



UNIVERSIDAD
NACIONAL
DE COLOMBIA

Ciclo III

Desarrollo de Software



Capa Lógica: API-REST

9

Jeisson Andrés Vergara Vargas

Departamento de Ingeniería de Sistemas e Industrial

<http://colswu.unal.edu.co/~javergarav/>
javergarav@unal.edu.co

2020

©

Objetivo de Aprendizaje

Analizar las operaciones de una API-REST y su implementación en FastAPI.

Introducción

API-REST

¿Qué es una **API-REST**?

Es una **interfaz de aplicación** que **expone** un **servicio web**, por medio de un conjunto de operaciones **HTTP** y siguiendo el estilo arquitectónico **REST**. Las operaciones más comunes son:



GET



POST

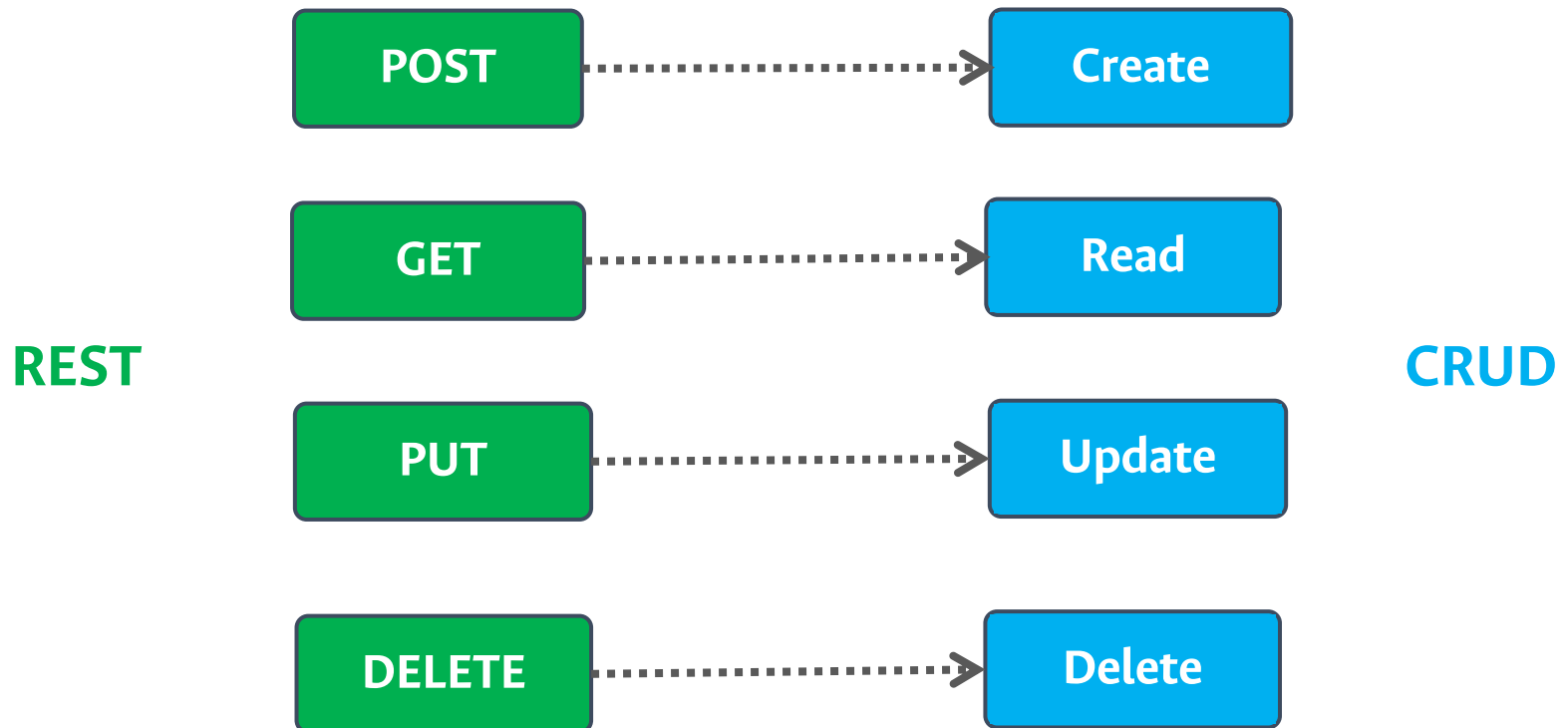


PUT



DELETE

Símil con CRUD



Estructura de un Petición

Una **petición** a una **API-REST** tiene la siguiente **estructura**:

- **Linea de inicio:** método y dirección.
- **Cabeceras (headers):** descripción de la petición.
- **Cuerpo (body):** parámetros de la petición.

En algunos casos, el **body** es opcional.

Estructura de una Respuesta

Las **respuestas** a las **peticiones** suelen tener el mismo formato, pero la **línea de inicio** se convierte en una **línea de estado**.

- **Línea de estado:** código de estado y mensaje.
- **Cabeceras (headers):** descripción de la respuesta.
- **Cuerpo (body):** retorno de la petición.

La **línea de estado** informa de forma **breve** el **resultado** de la **petición**.

Definición de Operaciones con FastAPI

Operaciones en FastAPI

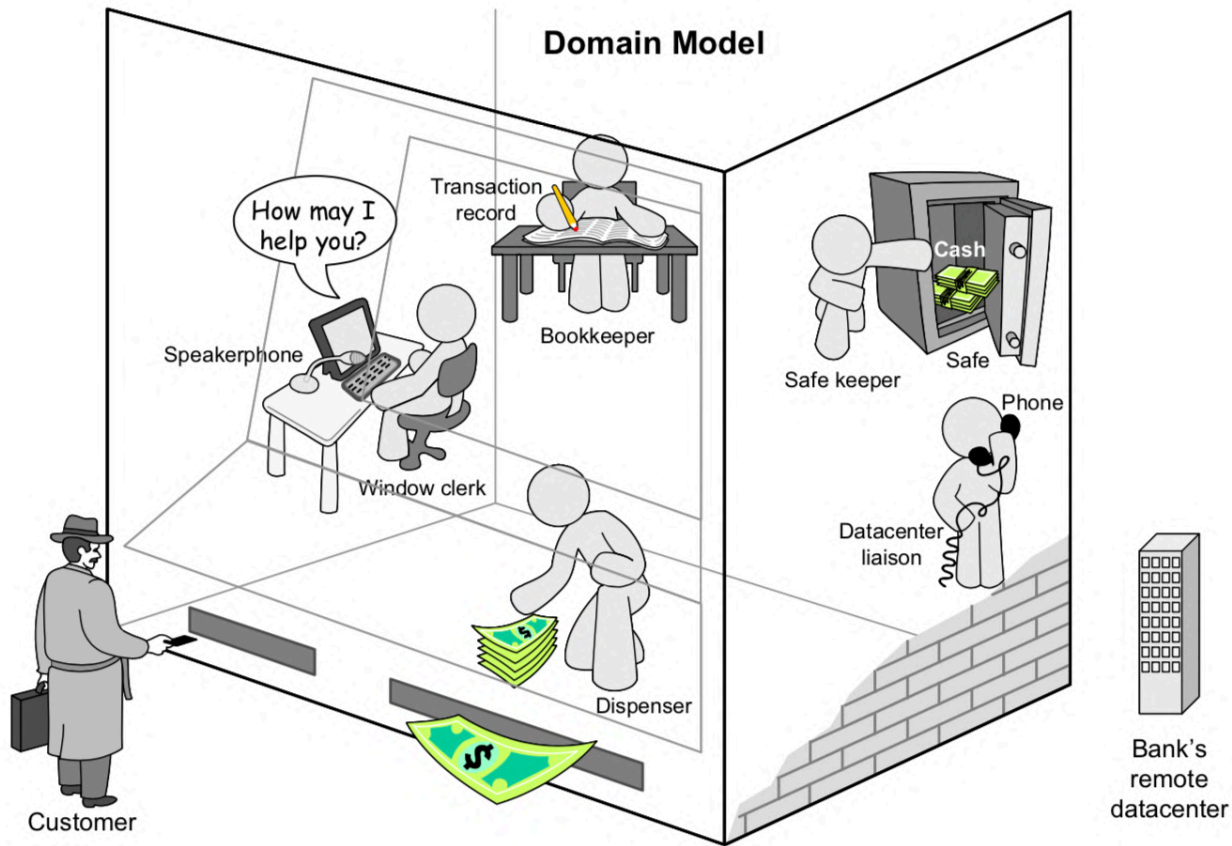
Para definir una **operación** en **FastAPI** se debe tener en cuenta:

- El **método** que se implementará.
- La **dirección** (URL) de la **operación**.
- Los **parámetros de entrada**: en la **dirección** o en el **body**.
- Las **posibles respuestas**: errores o recursos solicitados.

Los distintos **modelos de datos** evita la implementación de procesos de **validación**, **serialización** y **deserialización**.

Ejemplo

Software para un «ATM»



Operación – AUTENTICAR USUARIO

Permite identificar si un **usuario** hace parte del **sistema** y si su contraseña es correcta.

Tipo: POST

URL: .../user/auth/

Flujo de datos



Operación – CONSULTAR SALDO

Permite obtener el **saldo** del **usuario**:

Tipo: GET

URL:
.../user/balance/{usermane}

Flujo de datos



NOTA: **UserOut** contiene el **nombre de usuario** y el **saldo**.

Operación – RETIRAR DINERO

Permite a un **usuario** retirar **dinero**:

Tipo: PUT

URL:
.../user/transaction/

Flujo de datos

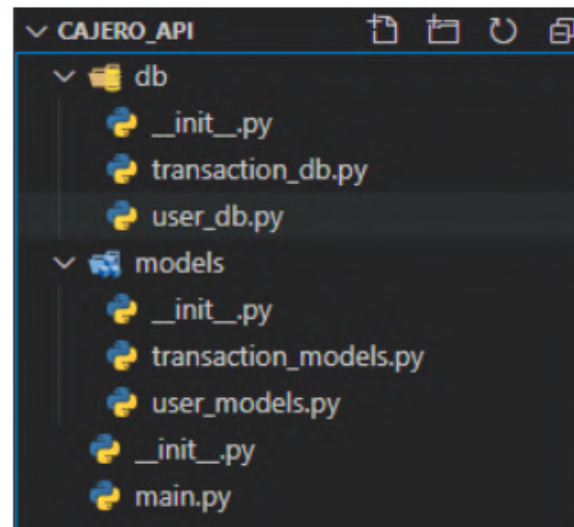


NOTA: diferencia entre **TransactionIn** y **TransactionOut**.

Implementación

Área de Trabajo: main.py

Estructura del proyecto realizada en la clase pasada:



Importando Modelos

Bloque 1: lo primero que se realiza es **importar** los **modelos** creados en la **clase anterior**:

```
from db.user_db import UserInDB
from db.user_db import update_user, get_user

from db.transaction_db import TransactionInDB
from db.transaction_db import save_transaction

from models.user_models import UserIn, UserOut

from models.transaction_models import TransactionIn,
TransactionOut
```

Importando Paquetes

Bloque 2: se importan algunos **paquetes** adicionales y se crea la **api** (un herramienta que agrupará las operaciones):

```
import datetime
from fastapi import FastAPI
from fastapi import HTTPException

api = FastAPI()
```

Implementando Operación `auth_user`

Bloque 3: se implementa la funcionalidad `auth_user`:

```
@api.post("/user/auth/")
async def auth_user(user_in: UserIn):
    user_in_db = get_user(user_in.username)
    if user_in_db == None:
        raise HTTPException(status_code=404,
                             detail="El usuario no existe")
    if user_in_db.password != user_in.password:
        return {"Autenticado": False}
    return {"Autenticado": True}
```

Implementando Operación `get_balance`

Bloque 4: se implementa la funcionalidad `get_balance`:

```
@api.get("/user/balance/{username}")
async def get_balance(username: str):
    user_in_db = get_user(username)
    if user_in_db == None:
        raise HTTPException(status_code=404,
                             detail="El usuario no existe")
    user_out = UserOut(**user_in_db.dict())
    return user_out
```

Implementando Operación `make_transaction`

Bloque 5: se implementa la funcionalidad `make_transaction`:

Parte 1:

```
@api.put("/user/transaction/")
async def make_transaction(transaction_in: TransactionIn):

    user_in_db = get_user(transaction_in.username)

    if user_in_db == None:
        raise HTTPException(status_code=404,
                             detail="El usuario no existe")

    if user_in_db.balance < transaction_in.value:
        raise HTTPException(status_code=400,
                             detail="Sin fondos suficientes")
```

Implementando Operación **make_transaction**

Bloque 5: se implementa la funcionalidad **make_transaction**:

Parte 2:

```
user_in_db.balance = user_in_db.balance - transaction_in.value  
update_user(user_in_db)
```

```
transaction_in_db = TransactionInDB(**transaction_in.dict(),  
                                     actual_balance = user_in_db.balance)  
transaction_in_db = save_transaction(transaction_in_db)
```

```
transaction_out = TransactionOut(**transaction_in_db.dict())  
return transaction_out
```

NOTA: ¡cuidado con las **identaciones**!

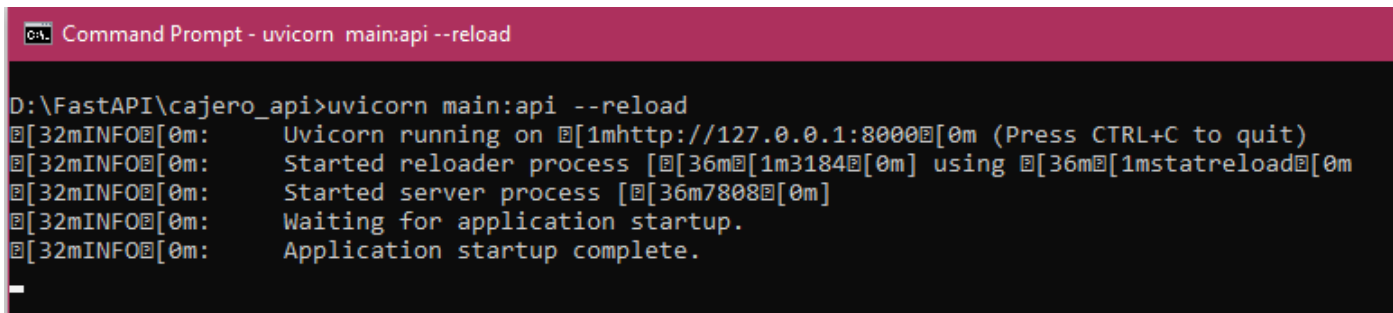
Ejecución

Ejecución del Componente

Para **ejecutar** el componente, **abrir una consola** y ubicarse en la **raíz del proyecto** (en la carpeta cajero_api) y ejecutar el siguiente **comando**:

uvicorn main:api --reload

Resultado



```
Command Prompt - uvicorn main:api --reload

D:\FastAPI\cajero_api>uvicorn main:api --reload
[32mINFO[0m:      Uvicorn running on [1mhttp://127.0.0.1:8000[0m (Press CTRL+C to quit)
[32mINFO[0m:      Started reloader process [36m[1m3184[0m] using [36m[1mstatreload[0m]
[32mINFO[0m:      Started server process [36m[1m7808[0m]
[32mINFO[0m:      Waiting for application startup.
[32mINFO[0m:      Application startup complete.
```


Prueba de la API

Uso del Cliente HTTP - Postman

Permite realizar **peticiones** a la **API** de manera **fácil** y **rápida**.



POSTMAN

Probando la Petición **auth_user**

Se debe ingresar la siguiente **información**:

Metodo: POST

URL: http://127.0.0.1:8000/user/auth/

Body (JSON): {

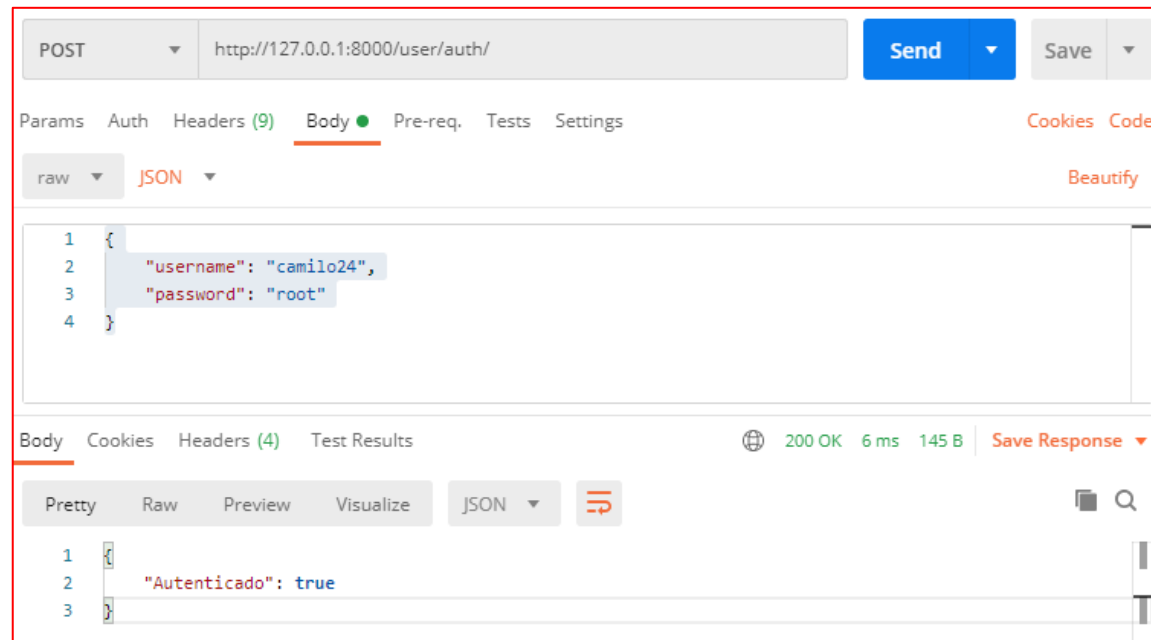
 "username": "camilo24",

 "password": "root"

}

Probando la Petición **auth_user**

En **POSTMAN** se podrá ver de la siguiente manera:



Probando la Petición **get_balance**

Se debe ingresar la siguiente **información**:

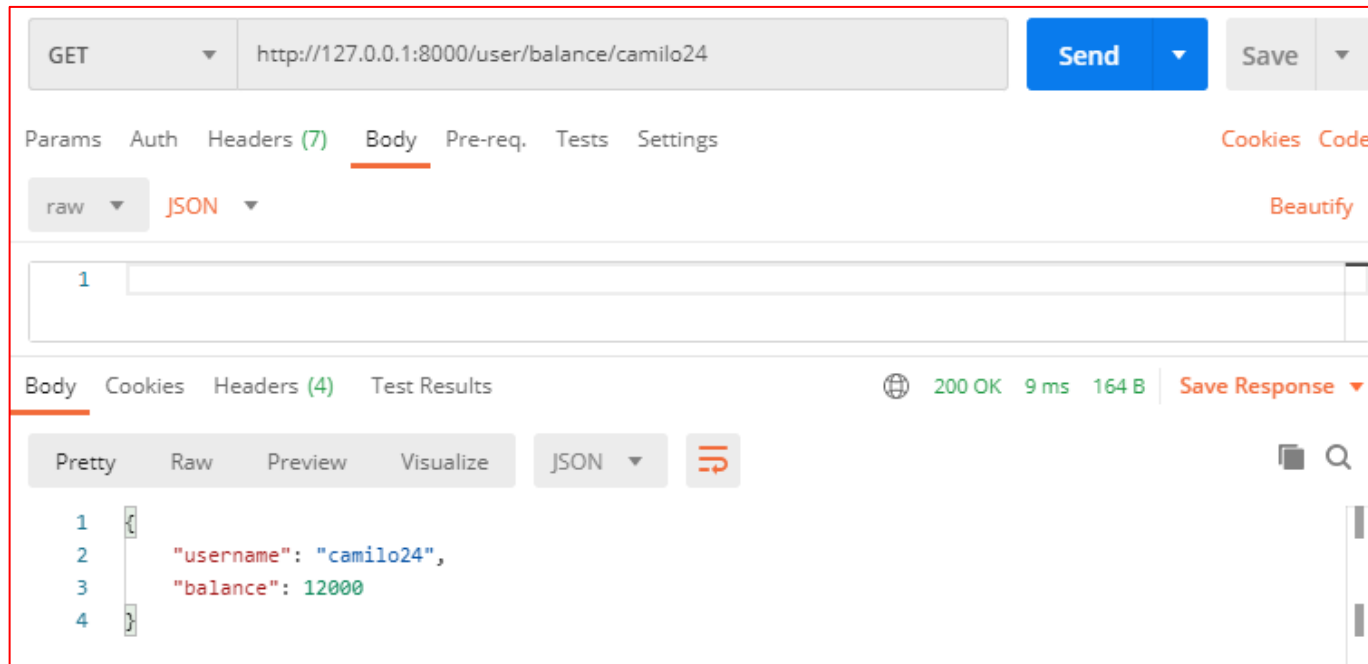
Metodo: GET

URL: `http://127.0.0.1:8000/user/balance/camilo24`

Body (JSON): {}

Probando la Petición `get_balance`

En **POSTMAN** se podrá ver de la siguiente manera:



Probando la Petición **make_transaction**

Se debe ingresar la siguiente **información**:

Metodo: PUT

URL: http://127.0.0.1:8000/user/transaction/

Body (JSON): {

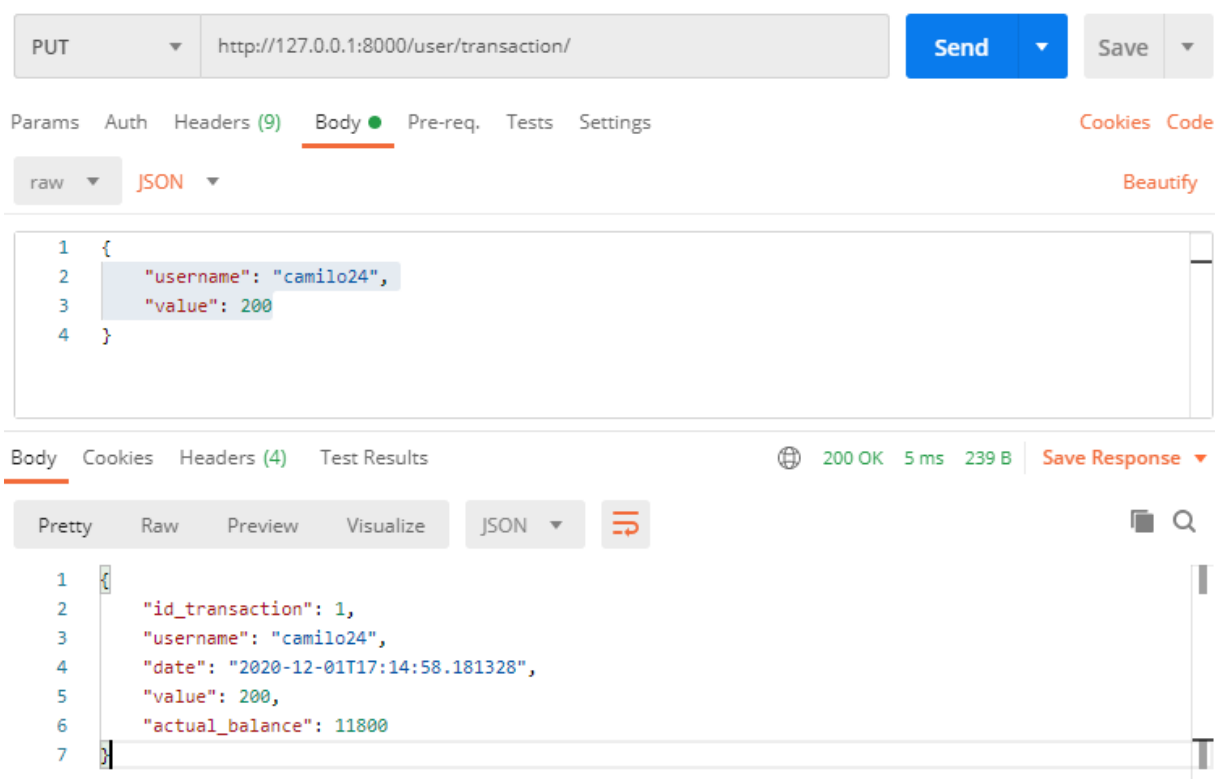
 "username": "camilo24",

 "value": 200

}

Probando la Petición **make_transaction**

En **POSTMAN** se podrá ver de la siguiente manera:



Documentación Generada

Documentación generada por FastAPI

FastAPI genera **documentación** de manera **automática**. Para consultarla, se debe **acceder a**:

- <http://127.0.0.1:8000/docs#/>
- <http://127.0.0.1:8000/redoc/>

(Dos **formatos** de **documentación diferentes**)

Referencias

- [FASTAPI] Comunidad FastAPI. (2020, noviembre). FastAPI. FastAPI.
<https://fastapi.tiangolo.com/>