



IES RIBERA DE CASTILLA

PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones Multiplataforma

Informática y Comunicaciones

Library of Ohara (Python API Version)

Año: 2025

Fecha de presentación: 11/02/2025

Nombre y Apellidos: Jaime González Bravo

Email: jaime.gonbra@educa.jcyl.es

Índice

Introducción:	3
Estado del Arte:	3
Arquitectura de Microservicios (API):	3
API (Interfaz de Programación de Aplicaciones):	3
Estructura de una API:	3
Formas de crear una API (FastAPI):	4
Descripción general de proyecto:	4
Objetivo:	4
Entorno de Trabajo:	4
Documentación técnica:	4
Análisis del sistema:	4
Diseño de la BD:	5
Implementación:	7
Pruebas:	14
Despliegue de la Aplicación:	20
Manuales	21
Manual de usuario:	21
Manual de instalación:	22
Conclusiones:	22
Bibliografía:	22

Introducción:

El proyecto consiste en un (intento) de API funcional en el que se trabajen con una base de datos relacional (en este caso PostgreSQL) dando servicio a diferentes funcionalidades que se verán en el mismo.

Como resumen, quería comprobar mis conocimientos de Python para probarme a mi mismo y conseguir una API parecida a la que quiero hacer en mi TFG.

Estado del Arte:

Arquitectura de Microservicios (API):

Es un método de desarrollo de aplicaciones de software que funciona como un conjunto de pequeños servicios que se ejecutan de manera independiente y autónoma proporcionando una funcionalidad de negocio completa. Los microservicios se comunican entre sí a través de APIs y cuentan con un sistema de almacenamiento propio (una base de datos). De esta forma, se facilita el uso y desarrollo de estas, ya no es una aplicación de gran tamaño sino varios mini proyectos que se encargan de una cosa en específico.

API (Interfaz de Programación de Aplicaciones):

es un conjunto de protocolos que se usa para diseñar e integrar el software de las aplicaciones. Permiten que tus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Por tanto, simplifica el desarrollo y permite ahorrar tanto tiempo como dinero.

Estructura de una API:

La estructura básica de una API suele seguir ciertas convenciones para permitir la interacción de clientes y servidores.

Como protocolo, se utiliza el de HTTP/HTTPS que es el estándar para todas las comunicaciones, siendo el HTTPS el mejor ya que es más seguro.

Los métodos más utilizados son:

GET: petición para obtener la información del servidor.

POST: petición para enviar datos (se suele utilizar para crear nuevos recursos).

PUT: petición para enviar datos (se suele utilizar para actualizar recursos).

PATCH: similar que el PUT, pero para un solo campo.

DELETE: petición para eliminar un recurso.

Las partes más características de una API son:

El esquema que hace referencia al protocolo (https://).

El dominio que es la dirección del servidor donde se encuentra la API (localhost:8000)

La ruta que es específica para cada recurso que la API maneja (/api/usuario/all)

Los parámetros, que ya sean los que se emplean en la propia ruta o bien, los parámetros que se utilizan en los cuerpos de las peticiones (body).

Formas de crear una API (FastAPI):

FastAPI es uno de los frameworks más populares y rápidos a la hora de crear APIs en Python debido a su rendimiento y facilidad. Para poder utilizarlo, requieres instalar el servidor ASGI (servidor de aplicaciones asíncronas) como el denominado Uvicorn. Además, este framework, implementa la documentación interactiva del api conocido como Swagger que explicará y nos dejará probar todas las APIs que hayamos creado.

Descripción general de proyecto:

Objetivo:

La idea principal del proyecto es crear una API similar a la que quiero utilizar para el TFG utilizando los conocimientos aprendidos durante las clases. Además de así practicar con Python.

Entorno de Trabajo:

Como Gestor de base de datos, he utilizado PostgreSQL ya que tiene buena sinergia trabajando con Python y, además, utilizo Docker para levantar el servicio. De esta manera, puedo ejecutar y lanzar el servicio en cualquier momento.

Para poder ver las tablas y los cambios realizados, utilizo PGAdmin que es la interfaz de usuario que se conecta a PostgreSQL para mostrar todos los campos de las tablas.

Como lenguaje de programación, he utilizado Python debido a su fácil uso además de Visual Studio Code como IDE para poder ejecutar el código. Como framework empleado para la creación de la API, empleo el ya denominado FastAPI.

Documentación técnica:

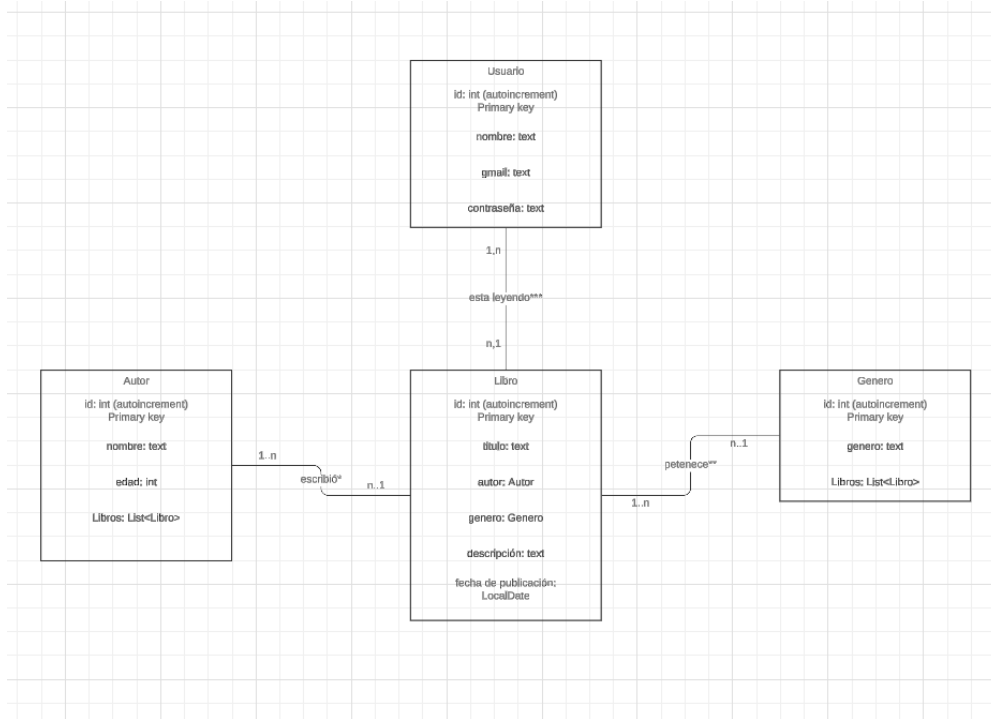
Análisis del sistema:

Debido a la falta de tiempo y en algunos casos, falta de conocimientos, no dispongo de aplicación móvil ni de sistema de autenticación. Lo que el usuario puede llegar a hacer es la de

creación, modificación visualización y eliminación de Usuarios, Libros, Géneros y Autores. Es bastante básico, pero es lo que me ha dado tiempo a realizar.

Diseño de la BD:

Esta es la idea de la cual partí



Finalmente, acabo así.

```

class Usuario(Base):
    __tablename__ = "usuario"
    id = Column(Integer, primary_key=True, autoincrement=True)
    nombre=Column(String)
    gmail=Column(String)
    contrasenna= Column(String)

class Genero(Base):
    __tablename__ = "genero"
    id = Column(Integer, primary_key=True, autoincrement=True)
    genero = Column(String)
    libros = relationship("Libro", backref="genero", cascade="all, delete-orphan")

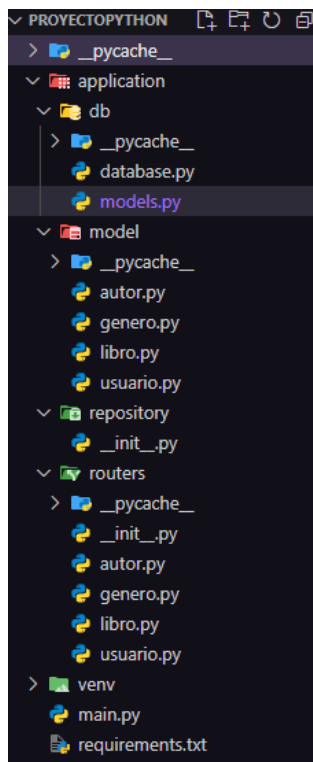
class Autor(Base):
    __tablename__ = "autor"
    id = Column(Integer, primary_key=True, autoincrement=True)
    nombre = Column(String)
    edad = Column(Integer)
    libros = relationship("Libro", backref="autor", cascade="all, delete-orphan")

class Libro(Base):
    __tablename__ = "libro"
    id = Column(Integer, primary_key=True, autoincrement=True)
    titulo= Column(String)
    genero_id = Column(Integer, ForeignKey("genero.id", ondelete = "CASCADE"))
    autor_id = Column(Integer, ForeignKey("autor.id", ondelete = "CASCADE"))
    descripcion= Column(String)
    fecha_publicacion= Column(DateTime, default=datetime.now, onupdate=datetime.now)

```

Como se puede ver en la imagen, lo más complicado fue el tema de las relaciones entre género, autor y libro respectivamente. Porque, aunque solo se guarde en la base de datos en libro los ids del autor y del género, cada vez que visualizo uno autor o un género, muestro también los libros a los que hace referencia dicha relación. (créeme, ha sido lo más complicado).

Implementación:



Aunque al principio parece todo muy enrevesado, es una organización “simple”.

Por un lado, tenemos la carpeta db con todo lo relacionado con la base de datos. En ella, tenemos el fichero models con el modelo de base de datos y el de database que configura todo lo necesario para la conexión a base de datos.

```

from application.db.database import Base
from sqlalchemy import Column,Integer,String,DateTime
from datetime import datetime
from sqlalchemy.schema import ForeignKey
from sqlalchemy.orm import relationship

class Usuario(Base):
    __tablename__ = "usuario"
    id = Column(Integer,primary_key=True,autoincrement=True)
    nombre=Column(String)
    gmail=Column(String)
    contrasenna= Column(String)

class Genero(Base):
    __tablename__ = "genero"
    id = Column(Integer,primary_key=True,autoincrement=True)
    genero = Column(String)
    libros = relationship("Libro", backref="genero", cascade="all, delete-orphan")

class Autor(Base):
    __tablename__ = "autor"
    id = Column(Integer,primary_key=True,autoincrement=True)
    nombre = Column(String)
    edad = Column(Integer)
    libros = relationship("Libro", backref="autor", cascade="all, delete-orphan")

class Libro(Base):
    __tablename__ = "libro"
    id = Column(Integer,primary_key=True,autoincrement=True)
    titulo= Column(String)
    genero_id = Column(Integer, ForeignKey("genero.id", ondelete = "CASCADE"))
    autor_id = Column(Integer, ForeignKey("autor.id", ondelete = "CASCADE"))
    descripcion= Column(String)
    fecha_publicacion= Column(DateTime, default=datetime.now, onupdate=datetime.now)

```

Utilizo la biblioteca de sqlalchemy que proporciona herramientas para interactuar con bases de datos relacionales de manera eficiente y flexible (parecido a un ORM).


```

application > db > database.py > ...
1  from sqlalchemy import create_engine
2  from sqlalchemy.ext.declarative import declarative_base
3  from sqlalchemy.orm import sessionmaker
4
5  SQLALCHEMY_DATABASE_URL = "postgresql://odoo:odoo@localhost:5342/library_of_ohara"
6  engine = create_engine(SQLALCHEMY_DATABASE_URL)
7  SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
8  Base = declarative_base()
9
10
11 def get_db():
12     db = SessionLocal() # Crear una nueva sesión
13     try:
14         yield db # Devuelve la sesión para su uso
15     finally:
16         db.close() # Cierra la sesión después de usarla
17

```

Siguiendo por orden, llega la carpeta modelos, en el que están como su nombre indica, las clases con los modelos que se van a utilizar para el correcto uso de la aplicación. Dichas clases son una imagen y semejanza con el de la base de datos añadiendo algunas cosas.

```

from datetime import datetime
# from typing import Optional
from pydantic import BaseModel

class Usuario (BaseModel):
    id: int
    nombre: str
    gmail: str
    contrasenna: str

class UsuarioDTO(BaseModel):
    nombre: str
    gmail: str
    contrasenna: str

```

Como podemos ver en la imagen, tenemos la clase usuario que es idéntica a la de la bd y la clase UsuarioDTO que me sirve a mi a la hora de insertar un nuevo usuario o modificar uno ya existente. Utilizo pydantic que es una biblioteca de Python que se utiliza principalmente para la validación de datos y para la creación de modelos de datos de manera sencilla, eficiente y segura.

En el fichero género, también tenemos una clase Genero que es idéntica al de la bd haciendo referencia a la lista con los libros relacionado y la del DTO para facilitarme el insertado y el modificado de datos.

```

from datetime import datetime
from typing import List

# from typing import Optional
from pydantic import BaseModel
from application.model.libro import Libro

class Genero(BaseModel):
    id: int
    genero: str
    libros: List[Libro] = []

    """class Config:
    |     orm_mode = True"""

class GeneroDTO(BaseModel):
    genero: str

```

El fichero de Autor es muy parecido al de género. En él, tenemos tanto la clase autor que hace referencia al de la base de datos y el DTO.

```

from datetime import datetime
from typing import List
# from typing import Optional
from pydantic import BaseModel
from application.model.libro import Libro

class Autor(BaseModel):
    id:int
    nombre:str
    edad: int
    libros: List[Libro]=[]

    """class Config:
    |     orm_mode = True"""

class AutorDTO (BaseModel):
    nombre:str
    edad: int

```

Por ultimo y no menos importante, tenemos el fichero de Libro con las clases de libro y libroDTO. En dicha clase, tenemos el id que hace referencia tanto al autor y al género al que pertenece dicho libro.

```
from datetime import datetime
from pydantic import BaseModel

class Libro(BaseModel):
    id: int
    titulo: str
    autor_id: int
    genero_id: int
    descripcion: str
    fecha_publicacion: datetime

    """class Config:
    | orm_mode = True"""

class LibroDTO(BaseModel):
    titulo: str
    autor_id: int
    genero_id: int
    descripcion: str
    fecha_publicacion: datetime
```

Siguiendo con la primera imagen, tendremos el archivo repository en el que se encapsula la lógica de acceso a la base de datos. En él, se debería de meter toda la funcionalidad que realiza nuestra API (cosa interesante de saber antes de realizar este proyecto). Para simplificar, todo ese código lo tengo en los routers.

Los routers están todos juntos y representan tanto la dirección como lo que van a realizar dichos microservicios. Para resumir, solo mostraré uno ya que el resto es bastante parecido la verdad.

Como podemos ver en la imagen, inicia con el router que sirve para indicarle a la aplicación que este fichero “controla” todas las acciones de autores. Es solo una manera de organizar y no tener todo en un mismo fichero. Tal y como hemos visto con anterioridad, tenemos cada uno de los métodos de las APIs (get,post,put,delete).

Como primer método, es el método get all que recibe todos los autores de la bd. (curiosidad, en este método siempre me mostraban los libros, en los otros siempre me daban problemas.) también está el de getByID en el que me busca en base de datos un autor que tenga dicho id que pasamos por parámetro. Si no existe, le enviamos al usuario un mensaje de que no existe dicho autor y si existe, le pasamos el autor.

```

from fastapi import APIRouter, Depends
from application.model.autor import Autor, AutorDTO
from application.model.libro import Libro
from application.db.database import get_db
from sqlalchemy.orm import Session
from application.db import models
from typing import List

router = APIRouter(prefix="/autor", tags=["Autores"])

@router.get("/all", response_model=List[Autor])
def getAutor(database: Session = Depends(get_db)):
    autores = database.query(models.Autor).all()
    return autores

@router.get("/{id}")
def getAutorByID(id: int, database: Session = Depends(get_db)):
    autor = autorByID(id, database)
    if not autor:
        return {"Respuesta": "Error al buscar genero: No existe dicho genero."}
    else:
        return {"autor": autor}

@router.post("/add")
def addAutor(autorDTO: AutorDTO, database: Session = Depends(get_db)):
    autor = models.Autor(nombre=autorDTO.nombre, edad=autorDTO.edad)
    if existeAutor(autor.nombre, database):
        return {"Respuesta": "Error al insertar: autor ya existente en bd."}
    else:
        database.add(autor)
        database.commit()
        database.refresh(autor)
        return {"Respuesta": "autor creado.", "Autor": autor}

```

Como vemos en la imagen, tenemos el metodo post para añadir un autor nuevo.

Para ello, compruebo si existe un autor con dicho nombre. En caso de que exista, no permite insertarlo, en caso contrario, lo añade a la bd y muestra el mensaje de añadido.

```

@router.patch("/{id}/update")
def updateAutor(id: int, autorDTO: AutorDTO, database: Session = Depends(get_db)):
    autor = autorByID(id, database)
    if not autor:
        return {"Respuesta": "Error al borrar el autor: No existe dicho autor."}
    else:
        autor.nombre = autorDTO.nombre
        autor.edad = autor.edad
        database.commit()
        return {
            "Respuesta": "autor modificado con éxito.",
            "genero": autor,
        }

@router.delete("/{id}/delete")
def deleteAutor(id: int, database: Session = Depends(get_db)):
    autor = autorByID(id, database)
    if not autor.first():
        return {"Respuesta": "Error al borrar el autor: No existe dicho autor."}
    else:
        database.delete(autor)
        database.commit()
        return {"Respuesta": "autor eliminado con éxito."}

def existeAutor(nombre, database: Session):
    data = database.query(models.Autor).all()
    existe = False
    for autorDB in data:
        if autorDB.nombre == nombre:
            existe = True
    return existe

```

Como vemos en la imagen, tenemos el metodo patch para modificar los campos de un autor. Para ello, compruebo si existe el autor con dicho id. En caso de que no exista, no lo modifica y manda un mensaje de error. En caso de que si, los campos del autor se sobrescriben con los que hemos enviado en el body de la operación.

En el metodo delete para borrar un autor. Al igual que el update, buscamos si existe dicho autor y en caso de que si, lo borra y como tengo en la base de datos el campo de delete-cascade (en teoria) deberia de borrar los libros a los que hace referencia dicho autor.

Por ultimo, tenemos una función para comprobar si existe un autor con dicho nombre. Devolverá true si existe o false en caso contrario.

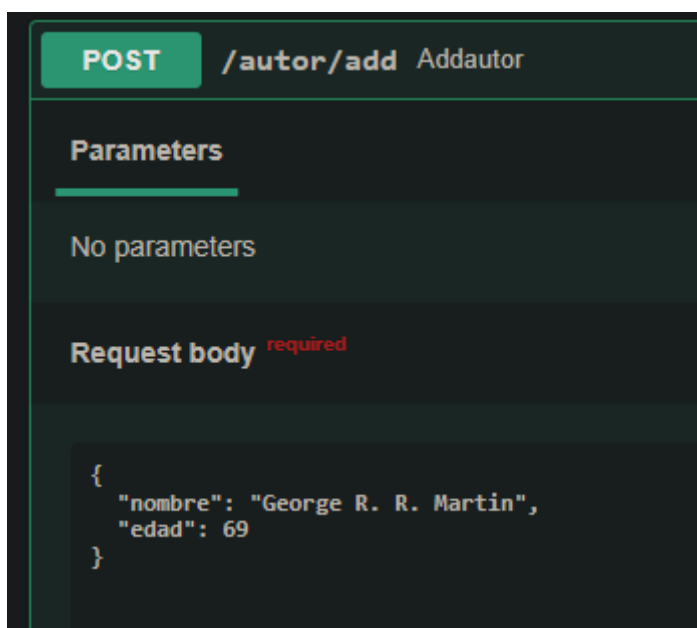
```
def autorByID(id: int, database: Session):
    autorBD = database.query(models.Autor).filter(models.Autor.id == id).first()

    if autorBD:
        autor = Autor(
            id=autorBD.id,
            nombre=autorBD.nombre,
            edad=autorBD.edad,
            libros=[
                Libro(
                    id=libro.id,
                    titulo=libro.titulo,
                    autor_id=libro.autor_id,
                    genero_id=libro.genero_id,
                    descripcion=libro.descripcion,
                    fecha_publicacion=libro.fecha_publicacion,
                )
                for libro in autorBD.libros
            ],
        )
        return autor
    else:
        return None
```

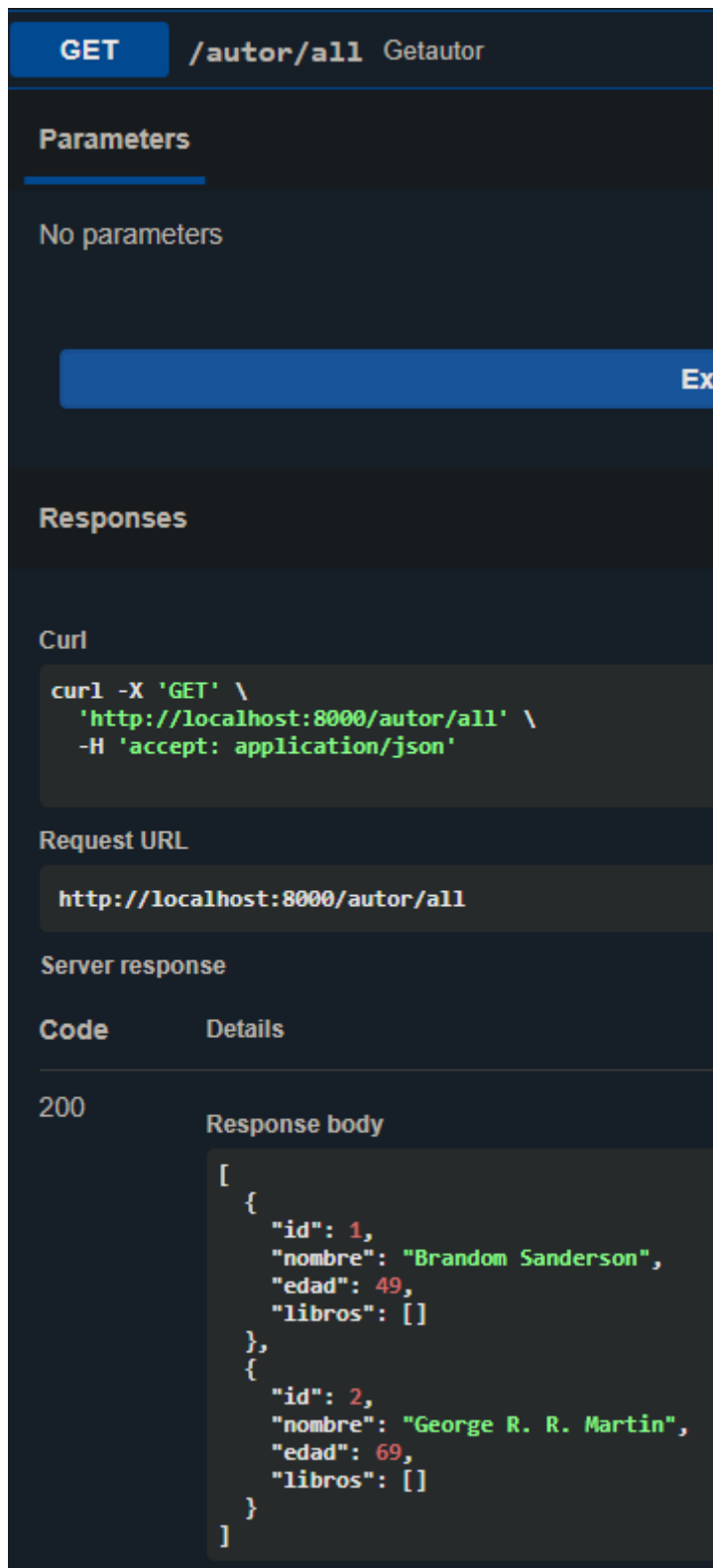
Por último, tenemos la guinda del pastel, la función que me devuelve el autor según su id. (es la que más quebraderos de cabeza me ha dado.) como podemos ver, buscamos el autor por id, y nos devuelve el autor, aquí tenía problemas porque no se me mostraba la lista de libros o me la mostraba vacía. Para solucionarlo, tuve que instanciar el libro y ponerle todos sus valores.

Pruebas:

Insertar un nuevo autor:



Le damos a ejecutar y nos insertará dicho autor (siempre y cuando, los datos estén bien.)



GET /autor/all Getautor

Parameters

No parameters

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8000/autor/all' \
  -H 'accept: application/json'
```

Request URL

http://localhost:8000/autor/all

Server response

Code	Details
200	<p>Response body</p> <pre>[{ "id": 1, "nombre": "Brandom Sanderson", "edad": 49, "libros": [] }, { "id": 2, "nombre": "George R. R. Martin", "edad": 69, "libros": [] }]</pre>

Después, creamos un genero

POST

/genero/add Addgeneros

Parameters

No parameters

Request body required

```
{  
  "genero": "fantasia"  
}
```


GET

/genero/allGetgeneros

Parameters

No parameters

Responses

Curl

curl -X 'GET' \n'http://localhost:8000/genero/all' \n-H 'accept: application/json'

Request URL

http://localhost:8000/genero/all

Server response

Code	Details
200	<div><div>Response body</div><div>[\n {\n "id": 1,\n "genero": "fantasia",\n "libros": []\n },\n {\n "id": 2,\n "genero": "historia",\n "libros": []\n },\n {\n "id": 3,\n "genero": "ficcion",\n "libros": []\n }\n]</div></div>

Y, por último, crearemos un libro al que le pondremos poner el genero y el autor correspondiente.



Con esto, insertaríamos un libro con dicho autor y género.

Ahora, si mostramos los datos de dicho autor, podremos comprobar que efectivamente se agrega el libro al autor

GET

/autor/{id} Getautorbyid

Parameters

Name	Description
id * required	
integer	
(path)	2

Execute

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8000/autor/2' \
  -H 'accept: application/json'
```

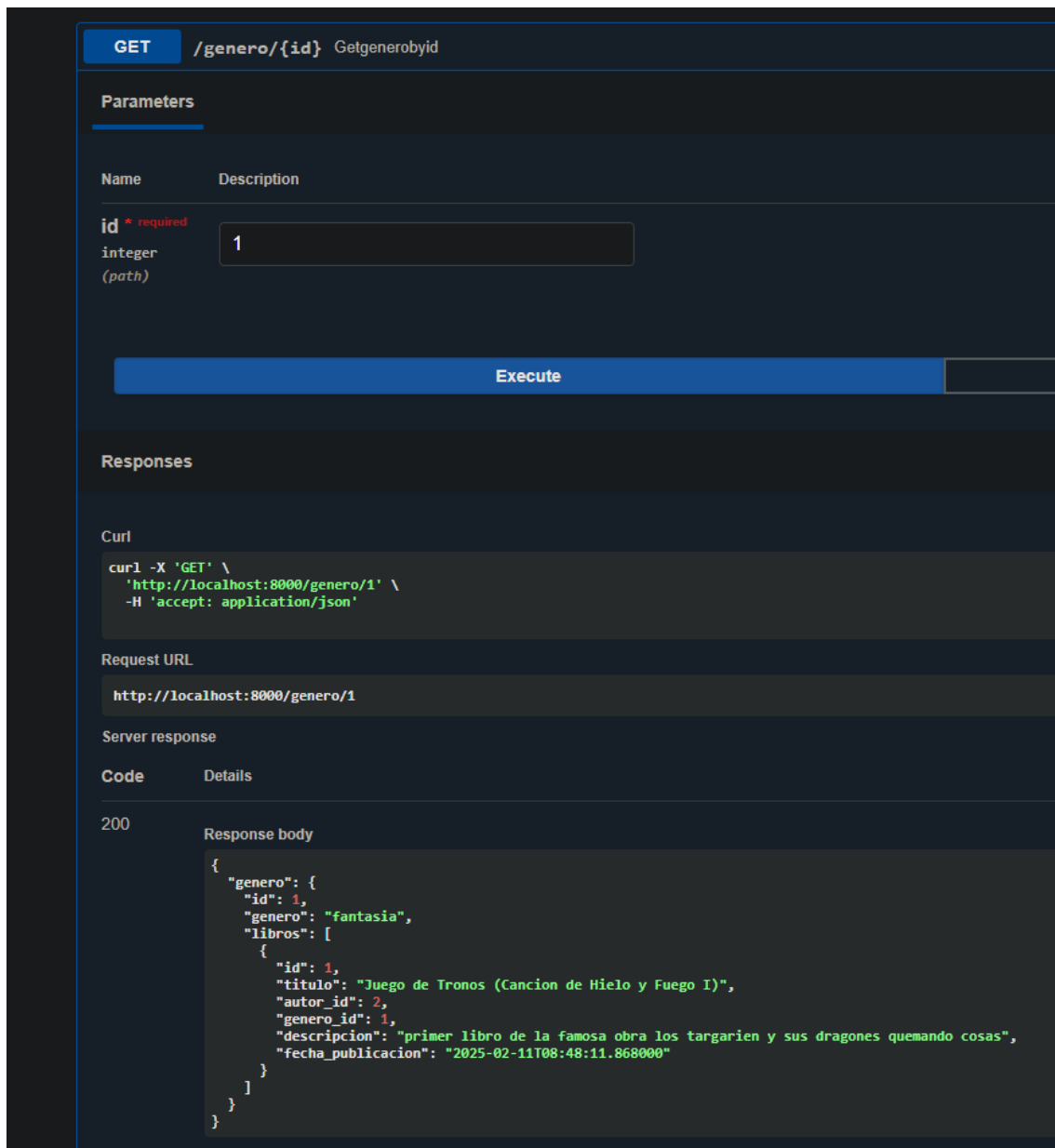
Request URL

```
http://localhost:8000/autor/2
```

Server response

Code	Details
200	<div>Response body</div> <pre>{ "autor": { "id": 2, "nombre": "George R. R. Martin", "edad": 69, "libros": [{ "id": 1, "titulo": "Juego de Tronos (Cancion de Hielo y Fuego I)", "autor_id": 2, "genero_id": 1, "descripcion": "primer libro de la famosa obra los targarien y sus dragones quemando cosas", "fecha_publicacion": "2025-02-11T08:48:11.868000" }] } }</pre>

Pasara lo mismo con el genero (en mi caso, fantasia)



Esto sin duda es lo que mas me ha llevado con diferencia y no veas la de horas invertidas solo para que pueda ver las listas. Por este motivo, no he podido añadir el tema de autorización ya que no me daba tiempo.

Despliegue de la Aplicación:

En mi caso, solo es la API y como utilizo Docker, pues levanto los contenedores que tengo en el Docker (similar a como lo tenía en el del manage).

Esto me levanta tanto el servicio de PostgreSQL donde esta la bd como pgAdmin en el puerto 80.

Por último, para correr la aplicación que corre la API, utilizo primero el comando para activar el entorno virtual

```
PS C:\Users\jaime.gonbra\Desktop\Gestor de Empresas\2ª eva\library_of_ohara> .\venv\Scripts\activate
```

Y el de ejecutar la aplicación.

```
(venv) PS C:\Users\jaime.gonbra\Desktop\Gestor de Empresas\2ª eva\library_of_ohara> python main.py
INFO: Will watch for changes in these directories: ['C:\\Users\\jaime.gonbra\\Desktop\\Gestor de Empresas\\2ª eva\\library_of_ohara']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [20164] using StatReload
INFO: Started server process [12360]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Ya cuando todo este corriendo, podemos ir a la url que vemos ahí o aquí:

<http://localhost:8000/docs#/> solo que cambiando una vez que le hayamos pulsado los caracteres “%23” por el carácter “#” y funciona.

Manuales

Manual de usuario:

Usuarios	
GET	/usuario/all Getusuarios
GET	/usuario/{id} Getusuariobyid
POST	/usuario/add Addusuarios
PATCH	/usuario/{id}/update Updateusuario
DELETE	/usuario/{id}/delete Deleteusuario
Generos	
GET	/genero/all Getgeneros
GET	/genero/{id} Getgenerobyid
POST	/genero/add Addgeneros
PATCH	/genero/{id}/update Updategenero
DELETE	/genero/{id}/delete Deletegenero
Autores	
GET	/autor/all Getautor
GET	/autor/{id} Getautorbyid
POST	/autor/add Addautor
PATCH	/autor/{id}/update Updateautor
DELETE	/autor/{id}/delete Deleteautor
Libros	
GET	/libro/all Getlibro
GET	/libro/{id} Getlibrobyid
POST	/libro/add Addlibro
PATCH	/libro/{id}/update Updatelibro
DELETE	/libro/{id}/delete Deletelibro

Como se puede ver en la imagen, aquí tenemos todas las operaciones que se pueden llegar a realizar utilizando este proyecto. Dichas acciones llegan desde crear, modificar, eliminar y visualizar ya sean todos los datos como solo por su id de libros, géneros, autores y usuarios.

Manual de instalación:

En mi caso, solo es la API, para ejecutarla solo tendremos que activar el entorno virtual para poder añadir todas las dependencias que requiere el proyecto.

```
PS C:\Users\jaime.gonbra\Desktop\Gestor de Empresas\2ª eva\library_of_ohara> .\venv\Scripts\activate
```

Después, ponemos el comando para ejecutar la aplicación. el de ejecutar la aplicación.

```
(venv) PS C:\Users\jaime.gonbra\Desktop\Gestor de Empresas\2ª eva\library_of_ohara> python main.py
INFO: Will watch for changes in these directories: ['C:\\Users\\jaime.gonbra\\Desktop\\Gestor de Empresas\\2ª eva\\library_of_ohara']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [20164] using StatReload
INFO: Started server process [12360]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Conclusiones:

Cristina, este proyecto ha sido una montaña rusa de emociones, en algunos casos todo bien porque funcionaban las cosas, pero una vez que quería tener todo lo de aquí en mi casa o viceversa, fue una completa depresión. En fin, que ya está hecho y no quiero volver a tocar Python hasta dentro de unos años por lo menos.

Dos posibles ampliaciones que se podrían haber dado son:

Inclusión de la tabla usuariosLibros que como su nombre indica, seria para tener registrado todos los libros que un usuario ha leído. En bd seria una relación n: n entre usuarios y libros.

Introducir el jwt en el proyecto para darle seguridad al proyecto que, aunque sí que lo hemos dado en AAD, se hacerlo en Java, pero no en Python.

Bibliografía:

Teoría de microservicios: decidesoluciones.es

Teoría de API: redhat.com

Enlace al proyecto en GitHub: [Library of Ohara](https://github.com/ohara-library)