



# PROYECTO SGE

CFGS Desarrollo de Aplicaciones Multiplataforma  
Informática y Comunicaciones

**Desarrollo de una aplicación mediante una API  
hecha con FastAPI**

*Año: 2025*

*Fecha de presentación: 11/02/2025*

**Nombre y Apellidos: Isaac González Adeva**

**Email: [isaac.gonade@educa.jcyl.es](mailto:isaac.gonade@educa.jcyl.es)**

# Índice

1. Introducción .....	3
2. Estado del arte .....	3
3. Descripción del proyecto .....	4
4. Documentación técnica .....	5
5. Manuales .....	11
6. Conclusiones .....	16
7. Bibliografía .....	17

## 1. Introducción

Este proyecto consiste en una API desarrollada con Python y con FastAPI consiste en la gestión y búsqueda de hoteles. La API contiene una base de datos que consta de 3 tablas, la tabla "Hotel" que contiene toda la información de un hotel además de las habitaciones que contiene, la tabla "Habitación" tiene todos los datos de las habitaciones y por último la tabla "User" que tiene el nombre de usuario y la contraseña necesarios para poder autenticarse y acceder a algunos métodos que están restringidos. La aplicación puede buscar hoteles, crearlos, crear habitaciones en un hotel, editarlas o borrarlas.

## 2. Estado del arte

### Definición de arquitectura de microservicios

La arquitectura de microservicios es un modelo de desarrollo de software que organiza una aplicación en múltiples servicios autónomos. Cada uno de estos servicios cumple una función específica y se comunica con los demás a través de API. Gracias a esta estructura, los microservicios pueden desarrollarse, implementarse y escalarse de forma independiente.

### Definición de API

Una API (*Application Programming Interface*) es un conjunto de reglas y definiciones que permite que diferentes sistemas, aplicaciones o servicios se comuniquen entre sí. Facilita la comunicación entre sistemas, incluso con distinto lenguaje. También permite acceder a funciones sin necesidad de conocer el código.

Hay diferentes tipos de API;

- **REST:** están basadas en HTTP y usan métodos como GET, POST, PUT o DELETE
- **SOAP:** utilizan XML y son más comunes en sistemas empresariales
- **GraphQL:** permiten consultar solo los datos necesarios
- **WebSockets:** facilitan la comunicación en tiempo real

### Estructura de una API

**Endpoints:** Un endpoint es una URL específica a través de la cual se accede a una API. Cada endpoint es como una dirección especial de sitio web que se utiliza para hablar con la API, cada uno puede representar una función o recurso único ofrecido por la API. Son cruciales en el diseño de una API porque definen su estructura y el modo en que los clientes pueden interactuar con ella.

**Recursos:** los recursos son los datos u objetos que una API puede proporcionar o manipular. Los recursos pueden ser entidades tangibles, como perfiles de usuario o productos, o conceptos abstractos, como un token de autenticación. Son el núcleo de una API, y los puntos finales definen cómo se accede a estos recursos o cómo se modifican.

**Métodos:** Los métodos, a menudo denominados métodos HTTP, son los verbos del mundo API. Describen las acciones que se pueden realizar sobre los recursos a través de los endpoints. El método le dirá al servidor lo que esperamos hacer con la información enviada en nuestra petición.

### Formas de crear una API en python: FastAPI

He utilizado FastAPI porque es lo visto en clase y además porque es un proceso sencillo y estructurado.

**Paso 1** – Configurar el entorno de desarrollo, instalar Python y crear un entorno virtual.

**Paso 2** – Estructurar el proyecto, crear una carpeta principal llamada “app” y dentro de ella crear otras carpetas para las diferentes clases; “routers”, “models”, “repository” y “db”, además del fichero requirements.txt con las dependencias del proyecto.

**Paso 3** – Configurar la base de datos con SQLAlchemy en la clase database.py en la carpeta “db”

**Paso 4** – Crear los modelos de la base de datos dentro de la carpeta “models”

**Paso 5** – Definir los esquemas con Pydantic en la carpeta “repository”

**Paso 6** – Crear los endpoints en la carpeta “routers”

**Paso 7** – Configurar el main.py para inicializar la aplicación

**Paso 8** – Ejecutar el servidor mediante uvicorn

## 3. Descripción del proyecto

### Objetivos

El objetivo del proyecto era crear una API que fuese funcional en todos los sentidos y que contase con seguridad al acceder a determinados métodos. Que se pudiesen agregar datos a la base de datos, recuperarlos, borrarlos o actualizarlos.

## Entorno de trabajo

En este trabajo he utilizado una base de datos en PostgreSQL, donde se reflejaban las operaciones de la API y sin necesitar Docker.

Como IDE he utilizado Pycharm ya que su interfaz es muy similar a la del IntelliJ al que ya estaba acostumbrado y como lenguaje Python.

## 4. Documentación técnica

### Análisis del sistema

Para comenzar, en el main.py creo las tablas de primeras

```
#Funcion para crear todas las tablas
def create_tables(): 1 usage 1 isaac
    Base.metadata.create_all(bind=engine)

create_tables()
```

Después para autenticar a un usuario utilizo un token el cual genero desde aquí comprobando que el usuario existe en la base de datos:

```
SECRET_KEY = "secreta_clave"
ALGORITHM = "HS256"

def create_token(data: dict): 1 usage 1 isaac
    token = jwt.encode(data, SECRET_KEY, algorithm=ALGORITHM)
    return token

@router.post("/token") 1 isaac
def login(form_data: OAuth2PasswordRequestForm = Depends(), db:Session = Depends(get_db)):
    user = db.query(User).filter(and_(User.username == form_data.username, User.password == hashlib.sha256(form_data.password.encode()).hexdigest())).first()
    if user:
        token = create_token(data={"sub":user.username})
        return {
            "access_token": token,
            "token_type":"bearer"
        }
```

Para obtener los hoteles por localidad o por categoría utilizo el mismo método, pero cambiando los datos, filtro dándole formato a la cadena introducida y después devuelvo los hoteles que lo cumplan:

```
@router.get("/localidad/{localidad}")  # isaac
def get_hotel_by_localidad(localidad:str, db:Session = Depends(get_db)):
    hoteles = db.query(models.Hotel).filter(models.Hotel.localidad.ilike(f"%{localidad}%")).all()
    return hoteles

@router.get("/categoria/{categoria}")  # isaac
def get_hotel_by_categoria(categoria:str, db:Session = Depends(get_db)):
    hoteles = db.query(models.Hotel).filter(models.Hotel.categoria.ilike(f"%{categoria}%")).all()
    return hoteles
```

Para insertar un nuevo hotel, creo un objeto de hotel y le paso todos los parámetros del json:

```
@router.get("/localidad/{localidad}")  # isaac
def get_hotel_by_localidad(localidad:str, db:Session = Depends(get_db)):
    hoteles = db.query(models.Hotel).filter(models.Hotel.localidad.ilike(f"%{localidad}%")).all()
    return hoteles

@router.get("/categoria/{categoria}")  # isaac
def get_hotel_by_categoria(categoria:str, db:Session = Depends(get_db)):
    hoteles = db.query(models.Hotel).filter(models.Hotel.categoria.ilike(f"%{categoria}%")).all()
    return hoteles
```

Para insertar una nueva habitación primero tengo que ver si existe el hotel donde la quiero insertar para ello le paso el id del hotel necesario y después en el body pongo los demás parámetros de la habitación gracias “shemas” donde tengo los datos para trabajar con las clases:

```
@router.post("/nueva/{id_hotel}/habitacion")  # isaac
def insert_habitacion(id_hotel: int, habitacion: schemas.Habitacion, db: Session = Depends(get_db), token:str = Depends(JWTAuth.oauth2_scheme)):
    hotel = db.query(models.Hotel).filter(models.Hotel.id == id_hotel).first()
    if not hotel:
        raise HTTPException(status_code=404, detail="Hotel no encontrado")

    newHabitacion = models.Habitacion()
    newHabitacion.tamano = habitacion.tamano
    newHabitacion.precio = habitacion.precio
    newHabitacion.desayuno = habitacion.desayuno
    newHabitacion.ocupada = habitacion.ocupada
    newHabitacion.id_hotel = id_hotel

    db.add(newHabitacion)
    db.commit()
    db.refresh(newHabitacion)

    return newHabitacion
```

## ISAAC GONZÁLEZ ADEVA MEMORIA PROYECTO FINAL

Para borrar una habitación simplemente le paso el id de la habitación a borrar:

```
@router.delete("/borrar/{id_habitacion}")  # isaac
def delete_habitacion(id_habitacion:int, db:Session = Depends(get_db), token:str = Depends(JWTAuth.oauth2_scheme)):
    habitacion = db.query(models.Habitacion).filter(models.Habitacion.id == id_habitacion).first()
    if habitacion:
        db.delete(habitacion)
        db.commit()
        return {"Mensaje": "Habitacion borrada"}
    else:
        raise HTTPException(status_code=404, detail="Habitacion no encontrada")
```

Para insertar lo mismo, le pido un id de habitacion y después modifico lo que quiera de la habitación:

```
@router.put("/editar/{id_habitacion}")  # isaac
def update_habitacion(id_habitacion:int, habitacion:schemas.Habitacion, db:Session = Depends(get_db), token:str = Depends(JWTAuth.oauth2_scheme)):
    newHabitacion = db.query(models.Habitacion).filter(models.Habitacion.id == id_habitacion).first()
    if newHabitacion:
        newHabitacion.tamano = habitacion.tamano
        newHabitacion.precio = habitacion.precio
        newHabitacion.desayuno = habitacion.desayuno
        newHabitacion.ocupada = habitacion.ocupada

        db.commit()
        return newHabitacion
    else:
        raise HTTPException(status_code=404, detail="Habitacion no encontrada")
```

La clase schemas tiene los constructores de las clases con los datos que sean necesarios:

```
# CREATE TABLE user (
#     id INT AUTO_INCREMENT PRIMARY KEY,
#     username VARCHAR(255) NOT NULL,
#     password VARCHAR(255) NOT NULL
# );
class User(BaseModel): 1 usage # isaac
    username: str
    password: str

# CREATE TABLE hotel (
#     id INT AUTO_INCREMENT PRIMARY KEY,
#     nombre VARCHAR(255) NOT NULL,
#     descripcion VARCHAR(255) NOT NULL,
#     categoria VARCHAR(255) NOT NULL,
#     piscina BOOLEAN,
#     localidad VARCHAR(255) NOT NULL
# );
class Hotel(BaseModel): 1 usage # isaac
    nombre: str
    descripcion: str
    categoria: str
    piscina: bool
    localidad: str

# CREATE TABLE habitacion (
#     id INT AUTO_INCREMENT PRIMARY KEY,
#     tamano INT NOT NULL,
#     precio INT NOT NULL,
#     desayuno BOOLEAN default false,
#     ocupada BOOLEAN default false,
#     id_hotel INT NOT NULL,
#     FOREIGN KEY (id_hotel) REFERENCES hotel(id) ON DELETE CASCADE
# );
class Habitacion(BaseModel): 2 usages # isaac
    tamano: int
    precio: int
    desayuno: bool
    ocupada: bool
```

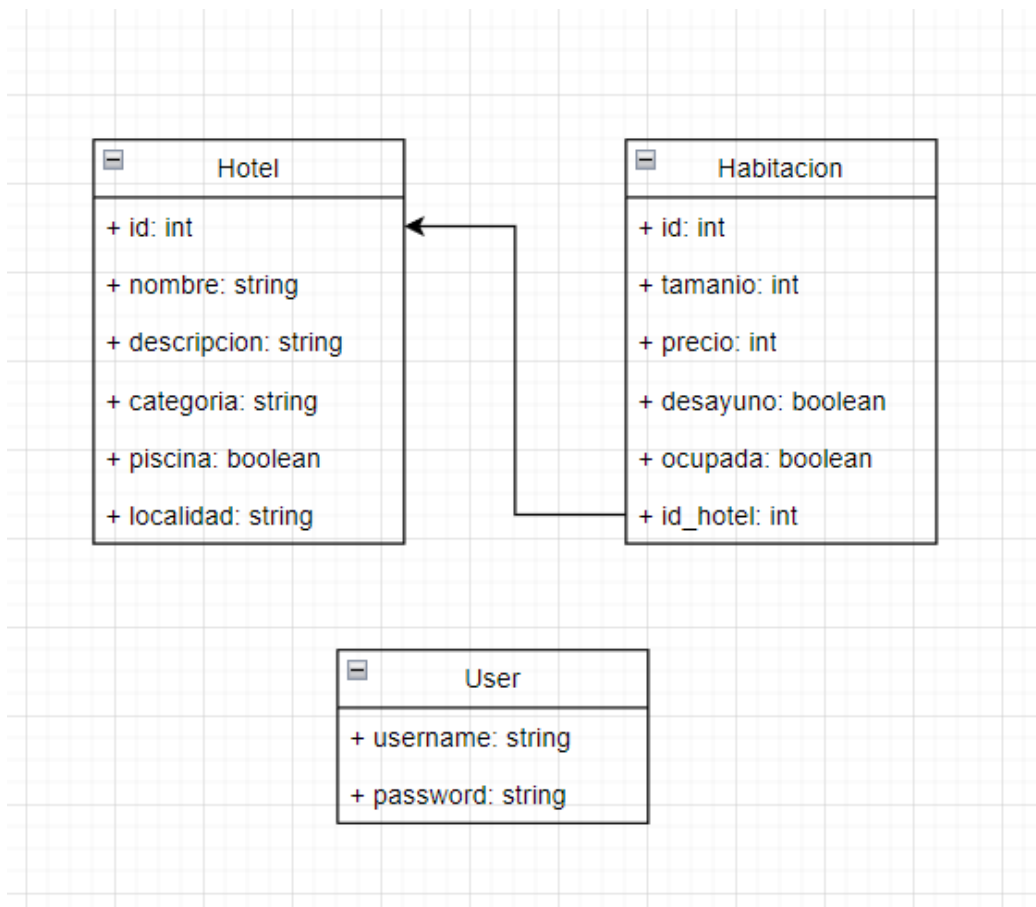
Para acceder a la dirección de la base de datos utilizo la clase database.py:

```
#url de la base de datos
SQLALCHEMY_DATABASE_URL = "postgresql://postgres:admin@localhost:5432/apiHoteles"
# CREAR MOTOR DE CONEXION QUE INTERACTUARA CON LA BD UTILIZANDO LA URL
engine = create_engine(SQLALCHEMY_DATABASE_URL)
# CONFIGURA UN GENERADOR DE SESIONS
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
# CREAR BASE PARA DEFINIR LOS MODELOS DE LA TABLA BD
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

## Diseño de la base de datos

En cuanto al diseño de la base de datos he creado 3 tablas diferentes;





La tabla User va aparte, pero un hotel puede tener muchas habitaciones (OneToMany) mientras que una habitación solo puede pertenecer a un hotel (ManyToOne).

Tabla Hotel:

```
class Hotel(Base):
    __tablename__ = "hotel"
    id = Column(Integer, primary_key=True, autoincrement=True)
    nombre = Column(String)
    descripcion = Column(String)
    categoria = Column(String)
    piscina = Column(Boolean, default=False)
    localidad = Column(String)

    habitaciones = relationship("Habitacion", back_populates="hotel",
                                cascade="delete,merge")
```

Tabla Habitación:

```
class Habitacion(Base):
    __tablename__ = "habitacion"
    id = Column(Integer, primary_key=True, autoincrement=True)
    tamaño = Column(Integer)
    precio = Column(Integer)
    desayuno = Column(Boolean, default=False)
    ocupada = Column(Boolean, default=False)
    id_hotel = Column(Integer, ForeignKey("hotel.id", ondelete="CASCADE"))

    hotel = relationship("Hotel", back_populates="habitaciones")
```

```
# CREATE TABLE habitacion (
#   id INT AUTO_INCREMENT PRIMARY KEY,
#   tamaño INT NOT NULL,
#   precio INT NOT NULL,
#   desayuno BOOLEAN default false,
#   ocupada BOOLEAN default false,
#   id_hotel INT NOT NULL,
#   FOREIGN KEY (id_hotel) REFERENCES hotel(id) ON DELETE CASCADE
# );
```

Tabla User:

```
class User(Base):
    __tablename__ = "user"
    id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String)
    password = Column(String)

# CREATE TABLE user (
#   id INT AUTO_INCREMENT PRIMARY KEY,
#   username VARCHAR(255) NOT NULL,
#   password VARCHAR(255) NOT NULL
# );
```

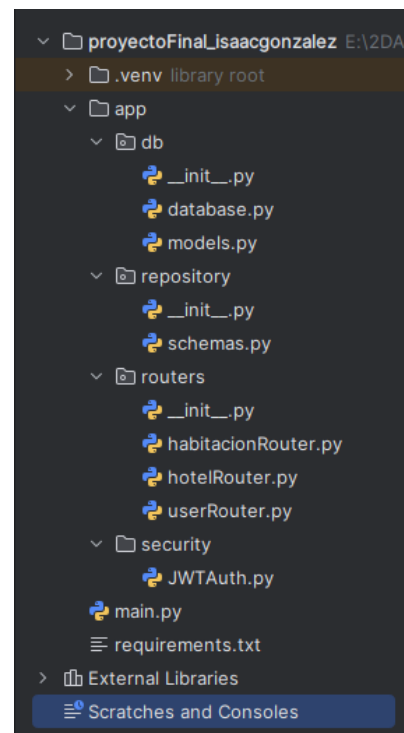
## Implementación

He creado la carpeta app y dentro de ella diferentes paquetes. El paquete db guarda la conexión a la base de datos y su estructura en el "models.py". el paquete repository contiene la clase schemas.py que sirve para manejar de los datos. Luego está la carpeta routers que son las rutas a las que se les piden cosas y por último la carpeta security ya que he metido el JWT para verificar que el usuario existe.

### Dependencias:

```
fastapi~=0.115.8
uvicorn~=0.34.0
psycopg2
SQLAlchemy~=2.0.38
jose~=1.0.0
pydantic~=2.10.6
jwt~=1.3.1
```

- FastApi: FastAPI es un framework de Python para construir APIs web de alto rendimiento.
- Uvicorn: Uvicorn es un servidor ASGI (*Asynchronous Server Gateway Interface*).
- Psycopg2: Es un adaptador de PostgreSQL para Python.
- SQLAlchemy: Es una biblioteca de ORM (*Object-Relational Mapping*).
- Jose: Es una implementación de JOSE (JavaScript Object Signing and Encryption) en Python.
- Pydantic: Es una biblioteca para validación de datos basada en tipado estático.
- Jwt: Es otra biblioteca para manejar JSON Web Tokens (JWT).



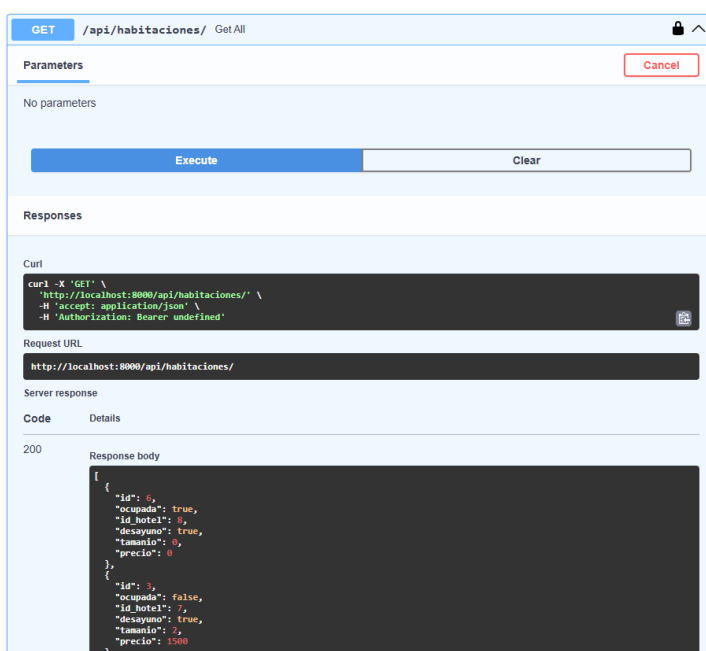
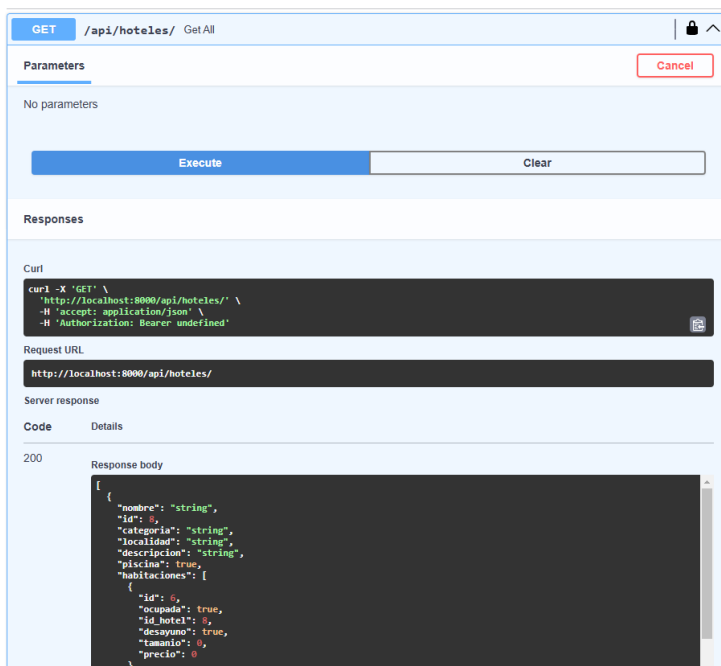
## Despliegue:

Se hará en un servidor local porque evita problemas de seguridad, permite una depuración más rápida y se tiene un mayor control a la hora de modificar el código y ver los cambios inmediatamente.

## 5. Manuales

### Manual de usuario

Como funcionalidades básicas están las de listar todos los hoteles o todas las habitaciones, que muestra una lista de todos sin ninguna restricción.



Además de listar todos los hoteles, también se puede buscar un hotel por su localidad o por su categoría

GET

/api/hoteles/localidad/{localidad}

Get Hotel By Localidad

⌵

Parameters

Cancel

Name	Description
localidad * required	
string	
(path)	

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8000/api/hoteles/localidad/madrid' \
  -H 'accept: application/json'
```

Request URL

http://localhost:8000/api/hoteles/localidad/madrid

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{   "nombre": "Hotel 1",   "id": 1,   "categoria": "3 estrellas",   "localidad": "madrid",   "descripcion": "hotel",   "piscina": true }</pre></div><div><div>Download</div></div></div>

GET

/api/hoteles/categoria/{categoria}

Get Hotel By Categoria

⌵

Parameters

Cancel

Name	Description
categoria * required	
string	
(path)	

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8000/api/hoteles/categoria/3estrellas' \
  -H 'accept: application/json'
```

Request URL

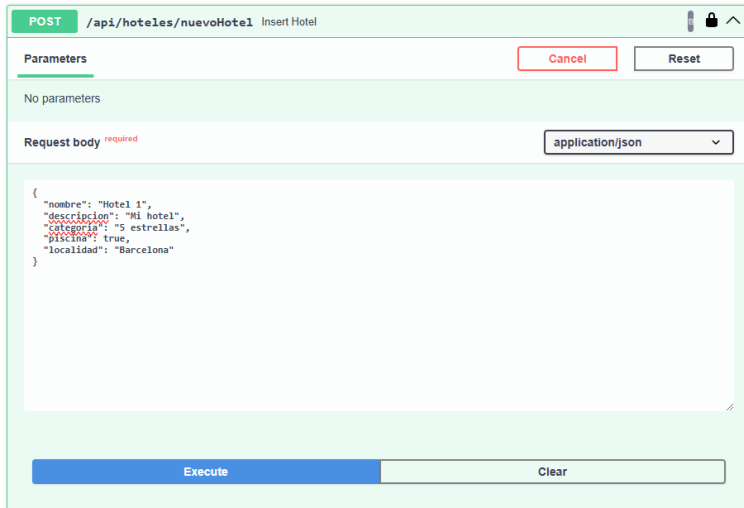
http://localhost:8000/api/hoteles/categoria/3estrellas

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{   "nombre": "Hotel 1",   "id": 1,   "categoria": "3 estrellas",   "localidad": "madrid",   "descripcion": "hotel",   "piscina": true }</pre></div><div><div>Download</div></div></div>

ISAAC GONZÁLEZ ADEVA  
MEMORIA PROYECTO FINAL

A la hora de insertar en la base de datos se puede crear un nuevo hotel y una nueva habitación, para insertar una habitación tiene que existir algún hotel primero ya que se pide el id del hotel para crearla:



POST /api/hoteles/nuevoHotel Insert Hotel

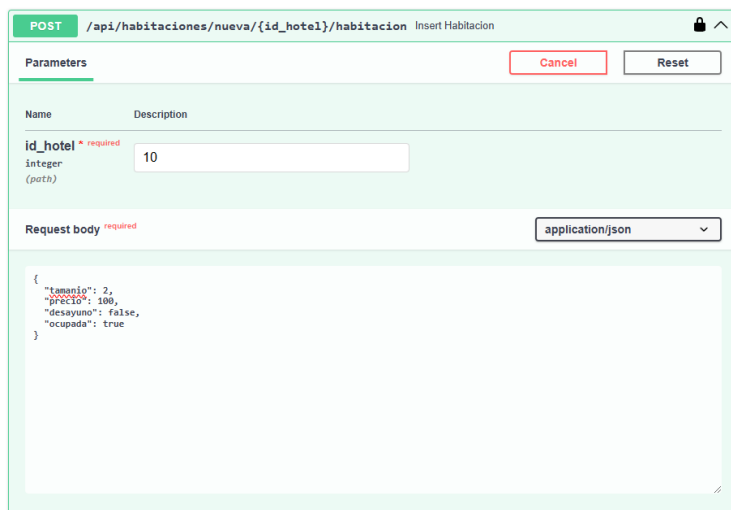
Parameters

No parameters

Request body <sup>required</sup> application/json

```
{
  "nombre": "Hotel 1",
  "descripcion": "Mi hotel",
  "categoria": "5 estrellas",
  "piscina": true,
  "localidad": "Barcelona"
}
```

Execute Clear



POST /api/habitaciones/nueva/{id\_hotel}/habitacion Insert Habitacion

Parameters

Name	Description
id_hotel * <sup>required</sup>	10

integer (path)

Request body <sup>required</sup> application/json

```
{
  "temperatura": 2,
  "precio": 100,
  "desayuno": false,
  "ocupada": true
}
```

Las habitaciones también se pueden eliminar y modificar:

DELETE /api/habitaciones/borrar/{id\_habitacion} Delete Habitación

Parameters

Name	Description
id_habitacion * required	
integer (path)	10

Execute Clear

PUT /api/habitaciones/editar/{id\_habitacion} Update Habitación

Parameters

Name	Description
id_habitacion * required	
integer (path)	10

Request body required application/json

```
{  "temperatura": 2,  "aportador": 80,  "desayunos": true,  "recapalar": false}
```

Cancel Reset

Para poder autenticarse se puede utilizar el usuario por defecto Root – pwd: Toor, pero si se desea tener mas usuarios con los que acceder también se pueden crear:

POST /api/users/crearUsuario Create User

Parameters

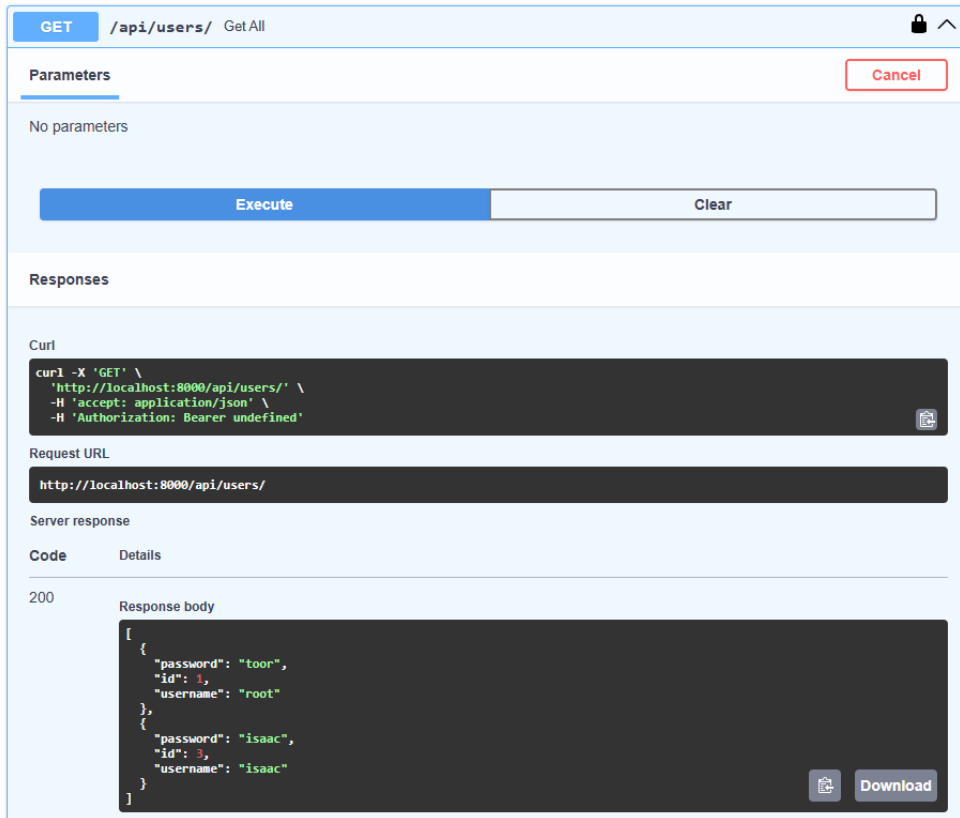
No parameters

Request body required application/json

```
{  "username": "isaac",  "password": "isaac"}
```

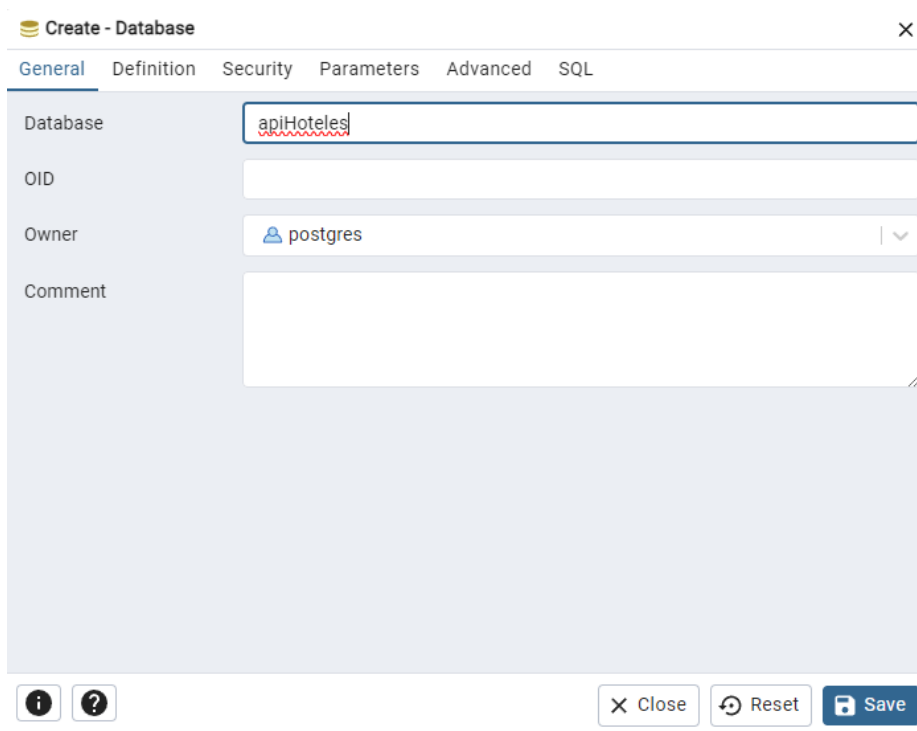
Execute Clear

Y luego los puedes listar para verlos:



## Manual de instalación

Lo primero que hay que hacer es instalar PostgreSQL y crear la base de datos en PgAdmin:



La ruta de la base de datos será:

"postgresql://username:password@localhost:5432/nombreBD"

Una vez creada la base de datos, se inicia la API desde VScode o Pycharm poniendo la ruta especificada anteriormente para que pueda acceder y que te salgan todos los métodos.

```
INFO: Will watch for changes in these directories: ['E:\\2DAM\\SGE\\EJERCICIOS\\FastAPIproy\\proyectoFinal_isaacgonzalez']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [6052] using StatReload
INFO: Started server process [13144]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

## 6. Conclusiones

Desarrollar una aplicación con FastAPI te permite realizar consultas de manera mas rapida y sencilla que los metodos normales, es bastante sencillo de hacer e intuitivo, aunque la autentificacion fue un poco complicada de entender como hacerla. He hecho lo que queria hacer, pero se puede mejorar de muchas formas.

Una posible ampliacion seria implementar la API en la aplicacion movil que por diferentes motivos no pude realizar. Seria una app de reservas de hoteles con opciones de modificacion para usuarios desarrolladores.



## 7. Bibliografía

<https://www.sensedia.com.es/pillar/deconstruccion-de-las-api-componentes-y-estructura>

<https://aws.amazon.com/es/what-is/api/>

<https://www.xataka.com/basics/api-que-sirve>

<https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces>

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

<https://blog.back4app.com/es/como-construir-una-api-rest/>

<https://learn.microsoft.com/es-es/azure/architecture/best-practices/api-design>