IES RIBERA DE CASTILLA

PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones Multiplataforma

Informática y Comunicaciones

Aplicación CRUD con fastApi

Año: *2025*

Fecha de presentación: *11-02-2025*

Nombre y Apellidos: Pablo Villagrán González

Email: pablo.vilgon@educa.jcyl.es

ESPECIFICACIONES DEL PROYECTO:

ESTE PROYECTO REPRESENTA EL 30% DE LA CALIFICACIÓN DE LA SEGUNDA EVALUACIÓN; EL 70% RESTANTE CORRESPONDE AL EXAMEN. PARA CALCULAR LA NOTA DE LA 2ª EVALUACIÓN, ES IMPRESCINDIBLE OBTENER AL MENOS UN 5 EN CADA UNA DE LAS PARTES.

Título: **APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL**

Este proyecto será individual.

Diseñar una aplicación móvil que consuma servicios de una API desarrollada con **FastAPI**. La aplicación móvil enviará peticiones HTTP a las rutas de la API y recibirá respuestas JSON para interactuar con los datos del sistema. La API desarrollada con **FastAPI** será el backend y la aplicación móvil será el frontend.

Este proyecto tiene como objetivo integrar y aplicar conocimientos de backend, frontend y seguridad en el desarrollo de una solución completa y funcional.

Consta de 2 partes: desarrollo del código y memoria

DESARROLLO DEL CÓDIGO: PARTES DEL PROYECTO:

1. **Creación de la aplicación móvil:** • Implementar la aplicación móvil utilizando una tecnología determinada, como **Flutter**, **Kotlin**, **React Native (Android)**, o **Swift (iOS)**… • Puede estar basada en una aplicación desarrollada previamente o diseñar una nueva desde cero.
2. **Realización de peticiones HTTP:** • Configurar la aplicación móvil para realizar peticiones HTTP a las rutas de la API de **FastAPI**. • Investigar y utilizar las dependencias necesarias para el manejo de peticiones en la tecnología seleccionada.
3. Creación de la API en FastAPI: • Desarrollar una API que sirva como backend para la aplicación móvil.

- Documentar las rutas de la API utilizando herramientas nativas de FastAPI (como Swagger o Redoc).

4. **Modelo de datos:**

- Diseñar un esquema de base de datos que incluya al menos 3 tablas y relaciones necesarias para el CRUD.
- Especificar el Sistema Gestor de Base de Datos (SGBD) a utilizar: **SQLite**, **MySQL**, **PostgreSQL**, etc.
- Entregar un diagrama entidad-relación (ER) o similar para visualizar el modelo de datos antes de su implementación (tablas, campos de las tablas, relaciones…)

5. **Autenticación y seguridad:**

- Implementar autenticación basada en **JWT** para proteger las rutas sensibles de la API.
- Configurar las dependencias necesarias para gestionar la seguridad de los endpoints.
- Implementar un sistema de login donde la aplicación móvil envíe las credenciales de usuario a una ruta específica de la API.
- Si las credenciales son correctas, la API debe devolver un **token JWT**. La aplicación móvil debe almacenar este token de forma segura y utilizarlo para acceder a rutas protegidas.

6. Manejo de errores:

- Implementar un sistema para capturar y manejar errores en la parte de la API
- Proveer mensajes de error claros para respuestas HTTP (por ejemplo, códigos 400, 401, 403, 404, 500).

7. **Pruebas unitarias:**

- Desarrollar pruebas unitarias para garantizar la funcionalidad y la calidad del código.
- Incluir pruebas para validar el comportamiento de los endpoints en la API.

8. Uso de Pydantic:

- Utilizar **Pydantic** en la API para definir y validar modelos de datos.
- Asegurarse de que los modelos Pydantic manejen las validaciones necesarias (campos

requeridos, formatos, etc.).

MEMORIA DEL PROYECTO: APARTADOS

La memoria debe contener una portada (utilizar la que aparece en este documento), un índice y los siguientes apartados:

1. Introducción

Resumen del proyecto: explicar brevemente qué va a realizar la aplicación

Crearemos una API REST desarrollada con python y FastAPI que permite la gestión de tiendas y productos. La app permitirá a los usuarios registrarse, autenticarse y realizar operaciones como crear, eliminar y listar.

Los datos los almacenaremos en sqlite3, siendo seguro y eficiente.

2. Estado del arte

Definición de arquitectura de microservicios

La arquitectura de microservicios es un enfoque de desarrollo de software en el que una app se construye como un conjunto de servicios independientes, cada uno se comunica entre si a través de APIs.

En mi proyecto, la arquitectura que podríamos aplicar dividiendo la API podría ser:

- Servicio de autenticación: manejamos el registro y login de usuarios.
- Servicio de gestión de tiendas: administra tiendas y propietarios.
- Servicio de gestión de productos: controla los productos de cada tienda.

Definición de API

Una API (interfaz de programación de aplicaciones) es un conjunto de reglas y protocolos que permite la comunicación entre diferentes sistemas. En una API REST, los clientes pueden realizar solicitudes a un servidor para crear, leer, actualizar o eliminar datos.

Hay varios tipos de Apis:

- REST (Representational State Transfer): basada en HTTP, utiliza JSON como formato de datos.
- SOAP (Simple Object Acces Protocol): utiliza XML y es más complejo.
- GraphQL: permite consultas personalizadas en una única solicitud

En nuestro proyecto utilizaremos API REST, ya que es simple, ligera y ampliamente utilizada actualmente.

Estructura de una API: protocolo utilizado, métodos, partes de las URL de una API…

Utilizaremos el protocolo HTTP para la comunicación cliente-servidor.

Método	Descripción
GET	Obtener datos
POST	Crear nuevos recursos
PUT	Actualizar recursos existentes
DELETE	Eliminar recursos

Partes de las url, ejemplo:

https://api.proyectosge.com/tiendas/5/productos/

- Protocolo: https:// -> Indica que usa HTTP seguro.
- Dominio: api.proyectosge.com -> Dirección del servidor
- Recurso: /tiendas/ -> Endpoint para gestionar tiendas.
- ID: /5/ -> Identificador de una tienda específica.
- Sobrecurso: /productos/ -> Lista de productos de la tienda con ID 5.

Formas de crear una API en python: FastAPI y Flask, indicando cuál se va a utilizar y por qué

Flask:

- Es un microframework ligero para crear APIs en Python.
- Fácil de aprender y usar.
- No incluye validación automática de datos ni documentación automática.

FastAPI:

- Mas rapido que Flask gracias a Starlette y Pydantic.
- Soporta tipado estático, lo que mejora la validación de datos.
- Genera documentación automática son Swagger (OpenAPI).

Nosotros utilizaremos FastAPI, ya que es más rápido y eficiente que Flask. Genera documentación automática con Swagger. Y, por último, facilita la validación de datos y autenticación con JWT.

3. Descripción general del proyecto

3.1. Objetivos

Descripción de lo que se ha pretendido alcanzar con el proyecto

El desarrollo de mi proyecto tiene como finalidad crear una API REST eficiente y segura para la gestión de tiendas y productos, utilizando FastAPI y una base de datos SQLite3.

Los objetivos claves son:

- Implementar una API estructurada y escalable

Crea la API bien organizada, siguiendo una buena estructura, lo que facilitara su mantenimiento y escalabilidad en el futuro.

- Gestionar tiendas y productos de forma eficiente

Permitir a usuarios operaciones como crear, actualizar, eliminar y consultar tiendas y productos, asegurando una interacción sencilla y rápida con la base de datos.

- Control de acceso y seguridad

Hemos incorporado autenticación con JWT, asegurando que solo los usuarios autorizados puedan acceder a determinadas funcionalidades dentro del sistema.

- Uso de tecnologías modernas y eficientes

Elegimos FastAPI por su rapidez y eficiencia, junto con SQLite3 para el almacenamiento de datos, garantizando una solución liviana y fácil de implementar.

- Documentación automática con Swagger

Facilitar el uso de la API proporcionando documentación automática con swagger, lo que nos permite interactuar fácilmente con los endpoints.

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

Explicar todas las herramientas, tecnologías, lenguajes de programación… utilizados para desarrollar el proyecto: Docker, Visual Studio, python, FastAPI, gestor de base de datos…

- Lenguaje de programación: Python

Este ha sido el lenguaje elegido para nuestro desarrollo del proyecto debido a su simplicidad, versatilidad y amplia comunidad. Además, cuenta con numerosos frameworks y bibliotecas que facilitan el desarrollo de APIs.

- Framework Web: FastAPI

Es un framework moderno y eficiente para construir APIs en Python. Se basa en Starlette y Pydantic, lo que permite una validación de datos automática y una ejecución rápida.

Nos permite un alto rendimiento, tipado estático y soporte nativo para JWT.

- Base de Datos: SQLite3

Hemos optado por SQLite3, un sistema de base de datos ligero y fácil de manejar.

- ORM: SQLAlchemy

Para manejar la base de datos de manera más estructurada, hemos implementado SQLAlchemy, un ORM que facilita la interacción con SQLite3.

Nos permite trabajar con bases de datos sin necesidad de escribir SQL puro y nos facilita en un futuro la migración a otras bases de datos.

- Autenticación con JWT

Para el control de acceso y autenticación de usuarios, se ha utilizado JWT con la biblioteca PyJWT.

Ya que nos permite autentificación segura sin necesidad de sesiones en la base de datos, los tokens son compactos y seguros y tiene una fácil integración con FastAPI.

- Documentación con Swagger

Generamos la documentación de la API en Swagger Ui y ReDoc, lo que facilita su uso y pruebas.

Accesible en: http://127.0.0.1:8000/docs

- Editor de código: Visual Studio Code

El desarrollo del proyecto se ha utilizado en Visual Studio Code, un editor de código potente y ligero.
Las extensiones utilizadas fueron python y SQLite.

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

4.1. Análisis del sistema (funcionalidades básicas de la aplicación)

Explicar con detalle las operaciones que realiza la aplicación: aplicación móvil, API, autenticación, manejo de errores, pruebas unitarias

Permite gestionar usuarios, tiendas y productos. Lo hemos diseñado de forma eficiente y segura, integrando autenticación con JWT y una documentación automática con OpenAPI.

Principales Endpoints:

Método	Endpoint	Descripción
POST	/auth/register	Registro de usuarios
POST	/auth/login	Inicio de sesión y obtención del JWT
GET	/tiendas/	Listar todas las tiendas
POST	/tiendas/	Crear nueva tienda
GET	/tiendas/{id}/productos	Obtenemos productos de una tienda
POST	/productos/	Crear un producto
PUT	/productos/{id}	Modificar un producto
DELETE	/productos/{id}	Eliminar un producto

Para mejorar la experiencia del usuario, la Api implementa un sistema de manejo de errores con FastAPI. Tipo de errores:

- **400 Bad Request:** datos de entrada incorrectos.
- **401 Unauthorized**: token JWT invalido o ausente.
- **403 Not Found:** Usuario sin permisos.
- **404 Not Found:** Recurso no encontrado.
- **500 Internal Server Error**: error inesperado con el servidor.

4.2. Diseño de la base de datos

Explicar el diseño de la base de datos. Incluir un diagrama con las tablas y sus relaciones

El sistema se compone de las siguientes tablas principales:

- Usuarios: almacena los datos de los usuarios registrados.
- Tiendas: presentan los comercios.
- Productos: contiene los productos disponibles en las tiendas.

4.3. Implementación

Explicar la estructura del código, principales librerías utilizadas. Incluir explicaciones y capturas del código de todas las partes de la aplicación

Estructura del código:

- Database.py: configuramos la base de datos y creamos las tablas necesarias.
- Models.py: define los modelos de datos utilizando SQLAlchemy Y Pydantic.
- Crud.py: contiene las funciones para interactuar con la base de datos.
- App.py: define las rutas de la api.
- Main.py: punto de entrada de la aplicación, maneja la autenticación y centraliza la ejecución del servidor.

DATABASE

database.py > ...

```python
import sqlite3
from crud import crear_conexion

def crear_tablas():
    conn = crear_conexion()
    cursor = conn.cursor()

    # Crear tabla de usuarios
    cursor.execute('''CREATE TABLE IF NOT EXISTS usuarios (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE NOT NULL,
        password TEXT NOT NULL
    )''')

    # Crear las tablas de tiendas y productos
    cursor.execute('''CREATE TABLE IF NOT EXISTS tiendas (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        nombre TEXT UNIQUE NOT NULL,
        direccion TEXT NOT NULL
    )''')

    cursor.execute('''CREATE TABLE IF NOT EXISTS productos (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        nombre TEXT NOT NULL,
        precio REAL NOT NULL,
        tienda_id INTEGER,
        FOREIGN KEY (tienda_id) REFERENCES tiendas(id)
    )''')

    conn.commit()
    conn.close()

# Llamar a esta función para inicializar la base de datos
crear_tablas()

```

Definimos crear_conexion() para conectarnos a la base de datos SQLite.

Crear_tablas() crea tres tablas: usuarios, tiendas y productos.

MODELOS

```python
# Declaración de Base
Base = declarative_base()


# Esquema de entrada para registrar un usuario
class UsuarioCreate(BaseModel):
    username: str
    password: str

# Esquema de salida para responder con información del usuario
class UsuarioResponse(BaseModel):
    id: int
    username: str

    class Config:
        orm_mode = True


# Modelo de Tienda
class Tienda(Base):
    __tablename__ = "tiendas"

    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String, unique=True, index=True, nullable=False)
    direccion = Column(String, nullable=False)
    productos = relationship("Producto", back_populates="tienda")

# Modelo de Producto
class Producto(Base):
    __tablename__ = "productos"

    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String, nullable=False)
    precio = Column(Float, nullable=False)
    tienda_id = Column(Integer, ForeignKey("tiendas.id"))

    tienda = relationship("Tienda", back_populates="productos")

# Esquema de entrada para crear una tienda
class TiendaCreate(BaseModel):
    nombre: str
    direccion: str
```

```python
# Esquema de respuesta para tienda
class TiendaResponse(TiendaCreate):
    id: int

    class Config:
        orm_mode = True

# Esquema de entrada para crear un producto
class ProductoCreate(BaseModel):
    nombre: str
    precio: float
    tienda_id: int  # Relación con la tienda

# Esquema de respuesta para producto
class ProductoResponse(ProductoCreate):
    id: int

    class Config:
        orm_mode = True
```

Definimos usuarioCreate para recibir las credenciales y UsuarioResponse para devolver datos del usuario. Se crean los modelos Tienda y Producto, que tienen una relación uno a muchos.

CRUD

```python
class ProductoBase(BaseModel):
    nombre: str
    precio: float

class ProductoCreate(ProductoBase):
    tienda_id: int

class ProductoResponse(ProductoBase):
    id: int
    tienda_id: int

class TiendaBase(BaseModel):
    nombre: str
    direccion: str

class TiendaCreate(TiendaBase):
    pass

class TiendaResponse(TiendaBase):
    id: int
    productos: List[ProductoResponse] = []

# Conexión a la base de datos
def crear_conexion():
    conn = sqlite3.connect("tiendas.db")
    conn.execute("PRAGMA foreign_keys = ON")
    return conn

# CRUD de Tiendas
def crear_tienda(tienda: TiendaCreate):
    with crear_conexion() as conn:
        cursor = conn.cursor()
        cursor.execute("INSERT INTO tiendas (nombre, direccion) VALUES (?, ?)",
                       (tienda.nombre, tienda.direccion))
        conn.commit()
        tienda_id = cursor.lastrowid
        return TiendaResponse(id=tienda_id, nombre=tienda.nombre, direccion=tienda.direccion)

def obtener_tiendas():
    with crear_conexion() as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM tiendas")
        tiendas = cursor.fetchall()
        return [TiendaResponse(id=t[0], nombre=t[1], direccion=t[2]) for t in tiendas]

def obtener_tienda(tienda_id: int):
    with crear_conexion() as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM tiendas WHERE id = ?", (tienda_id,))
        tienda = cursor.fetchone()
        if tienda:
            cursor.execute("SELECT * FROM productos WHERE tienda_id = ?", (tienda_id,))
            productos = cursor.fetchall()
            productos_list = [ProductoResponse(id=p[0], nombre=p[1], precio=p[2], tienda_id=p[3]) for p in productos]
            return TiendaResponse(id=tienda[0], nombre=tienda[1], direccion=tienda[2], productos=productos_list)
        return None

def eliminar_tienda(tienda_id: int):
    with crear_conexion() as conn:
        cursor = conn.cursor()
        cursor.execute("DELETE FROM tiendas WHERE id = ?", (tienda_id,))
        conn.commit()
```

```python
# CRUD de Productos
def crear_producto(producto: ProductoCreate):
    with crear_conexion() as conn:
        cursor = conn.cursor()
        cursor.execute("INSERT INTO productos (nombre, precio, tienda_id) VALUES (?, ?, ?)",
                       (producto.nombre, producto.precio, producto.tienda_id))
        conn.commit()
        producto_id = cursor.lastrowid
        return ProductoResponse(id=producto_id, nombre=producto.nombre, precio=producto.precio, tienda_id=producto.tienda_id)

def obtener_productos():
    with crear_conexion() as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM productos")
        productos = cursor.fetchall()
        return [ProductoResponse(id=p[0], nombre=p[1], precio=p[2], tienda_id=p[3]) for p in productos]

def obtener_producto(producto_id: int):
    with crear_conexion() as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM productos WHERE id = ?", (producto_id,))
        producto = cursor.fetchone()
        if producto:
            return ProductoResponse(id=producto[0], nombre=producto[1], precio=producto[2], tienda_id=producto[3])
        return None

def eliminar_producto(producto_id: int):
    with crear_conexion() as conn:
        cursor = conn.cursor()
        cursor.execute("DELETE FROM productos WHERE id = ?", (producto_id,))
        conn.commit()
```

Definimos funciones para crear, eliminar y obtener tiendas.

Manejamos contraseñas cifradas con passlib.

Generamos un token para la autenticacion.

RUTAS

```python
class TiendaBase(BaseModel):
    nombre: str
    direccion: str

class TiendaResponse(TiendaBase):
    id: int

class ProductoBase(BaseModel):
    nombre: str
    precio: float
    tienda_id: int

class ProductoResponse(ProductoBase):
    id: int


# Rutas para Tiendas
@app.post("/tiendas/", response_model=TiendaResponse)
def api_crear_tienda(tienda: TiendaBase):
    tienda_id = crear_tienda(tienda.nombre, tienda.direccion)
    return {"id": tienda_id, "nombre": tienda.nombre, "direccion": tienda.direccion}

@app.get("/tiendas/", response_model=List[TiendaResponse])
def api_obtener_tiendas():
    tiendas = obtener_tiendas()
    return [{"id": tienda[0], "nombre": tienda[1], "direccion": tienda[2]} for tienda in tiendas]

@app.get("/tiendas/{tienda_id}", response_model=TiendaResponse)
def api_obtener_tienda(tienda_id: int):
    tienda = obtener_tienda(tienda_id)
    if not tienda:
        raise HTTPException(status_code=404, detail="Tienda no encontrada")
    return {"id": tienda[0], "nombre": tienda[1], "direccion": tienda[2]}


# Rutas para Productos
@app.post("/productos/", response_model=ProductoResponse)
def api_crear_producto(producto: ProductoBase):
    # Validar que la tienda existe
    tienda = obtener_tienda(producto.tienda_id)
    if not tienda:
        raise HTTPException(status_code=400, detail="La tienda especificada no existe")

    producto_id = crear_producto(producto.nombre, producto.precio, producto.tienda_id)
    return {"id": producto_id, "nombre": producto.nombre, "precio": producto.precio, "tienda_id": producto.tienda_id}
```

Se definen rutas para crear tiendas, obtener tiendas y lo mismo con productos.

AUTH

```python
from datetime import datetime, timedelta
from jose import JWTError, jwt
from passlib.context import CryptContext

SECRET_KEY = "tu_secreto"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def get_password_hash(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)

def create_access_token(data: dict, expires_delta: timedelta = None):
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta if expires_delta else timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

def verify_token(token: str):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload.get("sub")
    except JWTError:
        return None
```

Gestiona la seguridad de los usuarios mediante hashing de contraseñas y token JWT. Encripta las contraseñas, verifica estas mismas. Después genera tokens y los valida.

MAIN

```python
# Rutas de Tiendas
@app.post("/tiendas/", response_model=TiendaResponse)
def api_crear_tienda(tienda: TiendaCreate, db=Depends(get_db)):
    return crear_tienda(tienda)

@app.get("/tiendas/", response_model=List[TiendaResponse])
def api_obtener_tiendas(db=Depends(get_db)):
    return obtener_tiendas()

@app.get("/tiendas/{tienda_id}", response_model=TiendaResponse)
def api_obtener_tienda(tienda_id: int, db=Depends(get_db)):
    tienda = obtener_tienda(tienda_id)
    if not tienda:
        raise HTTPException(status_code=404, detail="Tienda no encontrada")
    return tienda

@app.delete("/tiendas/{tienda_id}")
def api_eliminar_tienda(tienda_id: int, db=Depends(get_db)):
    tienda = obtener_tienda(tienda_id)
    if not tienda:
        raise HTTPException(status_code=404, detail="Tienda no encontrada")

    eliminar_tienda(tienda_id)
    return {"message": "Tienda eliminada correctamente"}
```

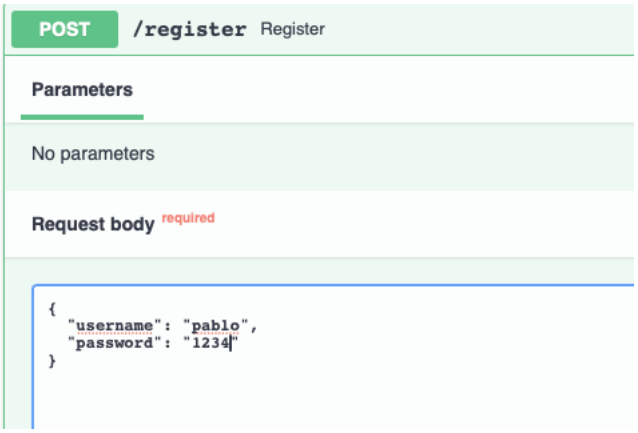
```python
# Login
@app.post("/login")
async def login(form_data: OAuth2PasswordRequestForm = Depends()):
    conn = crear_conexion()
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM usuarios WHERE username = ?", (form_data.username,))
    user = cursor.fetchone()

    if user is None or not verify_password(form_data.password, user[2]):
        raise HTTPException(status_code=401, detail="Credenciales incorrectas")

    access_token = create_access_token(data={"sub": form_data.username})
    conn.close()

    return {"access_token": access_token, "token_type": "bearer"}

@app.get("/protected")
async def protected_route(token: str = Depends(oauth2_scheme)):
    user = verify_token(token)
    if user is None:
        raise HTTPException(status_code=401, detail="Token no válido")

    return {"message": "Acceso autorizado", "user": user}
```

```python
# Rutas de Productos
@app.post("/productos/", response_model=ProductoResponse)
def api_crear_producto(producto: ProductoCreate, db=Depends(get_db)):
    tienda = obtener_tienda(producto.tienda_id)
    if not tienda:
        raise HTTPException(status_code=400, detail="La tienda especificada no existe")

    return crear_producto(producto)

@app.get("/productos/", response_model=List[ProductoResponse])
def api_obtener_productos(db=Depends(get_db)):
    return obtener_productos()

@app.delete("/productos/{producto_id}")
def api_eliminar_producto(producto_id: int, db=Depends(get_db)):
    producto = obtener_producto(producto_id)
    if not producto:
        raise HTTPException(status_code=404, detail="Producto no encontrado")

    eliminar_producto(producto_id)
    return {"message": "Producto eliminado correctamente"}

# Registro de usuario
@app.post("/register", response_model=UsuarioResponse)
async def register(usuario: UsuarioCreate):
    conn = crear_conexion()
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM usuarios WHERE username = ?", (usuario.username,))
    existing_user = cursor.fetchone()
    if existing_user:
        raise HTTPException(status_code=400, detail="Usuario ya existe")

    hashed_password = get_password_hash(usuario.password)
    cursor.execute("INSERT INTO usuarios (username, password) VALUES (?, ?)",
                   (usuario.username, hashed_password))

    conn.commit()
    usuario_id = cursor.lastrowid
    conn.close()

    return UsuarioResponse(id=usuario_id, username=usuario.username)
```

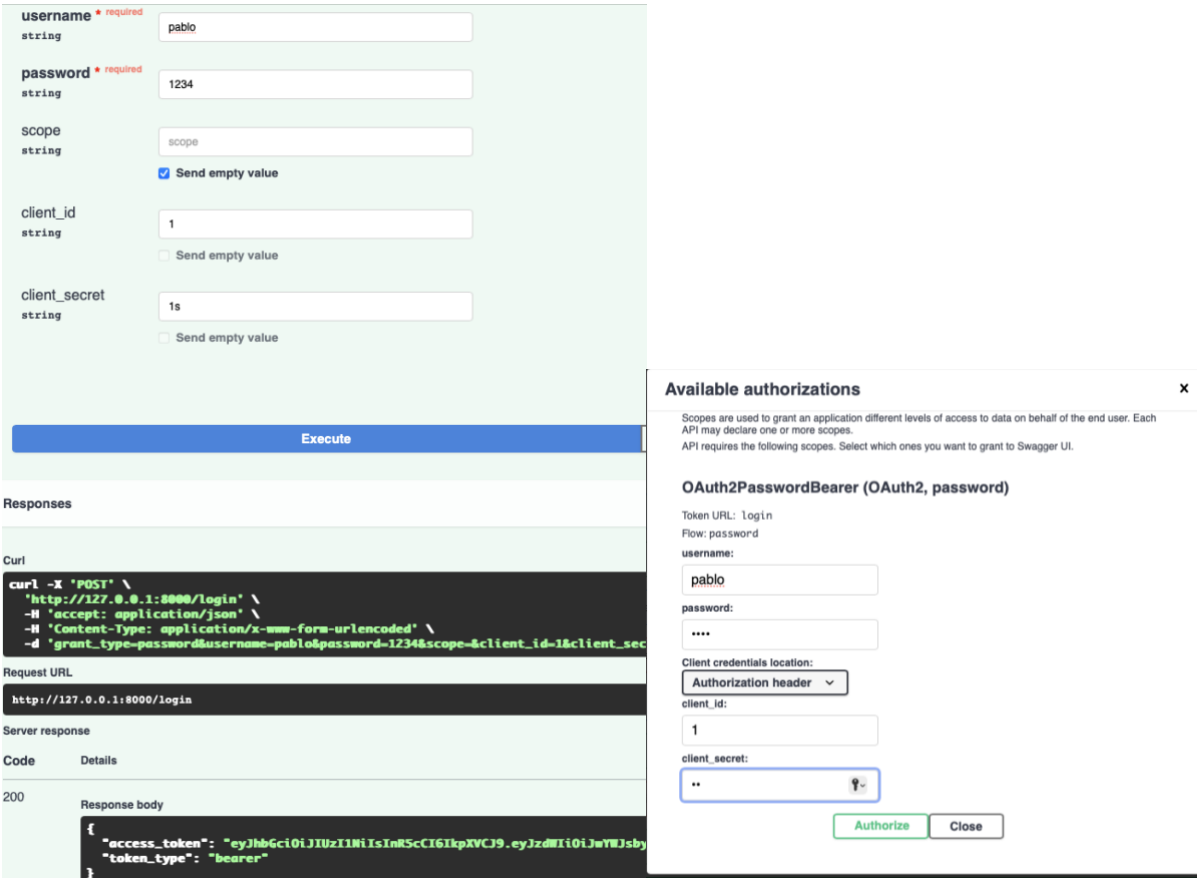
4.4. Pruebas

Realizar y documentar un mínimo de 2 pruebas: una para la API y otra para la integración entre la app móvil y la API

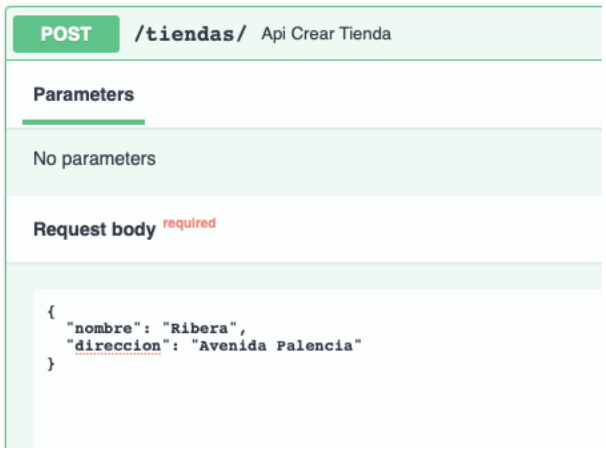
Primero de todo nos registramos.



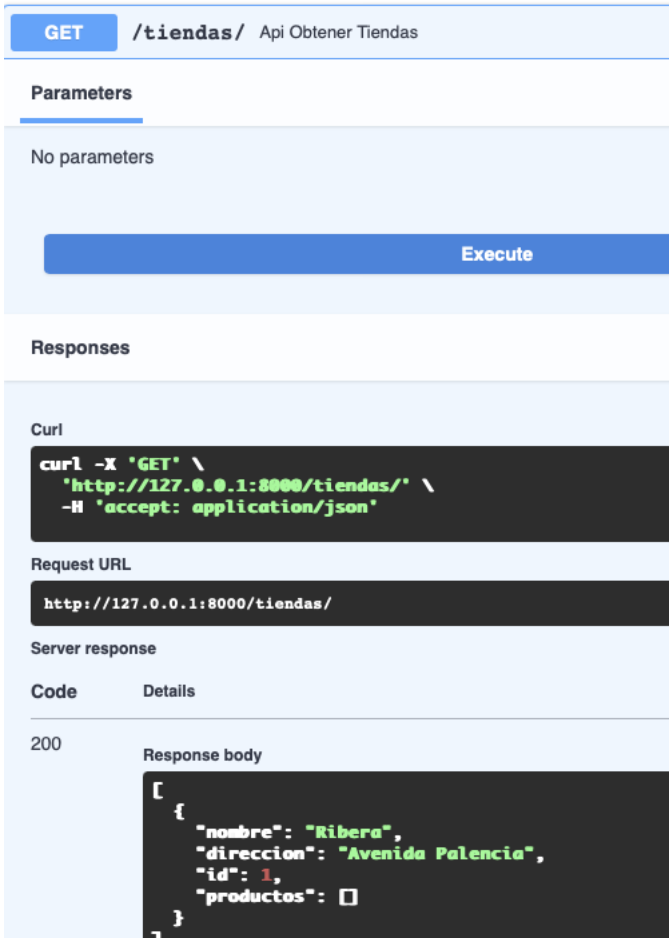
Despues, nos logeamos y nos pasa el token.



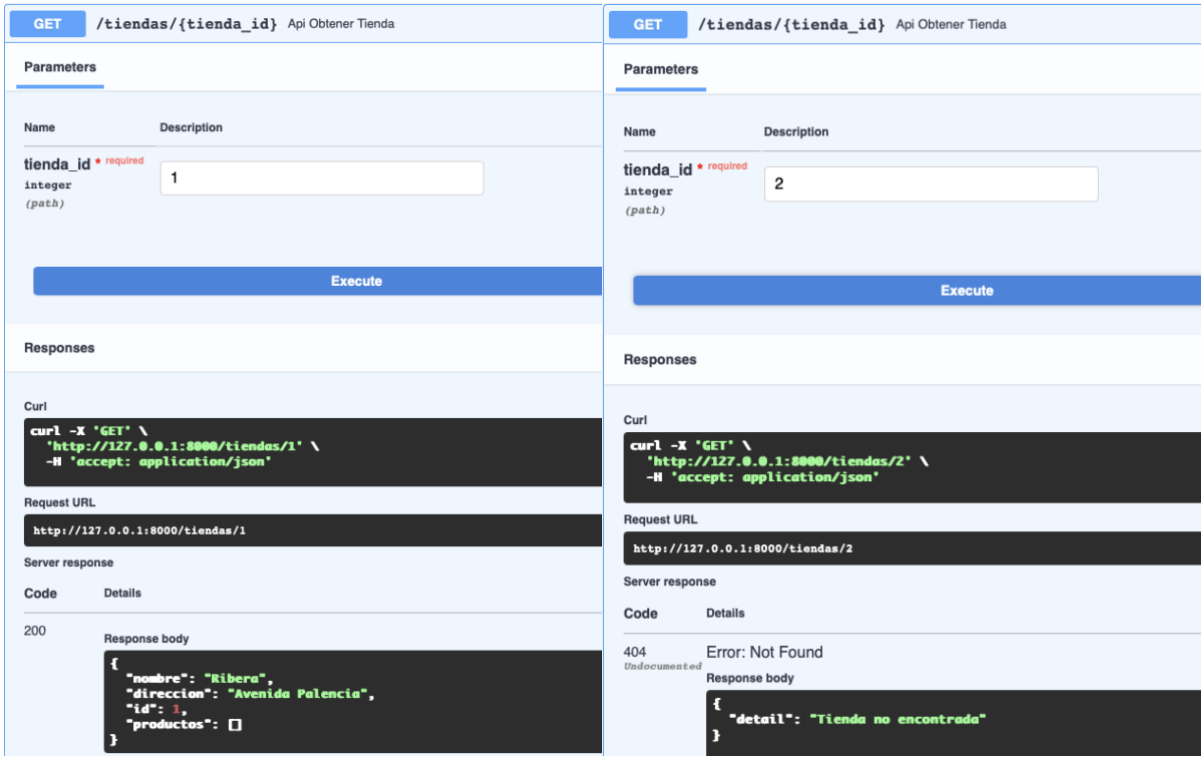
Creamos tienda:



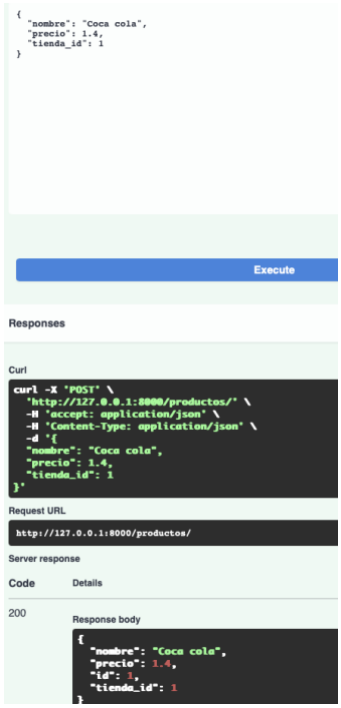
La obtenemos:



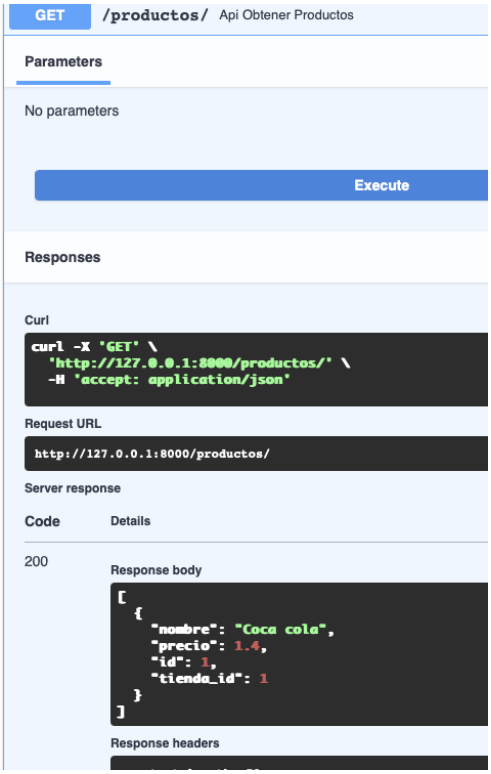
Obtener por id, si no tiene id muestra el error:



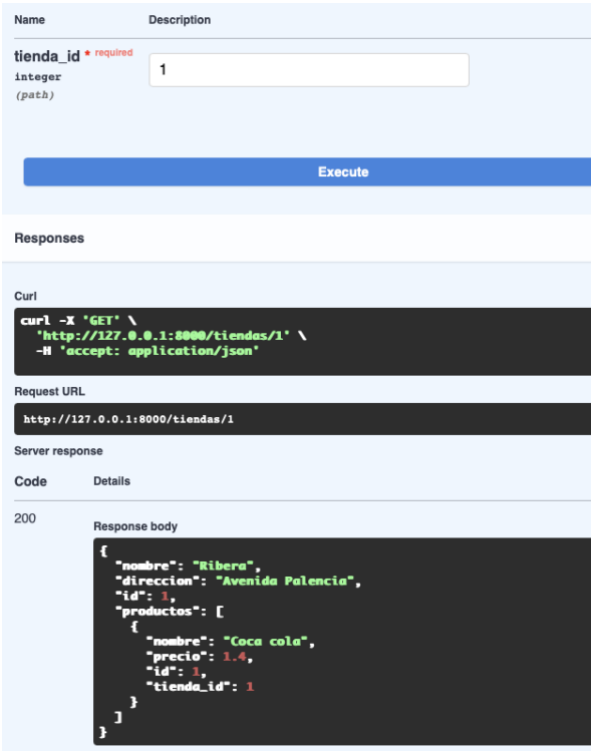
Crear producto:



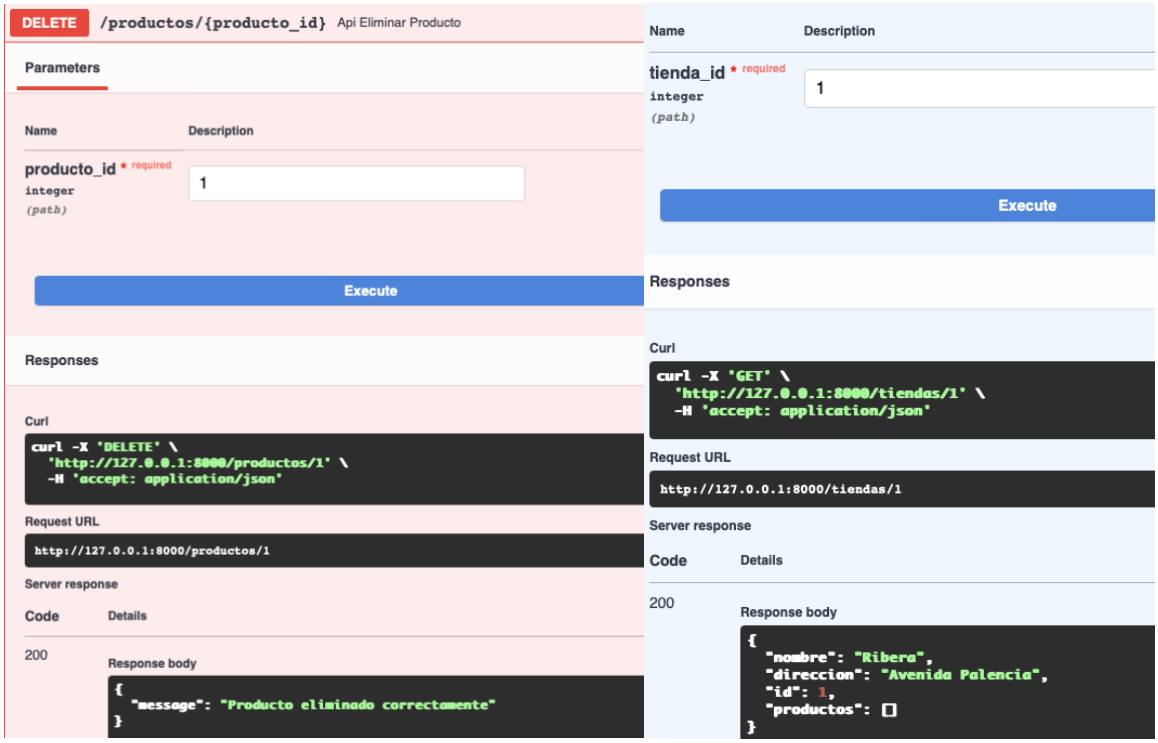
Obtener producto:



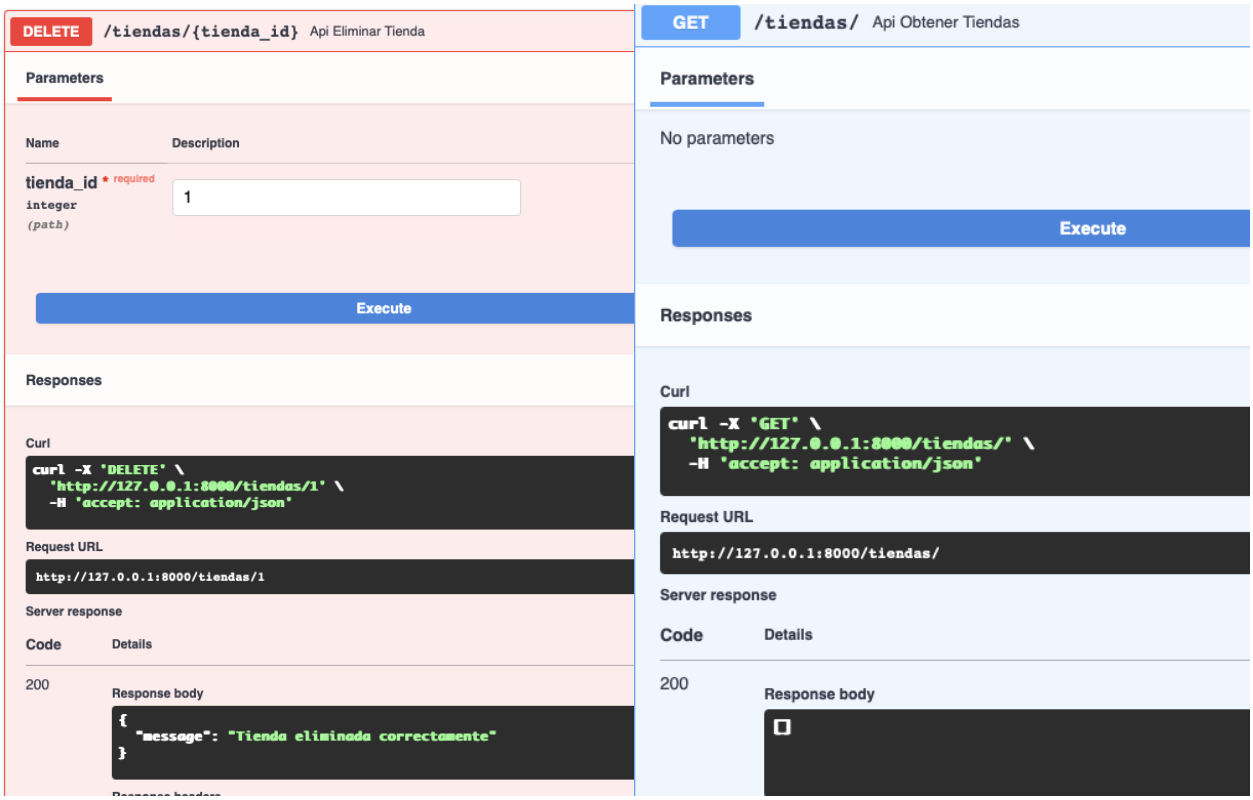
Obtener tienda y a la vez sus productos ya que existen:



Eliminar producto:



Eliminar tienda:



4.5. Despliegue de la aplicación

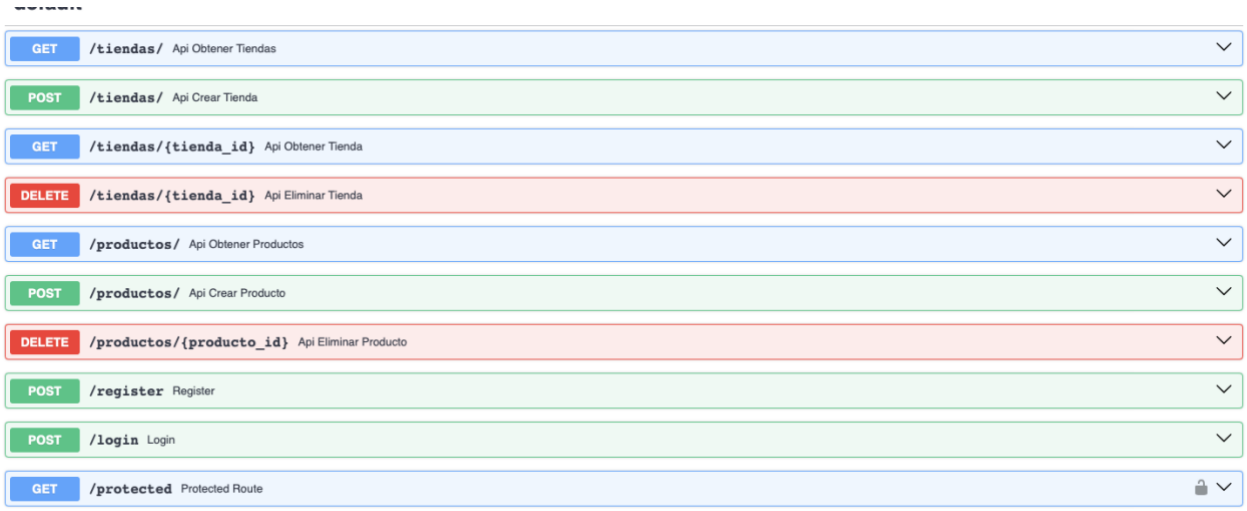
Nuestra Api solo estará disponible en local.

5. Manuales

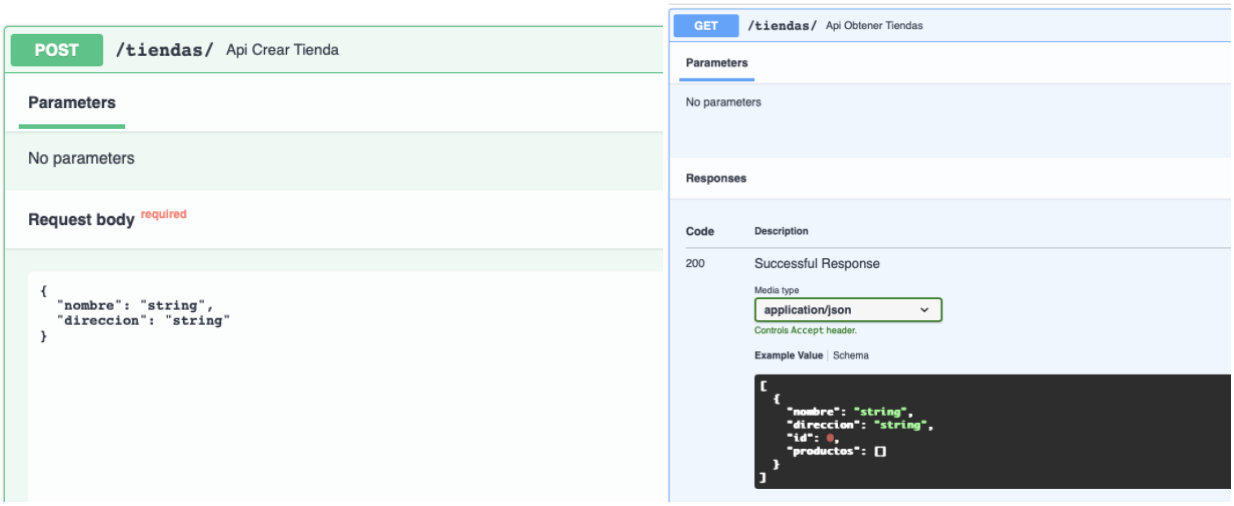
5.1. Manual de usuario:

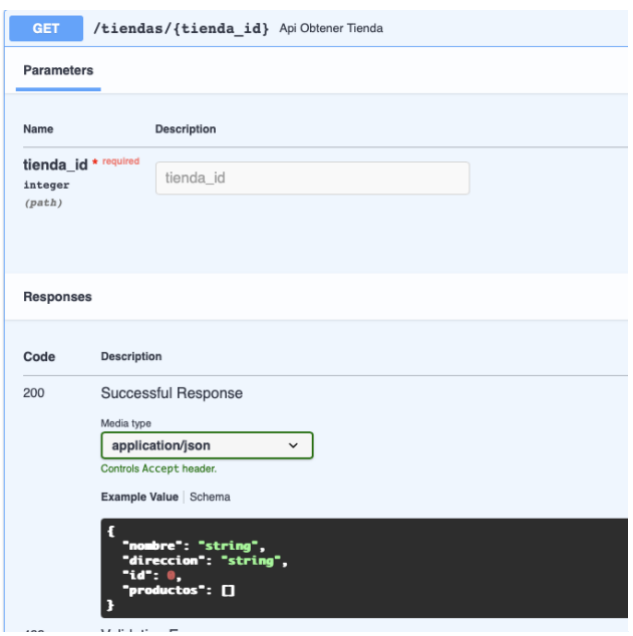
Interfaz de la aplicación: capturas de pantalla, breves explicaciones…

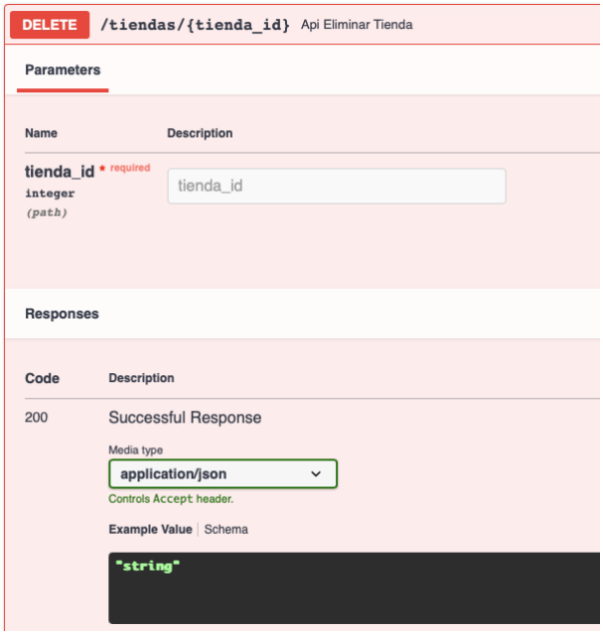
Nos encontramos de primeras con estas opciones en Swagger:



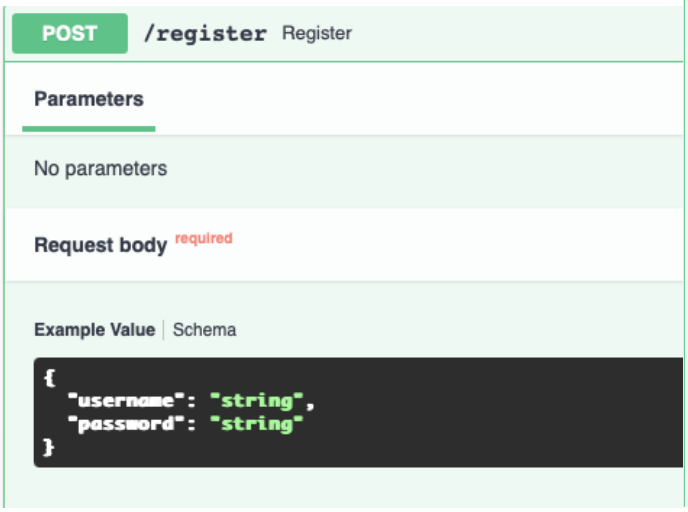
Despues podemos obtener todas las tiendas, por id, eliminarlas y añadir tiendas.

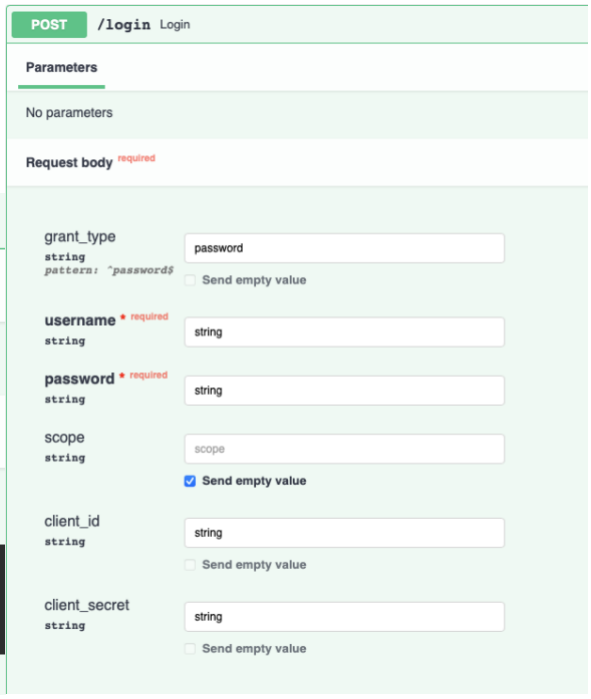






Podremos hacer lo mismo con producto. Pero, hay una opción de registro o login.





5.2. Manual de instalación:

Pasos que seguir para desplegar la aplicación: servidor local o remoto, otras formas de despliegue…

Tendremos que descargarnos el zip que está en GitHub, abrir el archivo con visual studio code, instalar por comandos algunas dependencias y ajustes y después arrancar la app.

Para ello habrá que usar este comando: uvicorn main:app –reload

6. Conclusiones y posibles ampliaciones

Dificultades encontradas en el desarrollo de la aplicación, grado de satisfacción en el trabajo realizado, aprendizaje…

Posibles ampliaciones: indicar al menos una

Inicialmente utilizamos SQLite, pero al aumentar la complejidad, surgieron problemas con concurrencia y escalabilidad. Por lo que sería mejor MySQL o PostgreSQL.

Mantener una estructura mejor dentro de mi proyecto en VisualStudio.

Nuestra satisfaccion va desde que la app cumple su función y permite gestionar todo con autenticación. Aprendimos sobre FastApi, SQLite y relacionarlo con Python.

Enviar correos cuando se registre una tienda o se agregue un nuevo producto seria mi ampliación.

7. Bibliografía

https://chatgpt.com/c/67ab7a77-7c30-800a-a68a-6c0d11eec6b7

https://anderfernandez.com/blog/como-crear-api-en-python/