FUNDAMENTOS DE FASTAPI

Requisitos necesarios para trabajar:

1. Visual Studio Code
2. Python 3.10+
3. Pip: Instalador de librerías de Python

Python es un lenguaje que tiene su potencia en que se comunica con todo.

Su gran incremento de demanda se debe a Hacking y Big Data.

Tenemos múltiples ramas dentro de Python:

* Hacking
* Big Data: Gráficos y análisis de datos
* Jupyter: Big data
* Programas sencillos
* Django: Aplicaciones Web
* FastApi: Creación de Apis con lenguaje Python

A todos estos elementos se les llama Frameworks. Un Framework es una arquitectura basada en un lenguaje y que genera una aplicación.

Un Api es un servicio que permite representar datos y también nos permite interactuar con dichos datos, ya sea para modificar, insertar o recuperar datos.

Un servicio Api es tan popular porque representa datos en su mínima expresión, es decir, devuelve texto.

El texto que devuelve es un estándar, es decir, un texto representativo con un estándar que puede ser tanto JSON como XML.

JSON es una representación textual de un objeto (más popular)

XML es una representación textual con etiquetas de un objeto. (España)

Un servicio Api está formado por diversas características:

* Controller: Son los routers que permiten acceder a unos determinados Endpoints.
* EndPoints: Son los accesos URLS para las peticiones a cada elemento de un Controller.
* Models: Son las clases que puedo recibir/enviar entre las peticiones a cada Controller.
* Responses: Las respuestas pueden ser simples o pueden ser algo más complejas, depende de nuestro servicio.
* Services: Son los elementos de dónde recuperamos los datos a utilizar en el Api y sus controladores.

Además de estas características un servicio puede tener una serie de métodos dentro de los controladores:

* GET: Este método se utiliza cuando recuperamos datos de cualquier Api
* PUT: Este método se utiliza cuando queremos modificar algún elemento dentro del Api.
* DELETE: Este método se utiliza cuando queremos eliminar algo del Api.
* POST: Este método tiene dos vertientes:
  + Insertar información en el Api.
  + Enviar objetos y hacer con ellos lo que deseemos, por ejemplo, en la seguridad de los Apis se utiliza información POST.

<https://apicruddepartamentoscore.azurewebsites.net/index.html>

Departamentos es nuestro **Controller**

GET: Api/Departamentos es un **EndPoint**

Lo que estamos visualizando es un Api con documentación **Swagger**

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Configuración de nuestro entorno de trabajo.

En algún momento crearemos un FastApi en la carpeta en la que tengo el documento de Word.

Documentación y proyecto de FastApi

<https://github.com/serraguti/FASTAPI>

Para consumir un Api no necesitamos casi nada, simplemente código de Python, librería **request** y un entorno de trabajo de Python

Un Environment es un entorno de trabajo aislado dentro de Python y que se utiliza para agrupar todas las librerías y características de un Framework.

En muchas ocasiones, crear un entorno de trabajo es opcional, pero en algunos Frameworks como, FastApi, Django o Jupyter es imprescindible.

En el Environment es dónde podremos agregar las librerías.

Creamos una nueva carpeta para nuestras clases Python llamada **consumoapispython**

Dentro de VS Code instalamos la extensión de Python de Microsoft

Texto

El contenido generado por IA puede ser incorrecto.

Abrimos la carpeta con VS Code

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

Tendremos un entorno creado y podremos ejecutar tanto Frameworks como simples programas de Python

Seleccionamos una ventana de terminal de tipo command prompt

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

También podemos crearlo por comandos el entorno virtual desde el terminal

Crear un entorno nuevo

python -m venv venv

Activar el entorno

venv\Scripts\activate.bat

Verificar que tenemos el entorno activado

echo $VIRTUAL\_ENV

El primer paso para utilizar Apis es consumir dichos Apis.

Lo bueno de un Api es que se puede consumir con cualquier lenguaje:

* JavaScript, JQuery, React, Angular, Java, C#, Python

Lo primero que vamos a realizar es crear una carpeta y consumir el Api de departamentos con un GET.

Necesitamos la librería **requests** para consumir cualquier api desde Python

Mediante **pip** podemos instalar cualquier librería en nuestro entorno.

Un entorno está aislado, las librerías no se comparten con otros entornos.

Escribimos en la línea de comandos la siguiente instrucción:

**pip install requests**

Texto

El contenido generado por IA puede ser incorrecto.

El api que vamos a leer será leer un Empleado (por ejemplo)

Necesitamos una librería que ya está integrada dentro del entorno de Python llamada **json**, ya que los datos que vienen del servicio vienen en dicho formato.

Creamos un nuevo programa de Python llamado **python01getempleado.py**

Imagen que contiene Aplicación

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Necesitamos el Request de acceso.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

**PYTHON01GETEMPLEADO.PY**

**import requests**

**print("Leyendo un empleado")**

**apiUrl = "https://apiejemplos.azurewebsites.net/api/Empleados/7839"**

**response = requests.get(apiUrl)**

**#ESTO NOS DEVUELVE UNA RESPUESTA**

**#QUEREMOS EL JSON**

**jsonObject = response.json()**

**#ESTO ES UN OBJETO QUE PODEMOS ACCEDER A SUS PROPIEDADES**

**print("Apellido: " + jsonObject["apellido"])**

**print(jsonObject)**

**print("Fin de programa")**

El siguiente ejemplo será leer un conjunto de datos.

Vamos a leer todos los departamentos, creamos un nuevo programa llamado **python02alldepartamentos.py**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

**PYTHON02ALLDEPARTAMENTOS.PY**

**import requests**

**print("Leyendo muchos departamentos")**

**apiUrl = "https://apiejemplos.azurewebsites.net/api/Departamentos"**

**response = requests.get(apiUrl)**

**departamentos = response.json()**

**#RECORREMOS TODOS LOS DEPARTAMENTOS**

**for row in departamentos:**

**numero = row["idDepartamento"]**

**nombre = row["nombre"]**

**localidad = row["localidad"]**

**print(*f*"Numero: {numero}, Nombre: {nombre}, Localidad {localidad}")**

**print("Fin de programa")**

El siguiente paso es centrarnos en FASTAPI

Requisitos necesarios:

1. Environment
2. Instalar con pip FastApi
3. Instalar uvicorn o cualquier server para ejecutar el Api
4. Podemos instalar más librerías, pero con estas por ahora nos bastarán

**Nota:** Si instalamos fastapi standard ya instala también una serie de características para utilizar FastApi como, por ejemplo: **uvicorn**

Qué características vamos a encontrar:

1. **Tipado de datos**: FastApi utiliza en el lenguaje Python el tipado de datos, elemento que no es necesario para nada en Python Standard



1. Librerías opcionales de parámetros: En los métodos de los Api, como hemos visto, tenemos la posibilidad de enviar información y recibirla. Para poder enviar información, además de indicar el tipo de información, debemos indicar si dicha información será obligatoria o no para nuestro método.

FastApi ya viene con la documentación generada basada en OpenApi:

1. **Swagger:** Documentación estándar que utilizan multitud de servicios en la nube: Java, Net Core (8.0), FastApi
2. **Redoc:** Otro tipo de documentación con otro diseño.

Necesitamos una clase principal que, normalmente, se denomina **main.py**.

Dentro de dicha clase es dónde estarán nuestros métodos EndPoints.

Por defecto, solamente tendremos un Controller llamado **Default**, aunque por supuesto, crearemos nuestros propios controllers.

Existirá una variable llamada como lo deseemos (normalmente **app**) que será de tipo **FastApi** y la que contendrá los métodos GET, POST, PUT o DELETE.

Dicha variable generará también los ENDPOINTS para cada método.

Comenzamos creando una nueva carpeta llamada **fastapi**

Abrimos la carpeta raíz con VS Code.

Creamos un nuevo Entorno dentro de la carpeta Fast Api

Abrimos Terminal (Command Prompt) y escribimos la siguiente instrucción para instalar FastApi

**pip install "fastapi[standard]"**

Texto

El contenido generado por IA puede ser incorrecto.

Podremos tener múltiples ficheros de acción, pero solamente uno será la raíz de ejecución de FastApi: **main.py**

Creamos un nuevo fichero llamado **main.py**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Dentro de nuestro programa **main** podemos tener todos los métodos **def()** que deseemos.

Pero, **app** decidirá que métodos vamos a lanzar hacia el Api en la documentación de Swagger y con un EndPoint.

Mediante la siguiente sintaxis crearemos EndPoints y su tipo de METODO (GET, POST, PUT, DELETE)

**@app.get**

**@app.post**

**@app.delete**

**@app.put**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Una vez que tenemos el método, simplemente probamos dicho método en la ejecución del Api

Para ejecutar el Api tenemos varias formas:

1. **Uvicorn:**

**uvicorn main:app**

1. **FastApi server:** Indicamos el Servidor a ejecutar

**fastapi run server main.py**

1. Lanzar directamente el servidor que tengamos instalado (uvicorn) y podremos visualizar los cambios al vuelo

**fastapi dev main.py**

1. **FastApi production:** Lanza el servicio para producción y no tendremos ni Debug ni visualizar al vuelo

**fastapi run main.py**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Tendremos una URL de acceso y podemos probar nuestro Servicio:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Esto que hemos hecho está mal.

Un servicio Api jamás devolverá un triste texto, la respuesta siempre tendrá un formato accesible.

Dicho formato será XML o JSON.

Siempre tiene que devolver un objeto KEY:VALUE

Todos los cambios son al vuelo, incluidos los errores que tengamos…

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Por supuesto, podemos crear más EndPoints, simplemente creando nuevas rutas y nuevos métodos.

Vamos a crear otro método

**Lo único que tengo que preservar es que los EndPoints no estén repetidos.**

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Otra gran ventaja es que ya tenemos la documentación montada al utilizar FastApi.

Tenemos dos tipos de documentación:

* Swagger: /docs

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

* Redoc: /redoc (Todavía está verde…)

Interfaz de usuario gráfica, Aplicación, Teams

El contenido generado por IA puede ser incorrecto.

Debemos aprender a recibir información en los métodos GET.

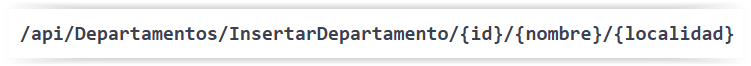
No será lo mismo recibir información en un GET que en un POST o un PUT.

La información de los GET se representa mediante URL.

La información de los POST/PUT se representa mediante URL o mediante Object.

En cualquier petición GET tenemos dos formas de recibir los datos:

1. **Mapping:** Un Mapping indica que recibiremos los parámetros en orden y en la posición que generamos dentro de un EndPoint. Se representa mediante {parametro} cada parámetro que necesitamos y Swagger es capaz de documentarlo



1. **QueryString:** Recibe los parámetros mediante la URL clásica y no importa el orden de los parámetros. También podemos hacer que los parámetros sean opcionales

Swagger no documenta QueryString

La URL de acceso es un standard:

<https://servicioapi.com?param1=valor1&param2=valor2>

Independientemente a esto, los parámetros recibidos deben cumplir dos normas:

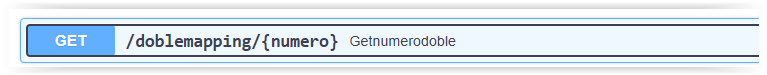
* Mismo nombre que en la declaración del **Routing**
* Los tipos de datos recibidos deben tener un tipado, tal y como os he expuesto antes.

Comenzamos con Mapping y vamos a crear un EndPoint que recibirá un número y nos devolverá el doble de dicho número.

**MAIN.PY**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.



Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

A continuación, vamos a crear un método que recibirá un nombre por QueryString y nos devolverá un Saludo.

No se declara dentro del Routing

Interfaz de usuario gráfica, Texto, Aplicación, Word

El contenido generado por IA puede ser incorrecto.

No aparece la declaración de nombre en el EndPoint

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

La variable de la URL **nombre** no es opcional, debe llamarse así.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Por supuesto, podemos recibir más de un parámetro.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

También podemos hacer que los parámetros de **QueryString** sean opcionales.

Para que los parámetros puedan ser opcionales, necesitamos de una nueva librería llamada **typing** y que contiene una clase llamada **Union**

Mediante **Union,** podemos indicar si el QueryString no lo recibe, que no suceda nada.

Modificamos el método **saludito()** y recibimos la afición de forma opcional.

Imagen que contiene Icono

El contenido generado por IA puede ser incorrecto.

En la definición del parámetro en el método le indicaremos que será **str** o None.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Por supuesto, podemos combinar tanto elementos QueryString como Mapping a la vez en un método.

Solamente tenemos que tener en cuenta que los parámetros del QueryString serán los últimos.

Primero van los parámetros del Mapping.

Vamos a probarlo dentro del método del Doble del Mapping.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**OBJETOS COMPUESTOS**

Estamos devolviendo objetos simples, es decir, un mensaje de doble, otra key con un saludo o un mensaje.

Si necesitamos devolver múltiples elementos, por ejemplo, un conjunto de Nombre o un conjunto de números. ¿Cómo lo hacemos?

Vamos a visualizar qué sucede devolviendo directamente un conjunto de números.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Esto es un JSON??

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Modificamos el método y devolvemos un Key/Value

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

El resultado es un JSON???

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Necesitamos poder representar los datos de cada número mediante un Key/Value utilizando un Diccionario.

Debemos devolver un diccionario de objetos.

En el momento de trabajar dentro de FastApi, existen varios tipos de datos, dependiendo de Simples o compuesto:

1. Simples: String, int, byte, float
2. Compuestos: Diccionarios (Key/Value), Listas []

Cualquiera de ellos son internamente “traducidos” dentro del Framework para poder devolverlos.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Estamos trabajando con datos simples, es decir, números que convertimos internamente a Diccionarios Key/Value pero, dependiendo de la complejidad del diccionario, puede que nos muestre caracteres de escape, es decir, que interprete nombres o textos que NO queremos.

Vamos a probar a devolver Textos en lugar de números

Siempre, debemos de indicar, en los objetos que devolvamos que tendrán una codificación.

Existen codificaciones que son automáticas, cuando tipamos el objeto explícitamente.

En algunas ocasiones, cuando recuperamos de files o servicios o bbdd debemos codificar de forma explicita nuestros objetos.

Tenemos una clase que nos permite codificar los objetos de forma explicita:

**from fastapi.encoders import jsonable\_encoder**

Se encarga de eliminar los caracteres especiales, como pueden ser las dobles comillas o elementos propios del código del origen que estemos leyendo.

Debe aplicarse a los objetos que creamos dinámicamente como, por ejemplo, el objeto **elemento** de número o de nombres.

**@app.get("/nombres")**

***def* dameNombres():**

**listaNombres = ["Adrian", "Lucia", "Manuel", "Antonia"]**

**#Vamos a crear un objeto diccionario para cada número**

**# Cada objeto que tengamos, lo almacenaremos en la colección de salida**

**salida = []**

**for name in listaNombres:**

**#Creamos un diccionario para cada numero**

**elemento = {"name": name}**

**codificado = jsonable\_encoder(elemento)**

**salida.append(codificado)**

**return {"nombres": salida}**

Al estar hablando de codificadores, dichos elementos se utilizan al devolver datos o al recibir datos en el servicio.

Estamos devolviendo datos y siempre deben ir codificados.

Pero, también podemos recibir datos en el servicio.

Dichos datos deben ser recibidos codificados para el Framework de FastApi y no podemos hacerlo con **jsonable\_encoder()** ya que es un método.

Debemos definir una clase y poder recibir dicha clase en el método PUT o POST.

Para poder recibir un objeto es imprescindible dos características de la definición:

1. Clase de Python con las propiedades tipadas
2. Debe heredar de **BaseModel**

BaseModel es una clase que viene de la librería **pydantic** que internamente codifica y permite recibir un objeto en los métodos.

Definición de una clase en el lenguaje Python

class NombreClase():

propiedad1 = valor

propiedad2 = valor

Herencia en Python:

class NombreClase(ClaseParent):

propiedad1 = valor

propiedad2 = valor

Vamos a emular un método PUT que recibirá un objeto y devolvemos el objeto en JSON.

Todo lo seguimos haciendo en el Main por ahora.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Hasta ahora tenemos el proyecto con un montón de elementos mezclados, tanto el elemento Class como el elemento Controller.

Pongamos que, a continuación, vamos a leer el contenido de algo para nuestro Servicio.

Vamos a leer un documento JSON para nuestro Servicio Api, como si de un origen de datos real se tratase.

Por ahora, lo vamos a poner en el mismo lugar, es decir, mezclando.

Descargamos los files y los ponemos en una carpeta llamada **data**

Ahora mismo esto nos va a servir para generar “datos” en nuestra App.

Dichos datos, con el concepto de Codificación, debemos codificarlos dentro de nuestro FastApi.

La primera forma que vamos a utilizar es “manualmente”, sin utilizar un Model.

La segunda forma, intentamos hacerlo con un Model.

Aunque ahora mismo está horrible, lo haremos en el **main.py**

Para leer los datos de JSON tenemos una librería nativa de Python llamada **json**

Dicha librería contiene un método llamado **load()** que permite leer de un fichero.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Aunque estemos viendo perfectamente los jugadores, no estamos siguiendo el patrón de FastApi, lo que quiere decir que NO siempre funcionará, debido a que internamente tiene que devolver formatos “reconocibles” con codificación.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Debemos aplicar siempre un codificador.

La primera forma es manual, leemos todo el contenido de nuestro JSON de jugadores y los codificamos con **jsonable\_encoder**

Modificamos nuestro código para recuperar todos los jugadores e ir codificando uno a uno a una colección de salida.

**#Recuperamos la librería de objetos JSON**

**import json**

**app = FastAPI()**

**@app.get("/players")**

***def* readPlayers():**

**#Existe un metodo llamado open() que nos permite**

**#abrir ficheros**

**info = open("./data/jugadores.json")**

**json\_data = json.load(info)**

**#Creamos una lista/colección de Salida**

**salida = []**

**#Recorremos todos los jugadores**

**for row in json\_data["jugadores"]:**

**#codificamos el objeto**

**item = jsonable\_encoder(row)**

**salida.append(item)**

**return {"players": salida}**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Otra forma de codificar es mediante **BaseModel**, crearnos nuestros propios Models dentro de la aplicación.

Con BaseModel necesitamos crearnos una clase.

Dentro del mismo **main.py** creamos una clase llamada **Player** que herede de **BaseModel**

**PLAYER**

Texto

El contenido generado por IA puede ser incorrecto.

Modificamos el código para devolver una colección/lista del tipo Player

Como hereda de BaseModel, **pydantic** obliga a pasar los argumentos del objeto en el momento de Instanciar la clase.

**@app.get("/players")**

***def* readPlayers():**

**#Existe un metodo llamado open() que nos permite**

**#abrir ficheros**

**info = open("./data/jugadores.json")**

**json\_data = json.load(info)**

**#Creamos una lista/colección de Salida**

**salida = []**

**#Recorremos todos los jugadores**

**for row in json\_data["jugadores"]:**

**player: Player = Player()**

**player.*id* = *int*(row["numero"])**

**player.*nombre* = row["nombre"]**

**player.*posicion* = row["posicion"]**

**player.*edad* = *int*(row["edad"])**

**salida.append(player)**

**return {"players": salida}**

El problema es que, a medida que vayamos creciendo, estamos poniendo el código todo mezclado.

Si tenemos Modelos (Player) deberíamos tener un modelo player en una carpeta separada: **models**

Si tenemos métodos que nos devuelvan los datos, deberíamos tener una carpeta para acceder a dichos datos: **services**

Vamos a limpiar un poco nuestro proyecto y empezar a organizar.

Sobre el proyecto, creamos una nueva carpeta llamada **models**

Sobre Models, creamos una nueva clase llamado **Player.py**

**PLAYER.PY**

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

**MAIN.PY**



**@app.get("/players")**

***def* readPlayers():**

**#Existe un metodo llamado open() que nos permite**

**#abrir ficheros**

**info = open("./data/jugadores.json")**

**json\_data = json.load(info)**

**#Creamos una lista/colección de Salida**

**salida = []**

**#Recorremos todos los jugadores**

**for row in json\_data["jugadores"]:**

**player: Player = Player()**

**player.*id* = *int*(row["numero"])**

**player.*nombre* = row["nombre"]**

**player.*posicion* = row["posicion"]**

**player.*edad* = *int*(row["edad"])**

**salida.append(player)**

**return {"players": salida}**

Una vez que hemos separado las capas de los modelos, también deberíamos separar las peticiones a nuestros EndPoints de las acciones que realizan.

Que estemos leyendo un documento JSON o una base de datos, no implica que tengamos que hacer todo en el mismo sitio, abrir, leer el documento, insertar o la acción que deseemos.

La siguiente separación radica en utilizar los Servicios.

Un Servicio es una petición para poder realizar acciones sobre los datos.

Diagrama

El contenido generado por IA puede ser incorrecto.

Vamos a crear una nueva carpeta llamada **services** y que leerá los datos del JSON y ya nos los ofrecerá directamente para el **MAIN**

Dentro de Services creamos una clase llamada **ServiceJugadores.py**

Tenemos dos opciones.

Una clase o métodos.

En Python no es necesario tener una CLASE (Class) para poder realizar peticiones.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Esta clase NO necesita ni constructor ni necesita nada.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

La diferencia entre crearse una clase de métodos o una Class radica en qué queremos hacer con la clase y el objeto.

Por ejemplo, ahora mismo es una clase que lee un fichero, no necesita nada de nada.

Pero, si nuestra clase fuera una clase de BBDD, necesita de conexiones y algunas características extras

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

En este ejemplo vamos a crear una clase solo de métodos.

**SERVICEJUGADORES.PY**

**import json**

**from models.Player import Player**

***def* getPlayers():**

**info = open("./data/jugadores.json")**

**json\_data = json.load(info)**

**#Creamos una lista/colección de Salida**

**salida = []**

**#Recorremos todos los jugadores**

**for row in json\_data["jugadores"]:**

**player: Player = Player()**

**player.*id* = *int*(row["numero"])**

**player.*nombre* = row["nombre"]**

**player.*posicion* = row["posicion"]**

**player.*edad* = *int*(row["edad"])**

**salida.append(player)**

**return salida**

***def* findPlayer(*idPlayer*: *int*):**

**info = open("./data/jugadores.json")**

**json\_data = json.load(info)**

**#CODIGO PARA BUSCAR UN JUGADOR POR ID**

**for jugador in json\_data["jugadores"]:**

**if jugador["numero"] == idPlayer:**

**player: Player = Player()**

**player.*id* = *int*(jugador["numero"])**

**player.*nombre* = jugador["nombre"]**

**player.*posicion* = jugador["posicion"]**

**player.*edad* = *int*(jugador["edad"])**

**return player**

Creamos otro EndPoint dentro de Main.py para recibir el ID por Routing.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Los controladores nos permiten separar los EndPoints en un conjunto de características, por ejemplo, la lógica de los Jugadores de la lógica de los Empleados o los Departamentos.

Un controlador es una clase que debe estar definida mediante una clase llamada **APIRouter**

Mediante **APIRouter** se crea un controlador con sus propios métodos EndPoints, ya sean GET o POST o cualquier otro.

Se debe crear una clase con APIRouter. Dicho objeto APIRouter será el que genere los EndPoints.

**CLASE CONTROLLER**

router = APIRouter()

router.get(“EndPoint”):

def metodo():

return algo

Posteriormente, es la aplicación Principal (**main.py**) es quién se encarga de registrar los diferentes Routers/Controllers

**MAIN.PY**

app = FastAPI()

app.include\_router(CLASE CONTROLLER)

Vamos a crear un ejemplo que NO tiene nada que ver con jugadores.

Devolvemos un método, lo montamos y probamos.

Posteriormente, montamos la lógica con Jugadores.

Creamos una nueva carpeta llamada **controllers** y creamos una clase llamada **DatosController.py**

**DATOSCONTROLLER**

**from fastapi import APIRouter**

**#creamos un objeto que lo utilizaremos posteriormente**

**#dentro de nuestra clase main**

**router = APIRouter()**

**#este objeto crea los metodos**

**@router.get("/pruebas")**

***def* misPruebas():**

**return {"data", "haciendo pruebas Controller"}**

**@router.get("/pruebas/{dato}")**

***def* pruebasDato(*dato*: *str*):**

**return {"data", "Recibiendo dato en controller " + dato}**

**MAIN.PY**

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Podremos visualizar cómo ha organizado todo correctamente

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

El siguiente paso es llevar la lógica de Jugadores a un Controller.

En el controller necesitamos **ServiceJugadores** y los **Models**

El **Main** ya no necesita los jugadores

Sobre **controllers** creamos una nueva clase llamada **JugadoresController.py**

**JUGADORESCONTROLLER.PY**

**from fastapi import APIRouter**

**import services.*ServiceJugadores* as servicio**

**from models.Player import Player**

**router = APIRouter()**

**@router.get("/players")**

***def* readPlayers():**

**players = servicio.getPlayers()**

**return {"players": players}**

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

***def* findPlayer(*id*: *int*):**

**player = servicio.findPlayer(id)**

**return player**

Agregamos nuestro nuevo Controller sobre **main.py**

**MAIN.PY**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Y podremos visualizar el resultado

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

**RESPUESTA WRAPPER DE DATOS**

Todos los Apis son iguales, no existe diferencia.

Nuestro Api devuelve diferentes respuestas dependiendo del método.

En la arquitectura de cualquier Api, las respuestas deberían ser IGUALEScon independencia del método de la llamada, es decir, si nos fijamos en algún método nuestro:

Texto

El contenido generado por IA puede ser incorrecto.

Por un lado devolvemos **player** y por otro devolvemos **players**

Esto NO está mal, pero si devolvemos siempre lo mismo, el desarrollador que consuma nuestro Api ya tiene un patrón con independencia de lo que estemos realizando.

A esta arquitectura se le denomina Patrón de respuesta Api o Response Wrapper

Existen otras características dentro de los Apis que NO tienen que ver con Apis, son herramientas, por ejemplo, si tenemos una clase que se encarga de cifrar los datos en las peticiones. La clase Response Wrapper tampoco tiene nada que ver con un Api, es una herramienta.

A estas herramientas se les denomina **utils** dentro de FastApi

Vamos a crear un carpeta llamada **utils** y una clase llamada **responsewrapper.py**

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Utilizamos dicho Wrapper en las clases Controllers

**JUGADORESCONTROLLER.PY**

Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

Con esta respuesta personalizada ya podemos decidir qué hacer con nuestras peticiones.

Por ejemplo, he creado un método FIND que busca un jugador por su ID

¿Qué sucede si no encuentra el jugador?

Tenemos dos opciones:

1. Devolver un response Wrapper formado con el error y lo que deseemos

**SERVICEJUGADORES**

Texto

El contenido generado por IA puede ser incorrecto.

**CONTROLLER**

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Veremos la respuesta correcta

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

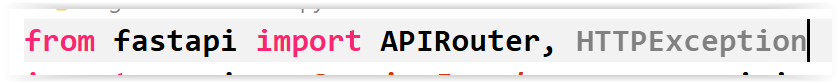
El contenido generado por IA puede ser incorrecto.

1. Lanzar una excepción HTTP dentro de FastApi

Dentro de FastApi tenemos una clase llamada **HttpException** que nos permite lanzar errores HTTP en nuestro Server

Dichos errores son lanzados mediante un **status\_code** y **detail**

**CONTROLLER**



Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**DEPENDENCIAS**

Una dependencia dentro de FastApi indica que un objeto dependerá de otro.

**Que para se ejecute un código es necesario utilizar otro código.**

Para ello, utiliza una clase que se llama **Depends** y que indica que un método dependerá de otro método.

Las dependencias pueden ser de tres formas:

1. Dependencia de método: Un método o varios dependen de una lógica
2. Dependencia de clase: Un método o varios dependen de una lógica de una clase
3. Dependencia de Aplicación: La aplicación necesita ejecutar “algo” en sus peticiones para ejecutarlas

Para esta lógica vamos a realizar dos métodos simples.

Por ejemplo, filtraremos jugadores por **Edad** y por **Posicion**

**SERVICIOJUGADORES.PY**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

**CONTROLLER**

Texto

El contenido generado por IA puede ser incorrecto.

Además de filtrar, me gustaría decidir un inicio y un final para mostrar los jugadores, es decir, quiero jugadores desde el 1 hasta el 5 de los datos que me devuelvas.

Modificamos los métodos del Servicio

***def* searchPlayersAge(*edad*: *int*, *inicio*: *int*, *fin*: *int*):**

**players = getPlayers()**

**filtro = *list*(filter(*lambda* *x*: x.*edad* > edad, players))**

**datos = []**

**for i in range(inicio, fin + 1):**

**if (i >= len(filtro)):**

**break**

**datos.append(filtro[i])**

**return datos**

***def* searchPlayersPosition(*posicion*: *str*, *inicio*: *int*, *fin*: *int*):**

**players = getPlayers()**

**filtro = *list*(filter(*lambda* *x*: x.*posicion*.lower() == posicion.lower(), players))**

**datos = []**

**for i in range(inicio, fin + 1):**

**if (i >= len(filtro)):**

**break**

**datos.append(filtro[i])**

**return datos**

***def* filterPlayers(*inicio*: *int*, *fin*: *int*):**

**players = getPlayers()**

**datos = []**

**for i in range(inicio, fin + 1):**

**if (i >= len(players)):**

**break**

**datos.append(players[i])**

**return datos**

**CONTROLLER**

Texto

El contenido generado por IA puede ser incorrecto.

Nuestros métodos tienen elementos y parámetros en común.

Mediante dependencias podemos hacer que TODOS lean y necesiten de inicio y de fin.

De esta forma, estamos indicando que los métodos necesitará de inicio y de fin para trabajar.

Si creamos 20 métodos más que tengan inicio y fin, tendremos que estar “pensando” cómo podemos hacer de nuevo lo mismo, control c + control v

Mediante una dependencia, lo que permite es indicar algo que van a necesitar los métodos y yo lo que hago es decir a los métodos que vayan a por ello antes de ser utilizados.



**CONTROLLER**

**from fastapi import APIRouter, HTTPException, Depends**

**import services.*ServiceJugadores* as servicio**

**from models.Player import Player**

**from utils.responsewrapper import apiResponse**

**router = APIRouter()**

***def* parametrosDepends(*inicio*:*int* = 0, *fin*: *int* = 100):**

**#aqui hacemos todo lo necesario con los parametros**

**return {"inicio": inicio, "fin": fin}**

**@router.get("/players")**

***def* readPlayers():**

**players = servicio.getPlayers()**

**return apiResponse(*data*=players, *message*="All players OK")**

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

***def* findPlayer(*id*: *int*):**

**player = servicio.findPlayer(id)**

**if (player is None):**

**raise HTTPException(*status\_code*=404,*detail*=*f*"Player not found {id}")**

**return apiResponse(*data*=player)**

**@router.get("/filteredad/{edad}")**

***def* jugadoresEdad(*edad*: *int*, *commons*: *dict* = Depends(parametrosDepends)):**

**players = servicio.searchPlayersAge(edad, commons["inicio"], commons["fin"])**

**return apiResponse(*data*=players)**

**@router.get("/filterposicion/{posicion}")**

***def* jugadoresEdad(*posicion*: *str*, *commons*: *dict* = Depends(parametrosDepends)):**

**players = servicio.searchPlayersPosition(posicion, commons["inicio"], commons["fin"])**

**return apiResponse(*data*=players)**

**@router.get("/filterplayers")**

***def* filterPlayers(*commons*: *dict* = Depends(parametrosDepends)):**

**players = servicio.filterPlayers(commons["inicio"], commons["fin"])**

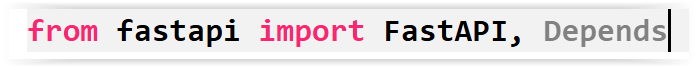
**return apiResponse(*data*=players)**

La segunda forma es lo mismo, pero en lugar de utilizar métodos, utiliza una clase.

La tercera forma, para mí, si que puede llegar a ser útil en algún momento en nuestros desarrollos.

Que nuestro **FastApi** dependa de algo para lanzar la acción en **todos** los métodos.

Para probarlo, tan sencillo como crear dos métodos en el **main.py**



Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Se realiza la llamada con cualquier método

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

También podemos recibir parámetros en la dependencia lo que quiere decir que TODOS los métodos deben enviar un parámetro a la fuerza.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Vamos a realizar una aplicación con todos los métodos y conceptos vistos.

Dicha aplicación llamará a una base de datos SQL Server en la nube

Haremos un POST, PUT, DELETE y GET

Creamos una nueva carpeta llamada **FastapiSeries**

Instalamos FastApi

**pip install fastapi[standard]**

Vamos a consumir SQL Server

Instalamos **pyodbc**

**pip install pyodbc**

MySql:

**pip install mysql.connector**

**MODEL SERIE**

Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

**SERVICESERIES**

**import pyodbc**

**from models.serie import Serie**

***class* ServiceSeries():**

**servidor = "sqltajamarpgs.database.windows.net"**

**bbdd = "AZURETAJAMAR"**

**usuario = "adminsql"**

**password = "Admin123"**

***def* \_\_init\_\_(*self*):**

***self*.*connectionString* = "DRIVER={ODBC Driver 17 for SQL Server}; SERVER=" + *self*.*servidor* + "; DATABASE=" + *self*.*bbdd* + "; UID=" + *self*.*usuario* + "; PWD=" + *self*.*password***

***self*.*connection* = pyodbc.connect(*self*.*connectionString*)**

***def* getSeries(*self*):**

**sql = "select \* from SERIES"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql)**

**data = []**

**for row in cursor:**

**serie: Serie = Serie()**

**serie.*id* = row.IDSERIE**

**serie.*nombre* = row.SERIE**

**serie.*imagen* = row.IMAGEN**

**serie.*year* = row.ANYO**

**data.append(serie)**

**cursor.close()**

**return data**

***def* findSerie(*self*, *idSerie*: *int*):**

**sql = "select \* from SERIES where IDSERIE=?"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql, (idSerie))**

**row = cursor.fetchone()**

**if (not row):**

**cursor.close()**

**return None**

**else:**

**serie: Serie = Serie()**

**serie.*id* = row.IDSERIE**

**serie.*nombre* = row.SERIE**

**serie.*imagen* = row.IMAGEN**

**serie.*year* = row.ANYO**

**cursor.close()**

**return serie**

***def* insertSerie(*self*, *idserie*:*int*, *nombre*: *str*, *imagen*: *str*, *year*: *int*):**

**sql = "insert into SERIES values (?,?,?,?)"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql, (idserie, nombre, imagen, year))**

**cursor.commit()**

**cursor.close()**

***def* updateSerie(*self*, *idserie*:*int*, *nombre*: *str*, *imagen*: *str*, *year*: *int*):**

**sql = "update SERIES set SERIE=?, IMAGEN=?, ANYO=? where IDSERIE=?"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql, (nombre, imagen, year, idserie))**

**cursor.commit()**

**cursor.close()**

***def* deleteSerie(*self*, *idserie*:*int*):**

**sql = "delete from SERIES where IDSERIE=?"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql, (idserie))**

**cursor.commit()**

**cursor.close()**

**SERVICESERIESMYSQL**

**import mysql.*connector***

**from models.serie import Serie**

***class* ServiceSeriesMySql():**

***def* \_\_init\_\_(*self*):**

***self*.*connection* = mysql.*connector*.connect(*host* = "servidorpythonfastapi.mysql.database.azure.com", *user* = "adminsql", *password* = "MySql123", *database* = "series")**

***def* getSeries(*self*):**

**sql = "select \* from SERIES"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql)**

**data = []**

**for row in cursor:**

**serie: Serie = Serie()**

**serie.*id* = row[0]**

**serie.*nombre* = row[1]**

**serie.*imagen* = row[2]**

**serie.*year* = row[3]**

**data.append(serie)**

**cursor.close()**

**return data**

***def* findSerie(*self*, *idSerie*: *int*):**

**sql = "select \* from SERIES where IDSERIE=?"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql, (idSerie))**

**row = cursor.fetchone()**

**if (not row):**

**cursor.close()**

**return None**

**else:**

**serie: Serie = Serie()**

**serie.*id* = row[0]**

**serie.*nombre* = row[1]**

**serie.*imagen* = row[2]**

**serie.*year* = row[3]**

**cursor.close()**

**return serie**

***def* insertSerie(*self*, *idserie*:*int*, *nombre*: *str*, *imagen*: *str*, *year*: *int*):**

**sql = "insert into SERIES values (?,?,?,?)"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql, (idserie, nombre, imagen, year))**

**cursor.commit()**

**cursor.close()**

***def* updateSerie(*self*, *idserie*:*int*, *nombre*: *str*, *imagen*: *str*, *year*: *int*):**

**sql = "update SERIES set SERIE=?, IMAGEN=?, ANYO=? where IDSERIE=?"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql, (nombre, imagen, year, idserie))**

**cursor.commit()**

**cursor.close()**

***def* deleteSerie(*self*, *idserie*:*int*):**

**sql = "delete from SERIES where IDSERIE=?"**

**cursor = *self*.*connection*.cursor()**

**cursor.execute(sql, (idserie))**

**cursor.commit()**

**cursor.close()**

**CONTROLLER**

**from fastapi import APIRouter**

**from services.serviceseries import ServiceSeries**

**#from services.serviceseriesmysql import ServiceSeriesMySql**

**from models.serie import Serie**

**service = ServiceSeries()**

**#service = ServiceSeriesMySql()**

**router = APIRouter()**

**@router.get("/series")**

***def* getSeries():**

**series = service.getSeries()**

**return {"data": series}**

**@router.get("/find/{idserie}")**

***def* findSerie(*idserie*: *int*):**

**serie = service.findSerie(idserie)**

**return {"data": serie}**

**@router.post("/insert")**

***def* insertSerie(*serie*: Serie):**

**service.insertSerie(serie.*id*, serie.*nombre***

**, serie.*imagen*, serie.*year*)**

**return {"data": "Success"}**

**@router.put("/update")**

***def* updateSerie(*serie*: Serie):**

**service.updateSerie(serie.*id*, serie.*nombre***

**, serie.*imagen*, serie.*year*)**

**return {"data": "Success"}**

**@router.delete("/delete/{idserie}")**

***def* updateSerie(*idserie*: *int*):**

**service.deleteSerie(idserie)**

**return {"data": "Success"}**

**PAGINA INICIAL EN API**

La última píldora es realizar una página de inicio para nuestro Api

Por defecto, solamente tenemos ahora mismo un maravilloso mensaje

Para montar una página de presentación necesitamos una serie de elementos:

1. Plantilla para nuestra página **index.html**

Creamos una carpeta llamada **templates** y una página llamada **index.html**

1. Esto se genera en **main.py**

Necesitamos importar un **templating** de fastapi

1. Indicar que devolveremos una respuesta HTML a un Request

**from fastapi import FastAPI, Request**

**from fastapi.responses import HTMLResponse**

**from fastapi.templating import Jinja2Templates**

**from controllers.seriescontroller import router as seriescontroller**

**app = FastAPI()**

**templates = Jinja2Templates(*directory*="templates")**

**app.include\_router(seriescontroller, *prefix*="/api", *tags*=["Series"])**

**@app.get("/", *response\_class*=HTMLResponse)**

***def* root(*request*: Request):**

**return templates.TemplateResponse("index.html"**

**, {"request": request**

**, "title": "Welcome" })**