FAST API

Requisitos para trabajar:

* Visual Studio Code
* Python 3.10 +
* Pip: Instalador de librerías de Python

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Python es un lenguaje de programación Back que contiene POO. Tiene definición de clases, creación de objetos y librerías.

La gran demanda de Python es gracias a dos tecnologías: Big Data, Hacking

Las tecnologías/Frameworks actuales que podemos utilizar con Python:

* Hacking: Código abierto y que puede atacar a cualquier cosa
* Big Data: Análisis y gráficos masivos de datos
* Jupyter: Permite analizar cantidades gigantes de datos de forma sencilla
* Django: Crear página Web MVC dentro del lenguaje Python
* Fast Api: Creación de Apis con lenguaje Python
* IA: Necesitamos un entorno para entrenar a los modelos/agentes.

A todos estos elementos se les llama Framework. Un Framework es una arquitectura basada en un lenguaje.

Fast Api es un servicio que permite representar datos o consumir datos desde un origen. Nos permite modificar, leer o insertar datos a través de peticiones HTTP.

Un Servicio Api es tan popular porque representa los datos en su mínima expresión, es decir, devuelve los datos Serializados con un formato reconocible como JSON/XML. Devuelve texto.

La serialización implica almacenar un objeto en un punto determinado con un formato.

La deserialización implica recuperar un objeto desde otro punto con la misma forma.

JSON es una representación textual de datos en formato Key/Value (más popular)

XML es una representación textual en formatos etiquetas/tags

Un servicio Api está por la siguiente arquitectura:

* **Controllers:** Un controlador es una clase/router que se encarga de manejar en su interior todas las peticiones HTTP que se realicen al servicio.
* **EndPoint:** Es el punto de acceso para las peticiones HTTP, podemos tener múltiples EndPoints con diferentes métodos.
* **Models:** Son las clases que se utilizan para recibir/enviar datos a nuestro Servicio Api.
* **Responses:** Son las respuestas que podemos devolver desde un Api, pueden ser simples o crear alguna respuesta compleja para el Servicio.
* **Services:** Un servicio es una clase que proporciona datos a nuestro Api, se puede utilizar un Servicio de BBDD para leer registros hacia nuestro Api.

Cada controlador (opcional) puede tener una serie de métodos:

* GET: Se utiliza para recuperar datos de un servicio Api.
* PUT: Este método se utiliza para modificar datos de un Servicio Api
* DELETE: Este método se utiliza para eliminar datos de un Servicio Api.
* POST: Este método tiene dos variantes:
  + Insertar información en el Api
  + Enviar objetos cifrados y utilizar dichos objetos para otras peticiones. Ejemplo: Seguridad

Aquí tenemos un ejemplo de un Servicio Api

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

Controller: **Departamentos**

EndPoint: **GET api/departamentos**

Lo que estamos viendo en la página Web en su representación visual es la documentación de un Api: **Swagger**

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Configuración de nuestro entorno de trabajo.

Por ahora, en algún momento crearemos un Fast Api pero, lo primero que vamos a realizar es consumir un Api.

Depende del tipo de proyecto, será necesario hacer algún paso más.

Por ejemplo, para ejecutar Frameworks, es necesario tener un entorno Python aislado para ejecutar las características.

Para ejecutar código simple que muestre un dato o realice bucles, simplemente necesitamos Python.

Un Environment es un espacio aislado de Python que contiene las librerías y características necesarias para trabajar con un Framework

Vamos a crear un entorno para un proyecto que utilizaremos para leer diferentes Apis.

El entorno lo vamos a crear mediante Visual Studio Code.

Sobre nuestro equipo, creamos una nueva carpeta de trabajo llamada **CONSUMOAPISMANANAS**

Necesitamos la siguiente extensión dentro de Visual Studio para trabajar con Python:

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Abrimos la carpeta de **consumoapis**

Vamos a crear un entorno/environment dentro de VS code y nuestra carpeta para instalar las librerías de consumo de apis y probar la funcionalidad.

En la parte superior de VS Code o con las teclas CONTROL + SHIFT + P y seleccionamos **Show And Run Commands**

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Seleccionamos/Escribimos **Python: Create Evironment**

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Seleccionamos **.venv**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Seleccionamos una librería y lo tenemos

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Tendremos un nuevo entorno creado (opcional) y ya podemos instalar librerías en dicho entorno.

El siguiente paso es visualizar nuestra ventana de terminal que será dónde escribamos las instrucciones para ejecutar los programas de Python.

Abrimos **Terminal** y en la parte de la derecha, al lado del Plus, seleccionamos **Command Prompt**

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Podremos comprobar que estamos dentro del Entorno porque tenemos antes de las instrucciones la palabra **(.venv)**

Interfaz de usuario gráfica, Texto, Aplicación

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 env**

Activar el entorno

**.venv\Scripts\activate.bat**

Verificar que tenemos el entorno activado

**echo $VIRTUAL\_ENV**

Un api se puede consumir con cualquier lenguaje:

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

Dentro de Python, existen librerías que vienen dentro del propio compilador de Python, por ejemplo: Math, Files…

Tenemos librerías para características externas como, por ejemplo, Fast Api o IA.

Las librerías externas debemos instalarlas dentro de nuestro entorno de trabajo y las utilizaremos ahí. Dichas librerías están aisladas, lo que quiere decir es que solamente podemos utilizarlas en un proyecto.

Para poder instalar librerías se utiliza el comando **pip**. Pip es un repositorio global de librerías, dicho repositorio pertenece a la comunidad y los desarrolladores van subiendo sus implementaciones/versiones a ese lugar.

Para consumir Apis tenemos una librería llamada **requests.** Dicha librería debemos instalarla en nuestro entorno.

Escribimos lo siguiente dentro de nuestra línea de comandos en Terminal:

**pip install requests**

Texto

El contenido generado por IA puede ser incorrecto.

Vamos a consumir un Empleado de el siguiente Api.

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

Necesitamos dos características:

1. URL: El lugar dónde está nuestra Api alojada
2. EndPoint: El punto de acceso para nuestra petición

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.

Necesitamos el Request de acceso

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Para organizarnos, vamos a escribir los nombres de programa secuenciales, es decir, los llamaremos **pythonXXdescripcion.py**

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

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.

**PYTHON01GETEMPLEADO.PY**

**#Python delimita todo por su tabulación**

**#Debemos escribir seguido hasta que hagamos "algo": if, for...**

**#Guardar: Control + S**

**#Ejecutar Command line: python NOMBREPROGRAMA.PY**

**#NECESITAMOS LA LIBRERIA REQUESTS**

**import requests**

**print("Ejemplo consumo api empleados id")**

**#PYTHON NO UTILIZA TIPADOS POR DEFECTO, AUNQUE PODEMOS HACERLO**

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

**#CUALQUIER PETICION API NOS DEVUELVE UNA RESPUESTA**

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

**#LA RESPUESTA YA DEPENDE DEL FORMATO DEL SERVICIO API (XML/JSON)**

**#RECUPERAMOS EL OBJETO JSON SERIALIZADO**

**jsonObject = response.json()**

**#Recuperamos el apellido**

**apellido = jsonObject["apellido"]**

**#EN PYTHON PODEMOS CREAR OBJETOS CON FORMATO JSON NATIVOS (KEY, VALUE)**

**#LLAMADOS DICCIONARIOS**

**print(*f*"Apellido: {apellido}")**

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

El siguiente ejemplo vamos a consumir múltiples datos.

En realidad, cada dato será un objeto diccionario dentro de Python.

Lo que nos devolverá será un conjunto de objetos en una colección/lista de Python.

Una lista en Python se representa con los corchetes: **[]**

lista = [3,4,5,6,7]

Vamos a consumir un conjunto de Departamentos

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Creamos un nuevo programa llamado **python02alldepartamentos.py**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

**PYTHON02ALLDEPARTAMENTOS.PY**

**import requests**

**print("Leyendo coleccion departamentos")**

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

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

**departamentos = response.json()**

**#COMENTAR EN BLOQUE**

**#COMENTAR: CONTROL + K + C**

**#DESCOMENTAR: CONTROL + K + U**

**#PARA RECORRER LOS OBJETOS PODEMOS UTILIZAR UN BUCLE CONTADOR**

**# for i in range(len(departamentos)):**

**#     #capturamos cada objeto departamento**

**#     row = departamentos[i]**

**#     print(row["nombre"])**

**#UTILIZANDO UN BUCLE DE REFERENCIA, QUE RECORRE OBJETOS EN LUGAR**

**#DE CONTADORES**

**for row in departamentos:**

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

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

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

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

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

El siguiente paso es crearnos nuestras propias Apis, es decir, centrarnos en FastApi

Requisitos necesarios:

* Proyecto de Fast Api (Carpeta distinta)
* Environment
* Instalar un servidor Web para desplegar el Api (Uvicorn)
* Pip: Instalador de aplicaciones de Python

**Nota:** Por defecto, FastApi utiliza uvicorn si lo instalamos como standard.

Características de un proyecto FastApi:

* **Tipado de datos:** Los datos dentro de los métodos de los controladores deben ser tipados.
* **Librerías de parámetros:** Dentro de FastApi, dependiendo de la lógica, necesitaremos algunas librerías para Modelos o para recibir simples variables en los métodos.
* **Documentación:** Por defecto, ya viene FastApi con documentación **swagger** o **redoc**

Necesitamos una clase principal llamada **main.py (Controller)**

Dentro de dicha clase es dónde estarán nuestros **EndPoints**.

Creamos otra carpeta distinta para comenzar con nuestro proyecto llamada **fastapimananas** y la abrimos con nuestro VS Code.

Creamos un nuevo entorno en esta carpeta.

El siguiente paso es agregar las librerías de FastApi, en el terminal, escribimos la siguiente instrucción:

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

Texto

El contenido generado por IA puede ser incorrecto.

Necesitamos un clase de inicio que será nuestro Controlador principal.

Posteriormente tendremos múltiples controladores pero, por ahora, solo un Controller y múltiples Endpoints.

La clase principal la podemos llamar como deseemos, pero se suele llamar **main.py**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Cada EndPoint dentro de FastApi es un método **def**. Es la palabra clave para crear métodos dentro de Python

Podemos crear métodos que NO sean del Api y métodos que SI deseamos que pertenezcan al Api.

Los métodos del Api los representa al aplicación (**app**) y debemos indicar el tipo de método:

**@app.get**

**@app.post**

**@app.put**

**@app.delete**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Una vez que tenemos métodos EndPoint es el momento de lanzar nuestro servidor.

Tenemos múltiples formas:

1. **Uvicorn:**

**uvicorn main:app**

1. Fast Api Server: indicamos el servidor a ejecutar

**fastapi run server main.py**

1. Fast Api Production: Sirve para ejecutar Fast Api una vez que estemos en producción, es lo mismo, pero no tiene Debug y no hace los cambios al vuelo

**fastapi run main.py**

1. Lanzar la aplicación con Uvicorn y depuración y cambios al vuelo

**fastapi dev main.py**

Texto

El contenido generado por IA puede ser incorrecto.

Tendremos una URL de acceso:

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

El contenido generado por IA puede ser incorrecto.

Por último, lo que hemos realizado está bien, pero está mal, es decir, estamos devolviendo en un Api un triste texto.

Lo que siempre debemos devolver dentro de un Api es formato JSON.

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.

Lanzamos nuestro servidor

**fastapi dev main.py**

Si no vemos (**.venv**) en CMD, lo activamos

**.venv\Scripts\activate.bat**

O lo creamos de nuevo

**python -m venv venv**

Podremos tener todos los métodos que deseemos dentro de nuestro FastApi. Lo único que debemos tener claro es que No se pueden repetir los Endpoints.

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.

Una gran ventaja radica en que la documentación ya está montada.

Tenemos dos documentaciones, por un lado tenemos:

1. **Swagger:** es un standard de documentación de Apis que utilizan múltiples lenguajes como Java, Net Core 8.0

<http://127.0.0.1:8000/docs>

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

1. **Redoc:** Es menos conocido, pero Microsoft se ha ido con ellos, así lo más seguro subirá como la espuma

<http://127.0.0.1:8000/redoc>

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

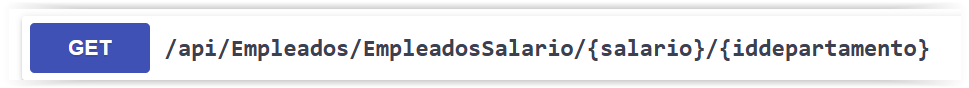
**PARAMETROS EN LOS METODOS**

Podemos recibir información en los métodos, no será lo mismo recibir información en un GET que en un POST.

La información en los métodos GET se debe realizar mediante URL.

Tenemos dos opciones en el momento de recibir datos:

1. **Mapping/Routing:** Un Mapping significa que recibiremos los parámetros en orden dentro de la URL y no tienen NAME en el momento de llamarlos. Swagger es capaz de documentar dichos parámetros.



1. **QueryString:** Recibe los parámetros mediante URL clásica. El orden de los parámetros NO importa y debemos indicar el **NAME** de cada parámetro.

Swagger no documenta está información

<https://servicioapi.com?param1=valor1&param2=valor&param3=valor3>

Todos los parámetros sin importar el tipo deben llamarse IGUAL a cómo los recibimos en el método Python

Realizamos un nuevo método para devolver el doble de un número mediante **Mapping**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

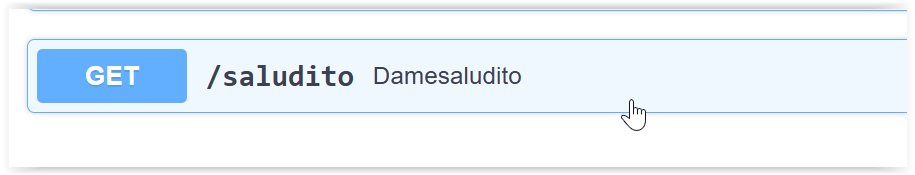
Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

El contenido generado por IA puede ser incorrecto.

La siguiente opción es crear un método para utilizar QueryString.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.



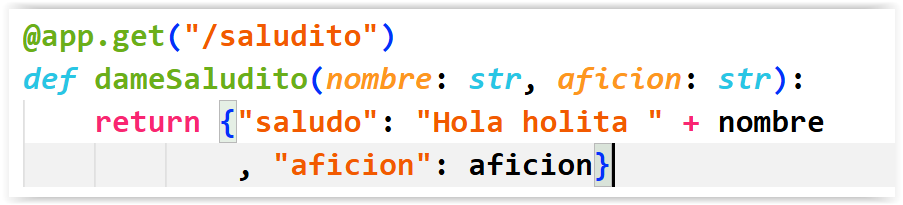
Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

El parámetro nombre es OBLIGATORIO



Por supuesto, podemos recibir más de un parámetro



Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Podemos hacer que los parámetros sean opcionales en **QueryString**

Para ello, debemos utilizar la librería **typing** y una clase llamada **Union**

Modificamos el método de **saludito** y hacemos que la afición sea opcional

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.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Podemos combinar tanto QueryString como Mapping en un mismo método

Modificamos el método **doble**

Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Esta opción se utilizaría para devolver elementos de una forma determinada.

* Los parámetros Mapping nos sirven para “filtrar” lo que desearíamos de un conjunto de datos
* Los parámetros QueryString serían para indicar cómo queremos que sean devueltos.

Como ejemplo, podríamos tener lo siguiente, imaginamos que tenemos unos Pedidos y necesitamos recuperar los Pedidos por un determinado Cliente

Mapping:

<http://127.0.0.1:8000/pedidos/MiCliente>

QueryString:

<http://127.0.0.1:8000/pedidos/MiCliente?orderby=Fecha>

**OBJETOS COMPUESTOS**

Estamos devolviendo objetos simples como, números o textos.

Si necesitamos devolver múltiples objetos necesitamos “algo extra”.

Vamos a visualizar cómo devolver objetos compuestos, una lista, por ejemplo.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Lo que estamos viendo en la salida no es un patrón JSON, KEY:VALUE

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Vamos a visualizar qué sucede si devolvemos key:value

Interfaz de usuario gráfica, Texto, Aplicación, Word

El contenido generado por IA puede ser incorrecto.

Sigue sin ser un patrón key:value

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Cuando trabajamos con FastApi, el Framework se encarga de “traducir” ciertos tipos de dato:

1. Simples: String, int, float, byte
2. Compuestos: Diccionarios, listas []

Debemos realizar nosotros la traducción. Modificamos el método.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Y ya tendremos el formato necesario.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Estamos devolviendo simples números, pero es posible que devolvamos objetos más complejos.

No debería importar devolver simples números o devolver objetos más complejos

Dependiendo de la complejidad o de dónde leamos los datos (files, bbdd, otro Api) el Framework es posible que NO pueda traducir los datos y nos mostraría caracteres de escape, por ejemplo:

Interfaz de usuario gráfica, Sitio web

El contenido generado por IA puede ser incorrecto.

Para ello, no importa si son conjuntos de datos simples o compuestos, por norma siempre debemos CODIFICAR los datos en la salida.

Para codificar tenemos dos formas:

1. **Manual:** Utilizando una clase que codifica nuestros datos de Diccionarios

Vamos a codificar los datos de nuestro diccionario de números.

Debemos importar una nueva librería de **fastapi.encoders** llamada **jsonable\_encoder**



Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

1. **Models:** Utilizando un modelo que se encarga de codificar los datos por nosotros.

Dentro de FastApi se utilizan modelos para recibir datos dentro del POST o del PUT.

También se pueden utilizar modelos para devolver datos en cualquier método.

Estos modelos deben estar definidos con una clase propia de FastApi, que se encarga de codificar los datos tanto en la recepción como en la salida.

Para esta acción, necesitamos Clases de Python y dichas clases deben cumplir dos normas:

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

**BaseModel** es la clase que utiliza FastApi y está dentro de la librería **pydantic**

Definición de una clase en Python

class NombreClase():

propiedad1 = valor1

propiedad2 = valor2

Herencia en Python

class NombreClase(**ClaseSuper**):

propiedad1 = valor1

propiedad2 = valor2

Vamos primero a crear una clase **BaseModel** que recibirá un método **PUT**

Por ahora, seguiremos haciendo todo dentro del **main.py**



Texto

El contenido generado por IA puede ser incorrecto.

Probamos nuestro nuevo método dentro de **Swagger**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

A continuación, vamos a leer los datos de un origen, por ejemplo, de un documento JSON.

Por ahora, vamos a seguir “mezclando” todo en el maravilloso **main.py**

Debemos leer el JSON y debemos codificar dicho JSON, ya sea manual o mediante BaseModel

Sobre nuestro proyecto, creamos una carpeta llamada **data** y copiamos **jugadores.json** dentro

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

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

Creamos un nuevo método que nos devolverá los jugadores

Texto

El contenido generado por IA puede ser incorrecto.

Aunque estamos viendo correctamente los datos JSON y “parece” que está bien, no estamos siguiendo el patrón de FastApi.

Debemos **siempre** codificar los datos de salida.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

También podemos devolver los datos mediante modelos/clases que hereden de BaseModel, de esta forma, no tenemos que estar haciendo la acción manual, lo hace internamente.

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

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.

Modificamos nuestro código para devolver la clase.

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

***def* readPlayers():**

**#Abrimos el fichero y lo cargamos en memoria**

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

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

**salida = []**

**#Recorremos todos los jugadores**

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

**jugador: Player = Player()**

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

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

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

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

**salida.append(jugador)**

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

## ORGANIZACIÓN DE FASTAPI

Actualmente tenemos todo mezclado dentro del propio **main.py**

Realmente, puede estar bien si es pequeño el proyecto, pero si estamos trabajando en un proyecto grande con bbdd, muchos modelos a devolver y multitud de métodos es una locura.

Debemos seguir un patrón de organización dentro de nuestro proyecto, es decir, separar las diferentes capas en clases y carpetas para organizar todo.

* **Models:** Aquí debemos tener todas nuestras clases BaseModel
* **Services:** Aquí tendremos todos los accesos para devolver datos de dónde sea, una bbdd otro api o lo que sea
* **Controllers:** Aquí separaremos las peticiones por EndPoints dependiendo del tipo de Objeto.
* **Utils:** Herramientas para utilizar dentro de nuestro FastApi, por ejemplo, cifrado para seguridad o Wrappers para mapear la respuesta de nuestros Apis.

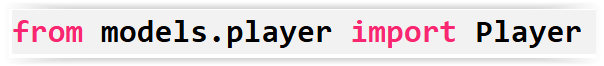
Sobre el proyecto, creamos una nueva carpeta llamada **models** y una clase llamada **player.py**

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

El contenido generado por IA puede ser incorrecto.

A continuación, dentro de **main.py** importamos nuestros **models** y la clase **Player**

**MAIN.PY**



A continuación, debemos separar la petición de los datos de los EndPoints.

Diagrama de objetos

Diagrama

El contenido generado por IA puede ser incorrecto.

Creamos una carpeta llamada **services** que leerá los datos JSON y los devolverá directamente al **main.py**

La clase de servicios podemos crearla de dos formas:

1. **Class**: Una clase tiene métodos, propiedades y constructor.

Solamente creamos un class si necesitamos propagar los objetos y construirlos, por ejemplo.

Si estamos consumiendo una base de datos, y solamente tenemos métodos, en cada método tendríamos que realizar la conexión y crearla.

Deberíamos crear una clase dónde, en el constructor, hagamos la conexión.

Y en los métodos utilizamos dicha conexión (Objeto)

1. **Methods Class**: Solamente tiene métodos, son solo acciones, como herramientas.

Solamente necesitamos funcionalidades, por ejemplo:

Interfaz de usuario gráfica, Texto, Aplicación, Word

El contenido generado por IA puede ser incorrecto.

En nuestro ejemplo, como estamos leyendo de un fichero, no necesitamos constructor ni nada, vamos a realizar un Servicio de solo métodos.

Creamos una nueva carpeta llamada **services** y una clase llamada **ServiceJugadores.py**

**import json**

**from models.player import Player**

***def* getPlayers():**

**#Abrimos el fichero y lo cargamos en memoria**

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

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

**salida = []**

**#Recorremos todos los jugadores**

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

**jugador: Player = Player()**

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

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

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

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

**salida.append(jugador)**

**return salida**

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

**#Abrimos el fichero y lo cargamos en memoria**

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

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

**#Recorremos todos los jugadores**

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

**if (row["numero"] == idPlayer):**

**jugador: Player = Player()**

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

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

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

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

**return jugador**

Por último, utilizamos nuestro nuevo **Service** dentro de **main.py**

Texto

El contenido generado por IA puede ser incorrecto.

Y ya tendremos la funcionalidad

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Los controladores nos permiten separar los Endpoints dependiendo de funcionalidades o conjuntos de elementos.

Un controlador dentro de FastApi es una clase de tipo **APIRouter**

Mediante un APIRouter separamos las características por funcionalidades y, dicha clase Controller estará referenciada dentro de la clase principal **main.py**

**CLASE CONTROLLER**

router = APIRouter()

router.get(“/EndPoint”)

def metodoAccion():

return “algo”

Posteriormente, en la clase principal **main.py** registramos el Router

**MAIN.PY**

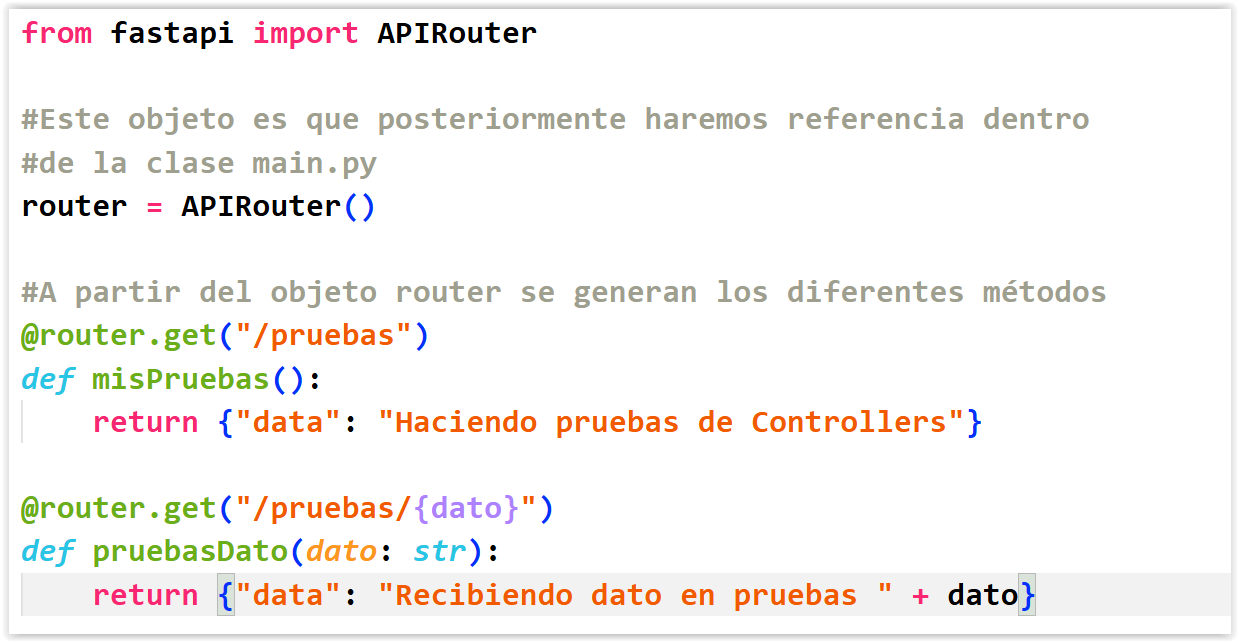
app = FastAPI()

app.**include\_router(CLASE CONTROLLER)**

Vamos a visualizar este concepto con una clase que no tenga nada que ver con los jugadores.

Devolvemos un método, lo montamos y lo probamos.

Creamos una carpeta llamada **controllers** y dentro una clase llamada **datoscontroller.py**



Posteriormente, registramos el nuevo controller dentro de **main.py**

**MAIN.PY**

Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Montamos la funcionalidad de los jugadores en un Servicio nuevo

Sobre **controllers** creamos un nuevo controlador llamado **jugadorescontroller.py**

**JUGADORESCONTROLLER**

**from fastapi import APIRouter**

**import services.*ServiceJugadores* as service**

**router = APIRouter()**

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

***def* readPlayers():**

**players = service.getPlayers()**

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

**@router.get("/findplayer/{idplayer}")**

***def* findPlayer(*idplayer*: *int*):**

**player = service.findPlayer(idplayer)**

**return {"player": player}**

Agregamos el nuevo controller en el main y eliminamos los métodos de players del main.

Texto

El contenido generado por IA puede ser incorrecto.

**CONTROL DE RESPUESTAS API**

Actualmente, estamos devolviendo datos “cerrados” y respuestas de datos.

Si quisiéramos personalizar las respuestas, además de devolver datos, deberíamos crear una clase Wrapper para devolver las respuestas personalizadas.

Por ejemplo, normalmente se devuelve información (si queremos) no solamente de los datos, también podemos devolver “algo más” como puede ser un mensaje, detalles, success code

Lo que nos permite la clase es realizar un estándar de respuesta para todo el Api.

Para ello, esta es una clase herramienta así que crearemos una nueva carpeta llamada **utils** y una clase llamada **response\_wrapper.py**

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.

Desde los controllers lo que haremos será personalizar las respuestas

**JUGADORESCONTROLLER.PY**

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.

El siguiente paso es visualizar cómo podemos devolver excepciones dentro de nuestro proyecto.

Tenemos dos posibles formas:

1. Utilizando el Wrapper de respuesta para incluir un error en nuestra respuesta. Esto se captura como respuesta.
2. Devolver una excepción desde el servidor. Esta acción se captura como una excepción

En nuestro código tenemos un método que busca un jugador.

Vamos a modificar el código y, en el caso de que no encuentre el jugador devolvemos distintas respuestas: Wrapper o Exception

Texto

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

La respuesta ya está mapeada en el Wrapper

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Para devolver excepciones HTTP debemos importar una librería de FastApi



Este objeto devolverá un status\_code y también devolverá (opcional) details

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

En FastApi a esta nueva arquitectura se le denomina Inyección de dependencias y es una mentira.

En FastApi lo que indica esta tecnología es que **Un código dependa de otro código para ser ejecutado.**

Además, nos permite agrupar grupos de características en peticiones a los métodos.

Es utilizado de dos formas:

1. Dependencias de **Endpoints:** Una dependencia de un EndPoint indica que, para ejecutar un método del **EndPoint** necesitará leer primero otro método que le indicaremos.
2. Dependencias de **App:** Indica que todos los métodos de nuestro Api tendrán que pasar por un método o métodos que nosotros indiquemos antes de ser ejecutados.

Se utiliza una clase llamada **Depends** que indica que el método dependerá de otro método para su ejecución.

Para nuestro ejemplo, vamos a realizar un filtrado de jugadores.

Vamos a realizar dos métodos: **Edad** y **Posición**

**SERVICEJUGADORES**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Implementamos los métodos en el Controller y comprobamos que funcionan

Texto

El contenido generado por IA puede ser incorrecto.

Además de filtrar, me gustaría poder decidir cuantos jugadores quiero en la respuesta de los filtros.

Por ejemplo, vamos a recibir un inicio y un final y devolvemos los jugadores en ese rango.

**SERVICEJUGADORES**

**#METODOS PARA FILTRAR**

***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**

**#METODOS PARA FILTRAR**

***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* filtrarJugadores(*inicio*: *int*, *fin*: *int*):**

**players = getPlayers()**

**#HACEMOS UN BUCLE DESDE INICIO HASTA FIN Y GUARDAMOS JUGADORES**

**datos = []**

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

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

**break**

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

**return datos**

Implementamos la petición dentro del Controller

**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 de un mismo método común que “podría” tener código que valide información sobre el inicio y el fin.

Si escribimos 50 métodos con inicio y fin, ya tenemos que estar pensando qué métodos lo reciben, además que podemos olvidarnos.

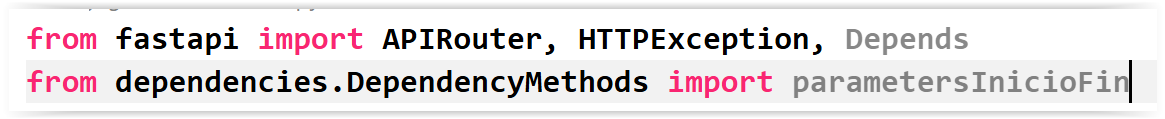
Una dependencia irá asociada a un método o varios.

Dentro de nuestro proyecto, creamos una carpeta llamada **dependencies** y una librería llamada **DependencyMethods.py**

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Volvemos a nuestro Router y debemos importar **Depends** y nuestra librería



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

***def* playersAge(*edad*: *int*, *commons*: *dict* = Depends(parametersInicioFin)):**

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

**return api\_response(*data*=players)**

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

***def* playersPosition(*posicion*: *str*,  *commons*: *dict* = Depends(parametersInicioFin)):**

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

**return api\_response(*data*=players)**

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

***def* playersFilter( *commons*: *dict* = Depends(parametersInicioFin)):**

**players = service.filtrarJugadores(commons["inicio"], commons["fin"])**

**return api\_response(*data*=players)**

Ya tenemos la funcionalidad común para varios métodos.

La ventaja está en que si necesitamos modificar algo común a todos los métodos, solamente tendríamos que cambiarlo en un UNICO lugar que es el método de dependencias.

La dependencia de aplicación implica que TODOS los métodos de mi aplicación tendrán que depender de otro método para ser ejecutados.

La inyección global se realiza desde el propio **main.py** y se realiza sobre la variable **app = FastApi()**

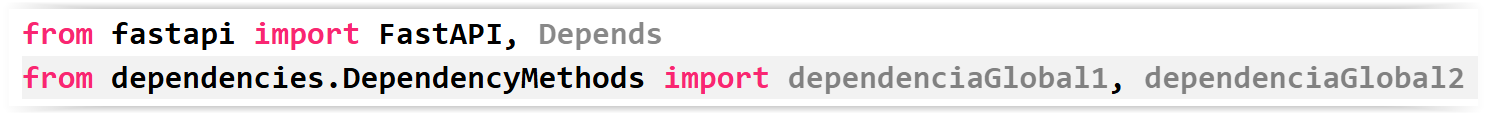
Vamos a incluir un par de métodos dentro de **dependencies**

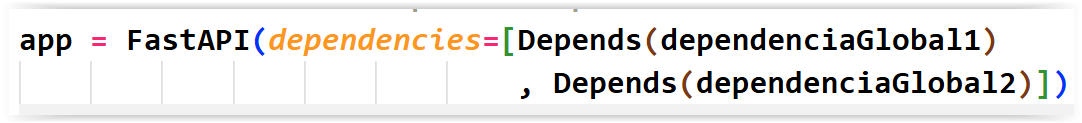
Texto

El contenido generado por IA puede ser incorrecto.

A continuación, dentro de **main.py** llamamos a nuestra librería con estos dos métodos e indicamos que dependerán de toda la aplicación.

**MAIN.PY**





Todos los métodos leerán las dependencias ANTES de ser ejecutados

Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

Imaginemos que TODOS los métodos de mi aplicación necesitan un TOKEN para poder ser ejecutados y yo validar el Token.

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

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Vamos a realizar una aplicación completa con todos los métodos y conceptos vistos en el curso.

Dicha aplicación llamará a una base de datos MySql.

Haremos un POST, PUT, DELETE y GET

Interfaz de usuario gráfica, Texto, Aplicación, Word

El contenido generado por IA puede ser incorrecto.

Creamos una nueva carpeta llamada **FASTAPISERIESMANANAS**