PRÁCTICA 3

PERMANENCIA DE DATOS CON UNA BASE DE DATOS

Equipo:
Beltrán Saucedo Axel Alejandro
Cerón Samperio Lizeth Montserrat
Higuera Pineda Angel Abraham
Lorenzo Silva Abad Rey
4BV1

ESCUELA SUPERIOR DE CÓMPUTO

TECNOLOGÍAS PARA EL DESARROLLO DE APLICACIONES WEB

13/10/2025

1. Introducción

Planteamiento del problema

El problema que se aborda es la falta de persistencia de datos en la aplicación. La aplicación guarda los datos en la memoria mientras está funcionando, pero se pierde al reiniciar la aplicación. Se necesita una solución que garantice lo siguiente:

- La información de cada item se almacena de forma permanente.
- Los usuarios pueden recuperar, actualizar o eliminar estos items en cualquier momento.
- La aplicación mantenga un registro organizado y estructurado de todos los paquetes.

Propuesta de solución

La solución propuesta es el desarrollo de una API RESTful utilizando FastAPI para la lógica de negocio y SQLModel para la gestión de la base de datos , permitiendo así la persistencia de datos. Componentes clave de la solución:

- **Framework Web (FastAPI):** Se utiliza para construir la interfaz de la API, definiendo las rutas (URLs) que permitirán a los usuarios interactuar con la información de los paquetes. Un framework web moderno y rápido para construir APIs con Python 3.7+ basado en estándares como OpenAPI y JSON Schema.
- **SQLModel:** Se emplea para definir la estructura de la información (qué es un "ítem", qué campos tiene, y cuál es su identificador único).
- **Base de datos SQLite:** Se utiliza un motor de base de datos SQLite (database.db) para almacenar los datos en un archivo físico. De esta manera, la información perdura aunque la aplicación se detenga o se reinicie.
- **SQLModel Engine y Session:** Se configura un motor de conexión (engine) y un sistema de sesiones (SessionDep) para establecer una comunicación eficiente y segura entre la aplicación y el archivo de la base de datos.

2. Implementación

Código:

En esta seccion tendremos los nuevos modulos que usamos para el funcionamiento de la practica.

Debemos entender la diferencia entre esta practica y la anterior, ya que pueden parecer iguales, pero tienen una diferencia crucial. En la practica anterior, nuestra API gestionaba los datos de los items en una lista en memoria, lo cual tenia una principal desventaja, que

al reiniciar nuestro servidor, todos los datos se perdian. En cambio, esta vez integramos una base de datos para asegurar la permanencia de los datos.

```python
#FastApi es el modulo principal, HTTPException es un manejador de errores, como el "no encontrado"
#Status es un modulo de indentificacion de codigos de estado HTTP, como el clasico 404
#Depends es un mecanismo para poder inyectar dependencias
from fastapi import FastAPI, HTTPException, status, Depends
#Field es usado para definir las propieddes de los campos del modelo
#create_engine se usa para crear un motor que conectara la base de datos con la aplicacion
#Session es el enncargado de manejar la comunicacion con la base de datos
#SQLModel sera la base para los modelos de datos
from sqlmodel import Field, create_engine, Session, SQLModel
#Optional y List son tipos de datos que permiten definir campos opcionales y listas
#Se usan para definir los modelos de datos y las respuestas de la API
from typing import Optional, List
#Annotated es una forma para escribir tipos con metadatos
from typing_extensions import Annotated
```

El principal cambio que podemos ver aqui es la adicion de la libreria SQLModel, la cual nos permitira interactuar con la base de datos, gracias a sus capacidades provenientes de SQLAlchemy y Pydantic.

```python
#Item contiene el indentificador unico (id) y hereda los campos de ItemBase
#y con table=True se marca como una tabla
class Item(ItemBase, table=True):
    id: Optional[int]=Field(default=None, primary_key=True)
#Los campos son opcionales, ya que podemos querer actualizar solo uno de ellos
#El valor por defecto es None
#primary_key marca el campo como la clave primaria(ID)
```

En esta parte, podemos ver que los modelos de datos fueron redefinidos usando SQLModel y asi poder mapearlos en una tabla en nuestra base de datos. Se agrego 'table=true', que le indica a SQLModel que la clase esta representando una tabla en la base. Otro cambio es en id, que ahora lo definimos como la clave primaria de nuestra tabla. Esto hara que la base de datos se encargue de ir generando los id's de forma automatica.

```python
#ItemOut nos ayudara a colocar en el orden deseado nuestras respuestas
class ItemOut(SQLModel):
    peso: float
    ganancia: float
    id: int
```

Esta funcion tiene el unico proposito de mostrarnos los datos en el orden deseado, el cual es peso/ganancia/id, ya que originalmente los mostraba en desorden.

```
# Conexion a la base de datos

#La cadena sqlite:///database.db indica que vamos a usar un archivo local llamda 'database.db'
#como nuestra base de datos
sql_url="sqlite:///databse.db"
#Crea el motor de la base de datos, que sera el encargado de gestionar la conexion
engine=create_engine(sql_url)
```

Aqui tenemos la nueva adicion mas importante, la cual se encargara de la configuracion y manejo de la conexion con la base de datos (que estara en SQLite). Esta sera almacenada en un archivo bajo el nombre de database.db. Tambien tenemos lo que vendria a ser nuestro 'motor', por decirlo de una forma, que funcionara como el centro de comunicaciones entre la API y nuestro database.db

```
#Funcion encarga de crear la tabla 'item' en la base de datos
def create_db_and_tables():
    #SQLModel.metadata... sera el que checara que todos los modelos que heredan de SQLModel,
    #y que ademas esten marcado con table=True, y les creara sus tablas en la base de datos
    SQLModel.metadata.create_all(engine)
```

Esta funcion sera la encargada de la creacion de las tablas y solo se ejecutara una vez al iniciar la aplicacion. Crea las tablas correspondientes en la base si es que no existen.

```
#Funcion encargada de crear y proporcionar una sesion de base de datos por cada peticion
def get_session():
    #Abre una nueva sesion con el motor(engine) de la base de datos
    with Session(engine) as session:
        #Yield dara la sesion al endpoint de nuestra API
        yield session
#SessionDep es una anotacion que usa a Depends para inyectar una sesion de base de datos
#en nuestras funciones
SessionDep=Annotated[Session, Depends(get_session)]
```

Aqui implementamos un sistema de sesiones por peticion. Esto se refiere a que cada vez que la API recibe una solicitud, una nueva sesion de comunicacion con la base de datos se abre, y esta misma se cierra en automatico al terminar la solicitud, lo cual nos garantiza un manejo seguro de las conexiones.

```python
#On_event le dice a FastAPI que ejecute lo siguiente una sola vez cuando iniciamos la aplicacion
@app.on_event("startup")
def on_startup():
    #Llamamos a create_db... para tener la seguridad de que la base de datos y sus tablas ya fueron
    #creadas antes de que la aplicacion reciba peticiones
    create_db_and_tables()
```

Este evento se ejecuta en el instante en el momento en que la API se termine de iniciar,antes de que cualquier usuario mande una peticion y solo se ejecuta una vez. Y esto no garantiza que la base de datos y sus tablas ya existan antes de que los usuarios empiezen a mandar peticiones.