



PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones
Multiplataforma
Informática y Comunicaciones

Proyecto final FastAPI

Año: 2025

Fecha de presentación: 10 de febrero

Nombre y Apellidos: Belén Añazco Torres
Email: belenat.99@gmail.com

Índice

1	Introducción.....	3
2	Estado del arte	3
3	Descripción general del proyecto	6
3.1	Objetivos.....	6
3.2	Entorno de trabajo (tecnologías de desarrollo y herramientas)	7
4	Documentación técnica	8
4.1	Análisis del sistema	8
4.2	Diseño de la base de datos	9
4.3	Implementación.....	10
4.4	Pruebas	16
4.4.1	Crear usuario.....	16
4.4.2	Obtener usuarios	16
4.4.3	Obtener usuario por id	17
4.4.4	Error de obtener venta que no existe.....	18
4.4.5	Eliminar producto	19
4.4.6	Eliminar producto que no existe.....	20
4.4.7	Visualizar datos en la BBDD	21
4.5	Despliegue de la aplicación.....	21
5	Manuales	22
5.1	Manual de usuario	22
5.2	Manual de instalación.....	22
6	Conclusiones y posibles ampliaciones	23
7	Bibliografía	23
8	Enlace GitHub	23

1 Introducción

Este proyecto está diseñado para demostrar cómo construir un API Rest utilizando FastAPI, un framework moderno y eficiente para aplicaciones web en Python.

La aplicación gestiona 3 tablas principales: Usuarios, Productos y Ventas, las cuales están relacionadas entre sí para simular un sistema básico de comercio electrónico.

Usuarios:

Representa a los clientes que interactúan con el sistema. Contiene información básica como nombre, apellido, correo, contraseña, además de tener un identificador único.

Productos:

Define los artículos disponibles. Cada artículo tiene atributos como nombre, descripción, precio, stock y un identificador.

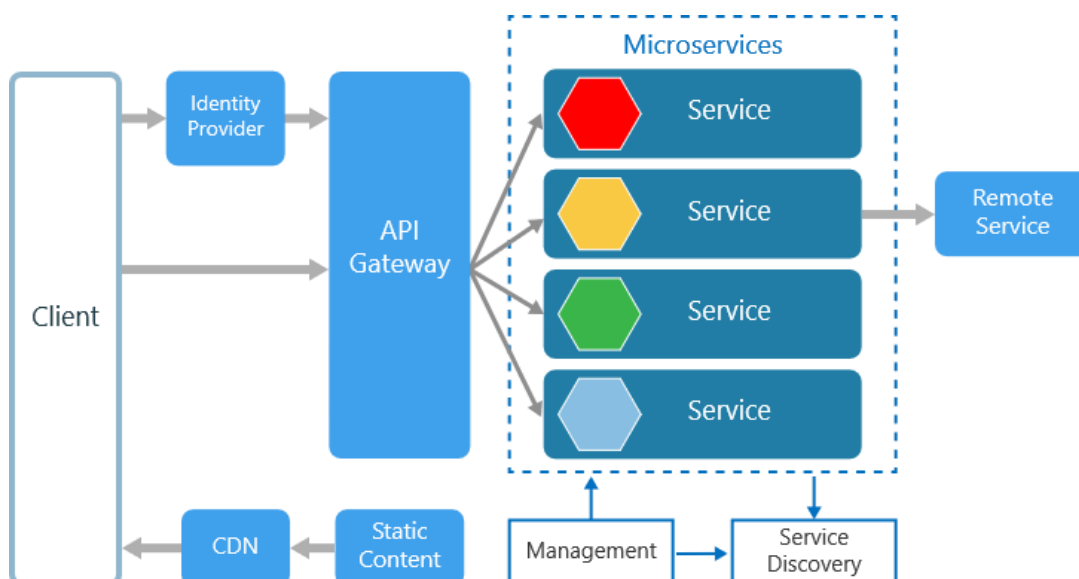
Ventas:

Registra las transacciones realizadas por los usuarios. Esta tabla relaciona a los usuarios con los productos vendidos.

2 Estado del arte

Definición de arquitectura de microservicios:

Una arquitectura de microservicios divide una aplicación en una serie de servicios implementables de forma independiente que se comunican a través de API. Este enfoque permite implementar y escalar cada servicio individual de forma independiente, así como la entrega rápida y frecuente de aplicaciones grandes y complejas. A diferencia de una aplicación monolítica, una arquitectura de microservicios permite a los equipos implementar nuevas funciones y hacer cambios más rápido, sin tener que volver a escribir una gran parte del código existente.



Definición de API:

Una API (Interfaz de Programación de Aplicaciones) es un conjunto de reglas y protocolos que permite que diferentes sistemas, aplicaciones o servicios se comuniquen entre sí. En términos simples, una API actúa como un intermediario que permite que dos aplicaciones intercambien datos y funcionalidades.

Tipos de APIs:

- **APIs RESTful:** Basadas en el protocolo HTTP, son las más comunes por su simplicidad y compatibilidad.
- **APIs GraphQL:** Permiten consultas flexibles y precisas sobre los datos.
- **APIs SOAP:** Más estructuradas y orientadas a sistemas empresariales.
- **APIs WebSocket:** Utilizadas para comunicaciones bidireccionales en tiempo real.

Estructura de una API:

Una API bien diseñada sigue una estructura organizada que facilita la interacción entre clientes y servidores. A continuación, se describe en detalle su estructura, el protocolo utilizado, métodos comunes y las partes que conforman una URL en una API.

La mayoría de las APIs modernas utilizan el protocolo HTTP(S) debido a su simplicidad y compatibilidad con navegadores y sistemas.

- **HTTP (HyperText Transfer Protocol):** Permite la comunicación entre cliente y servidor mediante solicitudes (requests) y respuestas (responses).
- **HTTPS:** Es la versión segura de HTTP, que encripta los datos transmitidos para proteger la información sensible.

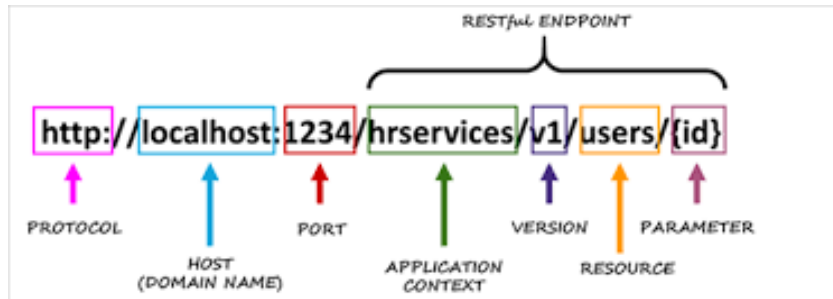
Métodos HTTP comunes:

- **GET:** recuperar datos o información de un recurso.
Ejemplo: Obtener la lista de productos.
GET /productos
- **POST:** crear un nuevo recurso en el servidor
Ejemplo: Agregar un nuevo producto.
POST /productos
- **PUT:** actualizar o reemplazar completamente un recurso existente
Ejemplo: Modificar un producto existente.
PUT /productos/{id}
- **PATCH:** actualizar parcialmente un recurso
Ejemplo: Cambiar solo el stock de un producto.
PATCH /productos/{id}
- **DELETE:** eliminar recurso existente
Ejemplo: Eliminar un producto específico.
DELETE /productos/{id}

Partes de una URL en una API:

La URL de una API define los recursos y las acciones disponibles. Su estructura típica es:

{protocolo}://{dominio}/{version}/{recurso}/{id}?{parametros}



1. **Protocolo**
http o https (se recomienda siempre usar https por seguridad).
2. **Dominio**
Dirección del servidor donde se encuentra alojada la API.
3. **Versión**
Indica la versión de la API para gestionar cambios sin afectar clientes existentes.
4. **Recurso**
Representa la entidad que se está gestionando (usuarios, productos, ventas).
5. **Identificador (opcional)**
Especifica un recurso en particular.
6. **Parámetros de consulta (Query Parameters)**
Se utilizan para filtrar o personalizar la solicitud.

Formas de crear una API en Python:

FastAPI

FastAPI es un framework moderno y de alto rendimiento para construir APIs, basado en las características de Python como las anotaciones de tipos. Está diseñado para ser rápido y fácil de usar, especialmente para APIs basadas en JSON.

Ventajas de FastAPI:

- **Rendimiento:** basado en ASGI (Asynchronous Server Gateway Interface) y el motor de Starlette, es más rápido que Flask para aplicaciones de alto rendimiento.
- **Validación automática:** utiliza Pydantic para validar datos automáticamente
- **Documentación automática:** genera documentación interactiva (Swagger y Redoc) a partir de las anotaciones de tipos de Python.
- **Soporte para asincronía**

Flask

Flask es un “micro” Framework escrito en Python y concebido para facilitar el desarrollo de Aplicaciones Web bajo el patrón MVC.

La palabra “micro” no designa a que sea un proyecto pequeño o que nos permita hacer páginas web pequeñas, sino que al instalar Flask tenemos las herramientas necesarias para crear una aplicación web funcional, pero si se necesita en algún momento una nueva funcionalidad hay un conjunto muy grande de extensiones (plugins) que se pueden instalar con Flask que le van dotando de funcionalidad.

Ventajas de Flask:

- **Ligero y flexible:** no incluye demasiadas dependencias, lo que te da libertad para elegir herramientas adicionales según lo necesites.
- **Amplia comunidad:** gran cantidad de extensiones disponibles.
- **Fácil de aprender:** su simplicidad lo hace ideal para principiantes.

Use	Use
FastAPI	FLASK
<ul style="list-style-type: none">• High-performance API development.• Microservices architecture.• Real-time applications and websockets.• Projects requiring extensive data validation.	<ul style="list-style-type: none">• Small to medium-sized web applications.• Prototyping and quick MVP development.• Educational projects and learning web development.• Projects where you need full control over the components and architecture.





3 Descripción general del proyecto

3.1 Objetivos









Gestión de usuarios:

Permitir el registro, consulta, actualización y eliminación de usuarios, con validación de datos mediante esquemas definidos en Pydantic.

-  **Gestión de productos:**
Ofrecer funcionalidades para agregar, actualizar, consultar y eliminar productos, garantizando una estructura clara para la manipulación de datos.
-  **Gestión de ventas:**
Registrar y consultar ventas, relacionando usuarios y productos de manera eficiente.
-  **Optimización y documentación:**
Aprovechar las ventajas de FastAPI para generar automáticamente documentación interactiva (Swagger y Redoc), lo que facilita la comprensión y uso de la API por parte de desarrolladores.
-  **Escalabilidad y buenas prácticas:**
Diseñar un sistema modular y limpio que permita su fácil mantenimiento y expansión, utilizando una arquitectura basada en routers y dependencias.

3.2 *Entorno de trabajo (tecnologías de desarrollo y herramientas)*

-  **Lenguaje de programación: Python**
 - Python es un lenguaje de programación versátil, legible y ampliamente utilizado en el desarrollo de aplicaciones web, inteligencia artificial, automatización y más.
 - Rol en el proyecto: Es el lenguaje principal para escribir la lógica de la API. Se utiliza junto con bibliotecas como FastAPI, Pydantic y SQLAlchemy.
-  **Framework principal: FastAPI**
 - Framework moderno para construir APIs en Python, diseñado para ser rápido, fácil de usar y con validación automática de datos.
 - Rol en el proyecto: Facilita la creación de rutas RESTful para gestionar usuarios, productos y ventas.
-  **Gestor de base de datos**
 - PostgreSQL es una base de datos relacional de código abierto que tiene el respaldo de 30 años de desarrollo, lo que la convierte en una de las bases de datos relacionales más consolidadas que hay disponibles. PostgreSQL debe su popularidad entre los desarrolladores y administradores a su gran flexibilidad e integridad.
-  **Docker**
 - Docker es una potente plataforma de código abierto que utiliza contenedores para simplificar la creación, despliegue y ejecución de aplicaciones. Estos contenedores permiten a los desarrolladores empaquetar una aplicación con todos sus componentes necesarios, como bibliotecas y otras dependencias, y enviarla como un único paquete.
 - Rol en el proyecto: Se utiliza para contenedores que incluyen la API de FastAPI y el gestor de base de datos.
-  **IDE: Visual Studio Code**
 - Entorno de desarrollo ligero y extensible, ideal para proyectos en múltiples lenguajes de programación.
 - Rol en el proyecto: Herramienta principal para escribir, depurar y probar el código.
-  **Pydantic**
 - Biblioteca de validación de datos y manejo de esquemas en Python.
 - Rol en el proyecto: Define y valida las estructuras de datos de las entidades principales: user, producto y venta.

Uvicorn

- Servidor ASGI que ejecuta aplicaciones basadas en FastAPI.
- Rol en el producto: Es el servidor que ejecuta la API y responde a las solicitudes HTTP.

GitHub

- GitHub: plataforma para colaborar, almacenar y gestionar el código en repositorios.

4 Documentación técnica

4.1 *Análisis del sistema*

La API esta diseñada para ser actualizada por clientes que necesitan interactuar con la aplicación. La API ofrece los siguientes endpoints:

Usuarios:

- GET /usuarios: Obtiene una lista de todos los usuarios
- POST /usuarios: Crea nuevo usuario
- GET /usuarios: Obtiene un usuario específico por su ID
- PUT /usuarios: Actualiza un usuario específico por su ID
- DELETE /usuarios: Elimina un usuario específico por su ID

Ventas:

- GET /ventas: Obtiene una lista de todas las ventas
- POST / ventas: Crea nueva venta
- GET / ventas: Obtiene una venta específica por su ID
- PUT / ventas: Actualiza una venta específica por su ID
- DELETE / ventas: Elimina una venta específica por su ID

Productos:

- GET /productos: Obtiene una lista de todos los productos
- POST / productos: Crea nuevo producto
- GET / productos: Obtiene un producto específico por su ID
- PUT / productos: Actualiza un producto específico por su ID
- DELETE / productos: Elimina un producto específico por su ID

Manejo de Errores

La aplicación maneja los errores de la siguiente manera:

- Errores de validación: La aplicación verifica los datos proporcionados por el cliente y devuelve un error si no cumplen con los requisitos de validación.
- Errores de base de datos: La aplicación maneja los errores de base de datos y devuelve un error si no se puede realizar la operación solicitada.

- Errores de autenticación: La aplicación maneja los errores de autenticación y devuelve un error si el token de autenticación no es válido.

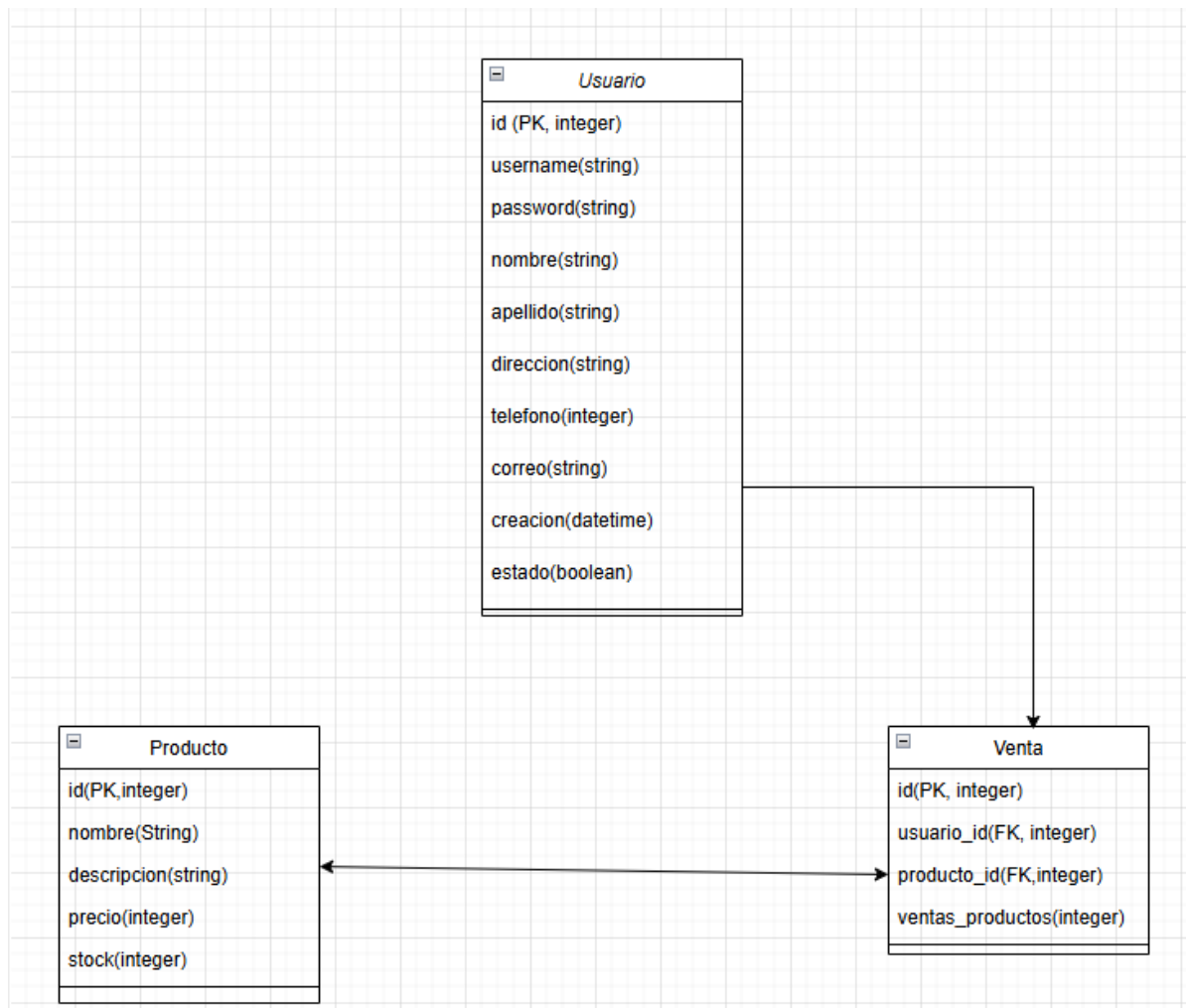
Autenticación

La aplicación utiliza autenticación basada en tokens para proteger los endpoints. Los clientes deben proporcionar un token de autenticación válido en cada solicitud para acceder a los recursos protegidos.

- Registro: Los clientes pueden registrarse en la aplicación proporcionando un nombre de usuario y una contraseña. La aplicación crea un token de autenticación para el cliente y lo devuelve en la respuesta.
- Inicio de sesión: Los clientes pueden iniciar sesión en la aplicación proporcionando su nombre de usuario y contraseña. La aplicación verifica las credenciales y devuelve un token de autenticación si son válidas.

4.2 Diseño de la base de datos

Diagrama de la BBDD.



4.3 Implementación

Estructura del código

- **app:** Es la carpeta principal de la aplicación, donde se encuentran los archivos de configuración y los módulos de la aplicación.
- **app/routers:** Es la carpeta donde se encuentran los módulos de routing de la aplicación, que definen las rutas de la API.
- **app/schemas:** Es la carpeta donde se encuentran los módulos de esquemas de la aplicación, que definen la estructura de los datos.
- **app/db:** Es la carpeta donde se encuentran los módulos de base de datos de la aplicación, que definen la conexión a la base de datos y los modelos de datos.

Principales librerías

las principales librerías utilizadas son:

- FastAPI: Es el framework utilizado para crear la API.
- PostgreSQL: Es la base de datos utilizada para almacenar los datos.
- SQLAlchemy: es la librería utilizada para interactuar con la base de datos.
- Pydantic: Es la librería utilizada para definir los esquemas de datos.

Capturas de código

Main.py

```
main.py > ...
1  from fastapi import FastAPI
2  import uvicorn
3  from app.routers import user, venta, producto
4  from app.db.database import Base, engine
5
6  def create_tables():
7      |   Base.metadata.create_all(bind=engine)
8
9  create_tables()
10
11
12  app = FastAPI()
13  app.include_router(user.router)
14  app.include_router(venta.router)
15  app.include_router(producto.router)
16
17  if __name__ == "__main__":
18      |   uvicorn.run("main:app", port=8000, reload=True)
```

Database.py

```
app > 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 #URL de la base de datos
6 SQLALCHEMY_DATABASE_URL = "postgresql://odoo:odoo@localhost:5342/fastapi-database"
7 engine = create_engine(SQLALCHEMY_DATABASE_URL)
8 SessionLocal = sessionmaker(bind=engine,autocommit=False,autoflush=False)
9 Base = declarative_base()
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
```

Schemas.py

```
app > schemas.py > Producto
1 from pydantic import BaseModel
2 from typing import Optional
3 from datetime import datetime
4
5 #User Model
6 class User(BaseModel): #Schema
7     id:int
8     username:str
9     password:str
10    nombre:str
11    apellido:str
12    direccion:Optional[str] #Parámetro opcional; es necesario importar Optional
13    telefono:int
14    correo:str
15    creacion_user:datetime=datetime.now() #Fecha por defecto
16
17
18
19 class Venta(BaseModel):
20     id:int
21     usuario_id:int
22     producto_id:int
23     venta:int
24     ventas_productos:int
25
26 class Producto(BaseModel):
27     id:int
28     nombre:str
29     descripcion:str
30     precio:int
31     stock:int
32
33 class UpdateUser(BaseModel):
34     username:str = None
35     password:str = None
36     nombre:str = None
37     apellido:str = None
38     direccion:str = None #Parámetro opcional; es necesario importar Optional
39     telefono:int = None
40     correo:str = None
41
```

User.py

```

app > routers > user.py > crear_usuario
1  from fastapi import APIRouter, Depends
2  from app.schemas import User, UpdateUser
3  from app.db.database import get_db
4  from sqlalchemy.orm import Session
5  from app.db import models
6
7  usuarios = []
8
9  router = APIRouter(
10     prefix="/user",
11     tags=["Users"]
12 )
13
14
15
16 @router.get("")
17 def obtener_usuarios(db:Session=Depends(get_db)):
18     data = db.query(models.User).all()
19     print(data)
20     return data
21
22 @router.post("")
23 def crear_usuario(user:User, db:Session=Depends(get_db)):
24     usuario=user.model_dump()
25     nuevo_usuario=models.User(
26         username=usuario["username"],
27         password=usuario["password"],
28         nombre=usuario["nombre"],
29         apellido=usuario["apellido"],
30         direccion=usuario["direccion"],
31         telefono=usuario["telefono"],
32         correo=usuario["correo"],
33     )
34     db.add(nuevo_usuario)
35     db.commit()
36     db.refresh(nuevo_usuario)
37     return {"Respuesta": "Usuario creado!!"}
38
39 @router.get("/{user_id}")
40 def obtener_usuario(user_id:int, db:Session=Depends(get_db)):
41     usuario=db.query(models.User).filter(models.User.id==user_id).first()
42     if not usuario:
43         return {"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe
44     return usuario
45
46
47 @router.delete("/{user_id}")
48 def eliminar_usuario(user_id: int, db: Session = Depends(get_db)):
49     usuario = db.query(models.User).filter(models.User.id == user_id).first()
50     if not usuario:
51         return {"respuesta": "Usuario no encontrado"}
52     db.delete(usuario)
53     db.commit()
54     return {"Respuesta": "Usuario eliminado correctamente"}
55
56 @router.patch("/{user_id}")
57 def actualizar_usuario(user_id:int, updateUser:UpdateUser, db:Session=Depends(get_db)):
58     usuario=db.query(models.User).filter(models.User.id==user_id)
59     if not usuario.first():
60         return {"respuesta": "Usuario no encontrado"}
61     usuario.update(updateUser.model_dump(exclude_unset=True))
62     db.commit()
63     return {"Respesuta": "Usuario actualizado correctamente"}
64

```

Ventas.py

```
from fastapi import APIRouter, Depends
from app.schemas import Venta
from app.db.database import get_db
from sqlalchemy.orm import Session
from app.db import models

ventas = []

router = APIRouter(
    prefix="/venta",
    tags=["Ventas"]
)

# @router.get("/ruta1")
# def ruta1():
#     return {"mensaje": "Hemos creado nuestra primera API!!!"}

@router.get("/")
def obtener_ventas(db: Session = Depends(get_db)):
    data = db.query(models.Venta).all()
    print(data)
    return ventas

@router.post("/")
def crear_venta(venta: Venta):
    venta = venta.model_dump()
    ventas.append(venta)
    return {"Respuesta": "Venta creada!"}

@router.post("/{venta_id}")
def obtener_venta(venta_id: int):
    for venta in ventas:
        if venta["id"] == venta_id:
            return {"venta": venta}
    return {"respuesta": "Venta no encontrada"}

# @router.post("/json")
# def obtener_venta_json(venta_id: int):
#     for user in usuarios:
#         if user["id"] == venta_id: #Acceder al id de user (tipo dict) y comparar con el id de venta_id que se p
#             return {"usuario": user}
#     return {"respuesta": "Usuario no encontrado"} #Si se pasa un id de un usuario que no existe

@router.delete("/{venta_id}")
def eliminar_venta(venta_id: int):
    for index, venta in enumerate(ventas):
        if venta["id"] == venta_id:
            ventas.pop(index)
            return {"Respuesta": "Venta eliminada correctamente"}
    return {"Respuesta": "Venta NO encontrada"}

@router.put("/{venta_id}")
def actualizar_venta(venta_id: int, updateVenta: Venta):
    for index, user in enumerate(ventas):
        if user["id"] == venta_id:
            ventas[index]["id"] = updateVenta.model_dump()["id"]
            ventas[index]["usuario_id"] = updateVenta.model_dump()["usuario_id"]
            ventas[index]["producto_id"] = updateVenta.model_dump()["producto_id"]
            ventas[index]["venta"] = updateVenta.model_dump()["venta"]
            ventas[index]["ventas_productos"] = updateVenta.model_dump()["venta_productos"]
            return {"Respuesta": "Venta actualizada correctamente"}
    return {"Respuesta": "Venta NO encontrada"}
```

Producto.py

```
from fastapi import APIRouter, Depends
from app.schemas import Producto
from app.db.database import get_db
from sqlalchemy.orm import Session
from app.db import models

productos = []

router = APIRouter(
    prefix="/producto",
    tags=["Productos"]
)

# @router.get("/ruta1")
# def ruta1():
#     return {"mensaje": "Hemos creado nuestra primera API!!!"}

@router.get("")
def obtener_productos(db: Session = Depends(get_db)):
    data = db.query(models.Producto).all()
    print(data)
    return productos

@router.post("")
def crear_producto(producto: Producto):
    producto = producto.model_dump()
    productos.append(producto)
    return {"Respuesta": "Producto creado!"}

@router.post("/{producto_id}")
def obtener_producto(producto_id: int):
    for producto in productos:
        if producto["id"] == producto_id:
            return {"producto": producto}
    return {"respuesta": "Producto no encontrado"}

# @router.post("/json")
# def obtener_usuario_json(user_id: UserId):
#     for user in usuarios:
#         if user["id"] == user_id.id:
#             return {"usuario": user}
#     return {"respuesta": "Usuario no encontrado"}

@router.delete("/{producto_id}")
def eliminar_producto(producto_id: int):
    for index, producto in enumerate(productos):
        if producto["id"] == producto_id:
            productos.pop(index)
            return {"Respuesta": "Producto eliminado correctamente"}
    return {"Respuesta": "Producto NO encontrado"}

@router.put("/{producto_id}")
def actualizar_producto(producto_id: int, updateProducto: Producto):
    for index, producto in enumerate(productos):
        if producto["id"] == producto_id:
            productos[index]["id"] = updateProducto.model_dump()["id"]
            productos[index]["nombre"] = updateProducto.model_dump()["nombre"]
            productos[index]["descripcion"] = updateProducto.model_dump()["descripcion"]
            productos[index]["precio"] = updateProducto.model_dump()["precio"]
            productos[index]["stock"] = updateProducto.model_dump()["stock"]
            productos[index]["ventas"] = updateProducto.model_dump()["ventas"]
            return {"Respuesta": "Producto actualizado correctamente"}
    return {"Respuesta": "Producto NO encontrado"}
```

Models.py

```
app > db > models.py > ...
1 from app.db.database import Base
2 from sqlalchemy import Column,Integer,String,Boolean,DateTime,ForeignKey
3 from sqlalchemy.orm import relationship
4 from datetime import datetime
5
6 class User(Base):
7     __tablename__ = 'usuario'
8     id = Column(Integer,primary_key=True,autoincrement=True)
9     username=Column(String,unique=True)
10    password=Column(String)
11    nombre = Column(String)
12    apellido = Column(String)
13    direccion = Column(String)
14    telefono = Column(Integer)
15    correo = Column(String,unique=True)
16    creacion = Column(DateTime,default=datetime.now,onupdate=datetime.now)
17    estado = Column(Boolean,default=False)
18    venta=relationship("Venta",backref="usuario", cascade="delete,merge")
19
20 class Venta(Base):
21     __tablename__="venta"
22     id=Column(Integer, primary_key=True,autoincrement=True)
23     usuario_id=Column(Integer,ForeignKey('usuario.id'))
24     producto_id = Column(Integer, ForeignKey('producto.id'))
25     producto_venta = relationship("Producto", backref="venta_producto")
26     ventas_productos=Column(Integer)
27
28 class Producto(Base):
29     __tablename__="producto"
30     id=Column(Integer, primary_key=True,autoincrement=True)
31     nombre=Column(String)
32     descripcion=Column(String)
33     precio=Column(String)
34     stock=Column(String)
35     ventas=relationship("Venta", backref="producto")
36
```

4.4 Pruebas

4.4.1 Crear usuario

POST /user Crear Usuario

Parameters

No parameters

Request body required

```

{
  "id": 5,
  "username": "pruebaaa",
  "password": "1111r",
  "nombre": "obetener",
  "apellido": "usuario",
  "direccion": "string",
  "telefono": 0,
  "correo": "string",
  "creacion_user": "2025-02-11T08:51:54.744324"
}
        
```

```

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

4.4.2 Obtener usuarios

```

{
  "id": 4,
  "username": "pruebaaa",
  "password": "obtener",
  "nombre": "string",
  "apellido": "string",
  "direccion": "string",
  "telefono": 0,
  "correo": "string",
  "creacion_user": "2025-02-11T08:51:54.744324"
},
{
  "id": 5,
  "username": "pruebaaa",
  "password": "1111r",
  "nombre": "obetener",
  "apellido": "usuario",
  "direccion": "string",
  "telefono": 0,
  "correo": "string",
  "creacion_user": "2025-02-11T08:51:54.744324"
}
        
```


4.4.3 Obtener usuario por id

POST `/user/{user_id}` Obtener Usuario

Parameters

Name	Description
user_id * required integer (path)	<input type="text" value="5"/>

Execute

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/user/5' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://localhost:8000/user/5
```

Server response

Code	Details
200	<div>Response body</div> <pre>{ "usuario": { "id": 5, "username": "pruebaaa", "password": "11111r", "nombre": "obetener", "apellido": "usuario", "direccion": "string", "telefono": 0, "correo": "string", "creacion_user": "2025-02-11T08:51:54.744324" } }</pre>

4.4.4 Error de obtener venta que no existe

POST `/venta/{venta_id}` Obtener Venta

Parameters

Name	Description
venta_id * required integer (path)	<input type="text" value="4"/>

Execute

Responses

Curl

```
curl -X 'POST' \
'http://localhost:8000/venta/4' \
-H 'accept: application/json' \
-d ''
```

Request URL

```
http://localhost:8000/venta/4
```

Server response

Code	Details
200	<div>Response body<pre>{ "respuesta": "Venta no encontrada" }</pre></div>

4.4.5 Eliminar producto

DELETE `/producto/{producto_id}` Eliminar Producto

Parameters

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

Execute

Responses

Curl

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

Request URL

```
http://localhost:8000/producto/1
```

Server response

Code	Details
200	<div>Response body</div> <pre>{ "Respuesta": "Producto eliminado correctamente" }</pre>

4.4.6 Eliminar producto que no existe

DELETE `/producto/{producto_id}` Eliminar Producto

Parameters

Name	Description
producto_id * required integer (path)	<input type="text" value="5"/>

Execute

Responses

Curl

```
curl -X 'DELETE' \
'http://localhost:8000/producto/5' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:8000/producto/5
```

Server response

Code	Details
200	<div>Response body<pre>{ "Respuesta": "Producto NO encontrado" }</pre></div>

4.4.7 Visualizar datos en la BBDD

1

```
select * from usuario
```

Data Output Mensajes Notificaciones

+ 📄 ▼ 📋 ▼ 🗑️ 📦 ⬇️ 📈 SQL

	id [PK] integer	username character varying	password character varying	nombre character varying	apellido character varying	direccion character varying	telefono integer
1	3	shhffffng	string	string	string	string	0
2	5	sg	stg	strg	sting	string	0

↗

4.5 Despliegue de la aplicación

Servidor local

- Se puede utilizar un servidor local como localhost o 127.0.0.1 para acceder a la aplicación.
- Se debe configurar el servidor para que escuche en el puerto deseado (por ejemplo, 8000).
- Se puede acceder a la aplicación desde cualquier dispositivo en la misma red local.

5 Manuales

5.1 Manual de usuario

Users

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

Ventas

GET	/venta Obtener Ventas	✓
POST	/venta Crear Venta	✓
POST	/venta/{venta_id} Obtener Venta	✓
DELETE	/venta/{venta_id} Eliminar Venta	✓
PUT	/venta/{venta_id} Actualizar Venta	✓

Productos

GET	/producto Obtener Productos	✓
POST	/producto Crear Producto	✓
POST	/producto/{producto_id} Obtener Producto	✓
DELETE	/producto/{producto_id} Eliminar Producto	✓
PUT	/producto/{producto_id} Actualizar Producto	✓

5.2 Manual de instalación

Despliegue en Servidor Local:

- Instalar dependencias: fastapi, uvicorn, psycpg2 y SQLAlchemy.
- Configurar la base de datos: Configurar la base de datos PostgreSQL en servidor local y actualizar la variable SQLALCHEMY_DATABASE_URL en el archivo database.py.
- Crear las tablas de la base de datos.
- Iniciar la aplicación: Ejecutar el comando `uvicorn main:app --host 0.0.0.0 --port 8000` para iniciar la aplicación en el servidor local.
- Acceder a la aplicación: Abrir un navegador web y acceder a la aplicación en `http://localhost:8000`.

6 Conclusiones y posibles ampliaciones

Las dificultades que he podido encontrar ha sido configurar la base de datos, ya que debía de asegurarme de que la conexión fuera segura y estable, otro ha sido la resolución de errores porque debía asegurarme de que los errores fueran resueltos y no afectara a la funcionalidad de la API.

Gracias a esta práctica aprendí a utilizar herramientas como FastAPI y SQLAlchemy, y a configurar la base de datos de manera segura. También aprendí a resolver errores y a mejorar la funcionalidad de la aplicación.

Como posible ampliación sería encriptar la contraseña y otra sería implementar un sistema de pedidos y facturas.

7 Bibliografía

<https://www.atlassian.com/es/microservices/microservices-architecture>

<https://santimacnet.wordpress.com/2017/01/20/azure-arquitecturas-para-microservicios-en-net/>

<https://openwebinars.net/blog/que-es-flask/>

<https://unfoldai.com/fastapi-vs-flask/>

8 Enlace GitHub

https://github.com/DBelen99/proyecto_curso_final.git