



PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones
Multiplataforma
Informática y Comunicaciones

APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

Año: 2024/2025

Fecha de presentación: 10/02/2025

Nombre y Apellidos: Estela de Vega Martín
Email: estela.vegmar@educa.jcyl.es

ÍNDICE

Introducción.....	3
Estado del arte	4
Arquitectura de microservicios	4
API	5
Estructura de una API	6
Formas de crear una API.....	7
Descripción general del proyecto	10
Objetivos.....	10
Entorno de trabajo	10
Documentación técnica	12
Análisis del sistema.....	12
Diseño de la base de datos	13
Tablas de la base de datos	13
Diagrama de la base de datos	14
Implementación	15
Pruebas	28
Despliegue de la aplicación.....	30
Manuales	31
Manual de usuario.....	31
Manual de instalación	43
Conclusiones y posibles ampliaciones.....	45
Dificultades encontradas en el desarrollo de la aplicación.....	45
Grado de satisfacción en el trabajo realizado	45
Aprendizaje durante el desarrollo.....	45
Posibles ampliaciones.....	46
Bibliografía.....	47

Introducción

Este proyecto tiene como objetivo desarrollar una API RESTful utilizando FastAPI como backend. De esta forma, el proyecto ofrecerá la gestión de datos relacionados con usuarios, cubos de Rubik y tiempos de resolución, permitiendo la creación, lectura, actualización y eliminación de registros (CRUD).

La aplicación está diseñada para interactuar con una base de datos relacional que almacena información estructurada en tres modelos principales: **Usuarios**, **Cubos** y **Tiempos de Resolución**. Para garantizar la seguridad de los datos y el acceso controlado a la información, se ha implementado un sistema de **autenticación basado en JWT (JSON Web Tokens)**.

El proyecto está diseñado para ser ejecutado por aplicaciones clientes a través de peticiones HTTP, utilizando el intercambio de datos en formato **JSON**.

Estado del arte

Arquitectura de microservicios

La **arquitectura de microservicios** es un enfoque moderno para el desarrollo de software que divide una aplicación en servicios independientes que se comunican entre sí mediante API. Esta estructura facilita la escalabilidad, la rapidez en el desarrollo y la flexibilidad en la implementación, en comparación con las arquitecturas monolíticas.

Características Clave:

- **Servicios Independientes:** Cada microservicio puede desarrollarse, desplegarse y mantenerse de forma autónoma sin afectar a otros.
- **Facilidad de Mantenimiento:** Permite actualizaciones rápidas, pruebas eficientes y un aislamiento efectivo de errores.
- **Equipos Pequeños y Autónomos:** Los servicios suelen ser gestionados por equipos pequeños que siguen metodologías ágiles y DevOps.
- **Enfoque en Capacidades Empresariales:** Los servicios se organizan en torno a funciones específicas del negocio.
- **Automatización:** Uso de CI/CD para facilitar la integración y el despliegue continuo de servicios.

Por ejemplo, en una tienda online, cada función (cuentas, inventario, pagos, envíos) puede gestionarse como un microservicio separado, facilitando su desarrollo y escalabilidad.

Ventajas vs. Monolitos:

A diferencia de las arquitecturas monolíticas, los microservicios permiten mayor flexibilidad, escalado independiente de cada componente y una rápida adaptación a cambios.

Tecnologías Comunes:

Contenedores: **Docker** es una plataforma que facilita la creación, distribución y ejecución de aplicaciones en contenedores ligeros y portátiles. Estos contenedores permiten que los microservicios se ejecuten de manera aislada, asegurando compatibilidad entre diferentes entornos (desarrollo, pruebas, producción).

Algunas de las ventajas de usar Docker en microservicios son:

- **Aislamiento:** Cada microservicio se ejecuta en su propio contenedor, lo que evita conflictos de dependencias.
- **Portabilidad:** Los contenedores pueden desplegarse en cualquier sistema que soporte Docker, ya sea local o en la nube.
- **Escalabilidad:** Docker permite replicar y escalar servicios fácilmente, adaptándose a la demanda del sistema.

- **Despliegue Rápido:** Los contenedores se inician rápidamente, lo que agiliza el ciclo de desarrollo e implementación continua (CI/CD).
- **Facilidad de Mantenimiento:** Los microservicios pueden actualizarse o reemplazarse sin afectar al resto de la aplicación.

Orquestación: Kubernetes para gestionar y escalar contenedores.

Este enfoque distribuye la carga de trabajo, mejora la resiliencia del sistema y permite una gestión más eficiente del software.

API

Una **API** (Application Programming Interface) es un conjunto de definiciones y protocolos que permite la comunicación entre diferentes aplicaciones de software. Actúa como un intermediario que define cómo deben interactuar dos sistemas, facilitando el desarrollo e integración de nuevas funcionalidades sin necesidad de conocer la lógica interna del software.

¿Para Qué Sirve una API?

- **Facilita el Desarrollo:** Permite a los desarrolladores integrar funciones existentes (como pagos con PayPal o inicios de sesión con Google) sin tener que programarlas desde cero.
- **Interconexión de Servicios:** Conecta aplicaciones entre sí, como apps móviles con redes sociales o servicios de notificaciones.
- **Reutilización de Código:** Evita "reinventar la rueda", utilizando APIs públicas o privadas para acelerar el desarrollo.
- **Seguridad y Control:** Permite que servicios externos accedan a funcionalidades específicas sin exponer el código fuente completo.

Tipos de API:

- **Públicas:** Abiertas para que cualquier desarrollador las utilice.
- **Privadas:** Restringidas para uso interno de una empresa.
- **De Partners:** Accesibles solo para socios estratégicos.
- **Locales o Remotas:** Según si la comunicación es dentro del mismo sistema o a través de Internet.

Ejemplos de Uso:

- Iniciar sesión en una app con tu cuenta de Facebook.

- Procesar pagos online usando la API de un banco.
- Consultar publicaciones de Twitter desde una app de terceros.
- Enviar datos de compra desde una web a un sistema de verificación de tarjetas.

Estructura de una API

Una API se organiza en varias partes clave que permiten la comunicación entre aplicaciones y la manipulación de datos de forma estructurada.

1. **Protocolo utilizado:** El protocolo más comúnmente utilizado para las APIs es **HTTP** (Hypertext Transfer Protocol), que es el mismo protocolo usado por los navegadores web para cargar sitios. Dentro de este protocolo, se utilizan diferentes **métodos HTTP** para definir las acciones que el cliente puede realizar sobre los recursos disponibles en la API.
2. **Métodos HTTP:** Los métodos son los verbos que indican qué acción se va a realizar sobre un recurso:
 - **GET:** Se utiliza para **recuperar** información del servidor. Por ejemplo, obtener los detalles de un producto.
 - **POST:** Se usa para **crear** nuevos recursos en el servidor, como agregar un nuevo usuario o producto.
 - **PUT:** Sirve para **actualizar** un recurso existente, como modificar la información de un producto o usuario.
 - **DELETE:** Se emplea para **eliminar** un recurso del servidor, por ejemplo, borrar un artículo de la base de datos.
3. **Partes de la URL de una API:** La URL de una API está dividida en varios elementos que permiten acceder y especificar los recursos o acciones a realizar:
 - **Base URL:** Es la dirección principal de la API, que sirve como punto de entrada. Ejemplo: <https://api.example.com>.
 - **Endpoint:** Es la ruta que indica el recurso o función específica de la API. En el caso de una API meteorológica, un endpoint podría ser `/weather/current`, que accedería a la información sobre el clima actual.
 - **Parámetros:** Son datos adicionales que modifican o especifican aún más la solicitud. Por ejemplo, un endpoint como

/weather?city=London puede incluir un parámetro que indique la ciudad para la cual se desea obtener el pronóstico del tiempo.

Formas de crear una API

En Python, existen varias formas de crear una API, pero **FastAPI** y **Flask** son dos de las opciones más populares. A continuación, vamos a detallar las características clave de ambas, así como los pasos de implementación, para comparar y justificar la elección de una u otra.

1. Flask: Micro-framework flexible

Flask es un micro-framework de Python muy popular para el desarrollo de aplicaciones web y APIs. Su diseño minimalista y su flexibilidad permiten al desarrollador construir aplicaciones web de manera sencilla. Aunque es conocido por su simplicidad, Flask no incluye muchas herramientas preconfiguradas, lo que significa que deberás implementar funciones adicionales de forma manual.

Características de Flask:

- **Micro-framework:** Proporciona lo esencial para crear una API o aplicación web, pero no incluye características como validación de datos o documentación de la API por defecto.
- **Flexibilidad:** Permite un control total sobre la estructura de la aplicación. Ideal para aplicaciones donde se necesita una configuración más detallada.
- **Documentación y validación manual:** Aunque puedes añadir bibliotecas como **Marshmallow** para validación o **Flasgger** para documentación, estos elementos no son nativos.

- **Pasos básicos para crear una API con Flask:**

1. **Instalación:** pip install flask

2. **Creación de una ruta simple:**

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api', methods=['GET'])
def get_data():
    return jsonify({"message": "Hello world!"})

if __name__ == '__main__':
    app.run(debug=True)
```

3. **Adición de validación y documentación:** Se necesitarían

bibliotecas externas, como **Marshmallow** para la validación y **Flasgger** para generar la documentación automática, lo cual implica mayor esfuerzo.

¿Cuándo usar Flask?

- Proyectos donde necesitas un alto nivel de personalización y control.
- Ideal para aplicaciones web más complejas donde puedes integrar la API como una parte de una aplicación más grande.
- Si no necesitas funcionalidades automáticas de validación o documentación.

2. FastAPI: Framework moderno y optimizado

FastAPI es un framework relativamente nuevo que ha ganado popularidad rápidamente por su alto rendimiento y características avanzadas para la construcción de APIs. Está diseñado para ser más rápido que Flask y Django en cuanto a la creación de APIs RESTful, y lo consigue gracias al uso de **ASGI** (Asynchronous Server Gateway Interface) y al aprovechamiento de **Python 3.6+**.

Características de FastAPI:

- **Alto rendimiento:** FastAPI está basado en **Starlette**, lo que le permite ser extremadamente rápido, especialmente cuando se manejan grandes cantidades de solicitudes.
- **Validación automática de datos:** Utiliza las funciones de **Pydantic** para la validación de datos. Esto reduce significativamente el trabajo manual que tendrías que hacer en Flask.
- **Documentación automática:** FastAPI genera documentación interactiva y fácil de usar gracias a **Swagger** y **ReDoc**. Con solo definir los tipos de datos, genera automáticamente la documentación de la API.
- **Soporte para asincronía:** Gracias al soporte completo de **asyncio**, FastAPI es ideal para aplicaciones que manejan múltiples solicitudes simultáneas de forma eficiente.

Pasos para crear una API con FastAPI:

1. Instalación:

```
pip install fastapi
```

```
pip install uvicorn
```


2. **Creación de una ruta simple:**
3. **Validación de datos:** FastAPI permite validar datos de entrada de forma automática utilizando **Pydantic**:

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: str = None

@router.post("/api/items")
def create_item(user: User):
    return {"name": user.name, "age": user.age}
```

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/my-first-api")
def hello():
    return {"Hello world!"}
```

4. **Documentación automática:** Con FastAPI, la documentación está lista desde el principio, accesible en /docs o /redoc.

¿Cuándo usar FastAPI?

- Ideal para APIs que requieren alto rendimiento y escalabilidad.
- Perfecto cuando se necesita validación y documentación automática sin tener que depender de bibliotecas adicionales.
- Proyectos donde la asincronía es importante (por ejemplo, aplicaciones de microservicios o servicios web que manejan múltiples solicitudes simultáneas).

Framework a utilizar

Para este proyecto, se optará por **FastAPI** por varias razones:

- **Velocidad y rendimiento:** FastAPI es significativamente más rápido, lo que es clave para un API que pueda manejar grandes volúmenes de tráfico.
- **Documentación automática:** La documentación generada automáticamente a través de **Swagger** y **ReDoc** facilita el trabajo y la integración con otros desarrolladores o equipos.
- **Validación de datos:** La validación automática usando **Pydantic** reduce la necesidad de manejar validaciones manualmente, lo que acelera el desarrollo.

Descripción general del proyecto

Objetivos

El objetivo principal de este proyecto es **desarrollar una API RESTful para la gestión de usuarios, cubos de Rubik y sus tiempos**, utilizando **FastAPI** como framework principal. Esta API permite a los usuarios crear y gestionar sus datos y los datos de los tipos de cubos y registrar y gestionar los tiempos de resolución de cubos de Rubik. Además, se ha integrado un sistema de **autenticación y autorización**, de manera que los usuarios solo puedan acceder a sus propios datos.

El proyecto tiene como objetivo adicional la **seguridad** de los datos a través de autenticación basada en **JWT**, el **almacenamiento** de los tiempos y los **detalles del usuario** en una base de datos, y la posibilidad de ampliar el proyecto para futuras funcionalidades, como la integración con una aplicación móvil.

Todo esto busca proporcionar una plataforma sencilla para la gestión de los usuario con sus tipos de cubos y sus tiempos.

Entorno de trabajo

Para desarrollar este proyecto, se han utilizado diversas herramientas y tecnologías para este desarrollo:

1. **Python**: El lenguaje de programación principal utilizado para este proyecto es **Python**, debido a su versatilidad, facilidad de uso y amplia compatibilidad con diversas librerías y frameworks. Python es ideal para la creación de APIs debido a su simplicidad y su gran ecosistema de herramientas.
2. **FastAPI**: es el framework utilizado para construir la API. Se eligió FastAPI por su rendimiento, facilidad de uso y la capacidad de generar documentación automática (con Swagger) que facilita la interacción con la API destacándose así por su rapidez en la creación de endpoints y su compatibilidad con la validación de datos mediante **Pydantic**.
3. **PostgreSQL con pgAdmin**: Para el almacenamiento de datos, se utilizó **PostgreSQL**, una base de datos relacional eficiente. **pgAdmin** es la herramienta utilizada para administrar y gestionar la base de datos de manera visual. PostgreSQL fue seleccionado por su capacidad para manejar datos, su integridad y confiabilidad en el manejo de las relaciones entre tablas.

4. **Docker:** se utilizó para **contenerizar** el proyecto y gestionar de manera aislada tanto el backend (FastAPI) como la base de datos (PostgreSQL). Gracias a Docker, se simplificó la configuración del entorno de desarrollo y se evitó la posible incompatibilidad entre entornos locales y de producción. Docker asegura que la aplicación se ejecute de manera consistente en cualquier máquina
5. **PyCharm:** fue el **IDE** utilizado para el desarrollo del código. Es un entorno de desarrollo especializado en Python, que facilita la escritura, depuración y gestión del código. PyCharm incluye características útiles como la integración con bases de datos y la visualización de estructuras de datos. Además, tiene soporte para la ejecución de contenedores Docker, lo cual permitió trabajar con la API de forma fluida dentro del entorno aislado.
6. **Git y GitHub:** Para el control de versiones y la gestión del código fuente, se utilizó **Git**, un sistema de control de versiones distribuido ampliamente utilizado en proyectos de software. El proyecto fue subido a **GitHub**, lo que permitió tener un control completo sobre los cambios realizados en el código.
7. **Vivaldi:** fue el navegador utilizado para probar la API y la interacción con el backend. A través directamente desde el navegador, se realizaron pruebas de los endpoints para verificar el correcto funcionamiento de la API.

Flujo de trabajo:

- **Desarrollo:** El desarrollo del backend se realizó dentro de un contenedor Docker que ejecuta FastAPI y PostgreSQL. Esto permitió realizar pruebas y ajustes de manera eficiente.
- **Base de datos:** La gestión de los datos de usuarios y cubos de Rubik se realizó en PostgreSQL, permitiendo una estructurada de la información en tablas relacionadas.
- **Control de versiones:** Se utilizó Git para gestionar los cambios en el proyecto, lo que permitió tener un historial de versiones y una integración continua de los cambios.

Documentación técnica

Análisis del sistema

La aplicación se compone de una **API RESTful** desarrollada con **FastAPI**, que permite la gestión de usuarios, cubos de Rubik y tiempos de resolución. A continuación se detallan las funcionalidades principales:

- **Autenticación y Autorización:**
 - Se utiliza **JWT (JSON Web Tokens)** para la autenticación y autorización de usuarios.
 - El proceso de autenticación incluye un **sistema de login** que genera un token JWT al verificar las credenciales del usuario.
 - Las **rutas protegidas** requieren que el usuario proporcione un token válido en la cabecera Authorization para acceder a recursos sensibles.
 - El token contiene información del usuario (ID) y una fecha de expiración para mayor seguridad.
- **Gestión de Usuarios:**
 - **Registro de Usuarios:** Permite crear nuevas cuentas de usuario, almacenando contraseñas de forma segura mediante hashing con **bcrypt**.
 - **Inicio de Sesión:** Autenticación de usuarios mediante la verificación de credenciales y generación de tokens JWT.
 - **Actualización de Datos:** Los usuarios pueden modificar su información personal de forma segura.
 - **Eliminación de Datos:** Los usuarios pueden eliminar su información personal de forma segura,
- **Manejo de Errores:**
 - Implementación de **excepciones personalizadas** para gestionar errores comunes.
 - Ejemplos de errores manejados:
 - **Credenciales inválidas:** Devuelve un código de estado 400 con un mensaje de error.

- **Token expirado o inválido:** Devuelve un error 401 indicando la necesidad de autenticación.
 - **Usuario no encontrado:** Devuelve un error 404 cuando se intenta acceder a un usuario inexistente.
- **Pruebas Unitarias:**

Verificar que el proceso de inicio de sesión (login) de un usuario funcione correctamente. En esta prueba, se valida que un usuario pueda iniciar sesión con sus credenciales correctas y recibir un token de acceso JWT como respuesta.

Diseño de la base de datos

El diseño de la base de datos para este proyecto tiene como objetivo almacenar y gestionar la información de los usuarios, los cubos de Rubik que poseen y los tiempos de resolución que registran. La base de datos está compuesta por tres tablas principales: **users**, **cubes** y **solve_times**, las cuales están interrelacionadas.

Tablas de la base de datos

1. Tabla users:

- Esta tabla almacena los datos básicos de los usuarios, como su **identificador** único (id), **nombre** de usuario (username), dirección de **correo electrónico** (email) y **contraseña** (password).
- Cada usuario puede tener uno o más cubos de Rubik. Además, cada usuario puede registrar múltiples tiempos de resolución, lo que crea una relación uno a muchos con la tabla solve_times.

2. Tabla cubes:

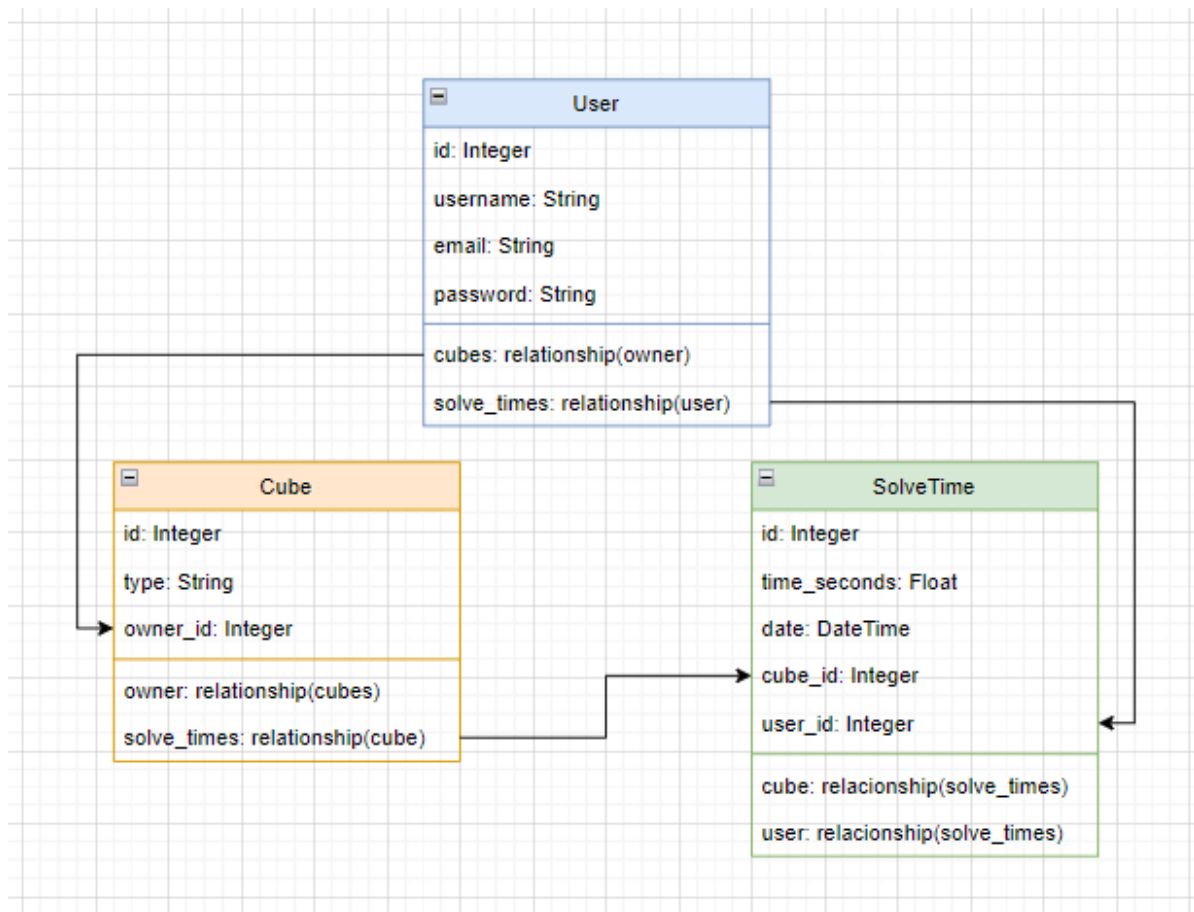
- Esta tabla almacena la información sobre los cubos de Rubik. Cada cubo tiene un **identificador** único (id) y un campo para el **tipo de cubo** (type), que indica qué tipo de cubo es (por ejemplo, 3x3, 4x4, etc.).
- Cada cubo está asociado a un usuario. Además, un cubo puede tener varios tiempos de resolución registrados, lo que crea una relación uno a muchos con la tabla solve_times.

3. Tabla solve_times:

- Esta tabla almacena los tiempos de resolución de los cubos de Rubik. Cada tiempo tiene un **identificador** único (id), el **tiempo** de resolución en segundos (time_seconds) y la **fecha del registro** (date), que se establece automáticamente con la fecha y hora actual.
- Cada tiempo está asociado a un tipo de cubo. Además, cada tiempo está asociado a un usuario.

Diagrama de la base de datos

Con lo descrito anteriormente, el modelo relacional de la base de datos quedaría de la siguiente forma:



Implementación

Estructura del Código

1. **app/main.py:** Aquí se inicializa y configura la aplicación FastAPI, donde se incluyen los routers que gestionan las rutas de la API.

```
from fastapi import FastAPI
import uvicorn
from app.routers import user, cube, solve_time, auth
from app.db.database import Base, engine

# FUNCION PARA CREAR LAS TABLAS DEFINIDAS EN LOS MODELOS
def create_tables():
    Base.metadata.create_all(bind=engine) # CREA LAS TABLAS SI NO EXISTEN

# CREAR TABLAS AL INICIAR LA API
create_tables()

# INSTANCIA PRINCIPAL DE LA API
app = FastAPI(
    title="Rubik API",
    description="API para gestionar usuarios, cubos de Rubik y tiempos de resolución",
    version="1.0.0"
)

# INCLUIAMOS LAS RUTAS DE USUARIO, CUBOS Y TIEMPOS
app.include_router(user.router)
app.include_router(cube.router)
app.include_router(solve_time.router)

# PUNTO DE ENTRADA PRINCIPAL
if __name__ == "__main__":
    # EJECUTAMOS EL SERVIDOR UVICORN EN EL PUERTO 8000 CON RECARGA AUTOMÁTICA
    uvicorn.run(app="app.main:app", host="127.0.0.1", port=8000, reload=True)
```

2. **app/models/:** En esta carpeta se definen los modelos de la base de datos utilizando SQLAlchemy. Los modelos son representaciones de las tablas en la base de datos, y son utilizados por SQLAlchemy para realizar las operaciones CRUD. Aquí se definen los atributos de los modelos:

- **User:**

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

# MODELO DE USUARIO
class User(Base):
    __tablename__ = "users" # NOMBRE DE LA TABLA EN LA BASE DE DATOS

    # COLUMNAS DE LA TABLA USERS
    id = Column(Integer, primary_key=True, index=True) # ID UNICO DEL USUARIO
    username = Column(String, unique=True, index=True, nullable=False) # NOMBRE DE USUARIO UNICO
    email = Column(String, unique=True, index=True, nullable=False, default=None) # EMAIL UNICO
    password = Column(String, nullable=False) # CONTRASEÑA

    # RELACIONES
    # UN USUARIO PUEDE TENER VARIOS CUBOS
    cubes = relationship(argument="Cube", back_populates="owner")
    # UN USUARIO PUEDE REGISTRAR VARIOS TIEMPOS
    solve_times = relationship(argument="SolveTime", back_populates="user")
```

○ Cube

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

# MODELO DE CUBO DE RUBIK
class Cube(Base): 9 usages 1 estelaV9
    __tablename__ = "cubes" # NOMBRE DE LA TABLA EN LA BASE DE DATOS

    # COLUMNAS DE LA TABLA CUBES
    id = Column(Integer, primary_key=True, index=True) # ID UNICO DEL CUBO
    type = Column(String, nullable=False) # TIPO DE CUBO

    # CLAVE FORANEA QUE RELACIONA EL CUBO CON UN USUARIO
    owner_id = Column(Integer, ForeignKey("users.id"))

    # RELACIONES CON OTRAS TABLAS
    # EL USUARIO ASOCIADO
    owner = relationship(argument="User", back_populates="cubes")
    # TIEMPOS DE RESOLUCION ASOCIADOS AL CUBO
    solve_times = relationship(argument="SolveTime", back_populates="cube")
```

○ Solve Time

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

# MODELO DE LOS TIEMPOS
class SolveTime(Base): 9 usages 1 estelaV9
    __tablename__ = "solve_times" # NOMBRE DE LA TABLA EN LA BASE DE DATOS

    # COLUMNAS DE LA TABLA SOLVE_TIMES
    id = Column(Integer, primary_key=True, index=True) # ID UNICO DEL TIEMPO
    time_seconds = Column(Float, nullable=False) # TIEMPO EN SEGUNDOS
    date = Column(DateTime, default=datetime.utcnow) # FECHA DEL REGISTRO

    # CLAVES FORANEAS PARA RELACIONAR EL CUBO Y EL USUARIO
    cube_id = Column(Integer, ForeignKey("cubes.id"))
    user_id = Column(Integer, ForeignKey("users.id"))

    # RELACIONES CON EL CUBO Y EL USUARIO
    cube = relationship(argument="Cube", back_populates="solve_times")
    user = relationship(argument="User", back_populates="solve_times")
```

3. **app/security/**: Aquí se encuentra un módulos que se encargan de la gestión de la seguridad, como el hash de contraseñas y la creación y verificación de tokens JWT. Esta verifica que las credenciales del usuario estén protegidos. Aquí se implementan funciones para el hasheo seguro de contraseñas (usando passlib) y la generación de tokens de acceso (usando python-jose).


```
from jose import JWTError, jwt
from passlib.context import CryptContext

SECRET_KEY = "hola"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 60 # TIEMPO DE EXPIRACION DEL TOKEN (1 HORA)

# CONTEXTO PARA HASHEAR Y VERIFICAR CONTRASEÑAS CON BCrypt
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# HASHEAR LA CONTRASEÑA ANTES DE GUARDARLA EN LA BD
def hash_password(password: str) -> str: 5 usages 1 estelaV9
    return pwd_context.hash(password)

# VERIFICAR SI LA CONTRASEÑA PROPORCIONADA ES CORRECTA
def verify_password(plain_password: str, hashed_password: str) -> bool: 4 usages 1 estelaV9
    return pwd_context.verify(plain_password, hashed_password)

# CREAR EL TOKEN JWT
from datetime import datetime, timedelta, timezone

def create_access_token(data: dict, expires_delta: timedelta = None): 4 usages 1 estelaV9 *
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.now(timezone.utc) + expires_delta
    else:
        expire = datetime.now(timezone.utc) + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

# DECODIFICAR EL TOKEN JWT Y VERIFICAR SU VALIDEZ
def verify_token(token: str): 2 usages 1 estelaV9
    try:
        token_data = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return token_data # SI ES VALIDO, DEVUELVE EL CONTENIDO DEL TOKEN
    except JWTError:
        return None
```

4. **app/crud/**: Esta carpeta contiene los módulos encargados de las operaciones CRUD (crear, leer, actualizar, eliminar) sobre los modelos de base de datos.

- **User:**

```
from sqlalchemy.orm import Session
from app.models.user import User
from app.schemas.user import UpdateUser, UserCreate

# CREAR UN NUEVO USUARIO
def create_user(db: Session, user: UserCreate):
    db_user = User(username=user.username, email=user.email, password=user.password)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user

# OBTENER UN USUARIO POR ID
def get_user(db: Session, user_id: int):
    return db.query(User).filter(User.id == user_id).first()

# ACTUALIZAR UN USUARIO
def update_user(db: Session, user_id: int, user: UpdateUser):
    db_user = db.query(User).filter(User.id == user_id).first()
    if db_user:
        if user.username:
            db_user.username = user.username
        if user.password:
            db_user.password = user.password
        if user.email:
            db_user.email = user.email
        db.commit()
        db.refresh(db_user)
    return db_user

# ELIMINAR UN USUARIO
def delete_user(db: Session, user_id: int):
    user = db.query(User).filter(User.id == user_id).first()
    if not user:
        return None
    db.delete(user)
    db.commit()
    return user
```

○ **Cube:**

```
from sqlalchemy.orm import Session
from app.models.cube import Cube
from app.schemas.cube import CubeCreate

# CREAR UN NUEVO CUBO
def create_cube(db: Session, cube: CubeCreate): 2 usages 1 estelaV9
    db_cube = Cube(type=cube.type, owner_id=cube.owner_id)
    db.add(db_cube)
    db.commit()
    db.refresh(db_cube)
    return db_cube

# OBTENER UN CUBO POR ID
def get_cube(db: Session, cube_id: int): 3 usages 1 estelaV9
    return db.query(Cube).filter(Cube.id == cube_id).first()

# OBTENER TODOS LOS CUBOS
def get_cubes(db: Session): 2 usages 1 estelaV9
    return db.query(Cube).all()

# ACTUALIZAR UN CUBO
def update_cube(db: Session, cube_id: int, cube: CubeCreate): 2 usages 1 estelaV9
    db_cube = db.query(Cube).filter(Cube.id == cube_id).first()
    if db_cube:
        db_cube.type = cube.type
        db_cube.owner_id = cube.owner_id
        db.commit()
        db.refresh(db_cube)
        return db_cube
    return None

# ELIMINAR UN CUBO
def delete_cube(db: Session, cube_id: int): 2 usages 1 estelaV9
    cube = db.query(Cube).filter(Cube.id == cube_id).first()
    if not cube:
        return None
    db.delete(cube)
    db.commit()
    return cube
```

○ **SolveTime:**

```
from sqlalchemy.orm import Session
from app.models.solve_time import SolveTime
from app.schemas.solve_time import SolveTimeCreate

# CREAR UN NUEVO TIEMPO
def create_solve_time(db: Session, solve_time: SolveTimeCreate): 2 usages 1 estelaV9
    db_solve_time = SolveTime(time_seconds=solve_time.time_seconds,
                               date=solve_time.date,
                               cube_id=solve_time.cube_id,
                               user_id=solve_time.user_id)
    db.add(db_solve_time)
    db.commit()
    db.refresh(db_solve_time)
    return db_solve_time

# OBTENER UN TIEMPO POR ID
def get_solve_time(db: Session, solve_time_id: int): 3 usages 1 estelaV9
    return db.query(SolveTime).filter(SolveTime.id == solve_time_id).first()

# OBTENER TODOS LOS TIEMPOS
def get_solve_times(db: Session): 2 usages 1 estelaV9
    return db.query(SolveTime).all()

# ACTUALIZAR UN TIEMPO
def update_solve_time(db: Session, solve_time_id: int, solve_time: SolveTimeCreate):
    db_solve_time = db.query(SolveTime).filter(SolveTime.id == solve_time_id).first()
    if db_solve_time:
        db_solve_time.time_seconds = solve_time.time_seconds
        db_solve_time.date = solve_time.date
        db_solve_time.cube_id = solve_time.cube_id
        db_solve_time.user_id = solve_time.user_id
        db.commit()
        db.refresh(db_solve_time)
        return db_solve_time
    return None

# ELIMINAR UN TIEMPO
def delete_solve_time(db: Session, solve_time_id: int): 2 usages 1 estelaV9
    solve_time = db.query(SolveTime).filter(SolveTime.id == solve_time_id).first()
    if not solve_time:
        return None
    db.delete(solve_time)
    db.commit()
    return solve_time
```

5. **app/db/database.py:** Este archivo se encarga de la configuración y conexión de la base de datos. Aquí se configura la conexión a la base de datos utilizando SQLAlchemy.

```
# PERMITE ESTABLECER LA CONEXION CON UNA BD ESPECIFICANDO LA URL DE CONEXION
from sqlalchemy import create_engine

# CLASE BASE PARA DEFINIR MODELOS DE TABLAS DE BD EN SQLAlchemy
from sqlalchemy.orm import declarative_base

# CLASE QUE CREA OBJETOS PARA MANEJAR SESIONES CON LA BD
from sqlalchemy.orm import sessionmaker

# URL DE LA BASE DE DATOS
""" - postgresql: El tipo de base de datos.
    - odoo:odoo: Las credenciales de usuario (usuario: odoo, contraseña: odoo).
    - localhost:5342: Dirección del servidor de base de datos (localhost y puerto 5342).
    - fastapi-database: Nombre de la base de datos. """
SQLALCHEMY_DATABASE_URL = "postgresql://odoo:odoo@localhost:5342/rubikapi-database"

# CREAR MOTOR DE CONEXION QUE INTERACTUARA CON LA BD UTILIZANDO LA URL
engine = create_engine(SQLALCHEMY_DATABASE_URL)

# CONFIGURA UN GENERADOR DE SESIONES
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)

# CREAR BASE PARA DEFINIR LOS MODELOS DE LA TABLA BD
Base = declarative_base() # LOS MODELOS HEREDAN DE ESTA CLASE

# FUNCION QUE GESTIONA EL CICLO DE VIDA DE UNA SESION
def get_db(): # 25 usages 1 estelaV9
    db = SessionLocal() # CREAR UNA NUEVA SESION
    try:
        yield db # DEVUELVE LA SESION PARA SU USO
    finally:
        db.close() # CIERRA LA SESION DESPUES DE USARLA
```

6. **app/routers/:** Aquí se definen los modulos con las diferentes rutas o endpoints de la API. Cada archivo de esta carpeta contiene un router específico que gestiona un conjunto de rutas relacionadas, como la autenticación de usuarios, la gestión de productos, etc.

- **Auth:**

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from app.db.database import get_db
from app.models.user import User
from app.security.security import verify_password, create_access_token, verify_token

router = APIRouter(
    prefix="/auth",
    tags=["Authentication"]
)

# CONFIGURACION DE OAUTH2 PARA GESTIONAR TOKENS JWT
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/login") # RUTA PARA EL LOGIN

# FUNCION PARA OBTENER EL USUARIO AUTENTICADO A PARTIR DEL TOKEN
def get_current_user(token: str = Depends(oauth2_scheme), db: Session = Depends(get_db)):
    token_data = verify_token(token)
    if token_data is None:
        raise HTTPException(status_code=401, detail="Token inválido o expirado")

    user_id = token_data.get("sub")
    if user_id is None:
        raise HTTPException(status_code=401, detail="Token inválido")

    user = db.query(User).filter(User.id == user_id).first()
    if user is None:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")
    return user # RETORNA EL USUARIO SI TODO HA SALIDO BIEN
```

```
# INICIAR SESION Y OBTENER EL TOKEN JWT
@router.post(path="/login", summary="Iniciar sesion para obtener el token JWT") # estelaV9
def login(form_data: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get_db)):
    # BUSCAR EL USUARIO POR NOMBRE DE USUARIO
    user = db.query(User).filter(User.username == form_data.username).first()

    # SI NO SE ENCUENTRA, SE LANZA UNA EXCEPCION
    if not user:
        raise HTTPException(status_code=400, detail="Credenciales incorrectas")

    # VERIFICAR LA CONTRASEÑA CON LA ALMACENADA EN LA BD
    # (La función 'verify_password' sirve para comparar la contraseña)
    if not verify_password(form_data.password, user.password):
        raise HTTPException(status_code=400, detail="Credenciales incorrectas")

    # SI LAS CREDENCIALES SON VALIDAS, SE CREA EL TOKEN JWT PARA AUTENTICAR AL USUARIO
    # CON EL ID DEL USUARIO COMO 'sub' (SUBJECT)
    access_token = create_access_token(data={"sub": user.id})

    # SE RETORNA EL TOKEN JWT GENERADO JUNTO CON EL TIPO DE TOKEN
    return {"access_token": access_token, "token_type": "bearer"}
```

○ User:

```
from fastapi import APIRouter, Depends, HTTPException
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session
from typing import List
from app.crud.user_crud import update_user, get_user, delete_user
from app.db.database import get_db
from app.models.user import User
from app.routers.auth import get_current_user
from app.schemas.user import UserResponse, UpdateUser, UserCreate
from app.security import verify_password, hash_password, create_access_token

router = APIRouter(
    prefix="/users",
    tags=["Users"],
    # PROTEGER LAS RUTAS DE ESTE ROUTER
)

# OBTENER TODOS LOS USUARIOS
@router.get("/", response_model=List[UserResponse], dependencies=[Depends(get_current_user)])
def obtener_usuarios(db: Session = Depends(get_db)): # LA RUTA RECIBE LA BD
    data = db.query(User).all()
    return data

# SE PONE , dependencies=[] PARA NO TENER QUE AUTENTICARSE
@router.post("/login", summary="Iniciar sesión para obtener el token JWT", dependencies=[])
def login(form_data: OAuth2PasswordRequestForm = Depends(), db: Session = Depends(get_db)):
    user = db.query(User).filter(User.username == form_data.username).first()
    if not user or not verify_password(form_data.password, user.password): # Verifica la contraseña
        raise HTTPException(status_code=400, detail="Credenciales incorrectas")
    access_token = create_access_token(data={"sub": user.id})
    return {"access_token": access_token, "token_type": "bearer"}
```

```
@router.post("/register", summary="Registrar un nuevo usuario")
def register(user_data: UserCreate, db: Session = Depends(get_db)):
    try:
        # VERIFICA SI EL NOMBRE DE USUARIO YA EXISTE
        existing_user = db.query(User).filter(User.username == user_data.username).first()
        if existing_user:
            raise HTTPException(status_code=400, detail="El nombre de usuario ya está en uso")

        # HASH DE LA CONTRASEÑA ANTES DE ALMACENARLA
        hashed_password = hash_password(user_data.password)

        # CREAR NUEVO USUARIO
        new_user = User(username=user_data.username, email=user_data.email, password=hashed_password)
        db.add(new_user)
        db.commit()
        db.refresh(new_user)

        # DEVOLVER MENSAJE DE ÉXITO
        return {"message": "Usuario creado correctamente", "user_id": new_user.id}

    except Exception as e:
        # SI FALLA, SE HACE UN ROLLBACK
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al crear el usuario: {str(e)}")

# CREAR UN NUEVO USUARIO
@router.post("/", summary="Crear un nuevo usuario", response_model=UserResponse, dependencies=[Depends(get_current_user)])
def create_new_user(user: UserCreate, db: Session = Depends(get_db)):
    try:
        db_user = User(username=user.username, email=user.email, password=user.password)
        db.add(db_user)
        db.commit()
        db.refresh(db_user)
        return db_user
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al crear el usuario: {str(e)}")
```

```
# OBTENER UN USUARIO POR ID
@router.get("/{user_id}", response_model=UserResponse, summary="Obtener un usuario por ID", dependencies=[Depends(get_current_user)])
def get_user_by_id(user_id: int, db: Session = Depends(get_db)):
    user = get_user(db, user_id)
    if user is None:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")
    return user

# ACTUALIZAR USUARIO
@router.put("/{user_id}", response_model=UserResponse, summary="Actualizar un usuario", dependencies=[Depends(get_current_user)])
def update_user_info(user_id: int, user: UpdateUser, db: Session = Depends(get_db)):
    db_user = update_user(db, user_id, user)
    if db_user:
        return db_user
    raise HTTPException(status_code=404, detail="Usuario no encontrado")

# ELIMINAR UN USUARIO
@router.delete("/{user_id}", summary="Eliminar un usuario", response_model=UserResponse, dependencies=[Depends(get_current_user)])
def delete_user_info(user_id: int, db: Session = Depends(get_db)):
    user = get_user(db, user_id)
    if user is None:
        raise HTTPException(status_code=404, detail="Usuario no encontrado")

    try:
        deleted_user = delete_user(db, user_id)
        return deleted_user
    except Exception as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al eliminar el usuario: {str(e)}")
```

○ Cube:

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.db.database import get_db
from app.schemas.cube import CubeCreate, CubeResponse
from app.crud.cube_crud import create_cube, get_cube, get_cubes, update_cube, delete_cube
from app.routers.auth import get_current_user

router = APIRouter(
    prefix="/cubes",
    tags=["Cubes"],
    # PROTEGER LAS RUTAS DE ESTE ROUTER
    dependencies=[Depends(get_current_user)]
)

# OBTENER TODOS LOS CUBOS
@router.get("/", summary="Obtener todos los cubos", response_model=list[CubeResponse])
def get_all_cubes(db: Session = Depends(get_db)):
    cubes = get_cubes(db)
    return cubes

# OBTENER UN CUBO POR ID
@router.get("/{cube_id}", summary="Obtener un cubo por ID", response_model=CubeResponse)
def get_cube_by_id(cube_id: int, db: Session = Depends(get_db)):
    cube = get_cube(db, cube_id)
    if cube is None:
        raise HTTPException(status_code=404, detail="Cubo no encontrado")
    return cube

# CREAR UN NUEVO CUBO
@router.post("/", summary="Crear un nuevo cubo", response_model=CubeResponse)
def create_new_cube(cube: CubeCreate, db: Session = Depends(get_db)):
    new_cube = create_cube(db, cube)
    return new_cube

# ACTUALIZAR UN CUBO
@router.put("/{cube_id}", summary="Actualizar un cubo", response_model=CubeResponse)
def update_cube_info(cube_id: int, cube: CubeCreate, db: Session = Depends(get_db)):
    updated_cube = update_cube(db, cube_id, cube)
    if updated_cube:
        return updated_cube
    raise HTTPException(status_code=404, detail="Cubo no encontrado")
```

```
# ELIMINAR UN CUBO
@router.delete("/{cube_id}", summary="Eliminar un cubo", response_model=CubeResponse) # estelaV9
def delete_cube_info(cube_id: int, db: Session = Depends(get_db)):
    # SE OBTIENE EL CUBO POR SU ID
    cube = get_cube(db, cube_id)
    if cube is None:
        raise HTTPException(status_code=404, detail="Cubo no encontrado")

    try:
        # SE ELIMINA EL CUBO Y SE DEVUELVE EL CUBO ELIMINADO
        deleted_cube = delete_cube(db, cube_id)
        return deleted_cube
    except Exception as e:
        # SI HAY UN ERROR AL ELIMINAR, SE LANZA UNA EXCEPCION
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al eliminar el cubo: {str(e)}")
```

○ SolveTime:

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.db.database import get_db
from app.routers.auth import get_current_user
from app.schemas.solve_time import SolveTimeCreate, SolveTimeResponse
from app.crud.solve_time_crud import create_solve_time, get_solve_time, get_solve_times, update_solve_time, \
    delete_solve_time

router = APIRouter(
    prefix="/solve-times",
    tags=["SolveTimes"],
    # PROTEGER LAS RUTAS DE ESTE ROUTER
    dependencies=[Depends(get_current_user)]
)

# OBTENER TODOS LOS TIEMPOS
@router.get("/", summary="Obtener todos los tiempos", response_model=list[SolveTimeResponse]) # estelaV9
def get_all_solve_times(db: Session = Depends(get_db)):
    solve_times = get_solve_times(db)
    return solve_times

# OBTENER UN TIEMPO POR ID
@router.get("/{solve_time_id}", summary="Obtener un tiempo por ID", response_model=SolveTimeResponse) # estelaV9
def get_solve_time_by_id(solve_time_id: int, db: Session = Depends(get_db)):
    solve_time = get_solve_time(db, solve_time_id)
    if solve_time is None:
        raise HTTPException(status_code=404, detail="Tiempo no encontrado")
    return solve_time

# CREAR UN NUEVO TIEMPO
@router.post("/", summary="Crear un nuevo tiempo", response_model=SolveTimeResponse) # estelaV9
def create_new_solve_time(solve_time: SolveTimeCreate, db: Session = Depends(get_db)):
    new_solve_time = create_solve_time(db, solve_time)
    return new_solve_time

# ACTUALIZAR UN TIEMPO
@router.put("/{solve_time_id}", summary="Actualizar un tiempo", response_model=SolveTimeResponse) # estelaV9
def update_solve_time_info(solve_time_id: int, solve_time: SolveTimeCreate, db: Session = Depends(get_db)):
    updated_solve_time = update_solve_time(db, solve_time_id, solve_time)
    if updated_solve_time:
        return updated_solve_time
    raise HTTPException(status_code=404, detail="Tiempo no encontrado")
```

```
# ELIMINAR UN TIEMPO
@router.delete("/{solve_time_id}", summary="Eliminar un tiempo", response_model=SolveTimeResponse)
def delete_solve_time_info(solve_time_id: int, db: Session = Depends(get_db)):
    # SE OBTIENE EL TIEMPO POR SU ID
    solve_time = get_solve_time(db, solve_time_id)
    if solve_time is None:
        raise HTTPException(status_code=404, detail="Tiempo no encontrado")

    try:
        # SE ELIMINA EL TIEMPO Y LO DEVUELVE
        deleted_solve_time = delete_solve_time(db, solve_time_id)
        return deleted_solve_time
    except Exception as e:
        # SI HAY UN ERROR AL ELIMINAR, SE LANZA UNA EXCEPCION
        db.rollback()
        raise HTTPException(status_code=500, detail=f"Error al eliminar el tiempo: {str(e)}")
```


7. **app/test/**: Aquí está la única prueba unitaria de la aplicación valida el login.

```
import pytest
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from fastapi.testclient import TestClient
from app.db.database import Base, get_db
from app.main import app
from app.models.user import User
from app.security.security import hash_password

# BASE DE DATOS EN MEMORIA PARA LAS 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)

# FIXTURE PARA LA BASE DE DATOS (se crea una sola vez para la sesión de test)
@pytest.fixture(scope="session")
def db():
    # CREAR LA BASE DE DATOS DE PRUEBA
    Base.metadata.create_all(bind=engine)
    db = TestingSessionLocal()
    try:
        yield db
    finally:
        db.close()

# FIXTURE PARA EL CLIENTE DE PRUEBAS
@pytest.fixture(scope="module")
def client(db):
    def override_get_db():
        try:
            yield db
        finally:
            # SE CIERRA LA SESIÓN
            pass
    app.dependency_overrides[get_db] = override_get_db
    client = TestClient(app)
    yield client
```

```
# FUNCION AUXILIAR PARA CREAR UN USUARIO DE PRUEBA
def create_test_user(db):
    test_user = User(
        username="testuser",
        email="test@example.com",
        password=hash_password("testpassword")
    )
    db.add(test_user)
    db.commit()
    db.refresh(test_user)
    return test_user

# PRUEBA: LOGIN EXITOSO
def test_login_success(client, db):
    # SE CREA UN USUARIO DE PRUEBA
    create_test_user(db)
    # DATOS DEL LOGIN
    login_data = {"username": "testuser", "password": "testpassword"}
    response = client.post("/auth/login", data=login_data)
    # SE ESPERA UN CODIGO DE 200 Y QUE INCLUYA EL access_token EN LA RESPUESTA
    assert "access_token" in response.json()
```

8. **app/schemas**: los **schemas** son clases que definen cómo deben estructurarse los datos que se envían y reciben en las solicitudes de la API. En FastAPI, los schemas se definen utilizando Pydantic, lo que permite validar y transformar los datos de manera sencilla. Los schemas utilizados en la aplicación:

- Cube Model: representan los cubos en la base de datos y para las operaciones CRUD (crear, leer, actualizar) de estos cubos.
 1. **CubeCreate**: Este schema se utiliza para crear un nuevo cubo.
 2. **CubeResponse**: para devolver información sobre un cubo.
 3. **UpdateCube** para actualizar la información de un cubo. Los campos son opcionales, lo que significa que no es necesario actualizar todos los campos.

```
from pydantic import BaseModel, EmailStr
from typing import Optional

# DEFINIR pydantic PARA TRABAJAR CON LOS DATOS
""" ***** CUBE MODEL ***** """
class CubeCreate(BaseModel): 6 usages  ▲ estelaV9
    type: str
    owner_id: int

# MODELO CubeResponse PARA DEVOLVER DATOS DE UN CUBO
class CubeResponse(BaseModel): 6 usages  ▲ estelaV9
    id: int
    type: str
    owner_id: int

# CUBE MODEL PARA UPDATE
class UpdateCube(BaseModel): # Schema  ▲ estelaV9
    # SE PONE OPCIONAL PARA QUE NO SEA OBLIGATORIO ACTUALIZARTODO
    id: Optional[str] = None
    type: Optional[str] = None
    owner_id: Optional[EmailStr] = None
```

- Solve Time: Estos schemas están relacionados con la creación y respuesta de los tiempos de resolución de cubos.
 1. **SolveTimeCreate**: Este schema se utiliza para crear un tiempo de resolución para un cubo específico.
 2. **SolveTimeResponse**: Este schema se utiliza para devolver información sobre un tiempo de resolución.

```
from pydantic import BaseModel
from datetime import datetime

# DEFINIR pydantic PARA TRABAJAR CON LOS DATOS
""" ***** SOLVE TIME MODEL ***** """
class SolveTimeCreate(BaseModel): 6 usages  ▲ estelaV9
    time_seconds: float
    cube_id: int
    user_id: int

# MODELO SolveTimeResponse PARA DEVOLVER DATOS DE UN TIEMPO
class SolveTimeResponse(BaseModel): 6 usages  ▲ estelaV9
    id: int
    time_seconds: float
    date: datetime
    cube_id: int
    user_id: int
```

- User: schemas relacionados con la gestión de usuarios en la aplicación.
 1. **UserCreate**: Este schema se utiliza para la creación de un nuevo usuario.
 2. **UserResponse**: Este schema se utiliza para devolver información sobre un usuario.
 3. **UpdateUser**: Este schema se utiliza para actualizar los datos de un usuario. Todos los campos son opcionales, lo que permite actualizar solo los campos que se deseen modificar.

```
from pydantic import BaseModel, EmailStr
from typing import Optional

# DEFINIR pydantic PARA TRABAJAR CON LOS DATOS
""" ***** USER MODEL ***** """

class UserCreate(BaseModel): # Schema 5 usages 1 estelaV9
    username: str
    # CORREO VALIDADO COMO DIRECCION DE CORREO
    email: EmailStr
    password: str

# MODELO UserResponse PARA DEVOLVER DATOS DE UN USUARIO
class UserResponse(BaseModel): # 6 usages 1 estelaV9
    id: int
    username: str
    email: EmailStr

# USER MODEL PARA UPDATE
class UpdateUser(BaseModel): # Schema 4 usages 1 estelaV9
    # SE PONE OPCIONAL PARA QUE NO SEA OBLIGATORIO ACTUALIZAR TODO
    username: Optional[str] = None
    password: Optional[str] = None
    email: Optional[EmailStr] = None
```

Principales Librerías Utilizadas

- **FastAPI:** Para la creación de la API.
- **SQLAlchemy:** ORM para la interacción con la base de datos.
- **passlib:** Para el hasheo seguro de contraseñas.
- **python-jose:** Para la creación y verificación de tokens JWT.
- **pytest:** Framework para la ejecución de pruebas unitarias.

Pruebas

En este proyecto se han implementado pruebas unitarias para asegurar que la API funcione correctamente y las rutas de la aplicación respondan como se espera. A continuación, se explica una de las pruebas implementadas para verificar la autenticación mediante el endpoint de inicio de sesión.

Prueba 1: Prueba de la API (Login Exitoso)

Esta prueba verifica que el endpoint de inicio de sesión (/auth/login) funcione correctamente. Se envían las credenciales de un usuario y, si son válidas, la API debe devolver un **token JWT** para la autenticación en futuras solicitudes.

Descripción:

- Se crea un usuario de prueba en la base de datos utilizando un fixture para la base de datos en memoria.
- Se envían las credenciales del usuario (nombre de usuario y contraseña) al endpoint /auth/login mediante una solicitud POST.
- Si las credenciales son correctas, la API debe devolver un código de estado 200 OK y un **access_token** en el cuerpo de la respuesta.

```
import pytest
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from fastapi.testclient import TestClient
from app.db.database import Base, get_db
from app.main import app
from app.models.user import User
from app.security.security import hash_password

# BASE DE DATOS EN MEMORIA PARA LAS 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)

# FIXTURE PARA LA BASE DE DATOS (se crea una sola vez para la sesión de test)
@pytest.fixture(scope="session") new *
def db():
    # CREAR LA BASE DE DATOS DE PRUEBA
    Base.metadata.create_all(bind=engine)
    db = TestingSessionLocal()
    try:
        yield db
    finally:
        db.close()

# FIXTURE PARA EL CLIENTE DE PRUEBAS
@pytest.fixture(scope="module") new *
def client(db):
    def override_get_db(): new *
        try:
            yield db
        finally:
            # SE CIERRA LA SESIONE
            pass
    app.dependency_overrides[get_db] = override_get_db
    client = TestClient(app)
    yield client
```

```
# FUNCION AUXILIAR PARA CREAR UN USUARIO DE PRUEBA
def create_test_user(db): 1 usage new *
    test_user = User(
        username="testuser",
        email="test@example.com",
        password=hash_password("testpassword")
    )
    db.add(test_user)
    db.commit()
    db.refresh(test_user)
    return test_user

# PRUEBA: LOGIN EXITOSO
def test_login_success(client, db): new *
    # SE CREA UN USUARIO DE PRUEBA
    create_test_user(db)
    # DATOS DEL LOGIN
    login_data = {"username": "testuser", "password": "testpassword"}
    response = client.post( url= "/auth/login", data=login_data)
    # SE ESPERA UN CODIGO DE 200 Y QUE INCLUYA EL access_token NE LA REPSUNTA
    assert "access_token" in response.json()
```

En el código realizado se configura un entorno de base de datos en memoria utilizando SQLite para las pruebas. fixture db crea la base de datos en memoria y la mantiene durante la ejecución de la prueba.

Se crea un usuario de prueba en la base de datos utilizando la función `create_test_user()`. En la prueba de login, se realiza una solicitud POST a la ruta `/auth/login` enviando las credenciales de usuario predefinidas (nombre de usuario y contraseña). Se valida que la respuesta tenga un código de estado 200 OK y que contenga un campo `access_token` que es el token JWT generado para el usuario.

```
(venv) PS C:\Users\admin\Documents\SistemasGestionEmpresarial\TrabajoFinal\rubik_api> pytest
===== test session starts =====
platform win32 -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: C:\Users\admin\Documents\SistemasGestionEmpresarial\TrabajoFinal\rubik_api
plugins: anyio-4.8.0
collected 1 item

app\test\test_protected.py . [100%]

===== 1 passed in 3.74s =====
```

Despliegue de la aplicación

El despliegue de la aplicación se realiza en un entorno de desarrollo local utilizando el servidor **Uvicorn** para ejecutar la aplicación FastAPI. El servidor se inicia utilizando el siguiente comando: `uvicorn app.main:app --reload`

Servidor Local:

Durante el desarrollo, la aplicación se ejecuta en el entorno local utilizando **Uvicorn**. Gracias a la opción `--reload` habilita la recarga automática de la aplicación cada vez que se realizan cambios en el código.

Contenedores Docker:

La aplicación está preparada para ser desplegada en contenedores Docker ya que permite crear un entorno aislado para la aplicación y sus dependencias. El uso de **docker-compose** permite gestionar los contenedor de la base de datos (PostgreSQL) y el contenedor de la aplicación, de manera automatizada simplificando así la configuración y el arranque.

Acceder a la aplicación:

Después de que el contenedor esté en ejecución, la aplicación estará disponible en **`http://localhost:8000`**, donde se puede interactuar con la API.

Manuales

Manual de usuario

Una vez que se haya ejecutado Docker y la aplicación de FastAPI, podemos acceder a la interfaz de documentación de la API a través de la siguiente URL: <http://127.0.0.1:8000/docs/>.

Al ingresar a esta URL, se accederá a una vista interactiva proporcionada por **FastAPI**. Esta interfaz permitirá probar todos los endpoints de la API. A continuación, se detallan las principales funcionalidades y cómo interactuar con ellas.

Rubik API 1.0.0 OpenAPI 3.1

openapi.json

API para gestionar usuarios, cubos de Rubik y tiempos de resolución

Authorize 

Authentication

POST /auth/login Iniciar sesión para obtener el token JWT

Users

GET /users/ Obtener Usuarios

POST /users/ Crear un nuevo usuario

POST /users/login Iniciar sesión para obtener el token JWT

POST /users/register Registrar un nuevo usuario

GET /users/{user_id} Obtener un usuario por ID

PUT /users/{user_id} Actualizar un usuario

DELETE /users/{user_id} Eliminar un usuario

Cubes

GET /cubes/ Obtener todos los cubos

POST /cubes/ Crear un nuevo cubo

GET /cubes/{cube_id} Obtener un cubo por ID

PUT /cubes/{cube_id} Actualizar un cubo

DELETE /cubes/{cube_id} Eliminar un cubo

SolveTimes

GET /solve-times/ Obtener todos los tiempos

POST /solve-times/ Crear un nuevo tiempo

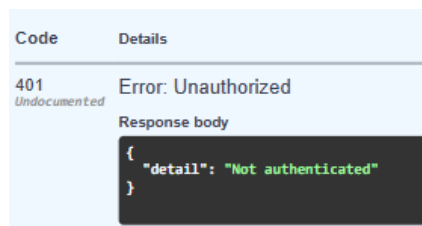
GET /solve-times/{solve_time_id} Obtener un tiempo por ID

PUT /solve-times/{solve_time_id} Actualizar un tiempo

DELETE /solve-times/{solve_time_id} Eliminar un tiempo

Autorización y Endpoints no protegidos

En la interfaz de **FastAPI**, se notara que algunos de los endpoints tienen un ícono de candado al lado de su nombre. Esto indica que estas rutas requieren autorización para ser ejecutadas. Si alguien intenta acceder a ellas sin estar autenticado, recibirás un error **401 (Unauthorized)**.



Para poder interactuar con estos endpoints protegidos, necesitas autenticarte utilizando un token JWT. Para obtener este token, sigue los siguientes pasos:

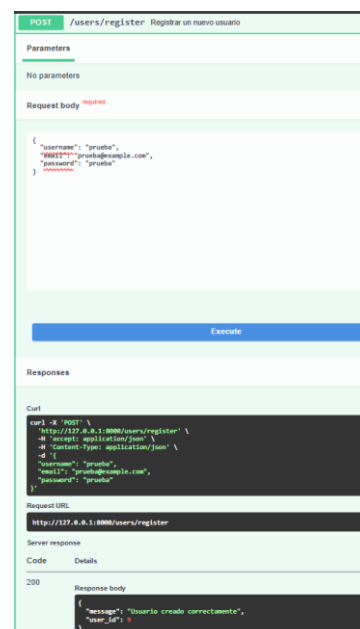
1. Crear un Usuario

Para empezar, es necesario crear un usuario. En la interfaz de **FastAPI**, podrás hacerlo utilizando el endpoint `/users/register`. Al enviar los datos requeridos (nombre de usuario, correo electrónico, contraseña), se creará un usuario en el sistema.

Ejemplo:

- **Username:** usuario_prueba
- **Email:** correo@ejemplo.com
- **Password:** contraseña

Esto te permitirá generar un usuario válido para poder autenticarte posteriormente.



2. Iniciar Sesión

Una vez que tienes tu usuario creado, se puede iniciar sesión en la API para obtener un **token JWT** que te permitirá autenticarte en los endpoints protegidos.

- Dirígete al endpoint **/auth/login**.
- Introduce tus credenciales (nombre de usuario y contraseña).
- Si las credenciales son correctas, recibirás una respuesta con el **access_token**. Este token será necesario para autorizar tus futuras solicitudes.

Authentication

POST /auth/login

Iniciar sesión para obtener el token JWT

Parameters

No parameters

Request body required

grant_type
string
pattern: "password"
☐ Send empty value

username * required
string
prueba

password * required
string
prueba

scope
string
☒ Send empty value

client_id
string
☐ Send empty value

client_secret
string
☐ Send empty value

Execute

Responses

Curl

```
curl -X "POST" \
"http://127.0.0.1:8080/auth/login" \
-H "accept: application/json" \
-d '{"Content-Type": "application/x-www-form-urlencoded"}' \
-d 'grant_type=password&username=prueba&password=prueba&scope=client_id:string&client_secret:string'
```

Request URL

```
http://127.0.0.1:8080/auth/login
```

Server response

Code

Details

200

Response body

{
 "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IWRkcCIsImVudCI6IjYyMDQwMmMuF3BpeWlkeS1kZDg5fmlnZWdldm9ucy9kaWEiLCJ0eXAiOiJKd1kiLCJhdwQiOiJlc3RhbnQifQ==",
 "token_type": "bearer"
}

Este es el mismo proceso que en el authorize y este te autoriza los demás endpoints.

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: /auth/login
Flow: password
username:
password:
Client credentials location:

Authorization header

client_id:
client_secret:

Authorize

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)

Authorized
Token URL: /auth/login
Flow: password
username: prueba
password: *****
Client credentials location: basic
client_id: *****
client_secret: *****

Logout

Close

Endpoints Protegidos

Usuario

Obtener usuarios

Users

GET /users/ Obtener Usuarios

Parameters Cancel

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
'http://127.0.0.1:8000/users/' \
-H 'accept: application/json' \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ3ZmI0I3IiwiaXhwIjo4NzR5MjIyOTQyYyQ.ODQMeTx8ZyPYR1MmPgA5otn0YxXngYxwZFXZrFajV8'
```

Request URL

```
http://127.0.0.1:8000/users/
```

Server response

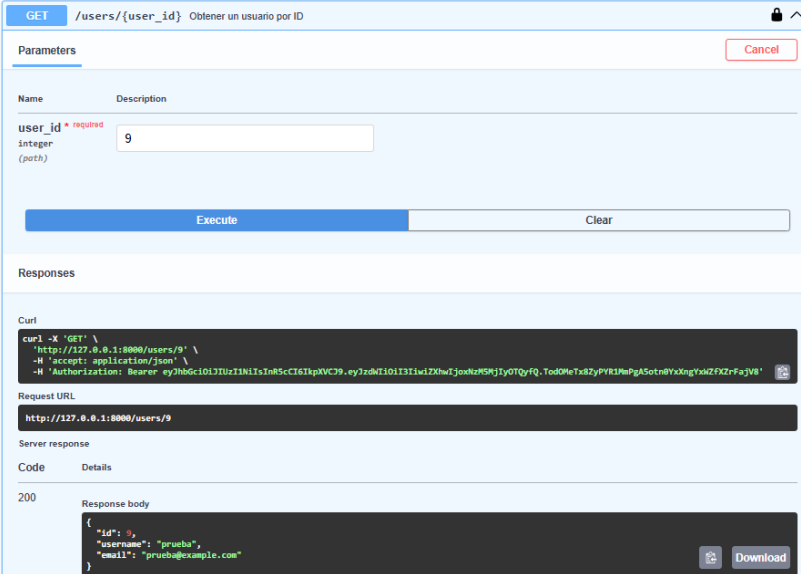
Code	Details
200	<div>Response body<pre>[{ "id": 1, "username": "d", "email": "d@d.com" }, { "id": 7, "username": "string", "email": "user@example.com" }, { "id": 8, "username": "z", "email": "z@example.com" }, { "id": 9, "username": "prueba", "email": "prueba@example.com" }, { "id": 10, "username": "s", "email": "s@example.com" }]</pre>Download</div>

Es lo mismo que el método registrar solo que con autorización.

pág. 35

Obtener un usuario por ID

Indicamos el id y retornará el usuario.

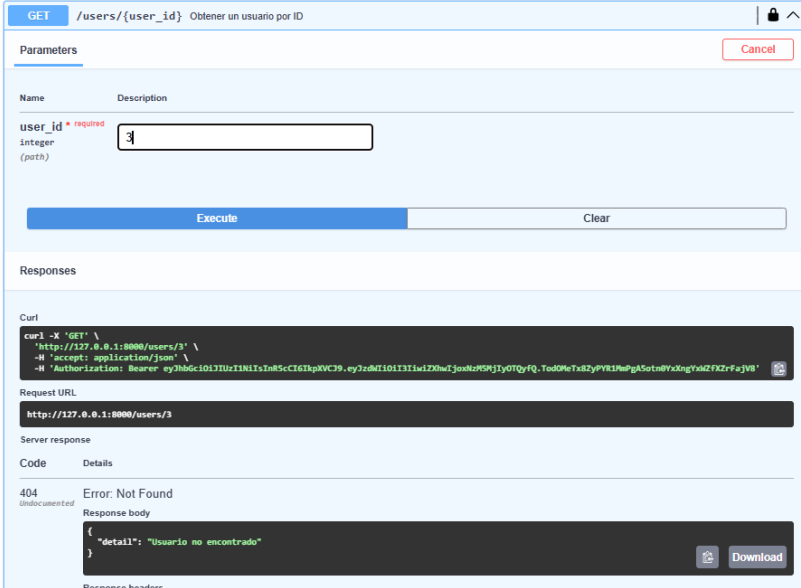


The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** /users/{user_id} Obten un usuario por ID
- Parameters:** user_id (required, integer, path) with value 9.
- Execute:** Button to execute the request.
- Responses:** Section showing the response details.
 - Code:** 200
 - Response body:**

```
{
  "id": 9,
  "username": "prueba",
  "email": "prueba@example.com"
}
```
 - Download:** Button to download the response body.

Si no existe mandara un mensaje



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

- Method:** GET
- URL:** /users/{user_id} Obten un usuario por ID
- Parameters:** user_id (required, integer, path) with value 3.
- Execute:** Button to execute the request.
- Responses:** Section showing the response details.
 - Code:** 404
 - Message:** Error: Not Found
 - Response body:**

```
{
  "detail": "Usuario no encontrado"
}
```
 - Download:** Button to download the response body.

Ponemos el id y el request body

Eliminar un usuario

Ponemos el id y se eliminará.

pág. 37

Cubes

Tiene el mismo mecanismo que user.

Obtener cubos

Cubes

GET /cubes/ Obtener todos los cubos

Parameters

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/cubes/' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1IiwiaWF0Ij0iY0YyQy50dG9keS2yPVR1bWpASotnVxngYmZlK2ZfZjV8'
```

Request URL

http://127.0.0.1:8000/cubes/

Server response

Code Details

200

Response body

```
{
  "id": 1,
  "type": "string",
  "owner_id": 1
}
```

Response headers

```
content-length: 39
content-type: application/json
date: Mon, 18 Feb 2025 20:39:28 GMT
server: uvicorn
```

Responses

Crear un nuevo cubo

POST /cubes/ Crear un nuevo cubo

Parameters

No parameters

Request body *required*

application/json

```
{
  "type": "nuevo",
  "owner_id": 8
}
```

Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/cubes/' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1IiwiaWF0Ij0iY0YyQy50dG9keS2yPVR1bWpASotnVxngYmZlK2ZfZjV8' \
  -H 'Content-Type: application/json' \
  -d '{
    "type": "nuevo",
    "owner_id": 8
  }'
```

Request URL

http://127.0.0.1:8000/cubes/

Server response

Code Details

200

Response body

```
{
  "id": 2,
  "type": "nuevo",
  "owner_id": 8
}
```

Response headers

Obtener un cubo por ID

GET /cubes/{cube_id} Obtener un cubo por ID

Parameters

Cancel

Name	Description
cube_id * required integer (path)	2

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/cubes/2' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1IiwiaWF0IjoiYjQyYzYyPVR1MmRlbnR5X2FjYV8'
```

Request URL

http://127.0.0.1:8000/cubes/2

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 2, "type": "nuevo", "owner_id": 1 }</pre> <p>Download</p>

Response headers

Actualizar un cubo

PUT /cubes/{cube_id} Actualizar un cubo

Cancel Reset

Name	Description
cube_id * required integer (path)	2

Request body * required

application/json

```
{
  "type": "nuevoActualizado",
  "owner_id": 8
}
```

Execute Clear

Responses

Curl

```
curl -X 'PUT' \
  'http://127.0.0.1:8000/cubes/2' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1IiwiaWF0IjoiYjQyYzYyPVR1MmRlbnR5X2FjYV8' \
  -H 'Content-Type: application/json' \
  -d '{
    "type": "nuevoActualizado",
    "owner_id": 8
  }'
```

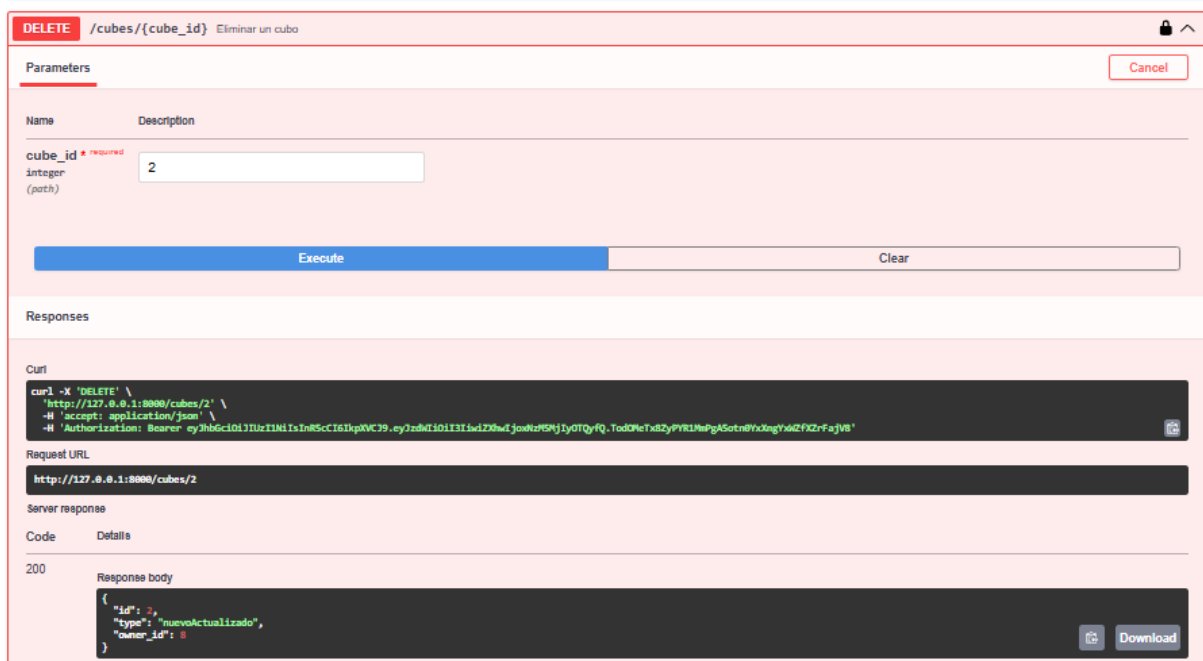
Request URL

http://127.0.0.1:8000/cubes/2

Server response

Code	Details
200	<p>Response body</p> <pre>{ "id": 2, "type": "nuevoActualizado", "owner_id": 8 }</pre> <p>Download</p>

Eliminar un cubo



DELETE /cubes/{cube_id} Eliminar un cubo

Parameters

Name	Description
cube_id <small>* required</small>	
integer (path)	2

Execute Clear

Responses

Curl

```
curl -X 'DELETE' \
  'http://127.0.0.1:8000/cubes/2' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IWRXVjQ9LWVudCI6IjE1IiwiaWF0Ij0iOTQyYQ.TodQMeTx82yPVRlMwPgASotn9VxXngYndZFXZrFajV8'
```

Request URL

```
http://127.0.0.1:8000/cubes/2
```

Server response

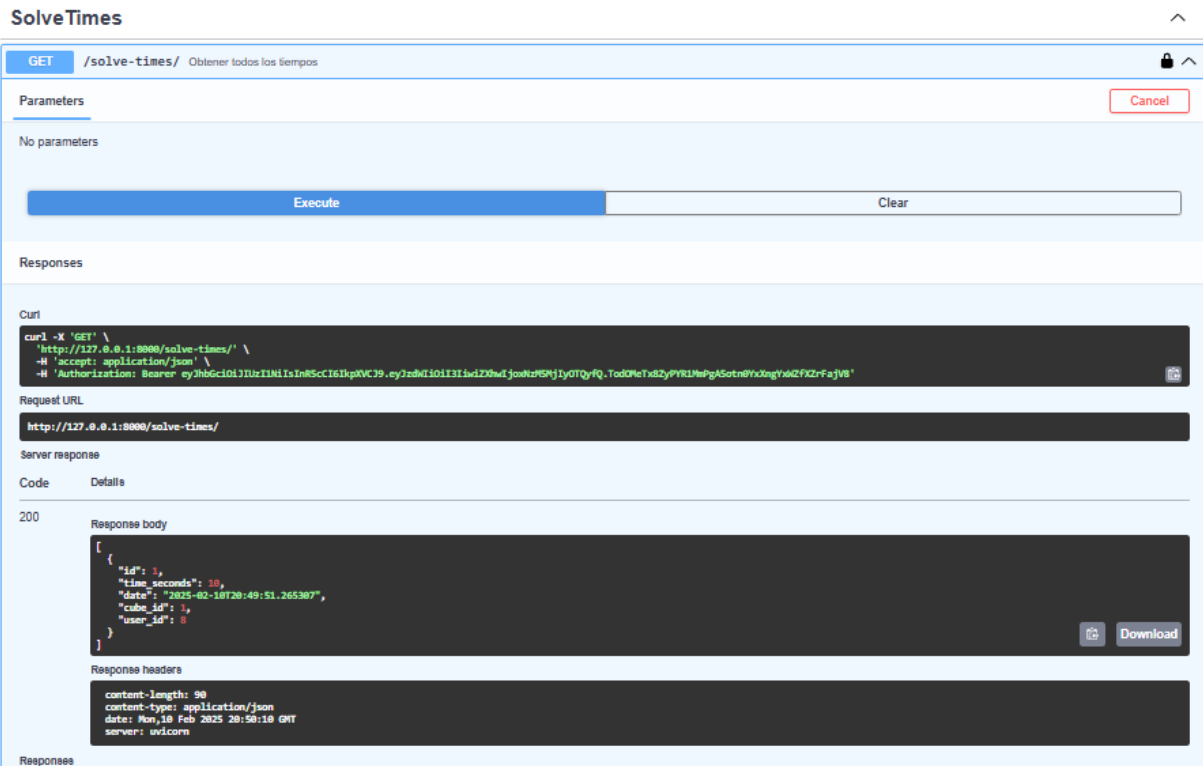
Code	Details
200	<p>Response body</p> <pre>{ "id": 2, "type": "nuevoActualizado", "owner_id": 0 }</pre>

Download

SolveTimes

Tiene el mismo mecanismo que user

Obtener tiempos



GET /solve-times/ Obtener todos los tiempos

Parameters

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/solve-times/' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IWRXVjQ9LWVudCI6IjE1IiwiaWF0Ij0iOTQyYQ.TodQMeTx82yPVRlMwPgASotn9VxXngYndZFXZrFajV8'
```

Request URL

```
http://127.0.0.1:8000/solve-times/
```

Server response

Code	Details
200	<p>Response body</p> <pre>[{ "id": 1, "time_seconds": 10, "date": "2025-02-10T20:49:51.265307", "cube_id": 1, "user_id": 0 }]</pre> <p>Response headers</p> <pre>content-length: 90 content-type: application/json date: Mon, 10 Feb 2025 20:50:10 GMT server: uvicorn</pre>

Download

Crear un nuevo tiempo

POST /solve-times/ Crear un nuevo tiempo

Cancel

Reset

Parameters

No parameters

Request body required

application/json

```
{
  "time_seconds": 10,
  "cube_id": 1,
  "user_id": 0
}
```

Execute

Clear

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/solve-times/' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1IiwiaWF0IjoiYjY0OTQyYQ.TodQMeTxd2pVRLMhPgASotn0VxXngYndZfXZrFajV8' \
  -d '{
    "time_seconds": 10,
    "cube_id": 1,
    "user_id": 0
  }'
```

Request URL

http://127.0.0.1:8000/solve-times/

Server response

Code

Details

200

Response body

```
{
  "id": 1,
  "time_seconds": 10,
  "date": "2025-02-10T20:49:51.265307",
  "cube_id": 1,
  "user_id": 0
}
```

Download

Response headers

Obtener un tiempo por ID

GET /solve-times/{solve_time_id} Obtener un tiempo por ID

Cancel

Parameters

Name

Description

solve_time_id required

integer

(path)

1

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/solve-times/1' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1IiwiaWF0IjoiYjY0OTQyYQ.TodQMeTxd2pVRLMhPgASotn0VxXngYndZfXZrFajV8'
```

Request URL

http://127.0.0.1:8000/solve-times/1

Server response

Code

Details

200

Response body

```
{
  "id": 1,
  "time_seconds": 10,
  "date": "2025-02-10T20:49:51.265307",
  "cube_id": 1,
  "user_id": 0
}
```

Download

Response headers

Actualizar un tiempo

PUT

/solve-times/{solve_time_id}

Actualizar un tiempo

Parameters

Cancel

Reset

Name	Description
solve_time_id * required integer <small>(path)</small>	<input type="text" value="1"/>

Request body required

application/json

```
{  
  "time_seconds": 11,  
  "cube_id": 1,  
  "user_id": 8  
}
```

Execute

Clear

Responses

Curl

```
curl -X 'PUT' \  
  'http://127.0.0.1:8000/solve-times/1' \  
  -H 'accept: application/json' \  
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6ImlzbnVjYyQyOQqQ.TodDheTxdZyPvKlMwPgASotnPhx0ngFndZfajYB' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "time_seconds": 11,  
    "cube_id": 1,  
    "user_id": 8  
  }'
```

Request URL

http://127.0.0.1:8000/solve-times/1

Server response

Code

Details

200

Response body

```
{  
  "id": 1,  
  "time_seconds": 11,  
  "date": "2025-02-10T20:53:34.831522",  
  "cube_id": 1,  
  "user_id": 8  
}
```

Download

Eliminar un tiempo

DELETE

/solve-times/{solve_time_id} Eliminar un tiempo

Parameters

Name	Description
solve_time_id ▲ required	
integer (path)	<input type="text" value="1"/>

Execute

Clear

Responses

Curl

```
curl -X 'DELETE' \  
  'http://127.0.0.1:8000/solve-times/1' \  
  -H 'accept: application/json' \  
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIjOiIxMTI1IiwiaWF0IjoiOTQyYyQyODkxMjYwZmFja3V8'`
```

Request URL

```
http://127.0.0.1:8000/solve-times/1
```

Server response

Code	Details
200	<div><div>Response body</div><pre>{ "id": 1, "time_seconds": 11, "date": "2025-02-10T20:53:34.831522", "cube_id": 1, "user_id": 0 }</pre></div>

Manual de instalación

Pasos para desplegar la aplicación en un entorno local, utilizando Docker, PostgreSQL, y FastAPI y poder ejecutarla.

Requisitos previos

Para ejecutarlo, primero debes asegurarte de que tienes las siguientes herramientas:

- **Docker:** Para crear y ejecutar contenedores de forma sencilla.
- **Docker Compose:** Para 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.

Pasos:

1. Descargar el proyecto de [GitHub](#)
2. Ejecutar Docker.
3. Configurar PostgreSQL en pgAdmin: Para gestionar la base de datos PostgreSQL de manera visual debes hacer:
 1. **Abrir pgAdmin:** Abre la aplicación. Si estás usando Docker, asegúrate de que el contenedor de PostgreSQL esté funcionando y accesible en el puerto especificado.
 2. **Conectar a la base de datos:** En pgAdmin, conecta a tu servidor PostgreSQL. Con Docker, la URL de conexión puede ser algo como:
 - **Host:** localhost
 - **Puerto:** 5432 (puerto predeterminado de PostgreSQL)
 - **Contraseña:** La contraseña que tengas en el archivo docker-compose.yml.
 3. **Crear la base de datos:** Una vez conectado, creas la base de datos para la aplicación si no se ha creado automáticamente al levantar los contenedores. Esto se puede hacer desde la interfaz de pgAdmin:
 - Haz clic derecho en el servidor de PostgreSQL y selecciona "Create > Database".
 - Asigna un nombre a la base de datos, rubikapi-database.
4. **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 Uvicorn:** Abrirla el proyecto y en la terminal poner `uvicorn app.main:app --reload`
 - **Verificar que la API esté funcionando:** Accede a la API a través de tu navegador <http://localhost:8000/docs> donde se mostrará una interfaz para probar los endpoints de la API.
 - **5. Consultas en FastAPI:** Una vez que la API esté funcionando, se puede comenzar a hacer consultas a los endpoints definidos. Por ejemplo, cómo hacer una consulta a un endpoint protegido que requiere autenticación JWT:
 1. **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.

Si no está creado el usuario, también se puede crear ya que no está restringida.

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

Conclusiones y posibles ampliaciones

Dificultades encontradas en el desarrollo de la aplicación

Durante el desarrollo de esta aplicación, la principal dificultad que encontré fue el entorno de trabajo. Tuve varios problemas con **PyCharm** y la actualización de **FastAPI**. A pesar de realizar cambios en el código, FastAPI no se actualizaba automáticamente, lo que me obligaba a reiniciar el ordenador cada vez que realizaba alguna modificación. Esta situación me hizo perder mucho tiempo, ya que en un principio no me di cuenta de que el problema no era del código, sino del entorno de desarrollo, lo que me generó frustración sabiendo que estaba bien, afectando directamente al ritmo del proyecto.

Grado de satisfacción en el trabajo realizado

En cuanto al grado de satisfacción, me siento en general **bajo**. El tiempo para realizar el proyecto fue limitado, ya que todo el trabajo lo tenía que completar en dos días, lo que no fue suficiente para asimilar bien los conceptos y aplicarlos de forma efectiva. Además, los problemas con el entorno de desarrollo hicieron mayor la sensación de frustración y estrés.

Aunque el proyecto se completó en gran medida y logré implementar las funcionalidades básicas de la API, no puedo evitar sentir que no aproveché el tiempo de manera óptima, y que el trabajo no refleja todo el potencial que podría haber alcanzado si hubiera tenido más tiempo para profundizar en los detalles y corregir posibles errores.

A pesar de todo esto, me siento **algo contenta** de haber logrado sacar el proyecto adelante. Aunque no esté completamente satisfecha con el resultado final, el hecho de haber podido implementarlo y entender parte de lo que se necesitaba, me deja algo satisfecha :).

Aprendizaje durante el desarrollo

En cuanto al aprendizaje, uno de los aspectos más destacados fue **la implementación de autenticación** en Python. A pesar de las dificultades, pude adquirir conocimientos sobre cómo gestionar usuarios, crear rutas protegidas y asegurar la API mediante autenticación.

Posibles ampliaciones

Una de las posibles ampliaciones para esta aplicación es **integrar el backend con una aplicación móvil**. Durante el desarrollo del proyecto, me gustaría haber podido avanzar en esta parte, pero no tuve suficiente tiempo ni conocimiento para implementarlo correctamente, ya que en un inicio quería implementarlo en una aplicación desarrollada en Kotlin. La integración de la API con una app móvil permitiría a los usuarios interactuar mejor con esta gestión de cubos de Rubik.

Bibliografía

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

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

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

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

https://www.youtube.com/watch?v=meX4tEOGpac&ab_channel=Programaci%C3%B3nAndroidbyAristiDevs