IES RIBERA DE CASTILLA

PROYECTO SGE
2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones Multiplataforma

Informática y Comunicaciones

Gestión de vuelos

Año: *2025*

Fecha de presentación: *11/02/25*

Nombre y Apellidos: Alejandro de Gregorio Miguel

Email: Alejandro.gremig@educa.jcyl.es

1 Introducción

Esta API, desarrollada con FastAPI, tiene como objetivo la gestión eficiente de vuelos y compañías aéreas, proporcionando un conjunto de endpoints organizados según el nivel de autorización requerido.

Por un lado, cualquier usuario puede acceder a las funcionalidades relacionadas con los vuelos, incluyendo la creación de nuevos vuelos, su listado y la búsqueda filtrada por origen, destino y número de escalas.

Por otro lado, la gestión de las compañías aéreas requiere autenticación, permitiendo a los usuarios autorizados crear, modificar, listar y eliminar compañías.

Además, la API incluye un sistema de gestión de usuarios que permite la creación de cuentas y la generación de tokens de autenticación, asegurando que solo los usuarios con los permisos adecuados puedan administrar las compañías aéreas.

2 Estado del arte

2.1 Definición de arquitectura de microservicios

La arquitectura de microservicios es un modelo de desarrollo de software en el que una aplicación se descompone en varios servicios independientes, cada uno encargado de una función específica. Estos servicios operan de manera autónoma, permitiendo su desarrollo, despliegue y escalado individual, y se comunican entre sí a través de APIs ligeras o sistemas de mensajería.

Este enfoque proporciona flexibilidad, escalabilidad y resiliencia, ya que permite modificar o actualizar partes del sistema sin afectar al resto de la aplicación. Además, facilita la integración de diferentes tecnologías, optimizando el rendimiento y mantenimiento del sistema.

2.2 Definición de API

Una API (Interfaz de Programación de Aplicaciones) es un conjunto de reglas y definiciones que permite la comunicación entre distintos sistemas o aplicaciones. A través de ella, los clientes pueden interactuar con un servicio siguiendo protocolos, métodos y estructuras de datos previamente definidos.

Existen diversas categorías de APIs, entre las cuales destaca REST (Representational State Transfer), ampliamente utilizada en aplicaciones web por su simplicidad y eficiencia. También existen alternativas como GraphQL, que permite realizar consultas más dinámicas y flexibles, y SOAP, que sigue un enfoque más estructurado y basado en XML.

2.3 Estructura de una API

Las APIs cuentan con una estructura específica que facilita la interacción entre clientes y servidores:

- Protocolo: Generalmente se basan en HTTP o HTTPS para garantizar una comunicación segura y eficiente.
- Métodos HTTP:
    - GET: Obtiene información.
    - POST: Envía datos y crea nuevos recursos.
    - PUT: Modifica recursos existentes.
    - DELETE: Elimina un recurso determinado.
- Componentes de una URL en una API:
    - Base URL: Dirección principal del servicio
    - Endpoint: Ruta específica del recurso.
    - Parámetros y Query Strings: Información adicional en la URL.

- Formato de respuesta: Generalmente se utiliza JSON, aunque algunas APIs también admiten XML.
- Seguridad y autenticación: Se emplean métodos como JWT, OAuth2 o API Keys para garantizar que solo los usuarios autorizados accedan a determinados recursos.

2.4 Formas de crear una API en Python

Python cuenta con varios frameworks que facilitan la creación de APIs, siendo los más destacados Flask y FastAPI:

- Flask:
    - Es un framework ligero y flexible.
    - Posee una curva de aprendizaje sencilla.
    - Requiere bibliotecas adicionales para validaciones y documentación automática.
- FastAPI:
    - Ofrece un alto rendimiento gracias a ASGI y su compatibilidad con async/await.
    - Integra validaciones automáticas mediante Pydantic.
    - Genera documentación automática con Swagger y Redoc.

En este trabajo he optado por utilizar FastAPI debido a que lo hemos manejado en clase y además al haberme documentado de los posibles Frameworks que podíamos utilizar es el que más me ha convencido.

3 Descripción general del proyecto

3.1 Objetivos

El objetivo de este proyecto ha sido desarrollar una **API segura y eficiente** para gestionar vuelos y compañías aéreas, permitiendo su creación, consulta y administración de manera organizada. Se ha buscado una solución **escalable y bien documentada**, facilitando su integración con otras aplicaciones.

La API permite **crear, listar y buscar vuelos sin autenticación**, brindando acceso abierto a esta información. En cambio, la gestión de compañías aéreas requiere **autenticación mediante tokens (JWT)**, asegurando que solo usuarios autorizados puedan modificar estos datos.

Para garantizar un **buen rendimiento y seguridad**, se ha utilizado **FastAPI**, aprovechando su capacidad de procesamiento asíncrono, validación automática y documentación integrada. Como resultado, se obtiene una API clara y funcional, optimizando la administración de vuelos y compañías aéreas.

3.2 Entorno de trabajo

Para el desarrollo de este proyecto, he utilizado diversas tecnologías y herramientas con el objetivo de garantizar eficiencia, escalabilidad y facilidad de despliegue.

- Lenguaje de programación: Python
  He elegido Python por su simplicidad y por el manejo que tengo con este lenguaje gracias a lo aprendido en Sistemas de Gestión Empresarial
- Framework: FastAPI
  Opté por FastAPI después de documentarme y por el control que tengo con este framework.
- Entorno de desarrollo: PyCharm
  He trabajado en PyCharm, ya que proporciona herramientas avanzadas para la depuración, organización y gestión de dependencias del proyecto y porque lo prefiero al parecerse más a otros entornos de desarrollo que he utilizado.
- Gestor de base de datos: PostgreSQL
  Para el almacenamiento de datos, he utilizado PostgreSQL, un sistema de base

de datos relacional que garantiza seguridad, escalabilidad y alto rendimiento en entornos de producción.

- Contenedores: Docker
  He implementado Docker para la utilización de la base de datos, facilitando su despliegue y asegurando un entorno de ejecución consistente en diferentes sistemas.

Gracias a todo esto, he logrado desarrollar una API para la gestión de vuelos y compañías aéreas.

4 Documentación técnica

4.1 Análisis del sistema

Las funciones de mi API son las siguientes:

- Crear, listar y buscar vuelos en la base de datos según diferentes criterios (origen, destino y número de escalas).
- Registrar nuevas compañías aéreas, así como editar o eliminar las existentes.
- Consultar la lista de compañías aéreas almacenadas en la base de datos.
- Creación de un usuario necesario para la autenticación

Dado que la API es la encargada de gestionar toda esta información, también debe asumir funcionalidades adicionales:

- Gestionar la autenticación de los usuarios mediante tokens para restringir el acceso a las operaciones sobre compañías aéreas.

4.2 Diseño de la base de datos

Una vez definida la aplicación y sus funcionalidades, se establece la estructura de la base de datos.

La base de datos se compone de las siguientes tablas:

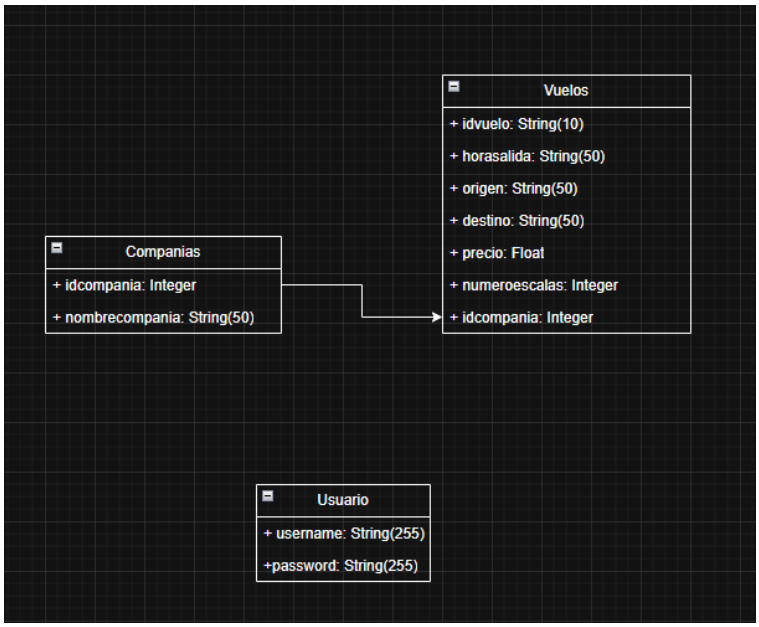
- Usuario (usuarios): Representa a los usuarios que interactúan con la API. Cada usuario se identifica mediante un nombre de usuario único

(username) y una contraseña (password), que se almacena de forma segura.

- Compañía (companias): Define las aerolíneas registradas en el sistema. Cada compañía cuenta con un identificador único (idcompania) y un nombre (nombrecompania).
- Vuelo (vuelos): Representa los vuelos gestionados por la API. Cada vuelo tiene un identificador único (idvuelo), la hora de salida (horasalida), el aeropuerto de origen (origen), el aeropuerto de destino (destino), el precio del billete (precio), y el número de escalas (numeroescalas). Además, cada vuelo está asociado a una compañía aérea específica mediante la clave foránea (idcompania).

Relaciones:

- Una compañía puede tener varios vuelos, mientras que cada vuelo pertenece únicamente a una compañía. Relación 1:N (One-to-Many).
- Los usuarios no tienen una relación directa con las demás tablas, ya que su función principal es la autenticación y control de acceso a determinadas operaciones de la API.



4.3 Implementación

auth: Contiene la lógica relacionada con la seguridad de la API, incluyendo la autenticación de usuarios y la generación de tokens JWT.

db: Gestiona la conexión con la base de datos (PostgreSQL en este caso) y define la configuración necesaria para interactuar con los datos.

routers: Agrupa los distintos endpoints de la API, organizándolos en módulos específicos para mejorar la legibilidad del código y facilitar la detección de errores.

services: Implementa la lógica de negocio de la API, separando las operaciones específicas de cada entidad, como vuelos, compañías y usuarios.

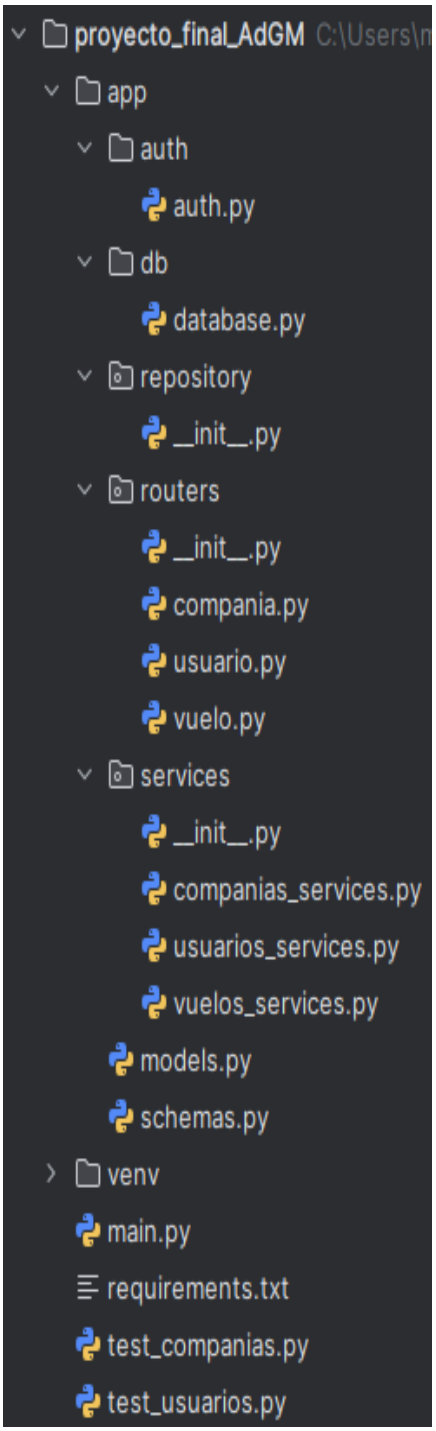
models.py: Define los modelos de datos utilizando SQLAlchemy, representando las tablas de la base de datos.

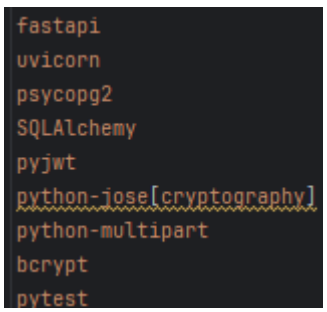
schemas.py: Contiene las validaciones y estructuras de datos utilizadas en las solicitudes y respuestas de la API.

main.py: Archivo principal que ejecuta la API y gestiona su configuración inicial.

requirements.txt: Lista las dependencias necesarias para ejecutar el proyecto.

test_companias.py / test_usuarios.py: Archivos de pruebas para verificar el correcto funcionamiento de la API.





4.3.1 Auth

```python
from app.models import Usuario
from app.db.database import get_db
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
import jwt
import os

router = APIRouter(prefix="/api", tags=["Token Control"])


oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/api/token")

SECRET_KEY = os.getenv("SECRET_KEY", "clave_super_secreta")
ALGORITHM = "HS256"

def create_token(data: dict):  1 usage
    """Genera un token JWT."""
    return jwt.encode(data, SECRET_KEY, algorithm=ALGORITHM)

@router.post("/token")
def login(form_data: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get_db)):
    """Autenticación de usuario y generación de token."""
    usuario = db.query(Usuario).filter(form_data.username == Usuario.username).first()

    if not usuario:
        raise HTTPException(status_code=401, detail="Usuario no encontrado")

    if not bcrypt.checkpw(form_data.password.encode(), usuario.password.encode()):
        raise HTTPException(status_code=401, detail="Contraseña incorrecta")

    token = create_token(data={"sub": usuario.username})
    return {
        "access_token": token,
        "token_type": "bearer"
    }

@router.get("/getToken")
def get_token(token: str = Depends(oauth2_scheme)):
    """Decodifica el token JWT y retorna la información del usuario."""
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if not username:
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Token inválido")
        return {"username": username}
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Token expirado")
    except jwt.PyJWTError:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Token inválido")
```

Este código define un módulo de autenticación en FastAPI que maneja la generación y validación de tokens JWT para el control de acceso de usuarios.

Permite a los usuarios autenticarse con su nombre y contraseña para recibir un token JWT, el cual luego se puede utilizar para acceder a recursos protegidos dentro de la API.

4.3.2 Database

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

SQLALCHEMY_DATABASE_URL = "postgresql://odoo:odoo@localhost:5342/fastapi_database"

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

def get_db():  19 usages
    db = SessionLocal() # Crear una nueva sesión
    try:
        yield db # Devuelve la sesión para su uso
    finally:
        db.close() # Cierra la sesión después de usarla
```

Este código configura la conexión a la base de datos utilizando SQLAlchemy en un proyecto con FastAPI.

Permite conectar la API con una base de datos PostgreSQL, administrar sesiones de manera eficiente y definir modelos usando SQLAlchemy.

4.3.3 Routers

4.3.3.1 Compania

```python
from fastapi import APIRouter, HTTPException, Depends
from typing import List
from sqlalchemy.orm import Session

from app.auth import auth
from app.schemas import CompaniaResponse, CompaniaCreate
from app.services.companias_services import CompaniasService
from app.db.database import get_db

router = APIRouter(prefix="/api/companias", tags=["Compañias"])

@router.get( path: "/", response_model=List[CompaniaResponse])
def get_all_companias(db: Session=Depends(get_db), token:str = Depends(auth.oauth2_scheme)):
    service=CompaniasService(db)
    try:
        return service.find_all()
    except Exception as e:
        raise HTTPException(status_code=400, detail="Error al obtener todas las compañias")

@router.post("/save")
def save_compania(compania: CompaniaCreate, db: Session = Depends(get_db), token:str = Depends(auth.oauth2_scheme)):
    service = CompaniasService(db)
    try:
        service.save_compania(compania)
        return "Compañia insertada correctamente"
    except Exception as e:
        raise HTTPException(status_code=400, detail="Error al guardar la nueva compañia")

@router.delete("/delete/{id}")
def delete_compania(id: int, db: Session = Depends(get_db), token:str = Depends(auth.oauth2_scheme)):
    service = CompaniasService(db)
    try:
        compania=service.find_by_id(id)
        if not compania:
            raise HTTPException(status_code=404, detail="No se encontró la compañia")
        else:
            service.delete_compania_by_id(id)
            return "Compañia eliminada correctamente"
    except Exception as e:
        raise HTTPException(status_code=400, detail="Error al eliminar la compañia por ID")

@router.put("/edit/{id}/{nombre}")
def edit_compania(id: int, nombre: str, db: Session = Depends(get_db), token:str = Depends(auth.oauth2_scheme)):
    service = CompaniasService(db)
    try:
        compania = service.find_by_id(id)
        if compania:
            raise HTTPException(status_code=404, detail="No se encontró la compañia")
        else:
            service.cambiar_nombre(id, nombre)
            return "Nombre de compañia editado correctamente"

    except Exception as e:
        raise HTTPException(status_code=400, detail="Error al editar la compañia")
```

Este código define un módulo en FastAPI para gestionar compañías dentro de una base de datos, permitiendo obtener, crear, eliminar y modificar compañías mediante endpoints protegidos con autenticación.

Permite **gestionar compañías** a través de una API protegida con OAuth2.
Los usuarios autenticados pueden **listar, crear, eliminar y editar compañías** en la base de datos.

4.3.3.2 Usuario

```python
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.models import Usuario
from app.schemas import UsuarioCreate
from app.services.usuarios_services import UsuariosService
from app.db.database import get_db

router = APIRouter(prefix="/api/usuarios", tags=["Usuarios"])

@router.post( path: "/usuarios/create", response_model=dict)
def create_usuario(usuario_data: UsuarioCreate, db: Session = Depends(get_db)):
    service = UsuariosService(db)


    usuario_existente = db.query(Usuario).filter(usuario_data.username == Usuario.username).first()
    if usuario_existente:
        raise HTTPException(status_code=400, detail="El usuario ya existe")

    try:
        service.save_usuario(usuario_data)
        return {"message": "Usuario creado exitosamente"}
    except Exception as e:
        raise HTTPException(status_code=400, detail=f"Error al crear usuario: {str(e)}")
```

Este código define un endpoint en FastAPI para la creación de usuarios en la base de datos.

Permite **crear nuevos usuarios** verificando previamente que el nombre de usuario **no esté repetido** en la base de datos.

4.3.3.3 Vuelo

```python
from fastapi import APIRouter, HTTPException, Depends
from typing import List

from app.models import Compania
from app.schemas import VueloResponse, VueloCreate
from sqlalchemy.orm import Session
from app.services.vuelos_services import VuelosService
from app.db.database import get_db
router = APIRouter(prefix="/api/vuelos", tags=["Vuelos"])

@router.get( path: "/", response_model=List[VueloResponse])
def get_all_vuelos(db: Session = Depends(get_db)):
    service=VuelosService(db)
    try:
        return service.find_all()
    except Exception:
        raise HTTPException(status_code=400, detail="Error al obtener todos los vuelos")

@router.post("/save")
def save_vuelo(vuelo_data: VueloCreate, db: Session = Depends(get_db)):
    service = VuelosService(db)
    compania = db.query(Compania).filter(Compania.idcompania == vuelo_data.idcompania).first()
    if not compania:
        raise HTTPException(status_code=400, detail="Compañía no encontrada")
    try:
        service.save_vuelo(vuelo_data)
        return "Vuelo guardado exitosamente"
    except Exception as e:
        raise HTTPException(status_code=400, detail=f"Error al guardar el vuelo{str(e)}")

@router.get( path: "/origenydestinovnumeroescalas/{origen}/{destino}/{numeroescalas}", response_model=List[VueloResponse])
def get_by_origen_destino_numeroescalas(origen: str, destino: str, numeroescalas: int, db: Session = Depends(get_db)):
    service = VuelosService(db)
    try:
        return service.find_vuelos_by_origen_destino_numeroescalas(origen, destino, numeroescalas)
    except Exception:
        raise HTTPException(status_code=400, detail="Error al obtener vuelos por los filtros citados")

@router.get( path: "/destino/{destino}", response_model=List[VueloResponse])
def get_by_destino(destino: str, db: Session = Depends(get_db)):
    service = VuelosService(db)
    try:
        return service.find_by_destino(destino)
    except Exception:
        raise HTTPException(status_code=400, detail="Error al obtener vuelos por destino")

@router.get( path: "/origen/{origen}", response_model=List[VueloResponse])
def get_by_origen(origen: str, db: Session = Depends(get_db)):
    service = VuelosService(db)
    try:
        return service.find_by_origen(origen)
    except Exception:
        raise HTTPException(status_code=400, detail="Error al obtener vuelos por origen")
```

Este código define un módulo en FastAPI para gestionar los vuelos en la API.

Permite: listar todos los vuelos, guardar un vuelo verificando la existencia de la compañía, filtrar vuelos por origen, destino y número de escalas, filtrar vuelos por origen o destino individualmente

4.3.4 Services

4.3.4.1 Companias

```python
from fastapi.params import Depends
from sqlalchemy.orm import Session
from typing import List, Optional, Tuple, Any, Type
from app.models import Compania
from app.schemas import CompaniaCreate, CompaniaResponse
from app.db.database import get_db
class CompaniasService:  7 usages
    def __init__(self, db: Session=Depends(get_db)):
        self.db = db

    def find_all(self) -> list[Type[Compania]]:  1 usage
        return self.db.query(Compania).order_by(Compania.idcompania).all()

    def find_by_id(self, id: int) -> Optional[CompaniaResponse]:  2 usages
        return self.db.query(Compania).filter(id == Compania.idcompania).first()

    def save_compania(self, compania_data: CompaniaCreate):  1 usage
        compania = Compania(nombrecompania=compania_data.nombrecompania)
        self.db.add(compania)
        self.db.commit()
        self.db.refresh(compania)
        return compania
    def delete_compania_by_id(self, id: int):  2 usages
        compania = self.db.query(Compania).filter(id == Compania.idcompania).first()
        if compania:
            self.db.delete(compania)
            self.db.commit()

    def cambiar_nombre(self, id: int, nombre_compania: str):  1 usage
        compania = self.db.query(Compania).filter(id == Compania.idcompania).first()
        if compania:
            compania.nombrecompania = nombre_compania
            self.db.commit()
            self.db.refresh(compania)
```

Este código maneja la lógica de negocio relacionada con las compañías en la API.

Este servicio encapsula la lógica de acceso a datos para la entidad Compania, facilitando su uso en los controladores.

4.3.4.2 Usuarios

```python
import bcrypt
from sqlalchemy.orm import Session
from app.models import Usuario
from app.schemas import UsuarioCreate

class UsuariosService:  4 usages
    def __init__(self, db: Session):
        self.db = db
    @staticmethod  1 usage
    def hash_password(password: str) -> str:
        return bcrypt.hashpw(password.encode(), bcrypt.gensalt()).decode()

    def save_usuario(self, usuario_data: UsuarioCreate):  2 usages
        # Verificar si el usuario ya existe
        usuario_existente = self.db.query(Usuario).filter(usuario_data.username == Usuario.username).first()
        if usuario_existente:
            raise ValueError("El usuario ya existe")

        # Crear nuevo usuario con contraseña hasheada
        nuevo_usuario = Usuario(
            username=usuario_data.username,
            password=self.hash_password(usuario_data.password)  # Hashear la contraseña
        )

        self.db.add(nuevo_usuario)
        self.db.commit()
        self.db.refresh(nuevo_usuario)

        return nuevo_usuario
```

Este código gestiona usuarios en la base de datos usando SQLAlchemy y bcrypt para el hash de contraseñas.

Este servicio registra usuarios en una API, garantizando que las contraseñas se almacenen de forma segura.

4.3.4.3 Vuelos

```python
from fastapi.params import Depends
from sqlalchemy.orm import Session
from typing import List, Type
from app.models import Vuelo, Compania
from app.db.database import get_db
from app.schemas import VueloCreate


class VuelosService:  6 usages
    def __init__(self, db: Session=Depends(get_db)):
        self.db = db

    def find_vuelos_by_origen_destino_numeroescalas(self, origen: str, destino: str, numeroescalas: int) -> list[Type[Vuelo]]:  1 usage
        return self.db.query(Vuelo).filter(
            *criterion: origen == Vuelo.origen,
            destino == Vuelo.destino,
            numeroescalas == Vuelo.numeroescalas
        ).all()

    def save_vuelo(self, vuelo_data: VueloCreate):  1 usage
        nuevo_vuelo = Vuelo(
            idvuelo=vuelo_data.idvuelo,
            horasalida=vuelo_data.horasalida,
            origen=vuelo_data.origen,
            destino=vuelo_data.destino,
            precio=vuelo_data.precio,
            numeroescalas=vuelo_data.numeroescalas,
            idcompania=vuelo_data.idcompania
        )
        self.db.add(nuevo_vuelo)
        self.db.commit()
        self.db.refresh(nuevo_vuelo)
    def find_by_destino(self, destino: str) -> list[Type[Vuelo]]:  1 usage
        return self.db.query(Vuelo).filter(destino == Vuelo.destino).all()

    def find_by_origen(self, origen: str) -> list[Type[Vuelo]]:  1 usage
        return self.db.query(Vuelo).filter(origen == Vuelo.origen).all()

    def find_all(self) -> list[Type[Vuelo]]:  1 usage
        return self.db.query(Vuelo).all()
```

Este código gestiona vuelos en la base de dato.

Este servicio encapsula la lógica de acceso a datos para la entidad Vuelo, facilitando su uso en los controladores.

4.3.5 Models.py

```python
from sqlalchemy import Column, Integer, String, Float, ForeignKey
from sqlalchemy.orm import relationship
from app.db.database import Base


class Compania(Base):  19 usages
    __tablename__ = "companias"
    idcompania = Column(Integer, primary_key=True, autoincrement=True)
    nombrecompania = Column(String(50), nullable=False)
    vuelos = relationship( argument: "Vuelo", backref="compania", cascade="all, delete-orphan")

class Vuelo(Base):  15 usages
    __tablename__ = "vuelos"
    idvuelo = Column(String(10), primary_key=True)
    horasalida = Column(String(50), nullable=False)
    origen = Column(String(50), nullable=False)
    destino = Column(String(50), nullable=False)
    precio = Column(Float, nullable=False)
    numeroescalas = Column(Integer, nullable=False)
    idcompania = Column(Integer, ForeignKey( column: "companias.idcompania", ondelete="CASCADE"))

class Usuario(Base):  11 usages
    __tablename__ = "usuarios"
    username = Column(String(255), primary_key=True)
    password = Column(String(255), nullable=False)
```

Este código define los modelos de base de datos.

4.3.6 Schemas.py

```python
from pydantic import BaseModel, Field
from typing import List


class VueloBase(BaseModel):  2 usages
    idvuelo: str = Field(..., max_length=10)
    horasalida: str = Field(..., max_length=50)
    origen: str = Field(..., max_length=50)
    destino: str = Field(..., max_length=50)
    precio: float
    numeroescalas: int
    idcompania: int


class VueloCreate(VueloBase):  4 usages
    pass


class VueloResponse(VueloBase):  6 usages
    model_config = {
        "from_attributes": True
    }

class CompaniaBase(BaseModel):  2 usages
    nombrecompania: str = Field(..., max_length=50)


class CompaniaCreate(CompaniaBase):  4 usages
    pass


class CompaniaResponse(CompaniaBase):  4 usages
    idcompania: int
    vuelos: List[VueloResponse]

    model_config = {
        "from_attributes":True
    }

class UsuarioBase(BaseModel):  2 usages
    username: str = Field(..., max_length=255)


class UsuarioCreate(UsuarioBase):  6 usages
    password: str = Field(..., max_length=255)


class UsuarioResponse(UsuarioBase):
    model_config = {
        "from_attributes": True
    }
```

Este código define esquemas Pydantic para validar y estructurar datos en la API.

4.4 Pruebas

4.4.1 Test creación de usuario

```python
import pytest
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from app.models import Base, Usuario
from app.schemas import UsuarioCreate
from app.services.usuarios_services import UsuariosService

# Configurar una base de datos en memoria para pruebas
SQLALCHEMY_DATABASE_URL = "sqlite:///:memory:"
engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False})
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)


@pytest.fixture(scope="function")  2 usages
def db_session():
    """
    Crea una sesión de base de datos en memoria para las pruebas.
    """
    Base.metadata.create_all(bind=engine)  # Crear las tablas en memoria
    db = TestingSessionLocal()
    yield db  # Proveer la sesión para el test
    db.close()
    Base.metadata.drop_all(bind=engine)  # Eliminar tablas después de la prueba


def test_save_usuario(db_session):
    """
    Prueba la creación de un usuario en UsuariosService.
    """
    service = UsuariosService(db_session)

    usuario_data = UsuarioCreate(username="test_user", password="securepassword")

    nuevo_usuario = service.save_usuario(usuario_data)

    # Verificar que el usuario se ha guardado correctamente
    assert nuevo_usuario.username == "test_user"
    assert nuevo_usuario.password is not None

    # Verificar que la contraseña está hasheada
    assert nuevo_usuario.password != "securepassword"  # No debe ser la misma que la original
```

Simplemente simula una base de datos y hace la inserción de un usuario.

4.4.2 Test de eliminación de compañía

```python
import pytest
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from app.models import Base, Compania
from app.services.companias_services import CompaniasService

# Configurar una base de datos en memoria para pruebas
SQLALCHEMY_DATABASE_URL = "sqlite:///:memory:"
engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False})
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

@pytest.fixture(scope="function")  6 usages
def db_session():
    """
    Crea una sesión de base de datos en memoria para las pruebas.
    """
    Base.metadata.create_all(bind=engine)  # Crear las tablas en memoria
    db = TestingSessionLocal()
    yield db  # Proveer la sesión para el test
    db.close()
    Base.metadata.drop_all(bind=engine)  # Eliminar tablas después de la prueba

def test_delete_compania_by_id(db_session):
    """
    Prueba la eliminación de una compañía .
    """
    service = CompaniasService(db_session)

    # Crear una compañía de prueba
    compania = Compania(idcompania=1, nombrecompania="Test Airlines")
    db_session.add(compania)
    db_session.commit()

    # Verificar que la compañía existe antes de eliminarla
    assert db_session.query(Compania).filter_by(idcompania=1).first() is not None

    # Eliminar la compañía
    service.delete_compania_by_id(1)

    # Verificar que la compañía ha sido eliminada
    assert db_session.query(Compania).filter_by(idcompania=1).first() is None
```

Simplemente simula una base de datos y crea una compañía para después eliminarla.

4.4.3 Resultados

4.5 *Despliegue de la aplicación*

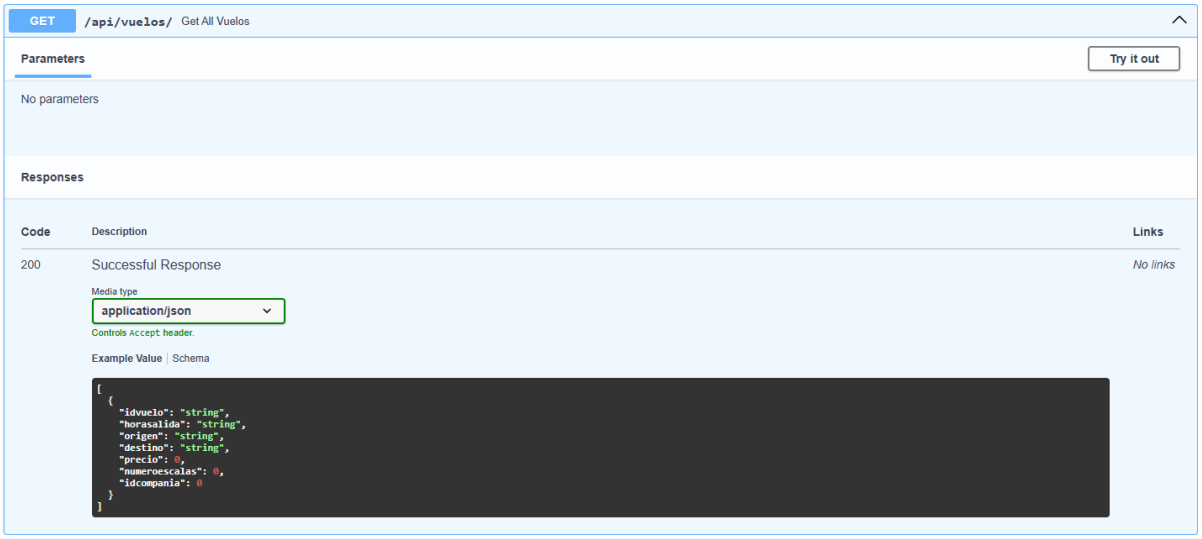
La API se encuentra desplegada en Swagger, la base de datos en pgAdmin junto con Docker.

5 Manuales

5.1 *Manual de usuario*

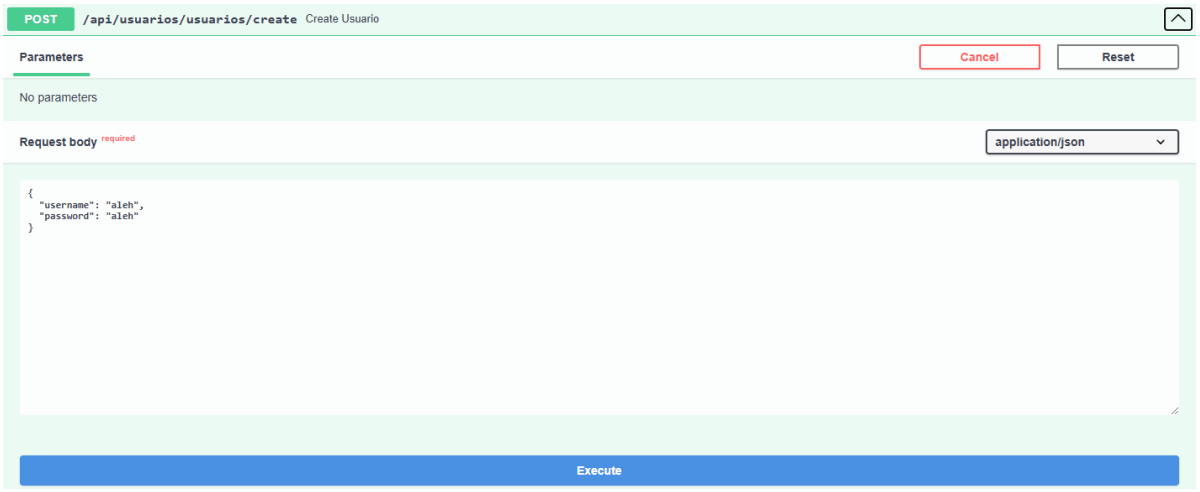
Antes de ejecutar el programa ejecutar Docker.

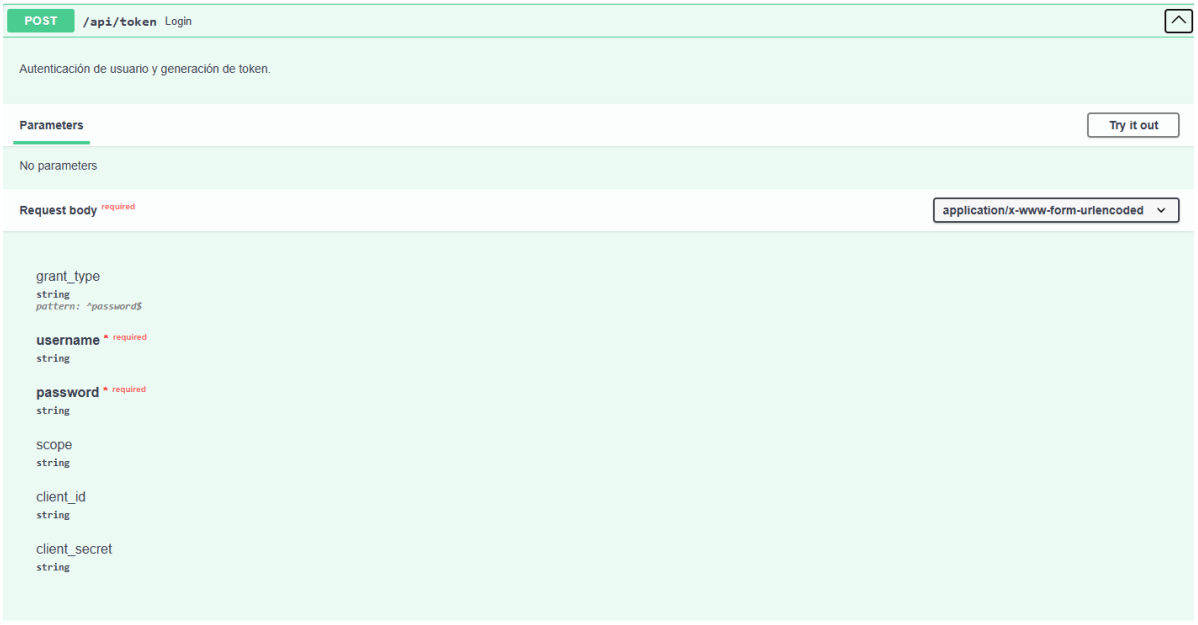
Dentro del programa ejecutar .\venv\Scripts\activate.ps1 para iniciar el entorno virtual y poder iniciar el código. Una vez iniciado poner en el navegador http://127.0.0.1:8000/docs y te llevará a Swagger. Dentro de este hay varios endpoints para gestionar la base de datos, algunos de estos protegidos por JWT.

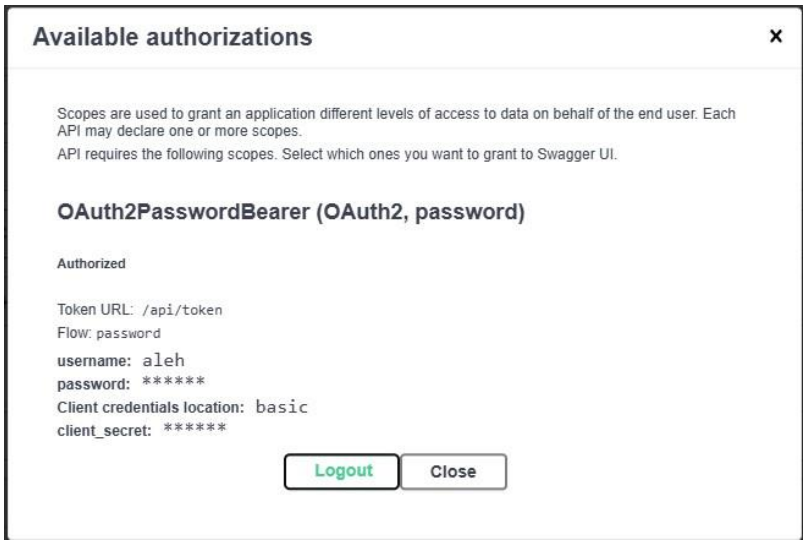


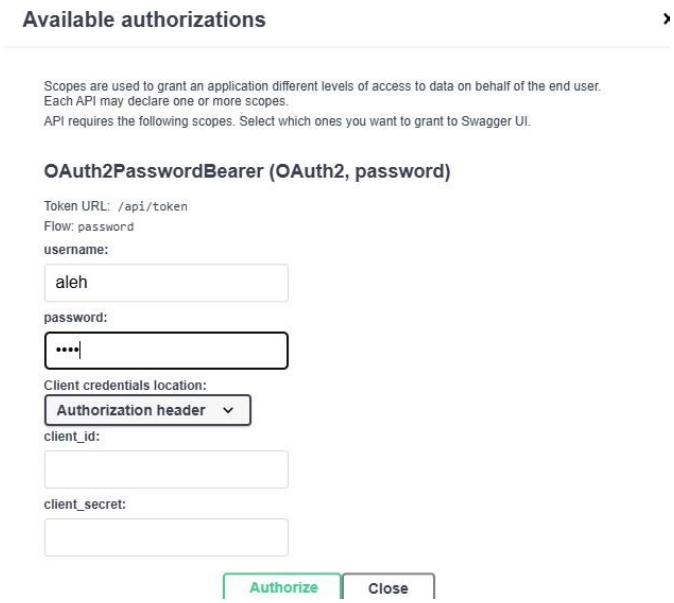
Para utilizarlo tienes que pulsar en Try it out y dependiendo del método tendrás que introducir datos o no. La parte inferior aparece el código de respuesta junto la información que devuelve.

Para la autenticación tendrás que crear un usuario con un nombre y una contraseña y después poner los datos en /api/token para generar un token y poder autenticarte en los diferentes endpoints que lo requieran.


POST /api/usuarios/usuarios/create Create Usuario
Parameters
Cancel
Reset
No parameters
Request body required
application/json
{
"username": "aleh",
"password": "aleh"
}
Execute


POST /api/token Login
Autenticación de usuario y generación de token.
Parameters
Try it out
No parameters
Request body required
application/x-www-form-urlencoded
grant_type
string
pattern: ^password$
username * required
string
password * required
string
scope
string
client_id
string
client_secret
string


Available authorizations
Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.
API requires the following scopes. Select which ones you want to grant to Swagger UI.
OAuth2PasswordBearer (OAuth2, password)
Authorized
Token URL: /api/token
Flow: password
username: aleh
password: ******
Client credentials location: basic
client_secret: ******
Logout
Close


Available authorizations
Scopes are used to grant an application different levels of access to data on behalf of the end user.
Each API may declare one or more scopes.
API requires the following scopes. Select which ones you want to grant to Swagger UI.
OAuth2PasswordBearer (OAuth2, password)
Token URL: /api/token
Flow: password
username:
aleh
password:
Client credentials location:
Authorization header
client_id:
client_secret:
Authorize
Close

5.2 Manual de instalación

Instalar Docker e iniciar sus contenedores-

Ir al navegador a http://127.0.0.1/ poner como usuario pgadmin4@pgadmin.org y como contraseña admin. Crear una nueva base de datos llamada fastapi_database.

En tu entorno de desarrollo debes poder ejecutar Python.

Dentro del proyect, en la consola, activar el entorno virtual e iniciar la API.

Ir al navegador y poner http://127.0.0.1:8000/docs y ya podrás manejar la API.

6 Conclusiones y posibles ampliaciones

En conclusión, me ha gustado realizar este trabajo ya que desarrollar una API en otro lenguaje de programación me ha supuesto un reto que he conseguido superar.

Como posibles ampliaciones me hubiera gustado añadir la aplicación móvil pero por mala gestión del tiempo no he podido realizarlo.

7 Bibliografía

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

https://decidesoluciones.es/arquitectura-de-microservicios/