

Trabajo Programación Entorno Servidor 2º DAW

Por Samuel Gómez, Carlos Cortés, Victor Marín y Adrián Herrero

Trabajo de Entorno Servidor para una Tienda de Productos



Índice

<u>Objetivo del proyecto</u>	3
<u>Alcance</u>	4
<u>Tecnologías empleadas</u>	4
Arquitectura del sistema	5
1. Estructura general del sistema.....	5
2. Capas del sistema.....	5
<u>Capa de Controladores (Controllers)</u>	5
<u>Capa de Servicios (Services)</u>	6
<u>Capa de Repositorios (Repositories)</u>	6
<u>Capa de Modelo (Entities / Documents / DTOs)</u>	6
3. Componentes principales de la arquitectura.....	7
<u>API REST</u>	7
<u>API GraphQL</u>	7
<u>WebSocket</u>	7
<u>Bases de datos</u>	8
4. Flujo general de funcionamiento.....	8
<u>Patrones Utilizados</u>	9
1. Patrón CRUD.....	9
2. Patrón DTO (Data Transfer Object).....	9
<u>Tecnologías</u>	10
<u>Planificación Trello</u>	11
<u>Requisitos funcionales y no funcionales</u>	12
<u>Requisitos Funcionales</u>	12
<u>Requisitos No Funcionales</u>	12
<u>Principios SOLID aplicados</u>	14
<u>Principio de responsabilidad única (SRP)</u>	14
<u>Principio de abierto/cerrado (OCP)</u>	14
<u>Principio de sustitución de Liskov (LSP)</u>	14
<u>Principio de segregación de interfaces (ISP)</u>	14
<u>Principio de inversión de dependencias (DIP)</u>	14
Diseño de la base de datos	15
1. Base de Datos Relacional: PostgreSQL.....	15
2. Base de Datos NoSQL: MongoDB.....	15
<u>API REST / GraphQL</u>	16
<u>WebSocket</u>	17
<u>Docker</u>	18
<u>Modelos</u>	19
<u>Diagramas entidad-relación</u>	20
<u>Gitflow</u>	21
<u>Estimación de costes</u>	22
Conclusiones	23

Introducción

Objetivo del proyecto

Este proyecto tiene como fin el desarrollo de una moderna plataforma de tienda en línea, que emplea Java 25 y Spring Boot, cuyo propósito es brindar un sistema sólido, escalable y sencillo de extender para administrar y comercializar productos. La aplicación ofrece sus funcionalidades a través de dos interfaces de acceso:

- Una API REST para las operaciones CRUD que se basan en HTTP.
- Una API GraphQL, que posibilita la realización de consultas más efectivas y dinámicas en función de lo que el cliente necesite, evitando el overfetching.

El sistema integra el empleo de dos tecnologías de almacenamiento que se complementan entre sí:

PostgreSQL, una base de datos relacional que es dinámica y potente, se emplea para gestionar de manera estructurada el catálogo de productos y otros elementos con relaciones definidas. Está preparada para la concurrencia, es decir, para manejar sin errores peticiones simultáneas que tienen el potencial de causar inconsistencias en los datos.

MongoDB es una base de datos NoSQL del tipo documento. Se utiliza para guardar información que necesita una estructura dinámica o flexible, como los datos auxiliares, los históricos o la información que no es necesariamente relacional. En nuestro caso, ha sido empleada para almacenar la información de los pedidos.

Asimismo, el proyecto incluye la comunicación en tiempo real a través de WebSocket, lo que hace posible que se notifique al cliente eventos como la creación, modificación o supresión de productos sin tener que actualizar la página ni realizar solicitudes periódicas. Esto permite un ambiente más interactivo y sensible.

Por último, la aplicación está totalmente lista para ser implementada por medio de Docker, lo que simplifica su distribución, ejecución en entornos aislados y recreación del entorno de desarrollo o producción sin configuraciones manuales complicadas. La contenerización garantiza que todos los servicios, como Spring Boot, Postgre, MongoDB y otros posibles servicios de soporte, puedan funcionar de manera consistente.

Alcance

- Alimentar un catálogo de productos que incluya nombre, precio, categoría, descripción e imagen.
- Permitir operaciones CRUD sobre los productos (crear, leer, actualizar, eliminar).
- Validar datos de entrada (por ejemplo: nombre no en blanco, precio mayor que cero, precio obligatorio, categoría obligatoria).
- Exponer la funcionalidad tanto vía REST como vía GraphQL.
- Integrar una capa de WebSocket para notificaciones o actualizaciones en tiempo real (por ejemplo: “nuevo producto añadido”, “producto actualizado”).
- Contenerización mediante Docker para facilitar el despliegue.
- Uso de Postgre como base relacional concurrente y MongoDB como base de datos no relacional para casos concretos (por ejemplo: historial de cambios, logs, o documentos de producto con estructura variable).

Tecnologías empleadas

- Lenguaje: Java 25.
- Framework principal: Spring Boot.
- Bases de datos: Postgre (relacional concurrente), MongoDB (no relacional).
- API: REST (endpoints HTTP) y GraphQL (queries y mutations).
- WebSocket para comunicación en tiempo real.
- Contenerización: Docker / docker-compose.
- Entorno de desarrollo: IntelliJ IDEA.
- Otros: (Redis, JUnit, Dokka).

Arquitectura del sistema

La arquitectura del sistema se fundamenta en un enfoque que es modular y escalable, el cual fusiona diversas tecnologías y paradigmas con la finalidad de proporcionar un ambiente resiliente, adaptable y de fácil mantenimiento. El proyecto está basado en Spring Boot, que es el núcleo principal, y se apoya en diversos elementos adicionales que funcionan de manera integrada para satisfacer las exigencias técnicas y comerciales.

1. Estructura general del sistema

El sistema sigue una arquitectura cliente–servidor donde:

- El Servidor (Backend)
 - Gestiona la lógica de negocio.
 - Expone los servicios mediante REST y GraphQL.
 - Administra conexiones con las bases de datos Postgre y MongoDB.
 - Emite eventos en tiempo real mediante WebSocket.
 - Proporciona una capa uniforme de validaciones, seguridad básica y gestión de datos.
- El Cliente
 - Consume las APIs REST/GraphQL.
 - Puede recibir notificaciones en tiempo real mediante WebSocket.
 - Procesa los datos para mostrar el catálogo de productos u otras vistas.

2. Capas del sistema

La aplicación sigue una arquitectura por capas típica de Spring Boot:

Capa de Controladores (Controllers)

Gestiona las peticiones externas. Aquí se definen:

- Endpoints REST
- Resolvers de GraphQL
- Suscripciones de WebSocket
- Gestión de códigos HTTP

Ejemplo:

- ProductoRestController
- ProductoGraphQLController

Capa de Servicios (Services)

Contiene la lógica de negocio. Aquí se encuentran los procesos como:

- Validación adicional de datos
- Transformación avanzada de objetos
- Gestión de transacciones
- Notificación de eventos a WebSocket
- Mapeo de DTOs ↔ Entidades
- Cacheo de elementos

Ejemplo:

- ProductoService

Capa de Repositorios (Repositories)

Se encarga de la interacción con las bases de datos:

- Postgre → repositorios JPA/Hibernate
- MongoDB → repositorios MongoRepository

Ejemplo:

- ProductoRepository (JPA)
- HistorialProductoRepository (MongoDB)

Capa de Modelo (Entities / Documents / DTOs)

Incluye:

- Entidades JPA para Postgre
- Documentos Mongo
- DTOs para entrada y salida en REST/GraphQL
- Ejemplo:
 - Producto (JPA)
 - PedidoDocument(Mongo)
 - POSTandPUTProductoRequestDTO

3. Componentes principales de la arquitectura

API REST

Permite operaciones CRUD tradicionales sobre productos mediante:

- GET /productos
- POST /productos
- PUT /productos/{id}
- DELETE /productos/{id}

Incluye validaciones automáticas con:

- @NotBlank
- @NotNull
- @Min

API GraphQL

Permite consultas más flexibles y precisas. Incluye:

- Queries: obtener productos, obtener producto por ID
- Esquemas en schema.graphqls

WebSocket

- Permite notificar al cliente:
 - Producto creado
 - Producto modificado
 - Producto eliminado
 - Pedido creado
 - Pedido modificado
 - Pedido eliminado
- El servidor publica eventos automáticamente desde ProductoService y desde PedidosService.

Bases de datos

Se combinan dos BBDD:

- Postgre → ideal para datos estructurados y relaciones simples.
- MongoDB → ideal para documentos dinámicos (históricos, datos auxiliares).

Docker

- El sistema está completamente preparado para contenedores:
 - Un contenedor para Spring Boot
 - Un contenedor para MongoDB
 - Un contenedor para Postgre
 - Un contenedor para Nginx con el reporte de tests generado por Jacoco
 - Un contenedor para Nginx con el reporte de tests estándar de JUnit
 - Un contenedor para Apache con la documentación generada por Dokka
 - Un contenedor Nginx con el proxy inverso que filtra todas las conexiones a la aplicación y el resto de servicios
 - Un contenedor para la caché Redis
- mediante el docker-compose.yml, que permite orquestar todos los servicios.

4. Flujo general de funcionamiento

- El cliente realiza una petición (REST/GraphQL) al servidor.
- El controlador procesa la entrada y realiza las validaciones iniciales incluídas en los DTO.
- El servicio ejecuta la lógica de negocio y convierte los DTO a los modelos de datos. También interacciona con la caché y notifica las acciones realizadas a través de WebSocket.
- El repositorio trabaja con los modelos proporcionados por el servicio e interactúa con la base de datos.
- Todo el flujo anteriormente descrito se realiza en la dirección inversa hasta que finalmente el controlador devuelve al cliente la información correspondiente mediante un DTO de salida.

Patrones Utilizados

En el desarrollo de este proyecto se han aplicado varios patrones de diseño y estructuración que facilitan la organización del código, la mantenibilidad del sistema y la escalabilidad de la aplicación. Algunos de los patrones utilizados son los siguientes:

1. Patrón CRUD

El proyecto implementa de forma clara el patrón CRUD (Create, Read, Update, Delete), ya que la aplicación gestiona un catálogo de productos, permitiendo:

- Crear productos
- Leer productos
- Actualizar productos
- Eliminar productos

2. Patrón DTO (Data Transfer Object)

Un DTO (Data Transfer Object) es un objeto simple utilizado para transportar datos entre distintas capas o procesos de una aplicación sin contener lógica de negocio, solo propiedades. Sirve para reducir el acoplamiento, evitar exponer directamente las entidades internas (modelos) y optimizar el intercambio de información.

Ejemplos en el proyecto:

- POSTandPUTProductoRequestDTO
- ProductoResponseDTO

Tecnologías

Este es un listado de todas las tecnologías de las que se ha hecho uso a lo largo de la práctica además del porqué han sido usadas:

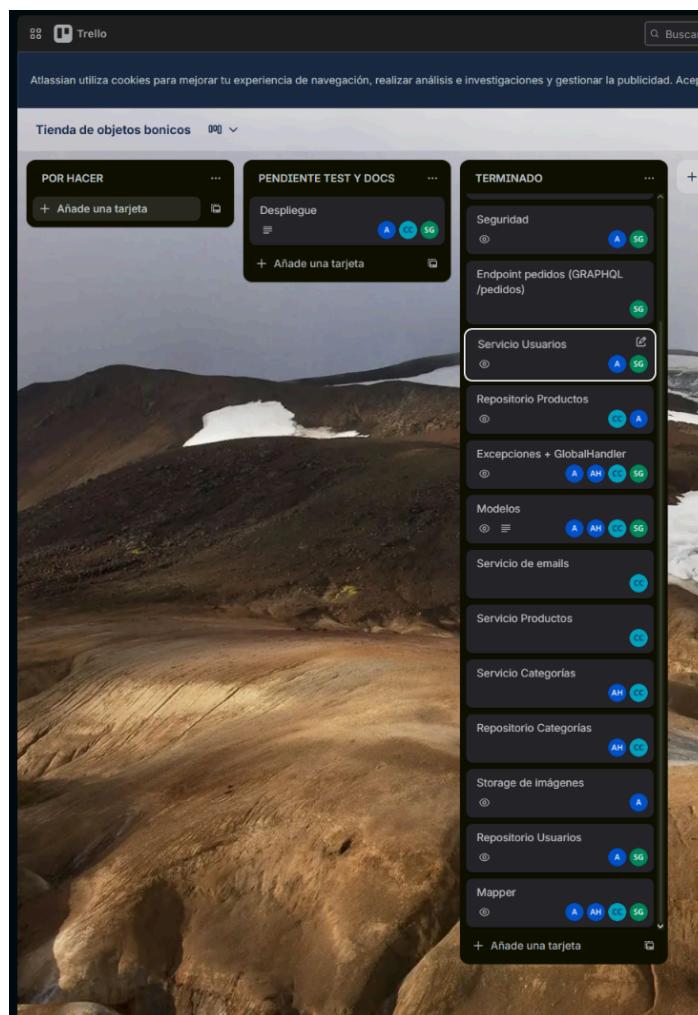
- IntelliJ: entorno de desarrollo principal usado para escribir y ejecutar el código. Se ha hecho uso de este entorno de desarrollo debido a que es el que se ha usado a lo largo de todo el curso, en parte por su facilidad a la hora de usar proyectos colaborativos y porque garantiza muy bien la calidad del código.
- Visual Studio Code: entorno de desarrollo secundario para revisar el código de los compañeros. Es el entorno más rápido si lo que se busca es simplemente visualizar código.
- Git: control de versiones para la gestión de cambios en el código. Se ha usado, además de porque es el estándar, ya que permite guardar un historial de cambios muy útil.
- GitHub: ha servido para alojar el repositorio del proyecto y poder gestionar los cambios desde ahí. Se ha usado ya que ofrece una integración directa con Git.
- Windows Terminal: para ejecutar los comandos y gestionar el proyecto desde terminal. Usada debido a que es la predeterminada y cubre todas las necesidades del proyecto.
- Trello: para el reparto de tareas. Se exigió en la anterior práctica y es muy cómodo en cuanto a gestionar qué parte hace cada miembro del equipo.
- Word: para poder redactar la documentación del proyecto. Usado ya que es el estándar en cuanto a procesadores de texto.
- Docker: para crear servicios garantizando su independencia y escalarlos de forma sencilla, mediante contenedores de virtualización ligera.
- Nginx: como servidor web y proxy inverso.
- Apache: como servidor web.
- Redis: como servicio de caché.

Planificación Trello

Para el reparto de tareas y poder planificar el proyecto eficientemente se ha hecho uso de Trello, un software de administración muy intuitivo que ha sido de gran ayuda.

A continuación, el Trello del proyecto:

<https://trello.com/b/FToRIHAM/tienda-de-objetos-bonicos>



Requisitos funcionales y no funcionales

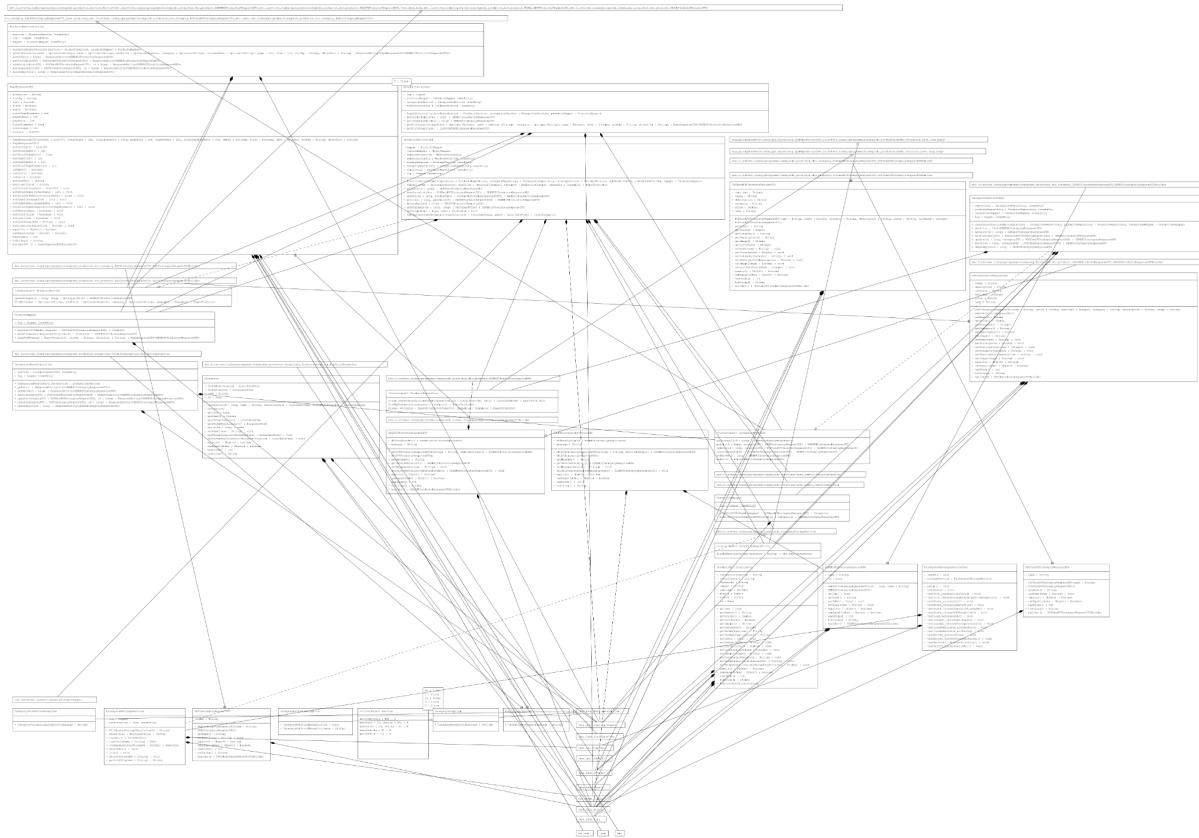
Requisitos Funcionales

- RF1: Se realizan operaciones CRUD (create, read, update y delete) para gestionar los productos, categorías y pedidos.
- RF2: Se asigna un ID único a cada entidad.
- RF3: Se exponen los datos mediante API REST.
- RF4: Se exponen los datos mediante GraphQL.
- RF5: Se implementa comunicación en tiempo real mediante WebSocket para notificaciones de creación, actualización o eliminación de productos y pedidos.
- RF6: Inicio de sesión y gestión de usuarios (usuario y contraseña encriptada con BCrypt).
- RF7: Tarea programada de envío de correo electrónico.

Requisitos No Funcionales

- RNF1: Uso de caché para mejorar el rendimiento de consultas frecuentes.
- RNF2: Cobertura de pruebas unitarias.
- RNF3: Ejecución mediante Jar y contenedores Docker.
- RNF4: Gestión del proyecto y control de versiones mediante GitHub.

Diagrama de clases



Principios SOLID aplicados

Principio de responsabilidad única (SRP)

Cada clase en el proyecto tiene una única responsabilidad. Por ejemplo, la clase ProductoService se encarga de la lógica de negocio de lo relativo a productos y categorías y ProductoController se encarga de procesar peticiones HTTP.

Esto asegura que cada clase cumpla con su función específica sin mezclarse con otras responsabilidades.

Principio de abierto/cerrado (OCP)

Las clases están diseñadas para ser extendidas sin necesidad de modificar su código original. Por ejemplo, en ProductoServicelImplementation, si se desea agregar un nuevo formato de archivo para guardar imágenes de productos, podemos extender la funcionalidad sin alterar el código existente, respetando así este principio.

Principio de sustitución de Liskov (LSP)

Las subclases pueden reemplazar a sus superclases sin afectar el funcionamiento del sistema.

Principio de segregación de interfaces (ISP)

Se prefieren interfaces específicas y enfocadas en lugar de grandes interfaces generales. Por ejemplo, la interfaz ProductoService se centra únicamente en las operaciones CRUD para Producto, mientras que la interfaz Service define operaciones CRUD genéricas para cualquier servicio. Esto evita interfaces infladas y mejora la mantenibilidad.

Principio de inversión de dependencias (DIP)

El proyecto depende de abstracciones en lugar de implementaciones concretas. Por ejemplo, ProductoServicelImplementation depende de la interfaz ProductoService, permitiendo cambiar las implementaciones sin afectar el código cliente.

Diseño de la base de datos

El sistema utiliza un diseño híbrido de bases de datos, combinando una base de datos relacional (PostgreSQL) y una NoSQL orientada a documentos (MongoDB). Esta arquitectura permite aprovechar las ventajas de ambos modelos según el tipo de información que maneja la aplicación.

1. Base de Datos Relacional: PostgreSQL

La base de datos PostgreSQL se utiliza para almacenar información estructurada y altamente relacionada. En este proyecto, se emplea principalmente para gestionar la información principal del catálogo de productos, categorías y usuarios.

La entidad sigue una estructura estándar JPA:

- PK auto-generada mediante `@GeneratedValue`.
- Validaciones con `@NotBlank`, `@NotNull`, `@Min`, etc.
- Enum mapeado como String para mayor claridad en la BD.

Relación con la API

Esta tabla es consultada y modificada a través de:

- Endpoints REST (`/productos`)
- Mutaciones y queries GraphQL
- Servicios y repositorios basados en JPA

2. Base de Datos NoSQL: MongoDB

La base de datos MongoDB se utiliza para almacenar información que puede requerir una estructura más flexible o variable. En nuestro proyecto, lo hemos usado para definir la estructura de Pedidos ya que lleva dos clases embebidas tales como Cliente y LineaPedido lo que puede requerir cambios constantes en el modelo.

Este enfoque es especialmente útil cuando los datos:

- No siguen un esquema rígido
- Pueden evolucionar con el tiempo
- Contienen información dinámica

API REST / GraphQL

Este proyecto es un backend en Java para gestionar una tienda, incluyendo productos, categorías y pedidos. Proporciona acceso a los datos mediante API REST y GraphQL.

Puntos clave:

- Gestión de la tienda: CRUD de Producto, Categoría y Pedido.
- REST: Endpoints para operaciones estándar como crear, listar, actualizar o eliminar productos y pedidos.
- GraphQL: Consultas flexibles que permiten obtener solo los datos necesarios, como productos por categoría o detalles de un pedido específico.
- Arquitectura modular: Separación en capas (modelos, servicios, repositorios), siguiendo principios SOLID para mantener el código limpio y extensible.
- Abstracciones: Interfaces y clases abstractas desacoplan la lógica de negocio de la persistencia, facilitando cambios en la base de datos o en la implementación de los repositorios.

WebSocket

Este proyecto utiliza WebSocket para ofrecer funcionalidades en tiempo real en la tienda, mejorando la interacción con los usuarios y la actualización instantánea de los datos.

Puntos clave:

- Actualización en tiempo real: Permite que los clientes reciban cambios inmediatamente, como actualizaciones de stock de productos o estado de pedidos.
- Comunicación unidireccional: Los clientes pueden recibir mensajes desde el servidor sin necesidad de hacer peticiones HTTP constantes.
- Arquitectura modular: La lógica de tiempo real está integrada con la capa de servicios y modelos existentes, respetando los principios SOLID.

Casos de uso típicos:

- Notificación inmediata de cambios en stock o disponibilidad de productos.
- Actualización automática del estado de pedidos.

Beneficio:

- Los WebSockets mejoran la experiencia del usuario al ofrecer información actualizada al instante, reduciendo la necesidad de recargar páginas o hacer consultas periódicas al servidor.

Docker

El proyecto se ejecuta dentro de contenedores Docker, lo que facilita el despliegue, la portabilidad y las pruebas automatizadas.

Puntos clave:

- Aislamiento y portabilidad: Cada componente (backend, base de datos) corre en su contenedor independiente, funcionando igual en cualquier sistema con Docker.
- Configuración simplificada: Uso de Dockerfile y docker-compose.yml para levantar contenedores fácilmente.
- Testcontainers: Se utilizan contenedores temporales para pruebas automatizadas, garantizando que los tests se ejecuten en un entorno limpio y controlado.
- Despliegue consistente: El proyecto funciona igual en desarrollo, pruebas y producción, evitando conflictos con el entorno local.

Casos de uso típicos:

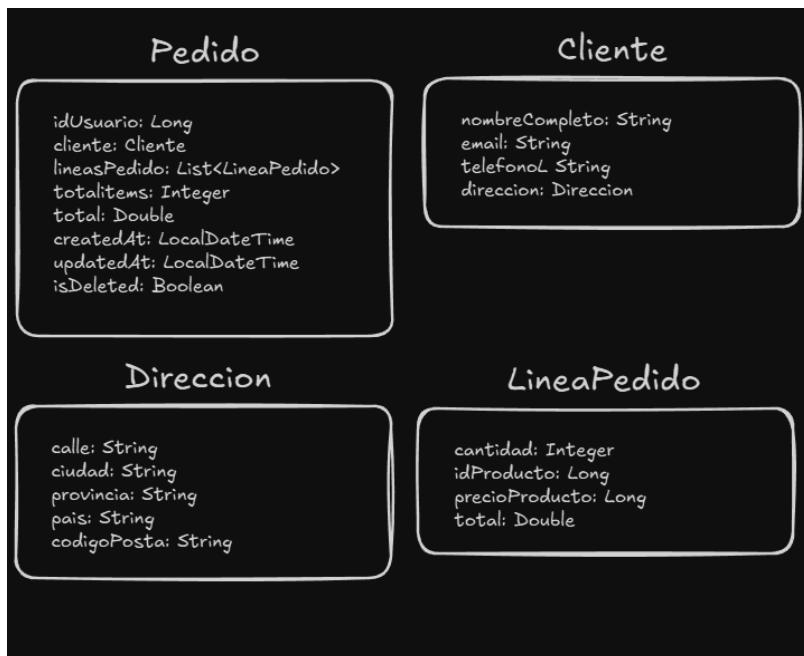
- Levantar la aplicación y la base de datos con un solo comando (docker-compose up).
- Ejecutar tests automatizados con bases de datos temporales mediante Testcontainers.
- Escalar o replicar servicios de forma sencilla.

Modelos

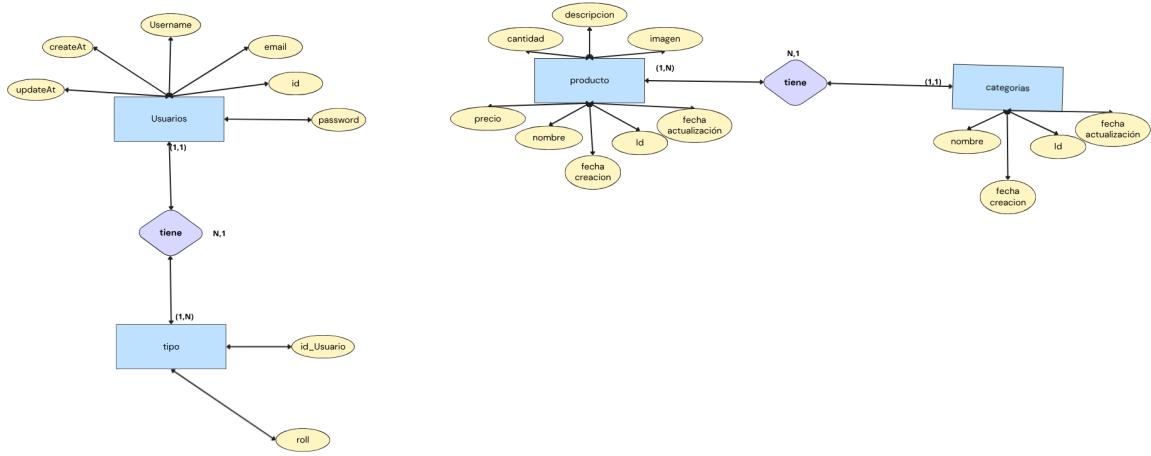
RELACIONAL:



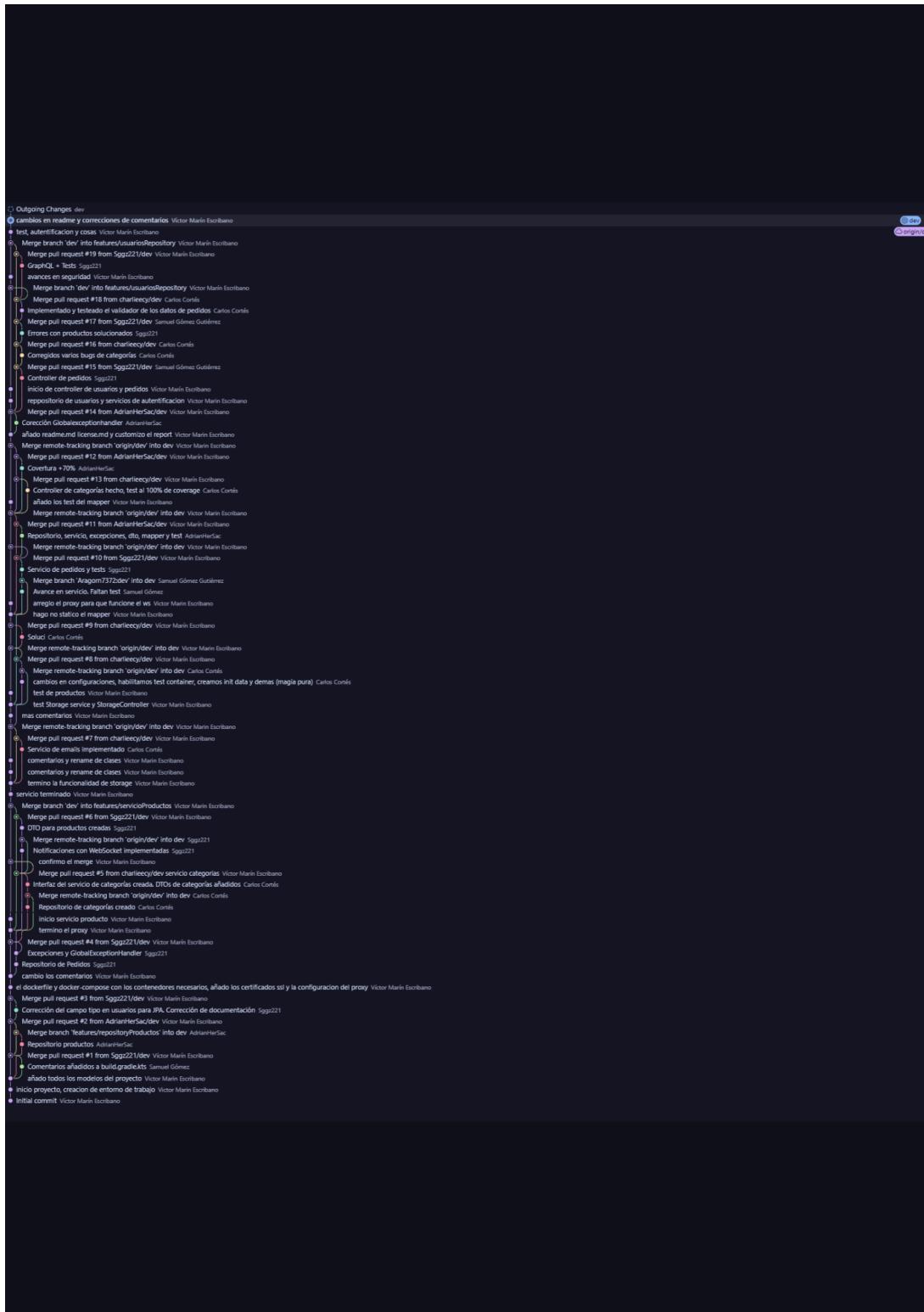
NO RELACIONAL:



Diagramas entidad-relación



Gitflow



Estimación de costes

HORAS DE TRABAJO ESTIMADAS			
Requisitos funcionales		Nombre	Horas estimadas
RF	Descripción	Nombre	Horas estimadas
RF1	Operaciones CRUD (productos, categorías y pedidos).		40
RF2	Id único a cada entidad.		1
RF3	API REST.		10
RF4	GraphQL.		5
RF5	WebSocket.		7
RF6	Usuarios y seguridad.		30
RF7	Tarea programada.		3
			96

HORAS DE TRABAJO ESTIMADAS			
Requisitos no funcionales		Nombre	Horas estimadas
RNF	Descripción	Nombre	Horas estimadas
RNF1	Caché Redis.		5
RNF2	Testeo del código.		25
RNF3	Docker.		20
RNF4	GitHub.		5
			55

HORAS DE TRABAJO TOTALES	
	151
PRECIO POR HORA DE TRABAJO	
	135,00 €
PRECIO POR HORA EXTRA DE TRABAJO	
	160,00 €
COSTE TOTAL DE LAS HORAS DE TRABAJO	
	20.385,00 €

ESTIMACIÓN DE COSTES					
Nº	Concepto	Desglose		Coste	
1	Horas de trabajo estimadas por requisitos	Precio por hora	135,00 €	151	20.385,00 €
		Precio por hora extra	15% de las horas totales		
2	Horas extra para cubrir imprevistos	160,00 €		22,65	3.624,00 €
		Costes antes de margen	17% sobre los costes		
3	Margen de beneficios	24.009,00 €		0,17	4.081,53 €
					28.090,53 €

Conclusiones

El proyecto demuestra una arquitectura modular y mantenible, siguiendo los principios SOLID, lo que facilita la extensión y el mantenimiento del código.

La API REST permite operaciones estándar mientras que GraphQL ofrece consultas flexibles, optimizando el acceso a los datos.

La integración de WebSockets garantiza información en tiempo real, mejorando la experiencia del usuario en la tienda.

La utilización de Docker y Test Containers asegura portabilidad, consistencia en el despliegue y entornos controlados para pruebas automatizadas.

En conjunto, estas tecnologías permiten un sistema de tienda robusto, escalable y fácil de probar, preparado tanto para desarrollo como para producción.