Taller: Desarrollo de una API RESTful con FastAPI, SQLAlchemy y Docker

1. Introducción

¿Qué es FastAPI?

FastAPI es un framework moderno, rápido (high-performance), basado en Python para construir APIs web. Su principal ventaja es la validación automática de tipos, documentación integrada y soporte para asincronía.

¿Qué es SQLAlchemy?

SQLAlchemy es una herramienta ORM (Object Relational Mapper) que permite mapear clases de Python a tablas en una base de datos, facilitando operaciones como inserciones, actualizaciones y consultas sin escribir SQL crudo.

¿Qué es Pydantic?

Pydantic permite validar datos de entrada/salida en FastAPI mediante modelos basados en clases. Asegura que los datos sean del tipo correcto y que cumplan con ciertas reglas.

2. Ejercicio

Generar una aplicación distribuida usando FastAPI, SQLAlchemy y Docker para el despliegue. A continuación se explican los detalles.

Tu equipo está desarrollando una API para una aplicación de recomendaciones donde los usuarios pueden agregar, listar y actualizar películas o series que han visto y recomendarían a otros.

Cada recomendación debe incluir:

- id: identificador único (autogenerado)
- titulo: nombre de la película o serie
- genero: categoría (ej. Acción, Drama, Ciencia Ficción)
- puntaje: calificación del 1 al 10
- comentario: opinión breve del usuario
- vista: booleano que indica si ya la vio

La API debe tener:

Método	Ruta	Acción
GET	/recomendaciones	Listar todas las recomendaciones
POST	/recomendaciones	Crear una nueva recomendación
GET	/recomendaciones/{id}	Obtener una recomendación por ID
PUT	/recomendaciones/{id}	Editar una recomendación
DELETE	/recomendaciones/{id}	Eliminar una recomendación

Modelo de datos

Una **recomendación** tiene los siguientes atributos:

Campo	Tipo	Requerido	Descripción
`id`	int	Auto	Identificador único (clave primaria)
`titulo`	str	Sí	Título de la película o serie
`genero`	str	Sí	Género/categoría (Ej. "Acción", "Comedia")
`puntaje`	int (1-10)	Sí	Calificación que el usuario le da
`comentario`	str	Sí	Opinión personal o breve reseña
`vista`	bool	No	Si ya ha sido vista (`true` o `false`)

Se debe crear una base de datos con PostgreSQL.

- La tabla se llama `recomendaciones`.
- La creación de la tabla se debe hacer automáticamente al iniciar FastAPI si no existe

Casos de uso y funcionamiento esperado

1. `GET /recomendaciones`

Funcionalidad:
Devuelve una **lista con todas las recomendaciones** guardadas.

`2. POST /recomendaciones`

Funcionalidad:
Crea una **nueva recomendación** con los datos recibidos en el cuerpo del request.

Validaciones:

- `puntaje` debe ser un entero entre 1 y 10.
- Todos los campos son obligatorios excepto `vista` (default: false).
- El backend debe validar y retornar errores si los datos no cumplen con el modelo.

3. `GET /recomendaciones/{id}`

Funcionalidad:
Devuelve una **recomendación específica** usando su `id`.

4. `PUT /recomendaciones/{id}`

Funcionalidad:
Actualiza **todos los campos** de una recomendación existente.

5. `DELETE /recomendaciones/{id}`

Funcionalidad:
Elimina una recomendación por ID.

El código debe estar bien estructurado (separar modelos, esquemas, lógica de base de datos, rutas).

- Usar nombres descriptivos en funciones y variables.
- El endpoint raíz (`/`) puede devolver un mensaje de bienvenida.

El código debe respetar las normas de Clean Code

Clean Code: Buenas prácticas que deben aplicar

Los estudiantes deben seguir las siguientes **reglas y principios de Clean Code** en su implementación:

Nombres claros y descriptivos:

- Usar nombres que expresen la intención (`get_all_recommendations`, `create_recommendation`, `RecomendacionBase`).
- Evitar abreviaciones confusas o genéricas como `r1`, `data`, `temp`.

Funciones pequeñas y con una sola responsabilidad:

- Cada función debe hacer una sola cosa y hacerla bien.
- Separar lógica de base de datos (`crud.py`) de rutas (`main.py`), modelos (`models.py`) y validaciones (`schemas.py`).

Código modular y organizado:

- Separar archivos por responsabilidad (`app/models.py`, `app/crud.py`, `app/database.py`, etc.).
- No poner todo el código en un solo archivo.

Evitar duplicación:

- Si un fragmento de código se repite, considera convertirlo en función reutilizable.

Comentarios solo cuando son necesarios:

- El código debe explicarse a sí mismo; los comentarios solo deben usarse para aclarar partes complejas o decisiones importantes.

Manejo adecuado de errores:

- Devolver errores claros (`404`, `400`) con mensajes explicativos.
- Usar excepciones en lugar de "if" excesivos cuando sea más legible.

Formato consistente:

- Usar un formateador automático como `black` o `ruff` para mantener la consistencia del estilo.

- Indentación clara, líneas no demasiado largas, y separación lógica entre bloques de código.

Evaluación (10 puntos)

Criterio	Puntos
API funcional con endpoints completos (GET, POST, PUT, DELETE)	2
Modelado correcto con SQLAlchemy	2
Validación con Pydantic	2
Dockerfile funcional y reproducible	2
Uso de Docker Compose para orquestación	2