



PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones
Multiplataforma
Informática y Comunicaciones

Proyecto final FastApi

Año: 2025

Fecha de presentación: 10/02/2025

Nombre y Apellidos: Saúl Mantecón de Caso
Email: saulmantecon9vdc@gmail.com

Índice

1. Introducción.....	3
2. Estado del arte.....	3
3. Descripción general del proyecto.....	4
3.1. Objetivos.....	4
El objetivo de este proyecto es crear una API que permita gestionar usuarios y reservas de pistas deportivas de manera eficiente y segura.	4
3.2. Entorno de trabajo.....	4
4. Documentación técnica.....	5
4.1. Análisis del sistema.....	5
La aplicación permite:	5
-Registro, gestión de usuarios y autenticación de usuarios. Para ello se usan los metodos get, post, delete y patch.	5
-Creación y consulta de pistas y reservas.....	5
4.2. Diseño de la base de datos.....	5
4.3. Implementación.....	6
4.3.1. Db.database.....	6
4.3.2. Db.models.....	7
4.3.3. Routers.usuario.....	8
4.3.4. Routers.pista.....	10
4.3.5. Routers.reserva.....	12
4.3.6. Schemas.....	14
4.3.7. Main.....	16
4.3.8. Requirements.txt.....	16
4.4. Pruebas.....	17
4.4.1. Crear usuario.....	17
4.4.2. Crear pista.....	18
4.4.3. Crear reserva.....	20
4.4.4. Delete Cascade.....	21
4.5. Despliegue de la aplicación.....	21
La aplicación puede desplegarse en un servidor local o en la nube mediante Docker. .	21
5. Conclusiones y posibles ampliaciones.....	22
6. Bibliografía.....	22

1. INTRODUCCIÓN.

El proyecto consiste en el desarrollo de una API para la gestión de reservas de pistas deportivas. La aplicación permite las operaciones CRUD para usuarios, pistas y reservas. Se ha implementado utilizando FastAPI y guardando los datos en una base de datos en PostgreSQL.

2. ESTADO DEL ARTE.

La arquitectura de microservicios consiste en dividir una aplicación en servicios pequeños e independientes, que pueden desplegarse y escalarse de manera autónoma.

Una API (Application Programming Interface) es un conjunto de reglas que permite la comunicación entre diferentes sistemas de software.

Una API sigue generalmente el protocolo HTTP y utiliza métodos como:

- GET: Obtener información
- POST: Crear recursos
- PUT: Actualizar recursos
- DELETE: Eliminar recursos

Las partes de una URL típica de una API son:

- Dominio: `https://miapi.com`
- Ruta: `/usuarios`
- Parámetros: `?id=1`

Flask: Flask es el segundo framework más popular de Python, el foco principal de éste es ser más liviano y flexible, basándose en la simplicidad.

Ventajas: facilidad para comenzar y agregar/modificar/eliminar código, bastante flexibilidad.

Desventajas: Pocos features soportados por defecto por lo que requiere instalación de varios paquetes externos y el posterior mantenimiento de los mismos, por ende, el desarrollador tiene que encargarse de mantenerlos actualizados y compatibles con el código propio.

- Al ser tan liviano y simple, no es la mejor opción para hacer proyectos grandes.

FastAPI: FastAPI es un framework moderno, rápido y robusto creado con la idea de aprovechar de forma nativa los type-hints de Python, enfocándose en la performance y en las necesidades de las aplicaciones de la actualidad.

Ventajas: las mismas que en flask, ASGI soportado de forma nativa, documentación autogenerada a partir de anotaciones, validación de tipos, incluso en JSON anidados, inyección de dependencias para validaciones automáticas, excelente performance, posibilidad de ser utilizado en proyectos de cualquier tamaño.

Desventajas: es un framework bastante nuevo, por lo tanto la comunidad no es tan grande y el material educativo (libros, cursos, ejemplos) no es tan extenso como en otros casos, el código no es estandarizado entre las diferentes empresas y la documentación oficial no es tan profunda, por lo que hacer implementaciones no tan triviales requiere un esfuerzo extra a la hora de resolver problemas o desafíos de implementación.

En este proyecto se ha elegido FastAPI debido a su rapidez, validación automática y compatibilidad con OpenAPI.

3. DESCRIPCIÓN GENERAL DEL PROYECTO.

3.1. Objetivos.

El objetivo de este proyecto es crear una API que permita gestionar usuarios y reservas de pistas deportivas de manera eficiente y segura.

3.2. Entorno de trabajo.

Las herramientas utilizadas son:

- **Lenguaje de programación:** Python.
- **Framework:** FastAPI.
- **Base de datos:** SQLAlchemy y PostgreSQL.
- **Gestión de dependencias:** Virtualenv.
- **Entorno de desarrollo:** PyCharm.

4. DOCUMENTACIÓN TÉCNICA.

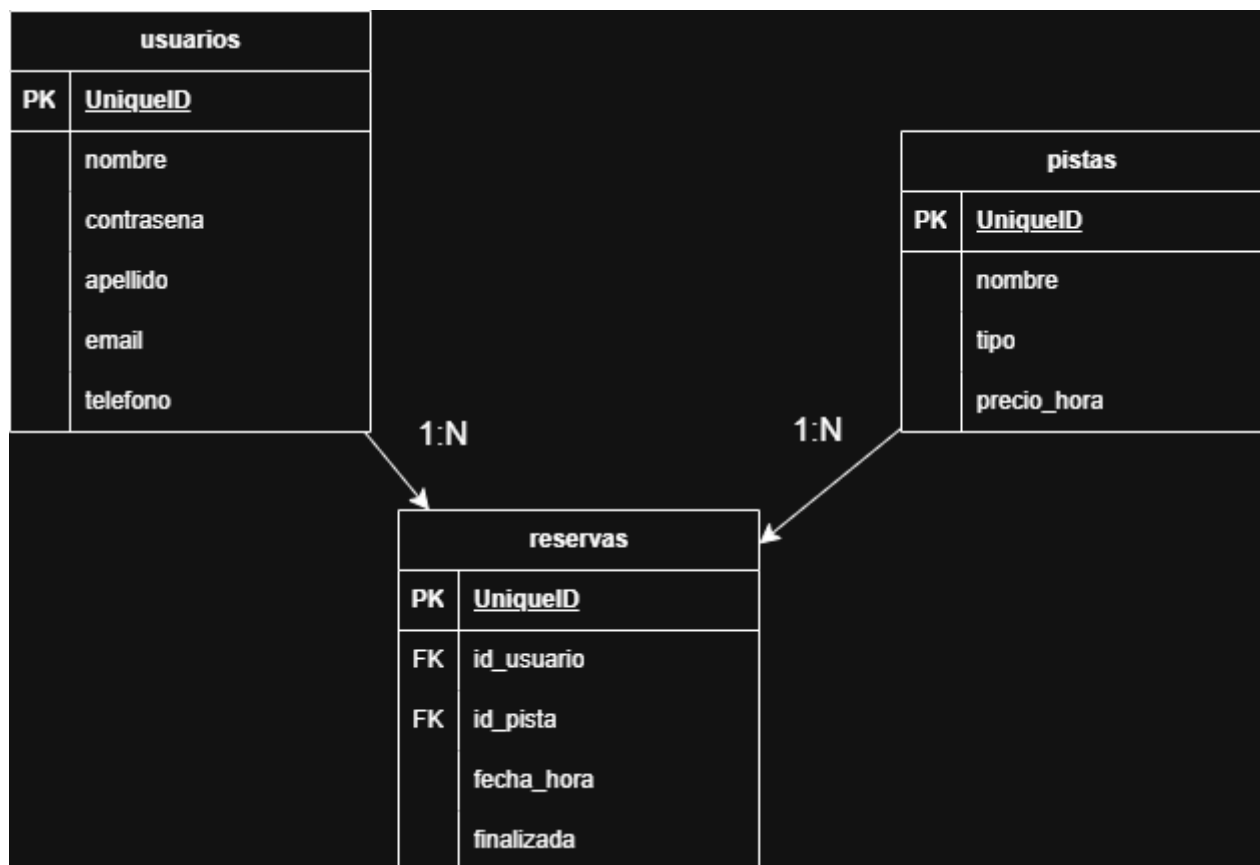
4.1. Análisis del sistema.

La aplicación permite:

-Registro, gestión de usuarios y autenticación de usuarios. Para ello se usan los metodos get, post, delete y patch.

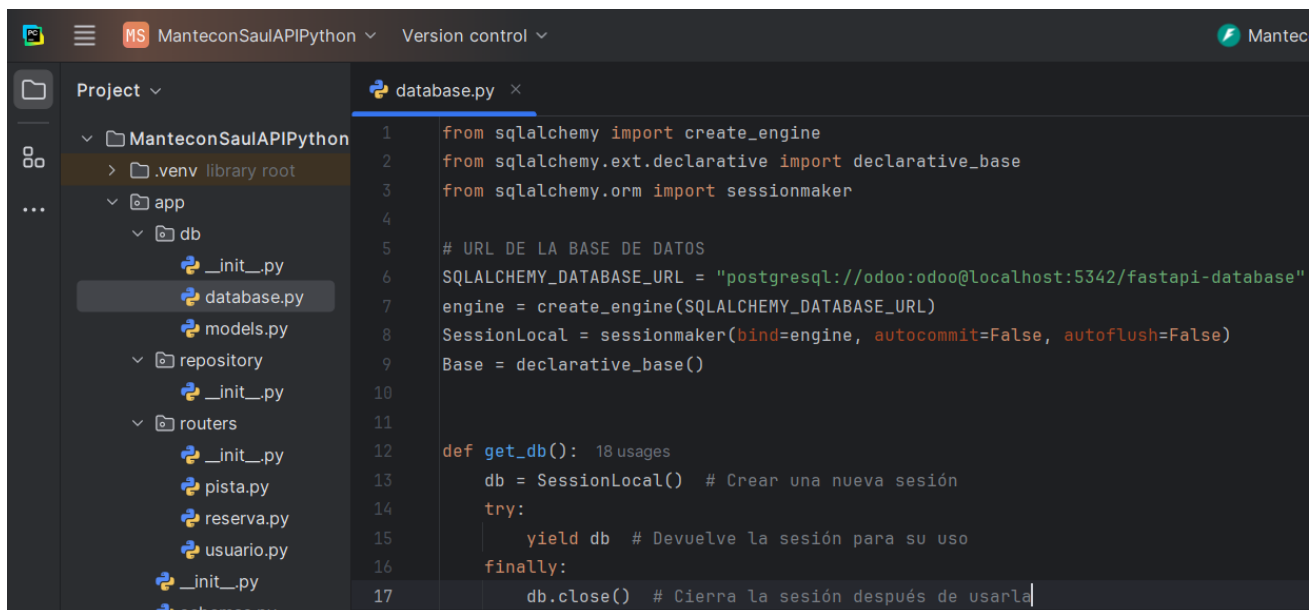
-Creación y consulta de pistas y reservas.

4.2. Diseño de la base de datos.



4.3. Implementación.

4.3.1. Db.database.



```
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"
7 engine = create_engine(SQLALCHEMY_DATABASE_URL)
8 SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
9 Base = declarative_base()
10
11
12 def get_db(): 18 usages
13     db = SessionLocal() # Crear una nueva sesión
14     try:
15         yield db # Devuelve la sesión para su uso
16     finally:
17         db.close() # Cierra la sesión después de usarla
```

-**from sqlalchemy import create_engine**: Se importa la función `create_engine`, que permite establecer la conexión con una base de datos especificando la URL de conexión.

- **from sqlalchemy.ext.declarative import declarative_base**: Importa `declarative_base`, una clase base que se utiliza para definir modelos de tablas de bases de datos en SQLAlchemy.

-**from sqlalchemy.orm import sessionmaker**: Importa `sessionmaker`, una clase que crea objetos para manejar sesiones con la base de datos.

-**SQLALCHEMY_DATABASE_URL**, define la URL de conexión a 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.

engine = create_engine(SQLALCHEMY_DATABASE_URL) Crea el motor de conexión (engine) que interactuará con la base de datos utilizando la URL especificada.

-**SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)**
Configura un generador de sesiones: `bind=engine`: Conecta las sesiones al motor de base de datos. `autocommit=False`: Indica que las transacciones no se confirman automáticamente; es necesario hacerlo manualmente. `autoflush=False`: Indica que no se envían automáticamente los cambios pendientes a la base de datos; esto se hace manualmente al confirmar.

4.3.2. Db.models

```

1  from sqlalchemy.orm import relationship
2  from sqlalchemy.schema import ForeignKey
3  from app.db.database import Base
4  from sqlalchemy import Column, Integer, String, Boolean, DECIMAL, DateTime
5  from datetime import datetime
6
7  class Usuario(Base): 8 usages
8      __tablename__ = 'usuarios' # Nombre de la tabla en la base de datos
9      id = Column(Integer, primary_key=True, index=True, autoincrement=True)
10     nombre = Column(String(100), nullable=False, unique=True)
11     contrasena = Column(String(100), nullable=False)
12     apellido = Column(String(100), nullable=False)
13     email = Column(String(100), unique=True, nullable=False)
14     telefono = Column(String(15))
15
16     # Relación con Reservas
17     reservas = relationship(argument="Reserva", backref="Usuario", cascade="all, delete")
18

```

```

↑
class Pista(Base): 8 usages
    __tablename__ = 'pistas'
    id = Column(Integer, primary_key=True, index=True, autoincrement=True)
    nombre = Column(String(100), nullable=False, unique=True)
    tipo = Column(String(100), nullable=False)
    precio_hora = Column(DECIMAL(precision=5, scale=2), nullable=False)

    # Relación con Reservas
    reservas = relationship(argument="Reserva", backref="Pista", cascade="all, delete")

```

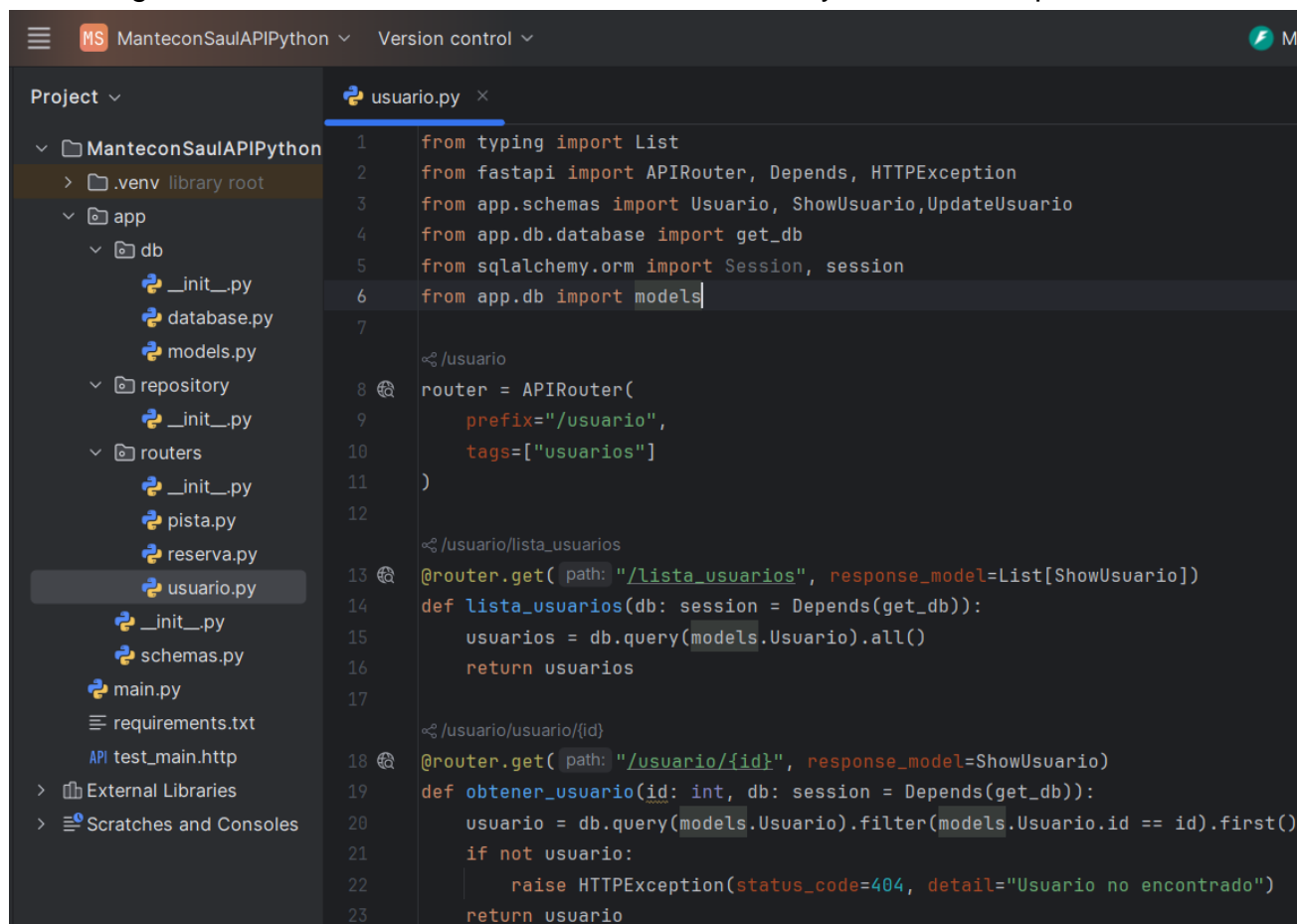
```

class Reserva(Base): 8 usages
    __tablename__ = 'reservas'
    id = Column(Integer, primary_key=True, index=True, autoincrement=True)
    id_usuario = Column(Integer, ForeignKey(column='usuarios.id', ondelete='CASCADE'), nullable=False)
    id_pista = Column(Integer, ForeignKey(column='pistas.id', ondelete='CASCADE'), nullable=False)
    fecha_hora = Column(DateTime, default=datetime.now, onupdate=datetime.now)
    finalizada = Column(Boolean, default=False)

```

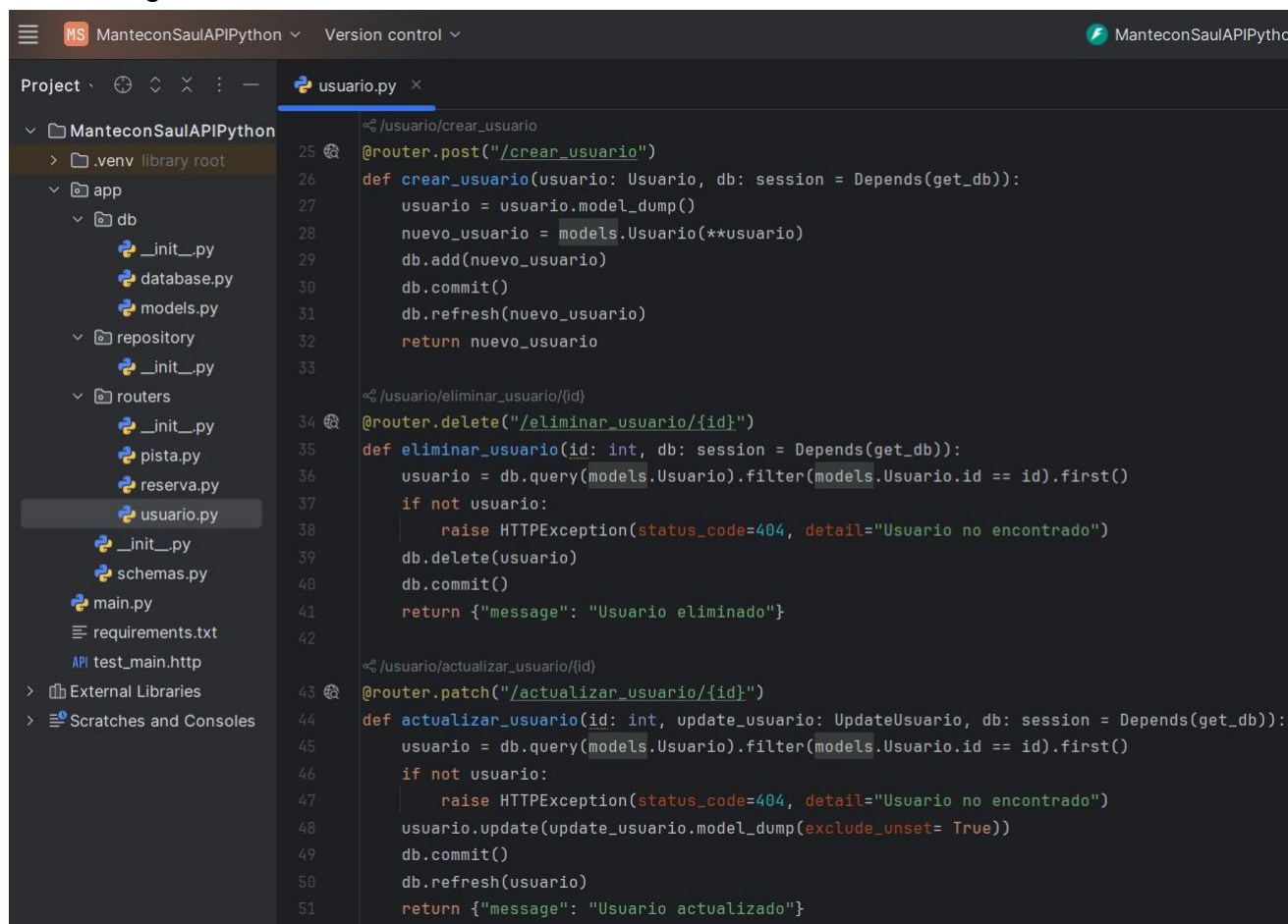
4.3.3. Routers.usuario.

Métodos get, listar todos los usuarios de la base de datos y listar usuario por id.



```
1 from typing import List
2 from fastapi import APIRouter, Depends, HTTPException
3 from app.schemas import Usuario, ShowUsuario, UpdateUsuario
4 from app.db.database import get_db
5 from sqlalchemy.orm import Session, session
6 from app.db import models
7
8 < /usuario
9 router = APIRouter(
10     prefix="/usuario",
11     tags=["usuarios"]
12 )
13
14 < /usuario/lista_usuarios
15 @router.get(path: "/lista_usuarios", response_model=List[ShowUsuario])
16 def lista_usuarios(db: session = Depends(get_db)):
17     usuarios = db.query(models.Usuario).all()
18     return usuarios
19
20 < /usuario/usuario/{id}
21 @router.get(path: "/usuario/{id}", response_model=ShowUsuario)
22 def obtener_usuario(id: int, db: session = Depends(get_db)):
23     usuario = db.query(models.Usuario).filter(models.Usuario.id == id).first()
24     if not usuario:
25         raise HTTPException(status_code=404, detail="Usuario no encontrado")
26     return usuario
```


Métodos para crear un usuario, eliminar por id y actualizar sin tener que rellenar todos los datos obligatoriamente.



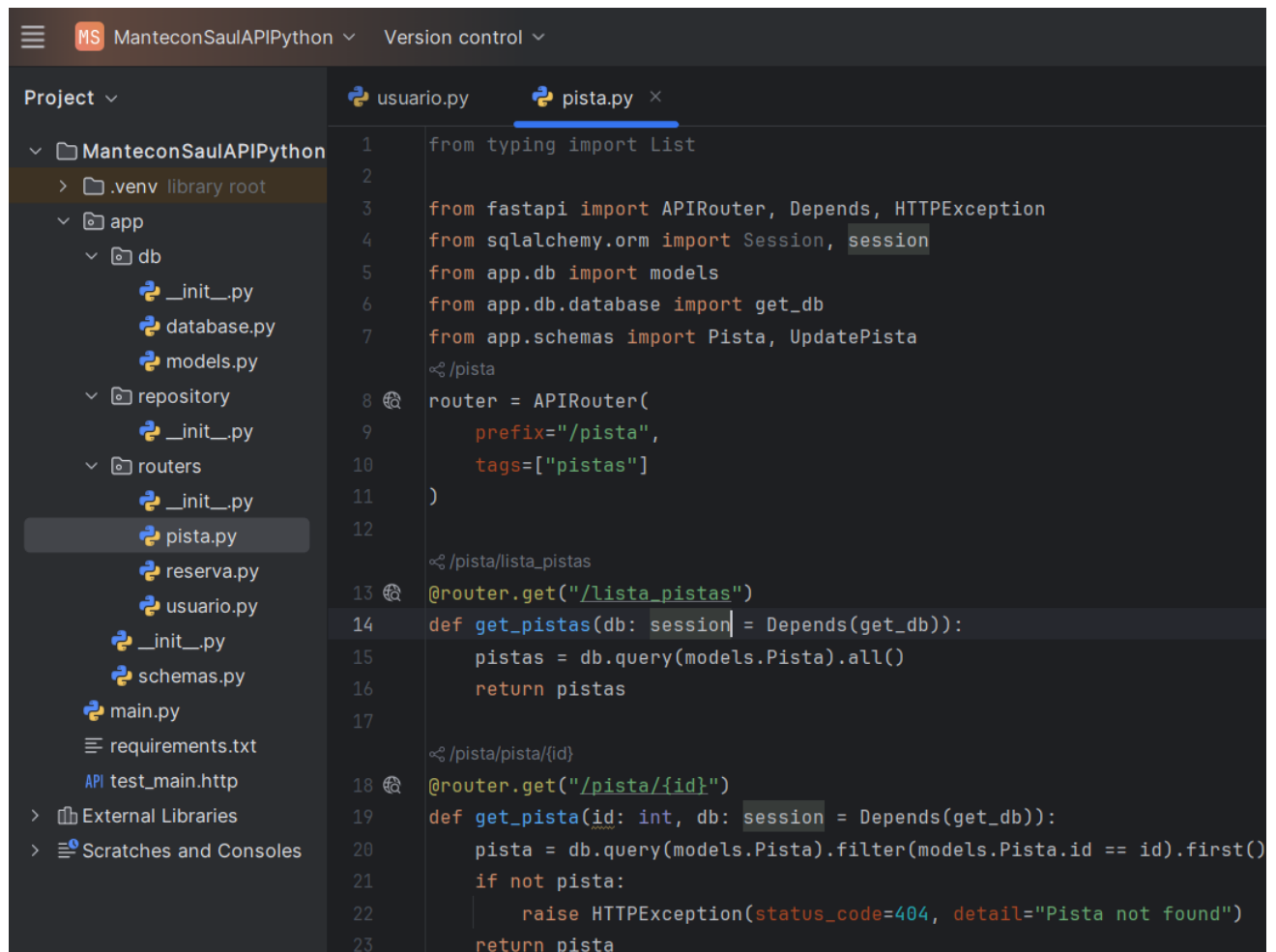
```
Project ▾ ManteconSaulAPIPython ▾ Version control ▾ ManteconSaulAPIPython

Project ▾
  ▾ ManteconSaulAPIPython
    ▾ .venv library root
    ▾ app
      ▾ db
        __init__.py
        database.py
        models.py
      ▾ repository
        __init__.py
      ▾ routers
        __init__.py
        pista.py
        reserva.py
        usuario.py
        __init__.py
        schemas.py
        main.py
        requirements.txt
        API test_main.http
    ▾ External Libraries
    ▾ Scratches and Consoles

usuario.py x

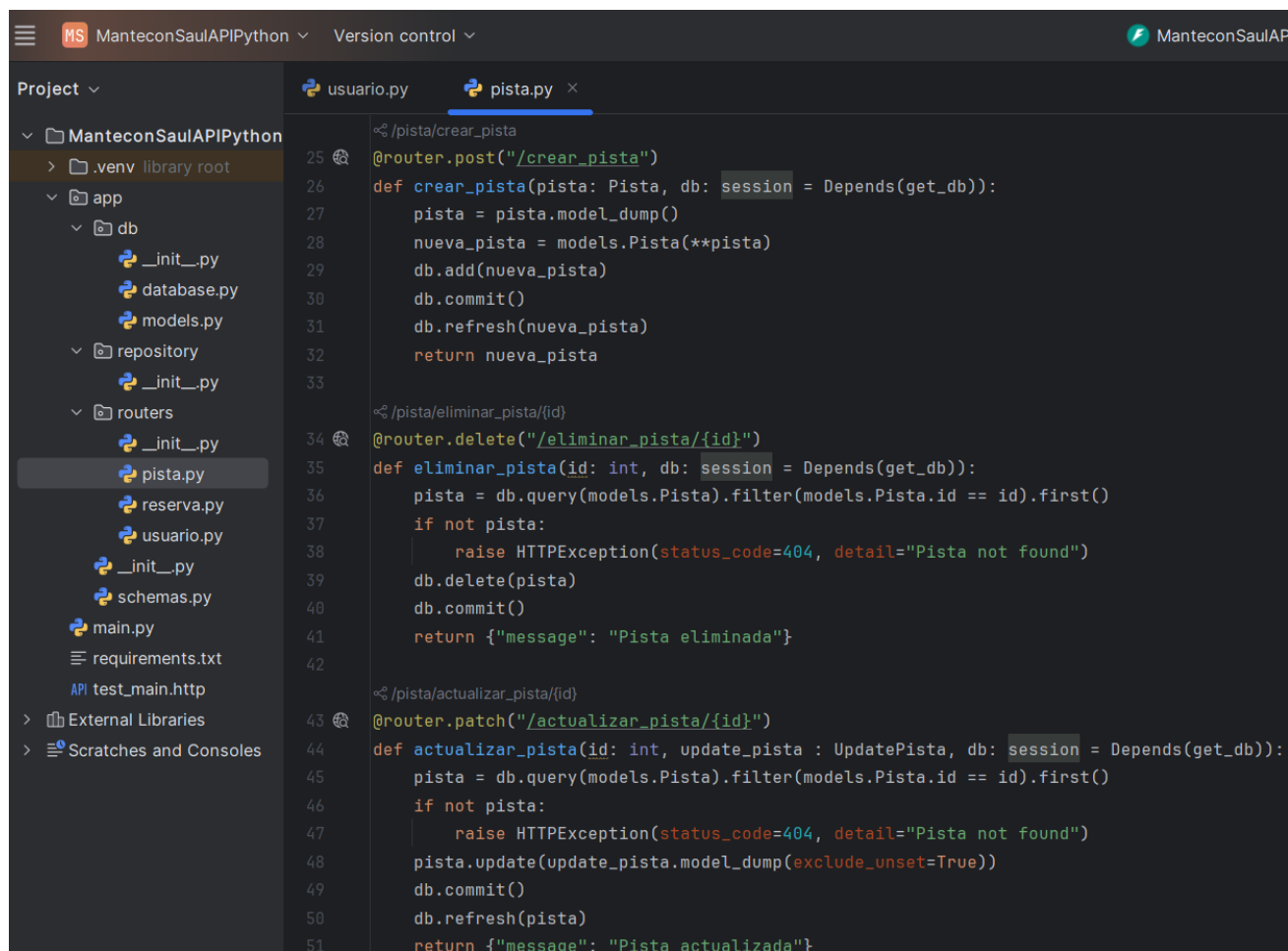
25 @router.post("/crear_usuario")
26 def crear_usuario(usuario: Usuario, db: session = Depends(get_db)):
27     usuario = usuario.model_dump()
28     nuevo_usuario = models.Usuario(**usuario)
29     db.add(nuevo_usuario)
30     db.commit()
31     db.refresh(nuevo_usuario)
32     return nuevo_usuario
33
34 @router.delete("/eliminar_usuario/{id}")
35 def eliminar_usuario(id: int, db: session = Depends(get_db)):
36     usuario = db.query(models.Usuario).filter(models.Usuario.id == id).first()
37     if not usuario:
38         raise HTTPException(status_code=404, detail="Usuario no encontrado")
39     db.delete(usuario)
40     db.commit()
41     return {"message": "Usuario eliminado"}
42
43 @router.patch("/actualizar_usuario/{id}")
44 def actualizar_usuario(id: int, update_usuario: UpdateUsuario, db: session = Depends(get_db)):
45     usuario = db.query(models.Usuario).filter(models.Usuario.id == id).first()
46     if not usuario:
47         raise HTTPException(status_code=404, detail="Usuario no encontrado")
48     usuario.update(update_usuario.model_dump(exclude_unset= True))
49     db.commit()
50     db.refresh(usuario)
51     return {"message": "Usuario actualizado"}
```

4.3.4. Routers.pista.



The screenshot shows a code editor with a project explorer on the left and a code editor on the right. The project explorer shows the file structure of the 'ManteconSaulAPIPython' project, with the 'pista.py' file selected under the 'routers' directory. The code editor displays the implementation of the 'pista' router, which includes imports for 'List', 'APIRouter', 'Depends', 'HTTPException', 'Session', and 'models'. The router is defined with a prefix of '/pista' and a tag of 'pistas'. It includes two endpoints: a GET endpoint for '/lista_pistas' that returns a list of 'Pista' objects, and a GET endpoint for '/pista/{id}' that returns a single 'Pista' object or raises a 404 error if not found.

```
1 from typing import List
2
3 from fastapi import APIRouter, Depends, HTTPException
4 from sqlalchemy.orm import Session, session
5 from app.db import models
6 from app.db.database import get_db
7 from app.schemas import Pista, UpdatePista
8
9 /pista
10
11 router = APIRouter(
12     prefix="/pista",
13     tags=["pistas"]
14 )
15
16 /pista/lista_pistas
17
18 @router.get("/lista_pistas")
19 def get_pistas(db: session = Depends(get_db)):
20     pistas = db.query(models.Pista).all()
21     return pistas
22
23 /pista/pista/{id}
24
25 @router.get("/pista/{id}")
26 def get_pista(id: int, db: session = Depends(get_db)):
27     pista = db.query(models.Pista).filter(models.Pista.id == id).first()
28     if not pista:
29         raise HTTPException(status_code=404, detail="Pista not found")
30     return pista
```

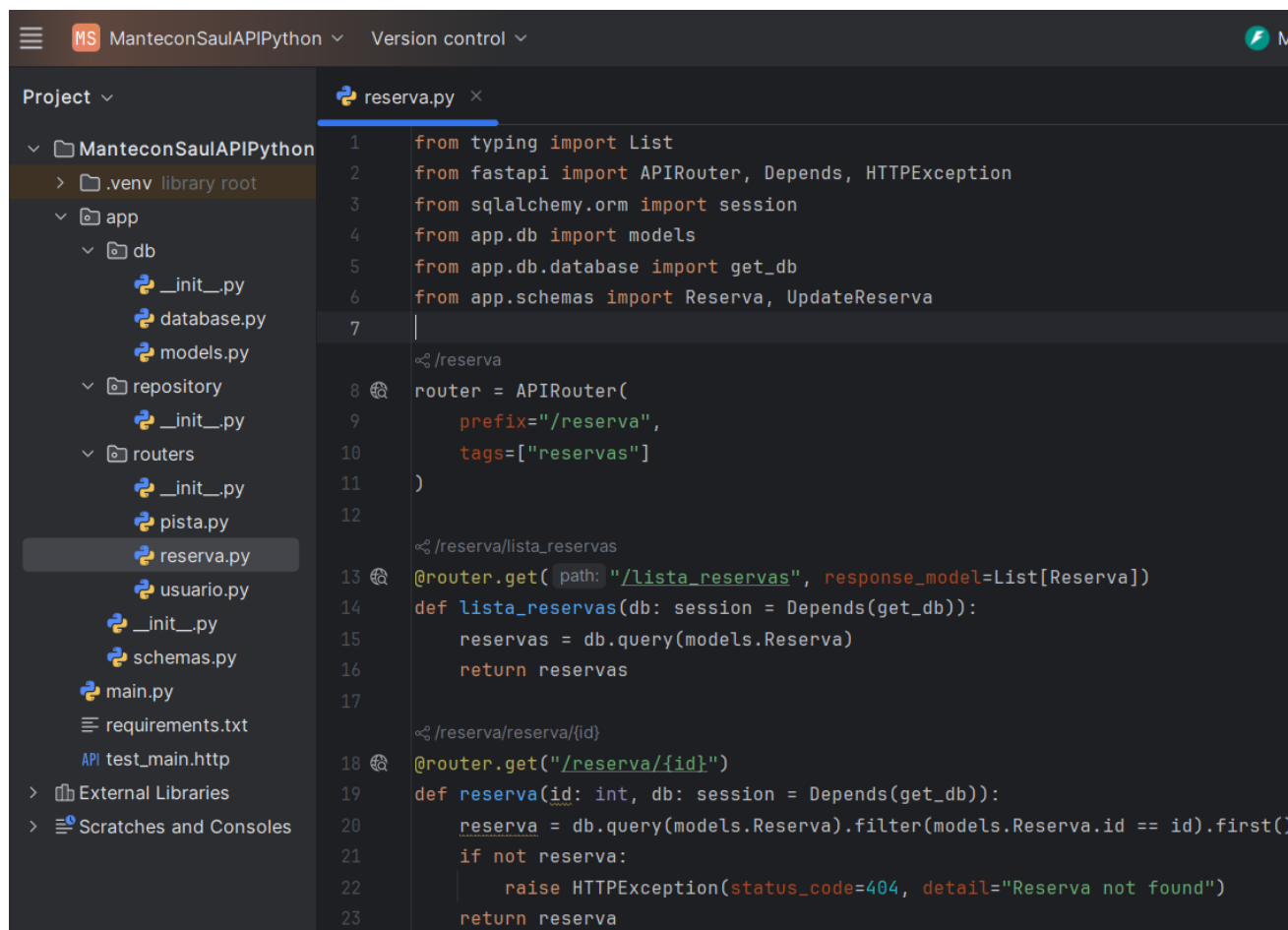


```
Project ▾
  ▾ ManteconSaulAPIPython
    > .venv library root
    ▾ app
      ▾ db
        __init__.py
        database.py
        models.py
      ▾ repository
        __init__.py
      ▾ routers
        __init__.py
        pista.py
        reserva.py
        usuario.py
        __init__.py
        schemas.py
        main.py
        requirements.txt
        API test_main.http
    > External Libraries
    > Scratches and Consoles

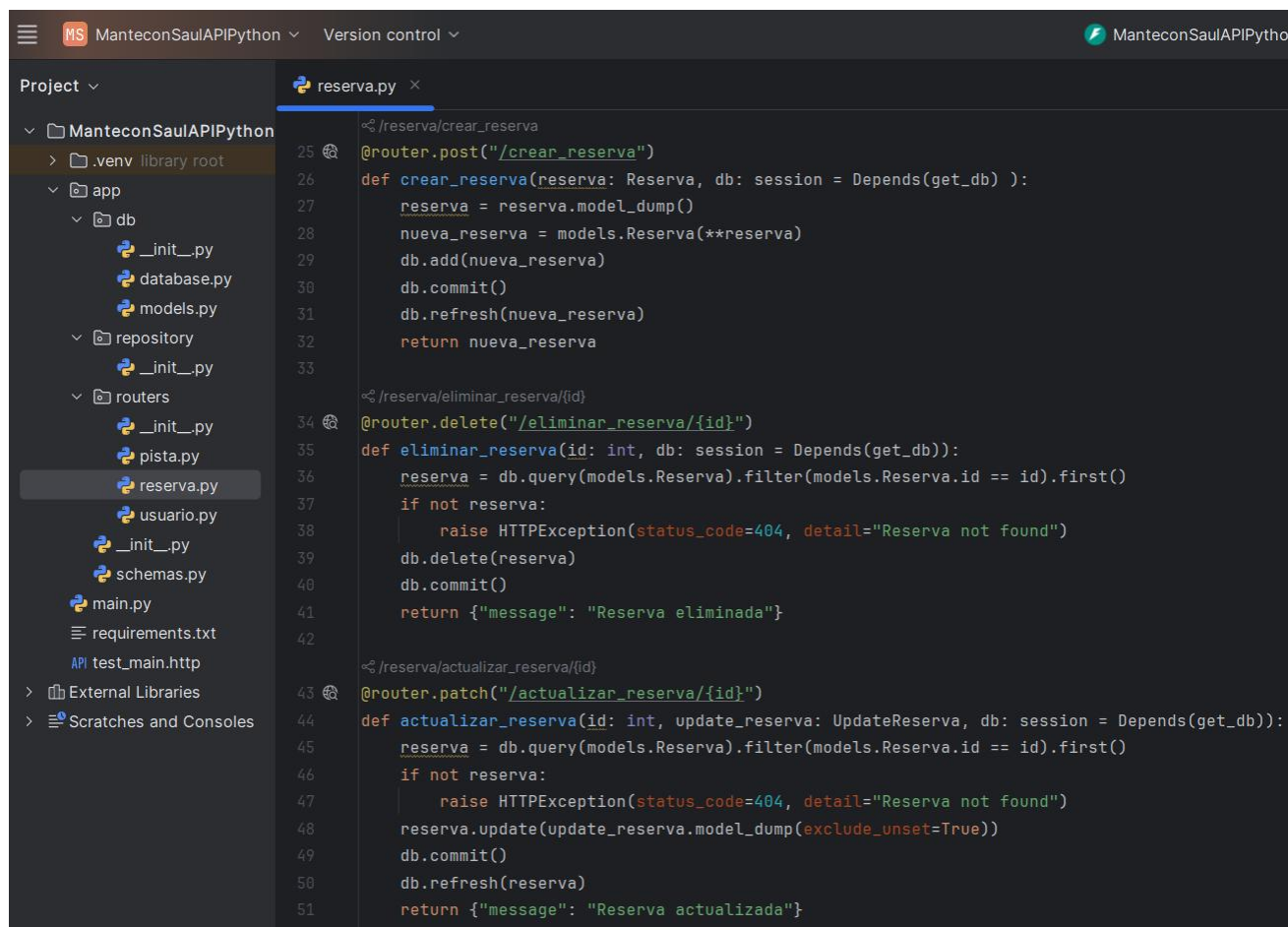
usuario.py  pista.py ×

< /pista/crear_pista
25 @router.post("/crear_pista")
26 def crear_pista(pista: Pista, db: session = Depends(get_db)):
27     pista = pista.model_dump()
28     nueva_pista = models.Pista(**pista)
29     db.add(nueva_pista)
30     db.commit()
31     db.refresh(nueva_pista)
32     return nueva_pista
33
< /pista/eliminar_pista/{id}
34 @router.delete("/eliminar_pista/{id}")
35 def eliminar_pista(id: int, db: session = Depends(get_db)):
36     pista = db.query(models.Pista).filter(models.Pista.id == id).first()
37     if not pista:
38         raise HTTPException(status_code=404, detail="Pista not found")
39     db.delete(pista)
40     db.commit()
41     return {"message": "Pista eliminada"}
42
< /pista/actualizar_pista/{id}
43 @router.patch("/actualizar_pista/{id}")
44 def actualizar_pista(id: int, update_pista : UpdatePista, db: session = Depends(get_db)):
45     pista = db.query(models.Pista).filter(models.Pista.id == id).first()
46     if not pista:
47         raise HTTPException(status_code=404, detail="Pista not found")
48     pista.update(update_pista.model_dump(exclude_unset=True))
49     db.commit()
50     db.refresh(pista)
51     return {"message": "Pista actualizada"}
```

4.3.5. Routers.reserva.



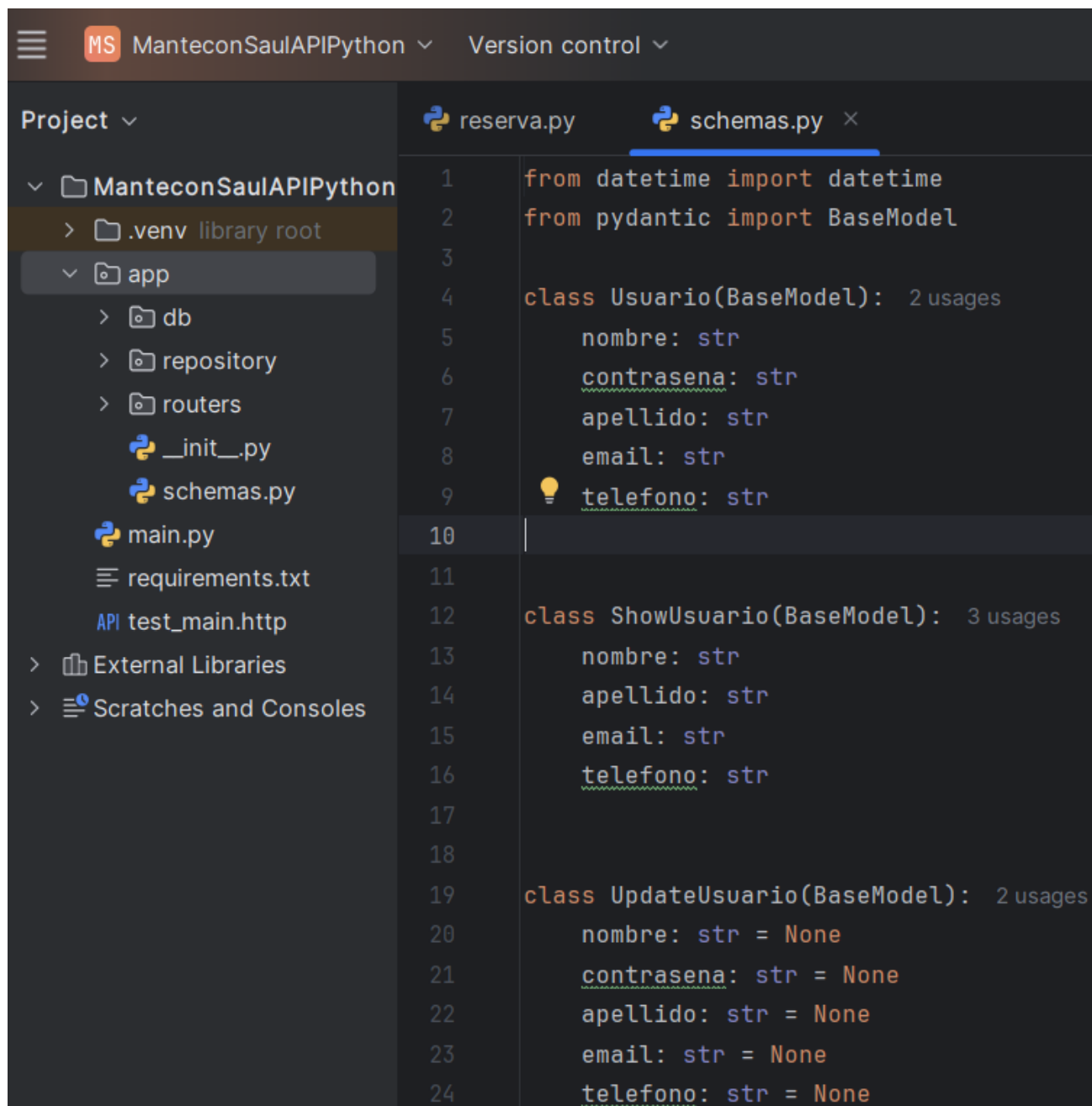
```
1 from typing import List
2 from fastapi import APIRouter, Depends, HTTPException
3 from sqlalchemy.orm import session
4 from app.db import models
5 from app.db.database import get_db
6 from app.schemas import Reserva, UpdateReserva
7
8 ~~~~~ /reserva
9 router = APIRouter(
10     prefix="/reserva",
11     tags=["reservas"]
12 )
13
14 ~~~~~ /reserva/lista_reservas
15 @router.get(path: "/lista_reservas", response_model=List[Reserva])
16 def lista_reservas(db: session = Depends(get_db)):
17     reservas = db.query(models.Reserva)
18     return reservas
19
20 ~~~~~ /reserva/reserva/{id}
21 @router.get("/reserva/{id}")
22 def reserva(id: int, db: session = Depends(get_db)):
23     reserva = db.query(models.Reserva).filter(models.Reserva.id == id).first()
24     if not reserva:
25         raise HTTPException(status_code=404, detail="Reserva not found")
26     return reserva
```



The screenshot displays a code editor interface for a Python project named 'ManteconSaulAPIPython'. The left sidebar shows the project structure, with the 'reserva.py' file selected under the 'routers' directory. The main editor area shows the code for three API endpoints: creating, deleting, and updating a reservation.

```
25 @router.post("/crear_reserva")
26 def crear_reserva(reserva: Reserva, db: session = Depends(get_db)):
27     reserva = reserva.model_dump()
28     nueva_reserva = models.Reserva(**reserva)
29     db.add(nueva_reserva)
30     db.commit()
31     db.refresh(nueva_reserva)
32     return nueva_reserva
33
34 @router.delete("/eliminar_reserva/{id}")
35 def eliminar_reserva(id: int, db: session = Depends(get_db)):
36     reserva = db.query(models.Reserva).filter(models.Reserva.id == id).first()
37     if not reserva:
38         raise HTTPException(status_code=404, detail="Reserva not found")
39     db.delete(reserva)
40     db.commit()
41     return {"message": "Reserva eliminada"}
42
43 @router.patch("/actualizar_reserva/{id}")
44 def actualizar_reserva(id: int, update_reserva: UpdateReserva, db: session = Depends(get_db)):
45     reserva = db.query(models.Reserva).filter(models.Reserva.id == id).first()
46     if not reserva:
47         raise HTTPException(status_code=404, detail="Reserva not found")
48     reserva.update(update_reserva.model_dump(exclude_unset=True))
49     db.commit()
50     db.refresh(reserva)
51     return {"message": "Reserva actualizada"}
```

4.3.6. Schemas.




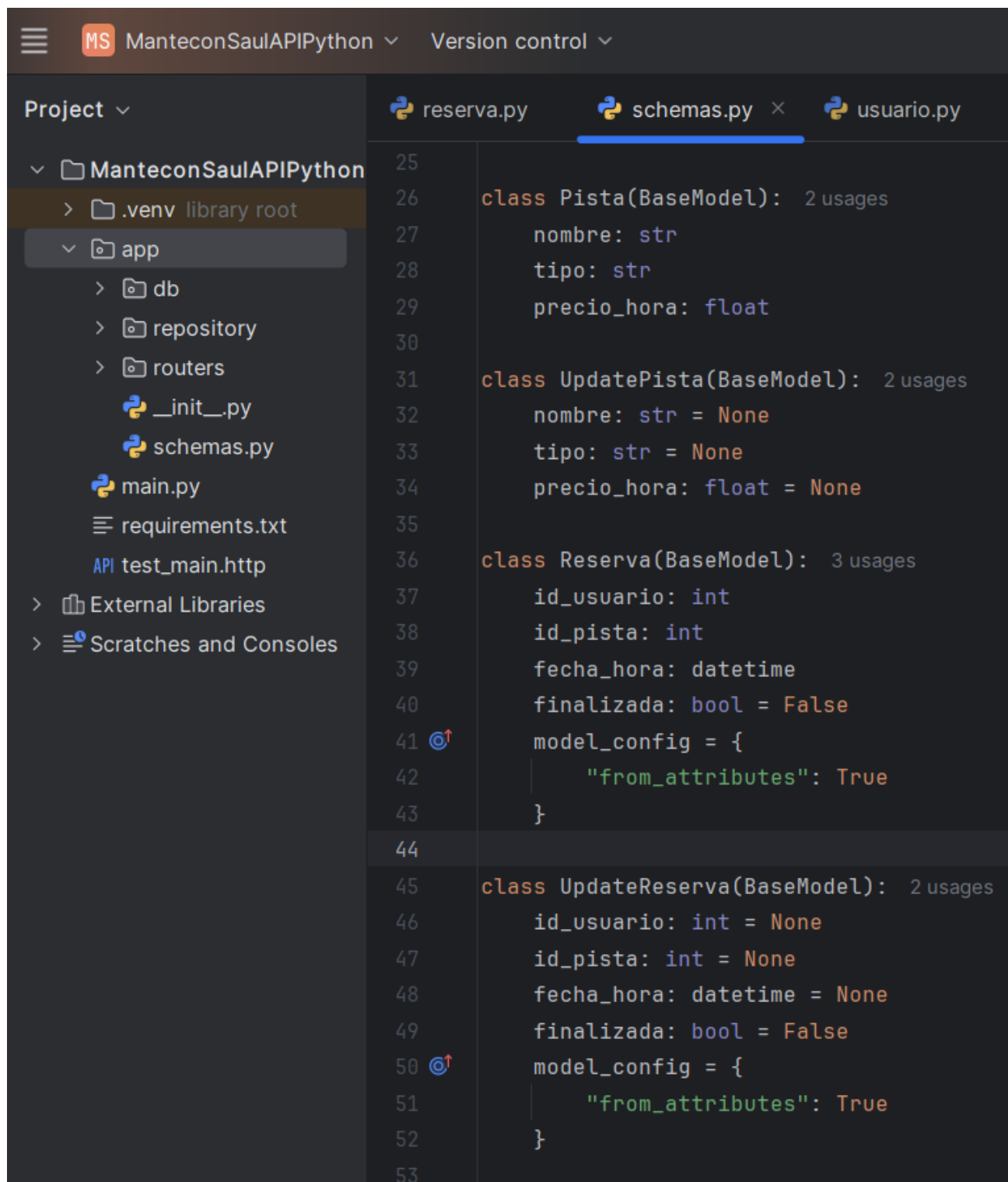
```
MS ManteconSaulAPIPython Version control
```

Project

- ▼ ManteconSaulAPIPython
 - > .venv library root
 - ▼ app
 - > db
 - > repository
 - > routers
 - __init__.py
 - schemas.py
 - main.py
 - requirements.txt
 - API test_main.http
 - > External Libraries
 - > Scratches and Consoles

reserva.py schemas.py

```
1 from datetime import datetime
2 from pydantic import BaseModel
3
4 class Usuario(BaseModel): 2 usages
5     nombre: str
6     contrasena: str
7     apellido: str
8     email: str
9      telefono: str
10
11
12 class ShowUsuario(BaseModel): 3 usages
13     nombre: str
14     apellido: str
15     email: str
16     telefono: str
17
18
19 class UpdateUsuario(BaseModel): 2 usages
20     nombre: str = None
21     contrasena: str = None
22     apellido: str = None
23     email: str = None
24     telefono: str = None
```



The screenshot shows a code editor interface for a project named 'ManteconSaulAPIPython'. The left sidebar displays the project structure, including a '.venv' library root and an 'app' directory with subdirectories 'db', 'repository', and 'routers'. Files listed include '__init__.py', 'schemas.py', 'main.py', 'requirements.txt', and 'test_main.http'. The main editor area shows the 'schemas.py' file with the following code:

```
25
26 class Pista(BaseModel): 2 usages
27     nombre: str
28     tipo: str
29     precio_hora: float
30
31 class UpdatePista(BaseModel): 2 usages
32     nombre: str = None
33     tipo: str = None
34     precio_hora: float = None
35
36 class Reserva(BaseModel): 3 usages
37     id_usuario: int
38     id_pista: int
39     fecha_hora: datetime
40     finalizada: bool = False
41     model_config = {
42         "from_attributes": True
43     }
44
45 class UpdateReserva(BaseModel): 2 usages
46     id_usuario: int = None
47     id_pista: int = None
48     fecha_hora: datetime = None
49     finalizada: bool = False
50     model_config = {
51         "from_attributes": True
52     }
53
```

4.3.7. Main.

```

1  from fastapi import FastAPI
2  import uvicorn
3  from app.routers import usuario, reserva, pista
4  from app.db.database import Base, engine
5
6  def create_tables(): 1 usage
7      Base.metadata.create_all(bind=engine)
8
9  create_tables()
10
11
12  /
13  app = FastAPI()
14  /usuario
15  app.include_router(usuario.router)
16  /pista
17  app.include_router(pista.router)
18  /reserva
19  app.include_router(reserva.router)

```

4.3.8. Requirements.txt.

```

1  fastapi
2  uvicorn
3  pydantic
4  psycpg2
5  SQLAlchemy

```


4.4. Pruebas.

4.4.1. Crear usuario.

POST **/usuario/crear_usuario** Crear Usuario

Parameters

No parameters

Request body required

```
{
  "nombre": "usuario1",
  "contrasena": "1234",
  "apellido": "usuariolapellido",
  "email": "usuariol@gmail.com",
  "telefono": "123456789"
}
```

Curl

```
curl -X 'POST' \
'http://127.0.0.1:8000/usuario/crear_usuario' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "nombre": "usuario1",
  "contrasena": "1234",
  "apellido": "usuariolapellido",
  "email": "usuario1@gmail.com",
  "telefono": "123456789"
}'
```

Request URL

`http://127.0.0.1:8000/usuario/crear_usuario`

Server response

Code	Details
200	<p>Response body</p> <pre>{ "nombre": "usuario1", "email": "usuario1@gmail.com", "apellido": "usuariolapellido", "contrasena": "1234", "id": 2, "telefono": "123456789" }</pre> <p>Response headers</p> <pre>content-length: 130 content-type: application/json date: Tue, 11 Feb 2025 00:34:44 GMT server: uvicorn</pre>

Responses

Code	Description
200	Successful Response

4.4.2. Crear pista.

POST /pista/crear_pista Crear Pista

Parameters

No parameters

Request body required

```
{
  "nombre": "pista de padell",
  "tipo": "padel",
  "precio_hora": 333
}
```

Execute

Curl

```
curl -X 'POST' \
'http://127.0.0.1:8000/pista/crear_pista' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "nombre": "pista de padell",
  "tipo": "padel",
  "precio_hora": 333
}'
```

Request URL

```
http://127.0.0.1:8000/pista/crear_pista
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "tipo": "padel", "id": 2, "nombre": "pista de padell", "precio_hora": 333 }</pre> <p>Response headers</p> <pre>content-length: 70 content-type: application/json date: Tue, 11 Feb 2025 00:37:04 GMT server: uvicorn</pre>

Responses

Code	Description
200	Successful Response

4.4.3. Crear reserva.

POST /reserva/crear_reserva Crear Reserva

Parameters

No parameters

Request body **required**

```
{
  "id_usuario": 2,
  "id_pista": 2,
  "fecha_hora": "2025-03-11",
  "finalizada": false
}
```

Execute

Curl

```
curl -X 'POST' \
'http://127.0.0.1:8000/reserva/crear_reserva' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "id_usuario": 2,
  "id_pista": 2,
  "fecha_hora": "2025-03-11",
  "finalizada": false
}'
```

Request URL

`http://127.0.0.1:8000/reserva/crear_reserva`

Server response

Code	Details
200	<div>Response body</div> <div><pre>{ "id_usuario": 2, "id": 3, "finalizada": false, "id_pista": 2, "fecha_hora": "2025-03-11T00:00:00" }</pre></div> <div>Response headers</div> <div><pre>content-length: 90 content-type: application/json date: Tue, 11 Feb 2025 00:39:33 GMT server: uvicorn</pre></div>

Responses

Code	Description
200	Successful Response

4.4.4. Delete Cascade.

Al eliminar un usuario o una pista, se borrará automáticamente la reserva.

The image displays two screenshots of REST client tools, likely Swagger UI or Postman, illustrating API endpoints for a system.

Left Screenshot (DELETE endpoint):

- Method:** DELETE
- Endpoint:** /usuario/eliminar_usuario/{id} Eliminar Usuario
- Parameters:** A table with columns 'Name' and 'Description'. The 'id' parameter is listed as 'integer (path)' and is marked as 'required'. The value '2' is entered in the input field.
- Execute:** A blue button to execute the request.
- Responses:** A section showing the response details.
 - Code:** 200
 - Response body:** { "message": "Usuario eliminado" }

Right Screenshot (GET endpoint):

- Method:** GET
- Endpoint:** /reserva/lista_reservas Lista Reservas
- Parameters:** No parameters.
- Execute:** A blue button to execute the request.
- Responses:** A section showing the response details.
 - Code:** 200
 - Response body:** []
 - Response headers:** content-length: 2, content-type: application/json, date: Tue, 11 Feb 2025 00:42:43 GMT, server: uvicorn

4.5. Despliegue de la aplicación.

La aplicación puede desplegarse en un servidor local o en la nube mediante Docker.

5. CONCLUSIONES Y POSIBLES AMPLIACIONES.

Las principales dificultades en el desarrollo de este proyecto han estado marcadas por la falta de tiempo y la escasa referencia disponible para tomar como guía. Esto me ha generado muchas dudas, especialmente al no comprender completamente algunos parámetros, lo que me ha llevado a cometer errores y a invertir bastante tiempo en resolverlos.

Sin embargo, como aprendizaje, este proyecto me ha permitido adquirir conocimientos sobre la creación de una API en Python con FastAPI. Ahora puedo afirmar que sé desarrollarlas tanto en este lenguaje como en Java, aunque sigo sin saber como implementarlas en una aplicación.

En cuanto a futuras ampliaciones, la prioridad sería precisamente esa: aprender a integrar esta API en un entorno de aplicación real, lo que completaría el proceso de desarrollo y la haría funcional en un escenario práctico.

6. BIBLIOGRAFÍA.

Apuntes de clase.

<https://es.linkedin.com/pulse/fastapi-vs-django-flask-qu%C3%A9-framework-aprender-agustin-marchi>