: Gran soporte en los editores con auto completado en todas partes. Gasta menos tiempo debugging.
* **Fácil**: Está diseñado para ser fácil de usar y aprender. Gastando menos tiempo leyendo documentación.
* **Corto**: Minimiza la duplicación de código. Múltiples funcionalidades con cada declaración de parámetros. Menos errores.
* **Robusto**: Crea código listo para producción con documentación automática interactiva.
* **Basado en estándares**: Basado y totalmente compatible con los estándares abiertos para APIs: [OpenAPI](https://github.com/OAI/OpenAPI-Specification" \t "_blank) (conocido previamente como Swagger) y [JSON Schema](https://json-schema.org/).

## **Características de FastAPI**

**FastAPI** te provee lo siguiente:

### Basado en estándares abiertos

* [**OpenAPI**](https://github.com/OAI/OpenAPI-Specification) para la creación de APIs, incluyendo declaraciones de path operations, parámetros, body requests, seguridad, etc.
* Documentación automática del modelo de datos con [**JSON Schema**](https://json-schema.org/) (dado que OpenAPI mismo está basado en JSON Schema).

<https://json-schema.org/>

* Diseñado alrededor de estos estándares después de un estudio meticuloso. En vez de ser una capa añadida a último momento.
* Esto también permite la **generación automática de código de cliente** para muchos lenguajes.

### Validación

* Validación para la mayoría (¿o todos?) los **tipos de datos** de Python incluyendo:
  + Objetos JSON (dict).
  + JSON array (list) definiendo tipos de ítem.
  + Campos de texto (str) definiendo longitudes mínimas y máximas.
  + Números (int, float) con valores mínimos y máximos, etc.
* Validación para tipos más exóticos como:
  + URL.
  + Email.
  + UUID.
  + ...y otros.

Toda la validación es manejada por **Pydantic**, que es robusto y sólidamente establecido.

### Seguridad y autenticación

La seguridad y la autenticación están integradas. Sin ningún compromiso con bases de datos ni modelos de datos.

Todos los schemes de seguridad están definidos en OpenAPI incluyendo:

* HTTP Basic.
* **OAuth2** (también con **JWT tokens**). Prueba el tutorial en [OAuth2 with JWT](https://fastapi.tiangolo.com/es/tutorial/security/oauth2-jwt/).

<https://fastapi.tiangolo.com/es/tutorial/security/oauth2-jwt/>

* API keys en:
  + Headers.
  + Parámetros de Query.
  + Cookies, etc.

Más todas las características de seguridad de Starlette (incluyendo **session cookies**).

Todo ha sido construido como herramientas y componentes reutilizables que son fácilmente integrados con tus sistemas, almacenamiento de datos, bases de datos relacionales y no relacionales, etc.

### Dependency Injection

FastAPI incluye un sistema de **Dependency Injection** extremadamente poderoso y fácil de usar.

* Inclusive las dependencias pueden tener dependencias creando una jerarquía o un **"grafo" de dependencias**.
* Todas son **manejadas automáticamente** por el framework.
* Todas las dependencias pueden requerir datos de los requests y aumentar las restricciones del path operation y la documentación automática.
* **Validación automática** inclusive para parámetros del path operation definidos en las dependencias.
* Soporte para sistemas complejos de autenticación de usuarios, **conexiones con bases de datos**, etc.
* **Sin comprometerse** con bases de datos, frontends, etc. Pero permitiendo integración fácil con todos ellos.

### "Plug-ins" ilimitados

O dicho de otra manera, no hay necesidad para "plug-ins". Importa y usa el código que necesites.

Cualquier integración está diseñada para que sea tan sencilla de usar (con dependencias) que puedas crear un "plug-in" para tu aplicación en dos líneas de código usando la misma estructura y sintáxis que usaste para tus path operations.

### Probado

* Cobertura de pruebas al 100%.
* Base de código 100% anotada con tipos.
* Usado en aplicaciones en producción.

## **Características de Starlette**

Starlette es un marco/kit de herramientas [ASGI](https://asgi.readthedocs.io/en/latest/) liviano , ideal para crear servicios web asíncronos en Python.

<https://www.starlette.io/>

**FastAPI** está basado y es completamente compatible con **[Starlette](https://www.starlette.io/" \t "_blank)**. Tanto así, que cualquier código de Starlette que tengas también funcionará.

FastAPI es realmente una sub-clase de Starlette. Así que, si ya conoces o usas Starlette, muchas de las características funcionarán de la misma manera.

Con **FastAPI** obtienes todas las características de **Starlette** (porque FastAPI es simplemente Starlette en esteroides):

* Desempeño realmente impresionante. Es uno [de los frameworks de Python más rápidos, a la par con **NodeJS** y **Go**](https://github.com/encode/starlette#performance).
* Soporte para **WebSocket**.
* Soporte para **GraphQL**.
* Tareas en background.
* Eventos de startup y shutdown.
* Cliente de pruebas construido con HTTPX.
* **CORS**, GZip, Static Files, Streaming responses.
* Soporte para **Session and Cookie**.
* Cobertura de pruebas al 100%.
* Base de código 100% anotada con tipos.

## **Características de Pydantic**

Pydantic es la biblioteca de validación de datos más utilizada para Python.

<https://docs.pydantic.dev/latest/>

**FastAPI** está basado y es completamente compatible con **[Pydantic](https://pydantic-docs.helpmanual.io/" \t "_blank)**. Tanto así, que cualquier código de Pydantic que tengas también funcionará.

Esto incluye a librerías externas basadas en Pydantic como ORMs y ODMs para bases de datos.

Esto también significa que en muchos casos puedes pasar el mismo objeto que obtuviste de un request **directamente a la base de datos**, dado que todo es validado automáticamente.

Lo mismo aplica para el sentido contrario. En muchos casos puedes pasarle el objeto que obtienes de la base de datos **directamente al cliente**.

Con **FastAPI** obtienes todas las características de **Pydantic** (dado que FastAPI está basado en Pydantic para todo el manejo de datos):

* **Sin dificultades para entender**:
  + No necesitas aprender un nuevo micro-lenguaje de definición de schemas.
  + Si sabes tipos de Python, sabes cómo usar Pydantic.
* Interactúa bien con tu **IDE/linter/cerebro**:
  + Porque las estructuras de datos de Pydantic son solo instances de clases que tu defines, el auto-completado, el linting, mypy y tu intuición deberían funcionar bien con tus datos validados.
* **Rápido**:
  + En [benchmarks](https://pydantic-docs.helpmanual.io/benchmarks/" \t "_blank) Pydantic es más rápido que todas las otras libraries probadas.
* Valida **estructuras complejas**:
  + Usa modelos jerárquicos de modelos de Pydantic, typing de Python, List y Dict, etc.
  + Los validadores también permiten que se definan fácil y claramente schemas complejos de datos. Estos son chequeados y documentados como JSON Schema.
  + Puedes tener objetos de **JSON profundamente anidados** y que todos sean validados y anotados.
* **Extensible**:
  + Pydantic permite que se definan tipos de datos a la medida o puedes extender la validación con métodos en un modelo decorado con el decorador de validación.
* Cobertura de pruebas al 100%.

## **Instalar FastAPI**

<https://fastapi.tiangolo.com/es/tutorial/?h=instalar#instala-fastapi>

pip install "fastapi[all]"

## **Primeros Pasos**

* Crear archivo main.py

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

async def root():

  return {"message": "Hello FastAPI!"}

@app.get("/url")

async def url():

    return {"url": "http://test.com"}

* Lanzar Servidor:

uvicorn main:app --reload

uvicorn Componente instalado que permite lanzar servidres

main => nombre de mi .py. Em mi caso main.py

app => Lanzar mi contexto app

--reload => Refresca automáticamente al haber un cambio en el código

Como me da error para arrancar el servidor:

Usar el Comando Correcto de Python: En lugar de utilizar el comando

python --version,

utiliza python -m uvicorn directamente en tu terminal.

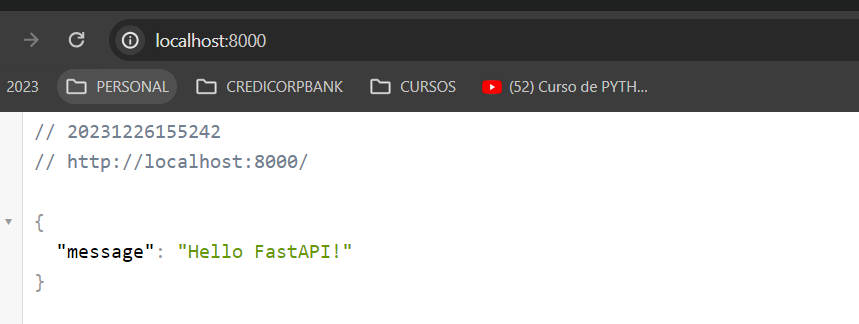
Intenta el siguiente comando:

python -m uvicorn main:app –reload

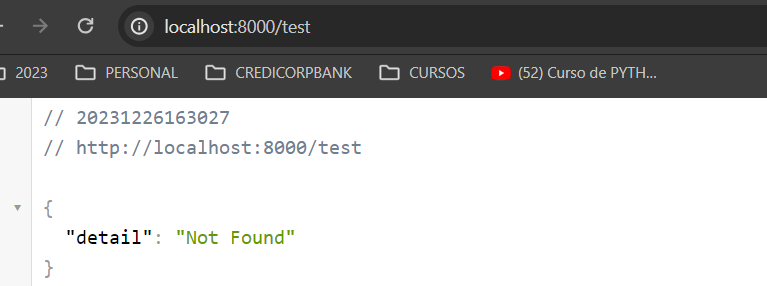
Esto asegurará que el comando uvicorn se ejecute utilizando la versión correcta de Python.

Este si inicio el servidor.

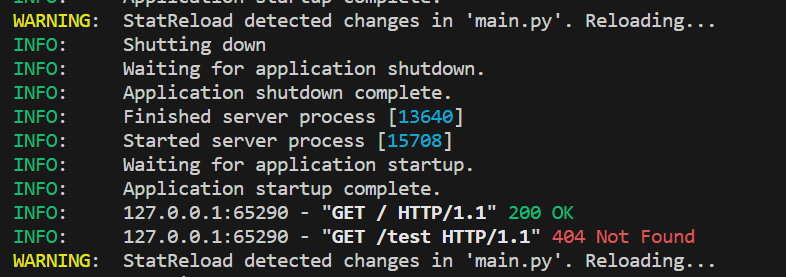
* Probar el llamado get



* Si llamo a una get no existente

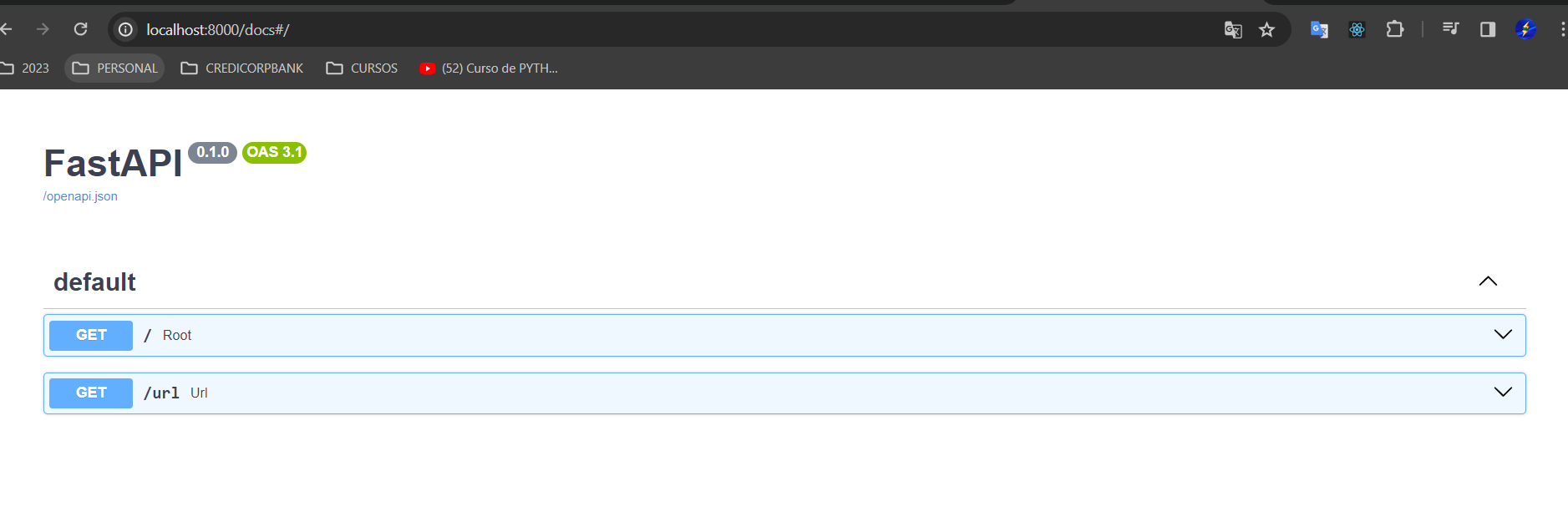


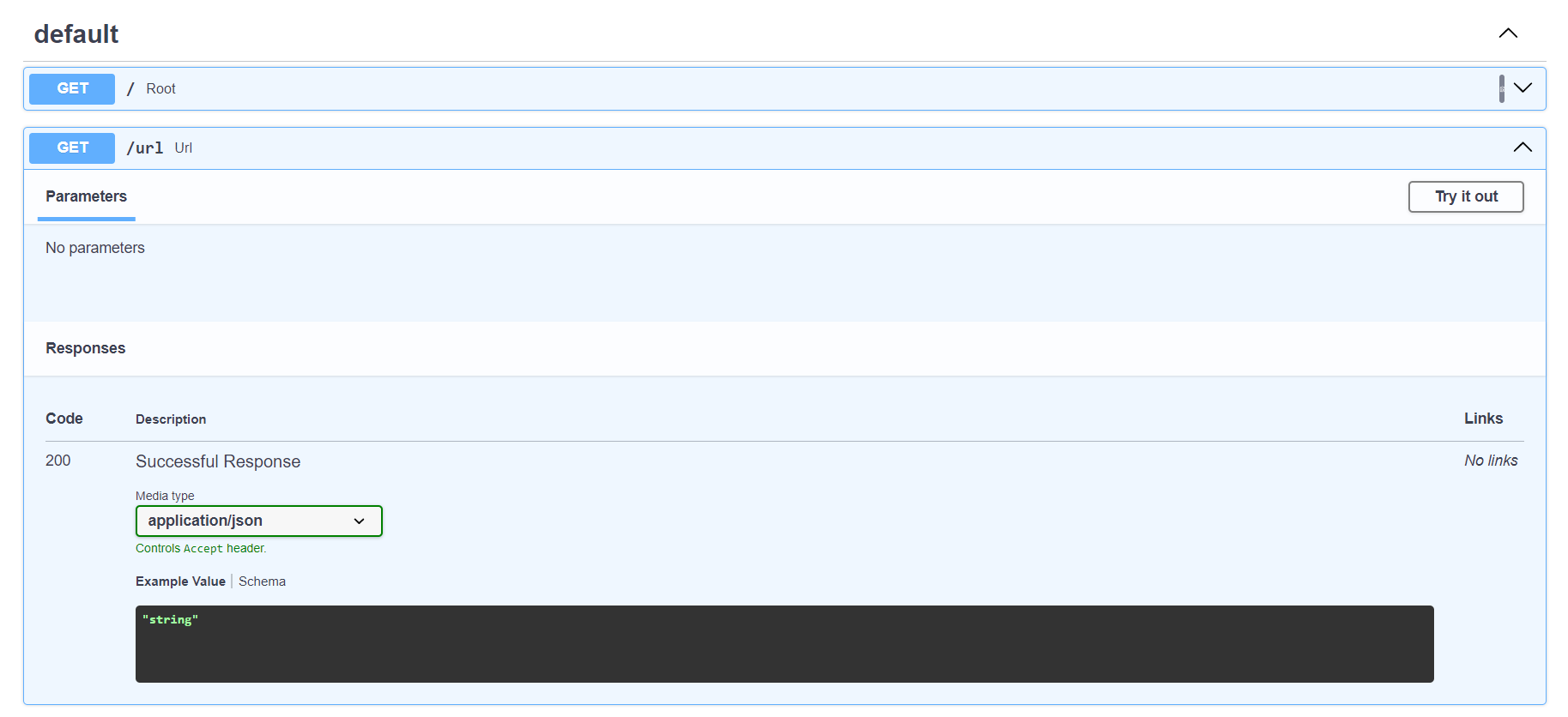
* En el servidor puedo ver la traza d elas llamadas



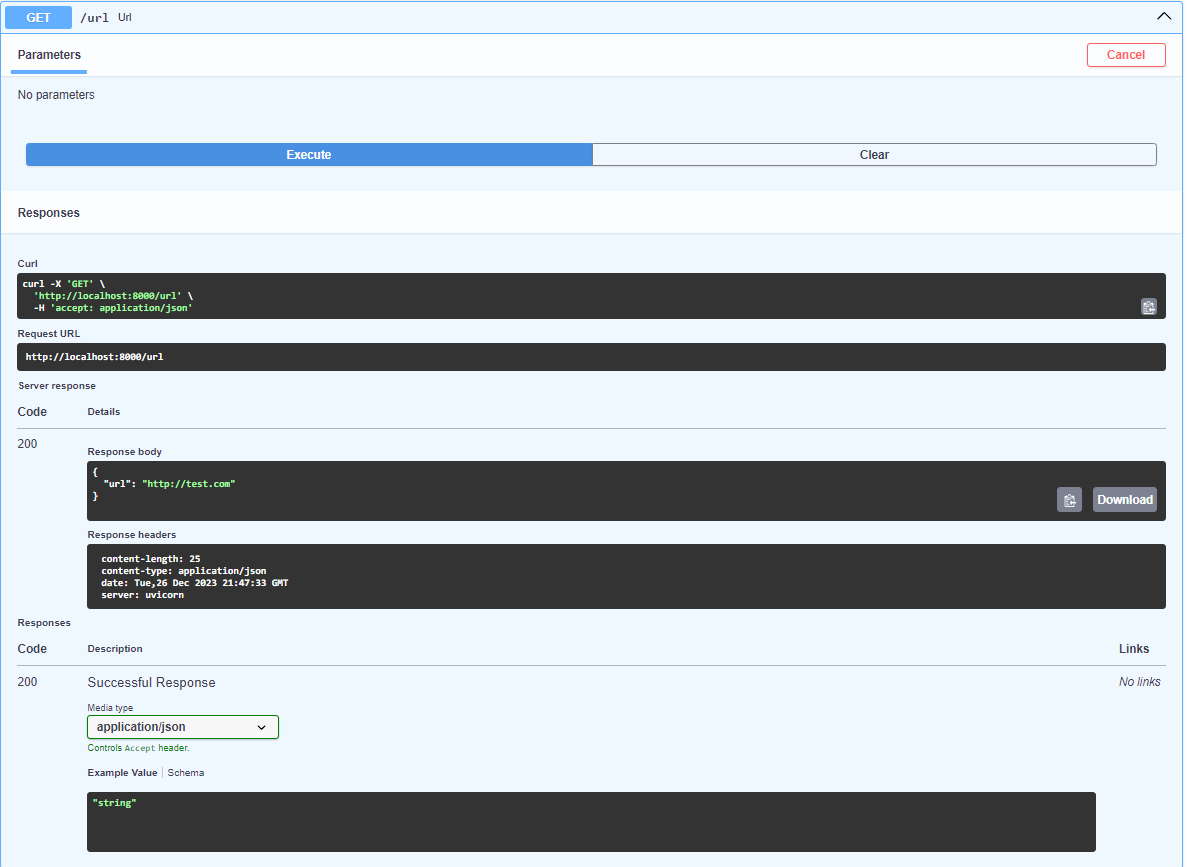
* Uso de Swagger: Una página web que nos da la información de la API.

<http://localhost:8000/docs>

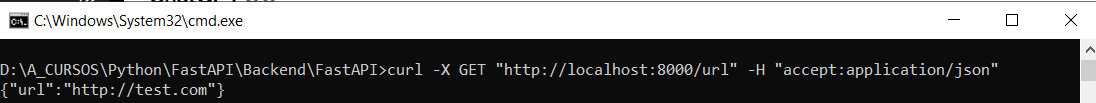




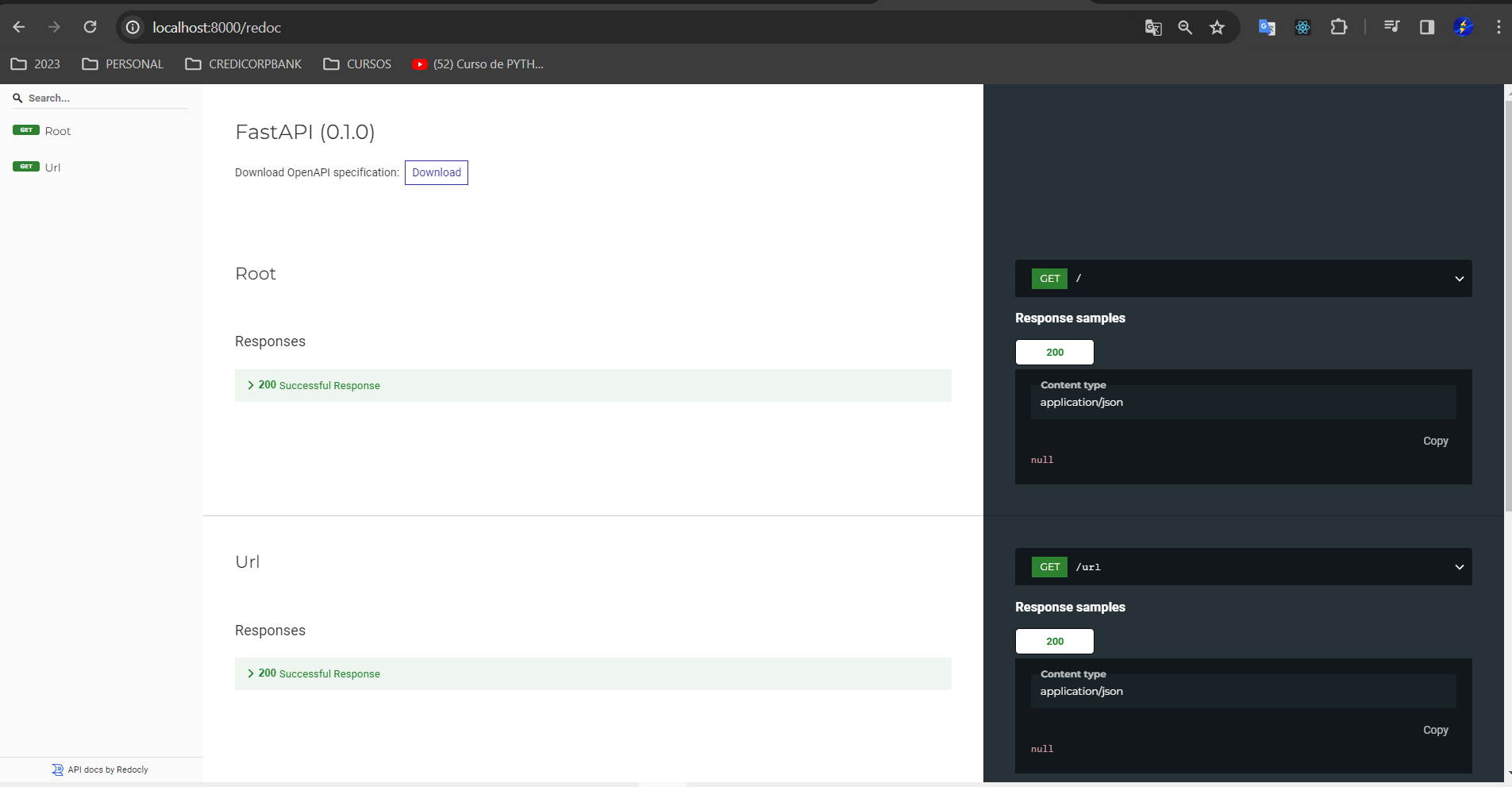
* Probar



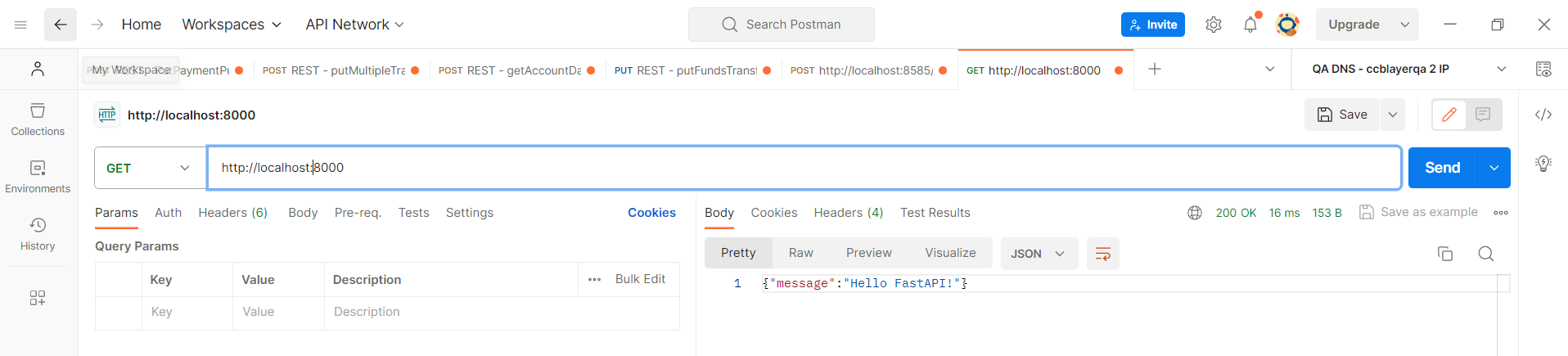
* Desde curl



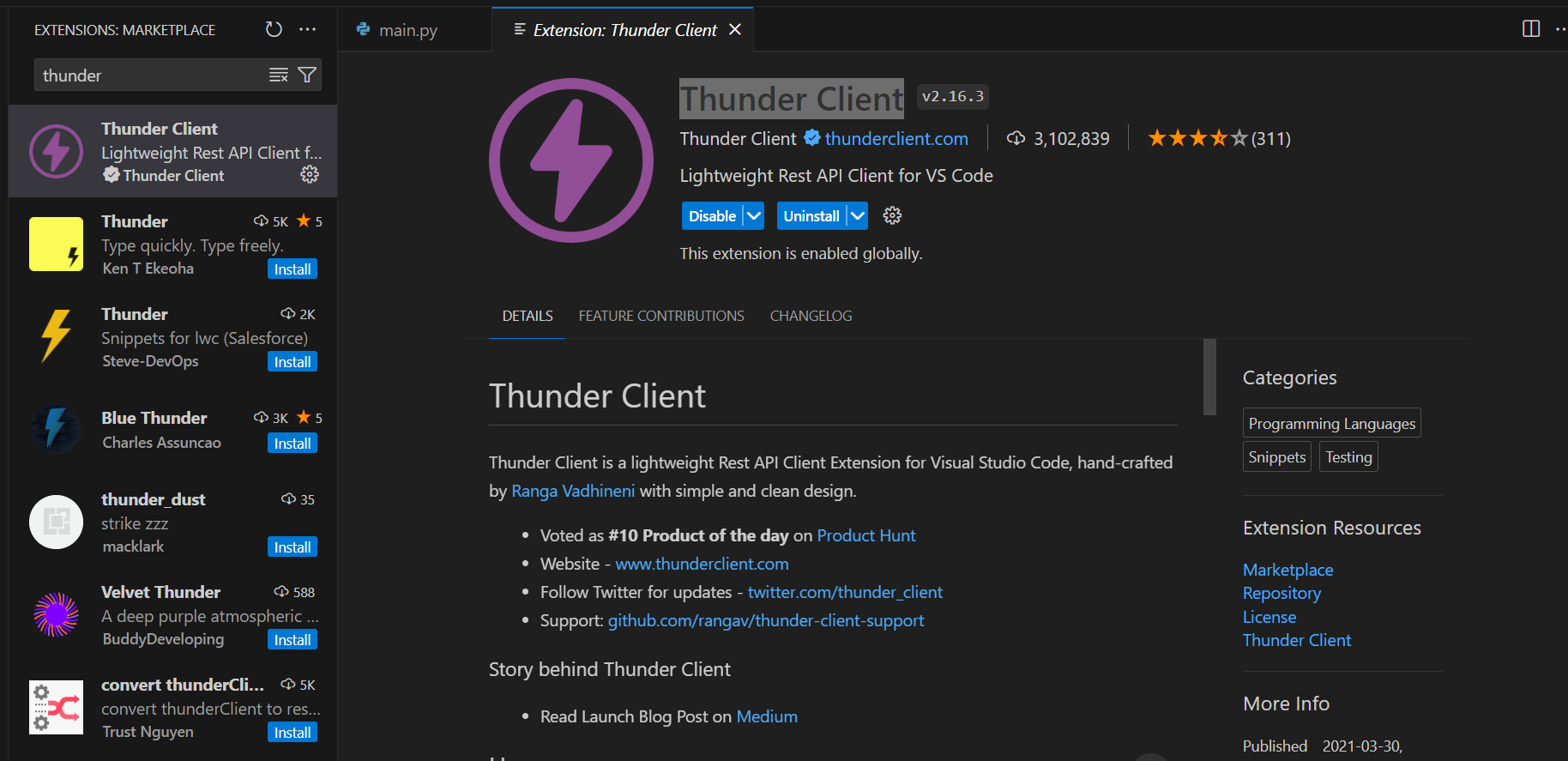
* Otra forma de probar y documentar redocly

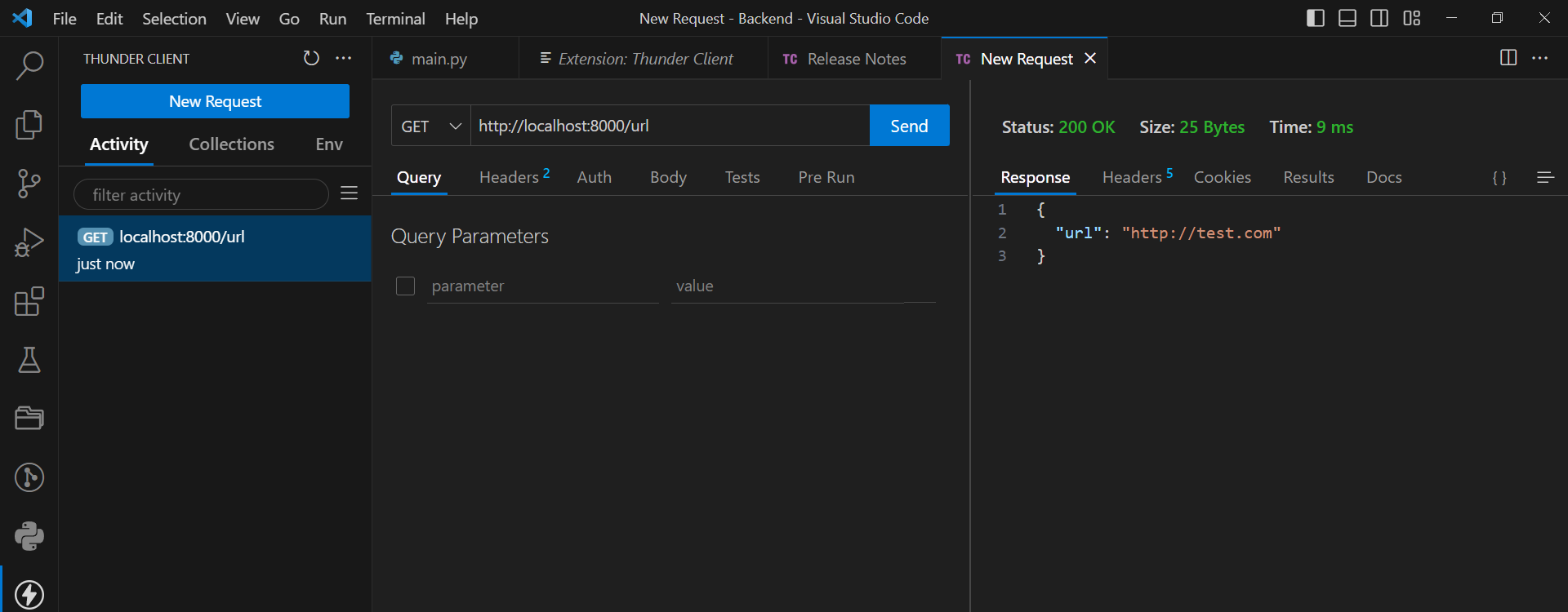
<http://localhost:8000/redoc>

* Usar **Postman**: cliente para probar servicios API



* Usar plugin VSCODE para probar servicios API (Thunder Client)





**Operaciones**

Cuando construyes APIs, normalmente usas uno de estos métodos específicos de HTTP para realizar una acción específica.

Normalmente usas:

* POST: para crear datos.
* GET: para leer datos.
* PUT: para actualizar datos.
* DELETE: para borrar datos.

Así que en OpenAPI, cada uno de estos métodos de HTTP es referido como una "operación".

Nosotros también los llamaremos "**operación**".

 los más exóticos:

* OPTIONS
* HEAD
* PATCH: Para actualizar un solo campo del registro
* TRACE

También puedes usar las otras operaciones:

* @app.post()
* @app.put()
* @app.delete()

**Operaciones CRUD para Usuarios**

* Crear método get para usuarios

from fastapi import FastAPI

app = FastAPI()

@app.get("/users")

async def users():

    return []

### Iniciar servidor: python -m uvicorn users:app --reload

* Agregar modelo

from pydantic import BaseModel

Genero Clase con uso de BaseModel y creo una lista de usuarios

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

# Entidad Usuario

class User(BaseModel):

    name:str

    username: str

    url: str

    age: int

# Lista DUMMY de usuarios

users\_fake\_db = [User("Leonel", "Messi", "https://leonel.com", 36)]

@app.get("/usersjson")

async def usersjson():

    return [{"name":"Leonel", "surname":"Messi", "url":"https://leonel.com", "age": 36},

            {"name":"Cristiano", "surname":"Ronaldo", "url":"https://cr7.com", "age": 37},

            {"name":"Neymar", "surname":"Junior", "url":"https://ney.com", "age": 32},

    ]

### Iniciar servidor: python -m uvicorn users:app --reload

* Finalmente

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

# Entidad Usuario

class User(BaseModel):

    name:str

    surname: str

    url: str

    age: int

# Lista DUMMY de usuarios

users\_list = [User(name="Leonel", surname= "Messi", url= "https://leonel.com", age= 35),

              User(name="Cristiano", surname= "Ronaldo", url= "https://cr7.com", age= 37),

              User(name="Neymar", surname= "Junior", url= "https://ney.com", age= 32)]

@app.get("/usersjson")

async def usersjson():

    return [{"name":"Leonel", "surname":"Messi", "url":"https://leonel.com", "age": 35},

            {"name":"Cristiano", "surname":"Ronaldo", "url":"https://cr7.com", "age": 37},

            {"name":"Neymar", "surname":"Junior", "url":"https://ney.com", "age": 32},

    ]

@app.get("/usersclass")

async def usersclass():

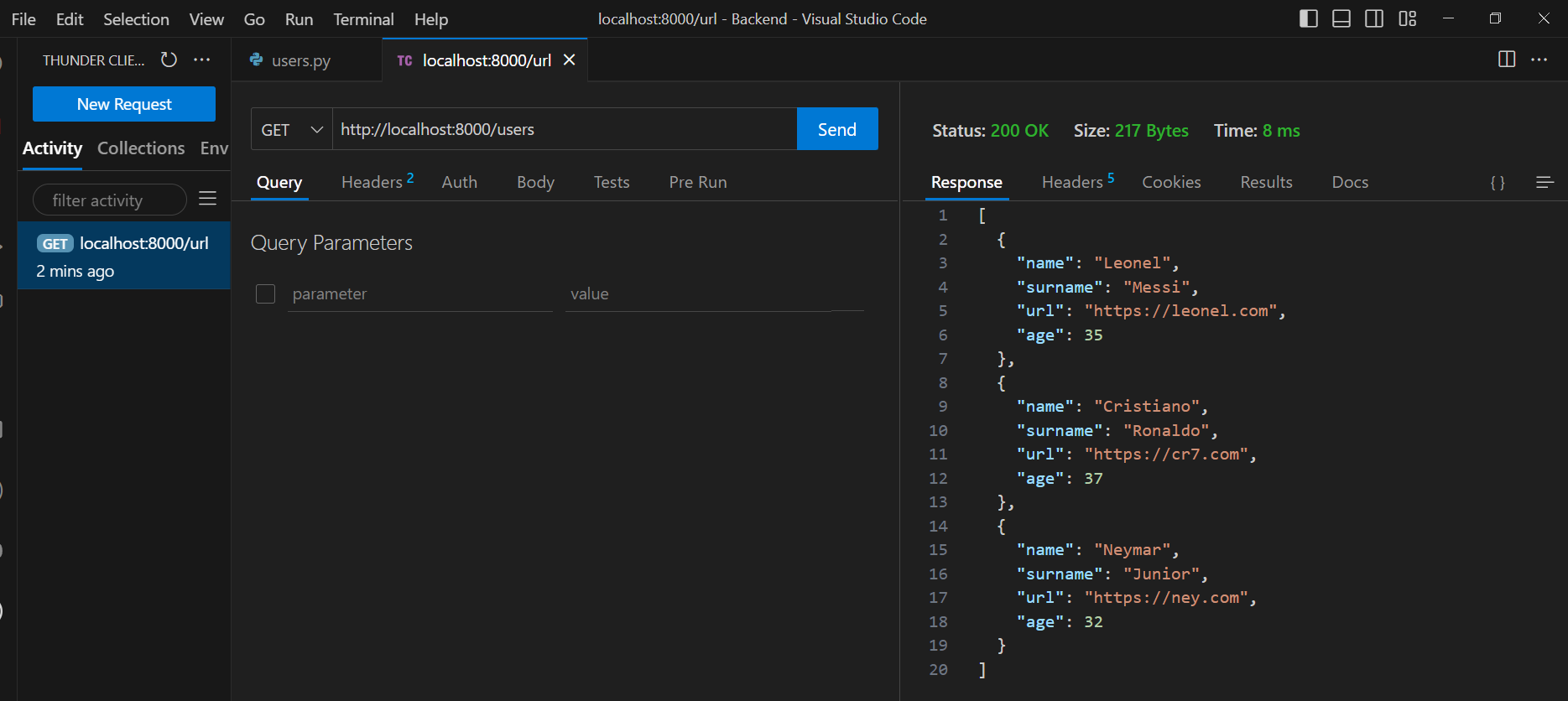
    return User(name="Leonel", surname="Messi", url="https://leonel.com", age= 36)

@app.get("/users")

async def users():

    return users\_list

### Iniciar servidor: python -m uvicorn users:app --reload



* Get con path

@app.get("/user/{id}")

async def users(id: int):

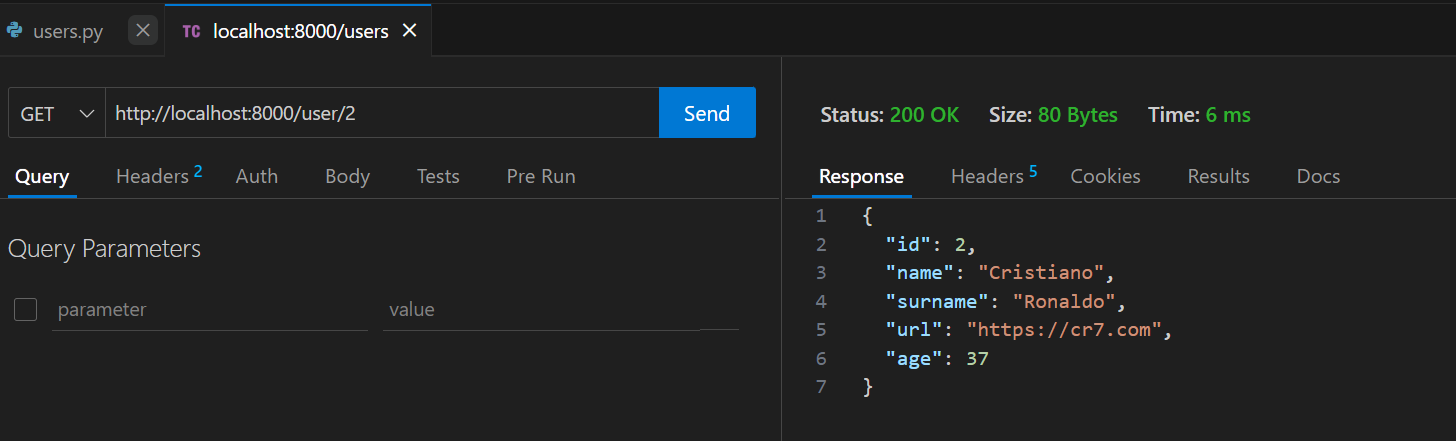
    users = filter(lambda user: user.id == id, users\_list)

    try:

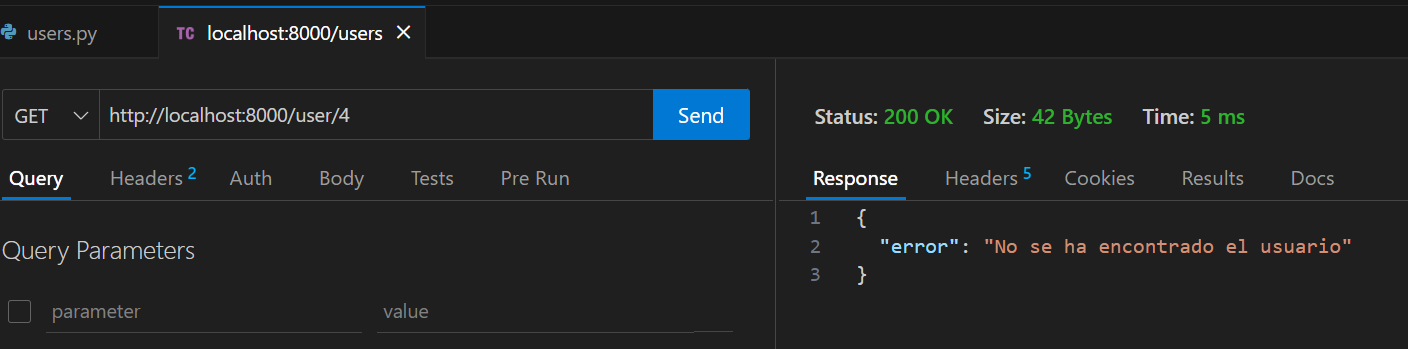
        return list(users)[0]

    except:

        return {"error": "No se ha encontrado el usuario"}



* Si no existe



* Get con Query

@app.get("/userquery")

async def users(id: int):

    users = filter(lambda user: user.id == id, users\_list)

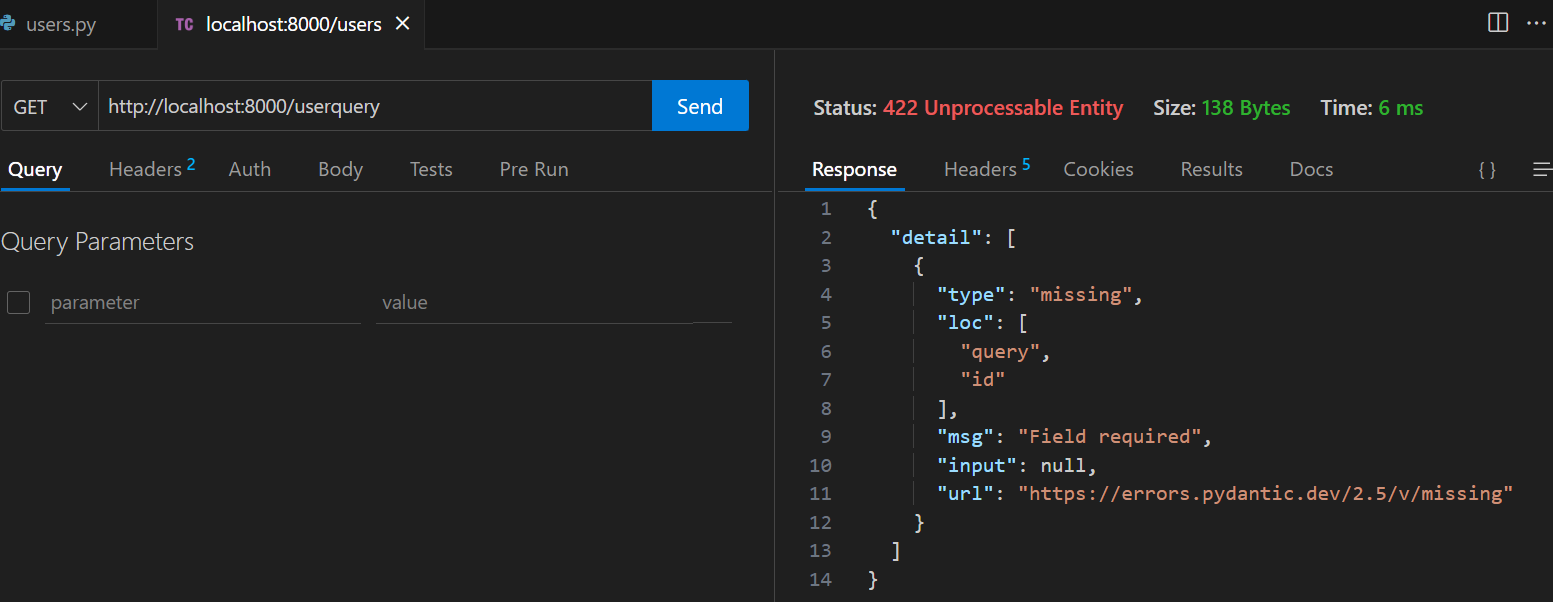
    try:

        return list(users)[0]

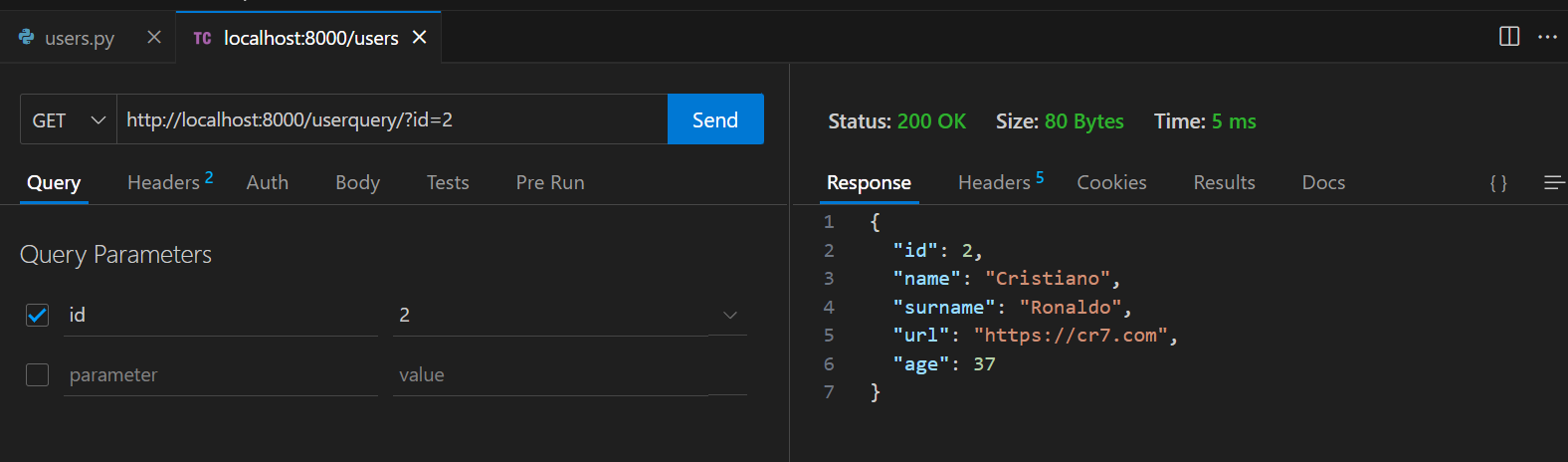
    except:

        return {"error": "No se ha encontrado el usuario"}

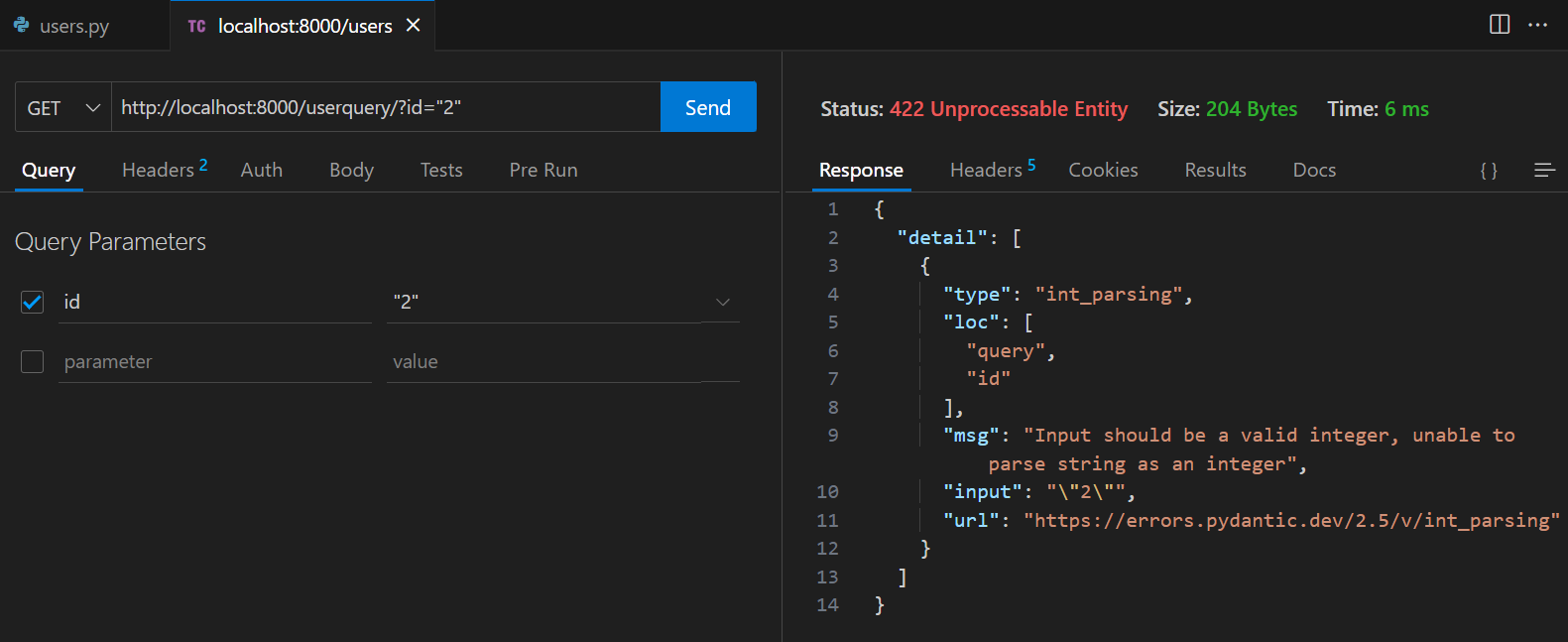
* Si lo llamo sin pasar parámetros indica error



* Si lo llamo con parámetros



* Si lo llamo con parámetro como String no entero como lo defini



* Si utilizó el mismo nombre funciona igual

# Con Path

@app.get("/user/{id}")

async def users(id: int):

   return search\_user(id)

# Con Query

@app.get("/user")

async def users(id: int):

    return search\_user(id)

* Incluso cambiando el nombre del parámetro funciona igual

# Con Path

@app.get("/user/{ide}")

async def users(ide: int):

   return search\_user(ide)

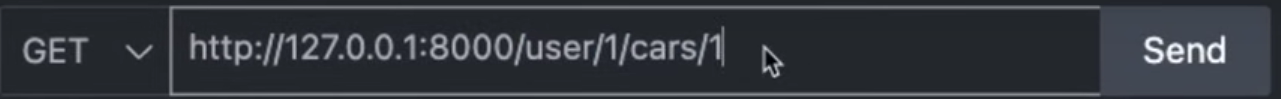
# Con Query

@app.get("/user")

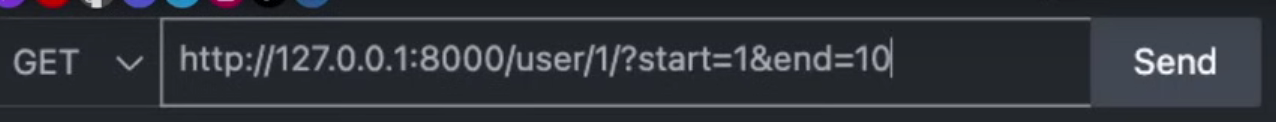
async def users(id: int):

    return search\_user(id)

* El uso de path con valores variables



* El uso de query se ve mucho en las consultas con paginaciones



**Post – Put - Delete**

* Agregar un usuario

@app.post("/user")

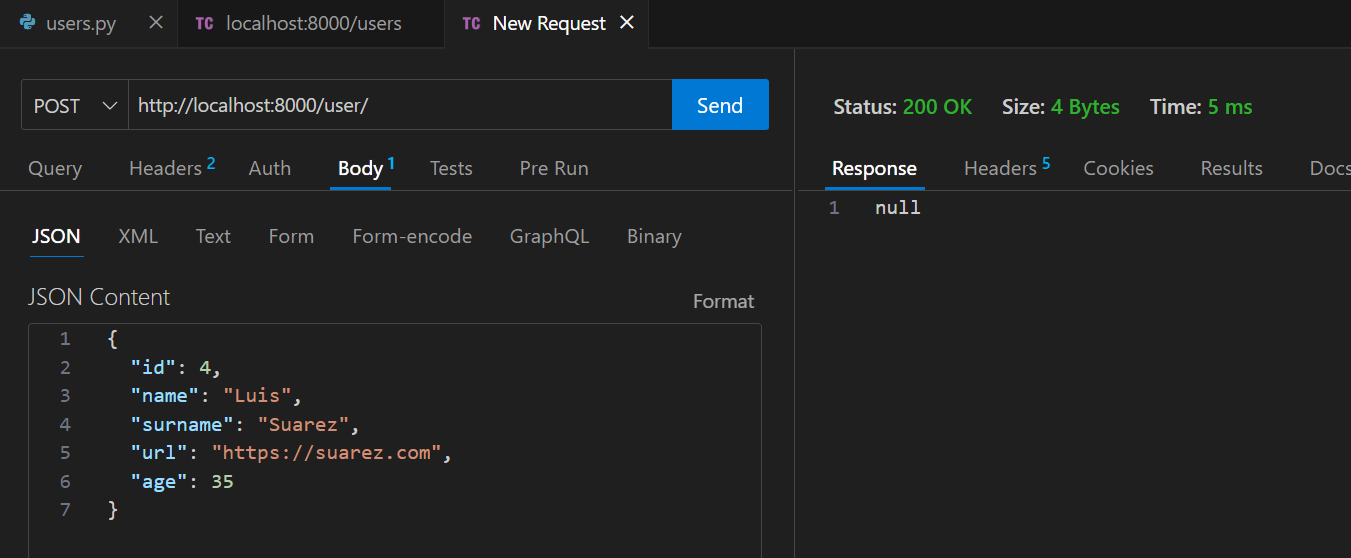
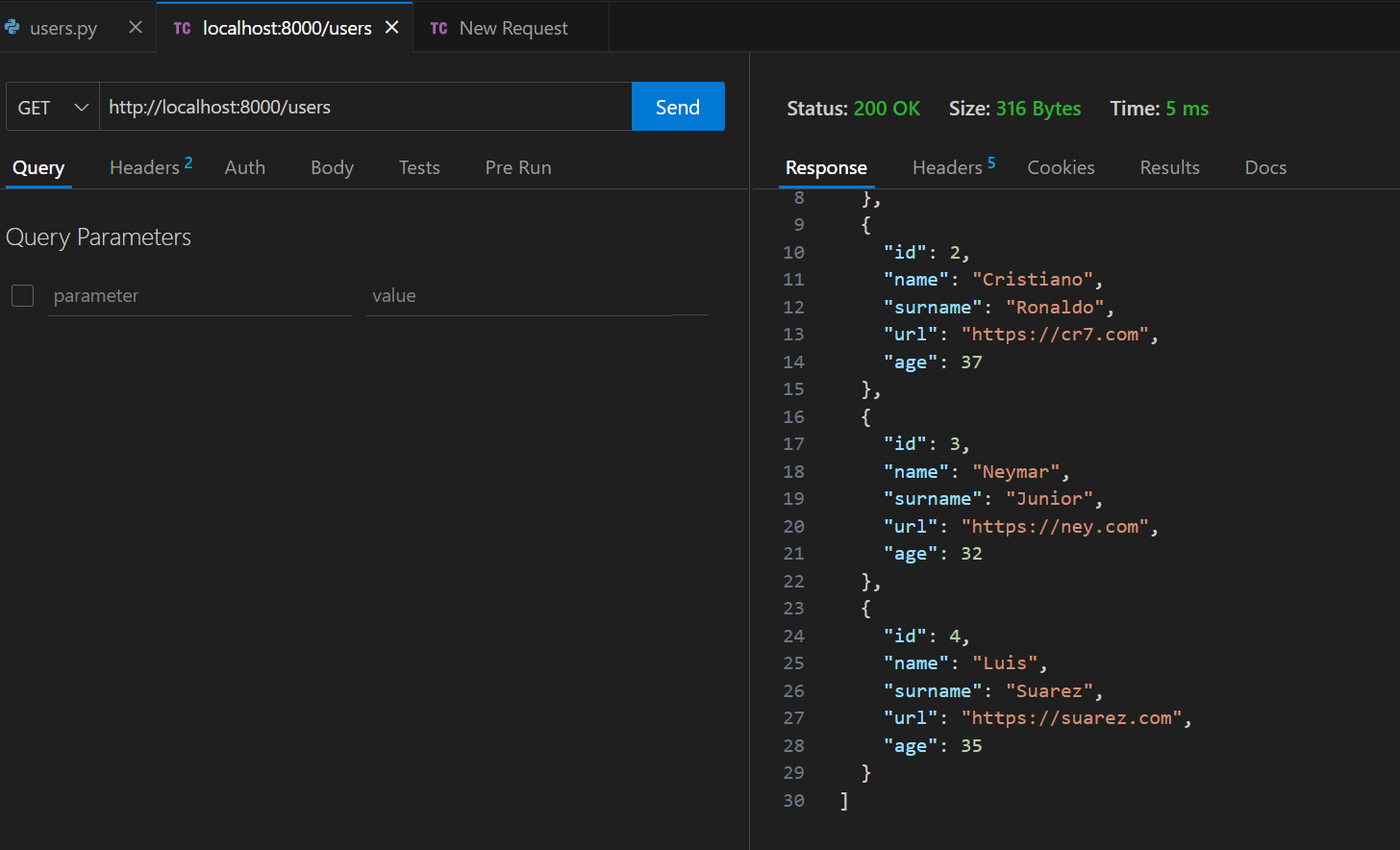
async def user(user: User):

    if type(search\_user(user.id)) == User:

        return {"error": "usuario ya existe"}

    else:

        users\_list.append(user)

* Modificar un Usuario

@app.put("/user")

async def user(user: User):

    found = False

    for index, save\_user in enumerate(users\_list):

        if save\_user.id == user.id:

            users\_list[index] = user

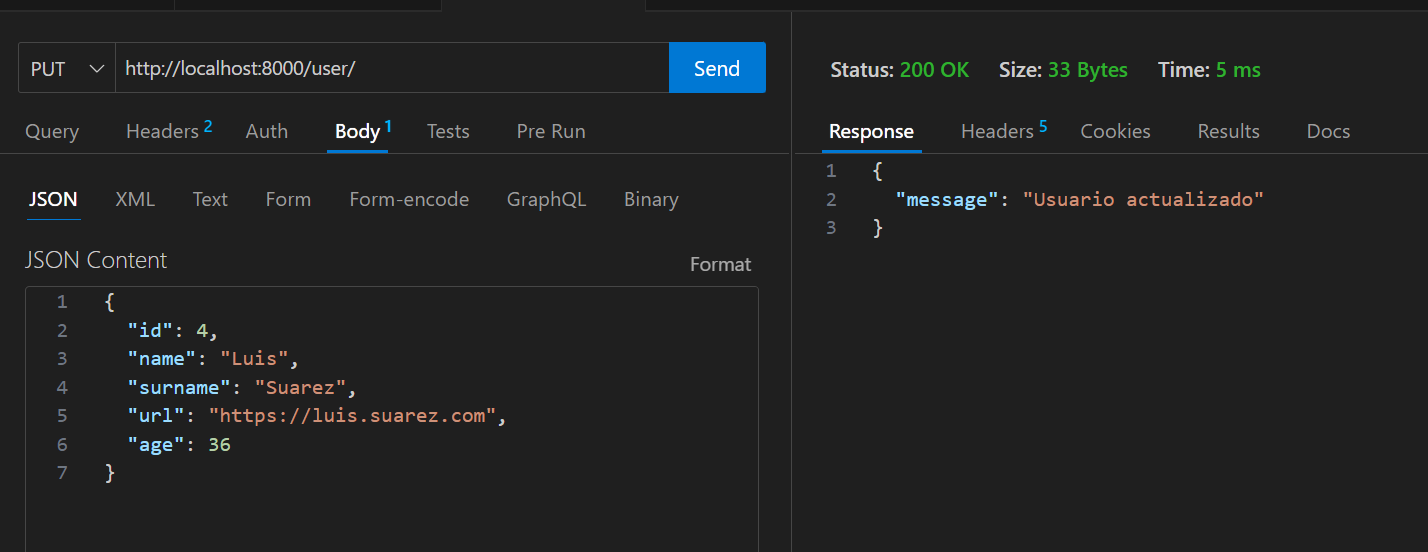
            found = True

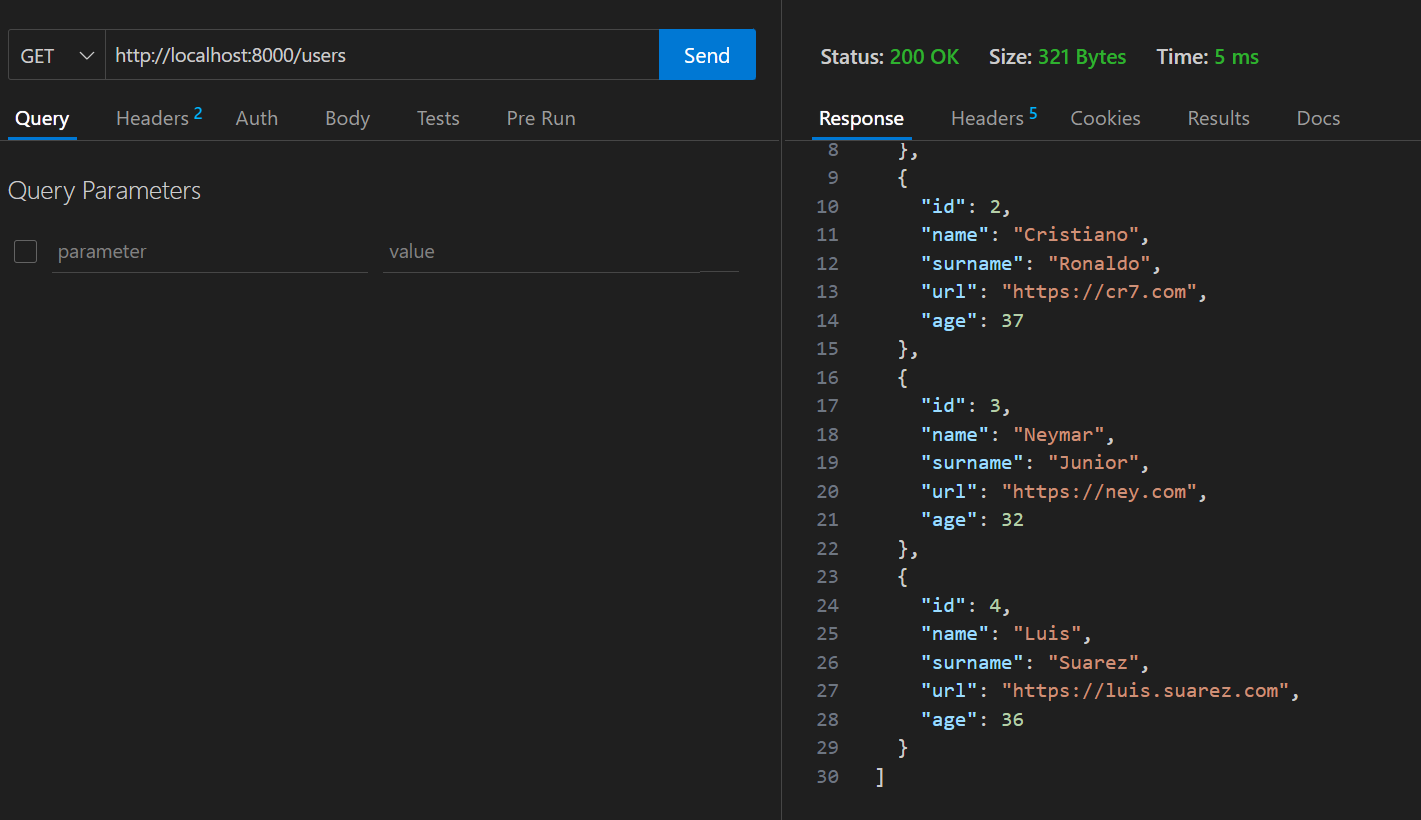
    if not found:

        return {"error": "No se ha actualizado el usuario"}

    else:

        return {"message": "Usuario actualizado"}





* Eliminar un usuario

# DELETE

@app.delete("/user/{id}")

async def user(id: int):

    found = False

    for index, save\_user in enumerate(users\_list):

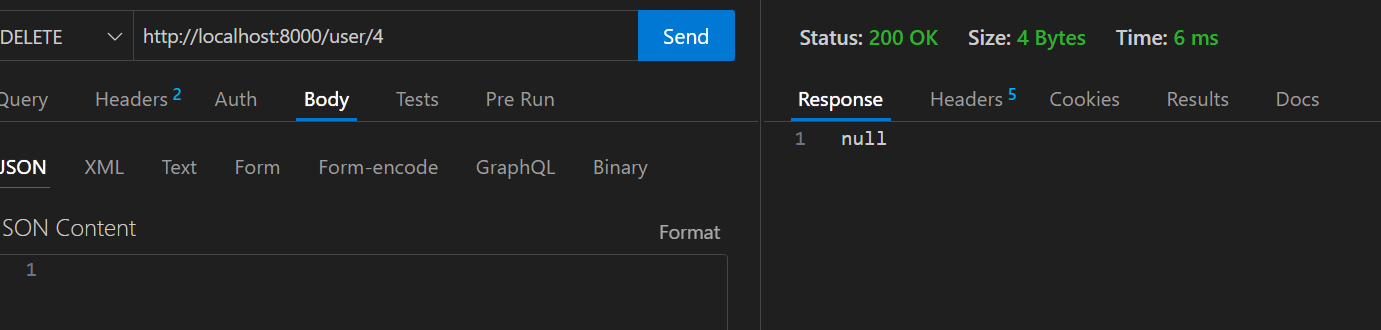
        if save\_user.id == id:

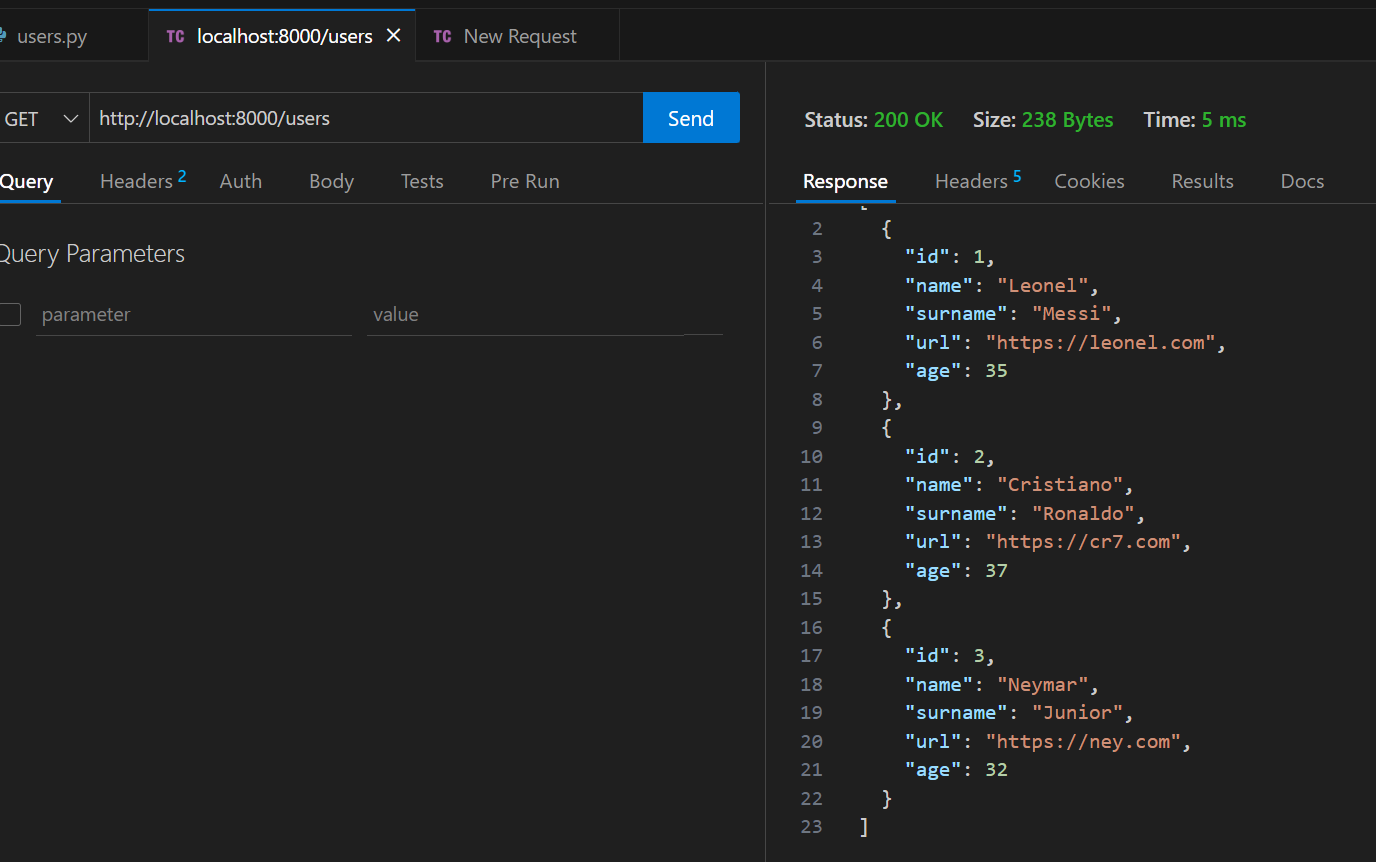
            del users\_list[index]

            found = True

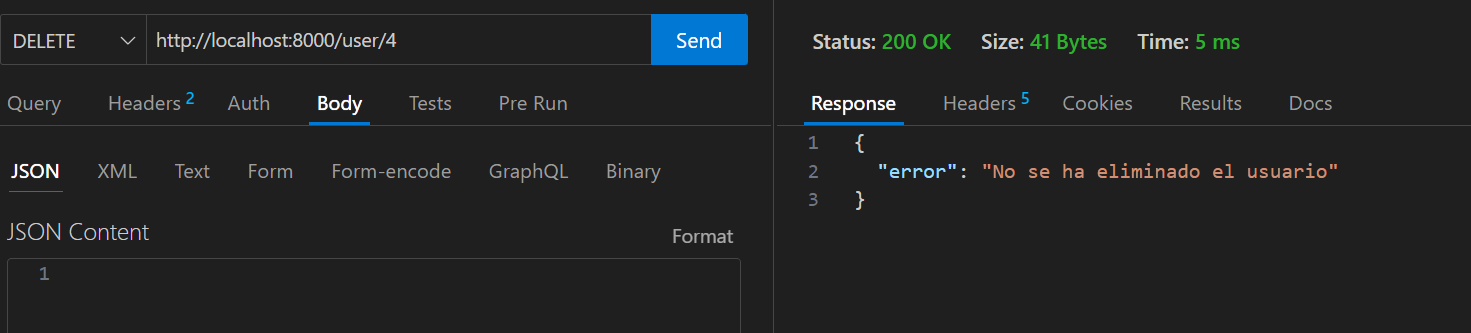
    if not found:

        return {"error": "No se ha eliminado el usuario"}





* Si repito el borrado

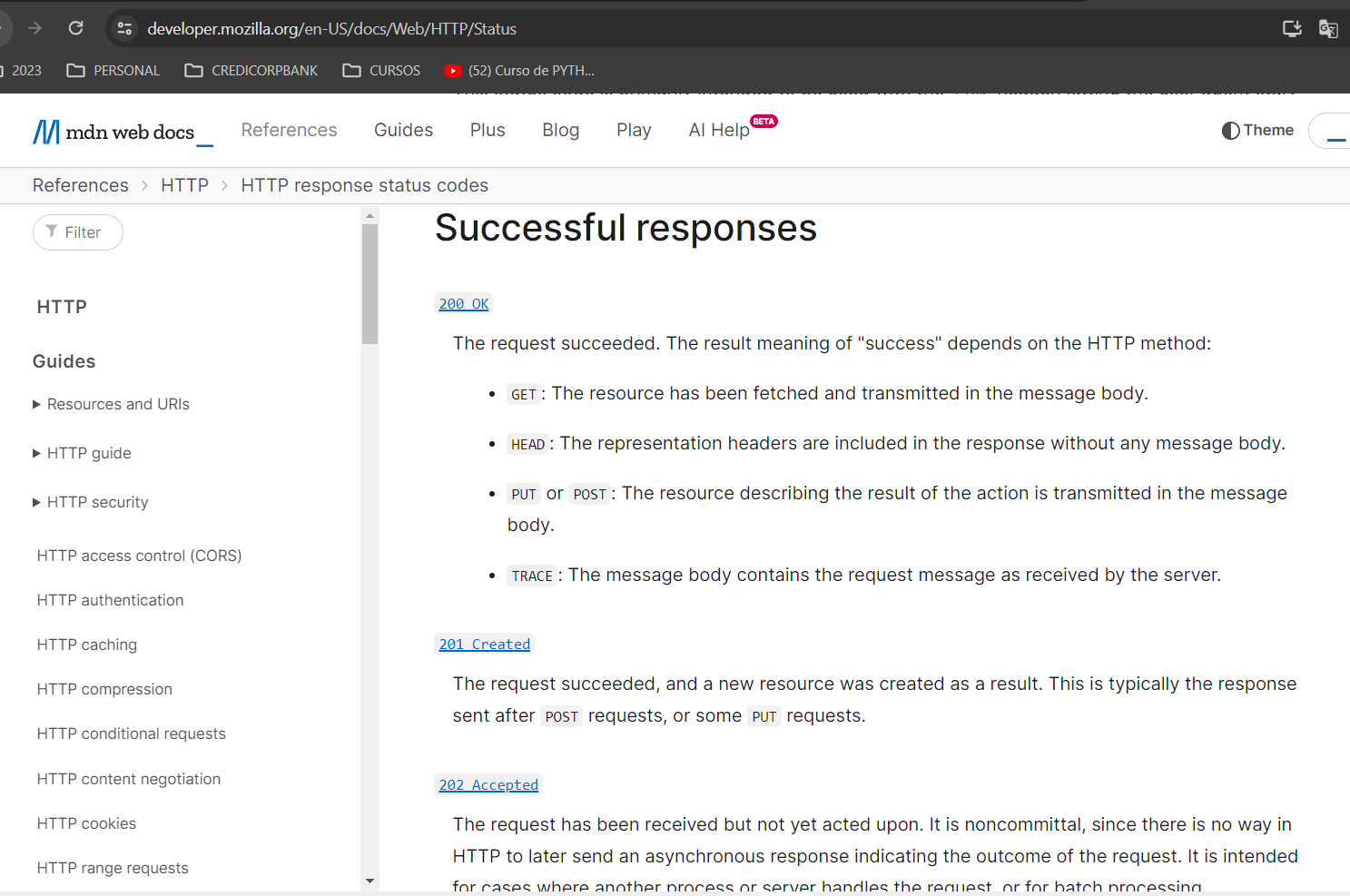


**Códigos de estado**

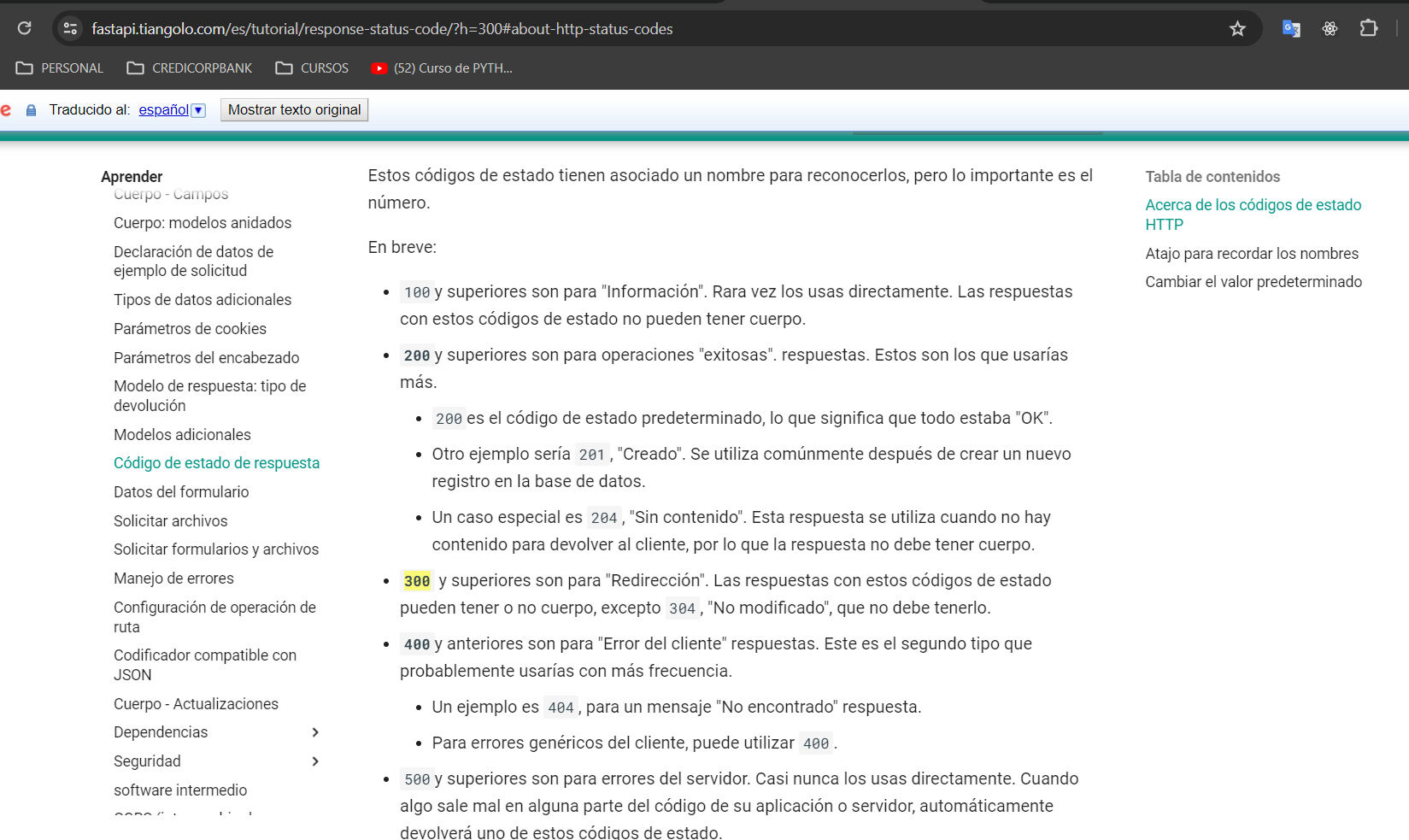
* Códigos de estado de respuesta HTTP

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

1. [Informational responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#information_responses) (100 – 199)
2. [Successful responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#successful_responses)  (200 – 299)
3. [Redirection messages](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#redirection_messages)  (300 – 399)
4. [Client error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses)  (400 – 499)
5. [Server error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#server_error_responses)  (500 – 599)



* En FastAPI: Los más estándares



* Si ajusto la creación de usuario (Post).

Agrego el parámetro status\_code=201 al @app.post(“/user”)

# POST

@app.post("/user",status\_code=201)

async def user(user: User):

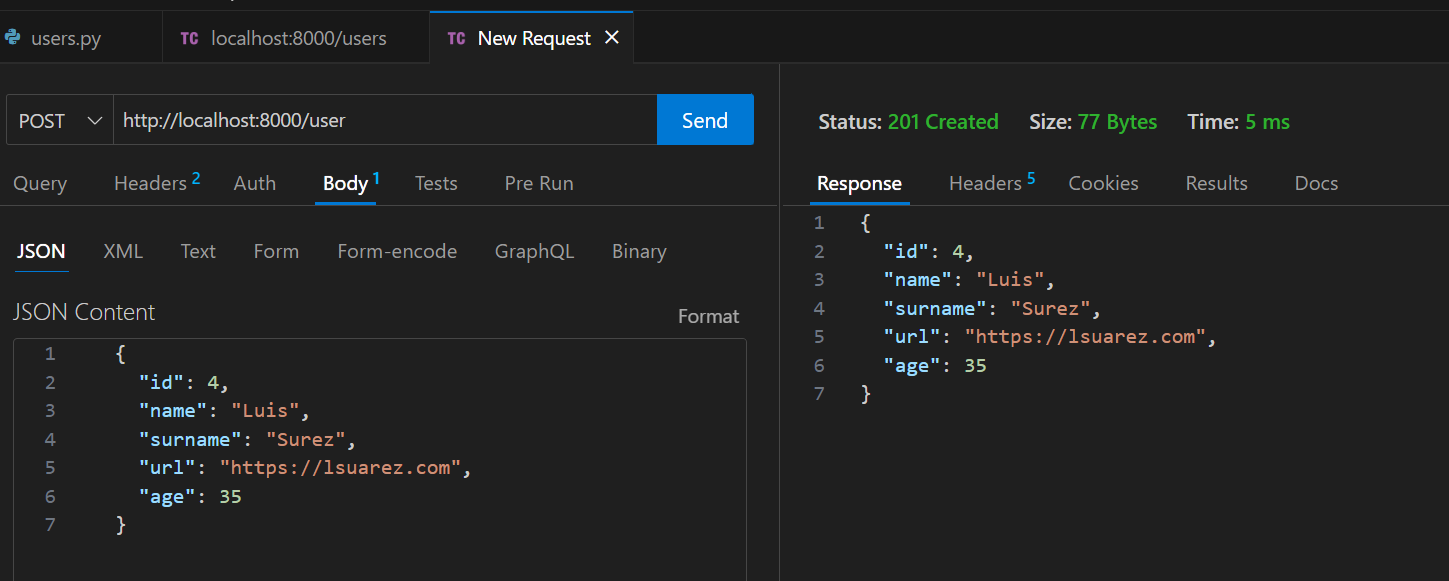
    if type(search\_user(user.id)) == User:

        return {"error": "usuario ya existe"}

    users\_list.append(user)

    return user

* Se retorna un 201 como código de estado de respuesta



* Enviar un error en caso de problema
  + Agregar HTTPException de FastAPI

from fastapi import FastAPI, HTTPException

* + Agregar Exception al código

# POST

@app.post("/user",status\_code=201)

async def user(user: User):

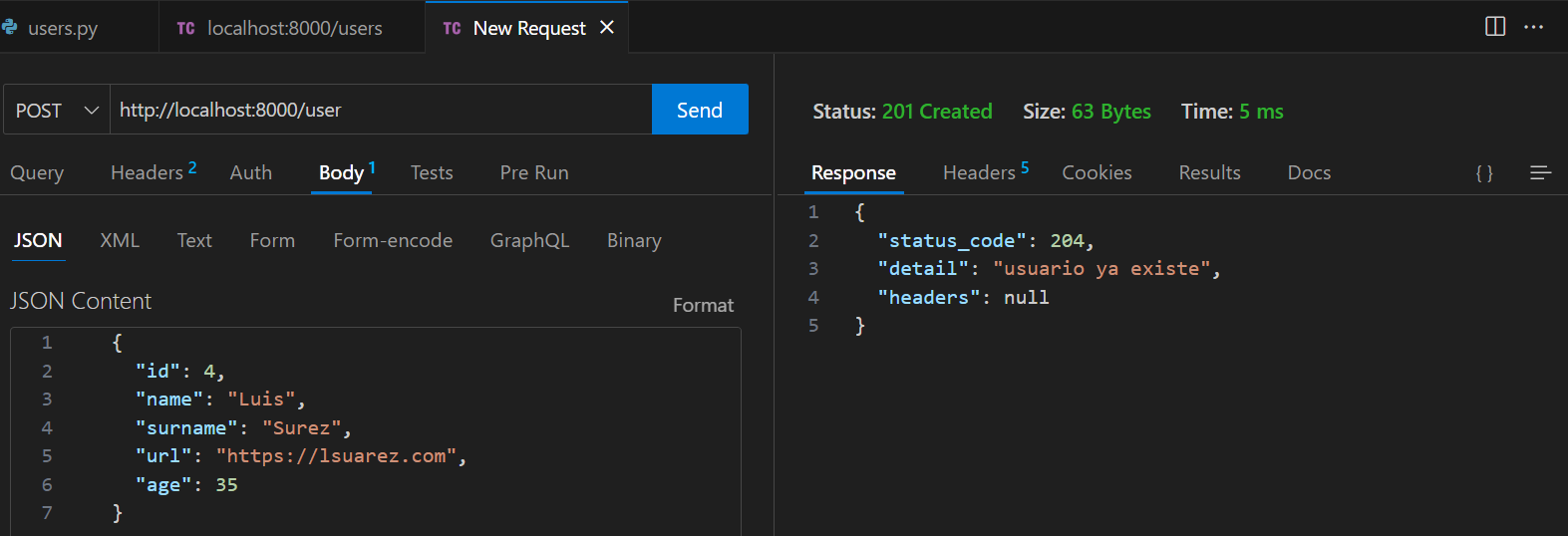
    if type(search\_user(user.id)) == User:

        return HTTPException(status\_code=204, detail="usuario ya existe")

    users\_list.append(user)

    return user

* Probar enviando dos veces el mismo usuario



* Sigue el problema. Se debe retornar a través de un raise: Lanzar la exception

# POST

@app.post("/user",status\_code=201)

async def user(user: User):

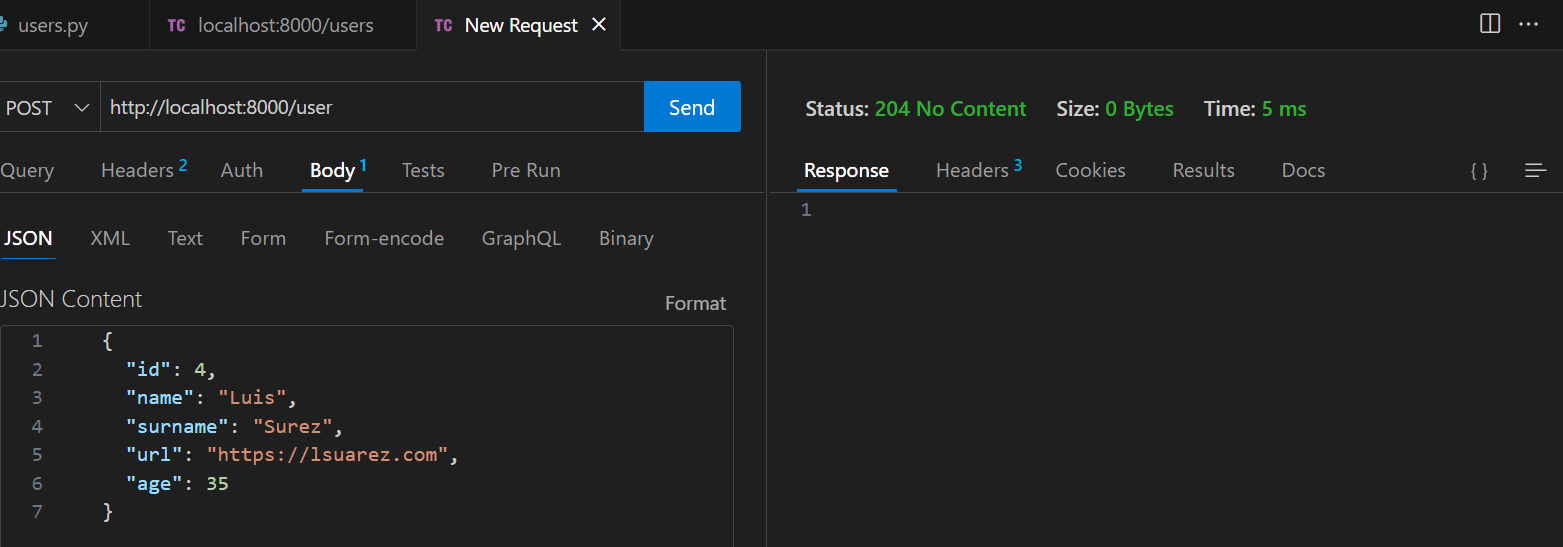
    if type(search\_user(user.id)) == User:

        raise HTTPException(status\_code=204, detail="usuario ya existe")

    users\_list.append(user)

    return user

* Probar enviando dos veces el mismo usuario



Como el 204

[204 No Content](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204)

No hay contenido para enviar para esta solicitud, pero los encabezados pueden resultar útiles. El agente de usuario puede actualizar sus encabezados almacenados en caché para este recurso con los nuevos.

No retorna el detalle que se envió en el backend

* Si se cambia por un 404

# POST

@app.post("/user",status\_code=201)

async def user(user: User):

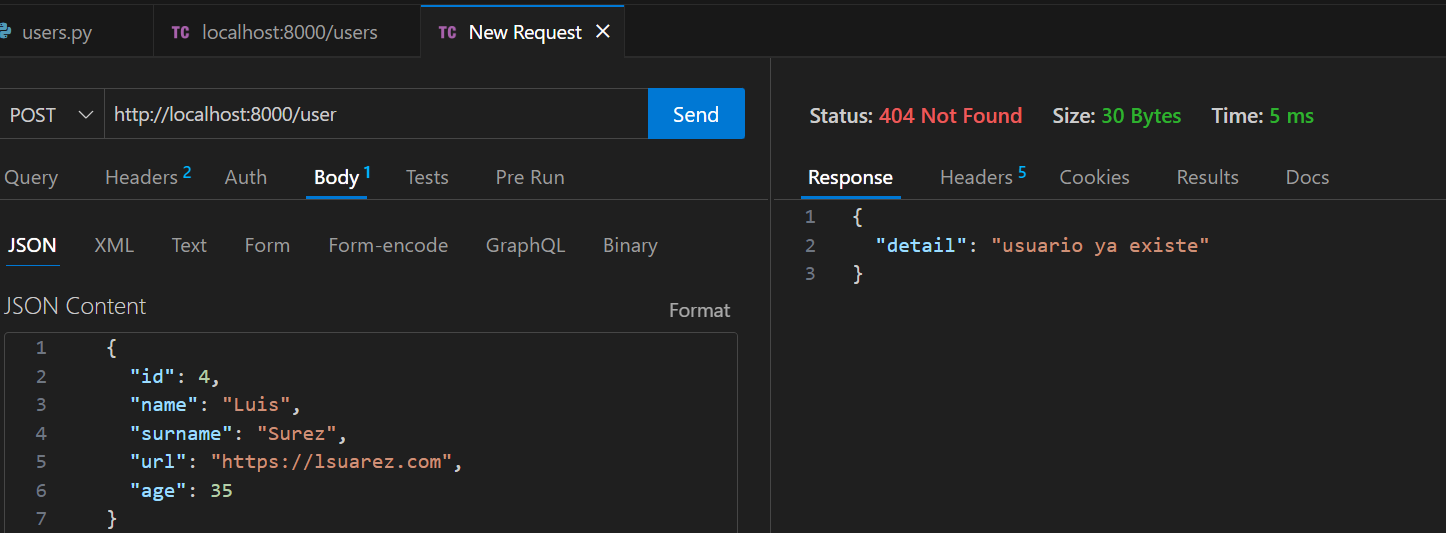
    if type(search\_user(user.id)) == User:

        raise HTTPException(status\_code=404, detail="usuario ya existe")

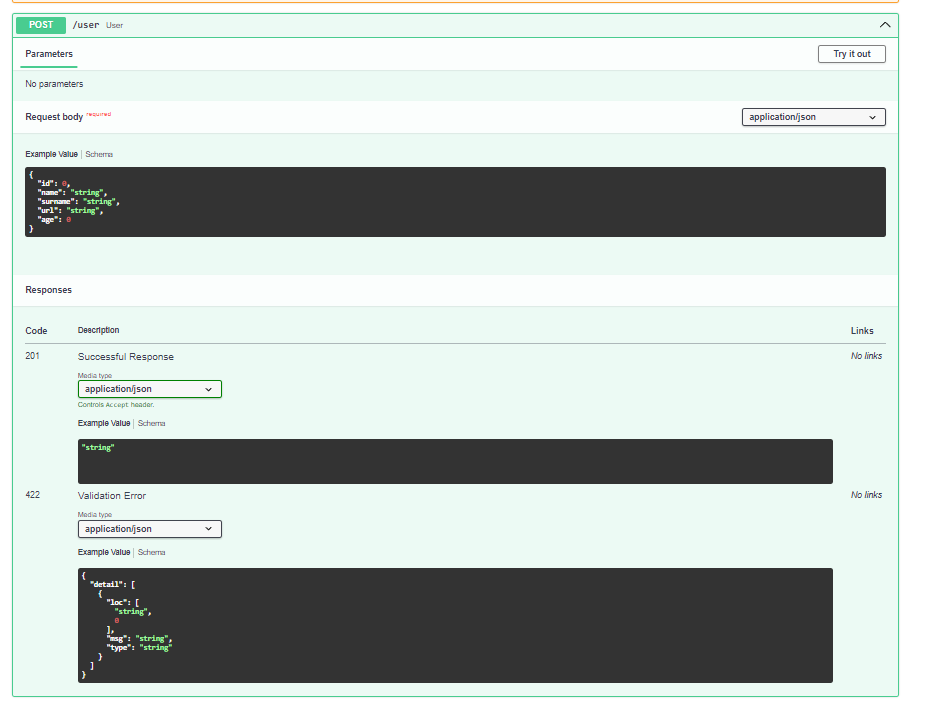
    users\_list.append(user)

    return user

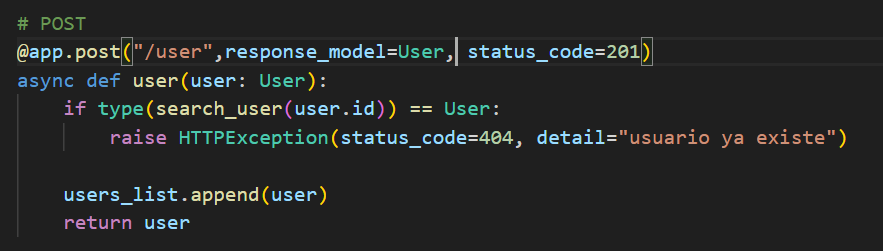
* Si despliega el detalle



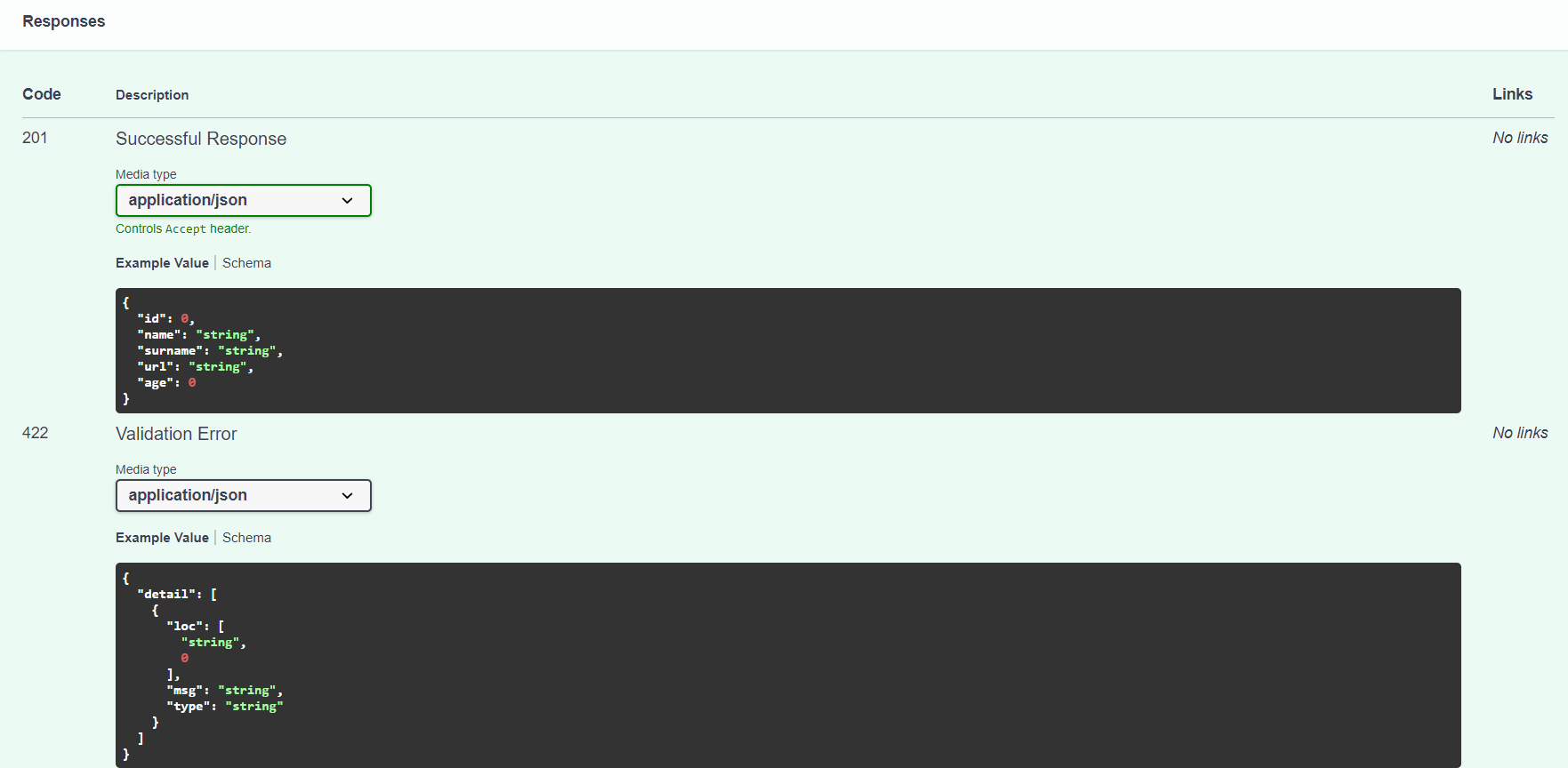
* Si vemos la documentación. Códigos de error hasta el modelo User



* Agregamos que debe responder en caso de éxito y que en caso de error
  + Response\_model y status\_code



* En la documentación ya muestra que debe responder en caso de éxito (201 con objeto User) y en caso de error por defecto 222



**Uso Routers**

from fastapi import FastAPI

from routers import products, users

app = FastAPI()

# Routers

app.include\_router(products.router)

app.include\_router(users.router)

@app.get("/")

async def root():

    return {"message": "Hello FastAPI!"}

@app.get("/url")

async def url():

    return {"url": "http://test.com"}

from fastapi import APIRouter

from pydantic import BaseModel

router = APIRouter(prefix="/products",

tags=["products"],

                  responses={404: {"message": "No encontrado"}})

# Entidad Producto

class Product(BaseModel):

    id:int

    value:str

products\_list = [Product(id= 1, value= "Producto 1"),

                 Product(id= 2, value= "Producto 2"),

                 Product(id= 3, value= "Producto 3"),

                 Product(id= 4, value= "Producto 4"),

                 Product(id= 5, value= "Producto 5")]

@router.get("/")

async def products():

    return products\_list

@router.get("/{id}")

async def product(id: int):

    return search\_product(id)

def search\_product(id: int):

    products = filter(lambda product: product.id == id, products\_list)

    try:

        return list(products)[0]

    except:

        return list(products)[0]

        #return {"error": "No se ha encontrado el producto"}

from fastapi import APIRouter, HTTPException

from pydantic import BaseModel

router = APIRouter()

# Entidad Usuario

class User(BaseModel):

    id:int

    name:str

    surname: str

    url: str

    age: int

# Lista DUMMY de usuarios

users\_list = [User(id=1, name="Leonel", surname= "Messi", url= "https://leonel.com", age= 35),

              User(id=2, name="Cristiano", surname= "Ronaldo", url= "https://cr7.com", age= 37),

              User(id=3, name="Neymar", surname= "Junior", url= "https://ney.com", age= 32)]

@router.get("/usersjson")

async def usersjson():

    return [{"name":"Leonel", "surname":"Messi", "url":"https://leonel.com", "age": 35},

            {"name":"Cristiano", "surname":"Ronaldo", "url":"https://cr7.com", "age": 37},

            {"name":"Neymar", "surname":"Junior", "url":"https://ney.com", "age": 32},

    ]

@router.get("/usersclass")

async def usersclass():

    return User(name="Leonel", surname="Messi", url="https://leonel.com", age= 36)

@router.get("/users")

async def users():

    return users\_list

# Con Path

@router.get("/user/{id}")

async def users(id: int):

   return search\_user(id)

# Con Query

@router.get("/userquery")

async def users(id: int):

    return search\_user(id)

# POST

@router.post("/user",response\_model=User, status\_code=201)

async def user(user: User):

    if type(search\_user(user.id)) == User:

        raise HTTPException(status\_code=404, detail="usuario ya existe")

    users\_list.append(user)

    return user

# PUT

@router.put("/user",response\_model=User, status\_code=202)

async def user(user: User):

    found = False

    for index, save\_user in enumerate(users\_list):

        if save\_user.id == user.id:

            users\_list[index] = user

            found = True

    if not found:

        raise HTTPException(status\_code=406, detail="No se ha actualizado el usuario")

    return user

# DELETE

@router.delete("/user/{id}")

async def user(id: int):

    found = False

    for index, save\_user in enumerate(users\_list):

        if save\_user.id == id:

            del users\_list[index]

            found = True

    if not found:

        return {"error": "No se ha eliminado el usuario"}

def search\_user(id: int):

    users = filter(lambda user: user.id == id, users\_list)

    try:

        return list(users)[0]

    except:

        return {"error": "No se ha encontrado el usuario"}

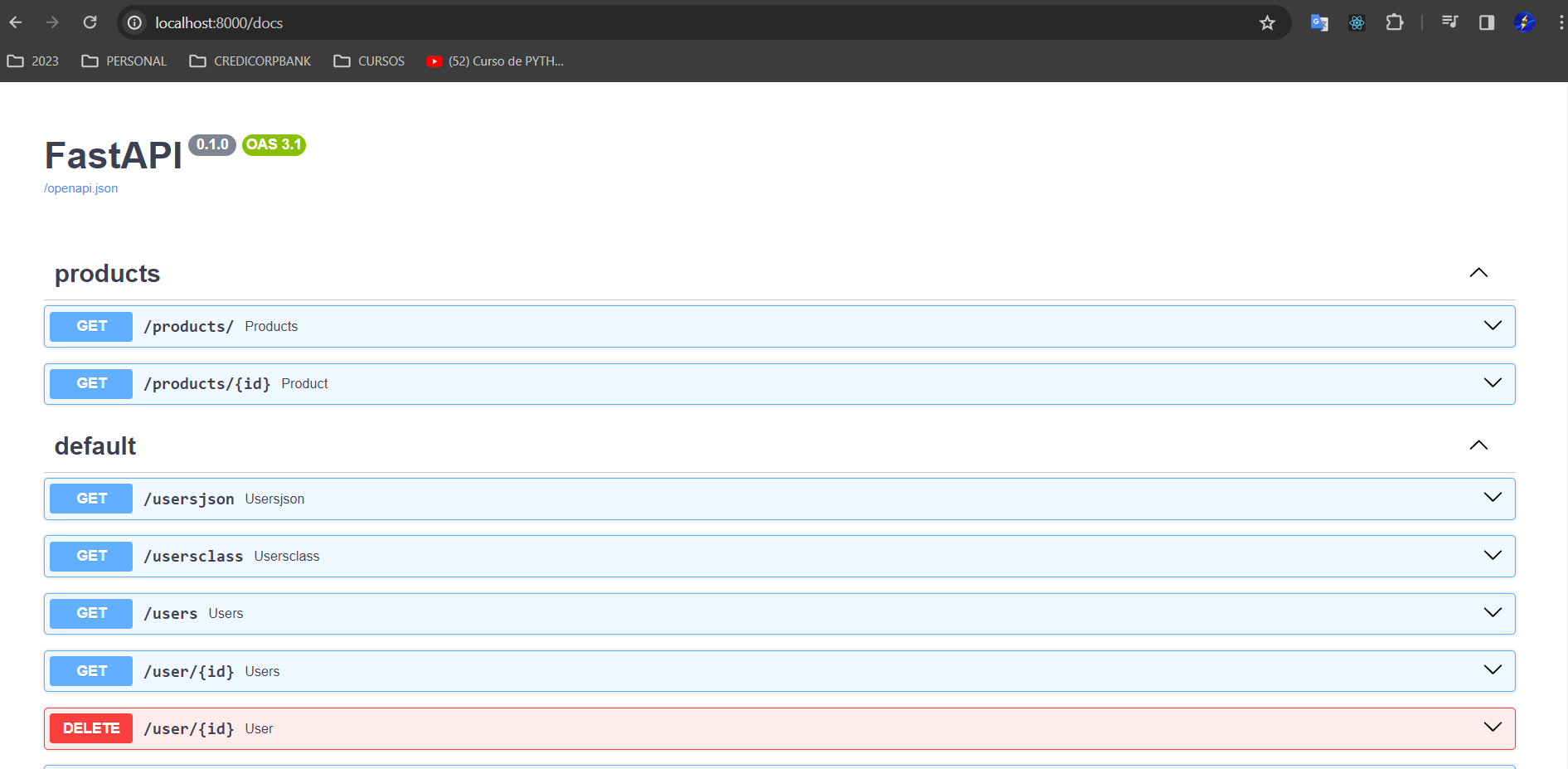
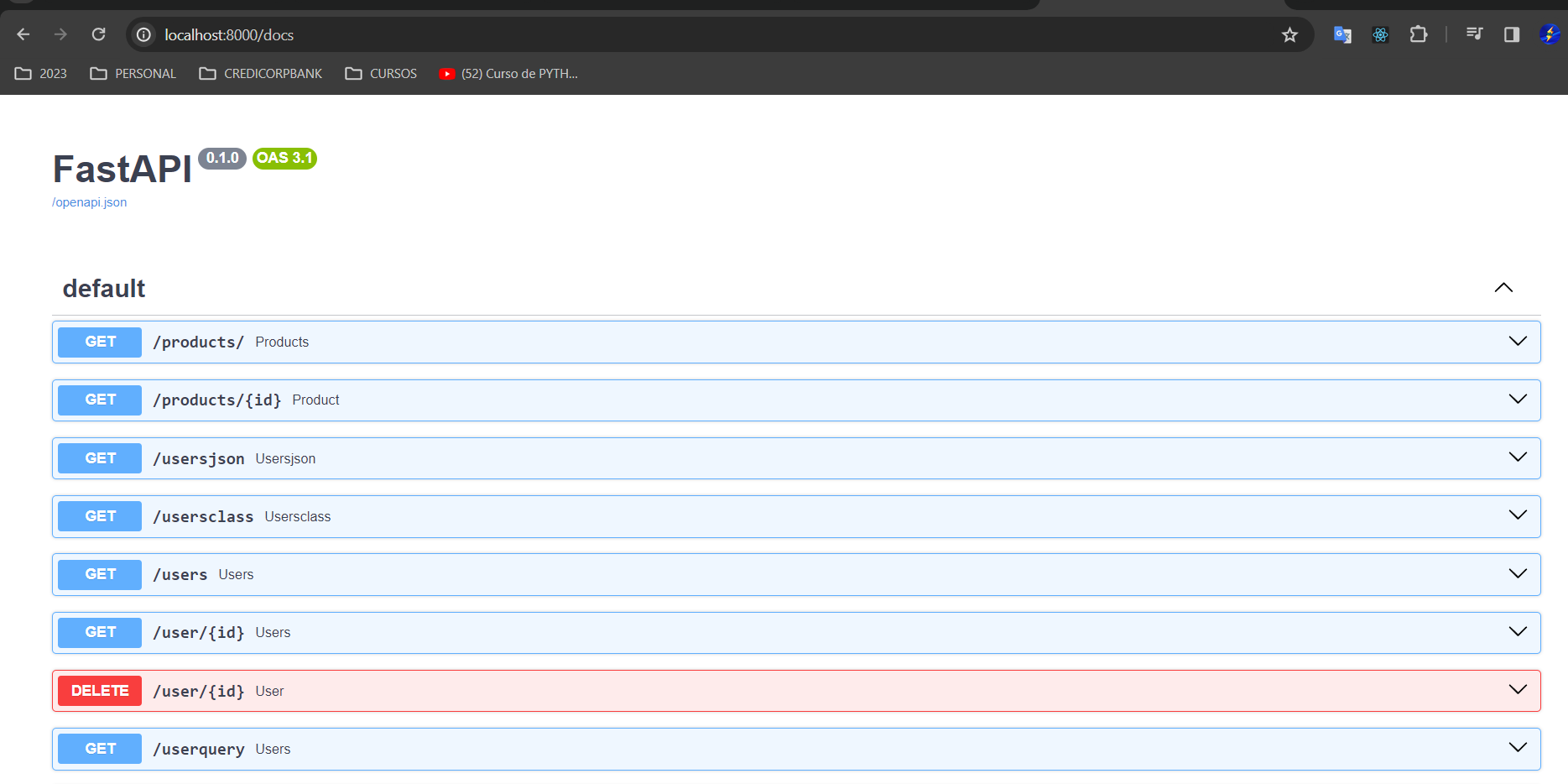
### Iniciar servidor: python -m uvicorn users:router --reload

* El uso de Tags en productos. Actualiza la documentación.

router = APIRouter(prefix="/products",

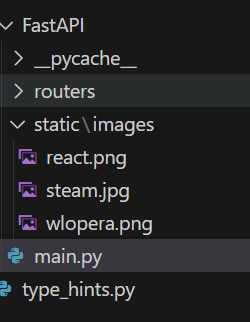
tags=["products"],

                  responses={404: {"message": "No encontrado"}})



**Exponer recursos estáticos**

* Creamos la carpeta @static/images” y agregamos 3 imágenes



* En main.py, agregamos el código para publicar y exponer nuestras imágenes.

from fastapi import FastAPI

from routers import products, users

from fastapi.staticfiles import StaticFiles

app = FastAPI()

# Routers

app.include\_router(products.router)

app.include\_router(users.router)

# Exponer recursos estaticos

app.mount("/static", StaticFiles(directory="static"), name="static")

* Llamaos la URL expuesta con el archivo que queramos de la lista

<http://localhost:8000/static/images/react.png>



