



PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones
Multiplataforma
Informática y Comunicaciones

APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

Año: 2025

Fecha de presentación: 10/02/2025

Nombre y Apellidos: Eduardo Dominguez Toribio

Email: Eduardo.domtor@educa.jcyl.es

Índice

1	Introducción	3
2	Estado del arte.....	3
2.1	Definición de arquitectura de microservicios	3
2.2	Definición de API	3
2.3	Estructura de una API:.....	4
2.4	Formas de crear una API en python: FastAPI y Flask, indicando cuál se va a utilizar y por qué	6
3	Descripción general del proyecto	8
3.1	Objetivos	8
3.2	Entorno de trabajo (tecnologías de desarrollo y herramientas)	8
4	Documentación técnica: análisis, diseño, implementación, pruebas y despliegue	9
4.1	Análisis del sistema	9
4.2	Diseño de la base de datos.....	10
4.3	Implementación	12
4.3.1	“db”	12
4.3.2	“model”	13
4.3.3	“Routers”	15
4.3.4	“schemas”	16
4.3.5	“security”	17
4.3.6	“main.py”	18
4.3.7	“requeriments.txt”	19
4.4	Pruebas	19
4.4.1	Prueba API “CREAR HOTEL”	19
4.4.2	Prueba API “CREAR HABITACIÓN”	21
4.5	Despliegue de la aplicación	22
5	Manuales	22
5.1	Manual de usuario:	22
5.1.1	Autenticación	23
5.1.2	Gestión de usuarios	24
5.1.3	Gestión de hoteles.....	24
5.1.4	Gestión de habitaciones	25
5.2	Manual de instalación:	26
6	Conclusiones y posibles ampliaciones	27
7	Bibliografía.....	29

1 Introducción

He creado un proyecto enfocado en el desarrollo de una API para la gestión de un hotel. La API proporciona una serie de servicios que permiten la administración de reservas, gestión de habitaciones, control de usuarios y autenticación de clientes. Para ello he implementado un sistema de autenticación seguro basado en JWT (JSON Web Token), junto con una base de datos para almacenar la información necesaria para la aplicación.

El proyecto surge como una extensión de una aplicación previa desarrollada en Java para la gestión hotelera. He optado por desarrollar una API REST utilizando FastAPI, permitiendo la comunicación entre la aplicación móvil y el servidor.

Entre las funcionalidades principales que ofrece la API se encuentran:

- Registro y autenticación de usuarios.
- Gestión de habitaciones y disponibilidad.
- Creación, modificación y cancelación de reservas.
- Gestión de usuarios y roles dentro del sistema.
- Integración con bases de datos para el almacenamiento de información.

El objetivo principal de esta API es proporcionar una solución eficiente y flexible para la gestión hotelera, permitiendo una fácil integración con diferentes aplicaciones cliente, como aplicaciones móviles y sistemas web. Además, se busca garantizar la seguridad en las transacciones.

2 Estado del arte

2.1 Definición de arquitectura de microservicios

La arquitectura de microservicios es un estilo de diseño de software en el que una aplicación se divide en pequeños servicios independientes, cada uno con una función específica. Estos servicios se comunican entre sí a través de APIs y pueden ser desarrollados, desplegados y escalados de forma independiente.

Entre sus ventajas destacan la escalabilidad, facilidad de mantenimiento y resiliencia ante fallos, ya que un servicio puede fallar sin afectar a toda la aplicación, su implementación es más compleja que la de una arquitectura monolítica, ya que requiere gestión de comunicación entre servicios, despliegue y monitoreos más avanzados.

2.2 Definición de API

Una API (Application Programming Interface) es un conjunto de reglas y protocolos que permite la comunicación entre distintas aplicaciones. Define cómo interactúan diferentes componentes de software, permitiendo el intercambio de datos y la ejecución de funciones de manera estructurada y segura.

Las APIs pueden ser de diferentes tipos, entre ellas:

- REST (Representational State Transfer): Utiliza HTTP para la comunicación y se basa en operaciones CRUD (Create, Read, Update, Delete).
- SOAP (Simple Object Access Protocol): Más rígida que REST, utiliza XML para la comunicación.
- GraphQL: Proporciona flexibilidad en la consulta de datos al permitir que el cliente especifique exactamente la información que necesita.

En este proyecto, se ha optado por una API RESTful debido a su simplicidad y amplia adaptación en el desarrollo web.

2.3 Estructura de una API:

Una API REST sigue un conjunto de principios y convenciones que permiten la comunicación entre sistemas de manera eficiente y escalable. En este apartado, se explican los elementos clave que conforman su estructura.

Protocolo utilizado

Las APIs RESTful utilizan HTTP o HTTPS (versión segura con cifrado SSL/TLS) como protocolo de comunicación. HTTPS es la opción recomendada, ya que protege la integridad y privacidad de los datos transmitidos.

Métodos HTTP (Operaciones CRUD)

En una API REST, cada recurso puede ser manipulado a través de métodos HTTP específicos. Los más comunes son:

<u>METODO HTTP</u>	<u>DESCRIPCIÓN</u>	<u>EJEMPLO DE USO</u>
GET	Obtiene datos de un recurso.	GET /usuarios (Obtener lista de usuarios)
POST	Crea un nuevo recurso.	POST /usuarios (Registrar un nuevo usuario)
PUT	Actualiza un recurso existente.	PUT /usuarios/1 (Modificar el usuario con ID 1)
PATCH	Modifica parcialmente un recurso.	PATCH /usuarios/1 (Actualizar solo un campo del usuario con ID 1)

DELETE	Elimina un recurso.	DELETE /usuarios/1 (Eliminar el usuario con ID 1)
--------	---------------------	---

Estructura de una URL en una API REST

Las URLs en una API RESTful deben ser claras, predecibles y seguir buenas prácticas. Su estructura se compone de varios elementos:

1. Base URL (Punto de acceso a la API)

Es la dirección principal donde se aloja la API. Ejemplo:

`https://api.hotelmanager.com/`

2. Recursos (Nombres en plural)

Los recursos representan las entidades del sistema y se nombran en plural para mayor claridad:

`https://api.hotelmanager.com/usuarios`

`https://api.hotelmanager.com/reservas`

`https://api.hotelmanager.com/habitaciones`

3. Identificadores de recursos (IDs en la URL)

Para acceder a un recurso específico, se usa su identificador único (ID):

GET `https://api.hotelmanager.com/usuarios/15` → Obtiene los datos del usuario con ID 15.

PUT `https://api.hotelmanager.com/usuarios/15` → Modifica el usuario con ID 15.

DELETE `https://api.hotelmanager.com/usuarios/15` → Elimina el usuario con ID 15.

4. Códigos de estado HTTP

Las APIs RESTful devuelven códigos de estado HTTP para indicar el resultado de una solicitud.

Algunos de los más comunes son:

200 OK - La solicitud fue exitosa.

201 Created - Un nuevo recurso fue creado.

204 No Content - La solicitud se procesó correctamente, pero no hay contenido en la respuesta.

400 Bad Request - La solicitud tiene errores de formato o datos inválidos.

401 Unauthorized - Falta autenticación para acceder al recurso.

403 Forbidden - El usuario no tiene permisos para acceder al recurso.

404 Not Found - El recurso solicitado no existe.

500 Internal Server Error - Error inesperado en el servidor.

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

Para el desarrollo de esta API, he utilizado FastAPI, un framework moderno que destaca por su alto rendimiento, validación automática de datos y generación de documentación integrada. Sin embargo, antes de justificar esta elección, compararé FastAPI con otra opción popular en Python: Flask.

Python ofrece múltiples frameworks para la creación de APIs, entre los cuales destacan Flask y FastAPI, dos de las opciones más populares debido a su facilidad de uso y rendimiento.

Flask

Flask es un microframework minimalista que permite desarrollar aplicaciones web de manera flexible. Fue diseñado para ser ligero y modular, lo que lo hace una excelente opción para proyectos pequeños y medianos.

Ventajas de Flask:

- Sencillez y flexibilidad en su uso.
- Gran comunidad y amplia documentación.
- Compatible con múltiples extensiones como Flask-SQLAlchemy para bases de datos.

Desventajas de Flask:

- No incluye validación automática de datos.
- Menor rendimiento en comparación con FastAPI debido a su naturaleza síncrona.
- No ofrece soporte nativo para documentación automática de la API.

Ejemplo de una API simple con Flask:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/hello", methods=["GET"])

def hello():

    return jsonify({"message": "Hola desde Flask"})
```

```
if __name__ == "__main__":
```

```
    app.run(debug=True)
```

FastAPI

FastAPI es un framework moderno basado en Python que aprovecha tipado estático y `async/await` para mejorar la velocidad y la eficiencia en la creación de APIs.

Ventajas de FastAPI:

- Alto rendimiento gracias a su compatibilidad con **`async/await`**.
- Generación automática de documentación con **Swagger UI** y **Redoc**.
- Validación de datos integrada mediante **Pydantic**.
- Soporte nativo para **tipado estático** de Python.

Desventajas de FastAPI:

- Comunidad más pequeña en comparación con Flask.
- Puede ser más complejo para quienes no están familiarizados con tipado estático y programación asíncrona.

Ejemplo de una API simple con FastAPI:

```
from fastapi import FastAPI
```

```
app = FastAPI()
```

```
@app.get("/hello")
```

```
def hello():
```

```
    return {"message": "Hola desde FastAPI"}
```

Justificación de la elección de FastAPI

En este proyecto he optado por **FastAPI** porque proporciona un rendimiento superior y simplifica el desarrollo al manejar validaciones de datos y documentación de forma automática. Además, su compatibilidad con asynchronous programming lo hace más eficiente para manejar múltiples solicitudes concurrentes, lo cual es ideal para aplicaciones escalables.

3 Descripción general del proyecto

3.1 Objetivos

El objetivo principal de esta API es desarrollar una aplicación móvil que interactúe con un backend en FastAPI para realizar operaciones CRUD de manera segura y eficiente. Además, se busca:

- Implementar autenticación segura mediante JWT (JSON Web Token).
- Diseñar una base de datos estructurada con un mínimo de tres tablas relacionadas.
- Aplicar pruebas unitarias para garantizar la calidad del código.
- Proporcionar una solución eficiente y flexible para la gestión hotelera.
- Permitir una fácil integración con diferentes aplicaciones cliente, como aplicaciones móviles y sistemas web.

3.2 Entorno de trabajo (tecnologías de desarrollo y herramientas)

Para el desarrollo del proyecto, he utilizado las siguientes herramientas, tecnologías y lenguajes de programación:

- **Docker:** Docker se ha utilizado para la creación y gestión de contenedores, permitiendo la consistencia y portabilidad del entorno de desarrollo y producción.
- **Visual Studio Code:** Como entorno de desarrollo integrado (IDE) principal, Visual Studio Code ha proporcionado una plataforma eficiente y personalizable para escribir y depurar el código.
- **Python:** Lenguaje de programación principal utilizado para el desarrollo de la API, debido a su simplicidad y versatilidad.
- **FastAPI:** Framework elegido para la creación de la API debido a su alto rendimiento, validación automática de datos y generación de documentación integrada.
- **PostgreSQL:** Gestor de base de datos utilizado para almacenar toda la información relevante de la aplicación, como usuarios, reservas y habitaciones.
- **JWT (JSON Web Token):** Sistema de autenticación seguro utilizado para proteger las transacciones y garantizar que solo los usuarios autorizados puedan acceder a los recursos.
- **GitHub:** Herramienta de control de versiones utilizada para gestionar el código fuente.

Además, he empleado otras herramientas auxiliares y librerías como:

- **Uvicorn:** Servidor ASGI ligero y rápido, utilizado para ejecutar la aplicación FastAPI.
- **Pydantic:** Librería utilizada para la validación y serialización de datos en FastAPI.
- **SQLAlchemy:** Herramienta de migración de bases de datos, para manejar cambios en la estructura de la base de datos.

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

4.1 *Análisis del sistema*

El sistema permite a los usuarios realizar las siguientes operaciones:

Registro y autenticación de usuarios:

- Los usuarios pueden registrarse en el sistema proporcionando sus datos personales y creando una cuenta.
- La autenticación se realiza mediante JWT (JSON Web Token), lo que garantiza que solo los usuarios autenticados puedan acceder a los recursos protegidos.

Gestión de habitaciones y disponibilidad:

- La API permite la creación, modificación y eliminación de habitaciones en el hotel.
- Los administradores pueden verificar la disponibilidad de las habitaciones y actualizar su estado (disponible, ocupada).

Gestión de habitaciones y disponibilidad:

- La API permite la creación, modificación y eliminación de los hoteles.

Gestión de usuarios y roles:

- Los administradores del sistema pueden gestionar los roles y permisos de los usuarios.

Manejo de errores:

- El sistema implementa un manejo de errores robusto para garantizar que se devuelvan respuestas significativas y útiles en caso de errores.
- Se utilizan códigos de estado HTTP apropiados para indicar el resultado de las solicitudes (por ejemplo, 200 OK, 400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Internal Server Error).

Pruebas unitarias:

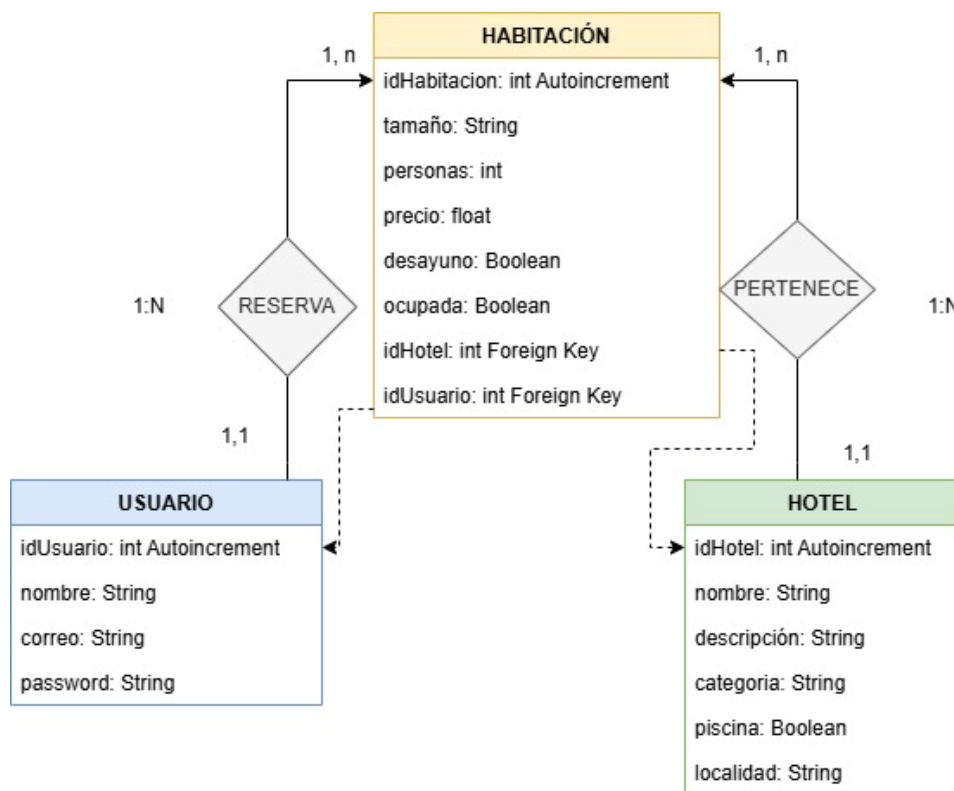
- Se han implementado pruebas unitarias para garantizar la calidad del código y la correcta funcionalidad de la API.

- Las pruebas cubren las operaciones principales como el registro de usuarios, la gestión de habitaciones y la autenticación.

Configuración de la base de datos y conexión:

- La conexión a la base de datos se realiza utilizando SQLAlchemy.
- El adaptador de base de datos utilizado es psycopg2 para PostgreSQL.
- Se ha configurado un generador de sesiones para interactuar con la base de datos de manera eficiente.

4.2 Diseño de la base de datos



El diseño de la base de datos para la gestión del hotel está estructurado para almacenar y gestionar información sobre usuarios, hoteles y habitaciones. La base de datos está organizada en tres tablas principales: usuarios, hoteles y habitaciones.

Tabla Usuario:

La tabla usuarios almacena información sobre los usuarios registrados en el sistema. Cada usuario tiene un identificador único y varios atributos que describen al usuario.

Columnas:

- **idUsuario:** Identificador único del usuario (clave primaria).
- **nombre:** Nombre del usuario.

- **correo:** Correo electrónico del usuario, que debe ser único.
- **password:** Contraseña del usuario.

Relaciones:

- Un usuario puede reservar varias habitaciones. Esta relación se refleja en la tabla habitaciones mediante una clave foránea (idUserio).

Tabla hoteles

La tabla hoteles almacena información sobre los hoteles registrados en el sistema. Cada hotel tiene un identificador único y varios atributos que describen al hotel.

Columnas:

- **idHotel:** Identificador único del hotel (clave primaria).
- **nombre:** Nombre del hotel.
- **descripcion:** Descripción del hotel.
- **categoria:** Categoría del hotel (por ejemplo, 5 estrellas).
- **piscina:** Indicador de si el hotel tiene piscina (booleano).
- **localidad:** Localidad donde se encuentra el hotel.

Relaciones:

- Un hotel puede tener muchas habitaciones. Esta relación se refleja en la tabla habitaciones mediante una clave foránea (idHotel).

Tabla habitaciones

La tabla habitaciones almacena información sobre las habitaciones de los hoteles. Cada habitación tiene un identificador único y varios atributos que describen la habitación.

Columnas:

- **idHabitacion:** Identificador único de la habitación (clave primaria).
- **tamaño:** Tamaño de la habitación.
- **personas:** Capacidad de personas de la habitación.
- **precio:** Precio de la habitación.
- **desayuno:** Indicador de si la habitación incluye desayuno (booleano).
- **ocupada:** Indicador de si la habitación está ocupada (booleano).
- **idHotel:** Identificador del hotel al que pertenece la habitación (clave foránea).
- **idUserio:** Identificador del usuario que ha reservado la habitación (clave foránea opcional).

Relaciones:

- Una habitación pertenece a un solo hotel.
- Una habitación puede ser reservada por un solo usuario a la vez.

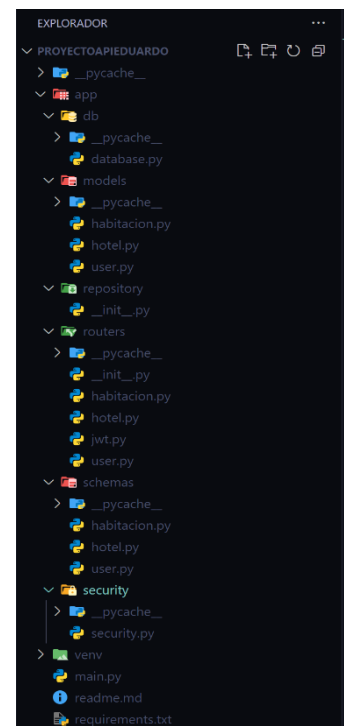
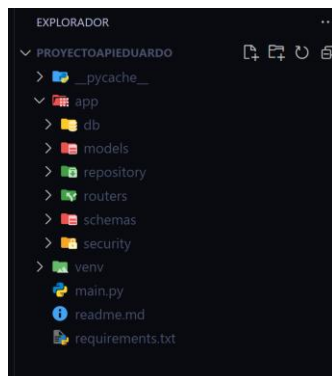
Relaciones entre tablas

La base de datos utiliza claves foráneas para establecer relaciones entre las tablas. Las relaciones principales son:

- **Usuario - Habitaciones:** Un usuario puede reservar varias habitaciones. Esta relación se representa mediante la clave foránea idUsuario en la tabla habitaciones.
- **Hotel - Habitaciones:** Un hotel puede tener varias habitaciones. Esta relación se representa mediante la clave foránea idHotel en la tabla habitaciones.

4.3 Implementación

Mi proyecto está compuesto por la siguiente estructura general:



Dentro de la carpeta APP tengo los siguientes directorios:

4.3.1 “db”

contiene los archivos necesarios para la configuración y gestión de la base de datos.

Contiene el archivo **database.py**:

- Este archivo contiene la configuración de la conexión a la base de datos utilizando SQLAlchemy. Define la URL de la base de datos, crea el motor de conexión, configura el generador de sesiones y define la clase base para los modelos de tablas.
- También incluye una función `get_db` que se utiliza para obtener una sesión de la base de datos.

```
database.py M x
app > db > database.py > ...
1 from sqlalchemy import create_engine
2 from sqlalchemy.ext.declarative import declarative_base
3 from sqlalchemy.orm import sessionmaker
4
5 #URL de La base de datos
6 SQLALCHEMY_DATABASE_URL = "postgresql://odoo:odoo@localhost:5342/hotelapiedu-database"
7
8 engine = create_engine(SQLALCHEMY_DATABASE_URL)
9 #Configura un generador de sesiones
10 SessionLocal = sessionmaker(bind=engine,autocommit=False,autoflush=False)
11 #Crea una clase base llamada Base, para definir los modelos de tablas de la base de datos.
12
13 def get_db():
14     db = SessionLocal() #Crear una nueva sesión
15     try:
16         yield db #Devuelvo la sesión para su uso
17     except Exception as e:
18         print(e)
19     finally:
20         db.close() #Cierro la sesión después de usarla
```

4.3.2 “model”

La carpeta models contiene los archivos que definen los modelos de las tablas de la base de datos.

Cada modelo representa una tabla en la base de datos y define sus columnas y relaciones.

- **hotel.py:** Define el modelo de la tabla Hotel.
- **habitacion.py:** Define el modelo de la tabla Habitacion.
- **user.py:** Define el modelo de la tabla User.

```
habitacion.py X hotel.py user.py
app > models > habitacion.py > Habitacion > personas
1 from sqlalchemy import Column, Integer, String, Boolean, DateTime, Text, Float, ForeignKey
2 from sqlalchemy.orm import relationship
3 from app.db.database import Base
4
5 # Modelo de la tabla Habitacion
6 class Habitacion(Base):
7     __tablename__ = "habitaciones"
8
9     idHabitacion = Column(Integer, primary_key=True, autoincrement=True)
10    tamaño = Column(String(100), nullable=False)
11    personas = Column(Integer, nullable=True)
12    precio = Column(Float, nullable=False)
13    desayuno = Column(Boolean, nullable=True)
14    ocupada = Column(Boolean, default=False)
15
16    # Una habitación pertenece a un solo hotel
17    idHotel = Column(Integer, ForeignKey("hoteles.idHotel"), nullable=False)
18    hotel = relationship("Hotel", back_populates="habitaciones")
19
20    # Una habitación puede ser reservada por un solo usuario a la vez
21    idUsuario = Column(Integer, ForeignKey("usuarios.idUsuario"), nullable=True)
22    usuario = relationship("User", back_populates="habitaciones")
```

```
habitacion.py hotel.py user.py X
app > models > user.py > User > nombre
1 from sqlalchemy import Column, Integer, String, Boolean, DateTime, Text, Float, ForeignKey
2 from sqlalchemy.orm import relationship
3 from app.db.database import Base
4
5 # Modelo de la tabla Usuario
6 class User(Base):
7     __tablename__ = "usuarios"
8
9     idUsuario = Column(Integer, primary_key=True, autoincrement=True, index=True)
10    nombre = Column(String, unique=True, index=True, nullable=False)
11    correo = Column(String, unique=True, index=True, nullable=False)
12    password = Column(String, nullable=False)
13
14    # Un usuario puede reservar varias habitaciones (relación uno a muchos)
15    habitaciones = relationship("Habitacion", back_populates="usuario")
```

```
habitacion.py hotel.py X user.py
app > models > hotel.py > ...
1 from sqlalchemy import Column, Integer, String, Boolean, DateTime, Text, Float, ForeignKey
2 from sqlalchemy.orm import relationship
3 from app.db.database import Base
4
5 # Modelo de la tabla Hotel
6 class Hotel(Base):
7     __tablename__ = "hoteles"
8
9     idHotel = Column(Integer, primary_key=True, autoincrement=True)
10    nombre = Column(String(100), nullable=False)
11    descripcion = Column(Text, nullable=True)
12    categoria = Column(String(50), nullable=False)
13    piscina = Column(Boolean, default=False)
14    localidad = Column(String(100), nullable=False)
15
16    # Un hotel tiene muchas habitaciones
17    habitaciones = relationship("Habitacion", back_populates="hotel")
18
```

4.3.3 “Routers”

La carpeta routers de tu proyecto contiene los archivos que definen los endpoints de la API. Estos routers permiten organizar y modularizar las rutas de la API, facilitando el mantenimiento y la escalabilidad del proyecto. El contenido de esta carpeta es el siguiente:

auth.py:

- Este archivo define los endpoints relacionados con la autenticación y gestión de tokens JWT. Incluye rutas para iniciar sesión y obtener el token de acceso, así como rutas protegidas que requieren autenticación.

user.py:

- Este archivo define los endpoints para la gestión de usuarios, incluyendo la creación, actualización, obtención y eliminación de usuarios.

hotel.py:

- Este archivo define los endpoints para la gestión de hoteles, incluyendo la creación, actualización, obtención y eliminación de hoteles.

habitacion.py:

- Este archivo define los endpoints para la gestión de habitaciones, incluyendo la creación, actualización, obtención y eliminación de habitaciones.

4.3.4 “schemas”

La carpeta schemas contiene los esquemas utilizados para la validación y serialización de datos en la API. Estos esquemas se definen utilizando Pydantic y son esenciales para garantizar que los datos enviados y recibidos en las solicitudes HTTP cumplan con los formatos esperados.

habitacion.py:

- Este archivo define los esquemas de datos relacionados con las habitaciones. Incluye esquemas para crear, actualizar y obtener información de habitaciones.

```
habitacion.py X
app > schemas > habitacion.py > HabitacionCreate
1 from pydantic import BaseModel
2 from typing import Optional
3
4 # Esquema para CREAR una habitación
5 class HabitacionCreate(BaseModel):
6     tamaño: str
7     personas: Optional[int] = None
8     precio: float
9     desayuno: Optional[bool] = False
10    ocupada: Optional[bool] = False
11    idHotel: int # Relación con el hotel
12
13 # Esquema para ACTUALIZAR habitación
14 class UpdateHabitacion(BaseModel):
15     tamaño: Optional[str] = None
16     personas: Optional[int] = None
17     precio: Optional[float] = None
18     desayuno: Optional[bool] = None
19     ocupada: Optional[bool] = None
20
21 # Esquema para RESPUESTA de habitación (con ID)
22 class HabitacionResponse(BaseModel):
23     id: int
24     tamaño: str
25     personas: Optional[int]
26     precio: float
27     desayuno: Optional[bool]
28     ocupada: Optional[bool]
29     idHotel: int
30
31 class Config:
32     from_attributes = True
```

hotel.py:

- Este archivo define los esquemas de datos relacionados con los hoteles. Incluye esquemas para crear, actualizar y obtener información de hoteles.


```

hotel.py X
app > schemas > hotel.py > HotelResponse
1 from pydantic import BaseModel
2 from typing import Optional
3
4 # Esquema para CREAR un hotel
5 class HotelCreate(BaseModel):
6     nombre: str
7     descripcion: Optional[str] = None
8     categoria: str
9     piscina: Optional[bool] = False
10    localidad: str
11
12 # Esquema para ACTUALIZAR hotel
13 class UpdateHotel(BaseModel):
14     nombre: Optional[str] = None
15     descripcion: Optional[str] = None
16     categoria: Optional[str] = None
17     piscina: Optional[bool] = None
18     localidad: Optional[str] = None
19
20 # Esquema para RESPUESTA de hotel (con ID)
21 class HotelResponse(BaseModel):
22     id: int
23     nombre: str
24     descripcion: Optional[str]
25     categoria: str
26     piscina: Optional[bool]
27     localidad: str
28
29     class Config:
30         from_attributes = True
31
32

```

user.py:

- Este archivo define los esquemas de datos relacionados con los usuarios. Incluye esquemas para crear, actualizar y obtener información de usuarios.

```

user.py M X
app > schemas > user.py > UserResponse > Config
1 from pydantic import BaseModel, EmailStr
2 from typing import Optional
3
4 # Esquema para CREAR un usuario (sin ID porque la BD lo genera)
5 class UserCreate(BaseModel):
6     username: str
7     email: EmailStr
8     password: str
9
10 # Esquema para ACTUALIZAR usuario (todos los campos opcionales)
11 class UpdateUser(BaseModel):
12     username: Optional[str] = None
13     email: Optional[EmailStr] = None
14     password: Optional[str] = None
15
16 # Esquema para RESPUESTA de usuario (con ID)
17 class UserResponse(BaseModel):
18     id: int
19     username: str
20     email: EmailStr
21
22     class Config:
23         from_attributes = True
24
25

```

4.3.5 “security”

Este archivo contiene funciones y configuraciones relacionadas con la seguridad de la API, como la generación y verificación de tokens JWT, y el hash y verificación de contraseñas.

```
security.py M X
app > security > security.py > ...
1  from datetime import datetime, timedelta
2  from jose import JWTError, jwt
3  from passlib.context import CryptContext
4
5  SECRET_KEY = "toor"
6  ALGORITHM = "HS256"
7  ACCESS_TOKEN_EXPIRE_MINUTES = 30 # 30 MINUTOS ES EL TIEMPO DE EXPIRACION DEL TOKEN
8
9  pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
10
11
12  # HASEO LA CONTRASEÑA ANTES DE ALMACENARLA EN LA BASE DE DATOS
13  def hash_password(password: str) -> str:
14      return pwd_context.hash(password)
15
16
17  # VERIFICO SI LA PASSWORD ES CORRECTA
18  def verify_password(plain_password: str, hashed_password: str) -> bool:
19      return pwd_context.verify(plain_password, hashed_password)
20
21
22  # CREAR TOKEN JWT
23  def create_access_token(data: dict, expires_delta: timedelta = None):
24      to_encode = data.copy()
25      if expires_delta:
26          expire = datetime.utcnow() + expires_delta
27      else:
28          expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
29
30      to_encode.update({"exp": expire}) # AÑADIMOS FECHA DE EXPIRACION AL TOKEN
31      encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
32      return encoded_jwt
```

4.3.6 “main.py”

El archivo main.py contiene el código necesario para inicializar y ejecutar la aplicación FastAPI.

```
main.py ×
main.py > ...
1 from fastapi import FastAPI
2 import uvicorn
3 from app.routers import jwt, user, hotel, habitacion
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
13 app.include_router(jwt.router)
14 app.include_router(user.router)
15 app.include_router(hotel.router)
16 app.include_router(habitacion.router)
17
18 if __name__ == "__main__":
19     uvicorn.run("main:app", port=8000, reload=True)
```

4.3.7 “requirements.txt”

En el archivo requirements.txt se especifican todas las dependencias y paquetes necesarios para ejecutar el proyecto. Este archivo permite instalar todas las dependencias de una vez utilizando pip.

```
requirements.txt ×
requirements.txt
1 fastapi
2 uvicorn
3 psycpg2
4 SQLAlchemy
```

4.4 Pruebas

4.4.1 Prueba API “CREAR HOTEL”

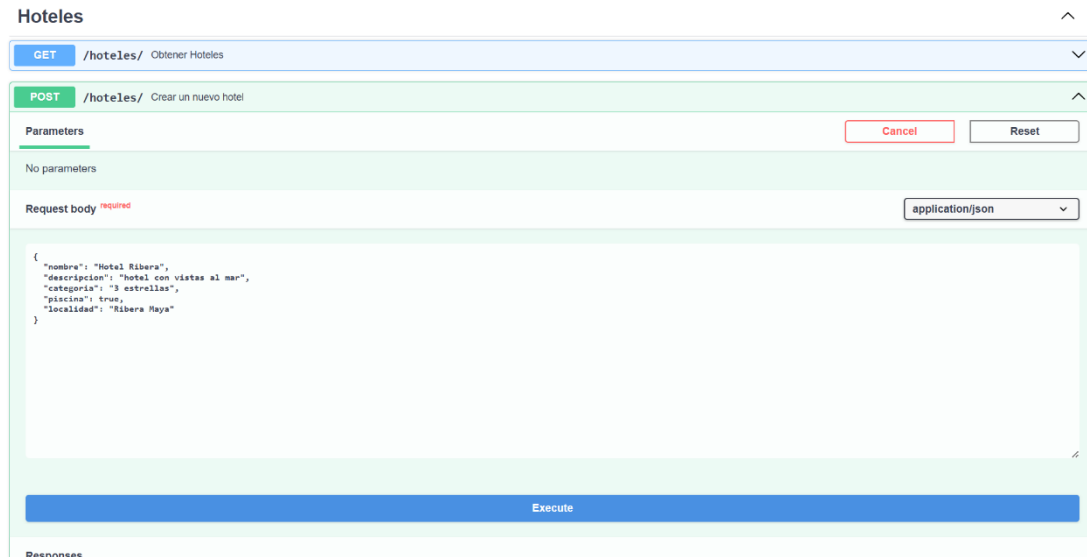
La primera prueba que realizare será sobre crear un hotel, para ello nos iremos a Swagger accediendo desde el siguiente enlace:

<http://127.0.0.1:8000/docs>

En la vista principal podremos ver los diferentes métodos CRUD que hemos creado.

Como lo que vamos a hacer es crear un hotel nos iremos al método POST de crear un nuevo hotel:

Aquí haremos click en “Try it out” y ahora introduciremos los datos correspondientes para poder crear un hotel, que yo he elegido que sean los siguientes:



Hoteles

GET /hoteles/ Obtener Hoteles

POST /hoteles/ Crear un nuevo hotel

Parameters

No parameters

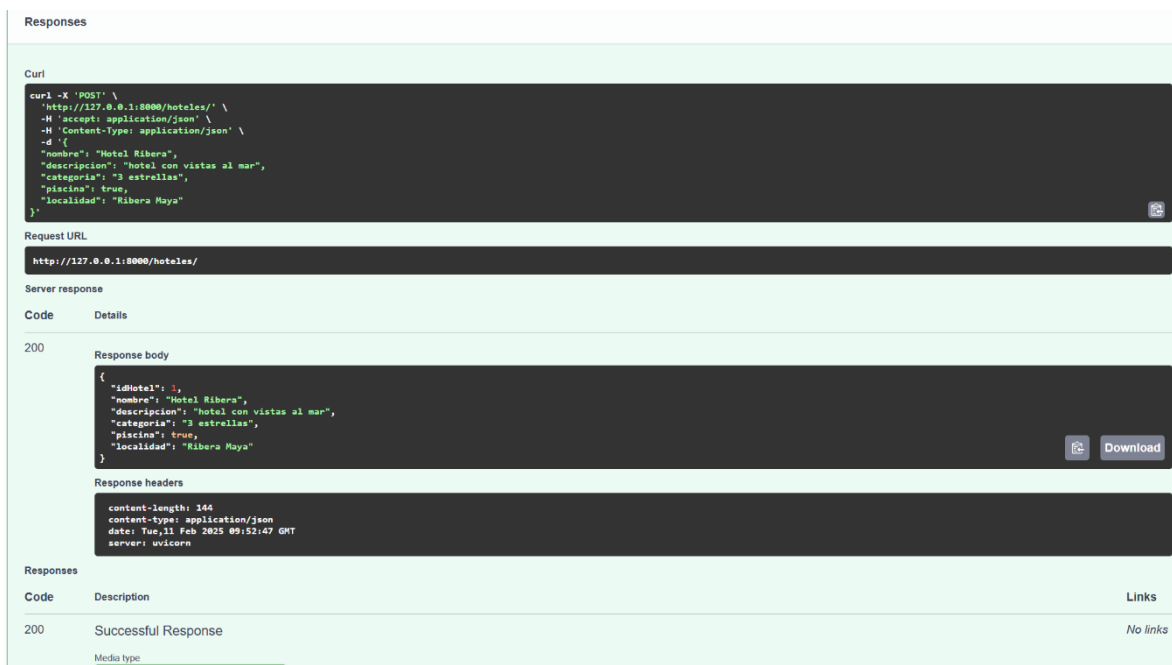
Request body **required** application/json

```
{
  "nombre": "Hotel Ribera",
  "descripcion": "hotel con vistas al mar",
  "categoria": "3 estrellas",
  "piscina": true,
  "localidad": "Ribera Maya"
}
```

Execute

Responses

Una vez hayamos pulsado en “Execute” para que se cree el hotel si todo ha salido bien nos devolverá una respuesta 200 indicándonos el hotel que se ha creado



Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/hoteles/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "nombre": "Hotel Ribera",
    "descripcion": "hotel con vistas al mar",
    "categoria": "3 estrellas",
    "piscina": true,
    "localidad": "Ribera Maya"
  }'
```

Request URL

http://127.0.0.1:8000/hoteles/

Server response

Code	Details
200	<p>Response body</p> <pre>{ "idHotel": 1, "nombre": "Hotel Ribera", "descripcion": "hotel con vistas al mar", "categoria": "3 estrellas", "piscina": true, "localidad": "Ribera Maya" }</pre> <p>Response headers</p> <pre>content-length: 364 content-type: application/json date: Tue, 11 Feb 2025 09:52:47 GMT server: uvicorn</pre>

Responses

Code	Description	Links
200	Successful Response	No links

Media type

Vamos a comprobar que el hotel se ha creado correctamente con el método get que tenemos para poder obtener todos los hoteles, simplemente como no tenemos que pasarle nada pulsamos en “Execute” y acto seguido nos devolverá el hotel que acabamos de crear:

Hoteles

GET /hoteles/ Obtener Hoteles

Parameters

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/hoteles/' \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8000/hoteles/

Server response

Code	Details
200	<p>Response body</p> <pre>{ "idHotel": 1, "nombre": "Hotel Ribera", "descripcion": "hotel con vistas al mar", "categoria": "3 estrellas", "piscina": true, "localidad": "Ribera Maya" }</pre>

4.4.2 Prueba API “CREAR HABITACIÓN”

Una vez creado un hotel pasaremos a crear una habitación, ya que estan relacionados entre si, una habitación pertenece a un solo hotel y en un hotel pueden existir varias habitaciones.

Volvemos a Swagger y nos vamos al método POST de crear una habitación, aquí añadiremos los datos de la habitación correspondiente indicando el id del hotel que acabamos de crear que es el 1.

Habitaciones

GET /habitaciones/ Obtener Habitaciones

POST /habitaciones/ Crear una nueva habitación

Parameters

No parameters

Request body ^{required}

application/json

```
{
  "tamano": "38 metros cuadrados",
  "personas": 2,
  "precio": 250.00,
  "desayuno": false,
  "ocupada": false,
  "idHotel": 1
}
```

Execute Clear

Responses

Una vez hayamos dado “Execute” podremos ver como nos aparece una respuesta 200 indicándonos que la habitación del hotel 1 se ha creado correctamente.

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/habitaciones/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "tamaño": "30 metros cuadrados",
    "personas": 2,
    "precio": 250.00,
    "desayuno": false,
    "ocupada": false,
    "idHotel": 1
  }'
```

Request URL

```
http://127.0.0.1:8000/habitaciones/
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "idHabitacion": 1, "tamaño": "30 metros cuadrados", "personas": 2, "precio": 250, "desayuno": false, "ocupada": false, "idHotel": 1 }</pre> <p>Response headers</p> <pre>content-length: 122 content-type: application/json date: Tue, 11 Feb 2025 10:00:03 GMT server: uvicorn</pre>

Responses

4.5 Despliegue de la aplicación

El despliegue de la aplicación se ha realizado utilizando Docker, una plataforma que permite empaquetar y desplegar aplicaciones de manera consistente y portátil en cualquier entorno. He utilizado Docker ya que facilita la creación de entornos de desarrollo y producción idénticos, garantizando que la aplicación funcione de la misma manera en cualquier lugar.

5 Manuales

5.1 Manual de usuario:

Acceso a la aplicación

Para acceder a la aplicación, abre tu navegador web y dirígete a la siguiente URL:

<http://127.0.0.1:8000/docs>

Esta URL te llevará a la interfaz de documentación interactiva de la API, donde podrás explorar y probar los diferentes endpoints disponibles.

Navegación por la interfaz

En la interfaz de la documentación interactiva te permite ver y probar todas las funciones de la API.

Estas son las secciones de mi API, que está dividido en USUARIOS, HOTELES, HABITACIONES:

FastAPI 0.1.0 OAS 3.1
/openapi.json

Usuarios	
GET	/usuario/obtener_usuarios Obtener Usuarios
GET	/usuario/obtener_usuario_por_id/{user_id} Obtener Usuario Por Id
POST	/usuario/crear_usuario Crear Usuario
DELETE	/usuario/eliminar_usuario/{user_id} Eliminar Usuario Por Id
PATCH	/usuario/modificar_usuario/{user_id} Actualizar Usuario Por Id
Hoteles	
GET	/hoteles/ Obtener Hoteles
POST	/hoteles/ Crear un nuevo hotel
GET	/hoteles/{hotel_id} Obtener un hotel por ID
PUT	/hoteles/{hotel_id} Actualizar un hotel
DELETE	/hoteles/{hotel_id} Eliminar un hotel
Habitaciones	
GET	/habitaciones/ Obtener Habitaciones
POST	/habitaciones/ Crear una nueva habitación
GET	/habitaciones/{habitacion_id} Obtener una habitación por ID
PUT	/habitaciones/{habitacion_id} Actualizar una habitación
DELETE	/habitaciones/{habitacion_id} Eliminar una habitación

5.1.1 Autenticación

La autenticación es necesaria para acceder a ciertas funciones protegidas de la API.

Los pasos para autenticarse son los siguientes:

Registrar un usuario (opcional):

Si aún no tienes un usuario registrado, puedes registrarte utilizando el endpoint `/users/`.

Envía una solicitud POST con los datos del usuario (nombre de usuario, correo electrónico y contraseña).

Iniciar sesión:

Utiliza el endpoint `/auth/login` para iniciar sesión.

Envía una solicitud POST con tu nombre de usuario y contraseña.

Recibirás un `access_token` que deberás usar en las solicitudes a endpoints protegidos.

Usar el token JWT:

Copia el `access_token` recibido al iniciar sesión.

Haz clic en el botón "Authorize" en la parte superior derecha de la interfaz de documentación.

Introduce el token en el campo correspondiente y haz clic en "Authorize".

5.1.2 Gestión de usuarios

Usuarios			^
GET	/usuario/obtener_usuarios	Obtener Usuarios	▼
GET	/usuario/obtener_usuario_por_id/{user_id}	Obtener Usuario Por Id	▼
POST	/usuario/crear_usuario	Crear Usuario	▼
DELETE	/usuario/eliminar_usuario/{user_id}	Eliminar Usuario Por Id	▼
PATCH	/usuario/modificar_usuario/{user_id}	Actualizar Usuario Por Id	▼
Hoteles			^

La gestión de usuarios incluye la creación, actualización, obtención y eliminación de usuarios.

Crear un usuario:

Endpoint: POST /users/

Envía una solicitud POST con los datos del usuario.

Obtener un usuario por ID:

Endpoint: GET /users/{user_id}

Envía una solicitud GET con el ID del usuario.

Actualizar un usuario:

Endpoint: PUT /users/{user_id}

Envía una solicitud PUT con los datos actualizados del usuario.

Eliminar un usuario:

Endpoint: DELETE /users/{user_id}

Envía una solicitud DELETE con el ID del usuario.

5.1.3 Gestión de hoteles

Hoteles			^
GET	/hoteles/	Obtener Hoteles	▼
POST	/hoteles/	Crear un nuevo hotel	▼
GET	/hoteles/{hotel_id}	Obtener un hotel por ID	▼
PUT	/hoteles/{hotel_id}	Actualizar un hotel	▼
DELETE	/hoteles/{hotel_id}	Eliminar un hotel	▼

La gestión de hoteles incluye la creación, actualización, obtención y eliminación de hoteles.

Crear un hotel:

Endpoint: POST /hoteles/

Envía una solicitud POST con los datos del hotel.

Obtener un hotel por ID:

Endpoint: GET /hoteles/{hotel_id}

Envía una solicitud GET con el ID del hotel.

Actualizar un hotel:

Endpoint: PUT /hoteles/{hotel_id}

Envía una solicitud PUT con los datos actualizados del hotel.

Eliminar un hotel:

Endpoint: DELETE /hoteles/{hotel_id}

Envía una solicitud DELETE con el ID del hotel.

5.1.4 Gestión de habitaciones

Habitaciones			^
GET	/habitaciones/	Obtener Habitaciones	▼
POST	/habitaciones/	Crear una nueva habitación	☑
GET	/habitaciones/{habitacion_id}	Obtener una habitación por ID	▼
PUT	/habitaciones/{habitacion_id}	Actualizar una habitación	▼
DELETE	/habitaciones/{habitacion_id}	Eliminar una habitación	▼

La gestión de habitaciones incluye la creación, actualización, obtención y eliminación de habitaciones.

Crear una habitación:

Endpoint: POST /habitaciones/

Envía una solicitud POST con los datos de la habitación.

Obtener una habitación por ID:

Endpoint: GET /habitaciones/{habitacion_id}

Envía una solicitud GET con el ID de la habitación.

Actualizar una habitación:

Endpoint: PUT /habitaciones/{habitacion_id}

Envía una solicitud PUT con los datos actualizados de la habitación.

Eliminar una habitación:

Endpoint: DELETE /habitaciones/{habitacion_id}

Envía una solicitud DELETE con el ID de la habitación.

5.2 Manual de instalación:

Este manual proporciona una guía paso a paso para desplegar la aplicación en un entorno local utilizando Docker, PostgreSQL y FastAPI, y poder ejecutarla de manera eficiente.

Requisitos previos

Para ejecutar la aplicación, primero tendremos que asegurarnos de tener las siguientes herramientas instaladas:

- **Docker:** Para crear y ejecutar contenedores de forma sencilla.
- **Docker Compose:** Para orquestar los contenedores, como PostgreSQL y la API.
- **pgAdmin:** Para gestionar la base de datos PostgreSQL de manera visual.
- **Python:** Para ejecutar la aplicación FastAPI.
- **pip:** Para gestionar las dependencias del proyecto.

1. Ejecutar Docker

Lo primero será asegurarnos de que Docker y Docker Compose están instalados y funcionando en nuestra máquina.

2. Configurar PostgreSQL en pgAdmin

Ahora habrá que gestionar la base de datos PostgreSQL, para ello habrá que seguir estos pasos:

Abrir pgAdmin: Abre la aplicación pgAdmin. Si estás usando Docker, asegúrate de que el contenedor de PostgreSQL esté funcionando y accesible en el puerto especificado.

Conectar a la base de datos:

Host: localhost

Puerto: 5432 (puerto predeterminado de PostgreSQL)

Usuario: pgadmin4@pgadmin.org

Contraseña: admin (o la que hayas definido en el archivo docker-compose.yml)

3. Crear la base de datos:

Haz clic derecho en el servidor de PostgreSQL y selecciona "Create > Database".

Asigna un nombre a la base de datos, en este caso, hotelapiedu-database.

4. Construir y ejecutar los contenedores de Docker

Desde el directorio del proyecto, ejecuta los siguientes comandos para construir y ejecutar los contenedores:

```
bash
```

```
docker-compose up --build
```

5. Instalar las dependencias de nuestro proyecto:

Este comando lee el archivo requirements.txt y descarga e instala automáticamente todas las librerías y paquetes necesarios para ejecutar la aplicación.

```
pip install -r requirements.txt
```

6. Iniciar la API de FastAPI

Si todo está configurado correctamente, se puede iniciar la API de FastAPI en el contenedor de Docker:

Ejecutar la API con FastAPI:

```
"Python main.py"
```

Verificar que la API esté funcionando: Accede a la API a través de tu navegador en <http://127.0.0.1:8000/docs>, donde se mostrará una interfaz para probar los endpoints de la API.

7. Obtener Token de JWT:

Obtener el token JWT: Realiza una solicitud POST a /auth/login con las credenciales de un usuario válido. Si el login es exitoso, recibirás un access_token en la respuesta.

Hacer una solicitud a una ruta protegida: Usa el token JWT obtenido anteriormente en las cabeceras de la solicitud para acceder a las rutas protegidas.

8. Hacer métodos CRUD en FastAPI

Una vez esté funcionando la API, podemos comenzar a hacer consultas a los endpoints definidos.

6 Conclusiones y posibles ampliaciones

Dificultades encontradas en el desarrollo de la aplicación: Durante el desarrollo de esta API para la gestión de un hotel, me he encontrado varias dificultades, entre las cuales destacan:

1. Configuración de la autenticación mediante JWT:

- Implementar un sistema de autenticación seguro fue un desafío, especialmente al asegurarse de que los tokens JWT fueran generados y verificados correctamente.

- Garantizar la seguridad de las contraseñas almacenadas en la base de datos mediante el uso de hashing también presentó complejidades.

2. Relaciones en la base de datos:

- Definir y gestionar las relaciones entre las tablas usuarios, hoteles y habitaciones requirió una comprensión sólida de SQLAlchemy y las claves foráneas.
- Asegurar la integridad referencial en la base de datos fue una tarea que exigió atención a los detalles.

3. Manejo de errores:

- Implementar un manejo de errores robusto para devolver respuestas significativas y útiles en caso de fallos en las solicitudes fue crucial, pero también complejo.

4. Validación y serialización de datos:

- Utilizar Pydantic para validar y serializar los datos entrantes y salientes en la API fue una parte esencial del desarrollo, pero requirió un aprendizaje considerable y ajustes continuos.

Grado de satisfacción en el trabajo realizado: A pesar de las dificultades encontradas, el grado de satisfacción en el trabajo realizado es alto. La implementación de la API ha permitido:

- **Centralización y facilidad de gestión:** La API facilita la gestión de usuarios, hoteles y habitaciones de manera centralizada y eficiente.
- **Autenticación segura:** La implementación de JWT garantiza que solo los usuarios autenticados puedan acceder a los recursos protegidos, aumentando la seguridad general del sistema.
- **Código estructurado y modular:** El uso de routers y esquemas ha permitido una organización clara y una fácil mantenibilidad del código.

Aprendizaje: El proceso de desarrollo de esta aplicación me ha proporcionado un aprendizaje en:

- **Desarrollo con FastAPI:** Adquirir habilidades en el uso de FastAPI para crear APIs rápidas y eficientes.

- **Gestión de bases de datos con SQLAlchemy:** Comprender mejor cómo definir modelos y gestionar relaciones en una base de datos.
- **Seguridad y autenticación:** Aprender a implementar sistemas de autenticación seguros utilizando JWT y hashing de contraseñas.
- **Validación y serialización con Pydantic:** Utilizar Pydantic para asegurar la validez de los datos entrantes y salientes en la API.

Posibles ampliaciones: Una posible ampliación para este proyecto podría ser la **implementación de un sistema de reservas**. Esto permitiría a los usuarios realizar reservas de habitaciones a través de la API, gestionando las fechas de entrada y salida, y asegurando la disponibilidad de las habitaciones. Esta funcionalidad añadiría un valor significativo al sistema, proporcionando una solución completa para la gestión hotelera.

7 Bibliografía

- **García, L. (2023).** *Desarrollo de aplicaciones web con FastAPI*. Editorial Tecnología y Software. Madrid, España.
- **Martínez, A., & López, P. (2021).** *Bases de datos y ORM con SQLAlchemy*. Editorial Técnicas Informáticas. Barcelona, España.
- *FastAPI Documentation*. Retrieved from <https://fastapi.tiangolo.com/>
- *SQLAlchemy Documentation*. Retrieved from <https://www.sqlalchemy.org/>
- *Pydantic Documentation*. Retrieved from <https://pydantic-docs.helpmanual.io/>