Documento de Diseño y Arquitectura

Prototipo de Página de Producto - Desafío Técnico

Edwin Fabian Vesga Escobar

26 de julio de 2025

Índice

1 Arquitectura y Decisiones de Diseño

1.1 Arquitectura General: Backend for Frontend (BFF) con Microservicios

La solución se diseñó bajo un patrón de microservicios, utilizando una arquitectura **Backend for Frontend (BFF)**. Esta consta de dos servicios independientes y containerizados:

1. **Servicio Backend (`backend`):** Una API RESTful pura y sin estado, construida con **FastAPI**. Su única responsabilidad es gestionar la lógica de negocio, el acceso a los datos y servir la información en formato JSON.
2. **Servicio Frontend (`frontend`):** Una aplicación web ligera, también construida con **FastAPI y plantillas Jinja2**. Su responsabilidad es renderizar la interfaz de usuario en el servidor (SSR), consumir la API del backend para obtener datos y presentarlos al usuario.

Justificación de la Elección

- **Separación de Responsabilidades:** Este desacoplamiento permite que el backend se enfoque exclusivamente en la lógica de datos, mientras que el frontend se enfoca en la presentación.
- **Escalabilidad Independiente:** Cada servicio puede ser escalado, actualizado o reemplazado de forma independiente sin afectar al otro.
- **Facilidad de Despliegue:** El uso de **Docker y Docker Compose** garantiza un entorno de desarrollo y producción consistente y reproducible con un solo comando, simplificando la configuración para cualquier desarrollador.

1.2 Diseño del Backend

- **Framework (FastAPI):** Se eligió FastAPI por su alto rendimiento, su sintaxis moderna basada en *type hints* de Python, la generación automática de documentación interactiva (Swagger UI) y su robusto sistema de validación de datos nativo con Pydantic.
- **Persistencia de Datos (Patrón de Repositorio):** Cumpliendo con el requisito de usar archivos locales, se implementó el **Patrón de Repositorio**. Esta capa abstrae completamente el origen de los datos (archivos JSON leídos con Pandas) de la lógica de la API. Esta decisión de diseño es crucial, ya que permitiría migrar a una base de datos relacional (como PostgreSQL) o NoSQL en el futuro con un impacto mínimo, cambiando únicamente la implementación del repositorio.
- **Robustez y Seguridad:** Se implementó un **rate limiting** (limitación de velocidad de peticiones) en los endpoints para proteger la API contra ataques de denegación de servicio (DoS) básicos. Además, se utiliza el sistema de excepciones de FastAPI (`HTTPException`) para devolver códigos de estado HTTP semánticos y mensajes de error claros (ej. 404 Not Found).

1.3 Diseño del Frontend

- **Tecnología (FastAPI + Jinja2):** Dado el enfoque del rol en el backend, se tomó la decisión pragmática de construir el frontend utilizando herramientas del ecosistema Python. Este enfoque de **renderizado en el servidor (SSR)** es ligero, rápido y demuestra la capacidad de entregar un producto funcional de extremo a extremo sin depender de frameworks complejos de JavaScript, lo cual es ideal para prototipos y aplicaciones donde la lógica de la interfaz no es excesivamente compleja.

2 Desafíos Técnicos y Soluciones

2.1 Desafío: Modelado de Datos para Productos Diversos

Problema: El requisito de manejar productos de categorías muy diferentes (Smartphones, Muebles, Papelería) presentaba el desafío de crear un modelo de datos que fuera a la vez consistente y flexible para atributos dispares. Un modelo plano resultaría en una gran cantidad de campos nulos.

Solución Implementada: Se optó por un **modelo de datos híbrido**. Se definió un conjunto de atributos comunes para todos los productos (ID, título, precio, marca, etc.) en el nivel superior del objeto JSON. Los atributos específicos de cada categoría se agruparon dentro de un objeto anidado llamado `specifications`. Este enfoque mantiene la estructura principal limpia, es fácilmente extensible para nuevas categorías y asegura que solo se almacene la información relevante para cada tipo de producto.

2.2 Desafío: Relevancia en las Recomendaciones de Productos

Problema: La implementación inicial del sistema de recomendaciones, basada únicamente en la similitud de texto (título y descripción), generaba resultados lógicamente incorrectos, como recomendar un armario al visualizar un tajalápiz.

Solución Implementada: Se rediseñó el algoritmo para seguir un proceso de dos pasos que imita mejor el razonamiento humano:

1. **Filtrado por Categoría:** El primer paso y el más importante es filtrar y considerar únicamente los productos que pertenecen a la **misma categoría** que el producto actual.
2. **Cálculo de Similitud de Contenido:** Una vez dentro de la categoría correcta, se aplica un modelo de **TF-IDF y Similitud de Coseno** sobre las características textuales (`title`, `description`) para encontrar los productos más similares dentro de ese subconjunto relevante.

Esta solución garantizó que las recomendaciones fueran siempre contextualmente apropiadas, mejorando drásticamente la calidad y la experiencia de usuario.