Sistema de Gestión de Inventario – Solución Full-Stack

Autor: Jairo Céspedes

Fecha: 2025-09-05

1. Introducción

Este documento describe la arquitectura, implementación y despliegue del Sistema de Gestión de Inventario construido como parte de la prueba técnica. La solución consta de un backend en FastAPI con base de datos PostgreSQL y un frontend en React (Vite + TypeScript) estilizado con Tailwind CSS. El proyecto demuestra una arquitectura en capas con patrones de repositorio y servicio, autenticación segura basada en JWT con control de roles (admin/usuario), un CRUD completo de productos, consultas avanzadas (búsqueda, filtrado y ordenamiento), pruebas automatizadas y una dockerización completa.

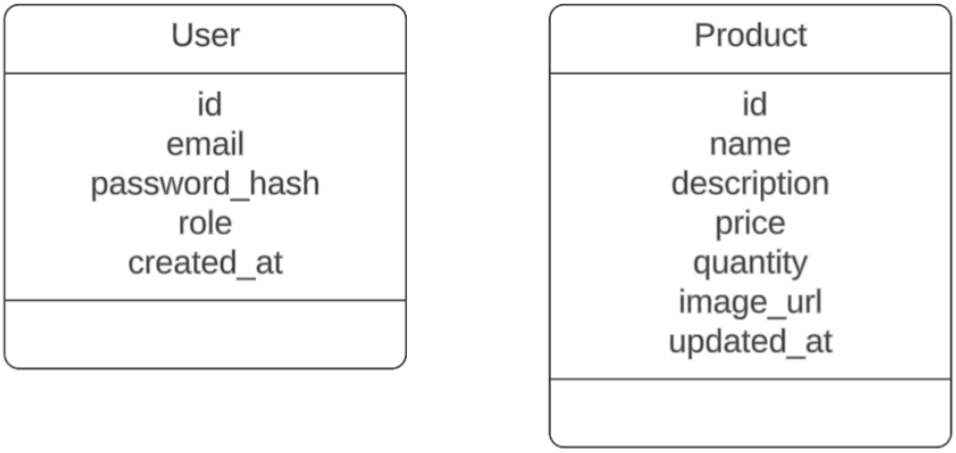
2. Resumen de Requerimientos de la Prueba

La prueba solicitó una aplicación full-stack para gestionar un inventario de productos con acceso autenticado. Los requerimientos incluían autenticación de usuarios, CRUD de productos, modelos de datos bien definidos y una separación clara de responsabilidades mediante el patrón de repositorio. En el frontend, se requería una interfaz responsiva con login/logout, dashboard de inventario y controles de CRUD. Los extras opcionales incluían un sistema de diseño (Tailwind), manejo de imágenes por URL, funciones avanzadas de búsqueda/filtro/ordenamiento, manejo de estado y pruebas automatizadas.

3. Arquitectura del Sistema

El sistema se divide en dos capas principales: Backend API y Frontend UI. Docker Compose orquesta tres servicios: PostgreSQL, backend y frontend (servido mediante Nginx). El backend expone endpoints REST protegidos con tokens JWT Bearer; el frontend los consume a través de Axios. Un healthcheck asegura que la base de datos esté lista antes de iniciar la API, y un inicializador ligero crea las tablas al inicio en entornos de desarrollo.

Diagrama UML (Modelo de Dominio):



4. Backend (FastAPI + PostgreSQL)

Tecnologías principales: FastAPI, SQLAlchemy, Pydantic, Passlib/Bcrypt, JWT, PyTest.

4.1 Estructura y Capas

- Modelos: definidos con SQLAlchemy ORM (Usuario y Producto).
- Esquemas: modelos Pydantic para validación de requests y responses.
- Repositorios: capa de acceso a datos, implementa el Patrón Repository.
- Servicios: lógica de negocio, orquesta repositorios y esquemas.
- Rutas API: routers que exponen endpoints y delegan en servicios.
- Dependencias: funciones inyectadas en FastAPI para DB session y autenticación.

4.2 Modelos de Datos

Usuario: id, email, password_hash, rol (admin|user), created_at.
Producto: id, nombre, descripción, precio, cantidad, image_url, updated_at.

4.3 Autenticación y Autorización (RBAC)

- Tokens JWT (HS256) enviados como Authorization Bearer.
- Claims: sub (id de usuario), role (admin|user).
- Contraseñas protegidas con hash bcrypt.
- Dependencias: extracción del usuario actual y validación de rol para RBAC.

4.4 Endpoints Principales

Método	Ruta	Descripción	Auth / Rol
POST	/auth/register	Registrar nuevo usuario	Público
POST	/auth/login	Iniciar sesión y	Público

		obtener JWT	
GET	/products	Listar productos con búsqueda/filtro/orden	Bearer (cualquiera)
POST	/products	Crear producto	Bearer (admin)
GET	/products/{id}	Obtener un producto	Bearer (cualquiera)
PUT	/products/{id}	Actualizar producto	Bearer (admin)
DELETE	/products/{id}	Eliminar producto	Bearer (admin)

Parámetros de query soportados en GET /products:

```
q, min_price, max_price, min_qty, has_image, sort_by
(name|price|quantity|updated_at), sort_dir (asc|desc)
```

4.5 Patrón Repository y Capa de Servicios

- El ProductRepository encapsula toda la lógica de acceso a datos. El ProductService adapta entidades ORM a DTOs Pydantic y centraliza validaciones. Esto permite mantener una arquitectura limpia y escalable.

- Para el usuario, tenemos el UserRepository que maneja la persistencia y consultas de usuarios (registro, búsqueda por email, validación de credenciales). El UserService se encarga de aplicar las reglas de negocio como el hashing de contraseñas, la generación de tokens JWT y la verificación de roles. Este diseño garantiza que la lógica de seguridad y autenticación esté desacoplada del acceso a datos.

4.6 Pruebas Automatizadas

Se implementaron pruebas automatizadas con PyTest:
• Pruebas de repositorios: búsqueda, filtros, ordenamientos.
• Pruebas de API: flujo de registro/login, permisos según rol, CRUD de productos.
• Configuración de BD de prueba con SQLite en memoria y fixtures.

4.7 Estilo de Código

El código sigue las guías de PEP8, con docstrings y type hints para mayor claridad.

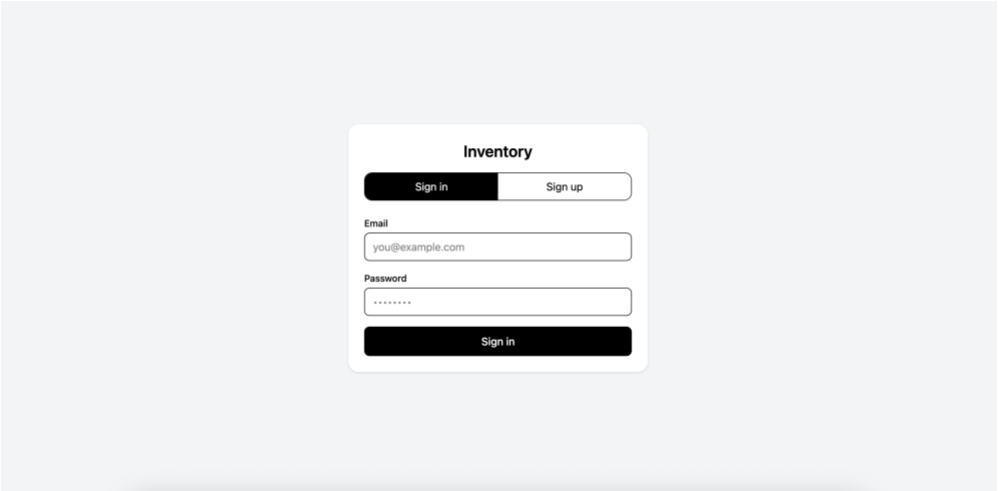
5. Frontend (React + Vite + TypeScript + Tailwind CSS)

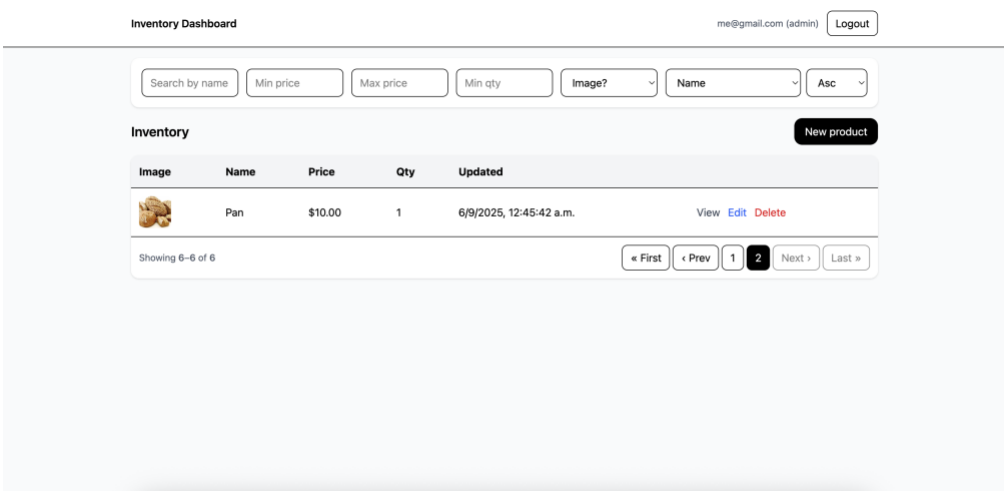
Librerías principales: React 18, Vite, TypeScript, Axios, Zustand, react-hook-form + zod, Tailwind CSS.

5.1 Funcionalidades

• Login y registro con selección de rol.
• Dashboard con tabla responsiva de productos.

• CRUD completo: crear, editar y ver detalles mediante modales.
• Búsqueda, filtrado y ordenamiento de productos.
• Paginación local cuando hay más de 5 productos.
• Estilo profesional consistente con Tailwind.





5.2 Manejo de Estado

Zustand administra el estado de autenticación y productos. Axios añade el token en headers.

5.3 Validación y Tipado

react-hook-form junto con zod garantizan validación robusta de formularios, TypeScript asegura tipado seguro.

5.4 Configuración de API

En desarrollo, VITE_API_URL apunta al backend local. En Docker, Nginx proxyea /api al backend.

5.5 Patrón Repository en el Frontend

En el frontend también se implementó el Patrón Repository, creando una capa específica para centralizar la comunicación con la API y desacoplarla de los componentes de UI y del manejo de estado:

- auth.repo.ts: encapsula las llamadas relacionadas con autenticación, incluyendo login, registro de usuarios y almacenamiento del token JWT.
- products.repo.ts: concentra todas las operaciones del CRUD de productos (listar, buscar, crear, actualizar, eliminar) y maneja la construcción de requests HTTP hacia el backend.

Ventajas del enfoque:

- Desacopla la lógica de acceso a datos de los componentes visuales.
- Permite que la UI y el store (Zustand) dependan de una interfaz clara y reutilizable.
- Facilita pruebas unitarias y potenciales cambios en la fuente de datos (ejemplo: reemplazar fetch por otra librería o añadir caché).

De esta manera, tanto en el backend como en el frontend, el patrón Repository asegura una arquitectura más limpia, escalable y mantenible.

6. Dockerización

Docker Compose orquesta los tres servicios: base de datos PostgreSQL, backend FastAPI y frontend Nginx. La base de datos tiene healthcheck; el backend espera a que esté lista y ejecuta init_db(); el frontend se sirve con Nginx y proxyea /api al backend.

6.1 Backend Dockerfile

```
FROM python:3.11-slim ... (instala dependencias, ejecuta init_db y uvicorn)
```

6.2 Frontend Dockerfile

```
FROM node:20-alpine AS build ... (npm run build, luego Nginx sirve los estáticos)
```

6.3 Configuración de Nginx

```
server { listen 80; location /api/ { proxy_pass http://backend:8000/; } }
```

6.4 docker-compose.yml

```
version: '3.9' ... incluye db, backend y frontend con sus dependencias y puertos.
```

7. Cómo Ejecutar

Modo desarrollo:

```
uvicorn app.main:app --reload
npm run dev
```

Con Docker:

```
docker compose up --build
Frontend: http://localhost:5173
Backend: http://localhost:8000/docs
```

8. Patrones y Buenas Prácticas

- Repository Pattern: aísla acceso a datos.
- Service Layer: centraliza lógica de negocio.
- DTOs con Pydantic: validación robusta.
- Inyección de dependencias en FastAPI.
- RBAC: control de acceso por roles.
- Separación clara de capas.
- Pruebas automatizadas.

9. Funcionalidades Opcionales Implementadas

- Tailwind CSS.
- URL de imágenes para productos.
- Búsqueda/filtros/ordenamiento.
- Zustand para manejo de estado.
- Paginación local.

10. Futuras Mejoras

- Paginación en backend.
- Migraciones con Alembic.
- CI/CD.
- Mejor manejo de errores y logging.
- Subida de imágenes a S3/GCS.