

Sistemas de Gestión Empresarial – 2 DAM

UD 9. Proyectos finales

Licencia



Reconocimiento – NoComercial - CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

ÍNDICE

1.	API CON FASTAPI	3
2.	CREACIÓN DEL ENTORNO VIRTUAL	3
3.	GENERAR DOCUMENTACIÓN	6
4.	PASOS PARA CREAR UNA API	8
5.	CONEXIÓN CON BBDD	30
6.	OPERACIONES CON LA BBDD	38
7.	BIBLIOGRAFÍA	51

1. API CON FASTAPI

En esta primera parte de la unidad vamos a aprender a crear APIs usando esta herramienta: FastAPI

Página oficial de FastAPI: <https://fastapi.tiangolo.com/>

Esta página se puede traducir a español. La documentación es muy amplia. La comunidad que tiene por detrás es muy potente: realiza una documentación muy completa.

- **Backend:** es lo que se está ejecutando en un servidor (página web, una app... se va a comunicar con un servidor para realizar operaciones. Las APIs se hacen a nivel de backend).
- **Frontend:** es lo que manipula el cliente final (interfaz gráfica: página web, aplicación móvil...). Parte gráfica y visual.

API: interfaz de programación de aplicaciones; es una capa de comunicación que implementa ciertos mecanismos para comunicar con el backend. Ejemplo: usuario y contraseña que se introduce en el frontend ejecutándose en un móvil, portátil...; tiene que existir un mecanismo para comunicar el front con el back y enviar esos datos (usuario y contraseña) para que se validen en el backend; eso es una API; establece unos protocolos y estándares comunes para establecer esa comunicación. FastAPI es un framework que permite crear APIs (se basa en estándares)

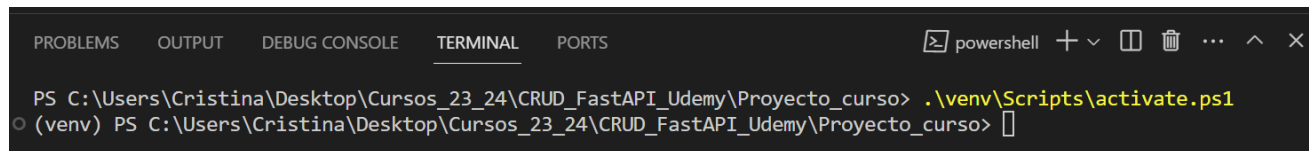
OpenAPI: marca cómo tienen que hacerse las APIs; normas, especificaciones para realizar APIs.

2. CREACIÓN DEL ENTORNO VIRTUAL

Vamos a crear una API con FastAPI. Lo primero que vamos a hacer es crear el entorno virtual:

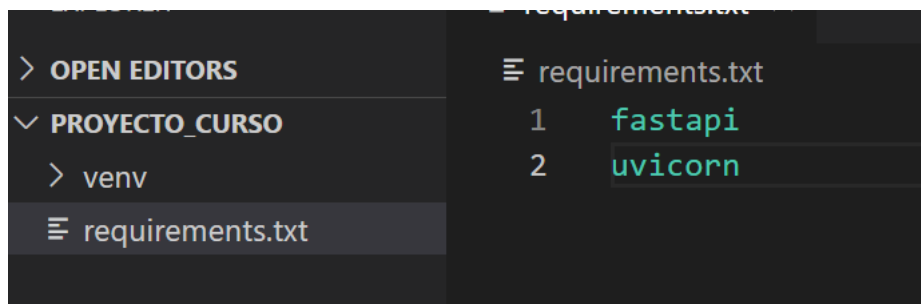
- Instalar python y pip
- Instalar Visual Studio Code (VSC)
- Crear una carpeta: FastAPI\Proyecto_curso
- A continuación, desde VSC, abrir la carpeta e instalar virtualenv: **pip install virtualenv**
- Para crear el entorno virtual: **virtualenv venv**

- Ahora hay que activar el entorno virtual:



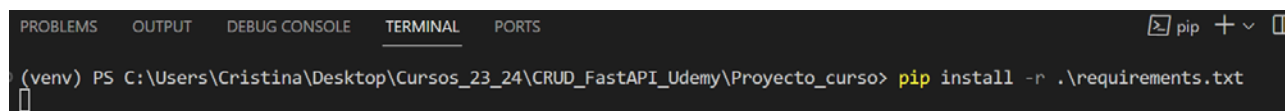
```
PS C:\Users\Cristina\Desktop\Cursos_23_24\CRUD_FastAPI_Udemy\Proyecto_curso> .\venv\Scripts\activate.ps1
(venv) PS C:\Users\Cristina\Desktop\Cursos_23_24\CRUD_FastAPI_Udemy\Proyecto_curso>
```

- Crear el archivo requirements.txt, con las librerías necesarias para el proyecto: FastAPI proporciona un servidor local para probar la API antes de pasar a producción: uvicorn



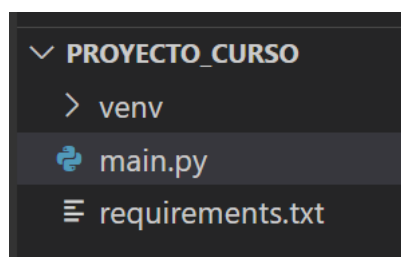
```
requirements.txt
1 fastapi
2 uvicorn
```

- Instalar las librerías:



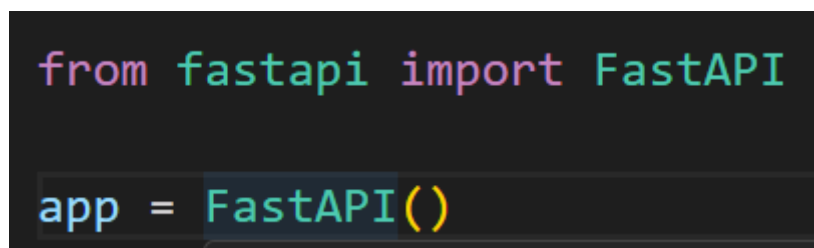
```
(venv) PS C:\Users\Cristina\Desktop\Cursos_23_24\CRUD_FastAPI_Udemy\Proyecto_curso> pip install -r .\requirements.txt
```

- Crear el fichero main.py, dentro de la carpeta proyecto_curso:



```
PROYECTO_CURSO
├── venv
├── main.py
└── requirements.txt
```

Escribir dentro de main.py:



```
from fastapi import FastAPI

app = FastAPI()
```

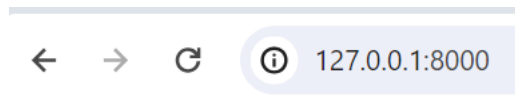
Para ejecutar FastAPI: FastAPI proporciona un servidor local para probar la API antes de pasar a producción. Se llama: uvicorn; ejecutar este comando: **uvicorn main:app**

uvicorn es el servidor

main es el nombre del fichero raíz que queremos arrancar

:app es la instancia de FastAPI

Ahora, vamos a un navegador y ponemos 127.0.0.1:8000; aparece:



```
{"detail": "Not Found"}
```

Otra forma de ejecutar FastAPI: para probarlo detener el servidor y añadir estas líneas al fichero main.py:

Para detener el servidor: **CTRL+C**

```
main.py > ...  
1  from fastapi import FastAPI  
2  import uvicorn  
3  
4  app = FastAPI()  
5
```

```
if __name__ == "__main__":  
    uvicorn.run("main:app", port=8000, reload=True)
```

reload recarga el contexto del servidor cada vez que se cambie algo en el fichero main (al hacer CTRL+S).

Para probarlo: **python .\main.py**

```
INFO: Finished server process [5316]
(venv) PS C:\Users\Cristina\Desktop\Cursos_23_24\CRUD_FastAPI_Udemy\Proyecto_curso> ^C
(venv) PS C:\Users\Cristina\Desktop\Cursos_23_24\CRUD_FastAPI_Udemy\Proyecto_curso> python .\main.py
INFO: Started server process [3348]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

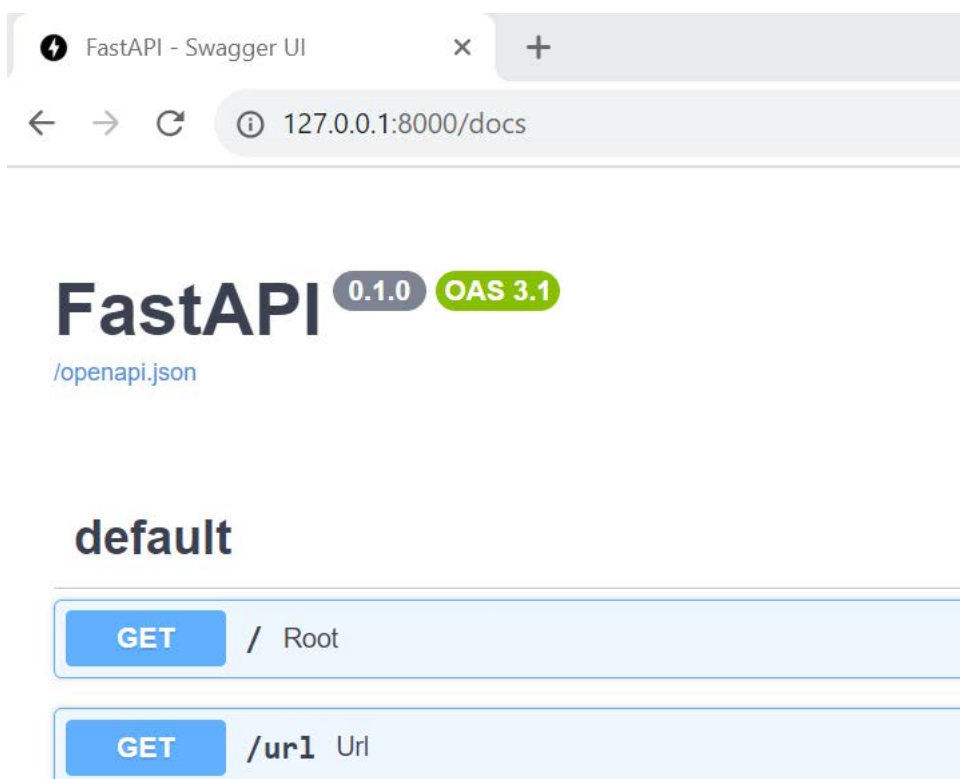
Crear el fichero readme.md y añadir información sobre las 2 formas de ejecutar FastAPI.

3. GENERAR DOCUMENTACIÓN

Actualmente, existen herramientas para generar documentación siguiendo estándares; una de las más usadas es: **swagger IU**. Permite crear una página web que funcione como documentación: indique las peticiones de la API, parámetros, endpoint...

<https://fastapi.tiangolo.com/tutorial/first-steps/#interactive-api-docs>

Si desde el explorador, indicamos la siguiente dirección, vemos que la documentación de la API se ha creado de forma automática:



Hay un botón para probarlo (hacer clic sobre él y luego en Ejecutar):

GET / Root

Parameters

No parameters

Servers

These operation-level options override the global server options.

/

Execute

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/
```

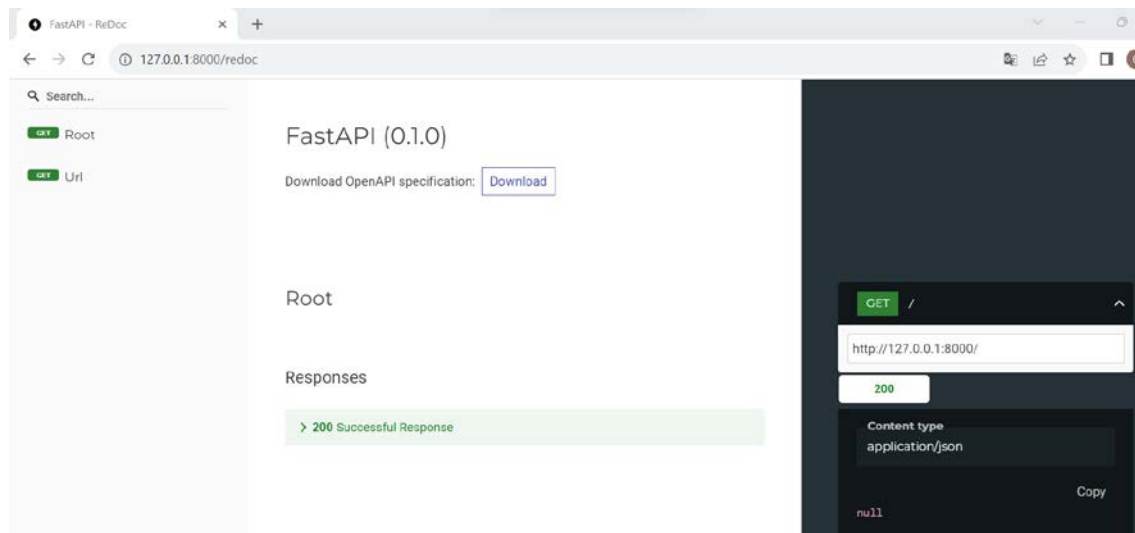
Server response

Code	Details
200	<p>Response body</p> <pre>"HOLA HOLA a todos!"</pre> <p>Response headers</p> <pre>content-length: 20 content-type: application/json date: Tue, 26 Dec 2023 08:07:22 GMT server: uvicorn</pre>

A medida que vamos programando, se va generando la documentación. Se pueden añadir comentarios.

Hay otro gestor que se usa mucho también: Redocly:

<https://fastapi.tiangolo.com/tutorial/first-steps/#alternative-api-docs>



Vemos que podemos descargar la especificación de la API; si lo probamos vemos que se descarga un JSON.

Después de cerrar el servidor, la aplicación..., para volver a activar el entorno y arrancar el servidor:

```
PS C:\Users\Cristina\Desktop\Cursos_23_24\CRUD_FastAPI_Udemy\Proyecto_curso> .\venv\Scripts\activate.ps1
(venv) PS C:\Users\Cristina\Desktop\Cursos_23_24\CRUD_FastAPI_Udemy\Proyecto_curso> python .\main.py
```

4. PASOS PARA CREAR UNA API

Añadir este código al fichero main.py y probarlo:


```
main.py > ruta1
1  from fastapi import FastAPI
2  import uvicorn
3
4  app = FastAPI()
5
6  @app.get("/ruta1")
7  def ruta1():
8      return {"mensaje": "Enhorabuena; has creado tu primera API!!!"}
9
10 if __name__ == "__main__":
11     uvicorn.run("main:app", port=8000, reload=True)
12
```

Para probarlo, **hacerlo desde la documentación de swager**

Hasta ahora, hemos hecho una petición de tipo GET. Vamos a realizar una operación de tipo POST.

Añadir la parte del post en el fichero main.py:

```
main.py > ruta2
1  from fastapi import FastAPI
2  import uvicorn
3
4  app = FastAPI()
5
6  @app.get("/ruta1")
7  def ruta1():
8      return {"mensaje": "Enhorabuena; has creado tu primera API!!!"}
9
10 @app.post("/ruta2")
11 def ruta2(user):
12     print(user)
13     return True
14
15
16 if __name__ == "__main__":
17     uvicorn.run("main:app", port=8000, reload=True)
18
```

Para probarlo desde swager: hay que introducir un nombre de user (query):

← → ↻ 127.0.0.1:8000/docs#/default/ruta2_ruta2_post ☆ 📄

FastAPI 0.1.0 OAS 3.1

/openapi.json

default

GET /ruta1 Ruta1

POST /ruta2 Ruta2

Parameters

Name	Description
user required	
(query)	Cristina

Execute

Cancel

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/ruta2?user=Cristina' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://127.0.0.1:8000/ruta2?user=Cristina
```

Server response

Code	Details
200	<div><div>Response body</div><pre>true</pre><div>Download</div></div> <div><div>Response headers</div><pre>content-length: 4 content-type: application/json date: Wed, 27 Dec 2023 16:18:07 GMT server: uvicorn</pre></div>

Responses	
Code	Description
200	Successful Response
Media type application/json	
Controls Accept header.	
Example Value Schema	
"string"	
422	Validation Error
Media type application/json	
Example Value Schema	
{ "detail": [{ "loc": ["string", 0], "msg": "string", "type": "string" }] }	

El siguiente paso será definir un modelo para trabajar con los datos. Para ello, se usa pydantic (BaseModel). Pydantic es una biblioteca de Python. Podemos buscar en el navegador: pydantic fastapi

Vamos a crear un modelo de usuario:

```
main.py > ...  
1  from fastapi import FastAPI  
2  import uvicorn  
3  from pydantic import BaseModel  
4  from typing import Optional  
5  from datetime import datetime  
6  
7  #User Model  
8  class User(BaseModel): #Schema  
9      id:int  
10     nombre:str  
11     apellido:str  
12     direccion:Optional[str] #Parámetro opcional; es necesario importar Optional  
13     telefono:int  
14     creacion_user:datetime=datetime.now() #Fecha por defecto
```

Y luego, para implementar este esquema: en la operación del post, hay que indicar que el usuario que va a recibir será igual al modelo que hemos creado:

```
17 app = FastAPI()
18
19 @app.get("/ruta1")
20 def ruta1():
21     return {"mensaje": "Enhorabuena; has creado tu primera API!!!"}
22
23 @app.post("/ruta2")
24 def ruta2(user: User):
25     print(user)
26     return True
27
28
29 if __name__ == "__main__":
30     uvicorn.run("main:app", port=8000, reload=True)
```

Para probarlo: vamos a la página de swagger y vemos que aparece el esquema del JSON:

The image shows the Swagger UI for the POST endpoint /ruta2 Ruta2. The 'Parameters' section is empty. The 'Request body' section is marked as 'required' and has a dropdown menu set to 'application/json'. Below this, the 'Example Value' tab is selected, displaying a JSON schema for a User object. The schema includes fields: id (integer), nombre (string), apellido (string), direccion (string), telefono (integer), and creacion_user (string with a date and timestamp).

```
{
  "id": 0,
  "nombre": "string",
  "apellido": "string",
  "direccion": "string",
  "telefono": 0,
  "creacion_user": "2023-12-27T17:46:44.049083"
}
```

Rellenar el JSON con datos ficticios:

The image shows the Swagger UI for the POST endpoint /ruta2 Ruta2, similar to the previous one, but with a filled JSON example in the 'Example Value' tab. The 'Request body' dropdown is still set to 'application/json'. The JSON example contains fictitious data for a user named Cristina Silvan.

```
{
  "id": 3,
  "nombre": "Cristina",
  "apellido": "Silvan",
  "direccion": "Valladolid",
  "telefono": 678,
  "creacion_user": "2023-12-27T17:46:44.049083"
}
```

Ejecutar y la respuesta es:

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8080/ruta2' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 3,
    "nombre": "Cristina",
    "apellido": "Silvan",
    "direccion": "Valladolid",
    "telefono": 678,
    "creacion_user": "2023-12-27T17:46:44.049083"
  }'
```

Request URL

```
http://127.0.0.1:8080/ruta2
```

Server response

Code Details

200

Response body

```
true
```

Response headers

```
content-length: 4
content-type: application/json
date: Wed, 27 Dec 2023 16:50:56 GMT
server: uvicorn
```

Responses

Code Description

200

Successful Response

Media type

Controls Accept header.

Example Value | Schema

```
"string"
```

422

Validation Error

Media type

Example Value | Schema

```
{
  "detail": [
    {
      "loc": [
        "string",
        0
      ],
      "msg": "string",
      "type": "string"
    }
  ]
}
```

Como hemos puesto un print, en el terminal de VSC se ve:

```
INFO: 127.0.0.1:60815 - "GET /openapi.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:60819 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:60819 - "GET /openapi.json HTTP/1.1" 200 OK
id=3 nombre='Cristina' apellido='Silvan' direccion='Valladolid' telefono=678 creacion_user=datetime.datetime(2023, 12, 27, 17, 46, 44, 49083)
INFO: 127.0.0.1:60832 - "POST /ruta2 HTTP/1.1" 200 OK
[]
```

Para acceder a esos datos; se puede hacer: `print(user.id)`:

```

23 @app.post("/ruta2")
24 def ruta2(user:User):
25     print(user)
26     print(user.id)
27     return True
28

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```

INFO: 127.0.0.1:61006 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:61006 - "GET /openapi.json HTTP/1.1" 200 OK
id=5 nombre='Mono' apellido='Poly' direccion='Valladolid' telefono=789 creacion_user=datetime.datetime(2023, 12, 27, 23, 14, 19, 795621)
)
5

```

Otro ejemplo, para acceder al nombre:

```

23 @app.post("/ruta2")
24 def ruta2(user:User):
25     print(user)
26     print(user.nombre)
27     return True
28

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```

INFO: Waiting for application startup.
INFO: Application startup complete.
id=5 nombre='Mono' apellido='Poly' direccion='Valladolid' telefono=789 creacion_user=datetime.datetime(2023, 12, 27, 23, 14, 19, 795621)
)
Mono

```

Para convertir el user que llega a la función ruta2 en un diccionario: se puede usar la función dict(); si la usamos, nos indica que la función dict() está deprecated y que use model_dump()::

```

23 @app.post("/ruta2")
24 def ruta2(user:User):
25     print(user)
26     print(user.model_dump())
27     return True
28

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```

INFO: 127.0.0.1:59426 - "GET /openapi.json HTTP/1.1" 200 OK
id=5 nombre='prueba' apellido='prueba' direccion='prueba' telefono=0 creacion_user=datetime.datetime(2023, 12, 28, 8, 24, 34, 402306)
{'id': 5, 'nombre': 'prueba', 'apellido': 'prueba', 'direccion': 'prueba', 'telefono': 0, 'creacion_user': datetime.datetime(2023, 12, 28, 8, 24, 34, 402306)}
INFO: 127.0.0.1:59435 - "POST /ruta2 HTTP/1.1" 200 OK

```

Ahora queremos agregar ese diccionario a una variable:

```

23 @app.post("/ruta2")
24 def ruta2(user:User):
25     usuario = user.model_dump()
26     print(usuario)
27     print(user)
28     return True

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```

INFO: 127.0.0.1:59658 - "POST /ruta2 HTTP/1.1" 200 OK
{'id': 5, 'nombre': 'Cris', 'apellido': 'Cris', 'direccion': 'Dir_Cris', 'telefono': 0, 'creacion_user': datetime.datetime(2023, 12, 28, 8, 46, 16, 271629)}
id=5 nombre='Cris' apellido='Cris' direccion='Dir_Cris' telefono=0 creacion_user=datetime.datetime(2023, 12, 28, 8, 46, 16, 271629)
INFO: 127.0.0.1:59659 - "POST /ruta2 HTTP/1.1" 200 OK

```

Vamos a continuar programando una API para crear un usuario:

Añadimos una lista llamada usuarios (usuarios = []) y cambiamos el nombre de la función ruta2 y le llamamos: crear_usuario. Podemos comprobar que en Swagger ya se ha actualizado la función.

Lo que queremos conseguir es que el usuario que enviamos por Swagger, al probar la API, se añada a la lista de usuarios, para ello modificamos la función crear_usuario de esta forma:

```
app = FastAPI()
usuarios = []

@app.get("/ruta1")
def ruta1():
    return {"mensaje": "Enhorabuena; has creado tu primera API!!!"}

@app.post("/crear_usuario")
def crear_usuario(user: User):
    usuario = user.model_dump()
    usuarios.append(usuario)
    return {"Respuesta": "Usuario creado!!!"}
```

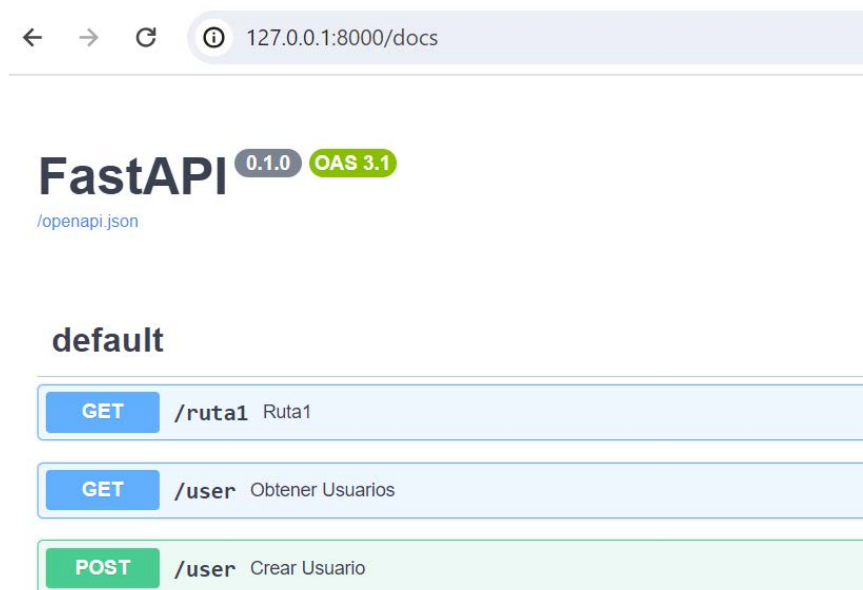
Con esto hemos programado una API para crear usuarios; ahora queremos comprobar que el usuario se ha añadido correctamente a la lista de usuarios → vamos a crear otra API para mostrar usuarios.

Para ello modificamos el nombre de la ruta /crear_usuario → /user. De esta forma, /user va a tener métodos de tipo POST, GET, DELETE...

Creamos el método GET de /user:

```
17 app = FastAPI()
18 usuarios = []
19
20 @app.get("/ruta1")
21 def ruta1():
22     return {"mensaje": "Enhorabuena; has creado tu primera API!!!"}
23
24 @app.get("/user")
25 def obtener_usuarios():
26     return usuarios
27
28 @app.post("/user")
29 def crear_usuario(user: User):
30     usuario = user.model_dump()
31     usuarios.append(usuario)
32     return {"Respuesta": "Usuario creado!!!"}
```

Si refrescamos la página de swagger, vemos que aparecen todas las APIs:



Si probamos la API de Obtener Usuarios, vemos que devuelve una lista vacía:

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/user' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/user
```

Server response

Code	Details
------	---------

200	
-----	--

Response body

```
[]
```

Response headers

```
content-length: 2
content-type: application/json
date: Thu, 28 Dec 2023 08:09:50 GMT
server: uvicorn
```

Vamos a crear un usuario, con la API POST, inventamos unos datos y probamos:

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/user' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 5,
    "nombre": "Santos",
    "apellido": "Inocentes",
    "direccion": "string",
    "telefono": 0,
    "creacion_user": "2023-12-28T09:05:56.840913"
  }'
```

Request URL

```
http://127.0.0.1:8000/user
```

Server response

Code

Details

200

Response body

```
{
  "Respuesta": "Usuario creado!!"
}
```

Response headers

```
content-length: 32
content-type: application/json
date: Thu, 28 Dec 2023 08:12:10 GMT
server: uvicorn
```

Ahora vamos a probar la API de obtener los usuarios de la lista usuarios[]:

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/user' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/user
```

Server response

Code

Details

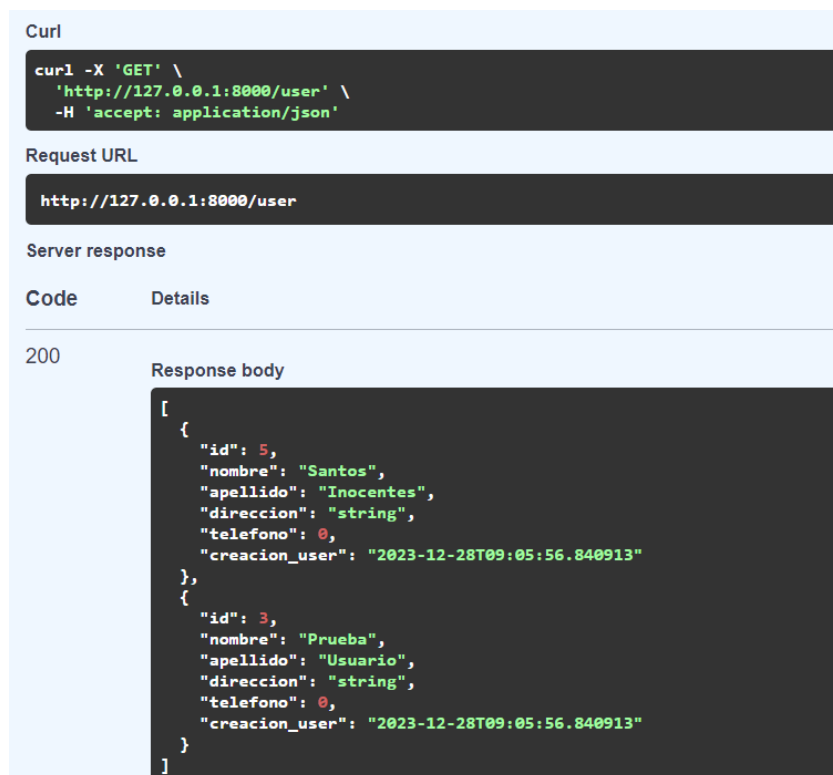
200

Response body

```
[
  {
    "id": 5,
    "nombre": "Santos",
    "apellido": "Inocentes",
    "direccion": "string",
    "telefono": 0,
    "creacion_user": "2023-12-28T09:05:56.840913"
  }
]
```

Response headers

Vemos que la respuesta es una lista de diccionarios donde cada índice representa un usuario. Podemos crear más usuarios y si volvemos a hacer el GET, aparecen el resto de los usuarios creados:



Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/user' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/user
```

Server response

Code	Details
200	<p>Response body</p> <pre>[{ "id": 5, "nombre": "Santos", "apellido": "Inocentes", "direccion": "string", "telefono": 0, "creacion_user": "2023-12-28T09:05:56.840913" }, { "id": 3, "nombre": "Prueba", "apellido": "Usuario", "direccion": "string", "telefono": 0, "creacion_user": "2023-12-28T09:05:56.840913" }]</pre>

Si ahora hacemos cualquier modificación en el código del fichero main.py (aunque sea mínima; por ejemplo, introducir un INTRO)→ si probamos el GET de Obtener usuarios, la respuesta que se obtiene es una lista vacía.

Para obtener un usuario concreto, usando su id: query parameter; ese parámetro llega mediante la URL.

Para hacerlo, en primer lugar, añadir en el fichero main.py el siguiente trozo de código:

```
34 @app.get("/user/{user_id}")
35 def obtener_usuario(user_id:int):
36     for user in usuarios:
37         print(user, type(user))
38
```

Para probarlo: crear 2 usuarios con id diferentes, usando el método POST y después probar la API que acabamos de añadir, pasando como parámetro un id. La respuesta de swagger es null, pero

desde VSC se ve que la respuesta es de tipo diccionario

The screenshot shows a REST client interface. At the top, there is a field for `user_id` with a red asterisk and the text `* required`. Below it, the type is `integer` and the path is `(path)`. The value `1` is entered in the input field. A blue `Execute` button is below the input field. Below the button, the `Responses` section is visible. It shows the `Curl` command: `curl -X 'GET' \ 'http://127.0.0.1:8000/user/1' \ -H 'accept: application/json'`. The `Request URL` is `http://127.0.0.1:8000/user/1`. The `Server response` section shows a `Code` of `200` and a `Details` of `Response body` which is `null`.

En la salida de VSC, vemos que la respuesta es de tipo es diccionario → se puede acceder directamente al id → `user["id"]`:

```
34 @app.get("/user/{user_id}")
35 def obtener_usuario(user_id:int):
36     for user in usuarios:
37         if user["id"] == user_id: #Acceder al id de user (tipo dict) y comparar con user_id que se pasa como query
38             return{"usuario":user}
39     return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
40
```

Para probarlo: crear un usuario y ver si posteriormente, al pasar su id, lo devuelve. Probar también si se pasa un id que no existe.

Name	Description
user_id * required integer (path)	<input type="text" value="1"/>

Execute

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/user/1' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/user/1
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "usuario": { "id": 1, "nombre": "Miguel", "apellido": "prueba1", "direccion": "prueba1", "telefono": 0, "creacion_user": "2023-12-28T10:53:49.639597" } }</pre>

Name	Description
user_id * required integer (path)	<input type="text" value="234"/>

Execute

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/user/234' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/user/234
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "respuesta": "Usuario no encontrado" }</pre>

Vemos en la documentación de Swagger que el id se pasa como query parameter:

<http://127.0.0.1:8000/user/1>

Se puede enviar también como tipo JSON → vamos a crear otro modelo cuyo contenido será el id de usuario:

```
#User Model
class User(BaseModel): #Schema
    id:int
    nombre:str
    apellido:str
    direccion:Optional[str] #Parámetro opcional; es necesario importar Optional
    telefono:int
    creacion_user:datetime=datetime.now() #Fecha por defecto

class UserId(BaseModel):
    id:int
```

Ahora hay que crear una nueva ruta.

Antes de continuar, hay que aclarar que cuando se usan query parameters, se puede usar el método GET o el POST. Pero no se puede crear una petición de tipo GET y enviarle un JSON porque el método GET no puede tener un body.

```
36 @app.post("/user/{user_id}")
37 def obtener_usuario(user_id:int):
38     for user in usuarios:
39         if user["id"] == user_id: #Acceder al id de user (tipo dict) y comparar con user_id que se pasa como query
40             return{"usuario":user}
41     return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
42
43 @app.post("/userjson")
44 def obtener_usuario_json(user_id:UserId):
45     for user in usuarios:
46         if user["id"] == user_id.id: #Acceder al id de user (tipo dict) y comparar con el id de user_id que se pasa como json
47             return{"usuario":user}
48     return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
49
```

Probarlo:

Request body **required**

```
{
  "id": 3
}
```

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/userjson' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 3
  }'
```

Request URL

```
http://127.0.0.1:8000/userjson
```

Server response

Code	Details
------	---------

200	
-----	--

Response body

```
{
  "usuario": {
    "id": 3,
    "nombre": "prueba3",
    "apellido": "prueba3",
    "direccion": "prueba3",
    "telefono": 0,
    "creacion_user": "2023-12-28T11:31:16.465520"
  }
}
```

Fichero main.py completo:


```

main.py > ...
1  from fastapi import FastAPI
2  import uvicorn
3  from pydantic import BaseModel
4  from typing import Optional
5  from datetime import datetime
6
7  #User Model
8  class User(BaseModel): #Schema
9      id:int
10     nombre:str
11     apellido:str
12     direccion:Optional[str] #Parámetro opcional; es necesario importar Optional
13     telefono:int
14     creacion_user:datetime=datetime.now() #Fecha por defecto
15
16     class UserId(BaseModel):
17         id:int
18
19     app = FastAPI()
20     usuarios = []
21
22     @app.get("/ruta1")
23     def ruta1():
24         return {"mensaje":"Enhorabuena; has creado tu primera API!!!"}
25
26     @app.get("/user")
27     def obtener_usuarios():
28         return usuarios
29

```

```

30     @app.post("/user")
31     def crear_usuario(user:User):
32         usuario = user.model_dump()
33         usuarios.append(usuario)
34         return {"Respuesta": "Usuario creado!!"}
35
36     @app.post("/user/{user_id}")
37     def obtener_usuario(user_id:int):
38         for user in usuarios:
39             if user["id"] == user_id: #Acceder al id de user (tipo dict) y comparar con user_id que se pasa como query
40                 return{"usuario":user}
41         return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
42
43     @app.post("/userjson")
44     def obtener_usuario_json(user_id:UserId):
45         for user in usuarios:
46             if user["id"] == user_id.id: #Acceder al id de user (tipo dict) y comparar con el id de user_id que se pasa como json
47                 return{"usuario":user}
48         return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
49
50     if __name__=="__main__":
51         uvicorn.run("main:app",port=8000,reload=True)
52

```

Resumen: existen 2 formas de obtener usuarios: por query parameter y otra creando un modelo para enviar un JSON.

Por ahora tenemos estas APIs:

FastAPI

0.1.0

OAS 3.1

/openapi.json

default

GET	/ruta1	Ruta1
GET	/user	Obtener Usuarios
POST	/user	Crear Usuario
POST	/user/{user_id}	Obtener Usuario
POST	/userjson	Obtener Usuario Json

Ahora, vamos a crear una API para eliminar un usuario; de momento añadimos este código en main.py:

```
@app.delete("/user/{user_id}")
def eliminar_usuario(user_id:int):
    for index,user in enumerate(usuarios): #Necesitamos saber el índice y el valor para ver si el user_id es = al que estamos recorriendo
        print(index,user)

if __name__=="__main__":
    uvicorn.run("main:app",port=8000,reload=True)
```

Para probarlo: crear 2 usuarios y probar a borrar uno (rellenando el id del usuario que queremos borrar como query parameter). El método DELETE devuelve null pero en VSC obtenemos esta salida (print(index, user)):

```
INFO: 127.0.0.1:60937 - "POST /user HTTP/1.1" 200 OK
INFO: 127.0.0.1:60938 - "GET /user HTTP/1.1" 200 OK
0 {'id': 5, 'nombre': 'Cris', 'apellido': 'string', 'direccion': 'string', 'telefono': 0, 'creacion_user': datetime.datetime(2023, 12, 28, 22, 21, 9, 69063)}
1 {'id': 7, 'nombre': 'Raul', 'apellido': 'string', 'direccion': 'string', 'telefono': 0, 'creacion_user': datetime.datetime(2023, 12, 28, 22, 21, 9, 69063)}
INFO: 127.0.0.1:60939 - "DELETE /user/7 HTTP/1.1" 200 OK
```

Continuamos modificando la API para borrar:

```
@app.delete("/user/{user_id}")
def eliminar_usuario(user_id:int):
    for index,user in enumerate(usuarios): #Necesitamos saber el índice y el valor para ver si el user_id es = al que estamos recorriendo
        if user["id"] == user_id:
            usuarios.pop(index)
            return {"Respuesta": "Usuario eliminado correctamente"}
    return {"Respuesta": "Usuario NO encontrado"}

if __name__=="__main__":
    uvicorn.run("main:app",port=8000,reload=True)
```

Probarlo: crear 2 usuarios, mostrar la lista de los usuarios, borrar uno de ellos indicando su id; mostrar de nuevo la lista de usuarios.

Ahora vamos a crear la API para modificar usuarios:

```
58 @app.put("/user/{user_id}")
59 def actualizar_usuario(user_id:int, updateUser:User):
60     for index,user in enumerate(usuarios): #Necesitamos saber el índice y el valor para ver si el user_id es = al que estamos recorriendo
61         if user["id"] == user_id:
62             usuarios[index]["id"] = updateUser.model_dump()["id"]
63             usuarios[index]["nombre"] = updateUser.model_dump()["nombre"]
64             usuarios[index]["apellido"] = updateUser.model_dump()["apellido"]
65             usuarios[index]["direccion"] = updateUser.model_dump()["direccion"]
66             usuarios[index]["telefono"] = updateUser.model_dump()["telefono"]
67             return {"Respuesta": "Usuario actualizado correctamente"}
68     return {"Respuesta": "Usuario NO encontrado"}
69
70 if __name__=="__main__":
71     uvicorn.run("main:app",port=8000,reload=True)
```

Probarlo: crear 1 usuario, mostrar la lista de los usuarios, actualizarlo indicando su id y los datos del usuario que queremos modificar; mostrar de nuevo la lista de usuarios y vemos que se ha modificado.

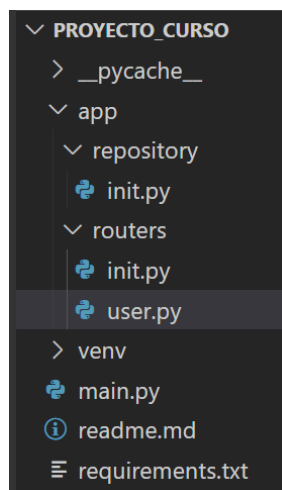
Vamos a estructurar el código. Hasta ahora lo tenemos todo dentro del archivo main.py. Ahora vamos a crear carpetas para añadir las rutas:

Un enrutador (router) en el contexto de frameworks web como FastAPI es una herramienta que permite organizar y agrupar rutas relacionadas en una aplicación. Un enrutador es un conjunto de rutas que comparten un prefijo común. Este prefijo se utiliza para agrupar lógicamente las rutas y facilitar la organización del código.

Vamos a crear un router que se encargue de toda la gestión de usuarios. Posteriormente, habría que añadir el router a nuestra aplicación.

Crear nueva carpeta: app; y dentro de ella, crear otras 2: routers y repository. Dentro de la carpeta routers y repository, crear el archivo __init__.py para que entienda que dentro de esas carpetas estamos creando módulos de Python.

Ahora vamos a definir un nuevo router de usuarios: user.py, dentro de la carpeta routers. La estructura de carpetas quedaría de esta forma:



Lo siguiente que tenemos que hacer es copiar las rutas que tenemos en el fichero main.py dentro de user.py y añadir las líneas correspondientes en la cabecera del fichero user.py:

```
from fastapi import APIRouter
```

Ahora habría que definir el router:

```
router = APIRouter(  
    prefix="/user",  
    tags=["Users"]  
)
```

APIRouter: es FastAPI, APIRouter es una clase que se utiliza para organizar y estructuras rutas de API.

prefix="/user": Este enrutador se configura con un prefijo de ruta "/user". Esto significa que todas las rutas definidas dentro de este enrutador estarán precedidas por "/user". Por ejemplo, si existe una ruta llamada "/crear_usuario", la ruta completa sería "/user/crear_usuario".

tags=["Users"]: Los tags se utilizan para organizar y documentar las rutas. En este caso, la ruta está etiquetada con "Users", lo que puede ser útil para la documentación automática generada por FastAPI. Los tags permiten clasificar y organizar las rutas según categorías específicas.

A continuación, hay que cambiar la palabra app del fichero user.py y poner router.

En este fichero user.py estamos usando esquemas que ahora mismo tenemos en el fichero main.py. Por ello, vamos a crear un archivo dentro de la carpeta app y se llamará: schemas.py.

Dentro de schemas copiamos las class User y UserId y los import necesarios.

```
from pydantic import BaseModel
```

```
from typing import Optional
```

```
from datetime import datetime
```

Ahora tenemos que importar los esquemas en user.py para poder usarlos, porque en user.py aparece User pero eso no lo reconoce. Añadir estas líneas en la parte de arriba:

```
from app.schemas import User,UserId
```

También hay que copiar la definición de la lista de usuarios dentro del fichero user.py:

```
usuarios = []
```

Ahora hay que incluir la ruta a nuestra aplicación en main.py y este fichero quedaría de la siguiente forma:

```
from fastapi import FastAPI
```

```
import uvicorn
```

```
from app.routers import user
```

```
app = FastAPI()
```

```
app.include_router(user.router)
```

```
if __name__=="__main__":
```

```
    uvicorn.run("main:app",port=8000,reload=True)
```

Probar la aplicación: crear usuario... Vemos que aparece esto:



Users

GET	/user/ruta1	Ruta1
GET	/user/user	Obtener Usuarios
POST	/user/user	Crear Usuario
GET	/user/user/{user_id}	Obtener Usuario
DELETE	/user/user/{user_id}	Eliminar Usuario
PUT	/user/user/{user_id}	Actualizar Usuario
POST	/user/user.json	Obtener Usuario .json

Para que en las rutas no aparezca user repetido tantas veces, en el archivo user.py se puede quitar el user de las rutas en @router.... Esto ocurre porque al crear el router pusimos como prefijo: /user
Comprobar que aparece bien, crear, actualizar, ver usuarios...

5. CONEXIÓN CON BBDD

Ahora vamos a conectar nuestra aplicación con una base de datos postgres: para ello, **arrancar el contenedor Docker de Odoo que contiene la base de datos postgres que hemos usado con Odoo.**

Para conectarnos a la base de datos, recordad lo que hicimos en la práctica 10 de este curso:

Estos datos de usuarios y contraseñas se encuentran en el fichero docker-compose.yaml que usamos para crear los contenedores de Docker.

Desde el navegador: **127.0.0.1:80**

puerto	80
PGADMIN_DEFAULT_EMAIL:	pgadmin4@pgadmin.org

PGADMIN_DEFAULT_PASSWORD	admin
POSTGRES_PASSWORD	odoo
POSTGRES_USER	odoo

Una vez conectados a la BBDD a través de pgAdmin4 → Dentro de servers → BBDD → Introducir las credenciales para el usuario odoo.

Botón derecho → Crear Base de datos y creo una nueva BBDD: fastapi-database

Para realizar la conexión, hay que instalar estas dependencias: psycopg2 y SQLAlchemy. Para ello, hay que añadirlas en el fichero requirements.txt y luego instalarlas (**pip install -r requirements.txt**) desde la carpeta proyecto_curso.

Dentro de la carpeta app, creamos una nueva carpeta: db; que contendrá los modelos y todo lo que tiene que ver con la conexión a la BBDD.

Dentro de ella, crear models.py, donde hay que traer la estructura de los usuarios (está en schemas.py) → De momento, copiarlo como comentario en models.py, para tenerlo como referencia.

El modelo se tiene que heredar para que toda la aplicación entienda a qué BBDD nos estamos refiriendo. Para ello crear el fichero database.py

Vamos al archivo **database.py** y escribimos lo siguiente:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
```

A continuación, definimos esta variable que va a ser la encargada de establecer la conexión:

```
#URL de la base de datos
SQLALCHEMY_DATABASE_URL = "postgresql://odoo:odoo@localhost:5342/fastapi-database"
```

odoo:odoo, es el nombre del usuario de postgres y la contraseña

En el caso del puerto, en la configuración de pdAdmin aparece el 5432. En la configuración de Docker, en PORTS aparece: 5342:5432. Poniendo 5342 funciona correctamente la conexión a la BBDD.

A continuación, irían estas líneas:

```
engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(bind=engine,autocommit=False,autoflush=False)
Base = declarative_base()
```

Create_engine se usa para crear un objeto **engine** de SQLAlchemy que gestiona la conexión a la base de datos PostgreSQL.

SessionLocal: Una sesión en SQLAlchemy representa una "sesión" de trabajo con la base de datos y proporciona un entorno para ejecutar operaciones de base de datos; sessionmaker se configura con el engine para indicar a qué base de datos se conectará. Los argumentos autocommit=False y autoflush=False son configuraciones adicionales de la sesión.

Base es una clase base para las clases de modelo que se definen utilizando SQLAlchemy. Las clases de modelo representan las tablas en la base de datos y se utilizan para realizar consultas de manera más orientada a objetos. declarative_base crea la clase base y la configura para usar el sistema de mapeo declarativo de SQLAlchemy.

Fichero **models.py** → importaciones:

```
from app.db.database import Base
from sqlalchemy import Column,Integer,String,Boolean,DateTime
from datetime import datetime
```

A continuación, vamos a crear la primera tabla que va a heredar de Base.

Creamos una tabla llamada "user"; con columnas (que son las mismas que aparecen en el fichero schemas.py), pero añadiendo 2 nuevas: correo y estado; que también las añadimos en schemas.py:


```
class User(Base):
    __tablename__ = "userprueba"
    id = Column(Integer,primary_key=True,autoincrement=True)
    nombre = Column(String)
    apellido = Column(String)
    direccion = Column(String)
    telefono = Column(Integer)
    correo = Column(String)
    creacion = Column(DateTime,default=datetime.now,onupdate=datetime.now)
    estado = Column(Boolean)
```

Ahora vamos a crear las tablas→ **main.py** y añadir la importación y la función `create_tables`. El fichero `main.py` quedaría así:

```
from fastapi import FastAPI
import uvicorn
from app.routers import user
from app.db.database import Base, engine

def create_tables():
    Base.metadata.create_all(bind=engine)

create_tables()

app = FastAPI()
app.include_router(user.router)

if __name__=="__main__":
    uvicorn.run("main:app",port=8000,reload=True)
```

Vamos a hacer una consulta→ **database.py** y crear una función que devuelve la sesión de la BBDD. Contenido de `database.py`:

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
```

#URL de la base de datos

```
SQLALCHEMY_DATABASE_URL = "postgresql://odoo:odoo@localhost:5342/fastapi-database"
```

```
engine = create_engine(SQLALCHEMY_DATABASE_URL)
```

```
SessionLocal = sessionmaker(bind=engine,autocommit=False,autoflush=False)
```

```
Base = declarative_base()
```

```
def get_db():
```

```
    db = SessionLocal()
```

```
    try:
```

```
        yield db
```

```
    finally:
```

```
        db.close()
```

Esta función devuelve la sesión de la BBDD.

Después nos vamos a **user.py** e importamos la función que devuelve la BBDD, Depends y la sesión:

```
from fastapi import APIRouter,Depends
```

```
from app.schemas import User,UserId
```

```
from app.db.database import get_db
```

```
from sqlalchemy.orm import Session
```

```
from app.db import models
```

Ahora vamos a modificar la ruta que devuelve los usuarios (obtener_usuarios): esta ruta siempre va a recibir la base de datos: db va a ser igual a la Session que va a depender de lo que devuelve get_db.

Ahora vamos a escribir el código de la consulta (query):

```
@router.get("/")
```

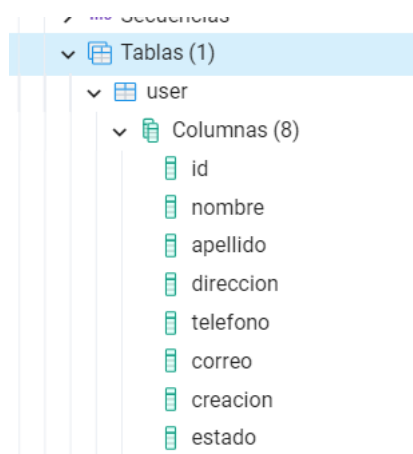
```
def obtener_usuarios(db:Session=Depends(get_db)):
```

```
    data = db.query(models.User).all()
```

```
    print(data)
```

```
    return usuarios
```

Vemos que al ejecutar la aplicación: **python .\main.py** funciona todo correctamente. Vemos que desde pdAdmin se ve la tabla creada con las columnas:



Si vamos a swagger: en obtener usuarios devuelve una lista vacía porque aún no tenemos datos en la tabla de la BBDD

Vamos a crear otra tabla (modelo) en la BBDD: ventas. Un usuario puede tener varias ventas.

Vamos a agregar también la contraseña al usuario → modificar la Class User del fichero models.py

Fichero **models.py**, modelo original:

```
class User(Base):
    __tablename__ = "user"
    id = Column(Integer, primary_key=True, autoincrement=True)
    nombre = Column(String)
    apellido = Column(String)
    direccion = Column(String)
    telefono = Column(Integer)
    correo = Column(String)
    creacion = Column(DateTime, default=datetime.now, onupdate=datetime.now)
    estado = Column(Boolean)
```

Modelo modificado:

```
class User(Base):
    __tablename__ = "user"
    id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String)
    password = Column(String)
    nombre = Column(String)
    apellido = Column(String)
    direccion = Column(String)
    telefono = Column(Integer)
    correo = Column(String)
    creacion = Column(DateTime, default=datetime.now, onupdate=datetime.now)
    estado = Column(Boolean)
```

Como hemos hecho este cambio en el modelo, tenemos que eliminar la tabla de la BBDD.

Lo siguiente que vamos a hacer es crear el modelo “venta”, dentro de models.py: campo id, la relación se llama usuario_id; hace referencia a la tabla usuario (cambiar el nombre de la tabla user→usuario).

Hay que realizar una importación para crear la Foreign Key (ver la imagen de abajo)

```
class Venta(Base):
    __tablename__ = "venta"
    id = Column(Integer, primary_key=True, autoincrement=True)
    usuario_id = Column(Integer, ForeignKey("usuario.id", ondelete="CASCADE"))
    venta = Column(Integer)
    ventas_productos = Column(Integer)
```

Después de crear la tabla venta, hay que crear la relación en la tabla usuario: agregar el campo venta (relationship): venta hace referencia a la clase (añadir la importación relationship).

El fichero **models.py** queda así:

```
app > db > models.py > Venta
```

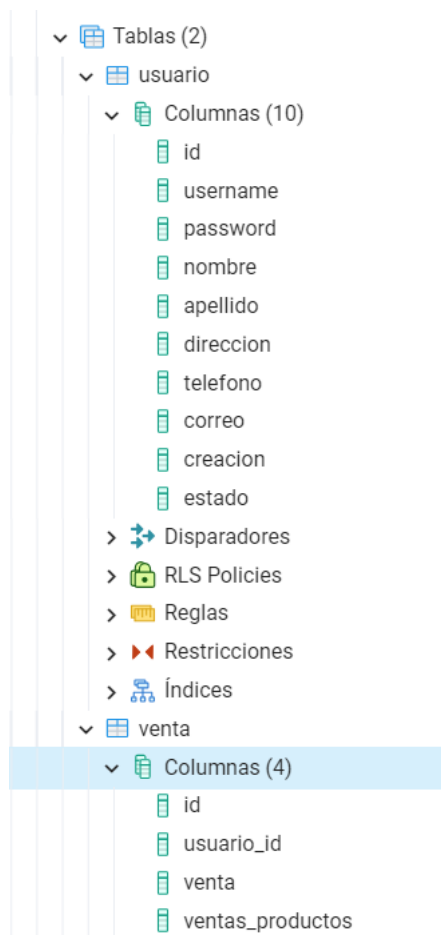
```
1  from app.db.database import Base
2  from sqlalchemy import Column,Integer,String,Boolean,DateTime
3  from datetime import datetime
4  from sqlalchemy.schema import ForeignKey
5  from sqlalchemy.orm import relationship
```

```
15 class User(Base):
16     __tablename__ = "usuario"
17     id = Column(Integer,primary_key=True,autoincrement=True)
18     username = Column(String)
19     password = Column(String)
20     nombre = Column(String)
21     apellido = Column(String)
22     direccion = Column(String)
23     telefono = Column(Integer)
24     correo = Column(String)
25     creacion = Column(DateTime,default=datetime.now,onupdate=datetime.now)
26     estado = Column(Boolean)
27     venta = relationship("Venta",backref="usuario",cascade="delete,merge")
```

```
class Venta(Base):
    __tablename__ = "venta"
    id = Column(Integer,primary_key=True,autoincrement=True)
    usuario_id = Column(Integer,ForeignKey("usuario.id",ondelete="CASCADE"))
    venta = Column(Integer)
    ventas_productos = Column(Integer)
```

Ahora hay que volver a ejecutar el proyecto (Python .\main.py) y ver si se han creado las tablas.

Si vamos a pdAdmin4, vemos que se han creado las tablas correctamente:



6. OPERACIONES CON LA BBDD

Vamos a ver cómo se insertan datos en la tabla de la BBDD.

En primer lugar, en el fichero `user.py`, en el endpoint de añadir usuarios (`crear_usuario`), comentar la línea `usuarios.append(usuario)`.

Otra cosa que hay que hacer es modificar El modelo `User`: estado por defecto a `False`. También hay que modificar el campo `correo` y `username` (`unique`).

```
class User(Base):
    __tablename__ = "usuario"
    id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, unique=True)
    password = Column(String)
    nombre = Column(String)
    apellido = Column(String)
    direccion = Column(String)
    telefono = Column(Integer)
    correo = Column(String, unique=True)
    creacion = Column(DateTime, default=datetime.now, onupdate=datetime.now)
    estado = Column(Boolean, default=False)
    venta = relationship("Venta", backref="usuario", cascade="delete, merge")
```

Ahora hay que eliminar las tablas de las BBDD.

Lo siguiente es modificar el esquema en el fichero schemas.py y añadir los campos username y password. Se borra el id porque es autoincremental; se añade: username, password:

```
class UpdateUser(BaseModel): #Schema
    username:str = None
    password:str = None
    nombre:str = None
    apellido:str = None
    direccion:str = None #Parámetro opcional; es necesario importar Optional
    telefono:int = None
    correo:str = None
```

Ahora vamos a ver cómo se crea un usuario: fichero user.py, función crear_usuario →

Función original:

```
25 @router.post("/")
26 def crear_usuario(user:User):
27     usuario = user.model_dump()
28     # usuarios.append(usuario)
29     return {"Respuesta": "Usuario creado!!"}
```

Función con las modificaciones:

- Conexión a la BBDD db
- Crear la variable nuevo_usuario: al endpoint le llegan los datos del esquema de usuario que

hay en schemas.py

```
25 @router.post("/")
26 def crear_usuario(user:User, db:Session=Depends(get_db)):
27     usuario = user.model_dump()
28     # user es lo que llega mediante el esquema del body
29     # usuarios.append(usuario)
30     nuevo_usuario = models.User(
31         username = usuario["username"],
32         password = usuario["password"],
33         nombre = usuario["nombre"],
34         apellido = usuario["apellido"],
35         direccion = usuario["direccion"],
36         telefono = usuario["telefono"],
37         correo = usuario["correo"],
38     )
39     db.add(nuevo_usuario)
40     db.commit()
41     db.refresh(nuevo_usuario)
42     return {"Respuesta": "Usuario creado!!"}
43
```

Para probarlo: **Python .\main.py**

Vemos que se han vuelto a crear las tablas de la BBDD (desde pgAdmin). Ahora desde **swagger**, vamos a crear un nuevo usuario:

Vemos el esquema definido en el fichero schemas.py:

Request body **required**

```
{
  "username": "string",
  "password": "string",
  "nombre": "string",
  "apellido": "string",
  "direccion": "string",
  "telefono": 0,
  "correo": "string",
  "creacion": "2023-12-30T11:52:11.323986"
}
```

Para crear el usuario, introducir en el esquema de arriba los datos a añadir; después ir a la BBDD y comprobar que el usuario se ha creado correctamente:

The screenshot shows a web application interface with a top bar containing 'Consulta' and 'Historial de Consultas'. Below this is a text area with a SQL query: `1 select * from usuario`. To the right is a 'Scratch Pad' tab. Below the query area is a 'Data Output' section with a table of results. The table has columns: id, username, password, nombre, apellido, direccion, telefono, correo, creacion, and estado. The first row shows data for a user with id 1, username 'cris', password '1234', nombre 'cris', apellido 'silvan', direccion 'valladolid', telefono '123456789', correo 'yo@gmail.com', creacion '2023-12-30 11:57:45.314917', and estado 'false'.

id	username	password	nombre	apellido	direccion	telefono	correo	creacion	estado
1	cris	1234	cris	silvan	valladolid	123456789	yo@gmail.com	2023-12-30 11:57:45.314917	false

El campo estado está a False. Si volvemos a intentar añadir el mismo usuario aparece un Internal Server Error porque el correo y username tienen la restricción unique.

Ahora vamos a modificar el endpoint de obtener un usuario, que llega como query parameter. El código original sería:

```
@router.get("/{user_id}")
def obtener_usuario(user_id:int):
    for user in usuarios:
        if user["id"] == user_id: #Acceder al id de user (tipo dict) y comparar con user_id que se pasa como quer
            return{"usuario":user}
    return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
```

Función con las modificaciones:

- Conexión a la BBDD db
- Tenemos que realizar una consulta mediante el ORM: db.query. Luego usamos filter; vemos que filter es una función que filtra mediante un criterio. En este caso, vamos a filtrar por el id que llega como query_parameter. El id es único.

```
44 @router.get("/{user_id}")
45 def obtener_usuario(user_id:int, db:Session=Depends(get_db)):
46     usuario = db.query(models.User).filter(models.User.id == user_id).first()
47     if not usuario:
48         return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
49     return usuario
```

Para probarlo, desde el navegador: 127.0.0.1:80 y vemos desde pgAdmin el id del usuario añadido es (en mi caso es el 1) → 127.0.0.1:8000/docs; probamos el endpoint de obtener usuario:

Name	Description
user_id * required integer (path)	<input type="text" value="1"/>

Execute

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/user/1' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/user/1
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "username": "cris", "nombre": "cris", "telefono": 123456789, "correo": "yo@gmail.com", "estado": false, "id": 1, "password": "1234", "apellido": "silvan", "direccion": "valladolid", "creacion": "2023-12-30T11:57:45.314917" }</pre>

Nos devuelve todos los campos; pero puede que sólo queramos devolver campos específicos (por ejemplo, es probable que no queramos devolver la contraseña).

Para ello, podemos definir un esquema que va a devolver; además en el endpoint añadimos `response_model` que es un parámetro que recibe; es un schema que hay que crear en el fichero `schemas.py` (`ShowUser`), con los campos que queremos que devuelva:

Modelo `ShowUser`:

```
class ShowUser(BaseModel): #Schema para devolver datos de un usuario
    username:str
    nombre:str
    correo:str
```

A continuación, en el fichero **user.py**, importamos el esquema ShowUser:

```
from app.schemas import User, ShowUser
```

Y lo añadimos en la ruta del endpoint, con el parámetro `response_model`. Con esto indicamos que esta API tiene que devolver los datos indicados en `response_model` (username, nombre y correo).

```
@router.get("/{user_id}", response_model=ShowUser)
def obtener_usuario(user_id:int, db:Session=Depends(get_db)):
    usuario = db.query(models.User).filter(models.User.id == user_id).first()
    if not usuario:
        return {"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
    return usuario
```

Para probarlo, desde el navegador: 127.0.0.1:80 y vemos que el id del usuario añadido es 1 → 127.0.0.1:8000/docs y probamos el endpoint de obtener usuario. Aparece Internal Server Error porque hay que indicar a qué ORM nos tenemos que conectar → modificar el schema a devolver:

```
class ShowUser(BaseModel): #Schema para devolver datos de un usuario
    username:str
    nombre:str
    correo:str
    class Config():
        orm_mode = True
```

Probar de nuevo; vemos que sólo devuelve los datos que le hemos indicado:

Name	Description
user_id * required integer (path)	<input type="text" value="1"/>

Execute

Responses

Curl

```
curl -X 'GET' \
'http://127.0.0.1:8000/user/1' \
-H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/user/1
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "username": "cris", "nombre": "cris", "correo": "yo@gmail.com" }</pre>

Vemos que aparece un warning en el VSC: **orm_mode has been renamed to from_attributes**.

```
INFO: Started reloader process [11812] using StatReload
C:\Users\Cristina\Desktop\Cursos_23_24\CRUD_FastAPI_Udemy\Proyecto_curso\venv\Lib\site-packages\pydantic\_internal\_config.py:321: UserWarning: Valid config keys have changed in V2:
* 'orm_mode' has been renamed to 'from_attributes'
warnings.warn(message, UserWarning)
```

Por ello, cambiamos esos valores en el esquema ShowUser:

```
class ShowUser(BaseModel): #Schema para devolver datos de un usuario
    username:str
    nombre:str
    correo:str
    class Config():
        orm_mode = True
```

```
class ShowUser(BaseModel): #Schema para devolver datos de un usuario
    username:str
    nombre:str
    correo:str
    class Config():
        from_attributes = True
```

Para probarlo, desde el navegador: 127.0.0.1:80 y vemos que el id del usuario añadido es 1 → 127.0.0.1:8000/docs y probamos el endpoint de obtener usuario. Funciona correctamente: vemos que sólo devuelve los datos que le hemos indicado y ya no aparece el warning.

Se puede probar a quitar y poner campos en el schema a devolver y a probar la salida del endpoint. Vemos que response_model realiza un filtro de las columnas a mostrar.

Podemos modificar **user.py**: quitar ruta1

Vamos a modificar el endpoint de obtener_usuarios; si lo probamos, vemos que devuelve una lista vacía porque aparece: return usuarios y eso es una lista que ya no usamos; está vacía.

Definimos data e indicamos return data:

```
@router.get("/")
def obtener_usuarios(db:Session=Depends(get_db)):
    data = db.query(models.User).all()
    print(data)
    return data
```

Si queremos devolver una lista de usuarios con el esquema ShowUser → en el endpoint de obtener_usuarios añadir response_model. Aquí se devuelve una lista de usuarios, es necesario importar List para luego usarlo en response_model. Estos son los cambios en user.py:

```
from typing import List
```

```
16 @router.get("/", response_model=List(ShowUser))
17 def obtener_usuarios(db:Session=Depends(get_db)):
18     data = db.query(models.User).all()
19     print(data)
20     return data
21
```

Si hacemos lo anterior, aparece un error y no se puede recargar la página de los endpoints. Solución: dejarlo como estaba; con esto se devuelve una lista de usuarios con todas las columnas:

```
@router.get("/")
def obtener_usuarios(db:Session=Depends(get_db)):
    data = db.query(models.User).all()
    print(data)
    return data
```

A continuación, vamos a modificar el endpoint de eliminar usuario. Tenemos que filtrar el usuario a eliminar. Modificaciones que realizar:

- Conexión a la BBDD: db
- Tenemos que realizar una consulta mediante el ORM: db.query. Usamos filter:

```
@router.delete("/{user_id}")
def eliminar_usuario(user_id:int):
    for index,user in enumerate(usuarios): #Necesitamos saber el índice y el valor para ver
        if user["id"] == user_id:
            usuarios.pop(index)
            return {"Respuesta": "Usuario eliminado correctamente"}
    return {"Respuesta": "Usuario NO encontrado"}
```

```
@router.delete("/{user_id}")
def eliminar_usuario(user_id:int, db:Session=Depends(get_db)):
    usuario = db.query(models.User).filter(models.User.id == user_id).first()
    if not usuario:
        return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
    usuario.delete(synchronize_session=False)
    db.commit()
    return{"respuesta": "Usuario eliminado corretamente"}
```

Para probarlo: 127.0.0.1:8000/docs → obtener todos los usuarios, ver el id y eliminar el usuario indicando su id.

Aparece Internal Server Error→

```
AttributeError: 'User' object has no attribute 'delete'
```

Para solucionarlo, quitar first del filter y añadirlo en if not usuario:

```
@router.delete("/{user_id}")
def eliminar_usuario(user_id:int, db:Session=Depends(get_db)):
    usuario = db.query(models.User).filter(models.User.id == user_id)
    if not usuario.first():
        return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
    usuario.delete(synchronize_session=False)
    db.commit()
    return{"respuesta": "Usuario eliminado corretamente"}
```

Ahora si lo probamos, vemos que funciona correctamente:

user_id ★ required

integer
(path)

Execute

Responses

Curl

```
curl -X 'DELETE' \
  'http://127.0.0.1:8000/user/1' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/user/1
```

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{ "respuesta": "Usuario eliminado corretamente" }</pre></div><div><div>Response headers</div><div><pre>content-length: 46 content-type: application/json date: Mon,01 Jan 2024 10:36:26 GMT server: uvicorn</pre></div></div></div>

Comprobamos que en la BBDD se ha borrado el usuario (desde pgAdmin).

Probar a crear 2 usuarios, obtener usuarios, ver en la BBDD que se han creado y eliminar uno de ellos

Ahora vamos a ver cómo actualizar usuarios: para ello, es necesario implementar el método patch y borrar el método put porque con put hay que modificar todas las columnas (username, password...). Sin embargo, con el patch tenemos la posibilidad de modificar sólo alguna columna.

Método con el código original:

```
@router.put("/{user_id}")
def actualizar_usuario(user_id:int, updateUser:User):
    for index,user in enumerate(usuarios): #Necesitamos saber el índice y el valor para ver si el user_id es = al que estamos recorriendo
        if user["id"] == user_id:
            usuarios[index]["id"] = updateUser.model_dump()["id"]
            usuarios[index]["nombre"] = updateUser.model_dump()["nombre"]
            usuarios[index]["apellido"] = updateUser.model_dump()["apellido"]
            usuarios[index]["direccion"] = updateUser.model_dump()["direccion"]
            usuarios[index]["telefono"] = updateUser.model_dump()["telefono"]
            return {"Respuesta": "Usuario actualizado correctamente"}
    return {"Respuesta": "Usuario NO encontrado"}
```

Lo primero que hay que hacer es cambiar put por patch. Comprobar que aparece en la página de los endpoints.

Vamos a hacer un nuevo esquema donde los parámetros no sean obligatorios: UpdateUser; para ello, ponemos sus parámetros a None. La razón es que, si usamos el esquema User, como todos los campos son obligatorios, al intentar pasar sólo los campos a modificar, da error:

```
#User Model para Update
class UpdateUser(BaseModel): #Schema
    username:str = None
    password:str = None
    nombre:str = None
    apellido:str = None
    direccion:str = None #Parámetro opcional; es necesario importar Optional
    telefono:int = None
    correo:str = None
```

Hay que importar el modelo en user.py y en el endpoint hay que indicar UpdateUser.

```
from app.schemas import User, ShowUser, UpdateUser
```



```
@router.patch("/{user_id}")
def actualizar_usuario(user_id:int, updateUser:UpdateUser, db:Session=Depends(get_db)):
    usuario = db.query(models.User).filter(models.User.id == user_id)
    if not usuario.first():
        return{"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
    usuario.update(updateUser.model_dump(exclude_unset=True))
    db.commit()
    return {"Respuesta": "Usuario actualizado correctamente"}
```

exclude_unset es un parámetro que indica que sólo se actualicen los campos que están llegando, no todos.

Probarlo de nuevo → endpoints → obtener usuarios, BBDD y vemos el id del usuario. Queremos modificar sólo el username

PATCH /user/{user_id} Actualizar Usuario

Parameters

Name	Description
user_id * required	
integer	3
(path)	

Request body required

```
{
  "username": "camela"
}
```

Consulta

Historial de Consultas

1

select * from usuario

Data Output

Mensajes

Notificaciones

+

📄

▼

📋

▼

🗑️

🗄️

⬇️

📈

	id [PK] integer	username character varying	password character varying	nombre character varying
1	3	camela	cris23	Cris

Funciona correctamente; indicando en el JSON sólo los campos que queremos actualizar. Si sólo pasamos un campo, hay que quitar la coma del final.

Últimos pasos: eliminar el endpoint obtener_usuario_json, el modelo UserId y la importación de ese modelo en el fichero user.py. También borrar la lista de usuarios de user.py

Los endpoints quedarían de esta forma:

FastAPI

0.1.0

OAS 3.1

</openapi.json>

Users

GET

/user/

Obtener Usuarios

POST

/user/

Crear Usuario

GET

/user/{user_id}

Obtener Usuario

DELETE

/user/{user_id}

Eliminar Usuario

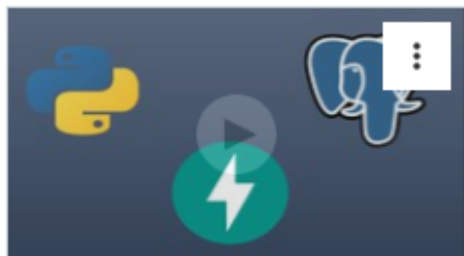
PATCH

/user/{user_id}

Actualizar Usuario

7. BIBLIOGRAFÍA

Apuntes del curso de Udemmy:



CRUD básico usando FastAPI

Andres Rojas

100 % completado



Deja una calificación