



PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones
Multiplataforma
Informática y Comunicaciones

<Título del Proyecto>

Año: 2024/25

Fecha de presentación: 10/2/2024

Nombre y Apellidos: Leire Yagüe Fernández
Email: leire.yagfer.1@educa.jcyl.es

ÍNDICE

1	Introducción.....	4
2	Estado del arte	4
2.1	Definición de arquitectura de microservicios.....	4
2.2	Estructura de una API: protocolo utilizado, métodos, partes de las URL de una API.....	4
2.3	Formas de crear una API en python: FastAPI y Flask, indicando cuál se va a utilizar y por qué	5
3	Descripción general del proyecto	5
3.1	Objetivos	5
3.2	Entorno de trabajo (tecnologías de desarrollo y herramientas)	6
4	Documentación técnica: análisis, diseño, implementación, pruebas y despliegue	6
4.1	Análisis del sistema (funcionalidades básicas de la aplicación).....	6
4.2	Diseño de la base de datos	7
4.3	Implementación – Estructura del código.....	7
4.3.1	APP	8
4.3.1.1	DB.....	8
4.3.1.1.1	DATABASE.PY	8
4.3.1.1.2	MODELS.PY	9
4.3.1.2	ROUTERS	9
4.3.1.2.1	PROYECTO.PY.....	10
4.3.1.2.2	TAREA .PY.....	11
4.3.1.2.3	USUARIO.PY	12
4.3.1.3	SCHEMAS.PY	13
4.3.2	MAIN.PY	14
4.3.3	PASOS_A_SEGUIR.PY	14
4.3.4	REQUIREMENTS.TXT	15
4.4	Implementación – Principales librerías utilizadas	15
4.5	Pruebas	16

4.6	Despliegue de la aplicación.....	17
5	Manuales	17
5.1	Manual de usuario	17
5.2	Manual de instalación.....	18
6	Conclusiones y posibles ampliaciones	19
7	Bibliografía	20

1 Introducción

He desarrollado una API que permite emplear el método CRUD en cada una de sus tres tablas. Es un gestor de proyectos, donde cada proyecto se desglosa en varias tareas, aunque cada tarea pertenece únicamente a un solo proyecto. Además, un usuario solo puede pertenecer a un proyecto, pero un proyecto puede estar compuesto por varios usuarios.

2 Estado del arte

2.1 Definición de arquitectura de microservicios

La arquitectura de microservicios es un método de desarrollo de aplicaciones software que funciona como un conjunto de pequeños servicios que se ejecutan de manera independiente y autónoma, proporcionando una funcionalidad de negocio completa. En ella, cada microservicio es un código que puede estar en un lenguaje de programación diferente, y que desempeña una función específica. Los microservicios se comunican entre sí a través de APIs, y cuentan con sistemas de almacenamiento propios, lo que evita la sobrecarga y caída de la aplicación. Definición de API.

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

Una API (Interfaz de Programación de Aplicaciones) define un conjunto de reglas y mecanismos para la comunicación entre sistemas. Su estructura se compone de varios elementos clave:

1. Protocolo utilizado

- **HTTP/HTTPS:** utilizado en APIs REST y GraphQL.
- **gRPC:** utiliza HTTP/2 y Protobuf para mejorar la eficiencia.
- **WebSockets:** para comunicación bidireccional en tiempo real.

2. Métodos HTTP (Verbos)

- **GET:** obtener datos.
- **POST:** crear un nuevo recurso.
- **PUT:** modificar o reemplazar un recurso existente.
- **PATCH:** modificar parcialmente un recurso.
- **DELETE:** eliminar un recurso.

3. Partes de una URL en una API REST

La URL de una API REST sigue una estructura clara y semántica:

- **Base URL:** dirección principal del servidor.
- **Recurso:** representa la entidad sobre la que se opera. Ejemplo: /usuarios
- **Identificador** (opcional): para referirse a un recurso específico. Ejemplo: /usuarios/123
- **Parámetros de consulta (Query Params):** Se utilizan para filtrar o modificar la respuesta. Ejemplo: ?nombre=Juan&edad=25
- **Cabeceras HTTP (Headers):** información adicional, como autenticación o tipo de contenido (Authorization, Content-Type).

4. Respuestas y Códigos de Estado HTTP

- **200 OK:** éxito en la solicitud.
- **201 Created:** recurso creado exitosamente.
- **400 Bad Request:** error en la solicitud del cliente.
- **401 Unauthorized:** falta autenticación.
- **404 Not Found:** recurso no encontrado.
- **500 Internal Server Error:** error en el servidor.

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

1. FastAPI es un framework rápido y ligero para construir APIs modernas en Python 3.6 o superior. Destacado como uno de los mejores frameworks de código abierto en 2021, ofrece varias ventajas:
 - **Velocidad:** comparable a Go y Node.js, siendo ideal para APIs rápidas.
 - **Facilidad de uso:** permite crear APIs RESTful listas para producción con pocas líneas de código.
 - **Documentación automática:** genera documentación detallada utilizando OpenAPI.
 - **Menos errores:** la validación de datos personalizada reduce los errores humanos en un 40%.
 - **Type hints:** utiliza la tipificación estática de Python para mejorar el soporte de los IDE y predecir errores con mayor precisión.
2. Flask es un framework ligero y flexible para crear APIs en Python. Para crear una API con Flask, solo se necesitan unas pocas líneas de código:
 - **Instalación:** se instala Flask mediante `pip install flask`.
 - **Definición de la aplicación:** se crea una instancia de la clase Flask.
 - **Rutas y métodos:** se definen las rutas y los métodos HTTP (GET, POST, etc.) utilizando decoradores.
 - **Respuesta:** se devuelven respuestas JSON utilizando `jsonify`.

Y, aunque sea fácil de aprender, no ofrece documentación automática ni validación de datos por defecto.

En mi proyecto he utilizado FastAPI para desarrollar la API, ya que es la tecnología que hemos estado trabajando en clase. Además, genera documentación automática y valida los datos con Pydantic.

3 Descripción general del proyecto

3.1 Objetivos

En el desarrollo de esta API, mi objetivo era aprender a crear una API utilizando FastAPI y asegurarme de que la implementación de los métodos CRUD funcionara correctamente en las tres tablas, teniendo en cuenta las relaciones entre ellas. Estas relaciones añadieron cierta complejidad al desarrollo, ya que, por ejemplo, un usuario solo puede pertenecer a un proyecto, lo que implica que antes de crear usuarios debe existir al menos un proyecto.

3.2 Entorno de trabajo (tecnologías de desarrollo y herramientas)

En el desarrollo de esta API, he utilizado varias herramientas y tecnologías que han facilitado su implementación y ejecución:

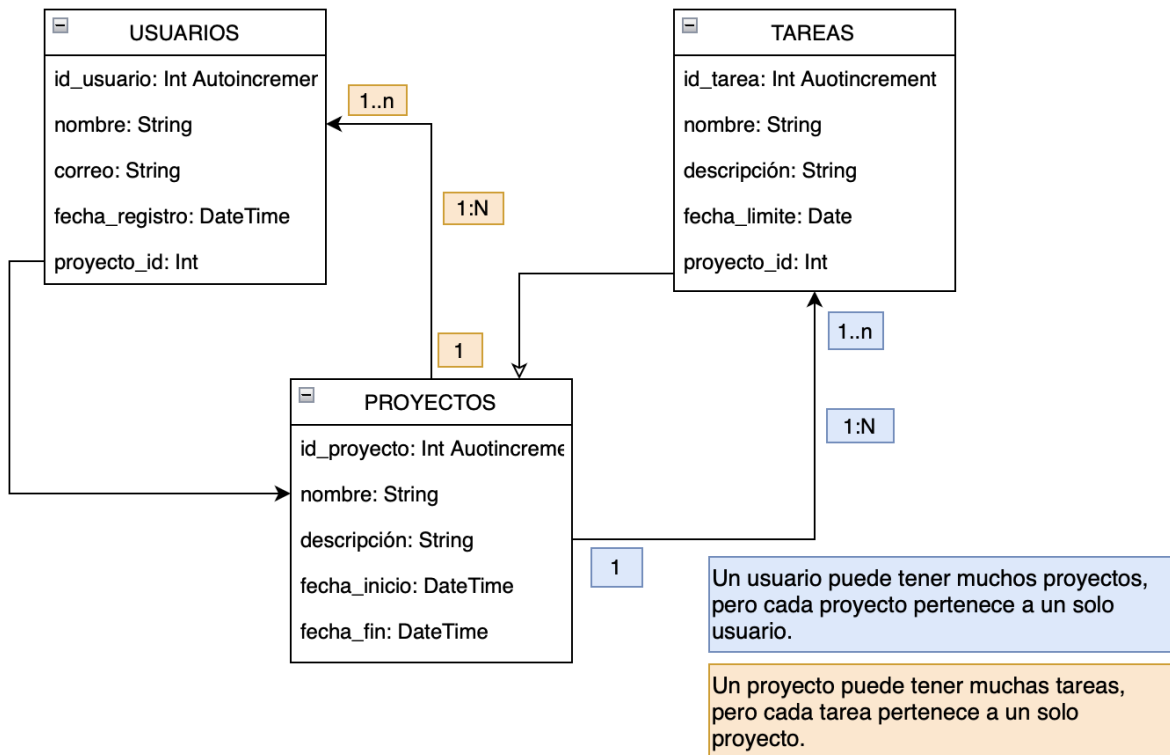
- **Python:** lenguaje de programación principal de la API.
- **FastAPI:** framework utilizado para el desarrollo de la API. Realiza la validación automática de datos y genera automáticamente la documentación.
- **PostgreSQL:** gestor de base de datos utilizado para almacenar la información de los proyectos, tareas y usuarios.
- **Docker:** utilizada para contenerizar la aplicación, asegurando que el entorno de desarrollo y producción sean consistentes, facilitando la implementación y despliegue de la API.
- **Visual Studio Code:** editor de código utilizado durante el desarrollo, ya que ofrece compatibilidad con los lenguajes y tecnologías empleadas.
- **Swagger:** herramienta integrada en FastAPI que permite generar automáticamente la documentación interactiva de la API

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

4.1 Análisis del sistema (funcionalidades básicas de la aplicación)

1. API: permite la comunicación entre el cliente y la base de datos a través de HTTP. Las operaciones CRUD son las siguientes:
 - **Crear (POST):** se utiliza para añadir nuevos registros a las tablas.
 - **Leer (GET):** permite obtener información de las tablas.
 - **Actualizar (PUT/PATCH):** se usa para modificar los registros existentes.
 - **Eliminar (DELETE):** permite eliminar registros de la base de datos.Las tres tablas en la API están interconectadas con relaciones, lo que permite gestionar los datos de manera coherente y asegurar la integridad referencial.
2. La API está diseñada para manejar de manera eficiente los errores que puedan surgir durante su ejecución. Esto incluye:
 - **Errores de validación:** si los datos enviados en una solicitud no cumplen con los requisitos (por ejemplo, un campo obligatorio está vacío o un formato no es válido), la API responde con un mensaje de error adecuado.
 - **Errores del servidor:** en caso de problemas internos del servidor, la API devolverá un error 500 (Internal Server Error) con detalles sobre la causa del fallo.
3. También se envía un mensaje de confirmación, indicando si la operación se ha completado correctamente, ya sea añadir, eliminar o actualizar un registro. Estos mensajes no solo informan al usuario o cliente de la API sobre el resultado de la acción, sino que también aportan claridad sobre lo ocurrido.

4.2 Diseño de la base de datos



4.3 Implementación – Estructura del código

Así se ve mi proyecto:

```

  SGE_PROYECTO2EVA-MAIN
  ├── __pycache__
  ├── app
  │   ├── db
  │   ├── repository
  │   ├── routers
  │   └── schemas.py
  ├── main.py
  ├── pasos_a_seguir.txt
  ├── README.md
  └── requirements.txt
  
```



```

  SGE_PROYECTO2EVA-MAIN
  ├── __pycache__
  ├── app
  │   ├── db
  │   │   ├── database.py
  │   │   ├── models.py
  │   ├── repository
  │   │   ├── __init__.py
  │   ├── routers
  │   │   ├── __init__.py
  │   │   ├── proyecto.py
  │   │   ├── tarea.py
  │   │   ├── usuario.py
  │   └── schemas.py
  ├── main.py
  ├── pasos_a_seguir.txt
  ├── README.md
  └── requirements.txt
  
```

4.3.1 APP

Carpeta que contiene todo lo relacionado con la lógica de la API y con todo el tema de la base de datos.

4.3.1.1 DB

Carpeta que contiene todo el tema de la base de datos, tanto la obtención de la base de datos, como de la propia base de datos.

4.3.1.1.1 DATABASE.PY

Clase que establece una conexión con una base de datos PostgreSQL usando SQLAlchemy, creando un motor de conexión, un generador de sesiones y una clase base para los modelos de tablas. Además, define una función para obtener y gestionar sesiones de base de datos.

```
app > db > database.py > ...
1  from sqlalchemy import create_engine
2  from sqlalchemy.ext.declarative import declarative_base
3  from sqlalchemy.orm import sessionmaker
4
5  #URL de la base de datos
6  SQLALCHEMY_DATABASE_URL = "postgresql://odoo:odoo@localhost:5342/fastapi-database-proyecto2eva"
7
8  '''
9  postgresql: El tipo de base de datos.
10 odoo:odoo: Las credenciales de usuario (usuario: odoo, contraseña: odoo).
11 localhost:5342: Dirección del servidor de base de datos (localhost y puerto 5342).
12 fastapi-database: Nombre de la base de datos.
13 '''
14
15 #Crea el motor de conexión (engine) que interactuará con la base de datos utilizando la URL espe
16 engine = create_engine(SQLALCHEMY_DATABASE_URL)
17 #Configura un generador de sesiones
18 SessionLocal = sessionmaker(bind=engine,autocommit=False,autoflush=False)
19 #Crea una clase base llamada Base, que se utiliza para definir los modelos de tablas de la base
20 Base = declarative_base()
21
22 def get_db():
23     db = SessionLocal() #Crear una nueva sesión
24     try:
25         yield db #Devuelvo la sesión para su uso
26     except Exception as e:
27         print(e)
28     finally:
29         db.close() #Cierro la sesión después de usarla
```


4.3.1.1.2 MODELS.PY

Clase que alberga las clases que definen los modelos de tres tablas en una base de datos utilizando SQLAlchemy: UsuarioTable para los usuarios, ProyectoTable para los proyectos y TareaTable para las tareas. Establecen relaciones entre ellas, como la relación entre usuarios y proyectos, y tareas y proyectos, con detalles como las claves foráneas y atributos adicionales de cada tabla.

```
app > db > models.py > UsuarioTable
1  from sqlalchemy.orm import relationship
2  from sqlalchemy import Column, Date,Integer,String,Boolean,DateTime, Text
3  from datetime import datetime
4  from sqlalchemy.schema import ForeignKey
5  from app.db.database import Base
6
7  #Modelo de la tabla usuarios
8  class UsuarioTable(Base):
9      __tablename__ = "usuarios"
10
11      id = Column(Integer, primary_key=True, autoincrement=True)
12      nombre = Column(String(100), nullable=False)
13      correo = Column(String(100), unique=True, nullable=False)
14      fecha_registro = Column(DateTime, default=datetime.now, onupdate=datetime.now)
15      id_proyecto = Column(Integer, ForeignKey("proyectos.id"), nullable=False)
16
17  #Modelo de la tabla proyectos
18  class ProyectoTable(Base):
19      __tablename__ = "proyectos"
20
21      id = Column(Integer, primary_key=True, autoincrement=True)
22      nombre = Column(String(100), nullable=False)
23      descripcion = Column(Text, nullable=True)
24      fecha_inicio = Column(Date, nullable=False)
25      fecha_fin = Column(Date, nullable=True)
26
27      #poner el id del proyecto en las tablas de usuario y tareas
28      usuarios = relationship("Usuario", backref="usuario") #no pongo cascade porque
29      tareas = relationship("Tarea", backref="tarea", cascade="delete, merge")
30
31  #Modelo de la tabla tareas
32  class TareaTable(Base):
33      __tablename__ = "tareas"
34
35      id = Column(Integer, primary_key=True, autoincrement=True)
36      nombre = Column(String(100), nullable=False)
37      descripcion = Column(Text, nullable=True)
38      estado = Column(String(50), nullable=False)
39      fecha_limite = Column(Date, nullable=True)
40      proyecto_id = Column(Integer, ForeignKey("proyectos.id"), nullable=False)
```

4.3.1.2 ROUTERS

Carpeta que declara todas las peticiones a la API de cada tabla.

4.3.1.2.1 PROYECTO.PY

Clase que define varias rutas de una API en FastAPI para gestionar proyectos en una base de datos. Incluye funciones para obtener todos los proyectos, obtener un proyecto específico por su ID, crear un nuevo proyecto, eliminar un proyecto por su ID y actualizar los detalles de un proyecto (específicamente las fechas de inicio y fin). Utiliza SQLAlchemy para interactuar con la base de datos y maneja la sesión a través de Depends(get_db).

```
app > routers > proyecto.py > crear_proyecto
1 from fastapi import APIRouter, Depends
2
3 from app.db import models
4 from app.db.database import get_db
5 from app.schemas import Proyecto, UpdateProyecto
6
7 from sqlalchemy.orm import Session
8
9 router = APIRouter(
10     prefix="/proyecto",
11     tags=["Proyectos"]
12 )
13
14 #OBTENER TODOS LOS PROYECTOS
15 @router.get("/obtener_proyectos")
16 def obtener_proyectos(db:Session=Depends(get_db)):
17     proyectos = db.query(models.ProyectoTable).all()
18     print(proyectos)
19     return proyectos
20
21
22 #OBTENER PROYECTO POR ID -> ÚTIL PARA VER LA INFO DE UN PROYECTO EN CONCRETO
23 @router.get("/obtener_proyecto_por_id/{proyecto_id}")
24 def obtener_proyecto_por_id(proyecto_id:int,db:Session=Depends(get_db)):
25     proyecto = db.query(models.ProyectoTable).filter(models.ProyectoTable.id == proyecto_id).first() #obtengo el proyecto cuyo id es el pasado por parámetro
26     if proyecto:
27         print(proyecto.nombre)
28         return proyecto
29     return {"Respuesta": "Proyecto no encontrado"}
30
31
32
33 #CREAR UN NUEVO PROYECTO
34 @router.post("/crear_proyecto")
35 def crear_proyecto(proyecto:Proyecto, db:Session=Depends(get_db)):
36     try:
37         newProject = proyecto.model_dump()
38         nuevo_proyecto = models.ProyectoTable(
39             #no añado ni el id ni la fecha de registro porque el id es autoincrementable y la fecha se obtiene de la que sea actualmente
40             nombre = newProject["nombre"],
41             descripcion = newProject["descripcion"],
42             fecha_inicio = newProject["fecha_inicio"],
43             fecha_fin = newProject["fecha_fin"],
44         )
45         db.add(nuevo_proyecto)
46         db.commit()
47         db.refresh(nuevo_proyecto)
48         return {"Respuesta": "Proyecto creado correctamente"}
49     except Exception as e:
50         return {"Error": e.args}
51
52
53 #ELIMINAR UN PROYECTO POR SU ID
54 @router.delete("/eliminar_proyecto/{proyecto_id}")
55 def eliminar_proyecto_por_id(proyecto_id:int, db:Session=Depends(get_db)):
56     deleteProject = db.query(models.ProyectoTable).filter(models.ProyectoTable.id == proyecto_id).first()
57     print(deleteProject)
58     if not deleteProject:
59         return {"Respuesta": "Proyecto no encontrado"}
60     db.delete(deleteProject)
61     db.commit()
62     return {"Respuesta": "Proyecto eliminado correctamente"}
63
64
65 #MODIFICAR PROYECTOS -> solo fecha de inicio y fin
66 @router.put("/modificar_proyecto/{proyecto_id}")
67 def actualizar_proyecto_por_id(proyecto_id:int, updateProject:UpdateProyecto, db:Session=Depends(get_db)):
68     actualizarProject = db.query(models.ProyectoTable).filter(models.ProyectoTable.id == proyecto_id).first()
69     if actualizarProject:
70         actualizarProject.fecha_inicio = updateProject.fecha_inicio
71         actualizarProject.fecha_fin = updateProject.fecha_fin
72         db.commit()
73         return {"Respuesta": "Proyecto actualizado correctamente"}
74     else:
75         return {"Respuesta": "Proyecto no encontrado"}
```

4.3.1.2.2 TAREA .PY

Clase que define varias rutas en una API de FastAPI para gestionar tareas asociadas a proyectos. Incluye funciones para obtener todas las tareas, obtener una tarea específica por ID, crear una nueva tarea, eliminar una tarea por ID y actualizar una tarea (específicamente su fecha límite y estado). Se utiliza SQLAlchemy para interactuar con la base de datos, y las tareas están almacenadas en la variable listatareas para pruebas.

```

app.router.add_api_route("/", tarea.py)
1 from fastapi import APIRouter, Depends
2 from app.db import models
3 from app.db.database import get_db
4 from app.schemas import Tarea, UpdateTarea
5
6 from sqlalchemy.orm import Session
7
8 router = APIRouter(
9     prefix="/tarea",
10    tags=["Tareas"]
11)
12
13#OBTENER TODAS LAS TAREAS
14@router.get("/obtener_tareas")
15def obtener_tareas(db:Session=Depends(get_db)):
16    tareas = db.query(models.TareaTable).all()
17    print(tareas)
18    return tareas
19
20
21#OBTENER TAREAS POR ID -> ÚTIL PARA VER LA INFO DE UNA TAREA CONCRETA
22@router.get("/obtener_tarea_por_id/{tarea_id}")
23def obtener_tarea_por_id(tarea_id:int,db:Session=Depends(get_db)):
24    tarea = db.query(models.TareaTable).filter(models.TareaTable.id == tarea_id).first() #obtengo la tarea cuyo id es el pasado por parámetro
25    if tarea:
26        print(tarea.nombre)
27        return tarea
28    return {"Respuesta": "Tarea no encontrado"}
29
30
31#OBTENER TAREAS POR ID DEL PROYECTO (FK) -> ÚTIL PARA VER CUÁNTAS TAREAS HAY RELACIONADAS A UN PROYECTO Y EN QUÉ CHISTE CADA UNA
32@router.get("/obtener_tarea_por_id_proyecto/{proyecto_id}")
33def obtener_tarea_por_id_proyecto(proyecto_id:int,db:Session=Depends(get_db)):
34    proyecto_tarea = db.query(models.TareaTable).filter(models.TareaTable.proyecto_id == proyecto_id).all() #obtengo todas aquellas tareas que tienen el id_proyecto igual
35    if proyecto_tarea:
36        return proyecto_tarea
37    return {"Respuesta": "Tareas pertenecientes al proyecto no encontradas"}
38
39
40#CREAR UNA NUEVA TAREA
41@router.post("/crear_tarea")
42def crear_tarea(tarea:Tarea, db:Session=Depends(get_db)):
43    newTask = tarea.model_dump()
44    nueva_tarea = models.TareaTable(
45        #no añadido el id porque es autoincrementable
46        nombre = newTask["nombre"],
47        descripcion = newTask["descripcion"],
48        estado = newTask["estado"],
49        fecha_limite = newTask["fecha_limite"],
50        proyecto_id = newTask["proyecto_id"],
51    )
52    db.add(nueva_tarea)
53    db.commit()
54    return {"Respuesta": "Tarea creada"}
55
56
57#ELIMINAR UNA TAREA POR SU ID
58@router.delete("/eliminar_tarea/{tarea_id}")
59def eliminar_tarea(tarea_id:int, db:Session=Depends(get_db)):
60    deleteTask = db.query(models.TareaTable).filter(models.TareaTable.id == tarea_id).first()
61    if not deleteTask:
62        return {"Respuesta": "Tarea no encontrada"}
63    db.delete(deleteTask)
64    db.commit()
65    return {"Respuesta": "Tarea eliminada correctamente"}
66
67
68#MODIFICAR TAREAS -> solo fecha límite y el estado
69@router.put("/modificar_tarea/{tarea_id}")
70def actualizar_tarea_por_id(tarea_id:int, updateTask:UpdateTarea, db:Session=Depends(get_db)):
71    actualizarTask = db.query(models.TareaTable).filter(models.TareaTable.id == tarea_id).first()
72    if actualizarTask:
73        actualizarTask.estado = updateTask.estado
74        actualizarTask.fecha_limite = updateTask.fecha_limite
75        db.commit()
76        return {"Respuesta": "Tarea actualizada correctamente"}
77    else:
78        return {"Respuesta": "Tarea no encontrada"}

```

4.3.1.2.3 USUARIO.PY

Clase que define varias rutas en una API de FastAPI para gestionar usuarios. Incluye funciones para obtener todos los usuarios, obtener un usuario por ID, crear un nuevo usuario, eliminar un usuario por ID y modificar un usuario (actualizando solo los campos enviados). Se utiliza SQLAlchemy para interactuar con la base de datos y gestionar las operaciones de los usuarios. Además, las respuestas se devuelven en formato JSON, proporcionando retroalimentación sobre el estado de cada operación.

```
app > routers > usuario.py > ...
1 from fastapi import APIRouter, Depends
2
3 from app.db import models
4 from app.db.database import get_db
5 from app.schemas import UpdateUsuario, Usuario
6
7 from sqlalchemy.orm import Session
8
9 router = APIRouter(
10     prefix="/usuario",
11     tags=["Usuarios"]
12 )
13
14 #OBTENER TODOS LOS USUARIOS
15 @router.get("/obtener_usuarios")
16 def obtener_usuarios(db:Session=Depends(get_db)):
17     usuarios = db.query(models.UsuarioTable).all()
18     return usuarios
19
20
21 #OBTENER USUARIO POR ID -> ÚTIL PARA VER A QUÉ PROYECTO PERTENECE
22 @router.get("/obtener_usuario_por_id/{user_id}")
23 def obtener_usuario_por_id(user_id:int,db:Session=Depends(get_db)):
24     usuario = db.query(models.UsuarioTable).filter(models.UsuarioTable.id == user_id).first() #obtengo el usuario cuyo id es el pasado por parámetro
25     if not usuario:
26         return {"Respuesta": "Usuario no encontrado"}
27     return usuario
28
29
30 #CREAR UN NUEVO USUARIO
31 @router.post("/crean_usuario")
32 def crear_usuario(user:Usuario, db:Session=Depends(get_db)):
33     try:
34         newUser = user.model_dump()
35         nuevo_usuario = models.UsuarioTable(
36             #no añado ni el id ni la fecha de registro porque el id es autoincrementable y la fecha se obtiene de la que sea actualmente
37             nombre = newUser["nombre"],
38             correo = newUser["correo"],
39             id_proyecto = newUser["id_proyecto"]
40         )
41         db.add(nuevo_usuario)
42         db.commit()
43         db.refresh(nuevo_usuario)
44         return {"Respuesta": "Usuario creado correctamente"}
45     except Exception as e:
46         return {"Error": e.args}
47
48
49
50 #ELIMINAR UN USUARIO POR SU ID
51 @router.delete("/eliminar_usuario/{user_id}")
52 def eliminar_usuario_por_id(user_id:int, db:Session=Depends(get_db)):
53     deleteUser = db.query(models.UsuarioTable).filter(models.UsuarioTable.id == user_id).first()
54     if not deleteUser:
55         return {"Respuesta": "Usuario no encontrado"}
56     db.delete(deleteUser)
57     db.commit()
58     return {"Respuesta": "Usuario eliminado correctamente"}
59
60
61 #MODIFICAR USUARIOS
62 @router.patch("/modificar_usuario/{user_id}")
63 def actualizar_usuario_por_id(user_id:int, updateUser:UpdateUsuario, db:Session=Depends(get_db)):
64     actualizarUser = db.query(models.UsuarioTable).filter(models.UsuarioTable.id == user_id).first()
65     if actualizarUser:
66         actualizarUser.nombre = updateUser.nombre
67         actualizarUser.correo = updateUser.correo
68         actualizarUser.id_proyecto = updateUser.id_proyecto
69         db.commit()
70         return {"Respuesta": "Usuario actualizado correctamente"}
71     else:
72         return {"Respuesta": "Usuario no encontrado"}
```

4.3.1.3 SCHEMAS.PY

Clase que define varias clases de modelos usando Pydantic para la validación de datos y la creación de esquemas de entrada y salida en una API de FastAPI. Las clases update están diseñadas para permitir actualizaciones parciales de registros en la base de datos.

```
app > schemas.py > Tarea
1  from datetime import date, datetime
2  from typing import Optional
3  from pydantic import BaseModel, EmailStr
4
5  #CLASE USUARIO
6  class Usuario(BaseModel):
7      nombre: str
8      correo: EmailStr #propio de Pydantic para validar que sea un correo válido
9      fecha_registro: datetime = datetime.now() #fecha actual por defecto
10     id_proyecto: int
11
12     #CLASE PROYECTO
13     class Proyecto(BaseModel):
14         nombre: str
15         descripcion: Optional[str]
16         fecha_inicio: date
17         fecha_fin: date
18
19     #CLASE TAREA
20     class Tarea(BaseModel):
21         nombre: str
22         descripcion: Optional[str]
23         estado: str
24         fecha_limite: date
25         proyecto_id: int
26
27     #Clases para modificar registros de la base de datos -> la creo para que no sean campos requeridos,
28     #CLASE PARA MODIFICAR UN USUARIO
29     class UpdateUsuario(BaseModel):
30         nombre: str = None
31         correo: EmailStr = None
32         id_proyecto: int = None
33
34     #CLASE PARA MODIFICAR UN PROYECTO
35     class UpdateProyecto(BaseModel):
36         fecha_inicio: date = None
37         fecha_fin: date = None
38
39     #CLASE PARA MODIFICAR UNA TAREA
40     class UpdateTarea(BaseModel):
41         estado: str = None
42         fecha_limite: date = None
```

4.3.2 MAIN.PY

Esta es la clase principal que inicializa una aplicación de FastAPI y configura las rutas para gestionar proyectos, tareas y usuarios. Además de crear las tablas en la base de datos.

```
main.py > ...
1  from fastapi import FastAPI
2  import uvicorn
3
4  from app.db.database import Base, engine
5  from app.routers import proyecto, tarea, usuario
6
7  def create_tables():
8      Base.metadata.create_all(bind=engine)
9
10
11  #create_tables() -> solo llamar a este método una vez porque
12
13  app = FastAPI()
14
15  app.include_router(proyecto.router)
16  app.include_router(usuario.router)
17  app.include_router(tarea.router)
18
19
20  #Ejecutar FastAPI
21  if __name__ == "__main__":
22      uvicorn.run("main:app", port=8000, reload=True)
```

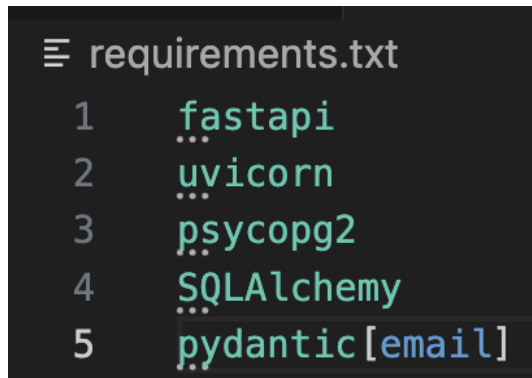
4.3.3 PASOS_A_SEGUIR.PY

Fichero creado para saber qué pasos son los que hay que seguir cuando se quiera iniciar el proyecto.

```
pasos_a_seguir.txt
1  1. Activar el entorno virtual:
2  .\env\Scripts\activate.ps1
3
4  2. Instalar las librerías que almacena requirements.txt:
5  pip install -r .\requirements.txt
6
7  3. Arrancar el servidor:
8  Python .\main.py
9
10  4. Acceso a FastAPI con Swagger:
11  http://127.0.0.1:8000/docs
12
13  5. Acceso a la base de datos:
14  http://127.0.0.1/browser/
```

4.3.4 REQUIREMENTS.TXT

Fichero que almacena las librerías que necesita el proyecto para que funcione correctamente. Principalmente los necesarios son los cuatro primeros, que son esenciales para que funcione la API y para acceder a la base de datos. El quinto es una ampliación que he querido emplear para que cuando se introduzca el correo de una persona se siga la estructura correcta de una dirección de correo electrónico.



```
≡ requirements.txt
1  fastapi
2  uvicorn
3  psycopg2
4  SQLAlchemy
5  pydantic[email]
```

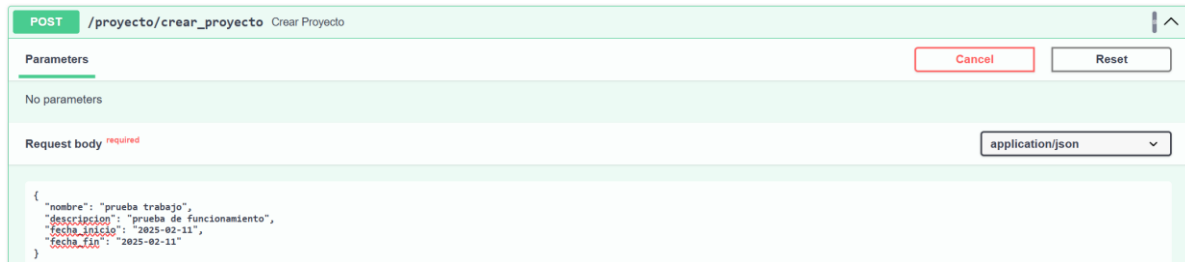
4.4 Implementación – Principales librerías utilizadas

1. **FastAPI:** framework para la creación de APIs en Python, que aprovecha las características asíncronas del lenguaje para ofrecer alto rendimiento y facilidad de uso en el desarrollo de aplicaciones web RESTful.
2. **Uvicorn:** servidor web rápido para aplicaciones ASGI, ideal para ejecutarse junto con FastAPI, permitiendo el manejo de solicitudes HTTP y la recarga automática durante el desarrollo.
3. **Psycopg2:** conector PostgreSQL para Python, utilizado para interactuar con bases de datos PostgreSQL, permitiendo la ejecución de consultas SQL y la gestión de conexiones.
4. **SQLAlchemy:** ORM que facilita la interacción con bases de datos SQL en Python. Se utiliza para definir modelos de datos, realizar consultas y operaciones CRUD de manera orientada a objetos.
5. **Pydantic:** librería para la validación y serialización de datos en Python, utilizada en FastAPI para definir los esquemas de entrada y salida de la API y garantizar la validez de los datos.
6. **Pydantic[email]:** extensión de Pydantic que agrega validación para direcciones de correo electrónico, utilizada en el modelo Usuario para asegurar que el correo proporcionado sea válido.
7. **Datetime:** módulo estándar de Python que proporciona clases para manejar fechas y horas, usado para gestionar fechas de registro y las fechas de inicio/fin de proyectos y tareas.
8. **Typing:** módulo estándar de Python que mejora la legibilidad del código mediante anotaciones de tipo, utilizado para definir tipos de datos como Optional y List en los modelos de datos.

4.5 Pruebas

Voy a realizar una prueba específica sobre la creación de un proyecto y la asignación de varias tareas a dicho proyecto, para luego mostrar todas las tareas asociadas.

1. Creo un nuevo proyecto



POST /proyecto/crear_proyecto Crear Proyecto

Parameters

No parameters

Request body *required* application/json

```
{
  "nombre": "prueba trabajo",
  "descripcion": "prueba de funcionamiento",
  "fecha_inicio": "2025-02-11",
  "fecha_fin": "2025-02-11"
}
```

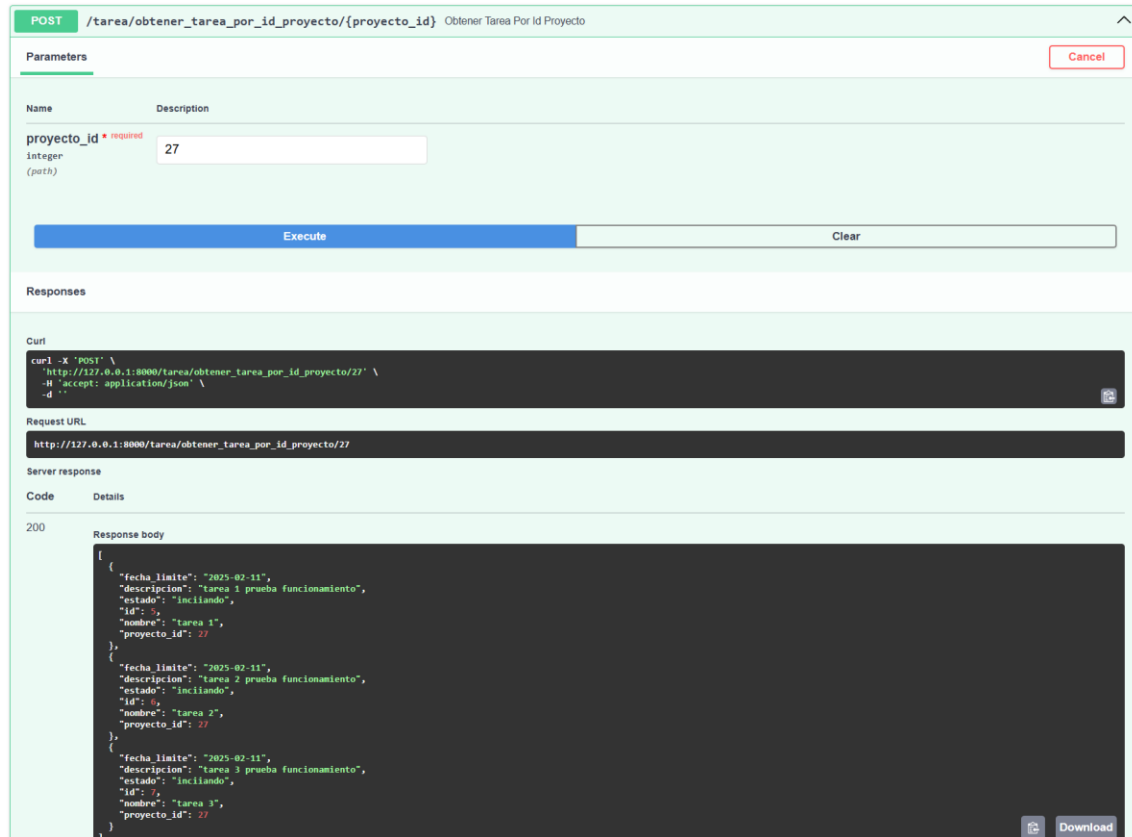
2. Encuentro el id que tiene dicho proyecto.

```
{
  "id": 27,
  "nombre": "prueba trabajo",
  "fecha_inicio": "2025-02-11",
  "descripcion": "prueba de funcionamiento",
  "fecha_fin": "2025-02-11"
}
```

3. Creo varias tareas asociadas a ese proyecto, comprobando que se han creado correctamente.

```
{
  "fecha_limite": "2025-02-11",
  "descripcion": "tarea 1 prueba funcionamiento",
  "estado": "inciando",
  "id": 5,
  "nombre": "tarea 1",
  "proyecto_id": 27
},
{
  "fecha_limite": "2025-02-11",
  "descripcion": "tarea 2 prueba funcionamiento",
  "estado": "inciando",
  "id": 6,
  "nombre": "tarea 2",
  "proyecto_id": 27
},
{
  "fecha_limite": "2025-02-11",
  "descripcion": "tarea 3 prueba funcionamiento",
  "estado": "inciando",
  "id": 7,
  "nombre": "tarea 3",
  "proyecto_id": 27
}
```


4. Hago la consulta que devuelve todas las tareas asociadas a ese id de proyecto.



POST /tarea/obtener_tarea_por_id_proyecto/{projecto_id} Obtener Tarea Por Id Proyecto

Parameters

Name	Description
projecto_id * required integer (path)	27

Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/tarea/obtener_tarea_por_id_proyecto/27' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

http://127.0.0.1:8000/tarea/obtener_tarea_por_id_proyecto/27

Server response

Code Details

200

Response body

```
{
  "fecha_limite": "2025-02-11",
  "descripcion": "tarea 1 prueba funcionamiento",
  "estado": "iniciando",
  "id": 1,
  "nombre": "tarea 1",
  "projecto_id": 27
},
{
  "fecha_limite": "2025-02-11",
  "descripcion": "tarea 2 prueba funcionamiento",
  "estado": "iniciando",
  "id": 2,
  "nombre": "tarea 2",
  "projecto_id": 27
},
{
  "fecha_limite": "2025-02-11",
  "descripcion": "tarea 3 prueba funcionamiento",
  "estado": "iniciando",
  "id": 3,
  "nombre": "tarea 3",
  "projecto_id": 27
}
```

Download

4.6 Despliegue de la aplicación

La API está diseñada para ejecutarse dentro de un contenedor Docker, lo que permite su despliegue tanto en un entorno local como en un servidor remoto sin complicaciones. Durante el desarrollo, se ha ejecutado en un entorno local mediante Docker, lo que garantiza un entorno de trabajo consistente. Gracias a la contenedorización, la API puede desplegarse fácilmente en cualquier servidor que soporte Docker, asegurando portabilidad y facilidad de implementación.

5 Manuales

5.1 Manual de usuario

En las siguientes imágenes se muestra cómo funciona cada uno de los métodos implementados en la API, junto con su respectiva documentación generada automáticamente por Swagger.

La API cuenta con tres vistosas separaciones que diferencian los diferentes métodos que se pueden llevar a cabo con cada tabla. Cada método cuenta con una breve descripción de lo que hace.

Proyectos	
GET	/proyecto/obtener_proyectos Obtener Proyectos
POST	/proyecto/obtener_proyecto_por_id/{projecto_id} Obtener Proyecto Por Id
POST	/proyecto/crear_proyecto Crear Proyecto
DELETE	/proyecto/eliminar_proyecto/{projecto_id} Eliminar Proyecto Por Id
PUT	/proyecto/modificar_proyecto/{projecto_id} Actualizar Proyecto Por Id

Usuarios	
GET	/usuario/obtener_usuarios Obtener Usuarios
GET	/usuario/obtener_usuario_por_id/{user_id} Obtener Usuario Por Id
POST	/usuario/crear_usuario Crear Usuario
DELETE	/usuario/eliminar_usuario/{user_id} Eliminar Usuario Por Id
PATCH	/usuario/modificar_usuario/{user_id} Actualizar Usuario Por Id
Tareas	
GET	/tarea/obtener_tareas Obtener Tareas
POST	/tarea/obtener_tarea_por_id/{tarea_id} Obtener Tarea Por Id
POST	/tarea/obtener_tarea_por_id_proyecto/{proyecto_id} Obtener Tarea Por Id Proyecto
POST	/tarea/crear_tarea Crear Tarea
DELETE	/tarea/eliminar_tarea/{tarea_id} Eliminar Tarea
PUT	/tarea/modificar_tarea/{tarea_id} Actualizar Tarea Por Id

5.2 Manual de instalación

1. Despliegue en un servidor local (localhost) con Docker

- Clonar el repositorio (si está en un repositorio).
- Construir la imagen de Docker.
- Ejecutar el contenedor.
- Acceder a la API en el navegador o con herramientas como Postman.
 - API: <http://localhost:8000>
 - Documentación Swagger: <http://localhost:8000/docs>

2. Despliegue en un servidor remoto con Docker

- Acceder al servidor por SSH.
- Clonar el repositorio o subir el código al servidor.
- Construir la imagen en el servidor.
- Ejecutar el contenedor exponiendo el puerto 8000.
- Configurar un proxy inverso (opcional) con Nginx o Caddy si se necesita acceso mediante un dominio.
- Acceder a la API desde cualquier lugar usando la IP pública del servidor:
 - API: http://ip_del_servidor:8000
 - Documentación Swagger: http://ip_del_servidor:8000/docs

6 Conclusiones y posibles ampliaciones

El principal problema que encontré durante el desarrollo de la aplicación fue la falta de tiempo, lo que impidió que pudiera completar la práctica tal como la tenía planeada. Esto afectó el desarrollo de algunas funcionalidades y me impidió cumplir con todos los requisitos establecidos.

En cuanto a la satisfacción, no es muy alta, ya que no logré finalizar todo el trabajo, pero sí me siento satisfecha con lo que he aprendido sobre el desarrollo de APIs utilizando FastAPI.

A pesar de las dificultades, el proceso me permitió adquirir conocimientos sobre la creación y gestión de APIs, y aunque no completé todo lo necesario, estoy contento con el progreso realizado.

Las ampliaciones que haría en mi trabajo se basan principalmente en completar lo que inicialmente se requería en la práctica, especialmente en la parte de autenticación con JWT. Implementar un sistema de autenticación y autorización mediante JWT permitiría gestionar de manera segura el acceso a la API, garantizando que solo los usuarios autorizados puedan acceder y modificar sus proyectos y tareas.

Relacionado con lo anterior, añadiría un atributo adicional a la tabla de usuarios que representaría una estructura jerárquica dentro de la empresa. Esta estructura permitiría que solo los usuarios con el rol de jefe pudieran crear proyectos y asignar un encargado. Tanto los jefes como los encargados tendrían permisos para modificar las fechas de inicio y fin de un proyecto concreto, así como la fecha límite de las tareas asociadas a ese proyecto. Todo esto requeriría un sistema de autenticación para garantizar que solo los usuarios autorizados pudieran realizar estas acciones, asegurando la integridad y seguridad de la información.

7 Bibliografía

- Decide Soluciones. Definición de arquitectura de microservicios:
<https://decidesoluciones.es/arquitectura-de-microservicios/>
- Sensedia. Estructura de una API: protocolo utilizado, métodos, partes de las URL de una API:
 - <https://www.sensedia.com/es/pillar/deconstruccion-de-las-api-componentes-y-estructura>
 - <https://docs.uipath.com/es/automation-suite/automation-suite/2023.10/api-guide/api-endpoint-url-structure>
 - https://es.semrush.com/blog/codigos-de-estado-http/?g_network=g&g_keyword=&g_acctid=951-454-0426&g_campaign=ES_SRCH_DSA_Blog_ES&g_keywordid=dsa-2229731903663&g_adtype=search&g_adid=678247171344&g_campaignid=19249322774&g_adgroupid=157746395729&kw=&cmp=ES_SRCH_DSA_Blog_ES&label=dsa_pagefeed&Network=g&Device=c&utm_content=678247171344&kwid=dsa-2229731903663&cmpid=19249322774&agpid=157746395729&BU=Core&extid=109498522492&adpos=&gad_source=1&gbraid=0AAAAADiv3HQFa4hSPOjXJVrS0QXQi dGfd&gclid=CjwKCAiAwaG9BhAREiwAdhv6Y045gEAF9P2k9iVlv7B107TiKHUI2ruXj66mmP02cln18HUjRniQwBoCQjMQAvD_BwE
- Kinsta. Formas de crear una API en Python: FastAPI y Flask:
 - <https://kinsta.com/es/blog/fastapi/>
 - <https://datascientest.com/es/programacion-de-api-web-en-python-con-flask>
- Para la consulta de errores desconocidos: ChatGPT.