



# PROYECTO SGE

## 2ª EVALUACIÓN

---

CFGS Desarrollo de Aplicaciones  
Multiplataforma  
Informática y Comunicaciones

---

**ProyectoFinal DAM24**

***Año: 2024-2025***

***Fecha de presentación: 10-02-2025***

**Nombre y Apellidos: Rebeca Martínez de León**  
**Email: [rebeca.marleo@educa.jcyl.es](mailto:rebeca.marleo@educa.jcyl.es)**

## Índice

1	Introducción.....	3
2	Estado del arte .....	3
3	Descripción general del proyecto .....	4
3.1	Objetivos.....	4
3.2	Entorno de trabajo (tecnologías de desarrollo y herramientas) .....	4
4	Documentación técnica: análisis, diseño, implementación, pruebas y despliegue. ....	5
4.1	Análisis del sistema (funcionalidades básicas de la aplicación).....	5
4.2	Diseño de la base de datos .....	7
4.3	Implementación .....	7
4.4	Pruebas .....	17
4.5	Despliegue de la aplicación.....	18
5	Manuales .....	18
5.1	Manual de usuario .....	18
5.2	Manual de instalación.....	42
6	Conclusiones y posibles ampliaciones .....	44
7	Bibliografía .....	44

# 1 Introducción

El proyecto realizado consiste en la creación de una aplicación móvil cuyo servidor esté basado en una API creada con FastAPI. La aplicación desarrollada en este proyecto consistirá en una herramienta para la realización y gestión de ventas y manejo de usuarios y productos para esas mismas ventas.

# 2 Estado del arte

La arquitectura de microservicios es un método de desarrollo de aplicaciones software que funciona como un conjunto de pequeños servicios que se ejecutan de manera independiente y autónoma, proporcionando una funcionalidad de negocio completa.

Una API, o interfaz de programación de aplicaciones, es un conjunto de reglas o protocolos que permite a las aplicaciones informáticas comunicarse entre sí para intercambiar datos, características y funcionalidades.

Las llamadas al API se implementan como peticiones HTTP, en las que la URL representa el recurso, el método (HTTP Verbs) representa la operación y el código de estado HTTP representa el resultado. Además, una API consta de varios métodos para interactuar con los datos, estos son:

- Método GET: Con el método GET, los datos que se envían al servidor se escriben en la misma dirección URL.  
Este método tiene algunas ventajas como que los parámetros URL se pueden guardar junto a la dirección URL como marcador. De esta manera, puedes introducir una búsqueda y más tarde consultarla de nuevo fácilmente. Sin embargo, esto también presenta inconvenientes como es su menor protección de los datos al estar expuestos en la url.
- Método POST: El método POST introduce los parámetros en la solicitud HTTP para el servidor. Por ello, no quedan visibles para el usuario. Además, la capacidad del método POST es ilimitada.  
En lo relativo a los datos, como, por ejemplo, al rellenar formularios con nombres de usuario y contraseñas, el método POST ofrece mucha discreción. Los datos no se muestran en el caché ni tampoco en el historial de navegación. La flexibilidad del método POST también resulta muy útil: no solo se pueden enviar textos cortos, sino también otros tipos de información, como fotos o vídeos. Por el contrario este método tiene el riesgo de envío de datos duplicados ya que al actualizar la página estos deben transferirse de nuevo.
- Método DELETE: Este método de petición permite eliminar un recurso específico. También es idempotente; es decir puede ser ejecutado varias veces y tiene el mismo efecto similar al PUT y GET. Semánticamente se utiliza para eliminar de información existente, es semejante a un DELETE de datos a nivel de base de datos.
- Método PUT: Es similar al método de petición POST, solo que el método PUT es idempotente; es decir puede ser ejecutado varias veces y tiene el mismo efecto, caso contrario a un POST que cada vez que se ejecuta realiza la agregación de un nuevo objeto, ya que semánticamente es como una inserción de un nuevo registro. Semánticamente el método

HTTP PUT se utiliza para la actualización de información existente, es semejante a un UPDATE de datos a nivel de base de datos. Los requests de un PUT usualmente se envían los datos por formularios, formato JSON entre otros. Si se compara con las sentencias SQL es similar a un UPDATE.

- Método PATCH: Este método se emplea para modificaciones parciales de un recurso en particular. Se debe revisar si el servidor es compatible con peticiones PATCH. Para averiguar si el servidor acepta peticiones PATCH notifica su compatibilidad en el header Allow o Access-Control-Allow-Methods, otra indicación de que estas peticiones están permitidas son el header Accept-Patch. Semánticamente es similar al PUT, pues actualiza una parte de un registro. Es decir, realiza una especie de UPDATE a nivel de base de datos.

Existen distintas formas de crear una API en Python como pueden ser FastAPI o Flask. Para este proyecto se ha escogido FastAPI por su capacidad de escalado, su mayor velocidad y su fuerte tipado que protege ante operaciones incorrectas.

## 3 Descripción general del proyecto

### 3.1 Objetivos

El proyecto desarrollado tiene como objetivo la creación de una aplicación móvil diseñada para optimizar la gestión de ventas, el manejo de usuarios y la administración de productos dentro de un entorno comercial. Para garantizar un rendimiento eficiente y una comunicación fluida entre la aplicación y el servidor, se ha implementado una API basada en FastAPI, una tecnología moderna y de alto rendimiento que permite una integración rápida y segura.

Esta aplicación ofrecerá a los usuarios una plataforma intuitiva y funcional que facilitará la ejecución de transacciones comerciales, permitiendo a los administradores gestionar el catálogo de productos, supervisar el historial de ventas y administrar los perfiles de los usuarios.

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

Para la realización de este proyecto se han utilizado una serie de tecnologías que han facilitado su desarrollo. Estas herramientas son las siguientes.

- Docker: la plataforma servirá como el entorno centralizado para alojar, administrar y optimizar la base de datos, asegurando un almacenamiento seguro y eficiente de la información. Esta base de datos será utilizada por la API para gestionar de manera estructurada todos los datos relacionados con los usuarios, productos y ventas, permitiendo consultas rápidas, actualizaciones en tiempo real y un control preciso sobre los registros almacenados.
- Visual Studio Code: este fue el editor de texto escogido para la creación y modificación de los archivos requeridos para el correcto funcionamiento del módulo.

- Python: El lenguaje de programación seleccionado para el desarrollo de la funcionalidad de la API ha sido elegido debido a su equilibrio entre simplicidad, eficiencia y velocidad de ejecución. Su sintaxis clara y legible facilita el proceso de desarrollo, reduciendo la complejidad del código y permitiendo una implementación más ágil y mantenible. Además, su robustez y amplio ecosistema de bibliotecas optimizan el rendimiento de la API, asegurando tiempos de respuesta rápidos y una gestión eficiente de las solicitudes. Gracias a estas características, este lenguaje se convierte en una opción ideal para garantizar un desarrollo fluido, escalabilidad y facilidad de integración con otras tecnologías.
- FastAPI: Para el desarrollo de este proyecto, se ha optado por utilizar FastAPI debido a sus múltiples ventajas, que lo convierten en una opción ideal para la creación de APIs modernas y eficientes. Entre sus principales beneficios se encuentra su alta capacidad de escalabilidad, lo que permite que la aplicación pueda manejar un creciente número de solicitudes sin comprometer el rendimiento.  
Además, FastAPI destaca por su notable velocidad de ejecución, optimizando el tiempo de respuesta de la API y asegurando una interacción fluida entre la aplicación móvil y el servidor. Otro aspecto clave de esta tecnología es su sistema de tipado estricto, el cual proporciona una mayor seguridad al detectar posibles errores en el manejo de datos antes de la ejecución, reduciendo así el riesgo de fallos y mejorando la estabilidad del sistema.  
Gracias a estas características, FastAPI no solo facilita el desarrollo rápido y eficiente de la API, sino que también contribuye a la creación de una arquitectura robusta, segura y preparada para el crecimiento a largo plazo.
- pgAdmin: Se ha elegido pgAdmin para albergar y gestionar la base de datos del proyecto debido a su capacidad para ofrecer una interfaz gráfica intuitiva y completa para la administración de bases de datos PostgreSQL. Esta herramienta permite una gestión eficiente de las tablas, consultas y estructuras de datos, facilitando tanto el monitoreo como la optimización del rendimiento del sistema.  
Además, pgAdmin proporciona herramientas avanzadas para la ejecución de consultas SQL, la configuración de permisos y la realización de copias de seguridad, lo que garantiza seguridad y estabilidad en el manejo de la información.

## 4 Documentación técnica: análisis, diseño, implementación, pruebas y despliegue.

### 4.1 *Análisis del sistema (funcionalidades básicas de la aplicación)*

La aplicación desarrollada está diseñada para ofrecer una gestión integral de usuarios, ventas y productos, proporcionando una serie de operaciones esenciales que facilitan la administración eficiente del sistema. Entre las principales funcionalidades disponibles se encuentran:

- Gestión de Usuarios:
  - Crear usuarios: Permite registrar nuevos usuarios en el sistema, almacenando sus datos de manera segura.
  - Modificar usuarios: Facilita la actualización de la información de los usuarios cuando sea necesario.

- Eliminar usuarios: Ofrece la opción de eliminar registros de usuarios que ya no sean necesarios en la plataforma.
- Mostrar usuarios: Permite visualizar la lista de usuarios registrados, con acceso a sus datos básicos.
- Filtrar usuarios: Proporciona herramientas para buscar y filtrar usuarios en función de criterios específicos, optimizando la administración de la información.
- Gestión de Ventas:
  - Crear ventas: Permite registrar nuevas transacciones en el sistema, asociando productos y usuarios a cada operación.
  - Modificar ventas: Facilita la edición de detalles en las ventas registradas, corrigiendo información o ajustando datos cuando sea necesario.
  - Eliminar ventas: Opción para eliminar registros de ventas en caso de errores o anulación de transacciones.
  - Mostrar ventas: Permite visualizar un historial detallado de todas las ventas realizadas en la plataforma.
  - Filtrar ventas: Brinda la posibilidad de aplicar filtros para consultar ventas según fechas, clientes, productos o montos específicos.
- Gestión de Productos:
  - Crear productos: Permite registrar nuevos productos en el sistema, incluyendo detalles como nombre, descripción, precio y stock disponible.
  - Modificar productos: Posibilita la actualización de información sobre los productos, como cambios en precios, nombres o descripciones.
  - Eliminar productos: Permite remover productos que ya no estén disponibles o que no sean necesarios en el catálogo.
  - Mostrar productos: Opción para listar todos los productos registrados, facilitando la visualización del inventario disponible.
  - Filtrar productos: Permite buscar y filtrar productos con base en diferentes criterios, como categoría, precio o disponibilidad.

Gracias a estas funcionalidades, la aplicación se convierte en una herramienta eficiente y flexible, permitiendo una administración completa de los elementos clave dentro del sistema, optimizando la organización de los datos y mejorando la experiencia del usuario.

## 4.2 Diseño de la base de datos

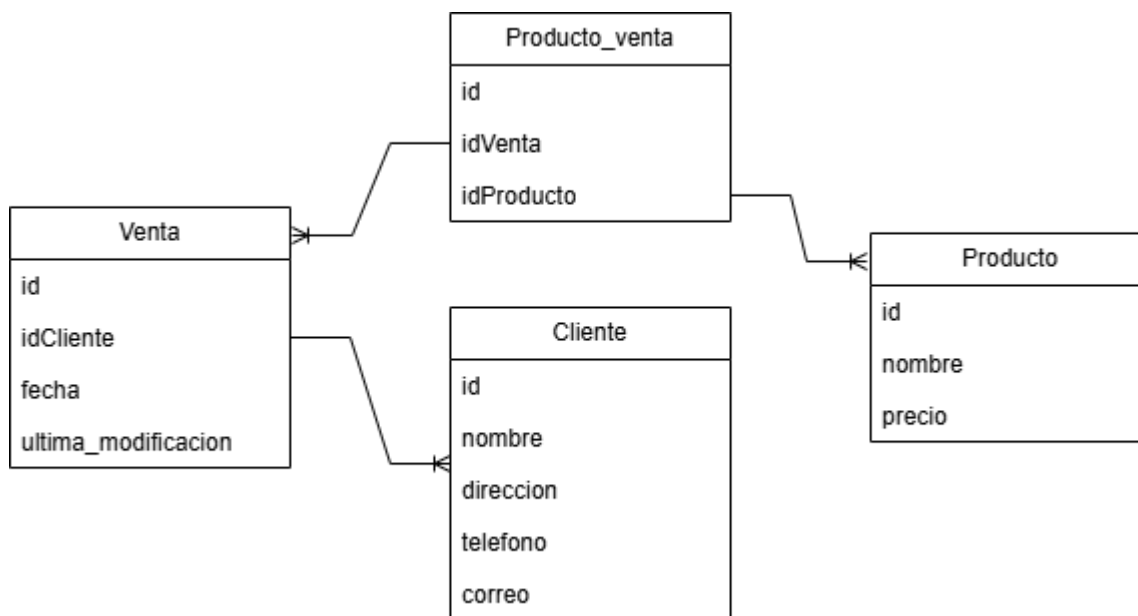
La base de datos consta de cuatro tablas: Cliente, Venta, Producto y Producto\_venta.

La tabla Cliente guarda los datos relativos a un cliente como su nombre, dirección, teléfono y correo junto a un id que lo identifica.

La tabla Venta guarda la información general de la venta, es decir, la fecha de realización de la venta, la fecha de la última modificación en caso de que se hayan realizado cambios y el id del cliente al que está asociada esa venta. Además, cada venta tiene un id propio.

La tabla Productos tiene el nombre, precio e identificador del producto.

La tabla Producto\_venta establece la relación de qué productos se han vendido en cada venta. Un mismo tipo de producto puede venderse a distintos clientes en diferentes ocasiones (una venta) y una venta puede tener múltiples productos. Para establecer guarda el identificador y el identificador del producto juntos. Así se puede ver qué productos pertenecen a una venta concreta, y en qué ventas se ha vendido determinado producto.



## 4.3 Implementación

El código de la API tiene la siguiente estructura.

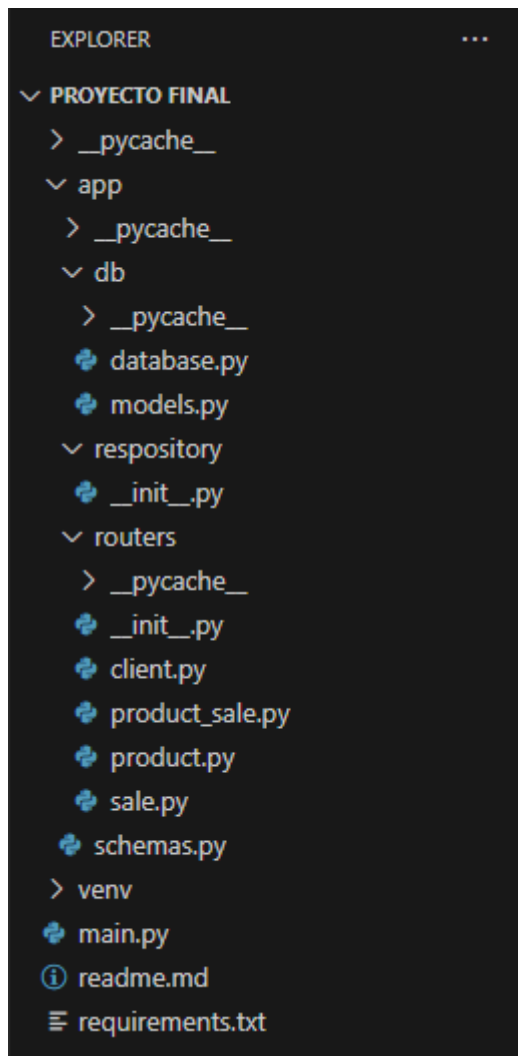
Para empezar, tiene un archivo main.py desde el que se lanza la aplicación. En este archivo se establece la creación de tablas de la base de datos en caso de que no existan. También se crea un objeto FastAPI para establecer los routers de todas las conexiones con las tablas. Por último contiene la configuración para lanzar el servidor de la API.

En la carpeta app podemos encontrar el archivo schemas.py. Este archivo contiene las clases que permiten crear objetos que se relacionen con las tablas de la base de datos.

En esta carpeta se guardan otras tres carpetas. La primera es la carpeta db, donde se guarda todo lo relacionado a la base de datos. Aquí podemos encontrar el archivo database.py que configura la conexión a la base de datos y posee la función que se lanza cuando se quiere establecer una conexión con ella, y el archivo models.py que establece los atributos de las tablas y sus relaciones entre ellas.

Después está la carpeta repository con su archivo \_\_init\_\_.py.

Por último tenemos la carpeta routers. En ella se encuentran los archivos con los routers de cada tabla de la API. Dentro de cada uno de estos archivos se encuentran las funciones necesarias para el correcto funcionamiento de cada parte de la API.



A continuación se muestran imágenes con el código de cada uno de los archivos antes nombrados.



- main.py:

```
main.py > ...
1  from fastapi import FastAPI
2  import uvicorn
3  from app.routers import client, product, sale, product_sale
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  app = FastAPI()
12  app.include_router(client.router)
13  app.include_router(product.router)
14  app.include_router(sale.router)
15  app.include_router(product_sale.router)
16
17  if __name__ == "__main__":
18      uvicorn.run("main:app", port=8000, reload=True)
```

- schemas.py:

```
app > schemas.py > UpdatedClient
1  from pydantic import BaseModel, EmailStr
2  from typing import Optional
3  from datetime import datetime
4
5  class Client(BaseModel):
6      nombre:str
7      direccion:Optional[str]
8      telefono:int
9      correo:str
10
11  class UpdatedClient(BaseModel):
12      nombre:str = None
13      direccion:Optional[str] = None
14      telefono:int = None
15      correo:str = None
16
17  class Sale(BaseModel):
18      id_cliente:int
19      fecha:datetime=datetime.now()
20      ultima_modificacion:datetime=datetime.now()
21
22  class UpdatedSale(BaseModel):
23      id_cliente:int = None
24      fecha:datetime = None
25      ultima_modificacion:datetime=datetime.now()
26
27  class Product(BaseModel):
28      nombre:str
29      precio:float
30
31  class UpdatedProduct(BaseModel):
32      nombre:str = None
33      precio:float = None
34
35  class ProductSale(BaseModel):
36      id_venta:int
37      id_producto:int
38
39  class UpdatedProductSale(BaseModel):
40      id_venta:int = None
41      id_producto:int = None
```

- database.py:

```
app > db > database.py > get_db
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/proyecto-final-database"
6  engine = create_engine(SQLALCHEMY_DATABASE_URL)
7  SessionLocal = sessionmaker(bind=engine,autocommit=False,autoflush=False)
8  Base = declarative_base()
9
10 def get_db():
11     db = SessionLocal()
12     try:
13         yield db
14     except Exception as e:
15         print("ERROR" ,e)
16     finally:
17         db.close()
```

- models.py:

```
app > db > models.py > Product_Sale
1  from app.db.database import Base
2  from sqlalchemy import Column, Integer, Float, String, DateTime
3  from datetime import datetime
4  from sqlalchemy.schema import ForeignKey
5  from sqlalchemy.orm import relationship
6
7  class Client(Base):
8      __tablename__ = "cliente"
9      id = Column(Integer, primary_key=True, autoincrement=True)
10     nombre = Column(String)
11     direccion = Column(String)
12     telefono = Column(Integer)
13     correo = Column(String)
14     venta = relationship("Sale", backref="cliente", cascade="delete,merge")
15
16     class Sale(Base):
17         __tablename__ = "venta"
18         id = Column(Integer, primary_key=True, autoincrement=True)
19         id_cliente = Column(Integer, ForeignKey("cliente.id", ondelete="CASCADE"))
20         productos = relationship("Product_Sale", backref="venta", cascade="delete,merge")
21         fecha = Column(DateTime, default=datetime.now)
22         ultima_modificacion = Column(DateTime, default=datetime.now, onupdate=datetime.now)
23
24     class Product(Base):
25         __tablename__ = "producto"
26         id = Column(Integer, primary_key=True, autoincrement=True)
27         ventas = relationship("Product_Sale", backref="producto", cascade="delete,merge")
28         nombre = Column(String)
29         precio = Column(Float)
30
31     class Product_Sale(Base):
32         __tablename__ = "producto_venta"
33         id = Column(Integer, primary_key=True, autoincrement=True)
34         id_venta = Column(Integer, ForeignKey("venta.id", ondelete="CASCADE"))
35         id_producto = Column(Integer, ForeignKey('producto.id'))
```

- client.py:

```
app > routers > client.py > eliminar_cliente
1  from fastapi import APIRouter, Depends
2  from app.schemas import Client, UpdatedClient
3  from app.db.database import get_db
4  from sqlalchemy.orm import Session
5  from app.db import models
6  from typing import List
7
8  router = APIRouter(
9      prefix="/client",
10     tags=["Client"]
11 )
12
13 # obtener
14 @router.get("")
15 def obtener_clientes(db:Session=Depends(get_db)):
16     data = db.query(models.Client).all()
17     print(data)
18     return data
19
20 @router.get("/{client_id}")
21 def obtener_cliente(client_id:int, db:Session=Depends(get_db)):
22     cliente = db.query(models.Client).filter(models.Client.id == client_id).first()
23     if not cliente:
24         return {"Respuesta": "Cliente no encontrado"}
25     return cliente
26
27 # crear
28 @router.post("")
29 def crear_cliente(client:Client, db:Session=Depends(get_db)):
30     cliente = client.model_dump()
31     # clientes.append(cliente)
32     nuevo_cliente = models.Client(
33         nombre = cliente["nombre"],
34         direccion = cliente["direccion"],
35         telefono = cliente["telefono"],
36         correo = cliente["correo"]
37     )
38     db.add(nuevo_cliente)
39     db.commit()
40     db.refresh(nuevo_cliente)
41     return {"Respuesta": "Cliente creado con éxito"}
42
43 # modificar
44 @router.patch("/{client_id}")
45 def actualizar_cliente(client_id:int, updatedClient:UpdatedClient, db:Session=Depends(get_db)):
46     cliente = db.query(models.Client).filter(models.Client.id == client_id)
47     if not cliente.first():
48         return {"Respuesta": "Cliente no encontrado"}
49     cliente.update(updatedClient.model_dump(exclude_unset=True))
50     db.commit()
51     return {"Respuesta": "Cliente actualizado con éxito"}
52
53 # eliminar
54 @router.delete("/{client_id}")
55 def eliminar_cliente(client_id:int, db:Session=Depends(get_db)):
56     cliente = db.query(models.Client).filter(models.Client.id == client_id).first()
57     if not cliente:
58         return {"Respuesta": "Cliente no encontrado"}
59     db.delete(cliente)
60     db.commit()
61     return {"Respuesta": "Cliente eliminado con éxito"}
```

- product\_sale.py:

```
app > routers > product_sale.py > obtener_producto_venta
1  from fastapi import APIRouter, Depends
2  from app.schemas import ProductSale, UpdatedProductSale
3  from app.db.database import get_db
4  from sqlalchemy.orm import Session
5  from app.db import models
6  from typing import List
7
8  router = APIRouter(
9      prefix="/product-sale",
10     tags=["Product sale"]
11 )
12
13 # obtener
14 @router.get("")
15 def obtener_productos_ventas(db:Session=Depends(get_db)):
16     data = db.query(models.Product_Sale).all()
17     print(data)
18     return data
19
20 @router.get("/{product_sale_id}")
21 def obtener_producto_venta(product_sale_id:int, db:Session=Depends(get_db)):
22     producto_venta = db.query(models.Product_Sale).filter(models.Product_Sale.id == product_sale_id).first()
23     if not producto_venta:
24         return {"Respuesta": "Relación producto-venta no encontrada"}
25     return producto_venta
26
27 # crear
28 @router.post("")
29 def crear_producto_venta(product_sale:ProductSale, db:Session=Depends(get_db)):
30     producto_venta = product_sale.model_dump()
31     nuevo_producto_venta = models.Product_Sale(
32         id_venta = producto_venta["id_venta"],
33         id_producto = producto_venta["id_producto"]
34     )
35     db.add(nuevo_producto_venta)
36     db.commit()
37     db.refresh(nuevo_producto_venta)
38     return {"Respuesta": "Relación producto-venta creada con éxito"}
39
40 # modificar
41 @router.patch("/{product_sale_id}")
42 def actualizar_producto_venta(product_sale_id:int, updatedProductSale:UpdatedProductSale, db:Session=Depends(get_db)):
43     producto_venta = db.query(models.Product_Sale).filter(models.Product_Sale.id == product_sale_id)
44     if not producto_venta.first():
45         return {"Respuesta": "Relación producto-venta no encontrada"}
46     producto_venta.update(updatedProductSale.model_dump(exclude_unset=True))
47     db.commit()
48     return {"Respuesta": "Relación producto-venta actualizada con éxito"}
49
50 # eliminar
51 @router.delete("/{product_sale_id}")
52 def eliminar_producto_venta(product_sale_id:int, db:Session=Depends(get_db)):
53     producto_venta = db.query(models.Product_Sale).filter(models.Product_Sale.id == product_sale_id).first()
54     if not producto_venta:
55         return {"Respuesta": "Relación producto-venta no encontrada"}
56     db.delete(producto_venta)
57     db.commit()
58     return {"Respuesta": "Relación producto-venta con éxito"}
```

- producto.py:

```
app > routers > product.py > ...
1  from fastapi import APIRouter, Depends
2  from app.schemas import Product, UpdatedProduct
3  from app.db.database import get_db
4  from sqlalchemy.orm import Session
5  from app.db import models
6  from typing import List
7
8  router = APIRouter(
9      prefix="/product",
10     tags=["Product"]
11 )
12
13 # obtener
14 @router.get("")
15 def obtener_productos(db:Session=Depends(get_db)):
16     data = db.query(models.Product).all()
17     print(data)
18     return data
19
20 @router.get("/{product_id}")
21 def obtener_producto(product_id:int, db:Session=Depends(get_db)):
22     producto = db.query(models.Product).filter(models.Product.id == product_id).first()
23     if not producto:
24         return {"Respuesta": "Producto no encontrado"}
25     return producto
26
27 # crear
28 @router.post("")
29 def crear_producto(product:Product, db:Session=Depends(get_db)):
30     producto = product.model_dump()
31     nuevo_producto = models.Product(
32         nombre = producto["nombre"],
33         precio = producto["precio"]
34     )
35     db.add(nuevo_producto)
36     db.commit()
37     db.refresh(nuevo_producto)
38     return {"Respuesta": "Producto creado con éxito"}
39
40 # modificar
41 @router.patch("/{product_id}")
42 def actualizar_producto(product_id:int, updatedProduct:UpdatedProduct, db:Session=Depends(get_db)):
43     producto = db.query(models.Product).filter(models.Product.id == product_id)
44     if not producto.first():
45         return {"Respuesta": "Producto no encontrado"}
46     producto.update(updatedProduct.model_dump(exclude_unset=True))
47     db.commit()
48     return {"Respuesta": "Producto actualizado con éxito"}
49
50 # eliminar
51 @router.delete("/{product_id}")
52 def eliminar_producto(product_id:int, db:Session=Depends(get_db)):
53     producto = db.query(models.Product).filter(models.Product.id == product_id).first()
54     if not producto:
55         return {"Respuesta": "Producto no encontrado"}
56     db.delete(producto)
57     db.commit()
58     return {"Respuesta": "Producto eliminado con éxito"}
```

- sale.py:

```
app > routers > sale.py > eliminar_venta
1  from fastapi import APIRouter, Depends
2  from app.schemas import Sale, UpdatedSale
3  from app.db.database import get_db
4  from sqlalchemy.orm import Session
5  from app.db import models
6  from typing import List
7
8  router = APIRouter(
9      prefix="/sale",
10     tags=["Sale"]
11 )
12
13 # obtener
14 @router.get("")
15 def obtener_ventas(db:Session=Depends(get_db)):
16     data = db.query(models.Sale).all()
17     print(data)
18     return data
19
20 @router.get("/{sale_id}")
21 def obtener_venta(sale_id:int, db:Session=Depends(get_db)):
22     venta = db.query(models.Sale).filter(models.Sale.id == sale_id).first()
23     if not venta:
24         return {"Respuesta": "Venta no encontrada"}
25     return venta
26
27 # crear
28 @router.post("")
29 def crear_venta(sale:Sale, db:Session=Depends(get_db)):
30     venta = sale.model_dump()
31     nueva_venta = models.Sale(
32         id_cliente = venta["id_cliente"],
33         fecha = venta["fecha"],
34         ultima_modificacion = venta["ultima_modificacion"]
35     )
36     db.add(nueva_venta)
37     db.commit()
38     db.refresh(nueva_venta)
39     return {"Respuesta": "Venta creada con éxito"}
40
41 # modificar
42 @router.patch("/{sale_id}")
43 def actualizar_venta(sale_id:int, updatedSale:UpdatedSale, db:Session=Depends(get_db)):
44     venta = db.query(models.Sale).filter(models.Sale.id == sale_id)
45     if not venta.first():
46         return {"Respuesta": "Venta no encontrada"}
47     venta.update(updatedSale.model_dump(exclude_unset=True))
48     db.commit()
49     return {"Respuesta": "Venta actualizada con éxito"}
50
51 # eliminar
52 @router.delete("/{sale_id}")
53 def eliminar_venta(sale_id:int, db:Session=Depends(get_db)):
54     venta = db.query(models.Sale).filter(models.Sale.id == sale_id).first()
55     if not venta:
56         return {"Respuesta": "Venta no encontrada"}
57     db.delete(venta)
58     db.commit()
59     return {"Respuesta": "Venta eliminada con éxito"}
```



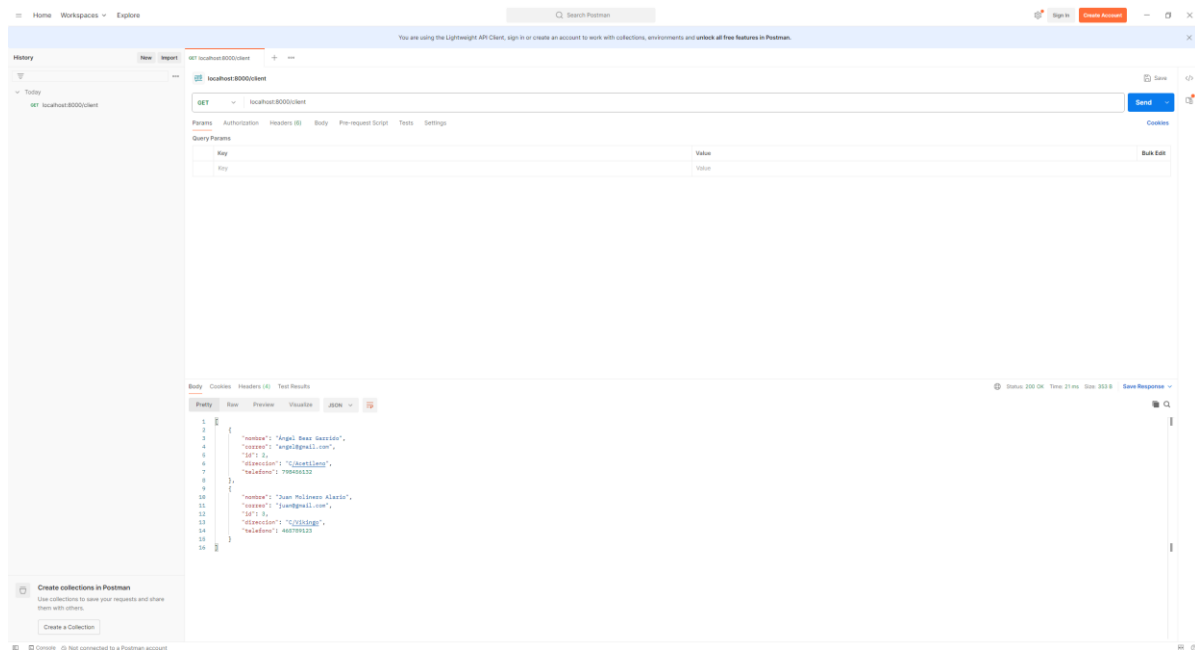
En el fichero requirements se encuentran las librerías que se deben instalar para el correcto funcionamiento de la API. Estas son:

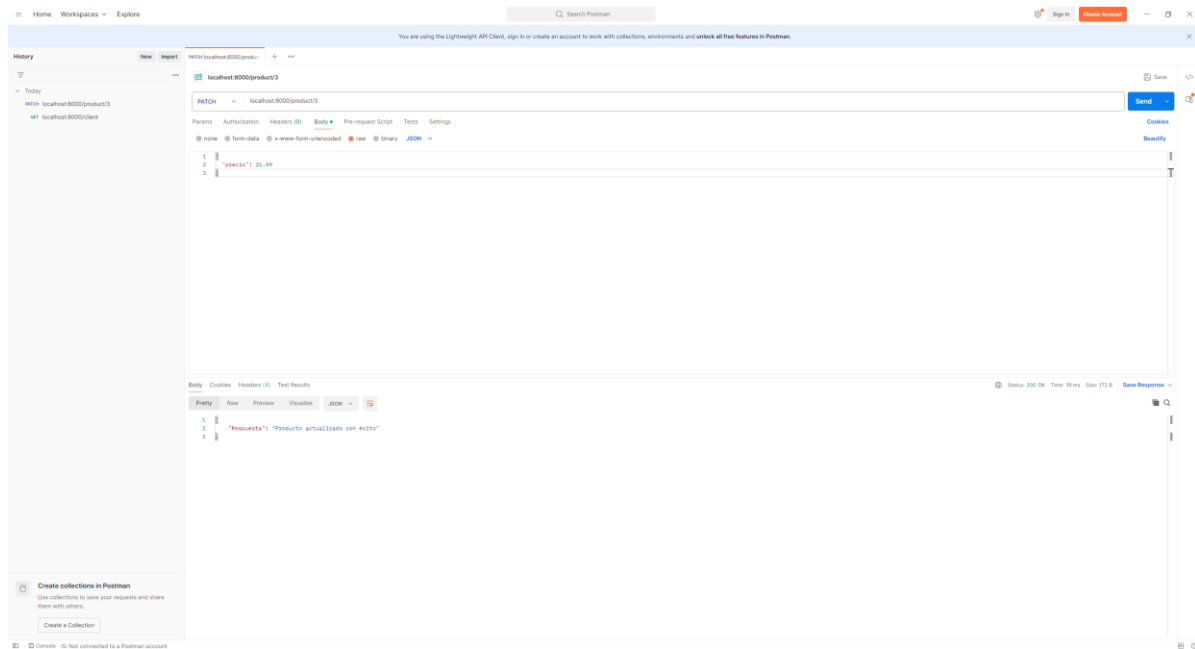
```
requirements.txt
1 fastapi
2 uvicorn
3 psycopg2
4 SQLAlchemy
```

- fastapi: es un framework de desarrollo web moderno y de gran rendimiento utilizado para construir APIs con Python basadas en el standard de pistas tipadas de Python.
- uvicorn: es una implementación de servidor web ASGI para Python.
- psycopg2: es el adaptador de bases de datos PostgreSQL más popular para programación en Python.
- SQLAlchemy: es una herramienta de desarrollo y mapeo de objetos relacionales que da a los desarrolladores la flexibilidad de SQL.

## 4.4 Pruebas

Pruebas de funcionamiento realizadas con Postman.





## 4.5 Despliegue de la aplicación

El despliegue de la aplicación se realizará en local mediante el uso del comando Python `.\main.py` que alberga la aplicación seleccionada en un servidor local en el puerto 8000. De esta forma podremos acceder a la API desde la dirección `127.0.0.1:8000`. En la dirección `127.0.0.1:8000/docs` podremos ver su documentación en Swagger. Por último, la interfaz gráfica de la base de datos será accesible desde el puerto 80 una vez inicializados los contenedores de Docker, por lo que será visible en la dirección `127.0.0.1:80`.

# 5 Manuales

## 5.1 Manual de usuario

El funcionamiento de la API consta de cuatro partes diferentes pero interconectadas. Se trata de los siguientes apartados.

**Cliente:**

## Creación de un nuevo cliente.

Client

POST /client/crear\_cliente

Parameters

No parameters

Request body

application/json

```
{
  "nombre": "Roberto Martínez de Lado",
  "email": "r.martinez@iesrc.es",
  "password": "12345678",
  "confirmar": "12345678"
}
```

Execute

Clear

Responses

200

```
{
  "mensaje": "cliente creado con éxito"
}
```

200

Response body

```
{
  "mensaje": "cliente creado con éxito"
}
```

Response headers

```
Content-Type: application/json
Content-Length: 41
Date: Mon, 19 Feb 2024 12:17:08 GMT
Server: Apache/2.4.18 (Ubuntu)
```

Responses

Code	Description	Links
200	Successful Response	No links
422	Validation Error	No links

GET /client/{client\_id} Obtener Cliente

PUT /client/{client\_id} Actualizar Cliente

Mostrar los clientes en la base de datos.

The image displays two screenshots of the Swagger UI for a FastAPI application, showing the endpoints for managing clients.

**Top Screenshot: GET /client**

- Endpoint:** GET /client
- Operation:** Obtener Clientes
- Parameters:** No parameters
- Response:** 200 OK
- Response Body:**

```

{
  "clients": [
    {
      "id": 1,
      "name": "Juan Martínez de Lera",
      "email": "juan.martinez@lca.com",
      "age": 35,
      "phone": "912345678",
      "address": "C/Alcalá, 123",
      "city": "Madrid",
      "country": "España"
    },
    {
      "id": 2,
      "name": "Ana María García",
      "email": "ana.maria@lca.com",
      "age": 28,
      "phone": "987654321",
      "address": "C/Gran Vía, 456",
      "city": "Madrid",
      "country": "España"
    },
    {
      "id": 3,
      "name": "Carlos Rodríguez",
      "email": "carlos.rodriguez@lca.com",
      "age": 42,
      "phone": "123456789",
      "address": "C/Plaza de España, 789",
      "city": "Madrid",
      "country": "España"
    }
  ]
}

```

**Bottom Screenshot: GET /client/{client\_id}**

- Endpoint:** GET /client/{client\_id}
- Operation:** Obtener Cliente
- Parameters:** client\_id (path)
- Response:** 200 OK
- Response Body:**

```

{
  "client": {
    "id": 1,
    "name": "Juan Martínez de Lera",
    "email": "juan.martinez@lca.com",
    "age": 35,
    "phone": "912345678",
    "address": "C/Alcalá, 123",
    "city": "Madrid",
    "country": "España"
  }
}

```

The bottom screenshot also shows a list of endpoints for the 'Product' resource:

- POST /product: Crear Producto
- GET /product: Obtener Productos
- PUT /product/{product\_id}: Actualizar Producto
- DELETE /product/{product\_id}: Eliminar Producto

Mostrar un cliente concreto por medio de su id.

**FastAPI** 0.105.0

**Client**

**GET** /client Obtain Clients

**POST** /client Create Client

**GET** /client/{client\_id} Obtain Client

**Parameters**

Name	Description
client_id	integer (required)

**Responses**

200

```

{
  "name": "Antonio Martinez de Lera",
  "company": "www.damas.com",
  "age": 32,
  "email": "a.lera@damas.com",
  "telephone": "912345678"
}

```

**Response headers**

```

content-length: 128
content-type: application/json
date: Mon, 10 Feb 2025 12:46:41 GMT
server: uvicorn

```

**PATCH** /client/{client\_id} Actualize Client

**DELETE** /client/{client\_id} Eliminate Client

**Product**

**GET** /product Obtain Products

**POST** /product Create Products

**GET** /product/{product\_id} Obtain Product

**PATCH** /product/{product\_id} Actualize Product

**DELETE** /product/{product\_id} Eliminate Product

**Sale**

**Responses**

200 Successful Response

Media type: application/json

Example Value: Schema

```

{
  "id": 1
}

```

422 Validation Error

Media type: application/json

Example Value: Schema

```

{
  "email": {
    "type": "string"
  },
  "name": {
    "type": "string"
  },
  "telephone": {
    "type": "string"
  }
}

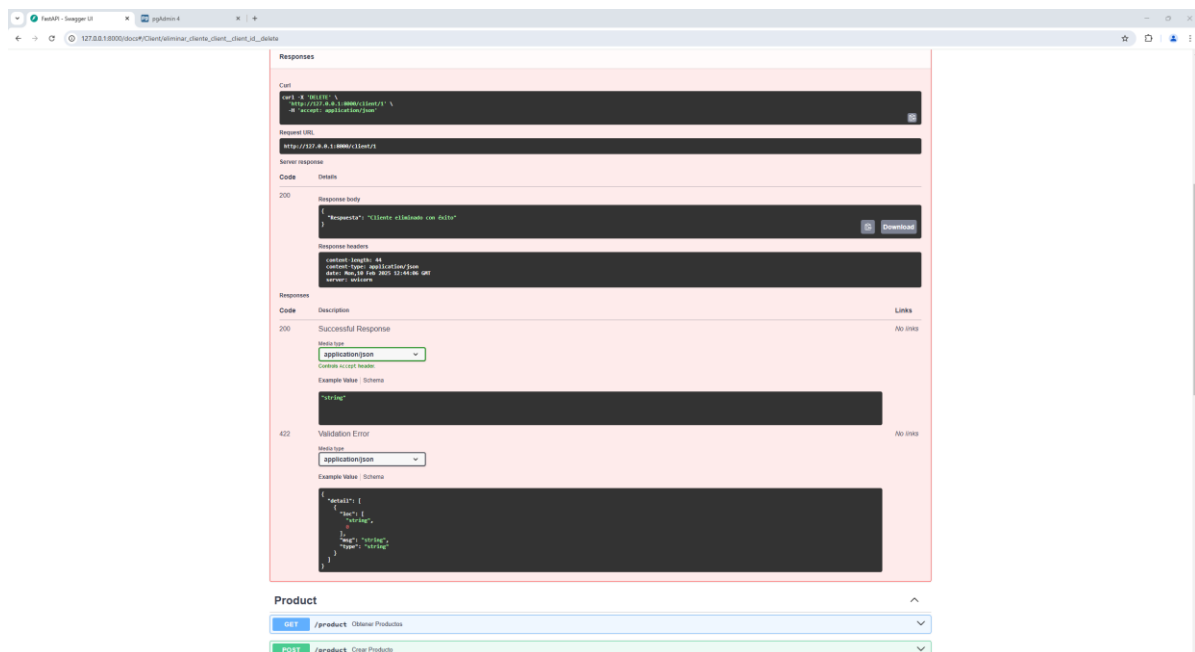
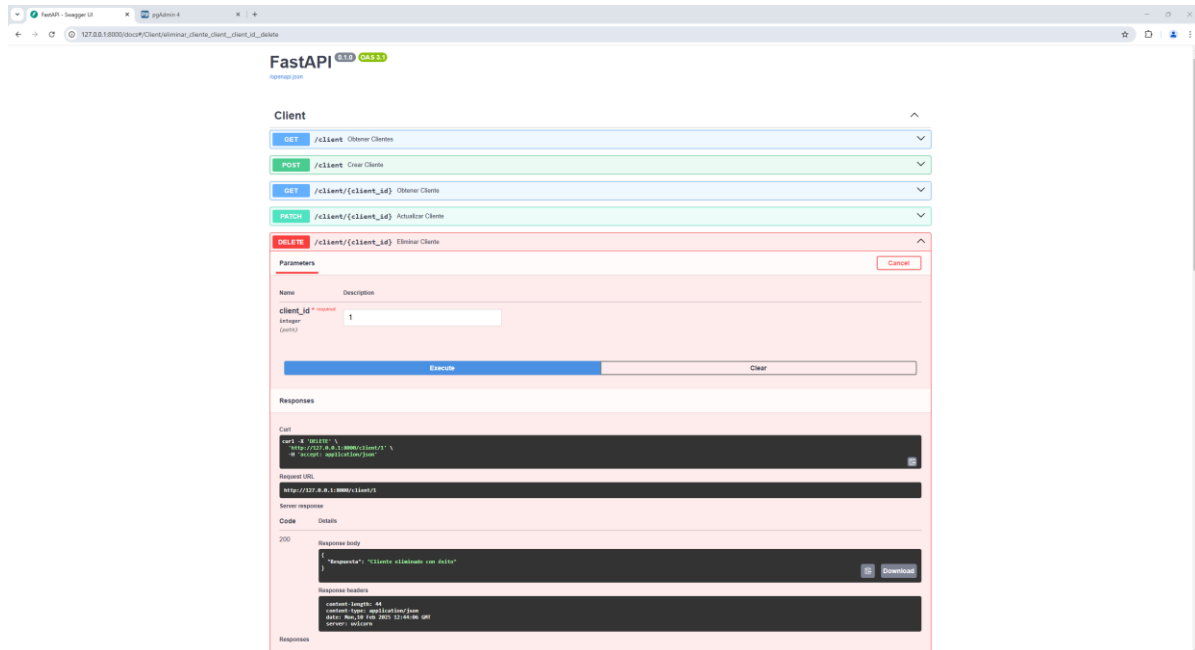
```

Modificar los datos correspondientes al cliente con el id proporcionado.

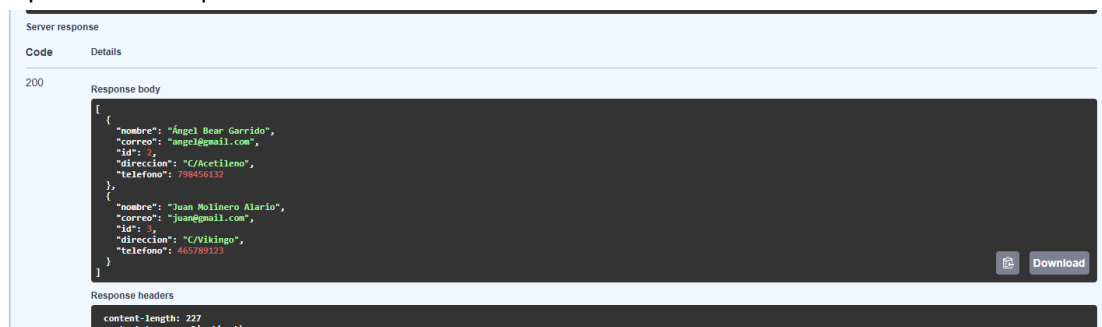
The first screenshot shows the Swagger UI interface for the PATCH endpoint `/client/{client_id}`. The parameters section shows `client_id` with a value of 1. The request body is a JSON object: `{ "id_cliente": "C/Luenda 87" }`. The response section shows a 200 status code with a response body: `{ "mensaje": "Cliente actualizado con éxito" }`.

The second screenshot shows the Swagger UI interface for the PATCH endpoint `/client/{client_id}`. The parameters section shows `client_id` with a value of 1. The request body is a JSON object: `{ "id_cliente": "C/Luenda 87" }`. The response section shows a 200 status code with a response body: `{ "mensaje": "Cliente actualizado con éxito" }`. Below the response section, there is a section for the DELETE endpoint `/client/{client_id}`.

Eliminar el cliente correspondiente al id proporcionado.



Comprobación de que el cliente se ha eliminado con éxito.



## Producto:

### Creación de un nuevo producto.

The screenshot shows the Swagger UI interface for a REST API. The selected endpoint is `POST /product` with the description "Crear Producto". The request body is set to `application/json` and contains the following JSON:

```
{
  "nombre": "Helicóptero",
  "precio": 400.00
}
```

The response status is 200, and the response body is:

```
{
  "mensaje": "Producto creado con éxito"
}
```

The response headers include:

```
Content-Length: 41
Content-Type: application/json
Date: Mon, 10 Feb 2025 12:47:12 GMT
Server: Apache
```

The screenshot shows the Swagger UI interface displaying a list of endpoints for the `Product` and `Sale` resources.

**Product Resource:**

- `GET /product/{product_id}` Obtener Productos
- `POST /product/{product_id}` Actualizar Producto
- `DELETE /product/{product_id}` Eliminar Producto

**Sale Resource:**

- `GET /sale` Obtener Ventas
- `POST /sale` Crear Venta



Mostrar los productos en la base de datos.

The screenshot shows the Swagger UI for the endpoint `GET /product: Obtener Productos`. The response body is a JSON array of product objects:

```

[
  {
    "id": 1,
    "name": "Producto 1",
    "description": "Descripción del producto 1"
  },
  {
    "id": 2,
    "name": "Producto 2",
    "description": "Descripción del producto 2"
  },
  {
    "id": 3,
    "name": "Producto 3",
    "description": "Descripción del producto 3"
  },
  {
    "id": 4,
    "name": "Producto 4",
    "description": "Descripción del producto 4"
  },
  {
    "id": 5,
    "name": "Producto 5",
    "description": "Descripción del producto 5"
  }
]

```

The response headers are:

```

Content-Length: 577
Content-Type: application/json
Date: Mon, 10 Jun 2025 12:45:45 GMT
Server: nginx

```

The response status is 200, indicating a successful response.

The screenshot shows the Swagger UI for the endpoint `GET /product: Obtener Productos`. The response body is a JSON array of product objects:

```

[
  {
    "id": 1,
    "name": "Producto 1",
    "description": "Descripción del producto 1"
  },
  {
    "id": 2,
    "name": "Producto 2",
    "description": "Descripción del producto 2"
  },
  {
    "id": 3,
    "name": "Producto 3",
    "description": "Descripción del producto 3"
  },
  {
    "id": 4,
    "name": "Producto 4",
    "description": "Descripción del producto 4"
  },
  {
    "id": 5,
    "name": "Producto 5",
    "description": "Descripción del producto 5"
  }
]

```

The response headers are:

```

Content-Length: 577
Content-Type: application/json
Date: Mon, 10 Jun 2025 12:45:45 GMT
Server: nginx

```

The response status is 200, indicating a successful response.

Below the response details, there is a list of other endpoints:

- `POST /product: Crear Producto`
- `GET /product/{product_id}: Obtener Producto`
- `PUT /product/{product_id}: Actualizar Producto`
- `DELETE /product/{product_id}: Eliminar Producto`
- `GET /sale: Obtener Ventas`

Mostrar un producto concreto por medio de su id.

Product

GET /product Obtain Products

POST /product Create Products

GET /product/{product\_id} Obtain Product

Parameters

Name Description

product\_id <sup>required</sup>  
integer (path) 2

Example Clear

Responses

200

Response body

```
{
  "id": 2,
  "name": "Producto 2",
  "description": "Descripción del producto 2"
}
```

Response headers

```
Content-Length: 48
Content-Type: application/json
Date: Mon, 10 Sep 2024 12:16:18 GMT
Server: Redhat
```

Responses

Code	Description	Links
200	Successful Response	No links

Media type: application/json

Example Value: Schema

200

Response body

```
{
  "id": 2,
  "name": "Producto 2",
  "description": "Descripción del producto 2"
}
```

Response headers

```
Content-Length: 48
Content-Type: application/json
Date: Mon, 10 Sep 2024 12:16:18 GMT
Server: Redhat
```

Responses

Code	Description	Links
200	Successful Response	No links
422	Validation Error	No links

Media type: application/json

Example Value: Schema

```
{
  "id": 2,
  "name": "Producto 2",
  "description": "Descripción del producto 2"
}
```

PATCH /product/{product\_id} Actualizar Product

DELETE /product/{product\_id} Eliminar Product

Sale

GET /sale Obtain Ventes

POST /sale Create Venta

GET /sale/{sale\_id} Obtain Venta

Modificar los datos correspondientes al producto con el id proporcionado.

The screenshot shows the Swagger UI interface for a REST API. The selected endpoint is `PATCH /product/{product_id}` with the description "Actualizar Producto". The parameters section shows a required path parameter `product_id` with a value of `3`. The request body is set to `application/json` and contains the JSON object `{ "precio": 7.99 }`. The response section shows a `200` status code with a response body containing a JSON object: `{ "mensaje": "Producto actualizado con éxito" }`.

This screenshot shows the detailed response for the `PATCH /product/{product_id}` endpoint. The status code is `200` (Successful Response). The response body is a JSON object: `{ "mensaje": "Producto actualizado con éxito" }`. The response headers include `Content-Length: 47`, `Content-Type: application/json`, `Server: Apache/2.4.18 (Ubuntu)`, and `Set-Cookie: PHPSESSID=...`. The response is categorized as a "Successful Response" with a status code of `200`. Below this, there is a "Validation Error" section with a status code of `422`, which is currently collapsed. At the bottom of the interface, there are links for other endpoints: `DELETE /product/{product_id}` (Eliminar Producto) and `GET /sale` (Obtener Ventas).

Eliminar el producto correspondiente al id proporcionado.

The first screenshot shows the REST client interface for the DELETE endpoint `/product/{product_id}`. The parameter `product_id` is set to 2. The response is a 200 status code with a message: `{ "mensaje": "Producto eliminado con éxito" }`.

The second screenshot shows the same endpoint but with a validation error response (422) when the media type is set to `application/json`. The error message is: `{ "mensaje": { "type": "validation_error", "errors": { "product_id": "El valor no es un número entero." } } }`.

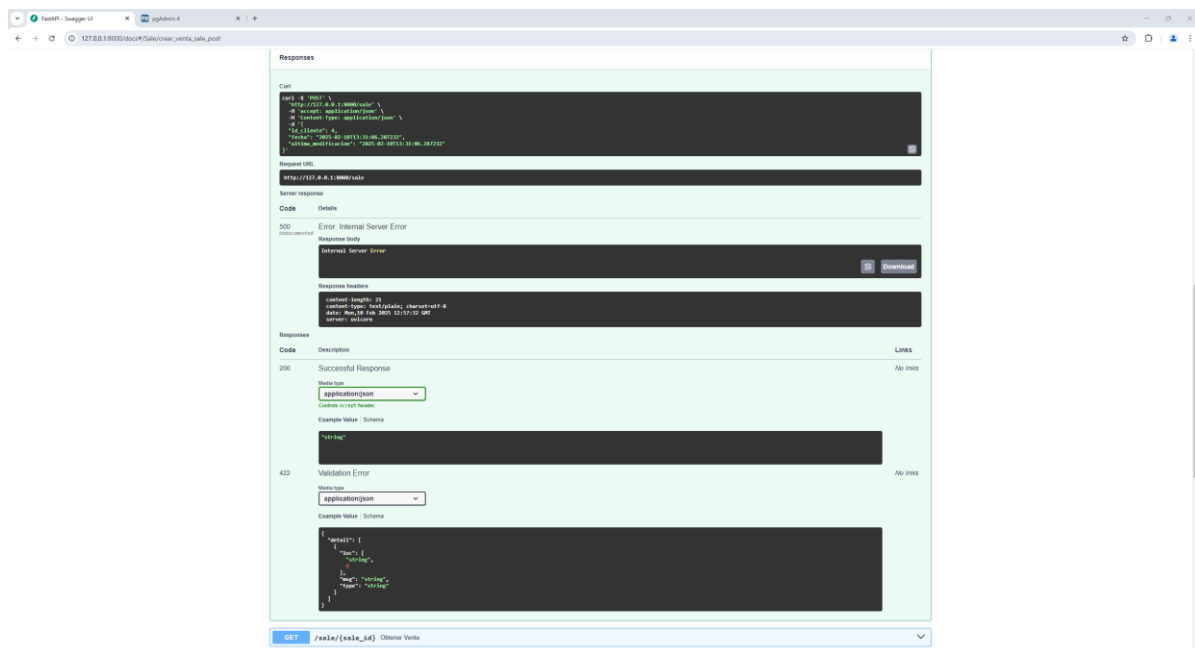
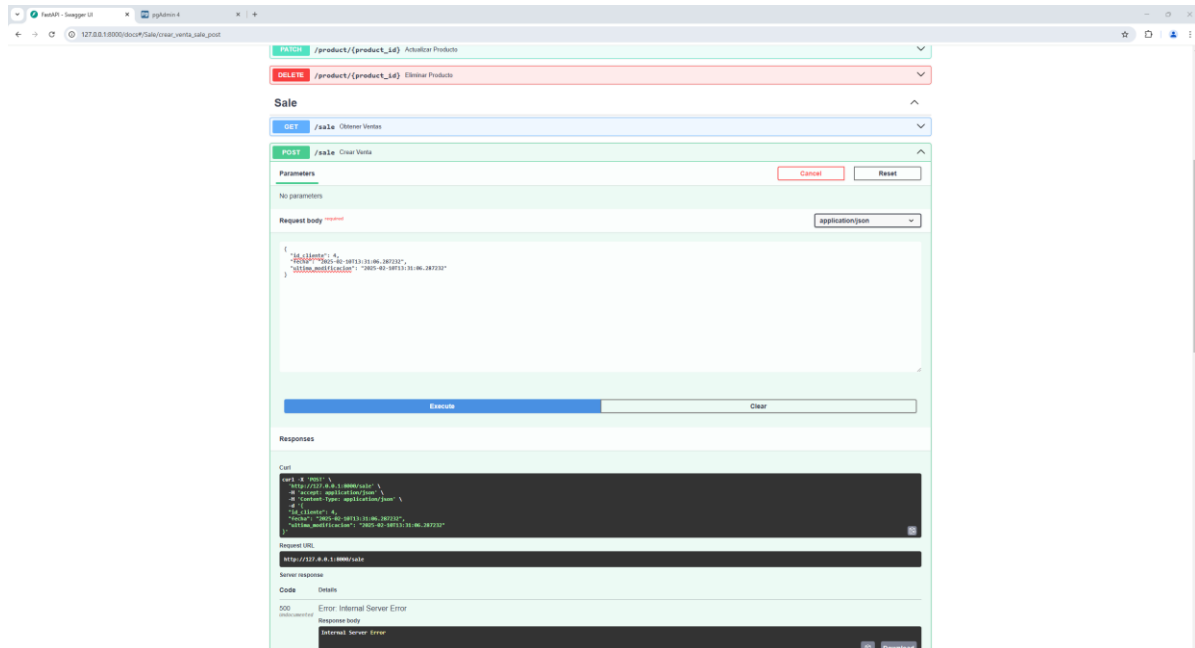
Comprobación de que el producto se ha eliminado con éxito.

The screenshot shows the REST client interface for the GET endpoint `/sale`. The response is a 200 status code with a message: `{ "id": 1, "precio": 499.99, "nombre": "Televisión" }, { "id": 4, "precio": 19.99, "nombre": "Alfombra" }, { "id": 3, "precio": 7.99, "nombre": "Peluche" }`.

Creación de una nueva venta.



Si se intenta crear una venta para un cliente que no existe devuelve un error.



Mostrar las ventas en la base de datos.

The screenshot shows the Swagger UI interface for a REST API. The selected endpoint is `GET /sale` with the description "Obtener Ventas". The parameters section is empty. The response section shows a 200 status with a JSON array of sales data. The response body is as follows:

```

{
  "data": [
    {
      "id_venta": 1,
      "id_producto": 1,
      "cantidad": 10,
      "precio": 100,
      "total": 1000,
      "fecha": "2023-01-01 10:00:00"
    },
    {
      "id_venta": 2,
      "id_producto": 2,
      "cantidad": 5,
      "precio": 200,
      "total": 1000,
      "fecha": "2023-01-01 10:00:00"
    },
    {
      "id_venta": 3,
      "id_producto": 3,
      "cantidad": 2,
      "precio": 500,
      "total": 1000,
      "fecha": "2023-01-01 10:00:00"
    }
  ]
}

```

The response headers are also visible:

```

Content-Length: 317
Content-Type: application/json
Date: Mon, 10 Jan 2023 17:06:11 GMT
Server: Apache/2.4.18

```

This screenshot shows the same Swagger UI interface, but with the response body expanded to show the JSON structure. The response is a 200 status with a JSON array of sales data. The response body is as follows:

```

{
  "data": [
    {
      "id_venta": 1,
      "id_producto": 1,
      "cantidad": 10,
      "precio": 100,
      "total": 1000,
      "fecha": "2023-01-01 10:00:00"
    },
    {
      "id_venta": 2,
      "id_producto": 2,
      "cantidad": 5,
      "precio": 200,
      "total": 1000,
      "fecha": "2023-01-01 10:00:00"
    },
    {
      "id_venta": 3,
      "id_producto": 3,
      "cantidad": 2,
      "precio": 500,
      "total": 1000,
      "fecha": "2023-01-01 10:00:00"
    }
  ]
}

```

The response headers are also visible:

```

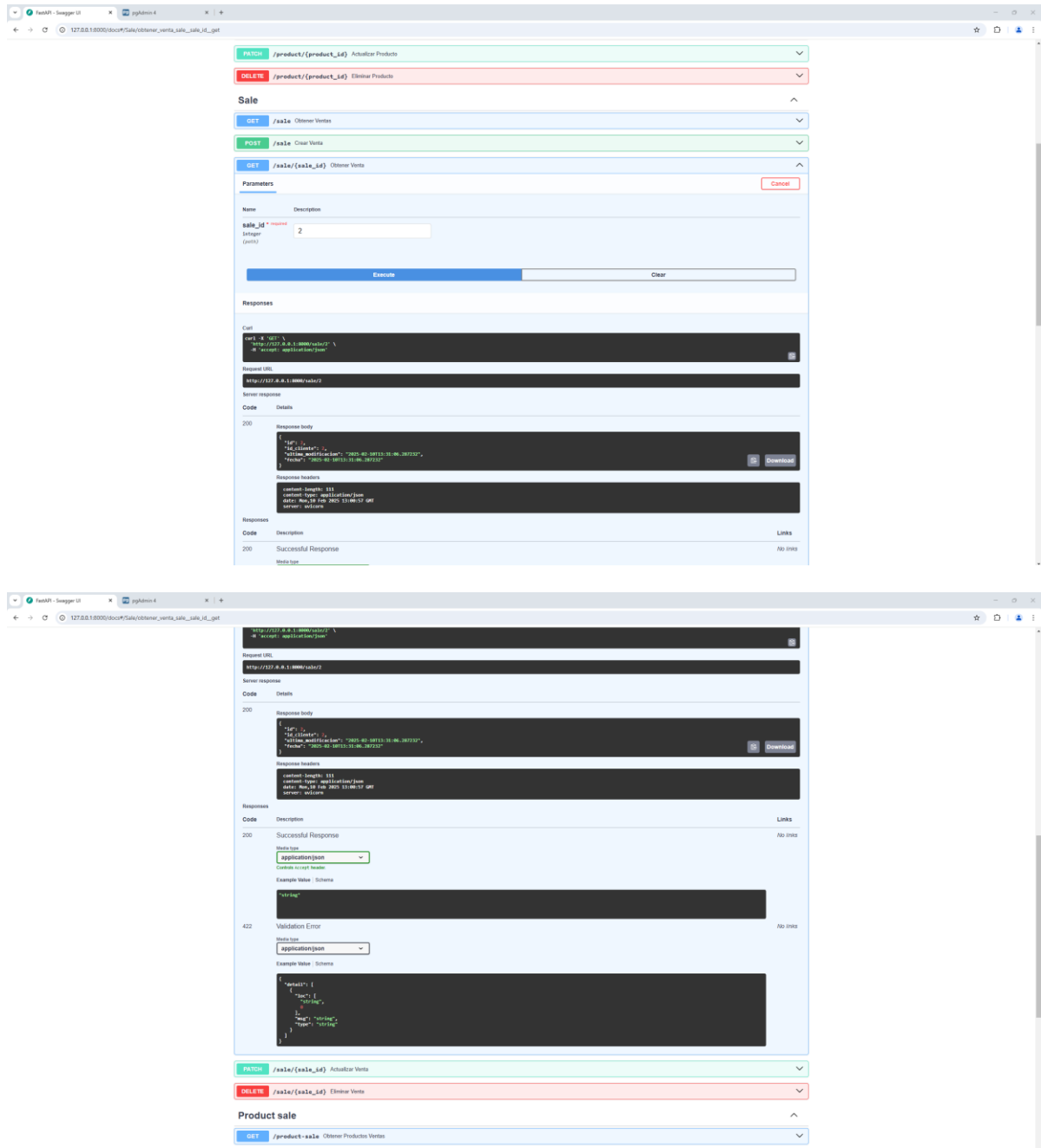
Content-Length: 317
Content-Type: application/json
Date: Mon, 10 Jan 2023 17:06:11 GMT
Server: Apache/2.4.18

```

Below the response details, there is a list of other API endpoints:

- `POST /sale` Crear Venta
- `GET /sale/{sale_id}` Obtener Venta
- `PATCH /sale/{sale_id}` Actualizar Venta
- `DELETE /sale/{sale_id}` Eliminar Venta
- `GET /product-sale` Obtener Productos Ventas
- `POST /product-sale` Crear Productos Ventas
- `GET /product-sale/{product_sale_id}` Obtener Productos Ventas

Mostrar una venta concreta por medio de su id.



The image displays two screenshots of the Swagger UI interface for an API, showing the details of the `GET /sale/{sale_id}` endpoint.

**Top Screenshot:** The endpoint is selected, and the "Responses" tab is active. It shows a successful response (200) with a JSON body. The response body is a JSON object containing sale details:

```

{
  "id": 1,
  "date": "2023-03-10T10:00:00Z",
  "total": 100.0,
  "items": [
    {
      "id": 1,
      "name": "Producto A",
      "price": 50.0,
      "quantity": 2
    },
    {
      "id": 2,
      "name": "Producto B",
      "price": 50.0,
      "quantity": 1
    }
  ]
}

```

**Bottom Screenshot:** The endpoint is selected, and the "Responses" tab is active. It shows a validation error (422) with a JSON body. The response body is a JSON object containing an error message:

```

{
  "error": {
    "code": 422,
    "message": "Invalid media type 'application/json' for response."
  }
}

```



## Modificar los datos correspondientes a la venta con el id proporcionado

The top screenshot shows the Swagger UI for the API. The selected endpoint is `PATCH /sale/{sale_id}` with the description "Actualizar Venta". The parameters section shows a required integer parameter `sale_id` with the value `2`. The request body is set to `application/json` and contains the following JSON:

```
{
  "id": 2,
  "descripcion": "Venta de 10kg de arroz",
  "precio": 1000,
  "fecha": "2023-10-26T12:00:00Z",
  "usuario": "usuario"
}
```

The bottom screenshot shows the response for the `PATCH /sale/{sale_id}` endpoint. The response status is `200` (Successful Response). The response body is `application/json` and contains the following JSON:

```
{
  "id": 2,
  "descripcion": "Venta de 10kg de arroz",
  "precio": 1000,
  "fecha": "2023-10-26T12:00:00Z",
  "usuario": "usuario"
}
```

The bottom screenshot also shows the `DELETE /sale/{sale_id}` endpoint with the description "Eliminar Venta". Below this, there is a section for "Product sale" with endpoints `GET /product-sale` (Obtener Productos Ventas) and `POST /product-sale` (Crear Productos Ventas).

Eliminar la venta correspondiente al id proporcionado.

The top screenshot shows the Swagger UI interface for the 'Sale' API. The 'DELETE /sale/{sale\_id}' endpoint is selected. The 'Parameters' section shows 'sale\_id' with a value of 5. The 'Responses' section shows a successful 200 response with a JSON body containing the sale details.

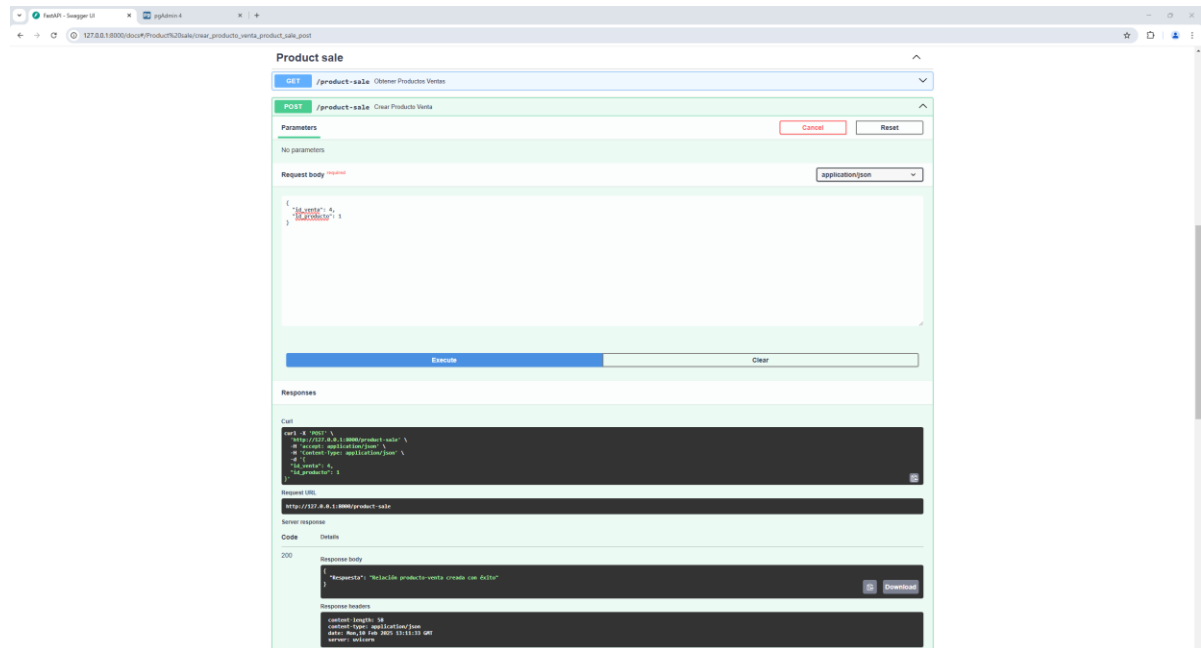
The bottom screenshot shows the same Swagger UI interface, but the 'sale\_id' parameter is set to 422. The 'Responses' section shows a 422 Validation Error response, indicating that the sale with the provided ID does not exist.

Comprobación de que la venta se ha eliminado con éxito.

The screenshot shows the 'Server response' section of the Swagger UI. The response is a 200 status code with a JSON body containing the sale details. The response headers show 'content-length: 225', 'content-type: application/json', 'date: Mon, 10 Feb 2025 13:03:24 GMT', and 'server: uvicorn'.

## Relación producto-venta:

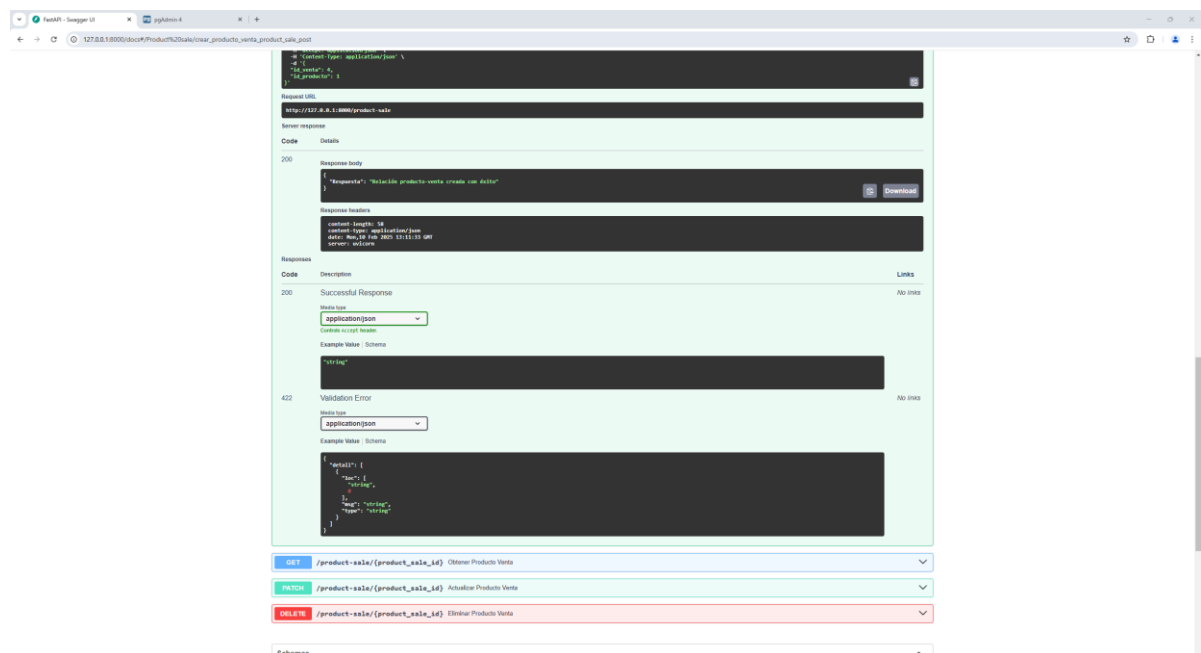
### Creación de una nueva venta.



The screenshot shows the Swagger UI for the 'Product sale' endpoint. The method is POST, and the request body is a JSON object with the following structure:

```
{
  "id_producto": 4,
  "id_venta": 1
}
```

The response is a 200 status code with the message: "Se ha ido producto-venta creado con éxito".



The screenshot shows the Swagger UI for the 'Product sale' endpoint. The response is a 200 status code with the message: "Se ha ido producto-venta creado con éxito". Below the response, there is a table with columns 'Code', 'Description', and 'Links'.

Code	Description	Links
200	Successful Response	No links
422	Validation Error	No links

Si se intenta crear una relación para una venta que no existe devuelve un error.

The top screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** /product-sale
- Request body:** application/json
- Request body content:**

```
{
  "id_venta": 10,
  "id_producto": 1
}
```
- Response:** 500 Internal Server Error
- Response body:** Internal Server Error

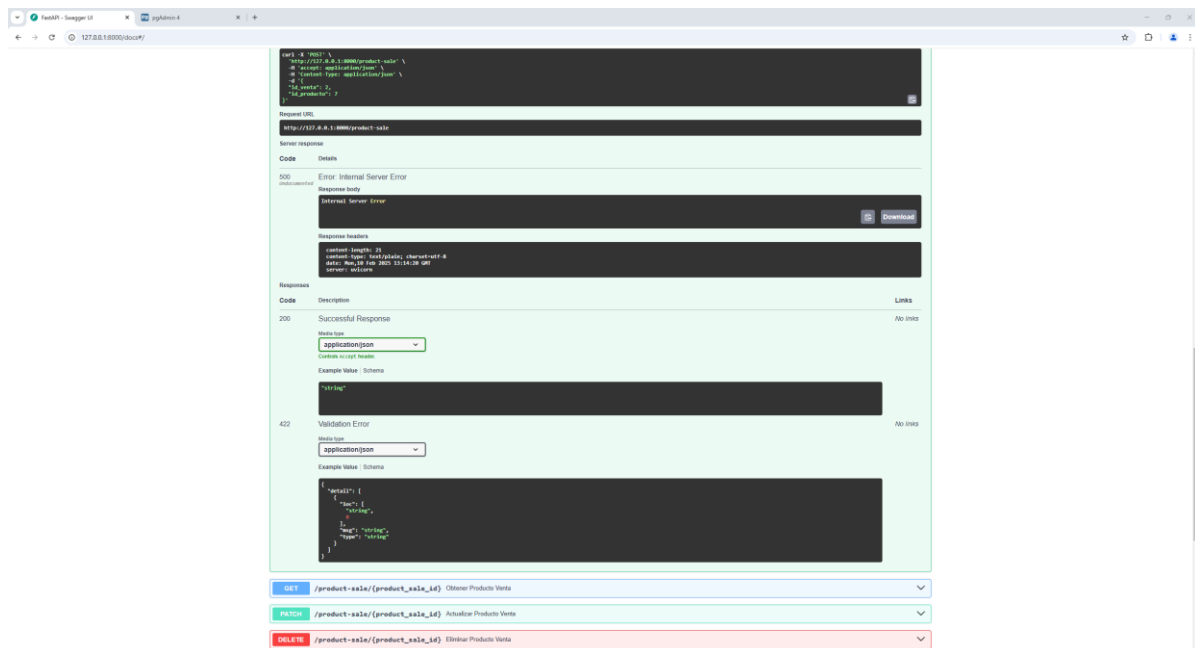
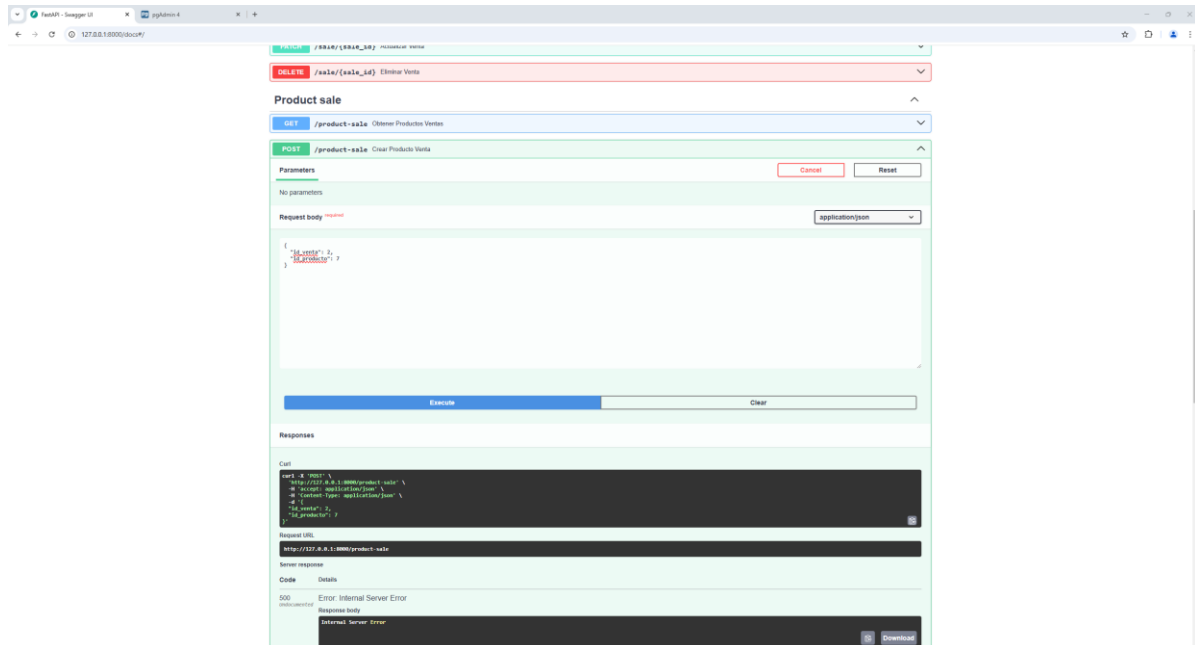
The bottom screenshot shows the same error response in detail, including the response body and headers:

- Code:** 500
- Description:** Error Internal Server Error
- Response body:** Internal Server Error
- Response headers:**

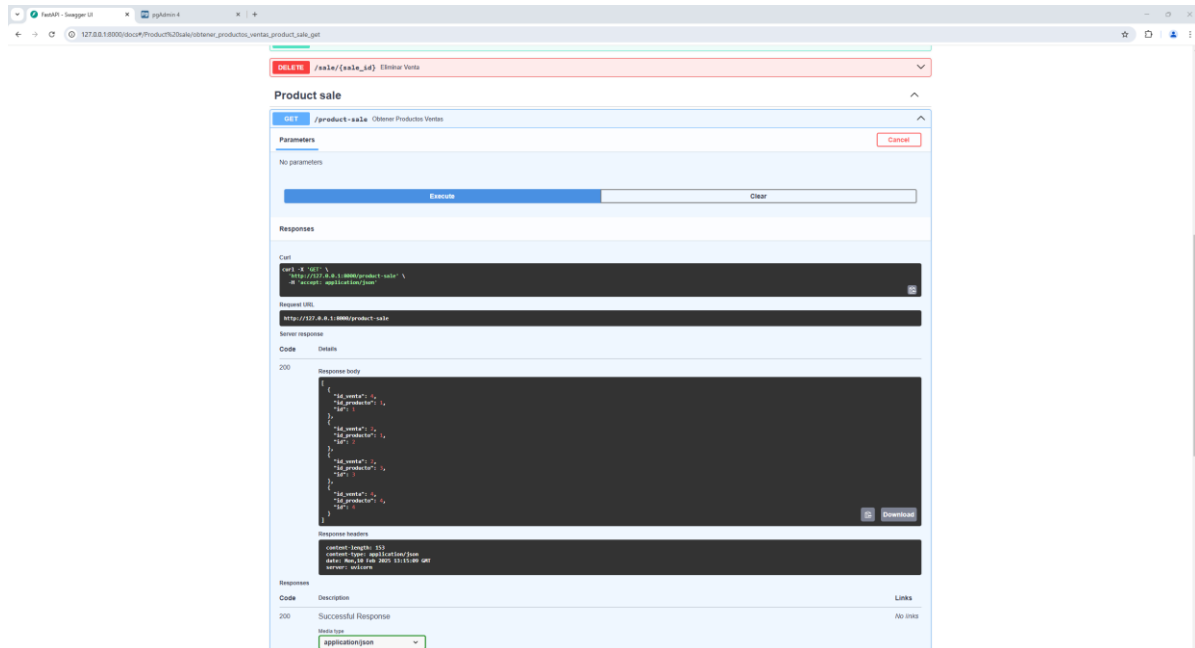
```
Content-Length: 21
Content-Type: text/html; charset=utf-8
Server: Apache/2.4.18 (Ubuntu)
```
- Responses table:**

Code	Description	Links
200	Successful Response	No links
422	Validation Error	No links

Si se intenta crear una relación para un producto que no existe devuelve un error.



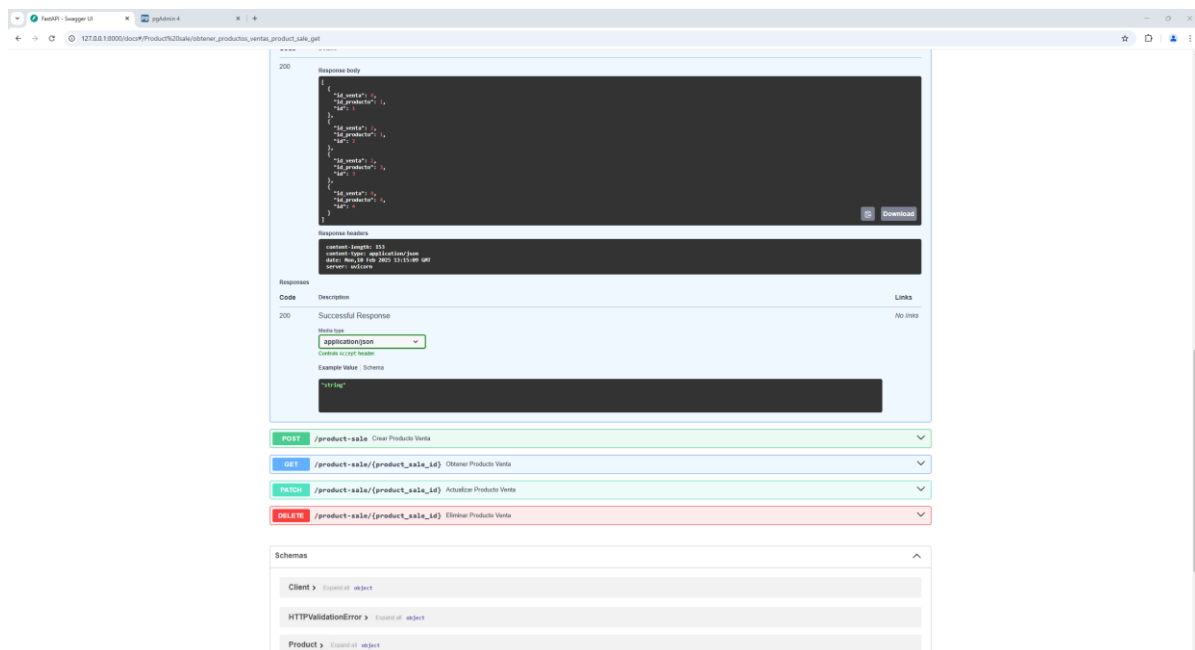
Mostrar las relaciones producto-venta en la base de datos.



Swagger UI interface showing a GET request to `/product-sale`. The response body is a JSON array of objects, each containing `id_venta`, `id_producto`, and `id_usuario`.

```

[
  {
    "id_venta": 1,
    "id_producto": 1,
    "id_usuario": 1
  },
  {
    "id_venta": 2,
    "id_producto": 2,
    "id_usuario": 2
  },
  {
    "id_venta": 3,
    "id_producto": 3,
    "id_usuario": 3
  },
  {
    "id_venta": 4,
    "id_producto": 4,
    "id_usuario": 4
  },
  {
    "id_venta": 5,
    "id_producto": 5,
    "id_usuario": 5
  },
  {
    "id_venta": 6,
    "id_producto": 6,
    "id_usuario": 6
  },
  {
    "id_venta": 7,
    "id_producto": 7,
    "id_usuario": 7
  },
  {
    "id_venta": 8,
    "id_producto": 8,
    "id_usuario": 8
  },
  {
    "id_venta": 9,
    "id_producto": 9,
    "id_usuario": 9
  },
  {
    "id_venta": 10,
    "id_producto": 10,
    "id_usuario": 10
  }
]
  
```



Swagger UI interface showing the API definition for the `/product-sale` endpoint. The response body is a JSON array of objects, each containing `id_venta`, `id_producto`, and `id_usuario`.

```

[
  {
    "id_venta": 1,
    "id_producto": 1,
    "id_usuario": 1
  },
  {
    "id_venta": 2,
    "id_producto": 2,
    "id_usuario": 2
  },
  {
    "id_venta": 3,
    "id_producto": 3,
    "id_usuario": 3
  },
  {
    "id_venta": 4,
    "id_producto": 4,
    "id_usuario": 4
  },
  {
    "id_venta": 5,
    "id_producto": 5,
    "id_usuario": 5
  },
  {
    "id_venta": 6,
    "id_producto": 6,
    "id_usuario": 6
  },
  {
    "id_venta": 7,
    "id_producto": 7,
    "id_usuario": 7
  },
  {
    "id_venta": 8,
    "id_producto": 8,
    "id_usuario": 8
  },
  {
    "id_venta": 9,
    "id_producto": 9,
    "id_usuario": 9
  },
  {
    "id_venta": 10,
    "id_producto": 10,
    "id_usuario": 10
  }
]
  
```

API Definition:

- POST** `/product-sale` Crear Producto Venta
- GET** `/product-sale/{product_sale_id}` Obtener Producto Venta
- PATCH** `/product-sale/{product_sale_id}` Actualizar Producto Venta
- DELETE** `/product-sale/{product_sale_id}` Eliminar Producto Venta

Schemas:

- Client**: `Client` object
- HTTPValidationError**: `HTTPValidationError` object
- Product**: `Product` object

Mostrar una relación producto-venta concreta por medio de su id.

The screenshot shows the Swagger UI interface for a REST API. The selected endpoint is `GET /product-sale/{product_sale_id}` with the description "Obtener Producto Venta". The parameter `product_sale_id` is set to `1`. The "Execute" button has been clicked, and the "Responses" section is expanded, showing a successful response with status code 200. The response body is a JSON object: `{ "id_venta": 1, "id_producto": 1, "id": 1 }`. The response headers include `Content-Length: 37`, `Content-Type: application/json`, `Date: Mon, 16 Feb 2020 11:16:41 GMT`, and `Server: Apache/2.4.18`.

The screenshot shows the "Schemas" section of the Swagger UI. It lists several schemas: `Client`, `HTTPValidationError`, `Product`, and `ProductSale`. Each schema is followed by a link that says "Expand all" and "object". The `ProductSale` schema is highlighted in blue.

The screenshot displays the Swagger UI for the DELETE endpoint `/sale/{sale_id}`. The interface is organized into several sections:

- Endpoints List:** A list of endpoints is shown at the top, including `GET /product-sale`, `POST /product-sale`, `GET /product-sale/{product_sale_id}`, and the selected `DELETE /product-sale/{product_sale_id}`.
- Parameters:** A section for defining parameters. The `product_sale_id` parameter is defined as a required integer (path) with a value of 1.
- Request body:** A section for defining the request body. The body is defined as a JSON object with a single property `id_producto` of type integer, with a value of 3.
- Responses:** A section for defining the response. The response is defined as a 200 status code with a JSON response body: `{ "mensaje": "Se eliminó producto-venta exitosamente con éxito" }`.



Eliminar la relación producto-venta correspondiente al id proporcionado.

The first screenshot shows the Swagger UI interface for the 'Product sale' section. It lists several endpoints, including a DELETE endpoint for deleting a product sale by ID. The 'product\_sale\_id' parameter is set to 1. The response body shows a success message: 'Eliminado producto-venta con éxito'.

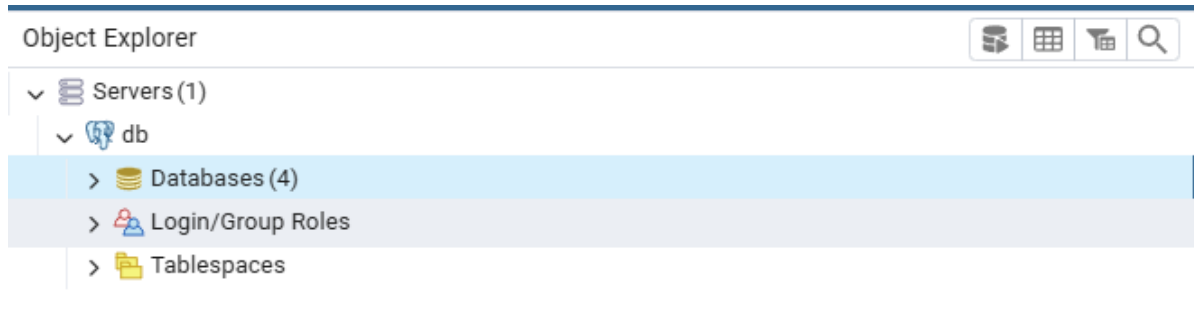
The second screenshot shows the Swagger UI interface for the 'Product sale' section, displaying the response body for the DELETE endpoint. The response is a 200 status code with a success message: 'Eliminado producto-venta con éxito'.

Comprobación de que la relación producto-venta se ha eliminado con éxito.

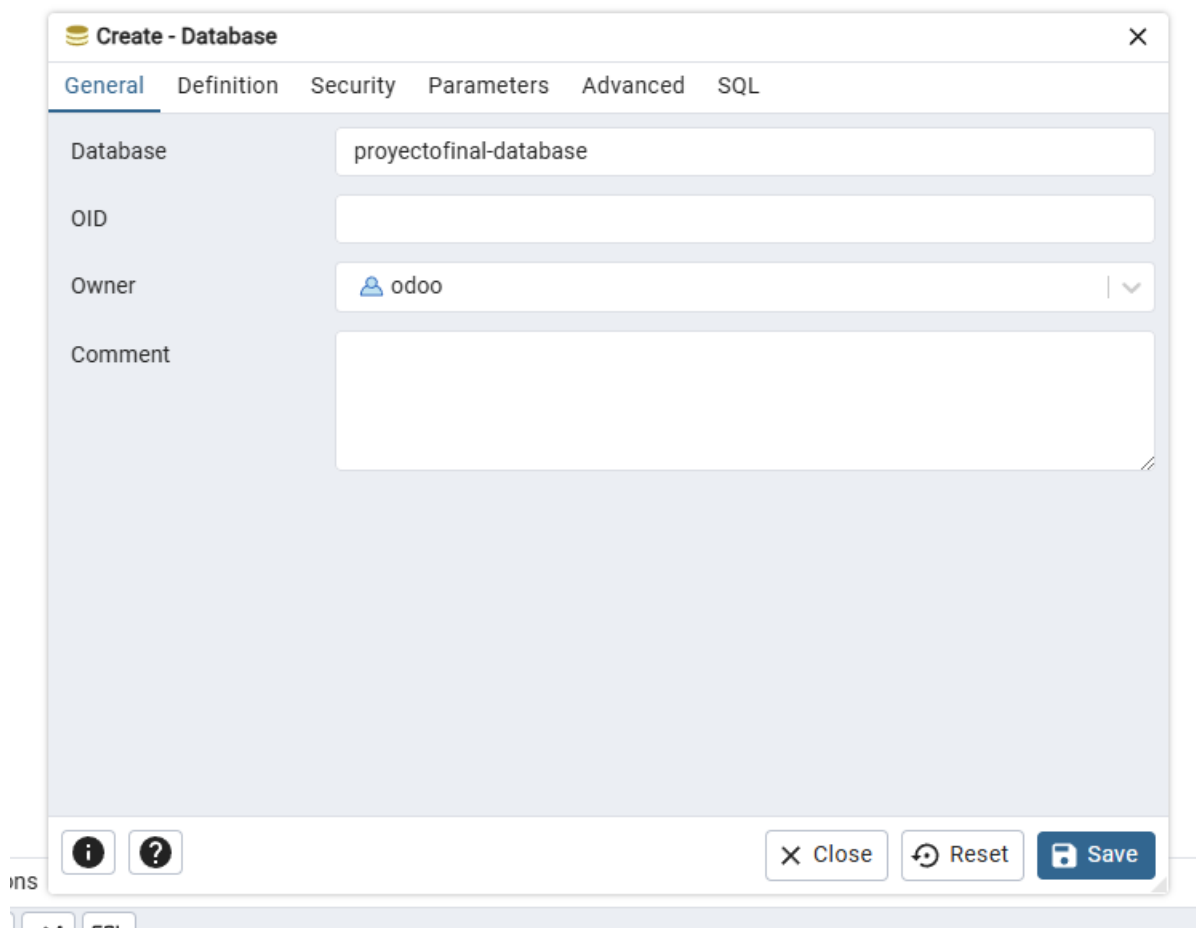
The screenshot shows the Swagger UI interface for the 'Product sale' section, displaying the response body for the DELETE endpoint. The response is a 200 status code with a success message: 'Eliminado producto-venta con éxito'.

## 5.2 Manual de instalación

Para instalar la API primero se debe crear la base de datos donde se guardará toda la información. Para ello hay que ir a pgadmin y en Databases hacer click derecho y pulsar Create > Database.



En el pop-up introducimos el nombre de la base de datos, guardamos y ya está lista para recibir la información.



A continuación nos dirigimos a Visual Studio Code y en la terminal ejecutamos los siguientes comandos.

- .\venv\Scripts\activate.ps1

```
PS C:\Users\rebeca.marleo\Documents\Sistemas de gestión empresarial\PythonSGE\FastAPI\proyecto final> .\venv\Scripts\activate.ps1
```

- `pip install -r .requirements.txt`

```
(venv) PS C:\Users\rebeca.marleo\Documents\Sistemas de gestión empresarial\PythonSGE\FastAPI\proyecto final> pip install -r .requirements.txt
Requirement already satisfied: fastapi in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from -r .requirements.txt (line 1)) (0.115.8)
Requirement already satisfied: uvicorn in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from -r .requirements.txt (line 2)) (0.34.0)
Requirement already satisfied: psycpg2 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from -r .requirements.txt (line 3)) (2.9.10)
Requirement already satisfied: SQLAlchemy in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from -r .requirements.txt (line 4)) (2.0.38)
Requirement already satisfied: starlette<0.46.0,>=0.40.0 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from fastapi->r .requirements.txt (line 1)) (0.45.3)
Requirement already satisfied: pydantic!=1.8,!1.8.1,!2.0.0,!2.0.1,!2.1.0,<3.0.0,>=1.7.4 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from fastapi->r .requirements.txt (line 1)) (2.10.6)
Requirement already satisfied: typing-extensions>=4.8.0 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from fastapi->r .requirements.txt (line 1)) (4.12.2)
Requirement already satisfied: click>=7.0 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from uvicorn->r .requirements.txt (line 2)) (8.1.8)
Requirement already satisfied: h11>=0.8 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from uvicorn->r .requirements.txt (line 2)) (0.14.0)
Requirement already satisfied: greenlet!=0.4.17 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from SQLAlchemy->r .requirements.txt (line 4)) (3.1.1)
Requirement already satisfied: colorama in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from click>=7.0->uvicorn->r .requirements.txt (line 2)) (0.4.6)
Requirement already satisfied: annotated-types>=0.6.0 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from pydantic!=1.8,!1.8.1,!2.0.0,!2.0.1,!2.1.0,<3.0.0,>=1.7.4->fastapi->r .requirements.txt (line 1)) (0.7.0)
Requirement already satisfied: pydantic-core==2.27.2 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from pydantic!=1.8,!1.8.1,!2.0.0,!2.0.1,!2.1.0,<3.0.0,>=1.7.4->fastapi->r .requirements.txt (line 1)) (2.27.2)
Requirement already satisfied: anyio<5,>=3.6.2 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from starlette<0.46.0,>=0.40.0->fastapi->r .requirements.txt (line 1)) (4.8.0)
Requirement already satisfied: idna>=2.8 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from anyio<5,>=3.6.2->starlette<0.46.0,>=0.40.0->fastapi->r .requirements.txt (line 1)) (3.10)
Requirement already satisfied: sniffio>=1.1 in c:\users\rebeca.marleo\documents\sistemas de gestión empresarial\pythonsg\fastapi\proyecto final\venv\lib\site-packages (from anyio<5,>=3.6.2->starlette<0.46.0,>=0.40.0->fastapi->r .requirements.txt (line 1)) (1.3.1)

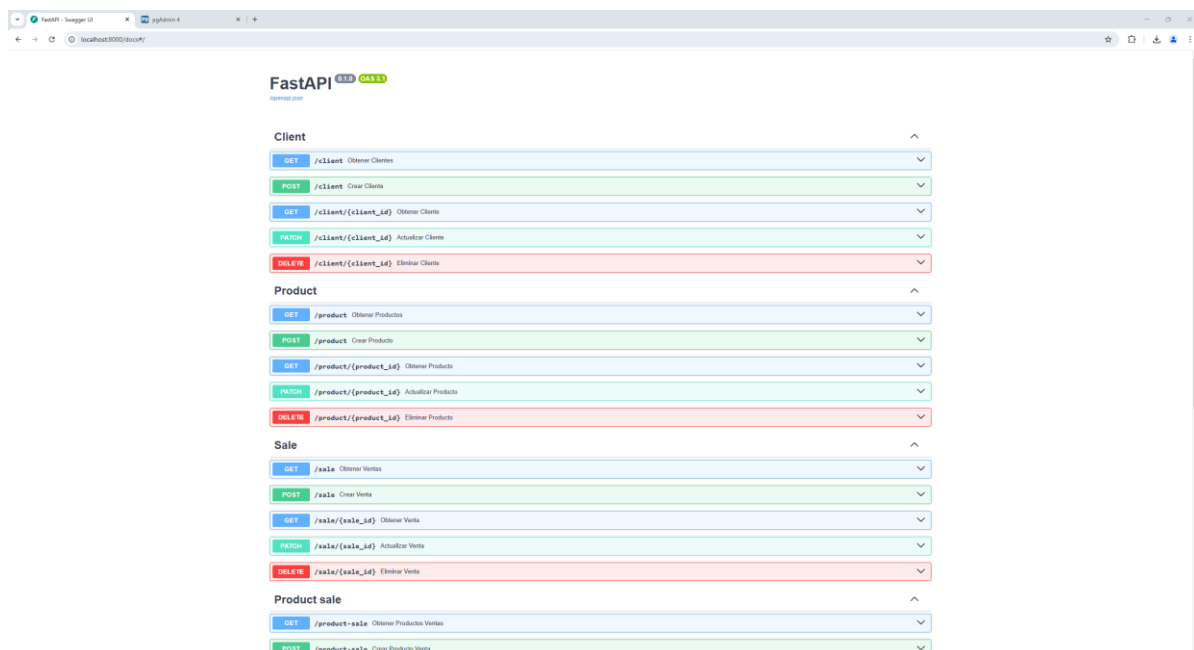
[notice] A new release of pip is available: 24.3.1 -> 25.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

Con esto ya tendríamos todos los recursos necesarios para lanzar el servidor de la API, lo cual se hace con el siguiente comando.

- `python .main.py`

```
(venv) PS C:\Users\rebeca.marleo\Documents\Sistemas de gestión empresarial\PythonSGE\FastAPI\proyecto final> python .main.py
INFO: Will watch for changes in these directories: ['C:\\Users\\rebeca.marleo\\Documents\\Sistemas de gestión empresarial\\PythonSGE\\FastAPI\\proyecto final']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [10068] using StatReload
INFO: Started server process [5876]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Accedemos a Swagger en la dirección localhost:8000/docs y la API ya estaría lista para usar.



## 6 Conclusiones y posibles ampliaciones

Uno de los principales retos enfrentados durante el desarrollo del proyecto fue la implementación de relaciones Many2many entre las tablas de la base de datos. Este tipo de relación es fundamental en sistemas donde un registro en una tabla puede estar vinculado con múltiples registros en otra tabla y viceversa. A pesar de su importancia, su correcta estructuración y gestión representó un desafío significativo debido a varios factores técnicos y conceptuales que debieron ser abordados para garantizar un funcionamiento óptimo de la base de datos y la API, como la creación de una tabla intermedia que gestionase correctamente la realización de ventas.

Una posible ampliación del proyecto sería la inclusión de diferentes tablas que proporcionarían información adicional. Estas tablas podrían ser las siguientes.

- Tabla de empleados: esta tabla asociaría cada venta a un empleado. De esta forma se podría saber que empleado ha realizado cada venta en caso de necesitar algún tipo de gestión.
- Tabla de proveedores: esta tabla contendría la información de los proveedores de los productos en venta. De esta forma se podría almacenar datos como su información de contacto o podría hacerse visible a los clientes para tener una referencia de la procedencia y calidad de los productos.
- Tablas de datos eliminados: estas tablas almacenarían los datos eliminados de sus correspondientes tablas. Por motivos legales, no se podrían conservar los datos de antiguos clientes o empleados pero sí podrían guardarse los datos de los productos o las ventas.

## 7 Bibliografía

<https://juanda.gitbooks.io/webapps/content/api/arquitectura-api-rest.html>

[https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/get-vs-post/?gad\\_source=1&gclid=CjwKCAiAwaG9BhAREiwAdhv6Y5-oZbHNyQDa4mlycGykWiocf\\_hDmTr5rFM1clUUqaHy-2yPzLHeHRoC2lgQAVD\\_BwE&gclid=aw.ds&itc=3VVVRR75-XIDLQ6-&utm\\_campaign=SGE-ES-DOM-DOMX-PMX-----&utm\\_content=DOM&utm\\_medium=cpc&utm\\_source=google](https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/get-vs-post/?gad_source=1&gclid=CjwKCAiAwaG9BhAREiwAdhv6Y5-oZbHNyQDa4mlycGykWiocf_hDmTr5rFM1clUUqaHy-2yPzLHeHRoC2lgQAVD_BwE&gclid=aw.ds&itc=3VVVRR75-XIDLQ6-&utm_campaign=SGE-ES-DOM-DOMX-PMX-----&utm_content=DOM&utm_medium=cpc&utm_source=google)

<https://estilow3b.com/metodos-http-post-get-put-delete/>

<https://decidesoluciones.es/arquitectura-de-microservicios/#:~:text=La%20arquitectura%20de%20microservicios%20es%20un%20m%C3%A9todo%20de%20desarrollo%20de,una%20funcionalidad%20de%20negocio%20completa.>

<https://www.ibm.com/es-es/topics/api#:~:text=Una%20API%2C%20o%20interfaz%20de,intercambiar%20datos%2C%20caracter%20ADsticas%20y%20funcionalidades.>

<https://www.docker.com/>

<https://fastapi.tiangolo.com/>

<https://www.uvicorn.org/>

<https://pypi.org/project/psycpg2/>

<https://www.sqlalchemy.org/>