CURSO DE FAST API CON PYTHON

de Sanjeev Thiyagarajan

[(curso de 19 horas en Youtuve)](https://www.youtube.com/watch?v=0sOvCWFmrtA)

Primero descargamos Python es instalamos en Windows especificando que se cree un path en las variables de entorno para hacerlo accesible globalmente.

Instalamos VisualStudio Code y la extensión de Python que permite Intellisense, etc.

Abrimos una carpeta para el proyecto y ahí creamos un archivo main.py. automáticamente tomará la versión de Python por defecto que tengamos, pero podríamos seleccionar otro intérprete mediante la paleta de comandos (ir a view en VS code), escribiendo Python select Interpreter. Podremos utilizar una que se encuentre en una determinada ruta.

Cuando usemos el framework de fastapi con una versión concreta y trabajemos con una versión de Python determinada, hemos de pensar que otros proyectos futuros pueden necesitar diferente versión de uno y otro, por lo que nos conviene construir un entorno virtual de desarrollo para cada proyecto. Eso lo conseguimos con con esta instrucción en la terminal, sobre la carpeta del proyecto:

**py -3 -m venv venv**

Esto genera una carpeta **venv** en la carpeta del proyecto

Le damos el path a Python.exe de la carpeta venv para fijar el intérprete de Python:

**.\venv\Scripts\python.exe**

Esto cambiará la versión añadiendo el nombre entre paréntesis (venv)

Para actualizar la consola y que tome las variables de nuestro entorno virtual con el intérprete nuevo, debemos ejecutar actívate.bat en la línea de consola. **venv\Scripts\activate.bat**

Instalamos en este entorno fastapi (ir al sitio web para ver la documentación -el tutorial-)

**pip instal fastapi[all]**

Incluye todas las dependencias opcionales, entre ellas el servidor **uvicorn**.

Con **pip freeze** podemos ver la lista de paquetes instalados.

Empezamos a codificar la api con fastapi

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

*async* *def* root():

    return {"message": "Welcome to my api"}

async no es necesario, es opcional

Hay una importación del paquete y una instancia abierta de fastAPI() en app

El decorador sobre la instancia fija un método http sobre un endpoint determinado, en este caso el path “/”

Activamos el servidor con:

**uvicorn main:app**

Si vamos a la máquina local en el navegador: <http://127.0.0.1:8000>, nos saldrá el texto devuelto por la función root(). El nombre de la función es irrelevante, podría haber sido cualquier otro, pero intentaremos que sea descriptivo y elocuente.

Si cambiamos el texto y volvemos a guardar, en el navegador seguirá lo mismo hasta que cerremos el servidor y lo volvamos a levantar.-

Para no tener que cerrarlo y volverlo a lanzar para ver en el navegador los cambios guardados del código, tenemos que usar **uvicorn main:app –reload**

**Si el archivo main ya no está bajo la carpeta de fastapi, sino bajo la carpeta app, entonces:**

**uvicorn app.main: app --reload**

Si hay igualdad de paths y operaciones para funciones diferentes, entonces a una petición sobre ese path ejecuta la primera que esté escrita en el código (obviamente esto no se suele programar así nunca)

Me falta añadir todo el uso de Postman para probar los endpoints

1:18:23 reanudar en el esquema de pandetyc para validación en la extracción de la información

Las rutas se nombran con plurales de manera standard en las APIS (por ejempolo “/posts” para todas las CRUD realizadas sobre los posts)

Tener cuidado con las rutas, para que estén un orden que permita encontrarlas sin confusión con parámetros

Regular la información sobre los errores: importar de fastapi el objeto Response, status y HTTPException. Luego en la función asociada al endpoint, incluimos como parámetro el **response: Response**

En caso de que queramos establecer una respuesta con un código determinado, haremos un juicio condicional introduciendo el objeto response adecuado y codificando el response.status = status.HTTP\_error\_XXX que elijamos de la lista de status.

Para que el código sea más legible mejor es lanzar una excepción en el condicional:

**raise HTTPException (status\_code=status.HTTP\_404\_NOT\_FOUND, detail\_f”post with id: {id} was not found”)**

Con esto bastaría y no hace falta incluir el objeto response en la *path operation function*

Para cambiar el código de status por defecto en una operación (se entiende la operación exitosa), incluiremos el parámetro status\_code = status.HTTP\_201\_create a continuación del endpoint, justo en la @app. En el caso de una operación de borrado, debería responder no con un 200 sino con un 204 que indica que no existe el contenido en la base de datos, y entonces no hay mensaje que responder, salvo

**return Response(status\_code\_ status.HTTP-204\_NO\_CONTENT)**

Con pydentic fijamos los modelos de los datos que queremos recibir del front, si no se ajusta lanzará un error. Por tanto, si vamos a recibir del front un objeto post crearemos un esquema de pydentic llamado Post para el post y especificaremos como parámetro de la path operation function el post: Post.

Fastapi construye la documentación automa´ticamente: damos el dominio-IP en el browser y añadimos /docs y accederemos a la documentación que construye FastAPI automáticamente, y podemos ver los ejemplos.

Tenemos documentación para Swagger UI pero también Redux añadiendo **/redoc**

**­­­­­­­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Hacemos una variación importante, ponemos todos los archivos incluido el main dentro de una carpeta llamada app, a la cual vamos a considerar un “paquete de Python” al añadirle dentro un archivo \_\_init\_\_.py que está vacío. Esto hace que la nueva carpeta se comporte como un Python package.

Esto obliga a cambiar la referencia del archivo main:

**uvicorn app.main:app –reload**

**BBDD de postgress instalación**

Creamos una base de datos de postgres antes de establecer la conexión con la base de datos, cuyo nombre necesitamos. Por defecto, la instalación de postgres origina automáticamente una BBDD llamada “postgres” por esta razón que hemos dicho.

Usamos pgAdmin 4 GUI para acceder a la BBDD de postgres y administrarla.

Los tipos de datos que admite Postgres tienen su correspondencia con Python casi completa:

Int,decimal,precisión ---------🡪 int, float

Varchar, text ------------🡪 string

Boolean 🡪 Boolean

Array 🡪 List

Hay que especificar la Primary Key para asegurar que la entrada es única. No permitirá duplicados de esa columna.

Se puede imponer el unique constraint en una columna para que no tenga duplicados, aunque no sea la Primary Key.

Not Null constraint: que no puede haber un vacío en una celda de esa columna, automáticamente no habrá un NULL

pgAdmin tiene su propia contraseña general para todas las BBDD de postgres que adminstremos.

Los datos de conexión habrá que pasárselos a pgAdmin 4, que detecta automáticamente las que haya y si queremos abrirla le metemos entonces la contraseña de la BBDD concreta.

Primer paso: crear instancia servidor con su contraseña. Luego creamos la BBDD que estimemos con el nombre que queramos. En ella veremos los Schemas y dentro las Tables. Pasaremos a crearlas. Sus nombres mejor en plural. Daremos nombre al atributo de cada columna, fijaremos el tipo de datos que recogerá y si tiene restricciones de algún tipo (podemos ver los data types posibles en la documentación de postgresql)

Hay un tipo llamado serial para crear automáticamente un número entero a modo de índice.

Se puede fijar otra condición: que tenga un valor por defecto

Es necesario (muy recomendable) grabar una columna de tiempo. Hay muchos tipos para ello.

“Timestamp With time zone” es el que el autor elige, y además le ordena que automáticamente lo cree la BBDD, diciendo que como ConstrainT tenga como default value NOW()

Con la Query tool podemos ejecutar peticiones a la BBDD. Todas deben terminar con un “;”

Cuando ordenamos por la columna tiempo, DESC indica primero los de más tiempo (más próximos al momento actual)

Para nuestra aplicación del back en Python con fastAPI necesitamos incluir los drivers de postgres para que pueda trabajar con una BBDD de postgres. Para ello elegimos psycopg- el PostgreSQL database adapter for Python, en concreto psycopg 2 (ir a psycopg documentation):

Pip install psycopg2

Usamos el try statement para la operación de conexión porque es una operación asíncrona.

Para manejar la conexión con los nombres de las columnas adecuados, debemos importar además RealDictCursor (from psycopg2.extras import REalDictCursor)

try:

conn = psycopg2.connect( host = “localhost”, database=’fastapi’, user=’postgres’, password= ‘password123’, cursor\_factory= RealDictCursor)

cursor=conn.cursor() #esto es lo que hará posible escribir sentencias SQL

except Exception as error:

print(“Connecting to database failed”)

print(“The Error was: “, error)

lo que hemos hecho de pasar la contraseña de la BBDD en el código está muy mal, hay que hacerlo usando variables de entorno para que el código sea seguro y dinámico.

Por otra parte la conexión a la BBDD puede fallar, queremos que esté en un loop while que lo intente cada ciertos segundos hasta que tenga éxito el try y entonces se salga de loop

Usando el objeto cursor podemos incluir entre triples comillas las sentencias SQL para manejar la BBDD. Primero se ejecuta la sentencia, luego se extrae del objeto cursor el resultado de la ejecución.

Pip freeze > requirements.txt genera el archivo con la info sobre packages necesarios de Python con su versión, para que se puedan regenerar allá donde se suban (git) sin subir las librerías a los repositorios.

Pip install -r requirements.txt genera esas librerías necesarias para el proyecto.