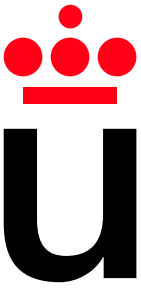


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

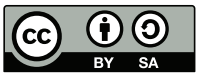
Grado en Ingeniería Informática

Curso 2024-2025

Trabajo Fin de Grado

LABOTICADEMAR: TIENDA ONLINE PARA LA VENTA DE PRODUCTOS DE PARAFARMACIA

Autor: Álex Murciano Pérez
Tutor: Michel Maes Bermejo



Agradecimientos

Quiero expresar mi más profundo agradecimiento a mi madre, cuya vida ejemplar me ha enseñado el valor del esfuerzo y la constancia. Su dedicación inquebrantable y su lucha diaria han sido siempre mi mayor inspiración, recordándome que no hay meta inalcanzable cuando se trabaja con pasión y determinación.

A mi hermana, la persona mas importante de mi vida, quiero expresarle mi profundo agradecimiento por el vínculo indescriptible que nos une. Disfruto del honor de aprender de su ejemplo día tras día, de experimentar el amor incondicional y libre de prejuicios que siempre me brinda, y de saber que siempre contaré con un hombro en el cual apoyarme.

A Elisa, mi compañera de vida y mejor amiga, cuya sonrisa ilumina y llena de alegría cada estancia, desprende una energía espontánea y, al mismo tiempo, cargada de cordura que me impulsa a avanzar siempre en la dirección correcta. Sin duda representa toda la fortuna que podía esperar en esta vida, y la admiro con orgullo en cada paso que da.

Por último, quiero dedicar un reconocimiento muy especial a mi abuelo Rafael, que aunque no podrá ver el final de esta gran etapa, nunca dejó de acompañarme con su espíritu. Su filosofía de vida, basada en la humildad y la fortaleza ante la adversidad, me ha impulsado a seguir adelante cada vez que los resultados no fueron los esperados. Su memoria sigue siendo mi guía.

Resumen

En el desarrollo de este proyecto se ha construido una tienda *online* para productos de parafarmacia que combina una experiencia de usuario intuitiva con una arquitectura sólida y robusta. A partir de las sesiones de captura de requisitos junto a la propietaria del negocio, se definieron las principales necesidades y la prioridad de estas.

La aplicación se ha implementado sobre Spring Boot, gestionada con Maven y respaldada por una base de datos MySQL. En el front-end se ha utilizado JSP y JavaScript para la creación de vistas dinámicas, estilizadas con Tailwind CSS para garantizar una interfaz moderna y *responsive*.

El código fuente reside en GitHub, donde un workflow de GitHub Actions automatiza las fases de integración continua y despliegue continuo, empaquetando la aplicación en un contenedor Docker, enlazándolo a la base de datos alojada en una instancia RDS de AWS y publicando la *webstore* en una instancia EC2 en AWS.

Se han incorporando prácticas DevOps que permiten un ciclo de vida del software más ágil, una infraestructura como código reproducible por otros desarrolladores y una monitorización permanente de la aplicación, asegurando así actualizaciones rápidas y un entorno de producción estable.

Para organizar el desarrollo se siguió la metodología Scrum mediante objetivos semanales, además del uso de tablones Kanban para visualizar el avance de cada tarea y adaptar el plan de trabajo según las prioridades de las tareas pendientes.

Palabras clave:

- Java
- SpringBoot
- Maven
- Tailwind
- JavaScript
- MySQL

- GitHub
- Docker
- AWS (Amazon Web Services)
- DevOps
- SaaP (Software as a Service)

Índice de contenidos

Índice de figuras

Índice de códigos

1 Introducción

En una era claramente dominada por el enfoque digital, la tecnología se ha convertido en una pieza clave para el crecimiento y la supervivencia de casi cualquier sector. Esta transformación ha cambiado las reglas del juego, modificando las expectativas de los clientes, las operaciones internas y los propios modelos de negocio. Da igual cuál sea el negocio del que hablemos, todos han tenido que adaptarse para seguir siendo relevantes y competitivos. En una sociedad donde no estar en internet es casi sinónimo de ser invisible, las tiendas online se han convertido en el escaparate de mayor alcance que puede tener cualquier entidad que ofrezca un producto.

1.1. Contexto

Dentro de este panorama de transformación digital, el sector sanitario, y más concretamente la farmacia a pie de calle, presenta un caso de estudio de particular interés. Las farmacias siempre han sido un pilar en la sociedad, un lugar basado en la confianza y el consejo cercano. Sin embargo, hoy en día las personas se han acostumbrado a la inmediatez del mundo online y empiezan a demandar ese mismo nivel de acceso y comodidad para los productos de parafarmacia. La diferencia fundamental, y la gran oportunidad, es que buscan este servicio otorgado desde la propia farmacia. Aunque esta clase de productos se pueden conseguir en otros comercios online, quien atiende detrás de esa pantalla no es un profesional cualificado con el conocimiento y el rigor que rodea al producto que se dispensa, un valor que solo la farmacia puede garantizar.

Este proyecto nace justo en el punto donde se encuentran esa tradición y la necesidad de innovar, con la idea de llevar el valor de la farmacia de barrio al mundo digital, no para reemplazarla, sino para ampliar su alcance de una forma lógica y necesaria. El valor añadido que se aplica al proceso de compra se traduce en tener un lugar de referencia, en el que el cliente sabe que está recibiendo algo más que el producto material; recibe el trato, la seguridad y la atención personalizada que ya se viene dando en el mostrador físico, pero esta vez a través de un mostrador digital con un alcance infinitamente superior.

Mi interés por la ingeniería informática no es reciente, sino que se remonta a mucho antes de mi etapa universitaria. Desde muy joven sentí una gran atracción por el mundo de la tecnología y los dispositivos electrónicos, y fue años antes de empezar la carrera cuando, de forma autodidacta, comencé a dar mis primeros pasos en la programación. Ya desde entonces, entendí el desarrollo de software no como un fin en sí mismo, sino como una herramienta increíblemente potente para crear soluciones, resolver problemas y aportar valor. Posteriormente, al iniciar el Grado en Ingeniería Informática, esta vocación inicial encontró un cauce formal. Con el paso de los años, sentía cómo mis conocimientos se ampliaban y, con ello, cómo aquel gran proyecto que siempre había rondado mi mente se volvía cada vez más factible.

Esa inquietud creativa y aquel proyecto latente encontraron finalmente un propósito claro y personal en mi entorno familiar: la farmacia que dirige mi madre. La idea de crear una página web para el negocio empezó casi como una broma recurrente en casa, pero a medida que avanzaba en mis estudios, se fue transformando en un reto cada vez más real y serio. Cada nueva asignatura, ya fuera sobre bases de datos, seguridad o experiencia de usuario, dejaba de ser un concepto teórico; para mí, se convertía en una pieza más que encajaba en el puzle de este proyecto, haciéndolo parecer no solo posible, sino necesario. De esta forma, lo que nació como una broma se convirtió en un objetivo definido y, finalmente, en la verdadera pasión que impulsa este trabajo.

A esta motivación personal se suma una ventaja estratégica fundamental: un conocimiento interno y práctico del negocio. Durante varios veranos, he tenido la oportunidad de trabajar directamente en la farmacia, colaborando en tareas como la preparación de pedidos de productos para residencias o la gestión del inventario. Esta experiencia me ha proporcionado una visión desde dentro, permitiéndome comprender no solo el flujo de trabajo diario y las herramientas utilizadas, sino también identificar de primera mano las necesidades no cubiertas, las in-eficiencias en los procesos y, sobre todo, los márgenes de mejora que la tecnología podría optimizar. Por tanto, este proyecto no se basa únicamente en una necesidad de mercado externa, sino que nace de un diagnóstico interno y detallado, lo que me permite diseñar una solución a medida que responda a problemas reales y específicos de la operativa de la farmacia.

Al llegar al final de la carrera, vi en este proyecto la oportunidad perfecta para

cerrar mi etapa como estudiante. Era el momento de aplicar y combinar todo lo aprendido durante años en un único proyecto de gran escala. Para mí, es mucho más que un requisito académico; lo he concebido como mi primer proyecto con un enfoque verdaderamente profesional. Un trabajo en el que he podido volcar todo mi esfuerzo, enfrentándome a los desafíos que implica crear un producto desde cero, con el gran aliciente de que su éxito ayudará directamente al negocio familiar. Por tanto, aquí es donde se unen mi vida académica y personal. Es mi forma de devolver, con una herramienta útil y actual, una parte de todo el apoyo que he recibido. En el fondo, es la materialización de todo mi aprendizaje; la prueba práctica de que la ingeniería informática puede tender puentes entre las necesidades de las personas y la tecnología, creando algo tangible que refleja mi crecimiento a lo largo de estos años.

Para entender del todo por qué este proyecto es tan relevante, es fundamental analizar el sector de la farmacia en España, que tiene unas características que lo convierten en un campo con mucho potencial para la innovación. Un rasgo muy notable es la edad media de sus propietarios. Generalmente, los titulares de farmacia son profesionales de gran trayectoria, muchos de los cuales superan los 55 años. Su conocimiento sanitario es el pilar del negocio, pero es comprensible que su relación con la tecnología sea, a veces, más distante. Esta diferencia generacional ha hecho que la digitalización en este sector haya sido más lenta que en otros. La gestión ha seguido modelos más clásicos, y la idea de invertir en un proyecto tecnológico complejo como una tienda online a menudo se ve como un reto demasiado grande o, simplemente, no se considera una prioridad.

A esto se suma que muchas farmacias son negocios familiares que pasan de generación en generación. Esto, que asegura su continuidad, también puede generar una cierta resistencia al cambio en la forma de trabajar, haciendo que la mayoría de farmacias ofrezcan servicios muy parecidos y principalmente físicos. Curiosamente, mientras que la pandemia de COVID-19 forzó una digitalización masiva en la mayoría de los sectores, en el ámbito farmacéutico su efecto fue, en muchos casos, el contrario. Al ser consideradas un servicio de primera necesidad, las farmacias fueron de los pocos negocios que permanecieron abiertos y no percibieron la obligación de cierre ni una drástica bajada de facturación como una amenaza inminente. Esta çondición especial", que reforzó su papel crucial en la sociedad, provocó que no sintieran esa necesidad imperante de migrar al sector digital como salvaguarda frente a cualquier condición externa, a diferencia de otros comercios. Por este motivo, a pesar de que la demanda de productos de parafarmacia online sí aumentó, muchas farmacias no acometieron el salto digital, ya que su modelo presencial seguía siendo viable y demandado. Esta inercia, fruto de su propia resiliencia, es lo que convierte la digitalización hoy en una oportunidad estratégica tan clara y en una ventaja competitiva aún mayor para quien decide dar el paso de adaptarse a los nuevos tiempos.

La puesta en marcha de esta plataforma se justifica, por tanto, desde varios

ángulos que demuestran su importancia. Desde el punto de vista académico, este Trabajo de Fin de Grado me ha obligado a aplicar y combinar conocimientos de todas las áreas clave de la carrera: desde la ingeniería del software y la gestión de bases de datos, hasta el desarrollo web completo (frontend y backend), la configuración de sistemas y, muy importante, la aplicación de medidas de ciberseguridad para proteger la información de los clientes. Es la prueba práctica de la formación recibida. Desde el punto de vista del negocio, el proyecto responde a una necesidad real y urgente, creando una herramienta pensada para tener un impacto positivo en las ventas, la visibilidad y la capacidad de la farmacia para competir en el mercado actual.

Finalmente, la importancia de este proyecto va más allá de ser solo un Trabajo de Fin de Grado o una ayuda para el negocio familiar, ya que lo he planteado con una visión de futuro. Es un proyecto vivo, en el que podré seguir trabajando y mejorando mucho después de entregarlo, lo que me permitirá ganar experiencia en el mantenimiento y la evolución de un producto de software real. Además, me sirve como mi principal carta de presentación profesional y como un modelo de negocio que se puede replicar. La plataforma está diseñada para ser modular y escalable, de modo que si esta primera implementación funciona bien, pueda servir como caso de estudio para presentarlo a otras farmacias en una situación parecida. La idea es poder adaptar este modelo en el futuro, llevando la solución a otros locales y haciendo crecer el proyecto. Esto no solo me abriría nuevas oportunidades, sino que me ayudaría a posicionarme como un profesional enfocado en la digitalización de este nicho de mercado. En definitiva, este proyecto representa la confluencia de un reto técnico, una motivación personal y una clara oportunidad de mercado, siendo el culmen de mi etapa formativa y el primer paso de mi carrera profesional.

2

Objetivos

2.1. Objetivos técnicos

- **Aplicar una arquitectura de software moderna:** Desarrollar la solución sobre un `stack` tecnológico actual y robusto, que garantice un buen rendimiento, mantenibilidad y escalabilidad futura.
- **Diseñar un modelo de datos optimizado:** La estructura de la base de datos relacional, deberá ser eficiente para gestionar las relaciones entre usuarios, productos, pedidos y categorías, asegurando la integridad de los datos y la rapidez en las consultas.
- **Implementar una API RESTful segura:** La comunicación entre el `frontend` (cliente) y el `backend` (servidor) se realizará a través de una API bien definida, siguiendo los principios REST y asegurada mediante tokens de autenticación.
- **Garantizar la seguridad y la conformidad normativa:** Aplicar las mejores prácticas en ciberseguridad, incluyendo el uso de certificados SSL/TLS para cifrar la comunicación (`HTTPS`).
- **Integración y despliegue continuo:** Hacer uso de metodologías que garanticen la integridad del código y el despliegue a producción estandarizado.
- **Calidad del código:** Asegurar la calidad del código mediante técnicas de encapsulación, organizacin del proyecto y una cobertura elevada mediante tests.

2.2. Objetivos funcionales

- **Desarrollar un catálogo de productos dinámico:** La plataforma permitirá la gestión integral de productos, incluyendo atributos como nombre, descripción, imágenes, precio, stock y categorización, de una forma intuitiva para el administrador.
- **Implementar un sistema de navegación y búsqueda eficiente:** El usuario final debe poder encontrar productos de manera rápida y sencilla, utilizando un sistema de categorías claro, un apartado de filtrado múltiple y un motor de búsqueda por texto predictivo.
- **Gestión de usuarios no registrados:** Un usuario no autenticado en la pagina, podrá hacer un uso idéntico al que hace un usuario autenticado a excepción de ejecutar la compra final.
- **Crear un panel de administración:** El propietario del negocio dispondrá de un área privada para la consulta de pedidos y devoluciones, la administración de productos, la consulta de datos básicos de clientes y diferentes estadísticas representativas del estado del negocio.
- **Crear un panel de empleado:** El empleado dispondrá de un área privada para la consulta de pedidos y devoluciones y la administración de ciertos campos de los productoss.
- **Creación de ventas y devoluciones** El usuario podrá realizar una compra de una lista de productos guardada y también podrá ejecutar una devolución de una selección de productos pertenecientes a una compra previa.
- **Asegurar un diseño adaptable (`Responsive Design`):** La interfaz de usuario deberá ser completamente funcional y ofrecer una experiencia de usuario óptima en dispositivos con pantallas medianas-grandes.

2.3. Objetivos de negocio

- **Creación de un nuevo canal de ingresos:** Establecer una vía de facturación adicional y complementaria a la venta física, operativa 24/7.
- **Incremento de la base de clientes:** Captar nuevos segmentos de mercado, especialmente usuarios digitalmente activos que priorizan la comodidad de la compra online, pero valoran la confianza de una farmacia real.
- **Fidelización y propuesta de valor:** Reforzar la relación con la clientela actual ofreciendo un servicio añadido que mejore su experiencia de compra mediante sistema cashback en conceptos de puntos.

3

Tecnologías, Herramientas y Metodologías

En este capítulo se listarán las tecnologías, herramientas y metodologías usadas en la aplicación **LaBoticaDeMar**.

3.1. Tecnologías

3.1.1. Java

Java [1] es el lenguaje de programación utilizado para la implementación del backend. Se ha seleccionado la versión **17**, tal como queda definida en la propiedad 'java.version' del POM, por su soporte a largo plazo y compatibilidad con Spring Boot 3.2.3.

3.1.2. Spring Boot

Spring Boot [2] es el framework principal para el desarrollo de la aplicación. Gracias al 'spring-boot-starter-parent' (versión **3.2.3**), el proyecto incorpora automáticamente:

- **spring-boot-starter-web** para exponer controladores REST y servir vis-

tas.

- **spring-boot-starter-data-jpa** para la integración con JPA/Hibernate.
- **spring-boot-starter-actuator** para monitorización y métricas.
- **spring-boot-devtools** (opcional) para recarga automática durante el desarrollo.

3.1.3. Maven

Maven [3] es la herramienta de construcción y gestión de dependencias utilizada en el proyecto. Facilita la compilación, empaquetado, pruebas e integración con otras herramientas del ecosistema Java.

3.1.4. Apache Tomcat

Apache Tomcat [4] es el servidor de aplicaciones embebido por defecto en Spring Boot. Gestiona las peticiones HTTP y permite ejecutar la aplicación como un `.jar` autónomo sin necesidad de servidor externo.

3.1.5. Spring Security

Spring Security [5] es el framework encargado de gestionar la autenticación y autorización en la aplicación. Se ha configurado para autenticación basada en usuario y contraseña, utilizando BCrypt para el cifrado de contraseñas. Además, se incluye:

- **spring-security-taglibs** [6] (versión 6.0.0) para etiquetas JSP que facilitan la implementación de controles de seguridad en vistas.

3.1.6. JSON

JSON [7] es el formato principal de intercambio de datos entre el frontend y el backend. Se utiliza en las APIs REST para enviar y recibir información estructurada, y se procesa con la biblioteca Jackson para la conversión entre objetos Java y texto JSON. Los DTOs definen la estructura de los datos intercambiados.

3.1.7. JSON Web Tokens (JWT)

JWT [8] se emplea para la autenticación sin estado en la aplicación. Los tokens generados con la librería JJWT (versión **0.11.5**) incluyen información del usuario, roles y fecha de expiración, y se firman usando HMAC-SHA con una clave secreta para garantizar su integridad.

3.1.8. MySQL

MySQL [9] es el sistema de gestión de bases de datos relacional utilizado en producción. Se emplea el driver JDBC `mysql-connector-java` en la versión 8.0.19 para la conexión con la base de datos.

3.1.9. Hibernate

Hibernate [10] es el framework ORM utilizado para mapear entidades Java a tablas de base de datos. Se integra mediante JPA con Spring Boot, facilitando operaciones CRUD.

3.1.10. H2

H2 [11] es la base de datos en memoria utilizada en entornos de pruebas e integración continua. Se integra mediante la dependencia `com.h2database:h2` en la versión 2.1.214.

3.1.11. JUnit 5

JUnit 5 (Jupiter) [12] es el framework de pruebas unitarias utilizado en el backend. Se integra a través de la dependencia `spring-boot-starter-test`, que incluye el motor de testeo, soporte para pruebas parametrizadas y extensiones específicas de Spring Boot.

3.1.12. Mockito

Mockito [13] es el framework de creación de dobles de prueba (mocks) para facilitar la simulación de dependencias en los tests unitarios. Se utiliza junto con JUnit 5 para verificar interacciones y comportamientos de componentes aislados.

3.1.13. Selenium

Selenium WebDriver [14] es la herramienta para pruebas end-to-end de la aplicación web. En este proyecto se emplea la versión **4.28.1** (‘org.seleniumhq.selenium:selenium-java‘) para automatizar la navegación, interacción y validación del UI en entornos reales de navegador.

3.1.14. JaCoCo

JaCoCo [15] es el plugin de Maven utilizado para el análisis de cobertura de código en Java. Se configura en la versión 0.8.11 y se emplea el goal `prepare-agent` para instrumentar el código durante los tests y el goal `report` en la fase `prepare-package` para generar el informe de cobertura.

3.1.15. JavaScript

JavaScript [16] es el lenguaje de scripting utilizado para añadir interactividad y dinamismo al frontend. Se incluyen bibliotecas y frameworks ligeros según las necesidades (por ejemplo, librerías de validación de formularios o manipulación del DOM).

3.1.16. HTML

HTML [17] es el lenguaje de marcado empleado para estructurar el contenido de las páginas web. Se usa JSP para generar HTML dinámico en el servidor, integrando etiquetas JSTL y fragmentos de Thymeleaf si es necesario.

3.1.17. CSS

CSS [18] es el lenguaje de estilo responsable de la presentación visual de las páginas. Se combinan estilos de Bootstrap y Tailwind CSS con reglas personalizadas para adaptar el diseño a la identidad de LaBoticaDeMar.

3.1.18. Jakarta Servlet API

Jakarta Servlet API [19] proporciona las clases y métodos necesarios para gestionar peticiones y respuestas HTTP en el servidor. Se utiliza con scope `provided` para compilar contra la versión del contenedor de servlets.

3.1.19. Tomcat Embed Jasper

Tomcat Embed Jasper [20] permite compilar y ejecutar páginas JSP dentro de la propia aplicación Spring Boot, sin necesidad de un servidor externo. Facilita el desarrollo al empaquetar el motor JSP junto al WAR.

3.1.20. JSTL

JSTL (Jakarta Standard Tag Library) [21] ofrece un conjunto de etiquetas personalizadas para control de flujo, iteración y manipulación de datos en JSP. Se incluyen la API (‘jakarta.servlet.jsp.jstl-api‘) y la implementación de GlassFish, versión 2.0.0.

3.1.21. Bootstrap

Bootstrap [22] se sirve desde WebJars (‘org.webjars:bootstrap:bootstrap.version‘) y proporciona estilos CSS y componentes interactivos responsivos, agilizando el diseño del frontend.

3.1.22. Tailwind CSS

Tailwind CSS [23] es un framework de CSS con enfoque *utility-first* que permite construir interfaces personalizadas mediante clases de estilo atómicas. Ofrece mucha flexibilidad y alta personalización del diseño mediante su archivo de configuración, además de la herramietna de compilación que permite incluir en nuestra hoja de estilos solo el CSS necesario.

3.1.23. ModelMapper

ModelMapper [24] en la versión 3.1.1 se utiliza para la conversión automática entre entidades y DTOs, simplificando el mapeo de objetos en el backend.

3.1.24. Lombok

Lombok [25] versión 1.18.26 se utiliza para reducir el código repetitivo en las clases Java mediante anotaciones como `@Getter`, `@Setter` y `@Builder`, mejorando la legibilidad y mantenimiento.

3.1.25. Apache POI

Apache POI [26], en concreto el módulo `poi-ooxml` versión 5.2.3, se emplea para la lectura y generación de documentos Office (Excel, Word, etc.) desde la aplicación.

3.1.26. dotenv-java

dotenv-java [27] versión 3.0.0 se emplea para cargar variables de entorno desde un fichero '.env', facilitando la gestión de configuraciones sensibles fuera del código fuente.

3.1.27. Docker

Docker [28] se utiliza para contenerizar la aplicación y sus dependencias, garantizando que se ejecute de forma consistente en distintos entornos. Se define un archivo `Dockerfile` para construir una imagen personalizada del backend, y se emplea `docker-compose` para orquestar servicios como la base de datos MySQL durante el desarrollo y pruebas.

3.2. Herramientas

3.2.1. Visual Studio Code

Visual Studio Code [29] es un editor de código fuente ligero y multiplataforma utilizado para la edición de archivos JSP, HTML, CSS, JavaScript y otros recursos estáticos del frontend. Ofrece extensiones y herramientas que facilitan el desarrollo web y la integración con sistemas de control de versiones.

3.2.2. Git

Git [30] es el sistema de control de versiones distribuido utilizado para gestionar el código fuente del proyecto. Permite el trabajo colaborativo mediante ramas, historial de cambios y fusiones controladas.

3.2.3. MySQL Workbench

MySQL Workbench [31] es la herramienta gráfica utilizada para modelado, administración y consultas sobre la base de datos MySQL. Facilita el diseño de esquemas, ejecución de sentencias SQL y monitoreo del servidor de bases de datos.

3.2.4. GitHub

GitHub [32] es la plataforma de alojamiento remoto del repositorio Git. Se usa para colaboración, revisión de código y automatización de flujos CI/CD mediante GitHub Actions.

3.2.5. Postman

Postman [33] es la herramienta utilizada para probar y documentar las APIs REST desarrolladas. Permite enviar peticiones HTTP, analizar respuestas y gestionar colecciones de endpoints durante el desarrollo y pruebas.

3.2.6. Docker Hub

Docker Hub [34] es el registro de imágenes de contenedores utilizado para almacenar y distribuir la imagen Docker de la aplicación. Facilita el despliegue desde cualquier entorno compatible con Docker.

3.2.7. Amazon EC2

Amazon EC2 [35] se ha utilizado para ejecutar la aplicación en un contenedor Docker dentro de una instancia Amazon Linux 2. También aloja el servidor Nginx que actúa como proxy inverso.

3.2.8. Amazon RDS

Amazon RDS [36] proporciona la base de datos MySQL en un entorno gestionado, permitiendo una conexión segura y fiable desde la aplicación desplegada en EC2.

3.2.9. Nginx

Nginx [37] es un servidor web ligero y eficiente que se ha utilizado como *proxy inverso* para redirigir el tráfico externo (HTTP y HTTPS) hacia la aplicación desplegada en Docker. Ha sido clave para habilitar la conexión segura a través del puerto 443.

3.2.10. Let's Encrypt

Let's Encrypt [38] es una autoridad certificadora gratuita. En este proyecto se ha empleado para generar certificados SSL que permiten cifrar las comunicaciones mediante HTTPS, garantizando la seguridad en las transacciones con el usuario.

3.2.11. Plesk

Plesk [39] es un panel de control web utilizado en este proyecto para la compra y gestión del dominio, así como para la configuración del DNS necesario para vincularlo con la instancia EC2 de AWS.

3.2.12. PuTTY

PuTTY [40] es un cliente SSH y Telnet ligero y multiplataforma, esencial para establecer conexiones seguras con servidores remotos durante el despliegue y la administración de la infraestructura. Permite gestionar múltiples sesiones, guardar configuraciones de conexión y transferir archivos mediante SCP/SFTP, facilitando la interacción directa con instancias en AWS o entornos on-premise.

3.2.13. Notion

Notion [41] es una plataforma de productividad y colaboración que integra notas, wikis, bases de datos y gestión de tareas en un único espacio de trabajo. En este proyecto se ha utilizado para llevar a cabo la metodología ágil aplicada.

3.3. Metodologías

A lo largo del desarrollo se han aplicado distintas metodologías adaptadas a la naturaleza individual del proyecto, con el objetivo de mantener una organización clara, un ritmo de trabajo constante y una entrega eficiente del software.

3.3.1. Agile

Para afrontar un proyecto de esta envergadura, se optó desde el principio por aplicar un marco de trabajo que permitiera la flexibilidad, el mantenimiento de la motivación y la claridad en los pasos a seguir. Por ello, la elección recayó en Scrum, una metodología ágil que, aunque normalmente se utiliza en equipos, ha sido adaptada en este caso para un contexto de desarrollador único.

En la práctica, esta adaptación ha significado la unificación de los roles clásicos de Scrum. Es la propietaria del negocio la que ha asumido las responsabilidades del "Product Owner", definiendo y priorizando el listado general de tareas del proyecto. Por otro lado, como único integrante del equipo de desarrollo, yo he representado tanto la figura de "Scrum Master", velando por mantener un flujo de trabajo constante y eliminar cualquier bloqueo; y, naturalmente, el role de desarrollador, encargandome de escribir y probar el código.

El ritmo de trabajo se ha estructurado en ciclos cortos de una semana, conocidos como sprints. Al inicio de cada sprint, se realizaba una fase de planificación (Sprint Planning) para seleccionar un conjunto de tareas prioritarias del listado general. Diariamente, se dedicaban unos minutos a un "auto-Daily Scrum", una breve revisión para determinar el punto de partida, el plan para el día actual y los posibles impedimentos. Al concluir la semana, se llevaba a cabo una revisión del incremento funcional desarrollado (Sprint Review) y una breve retrospectiva para analizar la gestión del tiempo y los procesos, identificando puntos de mejora para el siguiente ciclo.

Para materializar y gestionar todo este flujo de trabajo de forma visual, la herramienta clave ha sido un tablero Kanban [3.1]. Este tablero se ha utilizado como el centro de operaciones del proyecto, organizando todas las tareas en columnas simples pero efectivas: Sin empezar, En Progreso, En pausa, Terminado y Cancelado. Cada nueva funcionalidad fue representada como una tarjeta que transitaba por estas fases a medida que se completaba. Para la creación de este tablero se ha utilizado Notion [41] y una plantilla de gestión de proyectos.

El uso de esta herramienta visual ha resultado ser de gran utilidad por dos motivos principales. En primer lugar, proporcionaba en todo momento una fotografía instantánea y muy gráfica del estado real del proyecto, permitiendo ver de un solo vistazo qué quedaba por hacer y dónde se concentraba el trabajo. En segundo lugar, actuó como un importante factor de motivación, ya que el movimiento de las tarjetas hacia la columna "Terminado"transmitía una sensación tangible de progreso y permitía medir el avance de una forma mucho más clara que una simple lista de tareas.

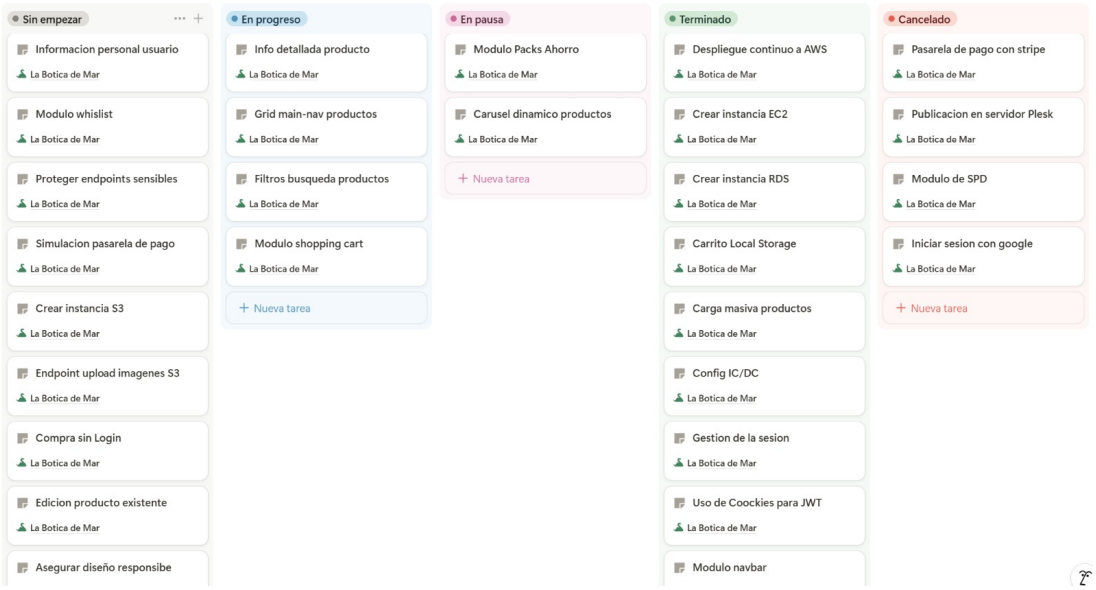


Figura 3.1: Tablero Kanban de organización de tareas.

3.3.2. DevOps

Para complementar la organización del código con Git Flow, se ha adoptado una filosofía DevOps con el objetivo de conectar de forma eficiente el desarrollo del software con su puesta en funcionamiento. Más allá de las herramientas específicas, el propósito de este enfoque ha sido construir un puente automatizado y fiable entre la creación de una nueva funcionalidad y su entrega al usuario final. En esencia, se ha buscado crear una especie de çadena de montaje"para el software que agiliza todo el proceso, aumenta la calidad y reduce significativamente el riesgo de errores humanos.

El primer pilar de esta filosofía se ha centrado en la confianza y la calidad del código. Se ha establecido una red de seguridad automatizada que, cada vez que se integraba una nueva pieza de código en la rama principal de desarrollo, verificaba que todo el proyecto en su conjunto seguía funcionando correctamente. Este chequeo constante y automático ha proporcionado una enorme tranquilidad y ha servido como un control de calidad continuo, asegurando que la base del proyecto se mantuviera siempre estable y funcional.

El segundo pilar se ha enfocado en la agilidad y fiabilidad de la entrega. El proceso de llevar una nueva versión al servidor de producción, que tradicionalmente puede ser complejo y manual, se ha transformado en un procedimiento completamente automatizado. Al dar por finalizada una nueva versión, `Docker` [34] se encarga de empaquetar la aplicación, prepararla para su lanzamiento y desplegarla en el servidor sin necesidad de intervención manual. Esto garantiza que cada lanzamiento se realiza siempre de la misma manera y de forma segura.

3.3.3. GitActions

Para reforzar nuestra filosofía DevOps y articular de forma coherente el desarrollo y la entrega continua, en **LaBoticaDeMar** hemos adoptado **GitActions** como metodología central del proyecto. Más que una herramienta puntual, GitActions actúa como columna vertebral de nuestra "cadena de montaje" de software, orquestando cada paso desde el commit hasta el despliegue en producción de manera fiable y reproducible.

El primer pilar de GitActions se basa en la calidad y la consistencia del código. Cada vez que se genera una nueva rama o se actualiza la principal, un conjunto de workflows automáticos ejecuta compilaciones, pruebas unitarias e integradas, análisis estático de código y validación de estilos. Gracias a este proceso, cualquier regresión se detecta al instante y antes de fusionarse, garantizando que la rama principal permanezca siempre en un estado desplegable y que el equipo pueda confiar plenamente en la estabilidad de la base de código.

El segundo pilar se orienta a la agilidad y seguridad en el despliegue. Al completar con éxito las etapas de validación, GitActions empaqueta la aplicación, genera imágenes Docker versionadas y las publica en un registro privado. A continuación, otros workflows se encargan de desplegar dichas imágenes en nuestros entornos de pruebas y, finalmente, en producción, mediante estrategias de rolling update o blue–green deployment. Este flujo automatizado minimiza la intervención manual, reduce drásticamente el tiempo de entrega y uniformiza cada despliegue, logrando un proceso predecible, auditable y libre de errores humanos.

4

Descripción Informática

4.1. Requisitos

En esta sección se describen los requisitos funcionales y no funcionales que definen el comportamiento esperado de la aplicación. Estos requisitos fueron establecidos durante la fase inicial del proyecto en consulta con el cliente, y sirven de guía para el desarrollo.

4.1.1. Requisitos funcionales

Usuario no registrado

- **RF1. Búsqueda de productos.** El usuario puede buscar productos mediante un campo de texto.
- **RF2. Navegación por categorías.** El usuario puede explorar productos navegando por categorías y subcategorías.
- **RF3. Búsqueda por marca.** El usuario puede filtrar productos por marca.
- **RF4. Visualización de detalles.** El usuario puede consultar la ficha de un producto con descripción, precio, disponibilidad y valoraciones.

- **RF5. Añadir al carrito.** El usuario puede añadir productos al carrito de compra.
- **RF6. Añadir a favoritos.** El usuario puede añadir productos a una lista de favoritos.
- **RF7. Ver promociones.** El usuario puede ver ofertas, descuentos y productos destacados.
- **RF8. Filtrado avanzado.** El usuario puede aplicar múltiples filtros simultáneos (por ejemplo, precio, categoría, marca).
- **RF9. Creación de cuenta.** El usuario puede registrarse proporcionando datos personales y dirección de envío.

Usuario registrado

- **RF10. Inicio de sesión.** El usuario puede autenticarse con correo electrónico y contraseña.
- **RF11. Gestión de direcciones.** El usuario puede gestionar sus direcciones y datos de contacto.
- **RF12. Favoritos persistentes.** El usuario puede mantener favoritos vinculados a su cuenta.
- **RF13. Carrito persistente.** El usuario puede mantener su carrito vinculado a su cuenta.
- **RF14. Realizar pedidos.** El usuario puede completar un pedido eligiendo método de envío y pago.
- **RF15. Códigos de descuento.** El usuario puede aplicar códigos promocionales al realizar la compra.
- **RF16. Devoluciones.** El usuario puede solicitar devoluciones y seleccionar el motivo.
- **RF17. Historial de pedidos.** El usuario puede consultar el historial y estado de sus pedidos.
- **RF18. Historial de devoluciones.** El usuario puede consultar el historial de sus devoluciones.
- **RF19. Valoraciones.** El usuario puede valorar y comentar productos adquiridos.

Empleado

- **RF20. Gestión de pedidos.** El empleado puede ver y actualizar el estado de los pedidos de los usuarios.
- **RF21. Historial de devoluciones.** El empleado puede consultar devoluciones y sus motivos.
- **RF22. Actualización de stock.** El empleado puede actualizar el stock de productos.
- **RF23. Gestión de códigos de producto.** El empleado puede modificar el código interno de los productos.
- **RF24. Creación de productos.** El empleado puede crear nuevos productos.
- **RF25. Edición de productos.** El empleado puede editar los datos de un producto, excepto precio y descuento.

Administrador

- **RF26. Modificación de precios.** El administrador puede actualizar el precio de productos.
- **RF27. Modificación de descuentos.** El administrador puede actualizar los descuentos de productos.
- **RF28. Activación de productos.** El administrador puede activar o desactivar productos.
- **RF29. Gestión de usuarios empleados.** El administrador puede gestionar cuentas y permisos de empleados.
- **RF30. Gestión de categorías y marcas.** El administrador puede gestionar categorías y marcas.
- **RF31. Informes de ventas.** El administrador puede consultar informes de ventas y métricas.
- **RF32. Estadísticas de devoluciones.** El administrador puede consultar estadísticas de devoluciones.
- **RF33. Gestión de promociones.** El administrador puede controlar promociones y banners.
- **RF34. Acceso a datos de usuarios.** El administrador puede consultar datos de usuarios registrados.

4.1.2. Requisitos no funcionales

- **RNF1. Compatibilidad.** El sistema debe funcionar en Chrome, Firefox, Safari y Edge.
- **RNF2. Usabilidad.** La interfaz debe ser intuitiva y fácil de usar.
- **RNF3. Retroalimentación.** El sistema debe proporcionar mensajes claros tras cada acción.
- **RNF4. Diseño responsive.** La interfaz debe adaptarse a distintos tamaños de pantalla.
- **RNF5. Seguridad.** Los datos personales deben protegerse con conexiones cifradas y autenticación segura.

4.2. Estructura del proyecto: package-by-layer

Para estructurar el proyecto de manera clara y modular, se ha adoptado el empaquetado por capas **(package-by-layer)**. Este enfoque garantiza la **mantenibilidad** del código y la **separación de responsabilidades** entre las diferentes partes de la aplicación:

A continuación se describe la estructura de carpetas de la aplicación, con sus respectivas agrupaciones y objetivos, sin detallar cada clase.

- `src/main/java/es/laboticademar/webstore/`
  - **config**: Configuraciones globales de la aplicación.
  - **controllers**
    - `advice`: Manejo de errores y atributos comunes para controladores MVC.
    - `restControllers`: Endpoints REST para la API JSON.
    - `controllers`: Endpoints para servir vistas HTML.
  - **services**
    - `interfaces`: Contratos de servicio.
    - `impl`: Implementaciones de la lógica de negocio.
  - **repositories**: Persistencia con Spring Data JPA.
  - **entities**: Clases JPA que representan tablas de la base de datos.
  - **dto**: Objetos de transferencia de datos agrupados por funcionalidad (cart, producto, usuario, venta, wishlist, etc.).

- **mappers**: Componentes para mapeo y carga masiva de entidades y DTOs.
- **enumerations**: Enumeraciones del modelo de datos.
- **exceptions**: Excepciones de negocio.
- **security**
  - `auth`: Clases de petición y respuesta para autenticación.
  - `config`: Configuración JWT y servicios de seguridad.
- **utils**: Utilidades y clases auxiliares.

- `src/main/resources/`
  - `static/`
    - `css/`: `input.css`, `output.css`.
    - `images/`: Iconos SVG para la interfaz.
    - `js/`
      - `admin/`: Scripts para dashboard de administración.
      - `auth/`, `main/`, `navbar/`, `products/`, `purchases/`, `user/`: Scripts de funcionalidad general.
  - `application-*.properties`: Configuraciones de Spring Boot (dev, prod, etc.).

- `src/main/webapp/WEB-INF/`
  - `lib/`: Bibliotecas del lado del servidor.
  - `views/`
    - `admin/`, `error/`, `includes/`, `main/`, `product/`, `purchases/`, `user/`: Vistas JSP organizadas por módulo.
  - `web.xml`: Configuración del despliegue.

- `src/test/java/es/labotica de mar/webstore/`
  - `integration/`: Tests de integración.
  - `selenium/`: Tests E2E con Selenium.
  - `unit/`: Tests unitarios.

- `node_modules/`, `package.json`, `webpack.config.js`: Dependencias y configuración del frontend.

4.3. Arquitectura y análisis

La arquitectura de **LaBoticaDeMar** se basa en un modelo de tres capas bien definido: presentación, lógica de aplicación y persistencia. Esta organización modular facilita la separación de responsabilidades, mejora la mantenibilidad del sistema y permite escalar o modificar cada componente de forma independiente. A continuación, se analiza cada capa en detalle, junto con el flujo de comunicación entre ellas.

4.3.1. Capa de presentación (Frontend)

La capa de presentación actúa como el puente directo entre el usuario y el sistema, y su diseño arquitectónico es crucial para equilibrar el rendimiento, la experiencia de usuario y la mantenibilidad. Su misión principal es doble: capturar las interacciones del usuario de forma intuitiva y representar los datos generados por la lógica de negocio de manera clara y coherente.

Las decisiones arquitectónicas fundamentales para esta capa son las siguientes:

- **Modelo de interacción híbrido:** Se ha optado por un modelo de tres velocidades para optimizar la eficiencia de cada interacción. Para las cargas iniciales de página, se utiliza el **Renderizado en Servidor (SSR) con JSP**, lo que favorece el SEO y la velocidad de la primera carga. Para actualizaciones de secciones, se realizan llamadas **AJAX** que devuelven fragmentos de HTML renderizados en el servidor, actualizando partes de la vista sin una recarga completa. Finalmente, para micro-interacciones como añadir un producto al carrito, se usan llamadas AJAX que devuelven respuestas ligeras en formato **JSON**, permitiendo actualizaciones mínimas y en tiempo real en la interfaz.

- **Sistema de diseño basado en Tailwind CSS:** Se ha elegido estratégicamente el framework **Tailwind CSS** por su metodología de «utilidad primero» (*utility-first*). A diferencia de frameworks basados en componentes como Bootstrap, este enfoque proporciona clases atómicas de bajo nivel que otorgan un control creativo total sobre el diseño, evitando un aspecto genérico y permitiendo construir una identidad visual única. Esta metodología fomenta un sistema de diseño mantenible y genera una hoja de estilos final altamente optimizada, ya que solo incluye las clases que realmente se utilizan en el proyecto.

- **Persistencia de estado en cliente con `localStorage`:** Para mejorar la experiencia de los usuarios no registrados, la arquitectura contempla la persistencia de estado en el cliente. Se utiliza la API `localStorage` del

navegador para guardar datos volátiles como el contenido del carrito y la lista de deseos. Esta decisión reduce la fricción inicial, ya que un visitante puede cerrar el navegador y encontrar su selección intacta al volver, lo que funciona como una potente herramienta para fomentar la conversión a un usuario registrado.

4.3.2. Capa de aplicación (Backend)

La capa de aplicación representa el núcleo lógico de la arquitectura. Su función es orquestar el flujo de datos, aplicar las reglas de negocio y asegurar la coherencia y seguridad del sistema. Se ha desarrollado con **Spring Boot**, que proporciona una infraestructura robusta y estandarizada. El diseño se fundamenta en los siguientes patrones y decisiones arquitectónicas clave:

- **Patrón Modelo-Vista-Controlador (MVC):** Organiza el flujo de control, donde los `Controllers` actúan como puerta de entrada para las peticiones HTTP. Estos delegan la ejecución de la lógica a la capa de servicio y, finalmente, empaquetan los datos en un objeto `Model` que la `View` (JSP) renderiza para el usuario.
- **Arquitectura en capas y desacoplamiento:** La lógica de negocio está rigurosamente encapsulada en una capa de **Servicio** (`Service`). Esta capa intermedia no se comunica directamente con la base de datos, sino que lo hace a través de una capa de abstracción adicional: la capa de **Repositorio** (`Repository`). Los repositorios definen las operaciones de acceso a datos (ej. `findById`, `save`), y es Spring Data JPA quien implementa estas interfaces. Este diseño desacopla por completo la lógica de negocio de la tecnología de persistencia, favoreciendo la mantenibilidad y facilitando las pruebas unitarias.
- **Uso de DTOs (Data Transfer Objects):** Se utiliza este patrón para crear un contrato claro entre el `frontend` y el `backend` sin exponer el modelo de dominio interno (las entidades JPA). Esto mejora la seguridad, al ocultar campos sensibles, y aporta flexibilidad a la API.
- **Seguridad sin estado (*stateless*):** Se optó por una autenticación *stateless* mediante tokens **JWT**, gestionada con **Spring Security**. Esta decisión es estratégica para la escalabilidad horizontal, ya que elimina la dependencia de almacenar sesiones en el servidor.
- **Exposición de una API RESTful:** La capa define una interfaz HTTP basada en principios **RESTful** que se comunica mediante **JSON**, garantizando la máxima interoperabilidad con futuros clientes (ej. aplicaciones móviles).

4.3.3. Capa de persistencia (Base de datos)

La capa de persistencia constituye el cimiento sobre el cual se almacenan y gestionan todos los datos del sistema. La decisión arquitectónica ha sido utilizar una base de datos relacional (MySQL), una elección justificada por la naturaleza intrínsecamente estructurada y transaccional de una aplicación de comercio electrónico. Un modelo relacional es idóneo para garantizar la integridad de los datos mediante el uso de claves foráneas y transacciones (propiedades ACID), lo cual es crítico para la operativa del negocio.

El diseño del esquema de la base de datos, ilustrado en la Figura 4.1, se ha guiado por principios clave de la ingeniería de datos para asegurar su robustez, mantenibilidad y eficiencia. En primer lugar, se ha aplicado un riguroso proceso de Normalización para eliminar la redundancia y prevenir anomalías en los datos. Esto se refleja en el uso de tablas de consulta (lookup tables) como `laboratorio` o `categoria`, donde un dato se almacena una única vez y se referencia desde múltiples productos.

En segundo lugar, se ha optado por el uso de Claves Subrogadas (Surrogate Keys), utilizando un campo `ID` de tipo `bigint AUTO_INCREMENT` como clave primaria en la mayoría de las tablas. Esta estrategia, frente al uso de claves naturales, desacopla la lógica interna de la base de datos de los datos de negocio que podrían cambiar (como un correo electrónico) y garantiza una mayor eficiencia en los índices y las uniones (*joins*) entre tablas.

Finalmente, las relaciones de muchos a muchos (como la que existe entre un carrito y los productos) se han modelado correctamente mediante Tablas de Unión (ej. `cart_item`). Estas tablas no solo enlazan dos entidades, sino que también almacenan atributos propios de la relación, como la `cantidad`. La integridad referencial se garantiza a nivel de base de datos mediante el uso estricto de restricciones de clave foránea (*foreign key constraints*).

A continuación, se detalla la función de las tablas más importantes que componen este esquema:

- `usuario`**:** Es la entidad central del modelo, almacenando toda la información de identidad, contacto, credenciales de acceso (con la contraseña hasheada mediante Bcrypt) y los consentimientos legales y de marketing del cliente.
- `producto`**:** Representa el núcleo del catálogo. Contiene la información descriptiva, comercial (precio, stock) y de estado de cada artículo, destacando el campo booleano `activo` para controlar su visibilidad. La imagen del producto se gestiona como un tipo `BLOB` a nivel de la entidad JPA (`@Blob`), asegurando la atomicidad de los datos al ser parte integral del registro.
- `familia`**,** `categoria` **y** `subcategoria`**:** Estas tres tablas implementan una

jerarquía de clasificación para los productos. Este diseño normalizado evita la redundancia de datos y facilita la gestión y el filtrado del catálogo.

- `laboratorio` y `tipo_producto`: Actúan como tablas de consulta (*lookup tables*) adicionales para atribuir características específicas a los productos, manteniendo el modelo normalizado.

- `shopping_cart` y `cart_item`: La tabla `shopping_cart` se vincula de forma única a cada usuario. La tabla de unión `cart_item` materializa la relación muchos a muchos entre el carrito y los productos, almacenando la cantidad específica de cada artículo.

- `venta` y `detalle_venta`: Modelan las transacciones completadas. La tabla `venta` guarda los datos globales del pedido, mientras que `detalle_venta` crea una instantánea histórica de cada línea de compra, almacenando el `precio_unitario` en el momento de la transacción para garantizar la integridad de los registros a lo largo del tiempo.

- `devolución` y `detalle_devolución`: Gestionan el proceso de devoluciones, manteniendo una relación con la venta original para una trazabilidad completa. El campo `motivo` utiliza un tipo `ENUM` para estandarizar las causas de la devolución.

- `usuario_roles`: Es la tabla de unión que implementa un sistema de Autorización Basado en Roles (RBAC), permitiendo que un mismo usuario pueda tener múltiples roles (ej. "USUARIO", "ADMIN").

- `usuario_preferencias`: Almacena un registro con el id de cada preferencia seleccionada por el usuario, para poder manejarlas con el `ENUM` de la app.

- `wishlist` y `wishlist_producto`: Siguen un patrón similar al del carrito de la compra para implementar la funcionalidad de la lista de deseos.

- `evaluación`: Almacena cada evaluación asociada a un producto y registra que usuario la realizó.

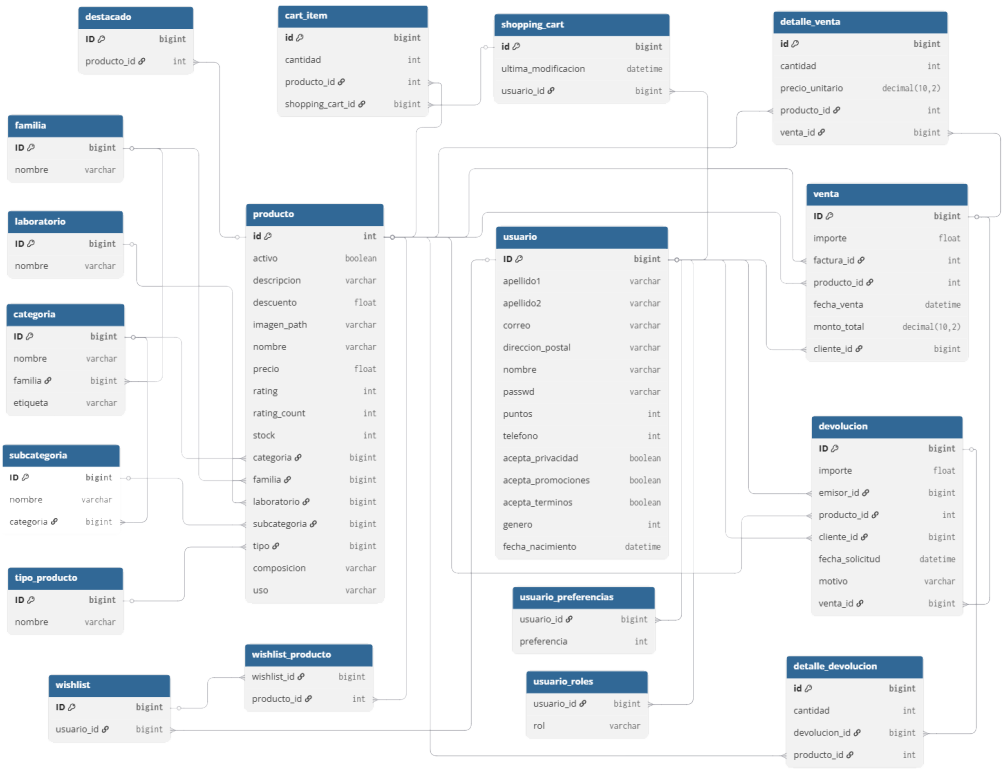


Figura 4.1: Esquema Entidad-Relación de la base de datos del proyecto.

4.3.4. Comunicación entre capas

La arquitectura en tres capas garantiza una separación clara de responsabilidades, donde cada capa se comunica únicamente con sus adyacentes a través de interfaces y contratos bien definidos. Este modelo de comunicación controlada es fundamental para asegurar un diseño desacoplado, mantenible y seguro [4.2].

- **Presentación ↔ Aplicación (Frontend ↔ Backend):** [4.2] La interacción entre el cliente y el servidor se materializa a través de dos patrones de comunicación HTTP complementarios:

  1. **Renderizado en servidor para carga de páginas:** Para la navegación principal y la carga inicial de vistas, se sigue un flujo síncrono. Una petición del navegador (ej. `GET /productos`) es recibida por un `@Controller` de Spring. Este invoca a la capa de servicio para obtener los datos necesarios (ej. un listado de productos destacados), los empaqueta en un objeto `Model` y devuelve el nombre de una vista JSP. El servidor renderiza la página HTML completa, inyectando los datos

del modelo en ella, y la envía al navegador. Este enfoque es ideal para el SEO, ya que al usar cargas dinámicas las herramienta de SEO interpretan plantillas vacías que no cumplen ninguna regla de búsqueda, además conseguimos una rápida primera impresión de carga.

2. **Interacción asíncrona (AJAX) para actualizaciones dinámicas:** Para operaciones que requieren una alta fluidez, como añadir un producto al carrito o aplicar un filtro, se utilizan llamadas asíncronas. El JavaScript del `frontend`, mediante la **API Fetch**, realiza peticiones a `endpoints` RESTful específicos como `POST /api/cart/add_item`. Estos son gestionados por un `@RestController` en el `backend`, que responde únicamente con los datos necesarios en formato **JSON** si fuese necesaria información de vuelta (ej. `{ïtemCount": 5}`), o también puede suceder que el cliente no espere ningún valor en concreto de vuelta, será entonces cuando el `@RestController` devolverá una simple `ResponseEntity.ok().build()`, con esto será suficiente para indicar al cliente que la acción que se debía realizar se llevo a cabo con éxito y se podrá continuar con el flujo que se estuviese llevando a cabo. El JavaScript del cliente recibe esta respuesta y actualiza únicamente los fragmentos del DOM pertinentes (como el icono del carrito), evitando una recarga completa de la página y creando una experiencia de usuario ágil y moderna.

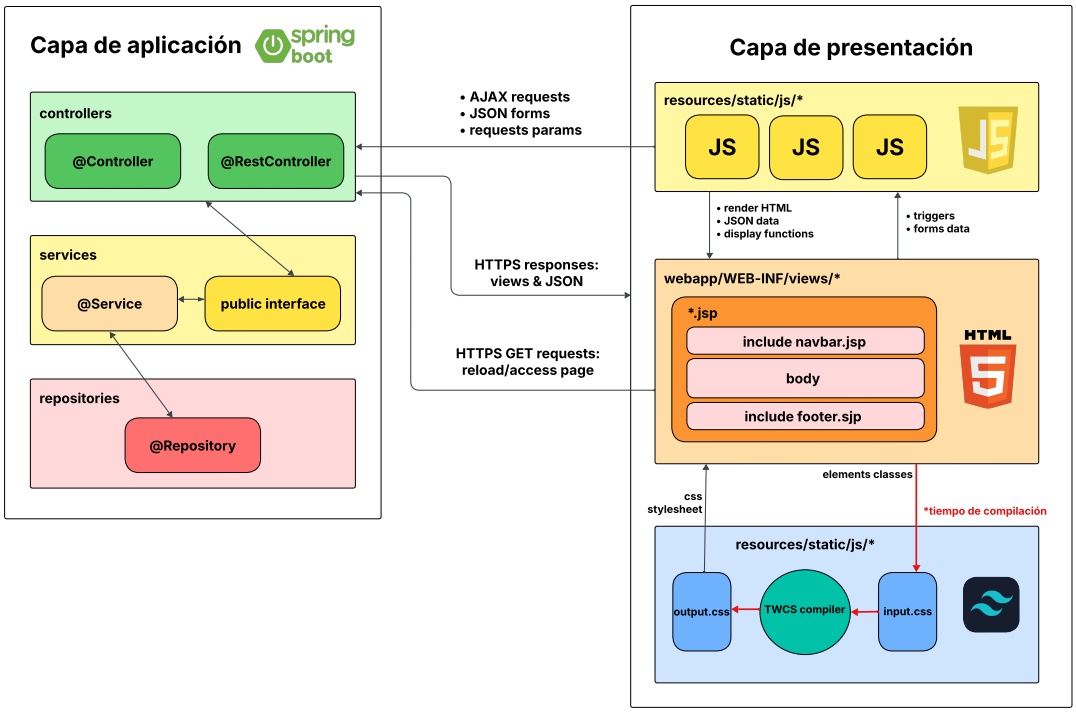


Figura 4.2: Esquema de la comunicación entre la capa de aplicación y la capa de presentación.

- **Aplicación ↔ Persistencia (Backend ↔ Base de Datos):** [4.3] La comunicación entre la lógica de negocio y la base de datos está completamente abstraída para maximizar la flexibilidad y reducir el acoplamiento.

  La capa de servicio no interactúa directamente con la base de datos, sino que lo hace a través de interfaces de Repositorio (ej. `ProductRepository`), siguiendo el patrón de diseño **DAO (Data Access Object)**. **Spring Data JPA** se encarga de generar automáticamente las implementaciones de estas interfaces en tiempo de ejecución, proporcionando métodos CRUD estándar y la capacidad de crear consultas derivadas a partir del nombre de los métodos (ej. `findByActivoTrue()`).

  Bajo esta capa de abstracción se encuentra **Hibernate**, el motor de **Mapeo Objeto-Relacional (ORM)**. Hibernate es el traductor que convierte las llamadas a métodos en los repositorios (ej. `repository.save(producto)`) en sentencias SQL optimizadas (`INSERT` o `UPDATE`) que la base de datos puede entender. De la misma manera, transforma los resultados de las consultas (`SELECT`) de tablas y filas en objetos Java plenamente funcionales. Esta arquitectura permite que la lógica de negocio sea agnóstica a la base de datos subyacente, pudiendo cambiar de MySQL a H2, por ejemplo, con mínimas modificaciones en la configuración.

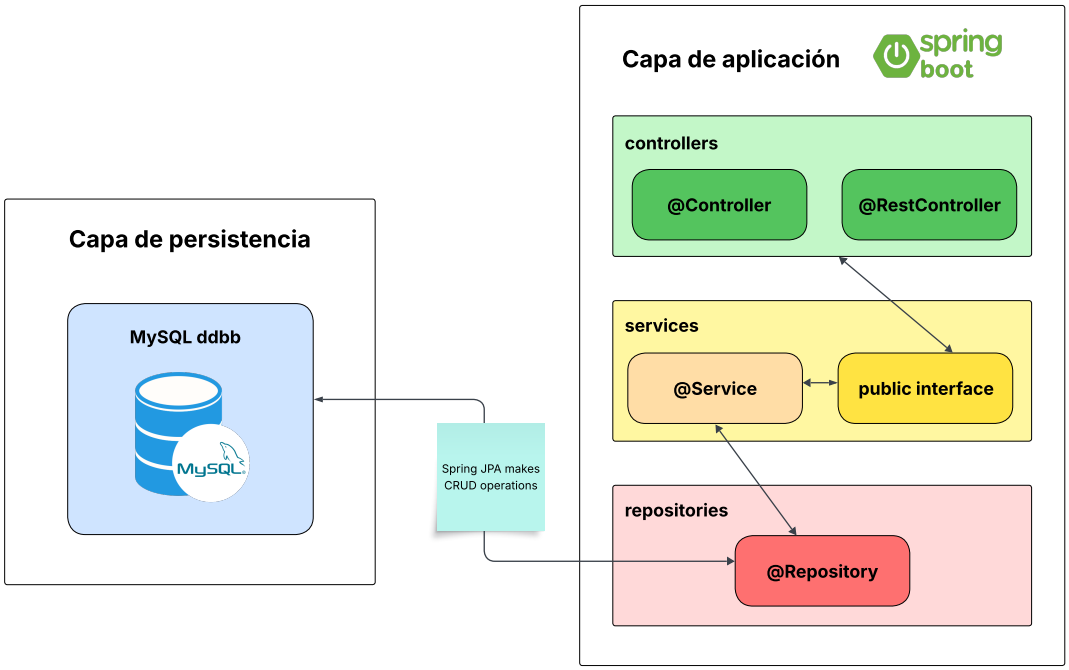


Figura 4.3: Esquema de la comunicación entre la capa de persistencia y la capa de aplicación.

4.3.5. Arquitectura global

La arquitectura global de **LaBoticaDeMar** está diseñada para ofrecer alta disponibilidad, seguridad y facilidad de escalado, distribuyendo responsabilidades entre el punto de entrada, la capa de aplicación y la persistencia de datos [4.4].

- **Plesk (Gestión de hosting y DNS):** Administración centralizada del dominio `laboticademar.com`, gestión de certificados SSL/TLS y configuración de entornos, garantizando seguridad perimetral y agilidad en el despliegue.

- **NGINX (Proxy inverso:)** Punto de entrada único en la instancia EC2 que maneja SSL offloading, balanceo de carga, cacheado de estáticos y enrutamiento a los contenedores, optimizando rendimiento y protección de la aplicación.

- **Docker y aplicación Spring Boot**: Contenedor que aísla el runtime de Java y todas las dependencias de la aplicación, facilitando portabilidad entre entornos y despliegues inmutables mediante imágenes versionadas.

- **Instancia EC2 en AWS**: Servidor virtual configurado para ejecutar NGINX y los contenedores Docker, con recursos ajustables (CPU, memoria, almacenamiento) para escalar de forma vertical según la demanda.

- **RDS MySQL (Persistencia):** Base de datos relacional gestionada en AWS, con backups automáticos, replicación y escalado de almacenamiento, que garantiza integridad, disponibilidad y continuidad de los datos.

En conjunto, estos componentes definen una infraestructura modular y distribuida, que maximiza la resiliencia y la flexibilidad para adaptarse a crecientes volúmenes de tráfico y futuras evoluciones del sistema.

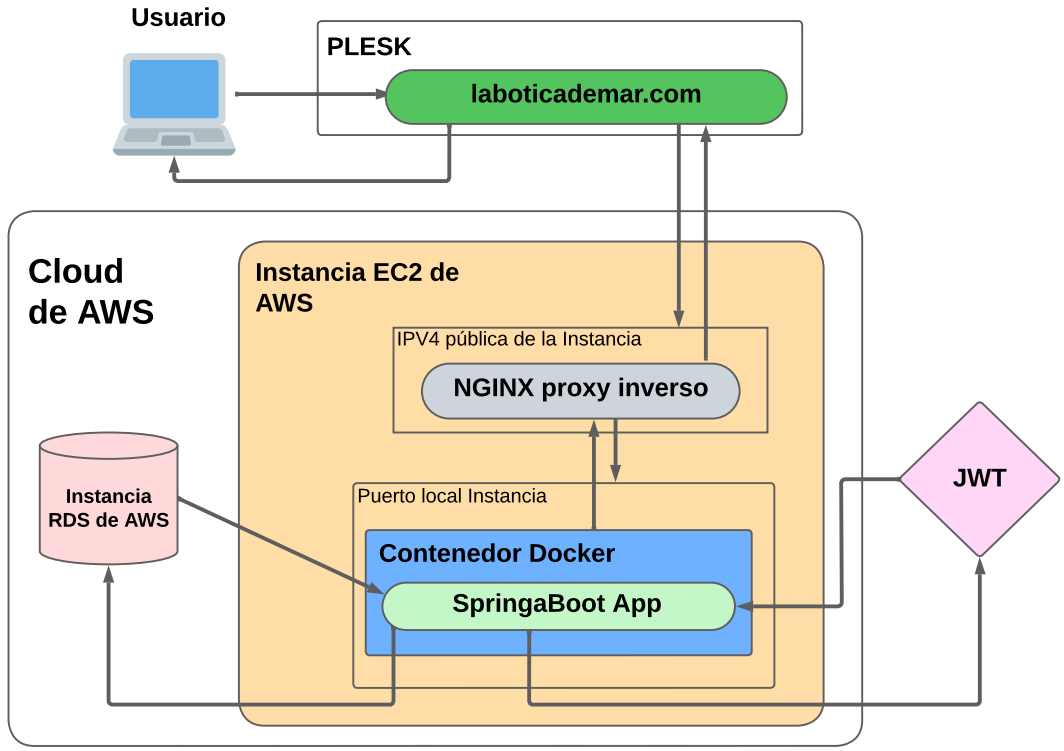


Figura 4.4: Arquitectura global del proyecto.

4.4. Diseño e Implementación

Esta sección profundiza en cómo se han aplicado las técnicas, herramientas y fragmentos de código específicos que se han utilizado para materializar las decisiones arquitectónicas previamente analizadas.

Toda la implementación del código puede consultarse en el repositorio público disponible en: `https://github.com/alex-m-perez/LaBoticaDeMar`.

4.4.1. Frontend

El frontend de la aplicación se ha implementado siguiendo un enfoque de renderizado en el servidor (SSR) con JavaServer Pages (JSP), progresivamente mejorado con una capa de interactividad dinámica mediante JavaScript. Esta arquitectura híbrida permite aprovechar la robustez y el buen rendimiento en la carga inicial del SSR, al tiempo que ofrece una experiencia de usuario fluida y reactiva, similar a la de una Single-Page Application (SPA), en las interacciones posteriores.

Una de las decisiones clave en el diseño ha sido la **encapsulación de la lógica de cliente en ficheros JavaScript modulares**. Cada página o componente complejo tiene su propio fichero `.js` asociado, que se encarga de gestionar sus eventos, sus llamadas AJAX y sus manipulaciones del DOM. Esta separación de responsabilidades entre la estructura (JSP) y el comportamiento (JS) mejora drásticamente la mantenibilidad y la organización del código.

Estructura y componentes JSP

La base de la interfaz se construye con un sistema de plantillas JSP. Se utilizan componentes reutilizables como `navbar.jsp` y `footer.jsp`, que se incluyen en todas las páginas principales para garantizar la consistencia visual. Las librerías JSTL son fundamentales para la generación de contenido dinámico. A continuación, se detalla el propósito de las vistas y componentes más relevantes:

- `welcome.jsp`: [4.5] Actúa como la página de inicio. Su principal responsabilidad es presentar una introducción visualmente atractiva a la tienda, utilizando componentes como carruseles de imágenes y una selección de productos destacados que son cargados desde el `backend` y renderizados con bucles de JSTL.

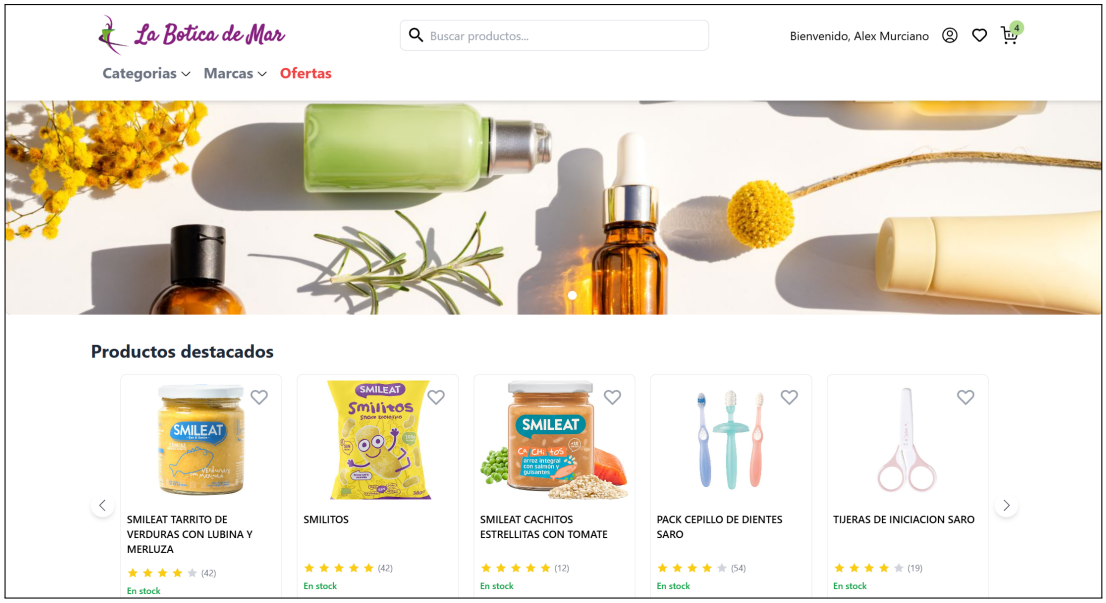


Figura 4.5: Vista principal de la página de inicio.

- `registro.jsp`: [4.5] El formulario de registro se organiza en tres pasos secuenciales para facilitar la experiencia del usuario. En el primer paso [4.6] se solicitan los datos personales básicos, como el nombre, apellidos, correo electrónico y contraseña. El segundo paso [4.7] recopila la información de

dirección postal, incluyendo calle, ciudad, código postal y país. Finalmente, el tercer paso [4.8] presenta una lista de preferencias relacionadas con intereses o categorías de productos, que el usuario puede seleccionar libremente para personalizar su experiencia dentro de la plataforma.

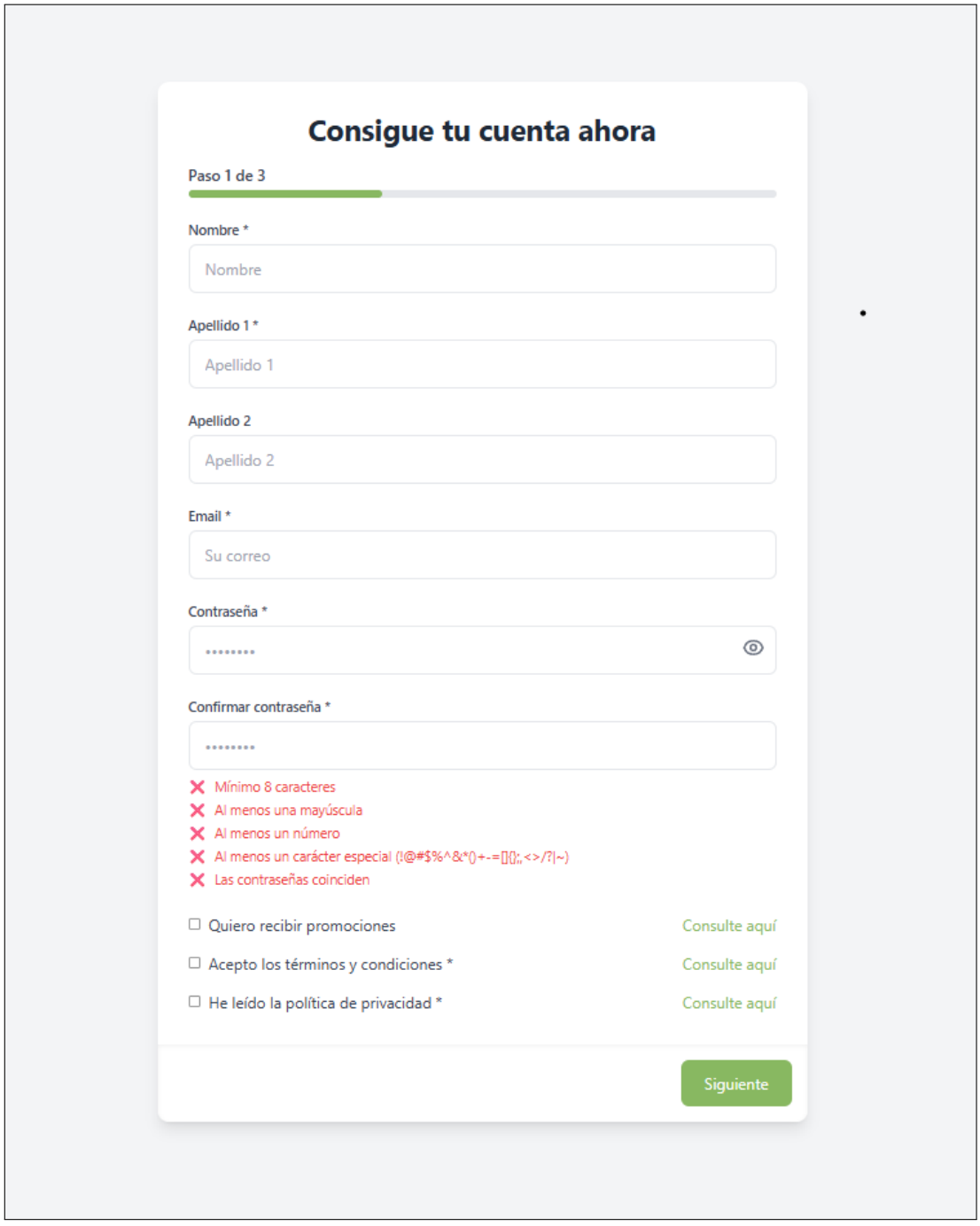


Figura 4.6: Etapa 1 del formulario de registro.

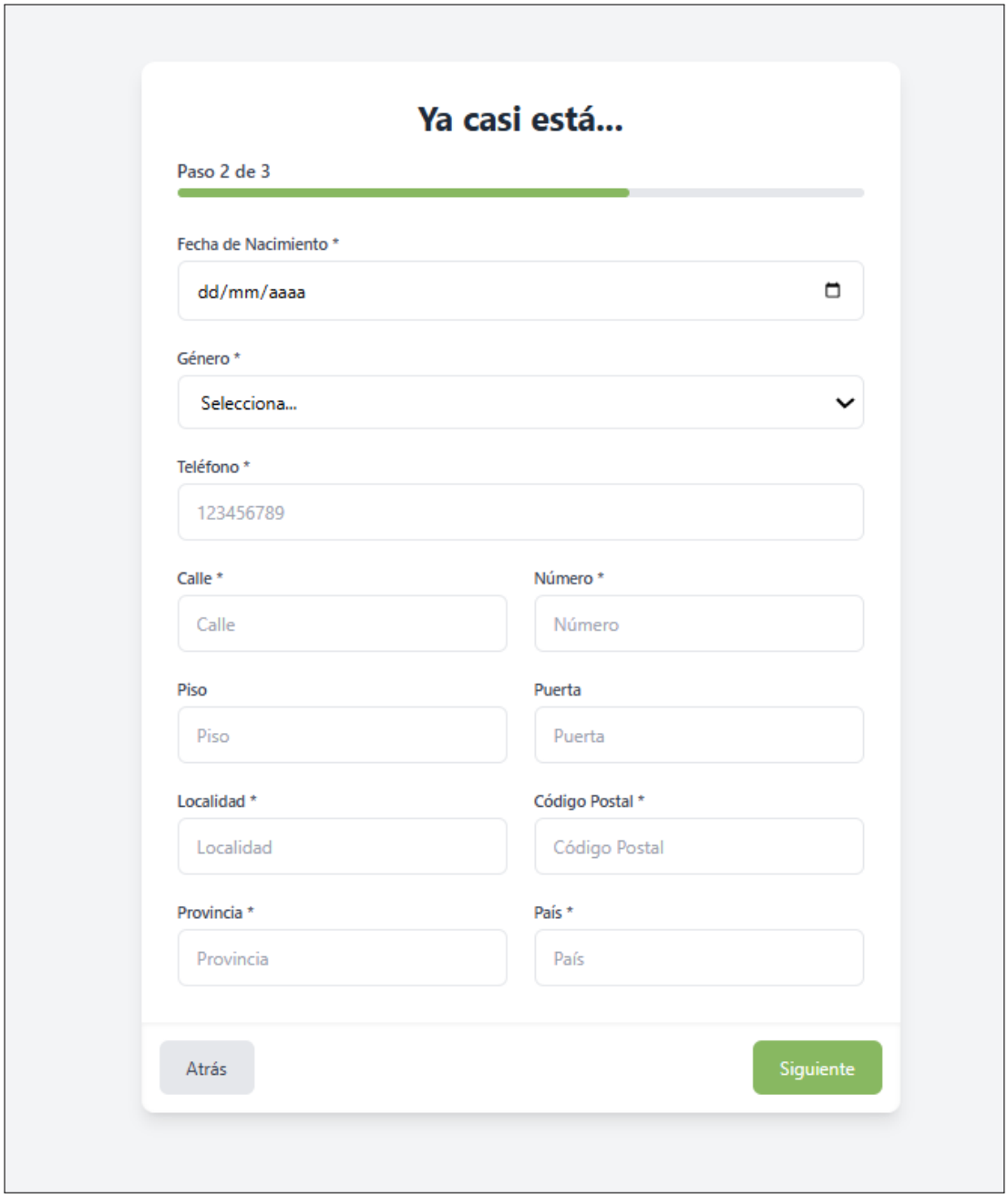


Figura 4.7: Etapa 2 del formulario de registro.

Figura 4.8: Etapa 3 del formulario de registro.

- `home.jsp` **(panel de administrador):** Es el punto de entrada al panel de administración. Funciona como una aplicación de tipo "shell.º contenedor. La página principal contiene la barra de navegación entre las distintas secciones y un área de contenido principal donde los diferentes módulos de gestión se cargan de forma dinámica mediante AJAX, sin necesidad de recargar la página completa. Las principales secciones son:

    1. **Gestión de productos:**[4.9] Es el núcleo del panel y permite una gestión integral del catálogo. Ofrece funcionalidades CRUD (Crear, Leer, Actualizar, Eliminar) para los productos. Incluye un potente formulario de filtrado para localizar artículos por múltiples atributos (código, nombre, categoría, stock, etc.) y la capacidad de realizar cargas masivas de productos mediante ficheros.

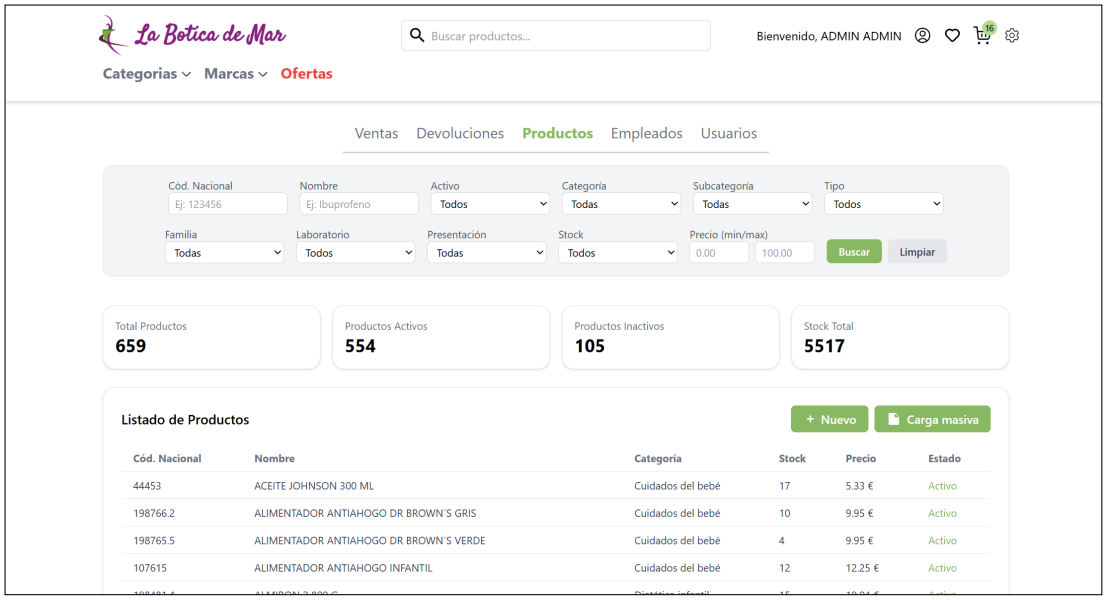


Figura 4.9: Módulo de gestión de productos dentro del panel de administrador.

Para la edición de un producto existente, se ha implementado una interfaz basada en ventanas modales [4.10]. Al hacer clic sobre una fila en la tabla de productos, se abre un modal que se rellena automáticamente con los datos del artículo seleccionado, permitiendo al administrador modificar sus atributos y guardar los cambios. Este patrón de interacción, que consiste en mostrar un listado y utilizar un modal para la edición del detalle, se ha aplicado de forma consistente en el resto de pestañas del panel de administración, garantizando una experiencia de usuario coherente en toda la sección.

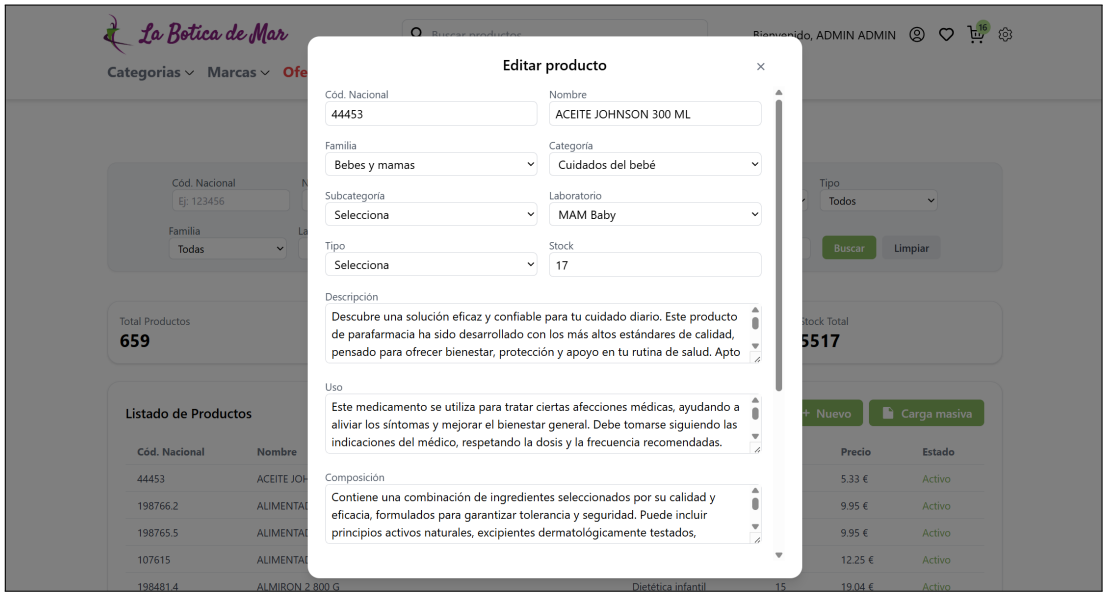


Figura 4.10: Módulo de gestión de productos dentro del panel de administrador.

2. **Gestión de ventas:** [4.11] Este módulo presenta un listado completo de todas las ventas realizadas. Permite al administrador buscar y filtrar pedidos por distintos criterios (fecha, cliente, importe) y acceder al detalle de cada venta para revisar los productos incluidos y el estado del envío.

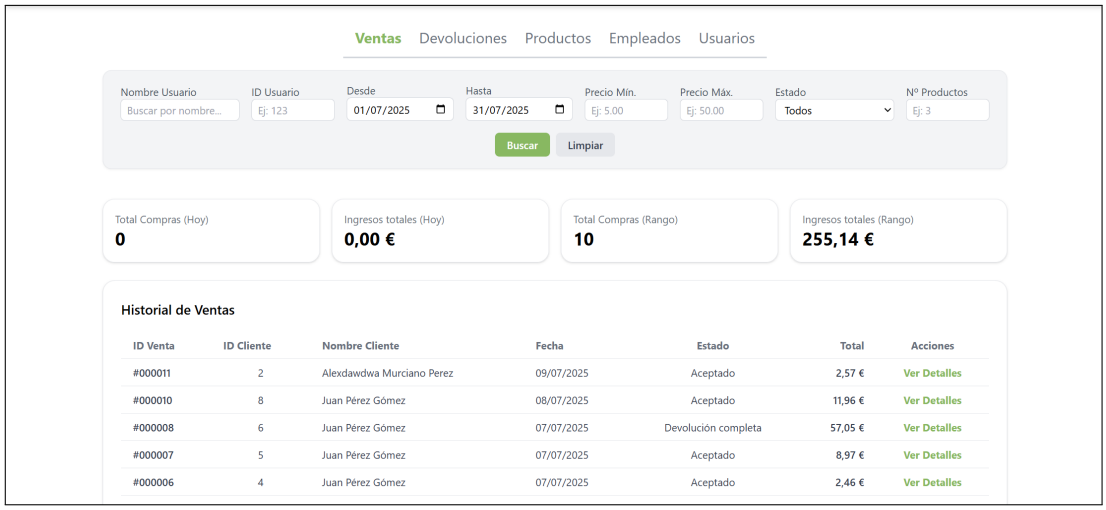


Figura 4.11: Modal con los detalles del producto.

Desde el modal de venta [4.12], el administrador puede consultar todos los detalles asociados a un pedido específico, incluyendo los productos adquiridos, el importe total y la información del cliente. Además, se incluye un botón "Actualizar estado"que abre un segundo modal [**??**], desde el cual es posible modificar el estado

del pedido (por ejemplo: En preparación, Enviado, Entregado). Esta funcionalidad permite dar seguimiento al proceso de compra y ofrecer retroalimentación actualizada al usuario sobre el estado de su pedido.

Figura 4.12: Modal de detalles de venta.

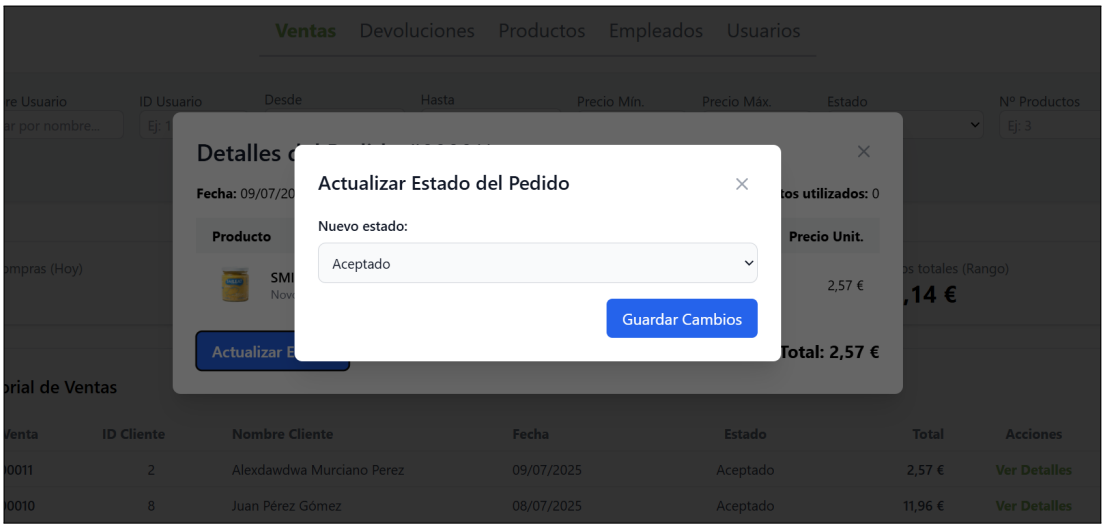


Figura 4.13: Modal para el cambio de estado de pedido.

3. **Gestión de devoluciones:** [4.14] Centraliza todas las solicitudes de devolución iniciadas por los clientes. El administrador puede revisar el motivo de cada solicitud, consultar la venta original asociada y gestionar el estado de la devolución (ej. pendiente, aprobada, rechazada).

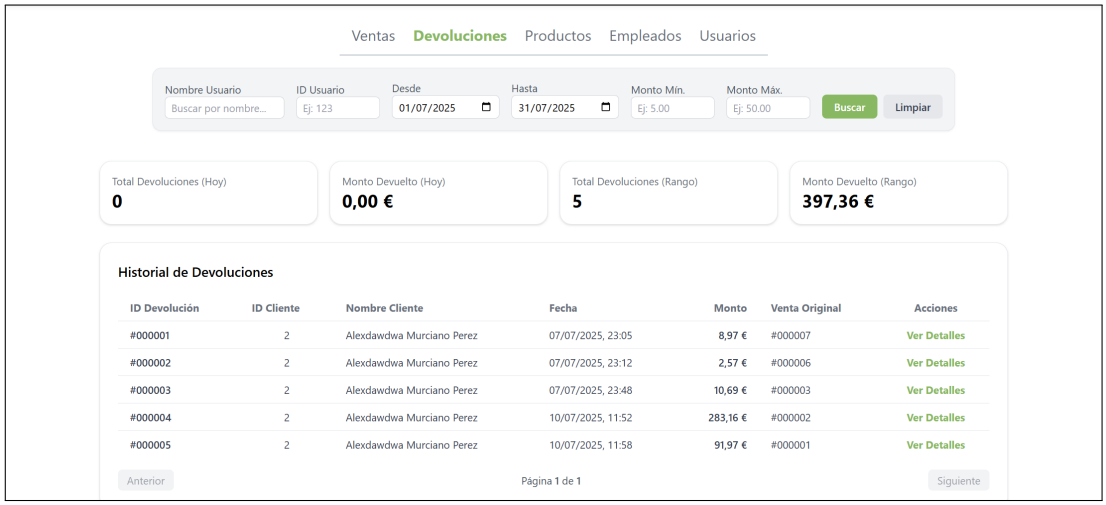


Figura 4.14: Módulo de gestión de devoluciones dentro del panel de administrador.

Al pulsar sobre una devolución en el listado, se muestra un modal con todos los detalles relacionados. Desde este panel emergente [4.15], el administrador puede revisar la información de la devolución, incluyendo el motivo, los productos involucrados, el estado actual del proceso y cualquier observación registrada por el cliente o el equipo de atención.

Figura 4.15: Modal para ficha de detalles de devolución.

4. **Gestión de empleados:** [4.16] Permite administrar las cuentas de usuario del personal con acceso al panel. Desde aquí se pueden crear nuevos perfiles de empleado, modificar sus datos y asignarles los roles y permisos correspondientes dentro del sistema.

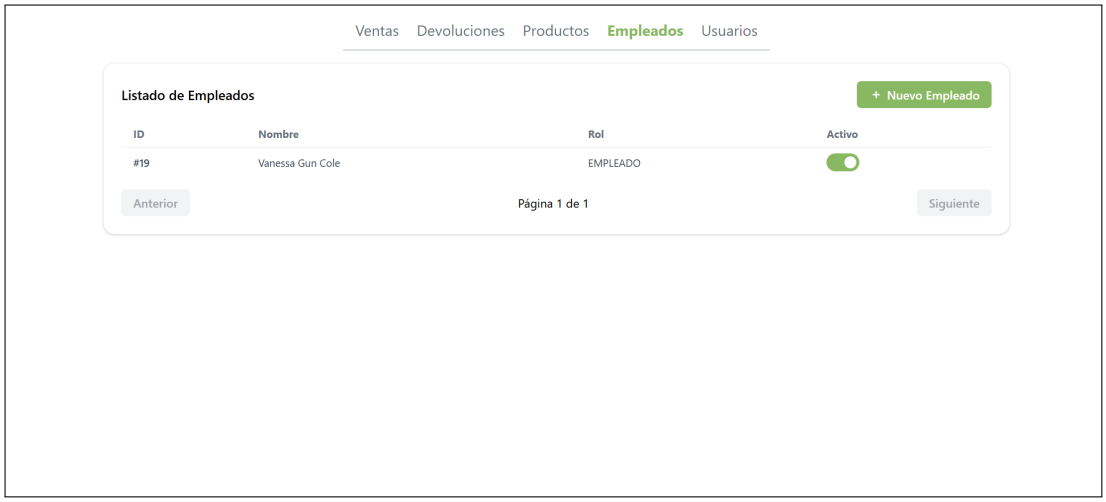


Figura 4.16: Módulo de gestión de empleados.

Al pulsar sobre una devolución en el listado, se muestra un modal [4.17] con todos los detalles relacionados. Desde este panel emergente, el administrador puede revisar la información de la devolución, incluyendo el motivo, los productos involucrados, el estado actual del proceso y cualquier observación registrada por el cliente o el equipo de atención.

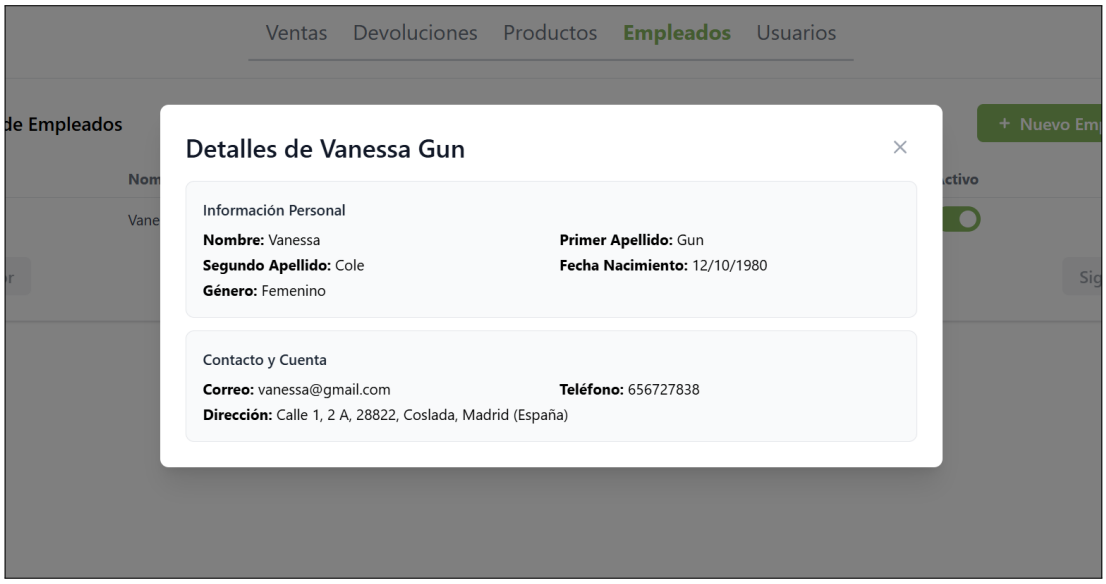


Figura 4.17: Modal para ficha de detalles de empleado.

5. **Gestión de usuarios:** [4.18] Ofrece una vista estadística de los clientes registrados en la plataforma, destacando a las personas que más compran, las que más gastan y aquellas que realizan más devoluciones. Esta información permite al administrador identi-

ficar patrones de comportamiento y tomar decisiones basadas en los hábitos de los usuarios.

Figura 4.18: Módulo de gestión de usuarios dentro del panel de administrador.

Al pulsar sobre un usuario en el listado, se muestra un modal [4.19] con los detalles completos de su perfil, incluyendo información de contacto, historial de compras, devoluciones y actividad reciente en la plataforma.

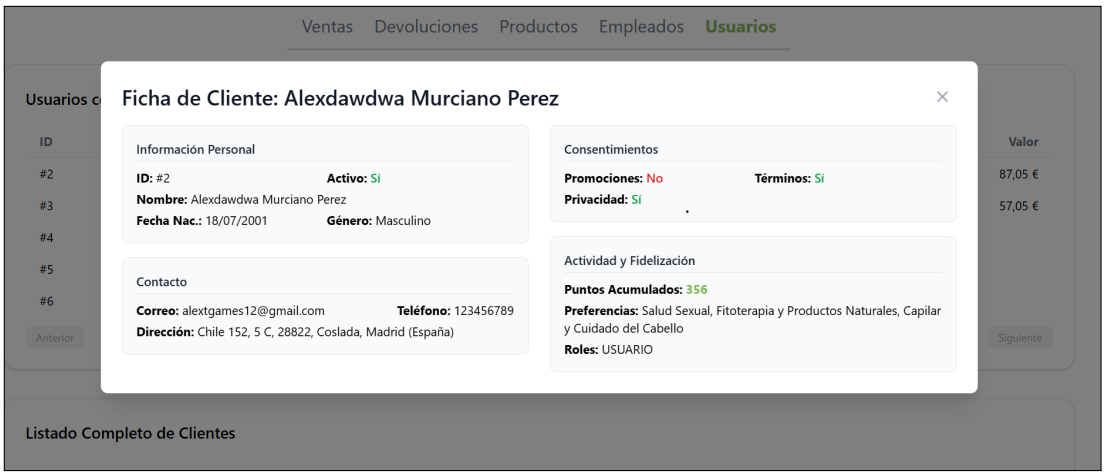


Figura 4.19: Módulo de gestión de usuarios dentro del panel de administrador.

- `products.jsp` **(Catálogo de Productos):**[4.20] Es la vista principal del catálogo para los usuarios. Su diseño presenta una barra lateral con un sistema de filtros interactivos, una parrilla para mostrar los productos y, en la parte superior, una ruta de navegación (*breadcrumbs*) [4.22] que se actualiza dinámicamente para reflejar los filtros de categoría aplicados. La carga inicial de la página se realiza desde el servidor, pero toda la interacción

posterior, como la aplicación de filtros o la paginación, es gestionada por JavaScript. Este se comunica con la API para actualizar únicamente la sección de la parrilla de productos sin recargar la página. En caso de que ninguna coincidencia sea encontrada, esta misma área muestra un mensaje de retroalimentación [4.21] visual al usuario.

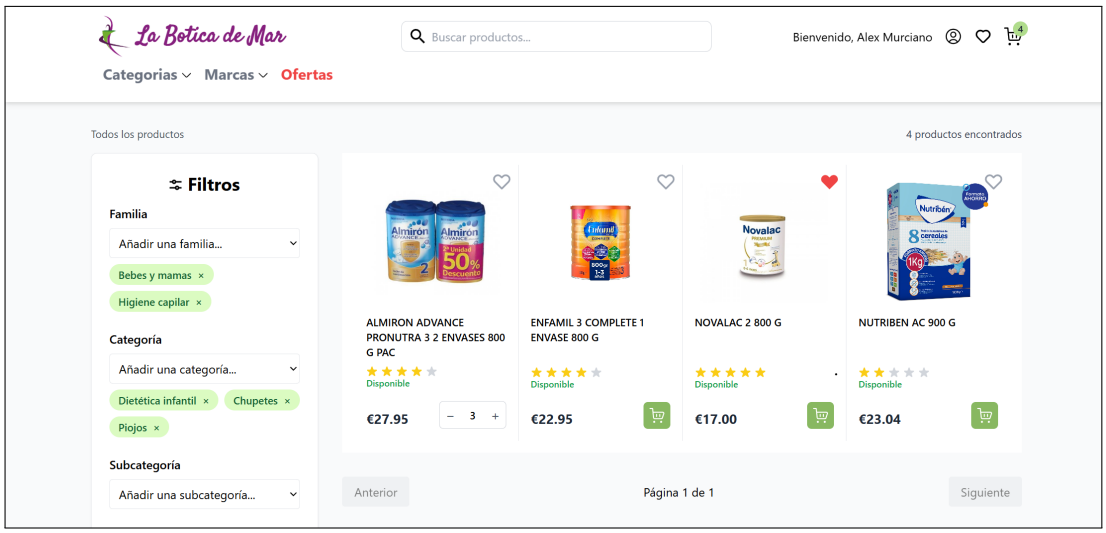


Figura 4.20: Vista del catálogo de productos con filtros laterales.

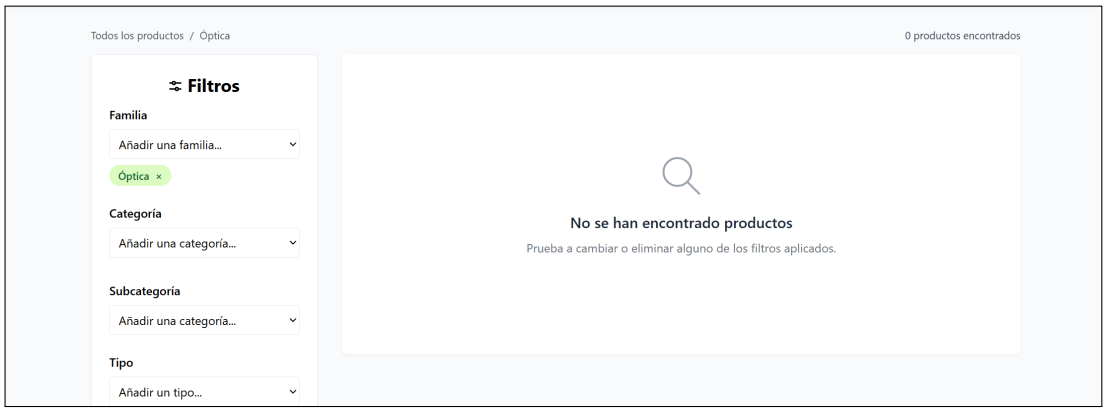


Figura 4.21: Mensaje mostrado en el catálogo cuando no se encuentran productos.

Figura 4.22: Migas de pan.

La sección de filtros se sitúa en el lateral izquierdo de la interfaz y agrupa todos los criterios de búsqueda de forma clara y accesible [4.23]. Cada bloque

Familia, *Categoría*, *Subcategoría* y *Tipo* se presenta como desplegable con búsqueda integrada; los valores seleccionados se muestran inmediatamente como *píldoras* verdes extraíbles para facilitar su edición. Debajo, el filtro de *Laboratorio* ofrece un selector similar que, por defecto, incluye Todos los laboratorios" y permite escoger uno o varios fabricantes cuyos nombres quedan también reflejados como *píldoras* eliminables. Más abajo, dos controles booleanos —*Sólo en stock* y *Oferta activa*— se representan mediante casillas de verificación intuitivas que refinan instantáneamente la búsqueda. El filtro de *Precio (€)* incorpora dos campos numéricos para rango mínimo y máximo, editables manualmente, y un deslizador interactivo que ajusta gráficamente los valores. Al pie, los botones `Filtrar` (estilo sólido) y `Limpiar` (contorno) ejecutan las acciones de envío o reinicio de criterios.

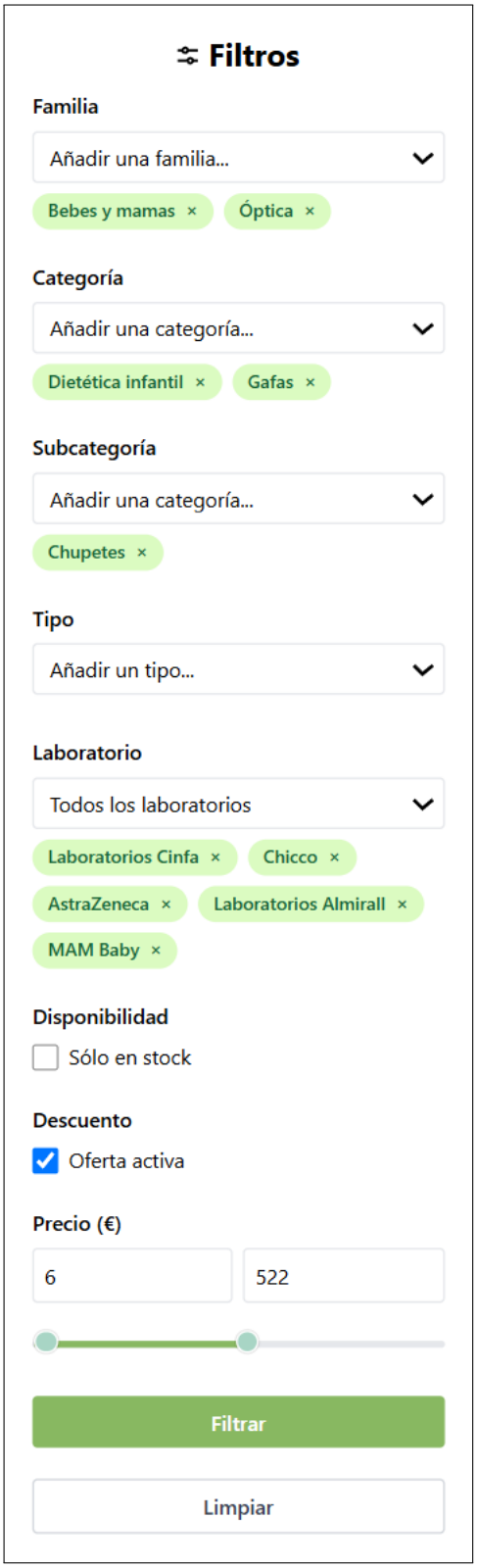


Figura 4.23: Sección de filtros.

- `product_info.jsp` **(detalle de dpoducto):** [4.24] Esta es la vista dedicada a mostrar toda la información de un único producto, diseñada para ofrecer al cliente todos los datos necesarios para tomar una decisión de compra. La página presenta la información detallada del artículo, incluyendo su imagen, nombre, precio y la valoración media (*rating*) otorgada por otros usuarios. Para organizar el contenido, se utilizan tres secciones o pestañas bien diferenciadas que detallan la **descripción**, el **modo de uso** y la **composición**. De forma prominente, se incluyen los botones de acción principales para **añadir el producto al carrito** y a la **lista de deseos**. Finalmente, en la parte inferior de la página, se ha implementado un **carrusel de productos recomendados** [4.25] para fomentar la venta cruzada (*cross-selling*), sugiriendo artículos relacionados o populares.

Figura 4.24: Vista de producto en detalle.

Figura 4.25: Carusel de productos recomendados.

- `shopping_cart.jsp` [4.26] y `wishlist.jsp` [4.27]: Son vistas dedicadas que muestran listados de productos específicos del usuario. Utilizan JSTL para iterar sobre los datos proporcionados por el `backend` y presentar la información en tablas claras y estructuradas. Toda la interactividad (cambiar cantidades, eliminar productos) se delega a sus scripts de cliente asociados.

  En la vista del carrito de la compra se incluye, además, un módulo de pago que solicita al usuario el número de tarjeta, la fecha de caducidad y el código CVC. Este formulario valida que los datos introducidos cumplan con el formato correcto antes de permitir la transacción. El módulo de pago solo se habilita si el usuario está autenticado, por lo que el proceso de compra requiere haber iniciado sesión previamente en la plataforma.

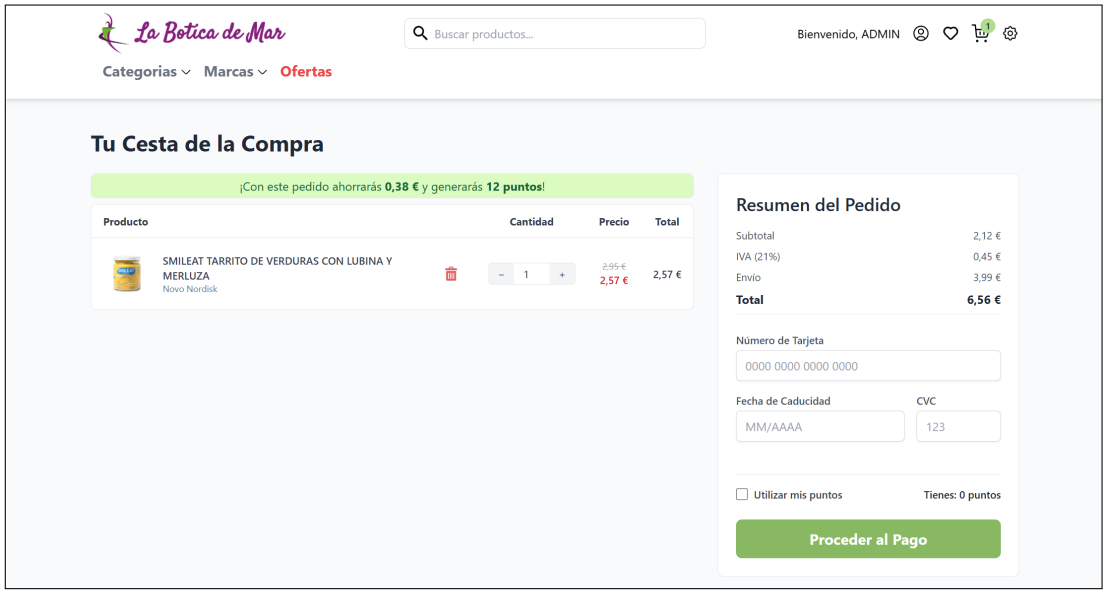


Figura 4.26: Vista del carrito de la compra.

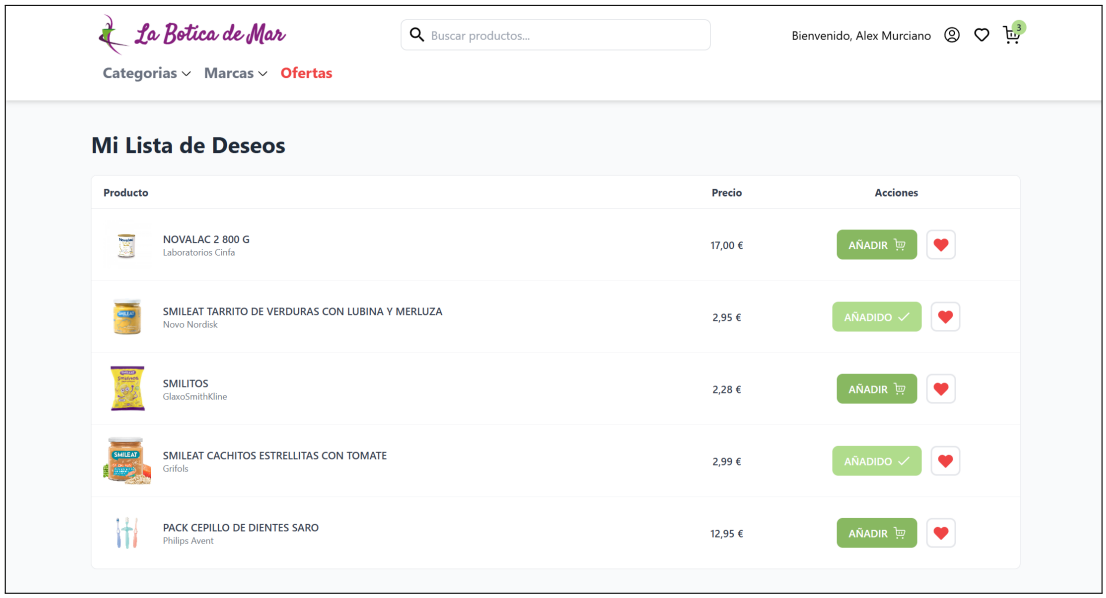


Figura 4.27: Vista de la lista de deseos.

- **Componentes Reutilizables (`navbar.jsp` y `footer.jsp`):** Para garantizar la consistencia, la aplicación se apoya en módulos que se inyectan en las vistas principales. El componente más complejo es la `navbar.jsp`, que centraliza la navegación y varias funcionalidades interactivas clave:

  1. **Autorización Visual:** La estructura de la barra de navegación se adapta al rol del usuario mediante la **librería de etiquetas de Spring Security**. El botón de acceso al panel de administración, por ejemplo,

está encapsulado en una etiqueta `<sec:authorize access="hasRole('ADMIN')">`, asegurando que el HTML de este elemento solo se envíe al navegador si el usuario cumple con dicho rol.

2. **Barra de Búsqueda Predictiva:** [4.28] Se ha implementado una funcionalidad de autocompletar. A medida que el usuario escribe, un script de JavaScript envía peticiones AJAX a un `endpoint` del backend, que devuelve una lista de nombres de productos coincidentes para construir una lista de sugerencias dinámica.

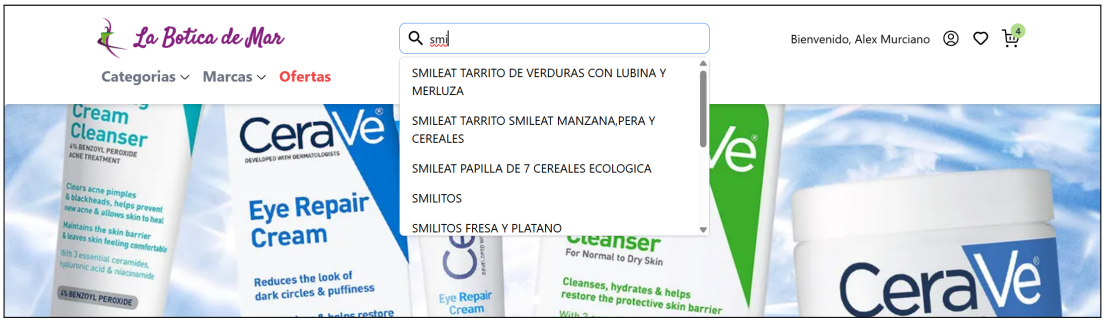


Figura 4.28: Ejemplo de la barra de búsqueda con sugerencias predictivas.

3. **Menú de Categorías Jerárquico:** [4.29] El menú Çategorías"despliega un mega-menú interactivo. La lógica, gestionada por JavaScript, muestra las familias de productos y, al pasar el cursor sobre una, actualiza al instante una segunda columna con las categorías específicas de esa familia.

Figura 4.29: Despliegue del menú de categorías jerárquico.

4. **Navegador de Marcas por índice alfabético:** [4.30] La sección "Marcas"presenta un índice alfabético. La interacción del usuario con una letra filtra y muestra dinámicamente en una zona adyacente todas las marcas que comienzan por esa inicial.

Figura 4.30: Despliegue del navegador de marcas por índice alfabético.

Lógica de cliente y encapsulación en JavaScript

El comportamiento dinámico de la aplicación reside en su capa de JavaScript, que se ha estructurado en módulos con responsabilidades bien definidas. Cada script encapsula la lógica de una vista o componente específico, comunicándose con otros a través de un sistema de eventos desacoplado.

- **Gestión del panel de administración (`products.js` y similares):** La interactividad de los distintos módulos del panel de administración (productos, ventas, etc.) se gestiona mediante scripts dedicados. El más complejo, `products.js`, sirve como plantilla para los demás y encapsula toda la lógica de la sección de gestión de productos. Sus responsabilidades son:
  - **Renderizado y paginación asíncrona:** Gestiona la comunicación con la API para obtener listados paginados de entidades. Al aplicar filtros o cambiar de página, envía una petición AJAX, recibe los datos en formato JSON y renderiza dinámicamente la tabla de resultados sin recargar la página.
  - **Gestión de modales:** Controla la apertura, cierre y envío de los formularios contenidos en ventanas modales, como los de "Nuevo Producto.º Çarga Masiva". Las peticiones de creación o actualización de datos se envían también de forma asíncrona.
  - **Sugerencias de búsqueda:** Implementa la funcionalidad de autocompletar en los campos de búsqueda, realizando peticiones a la API a medida que el usuario escribe para ofrecer sugerencias relevantes.
- **Interfaz de filtrado de productos (`filter-ui.js`):** Este script se dedica exclusivamente a gestionar la compleja interfaz de la barra lateral de filtros del catálogo de productos. Inicializa y controla los selectores de categorías, el slider de precios y los checkboxes. Su lógica clave es el manejo de las **dependencias jerárquicas**: al seleccionar una familia, el script actualiza dinámicamente las opciones disponibles en el selector de categorías, creando una experiencia de filtrado guiada e intuitiva. También gestiona la creación y eliminación de "píldoras"visuales que representan los filtros activos.

- **Gestión de estado y sincronización en vistas públicas:** Los scripts de las vistas orientadas al cliente (`product-grid.js`, `product_detail.js`, `shopping_cart.js`, `wishlist.js`, `welcome.js`) comparten una estrategia común para la gestión del estado del carrito y la lista de deseos.
  - **Estado dual:** El sistema distingue entre usuarios autenticados y visitantes. Para los visitantes, el estado se persiste localmente en el `localStorage` del navegador. Para los usuarios autenticados, el estado se inicializa con los datos proporcionados por el servidor y cada cambio (añadir un producto, eliminarlo, etc.) se sincroniza de forma inmediata mediante una llamada AJAX al backend.
  - **Comunicación desacoplada por eventos:** Para notificar cambios de estado entre componentes (ej. una tarjeta de producto y la barra de navegación), se utiliza un sistema de eventos nativo. Cuando un script modifica el estado del carrito, dispara un evento global con `window.dispatchEvent(new CustomEvent('cartUpdated'))`. El script de la barra de navegación (`navbar.js`) está escuchando este evento y, al recibirlo, actualiza el icono del contador de productos. Este patrón de publicador-suscriptor permite que los componentes sean completamente independientes y reutilizables.
- **Asistente de registro de usuario (`registro_nuevo.js`):** Este script implementa la lógica de un asistente de registro multipaso. Gestiona la transición entre los diferentes pasos, la barra de progreso y, lo más importante, la **validación de datos en el lado del cliente**. Antes de permitir al usuario avanzar o enviar el formulario, el script verifica que se cumplan todos los requisitos: campos obligatorios, formato de correo electrónico válido y una política de contraseñas seguras (longitud mínima, uso de mayúsculas, números y caracteres especiales), proporcionando retroalimentación visual e instantánea al usuario.

4.4.2. Backend

La implementación del backend se ha realizado con el ecosistema de **Spring Boot**, que proporciona una base robusta para construir una aplicación mantenible, segura y escalable. A continuación, se detallan los componentes clave de su implementación, desde la estructura de paquetes hasta la lógica de negocio y seguridad.

Estructura de paquetes y componentes principales

La organización del código fuente sigue una estructura modular y lógica, donde cada paquete tiene una responsabilidad claramente definida:

- `controllers`: Este paquete se subdivide en dos categorías principales. Los `Controllers` estándar (ej. `InicioController`, `ProductController`) se encargan de gestionar las peticiones web que devuelven vistas JSP renderizadas en el servidor. Por otro lado, los `restControllers` (ej. `AdminRestController`, `ProductRestController`) exponen la API RESTful, gestionando peticiones AJAX y devolviendo datos en formato JSON. Adicionalmente, este paquete contiene una subcarpeta clave:
  - **Subpaquete `advice`:** Utiliza la anotación `@ControllerAdvice` de Spring para implementar lógica transversal que se aplica a todos los controladores, manteniendo su código limpio y enfocado. Contiene dos clases:
    - `ControllerGlobalAdvice`: Actúa como un proveedor de datos global para las vistas. Sus métodos, anotados con `@ModelAttribute`, se ejecutan en cada petición y añaden automáticamente al `Model` datos comunes que son necesarios en múltiples páginas. Esto incluye información como el nombre del usuario autenticado, el estado de su carrito y lista de deseos en formato JSON, y las estructuras de datos complejas para los menús de navegación de categorías y marcas. Centralizar esta lógica aquí evita una gran duplicación de código en los controladores.
    - `GlobalExceptionHandler`: Implementa un gestor de excepciones centralizado. Mediante la anotación `@ExceptionHandler`, captura cualquier excepción no controlada que pueda ocurrir en cualquier controlador. Su función es doble: por un lado, registra el error detallado en los logs del sistema para facilitar la depuración; por otro, redirige al usuario a una página de error personalizada y amigable en lugar de mostrar el error por defecto del servidor, mejorando así la experiencia de usuario ante fallos inesperados.
- `services`: Siguiendo el principio de inversión de dependencias, este paquete contiene las interfaces de los servicios (en `services.interfaces`) y sus implementaciones concretas (en `services.impl`). Existen servicios para cada entidad principal del dominio, como `ProductService` o `UsuarioService`.
- `entities`: Contiene las clases de dominio que, mediante anotaciones de JPA, se mapean a las tablas de la base de datos.
- `repositories`: Define las interfaces de Spring Data JPA (ej. `ProductoDAO`) que abstraen por completo el acceso a la capa de persistencia.
- `dto` y `mappers`: El paquete `dto` contiene los Objetos de Transferencia de Datos que actúan como contrato con el frontend. Los `mappers`, como el `ExcelToProductMapper`, encapsulan la lógica para convertir datos de un formato a otro, en este caso, para la funcionalidad de carga masiva de productos.

- `security`: Agrupa todas las clases relacionadas con la seguridad de la aplicación, incluyendo la configuración de Spring Security, la gestión de tokens JWT y las clases DTO para la autenticación y el registro.

- `config`: Centraliza la configuración de beans de Spring y la gestión de comportamientos específicos del entorno, promoviendo un código más limpio y desacoplado.

    - `DotenvConfig`: Esta clase de configuración tiene como única responsabilidad facilitar el desarrollo en un entorno local. Utilizando la librería `java-dotenv`, carga automáticamente las variables de entorno desde un fichero `.env` situado en la raíz del proyecto. Gracias a la anotación `@Profile("dev")`, esta configuración solo se activa cuando el perfil de Spring es "dev".

      La estrategia de seguridad asociada a este fichero es crucial: el fichero `.env` contiene todas las credenciales sensibles (conexión a la base de datos, clave secreta de JWT, etc.) y está explícitamente excluido del control de versiones mediante una entrada en el fichero `.gitignore`. De esta forma, se previene cualquier exposición accidental de secretos en el repositorio de código. Para facilitar la configuración a otros desarrolladores, en el repositorio se incluye únicamente una plantilla, `.env.example`, que documenta las variables necesarias sin sus valores. En el entorno de producción, estas variables son inyectadas de forma segura directamente en el contenedor Docker, y esta clase de configuración no se activa.

    - `ModelMapperConfig`: Esta clase configura un bean de `ModelMapper` para la conversión automática de objetos, principalmente entre entidades JPA y DTOs. Se han definido **mapeos personalizados** para gestionar correctamente las relaciones anidadas (por ejemplo, obtener el nombre de una categoría de un producto). Centralizar esta lógica en un único punto asegura la consistencia en la transformación de datos en toda la aplicación.

- `utils`: Este paquete contiene clases de utilidad reutilizables que encapsulan lógica auxiliar y transversal, ayudando a mantener el resto del código —especialmente los controladores— más limpio y enfocado en su responsabilidad principal.

    - `BreadcrumbUtils.java`: Es una clase de utilidad estática responsable de la **generación dinámica de las rutas de navegación** (*breadcrumbs*) que se muestran en la parte superior de las páginas de productos. Su lógica principal analiza los filtros activos en la URL de la petición para construir una ruta jerárquica y coherente. Por ejemplo, solo construye la ruta completa si la selección de categorías es única y no ambigua. Además, proporciona un método específico para generar

la ruta de navegación en la página de detalle de un producto, mostrando su jerarquía completa. Centralizar esta lógica compleja aquí permite que los controladores simplemente invoquen un método para obtener las migas de pan, sin tener que preocuparse por los detalles de su construcción.

- `CreditCardUtils.java`: Clase de utilidades estáticas encargada de la **validación integral de los datos de pago** recibidos en un objeto `PaymentDTO`. Su método público `validarDatosDePago` orquesta la comprobación de nulidad y delega en tres validaciones específicas:
  - *Número de tarjeta*: elimina espacios y verifica que sean exactamente 16 dígitos numéricos.
  - *Fecha de caducidad*: comprueba que siga el formato MM/AAAA, que el mes esté entre 01 y 12, que la tarjeta no esté caducada y que el año no supere en más de 15 años la fecha actual.
  - *CVC*: asegura que el valor entero exista y esté en el rango de 000 a 999 (tres dígitos).

Modelo de dominio y entidades JPA

La persistencia de los datos se gestiona a través de las siguientes entidades JPA clave:

- `Usuario`: Es la entidad central para la gestión de identidades. Almacena datos personales, credenciales de acceso (con el campo `passwd` hasheado) y consentimientos. Utiliza la anotación `@ElementCollection` de JPA para gestionar eficientemente las colecciones de datos simples como los roles y las preferencias del usuario, que se almacenan en tablas separadas (`usuario_roles` y `usuario_preferencias`) pero se gestionan como parte del ciclo de vida de la entidad `Usuario`.

- `Producto`: Representa un artículo del catálogo. Además de sus atributos básicos, implementa las relaciones `@ManyToOne` con todo el sistema de categorización (`Familia`, `Categoria`, `Laboratorio`, etc.), creando un modelo de datos normalizado y relacional.

- `ShoppingCart`: Modela el carrito de la compra. Se define una relación `@OneToOne` con la entidad `Usuario`, asegurando que cada usuario tenga un único carrito. La relación con los productos del carrito se gestiona a través de la entidad `CartItem` mediante una relación `@OneToMany` con las opciones `cascade = CascadeType.ALL` y `orphanRemoval = true`. Esto significa que los ítems del carrito se gestionan automáticamente junto con el ciclo de vida del carrito: si se elimina el carrito, se eliminan todos sus ítems.

- `Wishlist`: Implementa la lista de deseos. A diferencia del carrito, se modela con una relación `@ManyToMany` directa con la entidad `Producto`, utilizando la anotación `@JoinTable` para configurar la tabla intermedia `wishlist_producto`.

- `Venta`: Modela un pedido o venta completada por un cliente. Se vincula de forma obligatoria a un `Usuario` mediante una relación `@ManyToOne`. Almacena datos clave de la transacción como la `fechaVenta` y el `montoTotal`, utilizando un tipo `Float` para la represenatación de céntimosa. La relación con los productos vendidos se gestiona a través de la entidad `DetalleVenta` mediante una relación `@OneToMany`. La configuración `cascade = CascadeType.ALL` y `orphanRemoval = true` asegura que el ciclo de vida de los detalles del pedido esté completamente ligado al del pedido principal: al eliminar una venta, todos sus detalles se eliminan automáticamente.

- `Devolucion`: Representa una solicitud de devolución iniciada por un cliente. Se asocia de forma obligatoria tanto al `Usuario` que la solicita como a la `Venta` original, garantizando una trazabilidad completa del proceso. Una decisión de diseño importante es el uso de una enumeración de Java (`DevolucionEnum`) para el campo `motivo`, que se persiste como una cadena de texto gracias a la anotación `@Enumerated(EnumType.STRING)`. Este enfoque garantiza la consistencia de los datos y hace el código más legible y mantenible. De forma análoga a la venta, la relación con los productos devueltos se gestiona con la entidad `DetalleDevolucion` a través de una relación `@OneToMany` con borrado en cascada.

- `Evaluation`: Representa la puntuación que un `Usuario` otorga a un `Producto`, persistida en la tabla `evaluacion`. Incluye un campo `score` de tipo `Integer` (1–5) y dos relaciones `@ManyToOne` obligatorias con `Producto` y `Usuario` mediante `@JoinColumn(nullable=false)`, garantizando integridad referencial y trazabilidad de las valoraciones. El constructor explícito facilita la creación de instancias sin exponer el `id`, manteniendo limpia la lógica de negocio y centralizando el modelo de datos.

Implementación de servicios clave

La lógica de negocio se encapsula en la capa de servicio. Aunque existen servicios para cada entidad principal, algunos son particularmente notables por su criticidad:

- `AuthenticationService`: [4.1] Orquesta el proceso de registro y login de usuarios. En el registro, se encarga de transformar el DTO `RegisterRequest` en una entidad `Usuario`, hashear la contraseña utilizando el `PasswordEncoder` y persistir el nuevo usuario. Tras guardar, invoca al `JwtService` para generar un token y devolverlo al controlador.

Código 4.1: Construcción de JWT token.

```
public AuthenticationResponse authenticate(AuthenticationRequest request) {
    authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(
            request.getCorreo(),
            request.getPassword()
        )
    );

    var usuario =
        usuarioRDAO.getByCorreo(request.getCorreo()).orElseThrow();
    var usuarioPrincipal = new UsuarioPrincipal(usuario);
    var jwtToken = jwtService.generateToken(usuarioPrincipal);

    return AuthenticationResponse.builder().token(jwtToken).build();
}
```

- `ProductService`: Este es el servicio central para todas las operaciones relacionadas con el catálogo de productos. Su responsabilidad va mucho más allá de las operaciones CRUD básicas. Sus funciones clave son:

  - **Búsqueda y filtrado dinámico:** Es responsable de construir consultas dinámicas y paginadas a la base de datos utilizando la **API de Especificaciones de JPA (JPA Specification API)**. Este enfoque permite combinar de forma flexible múltiples criterios de filtro (nombre, categorías, rango de precios, stock, etc.) recibidos desde la interfaz. Además, aplica reglas de negocio de seguridad, como forzar la condición de "activo" en las búsquedas públicas.
  - **Carga masiva de datos desde ficheros Excel:** [4.2] Se ha implementado una funcionalidad crítica para la administración que permite la carga masiva de productos desde un fichero Excel. El proceso es orquestado por el `ProductService` y utiliza la librería **Apache POI** para leer y procesar el contenido del archivo.

    Para cada fila del fichero, el servicio primero realiza un chequeo de duplicados para optimizar el rendimiento, extrayendo únicamente el ID del producto para verificar si ya existe en la base de datos. Si el producto es nuevo, se delega la compleja tarea de transformar la fila de Excel en un objeto `Producto` a un componente especializado: el `ExcelToProductMapper`.

    Este `mapper` es responsable de convertir los datos de cada celda al tipo de dato correspondiente en la entidad. Una parte crucial de su lógica es la gestión de las relaciones. Para campos como "familia" o "laboratorio", el `mapper` no solo lee el texto, sino que utiliza los servicios correspondientes (ej. `FamiliaService`) para buscar la entidad por su nombre en la base de datos. Este enfoque garantiza la **integridad referencial**: solo se establecen relaciones con entidades que ya existen en el sistema, evitando datos corruptos. Finalmente, el `mapper` [4.3] devuelve el objeto `Producto` completamente construido al servicio, que lo persiste en la base de datos.

Código 4.2: Carga masiva de productos.

```
public List<Producto> bulkUpload(MultipartFile file) throws Exception
    {
    List<Producto> duplicados = new ArrayList<>();
    try (Workbook wb = new XSSFWorkbook(file.getInputStream())) {
        Sheet sheet = wb.getSheetAt(0);

        for (Row row : sheet) {
            if (row.getRowNum() == 0) continue;  // salto cabecera
            Producto nuevo = mapper.map(row);
            productDAO.save(nuevo);
        }
    }
    return duplicados;
}
```

Código 4.3: Mapeo de atributos.

```
public Producto map(Row row) {
    Producto prod = new Producto();

    String codigoTxt = fmt.formatCellValue(row.getCell(0)).trim();
    prod.setId(new BigDecimal(codigoTxt));

    String famTxt = fmt.formatCellValue(row.getCell(2)).trim();
    Optional.of(famTxt)
            .filter(s -> !s.isEmpty())
            .flatMap(familiaService::findByNombre)
            .ifPresent(prod::setFamilia);

    /* otros campos*/
    String precioTxt = fmt.formatCellValue(row.getCell(9))
                          .replace("E", "")
                          .replace(".", "")
                          .replace(",", ".")
                          .trim();
    prod.setPrice(Float.valueOf(precioTxt));

    return prod;
}
```

  - **Soporte a la interfaz de usuario:** Proporciona métodos específicos que soportan funcionalidades del frontend, como la búsqueda predictiva para la barra de navegación (devolviendo una lista limitada de nombres) y el cálculo de métricas agregadas (KPIs) para el dashboard del administrador (conteo total de productos, suma de stock, etc.).
  - **Transformación de datos (DTO):** Se encarga de mapear las entidades `Producto` obtenidas de la base de datos a objetos `ProductoDTO`, asegurando que solo la información necesaria se exponga a través de la API.

- `ShoppingCartService`: Este servicio encapsula toda la lógica de negocio relacionada con el carrito de la compra. Su diseño se fundamenta en la seguridad, la eficiencia y la integridad de los datos. Sus responsabilidades principales son:

  - **Creación "Lazy"del carrito:** El servicio no crea un carrito para cada usuario al registrarse, sino que lo hace bajo demanda. El método

`getOrCreateShoppingCartFromPrincipal` busca primero un carrito existente para el usuario y, solo si no lo encuentra, crea y persiste uno nuevo. Esta estrategia es más eficiente en el uso de recursos de la base de datos.

- **Lógica de negocio transaccional:** El método `addItem` contiene una lógica transaccional (`@Transactional`) robusta. Antes de añadir un producto, verifica que este exista y que esté activo. A continuación, comprueba si el producto ya está en el carrito para decidir si debe crear un nuevo ítem o simplemente actualizar la cantidad de uno existente, gestionando también la decrementación o eliminación de productos.
- **Sincronización con el frontend:** [4.4] Proporciona un método, `getCartStateForUser`, que transforma el estado del carrito del backend en una estructura de datos simple (un `Map<String, Integer>`). Este mapa es el que se envía en formato JSON al cliente para que el JavaScript pueda sincronizar la interfaz de usuario con el estado real del carrito en el servidor.

Código 4.4: Obtención de carrito de usuario autenticado.

```
1         public Map<String, Integer> getCartStateForUser(Principal
              principal) {
2             if (principal == null) {
3                 return Collections.emptyMap();
4             }
5
6             ShoppingCart cart =
                  this.getOrCreateShoppingCartFromPrincipal(principal);
7
8             if (cart == null || cart.getItems() == null ||
                  cart.getItems().isEmpty()) {
9                 return Collections.emptyMap();
10            }
11
12            Map<String, Integer> cartStateMap = new HashMap<>();
13
14            for (CartItem item : cart.getItems()) {
15                String productId =
                      item.getProducto().getId().toString();
16                Integer quantity = item.getCantidad();
17                cartStateMap.put(productId, quantity);
18            }
19
20            return cartStateMap;
21        }
```

- `VentaService`**:** Este servicio orquesta el proceso más crítico del negocio: la conversión de un carrito de la compra en un pedido finalizado. Su principal responsabilidad es garantizar la integridad y atomicidad de cada venta mediante operaciones transaccionales. Sus funciones clave son:
  - **Procesamiento transaccional de pedidos:** El método principal de este servicio gestiona la creación de una nueva venta. Esta operación, marcada como `@Transactional`, ejecuta una secuencia

de pasos que deben completarse con éxito en su totalidad: primero, valida la disponibilidad de stock de todos los productos del carrito; a continuación, crea las entidades `Venta` y `DetalleVenta`, guardando una instantánea del precio en el momento de la compra; después, decrementa el stock de los productos vendidos; y finalmente, vacía el carrito de la compra del usuario.

- **Gestión del historial de pedidos:** Proporciona los métodos necesarios para que los usuarios puedan consultar su historial de pedidos de forma segura (utilizando siempre el `Principal`) y para que los administradores puedan buscar y filtrar el conjunto completo de ventas de la plataforma.
- **Orquestación de servicios:** Actúa como un coordinador que interactúa con otros servicios. Colabora con el `ProductService` para verificar el stock, con el `ShoppingCartService` para obtener los artículos del carrito y está diseñado para integrarse con futuros servicios, como pasarelas de pago o sistemas de notificación por correo electrónico.

Implementación detallada del sistema de seguridad

La estrategia de seguridad se materializa en un flujo de trabajo bien definido que protege la aplicación en cada petición, basado en los siguientes componentes y principios:

- **Flujo de autenticación por petición con JWT:** El corazón del sistema reside en el filtro `JwtAuthentication`, una clase personalizada que se ejecuta en cada petición entrante. Su primera tarea es extraer el token JWT desde una `Cookie HttpOnly` y `Secure`, una medida para mitigar ataques XSS. A continuación, delega la validación al `JwtService`, que verifica la firma criptográfica y la fecha de expiración del token. Si el token es válido, se utiliza para cargar los detalles del usuario y establecer su identidad en el contexto de seguridad de Spring (`SecurityContextHolder`), autenticando así la petición.
- **Configuración de seguridad y permisos:** [4.5] La clase `SecurityConfig` define las reglas globales de seguridad. Establece una política de sesiones completamente sin estado (`STATELESS`) y define un sistema de autorización granular por rutas. Este enfoque permite centralizar todas las reglas de acceso en un único lugar, como se ilustra en el siguiente fragmento:

Código 4.5: Definición de permisos de acceso por ruta en `SecurityConfig`.

```
1  .authorizeHttpRequests(requests -> requests
2      .requestMatchers("/css/**", "/login", "/register",
           "/auth/**").permitAll()
```

```
3
4       .requestMatchers("/profile/**").hasAnyRole("USUARIO", "ADMIN")
5       .requestMatchers("/admin/**").hasRole("ADMIN")
6
7       .anyRequest().authenticated()
8   )
```

- **Gestión de credenciales y autenticación:** La clase `ApplicationConfig` configura los componentes para el proceso de *login* inicial. Define el `PasswordEncoder` (utilizando `BCryptPasswordEncoder`), proporciona el `UserDetailsService` para cargar usuarios desde la base de datos, y configura el `AuthenticationProvider` que une los dos componentes anteriores para orquestar la validación de las credenciales.

- **Principio de confianza cero en la lógica de negocio** Para garantizar la integridad y privacidad de los datos, se ha adoptado un principio de "confianza cero" hacia la información del cliente. Cualquier petición que implique la consulta o modificación de datos de un usuario (carrito, perfil, etc.) ignora sistemáticamente cualquier identificador que pueda ser enviado en la petición. En su lugar, la lógica de negocio se basa exclusivamente en el objeto `Principal` que Spring Security inyecta en el contexto tras una autenticación exitosa [4.6]. El siguiente código ilustra cómo el controlador del carrito obtiene los datos basándose en el `Principal` del usuario autenticado, haciendo imposible que un usuario acceda a los datos de otro.

Código 4.6: Service entrega datos del carrito del `Principal` al Model.

```
1   @Controller
2   @RequestMapping("/cart")
3   @RequiredArgsConstructor
4   public class ShoppingCartController {
5
6       private final ShoppingCartService shoppingCartService;
7
8       @GetMapping({"/", ""})
9       public String goShoppingCart(Principal principal, Model model) {
10          model.addAttribute("shoppingCart",
11              shoppingCartService.getOrCreateShoppingCartFromPrincipal(principal));
12          return "purchases/shopping_cart";
13      }
14  }
```

4.5. Pruebas

Con el objetivo de garantizar la fiabilidad y calidad del sistema, se han implementado distintos tipos de pruebas que cubren diferentes niveles de la aplicación: pruebas unitarias para la lógica de negocio, pruebas de integración para verificar la correcta comunicación entre capas, pruebas de API con Postman para validar la interfaz REST, y pruebas de sistema con Selenium que comprueban la interacción completa desde la perspectiva del usuario.

4.5.1. Pruebas Unitarias

Las pruebas unitarias se han desarrollado con el framework **JUnit 5**, utilizando **Mockito** para la simulación de dependencias. Estas pruebas se centran en la lógica de negocio contenida en los servicios y clases auxiliares, aislándolas de la infraestructura externa.

Para lograr esta separación, se emplean las anotaciones `@Mock` y `@InjectMocks`. La primera permite crear objetos simulados de componentes externos (como repositorios), mientras que la segunda inyecta dichos mocks en la clase que se desea probar, facilitando así un entorno de test controlado.

Cada test unitario sigue una estructura común basada en buenas prácticas:

- **Preparación**: se crean datos de prueba (entidades simuladas) y se configuran los comportamientos esperados de las dependencias usando instrucciones como `when(...).then(...)`.
- **Ejecución**: se invoca el método del servicio que se quiere probar.
- **Verificación**: se comprueba el resultado usando aserciones (`assertThat()`, `assertEquals()`, etc.) y se verifica que se han llamado los métodos esperados en los mocks.

Los tests cubren las siguientes áreas:

- **Servicios**: como `AuthenticationService`, donde se valida que los datos del registro se transforman correctamente y se generan los tokens adecuados.
- **Modelos**: como `UsuarioPrincipal`, verificando los métodos relacionados con seguridad y autenticación.
- **Seguridad**: pruebas sobre la clase `JwtService`, que garantizan la integridad de los tokens y su correcta validación.

4.5.2. Pruebas de Integración

Las pruebas de integración tienen como objetivo verificar que los distintos componentes de la aplicación funcionan correctamente de forma conjunta. A diferencia de las pruebas unitarias —centradas en una única clase aislada—, estas pruebas validan el comportamiento de la aplicación en su contexto real de ejecución, asegurando la correcta interacción entre capas.

En este proyecto se ha utilizado la anotación `@SpringBootTest` para lanzar el contexto completo de Spring Boot, junto con `@AutoConfigureMockMvc`, lo que

permite simular peticiones HTTP mediante la clase `MockMvc`. Además, se ha definido un perfil de pruebas mediante `@ActiveProfiles("test")` para aislar el entorno de test del de producción. Algunas dependencias se han simulado con `@MockBean` para mantener el enfoque sobre los componentes bajo prueba.

Se han implementado pruebas de integración principalmente con dos fines:

- **Controllers:** Se han realizado pruebas de integración de la capa web para validar el comportamiento de los controladores. El objetivo de estas pruebas es doble: por un lado, verificar que las rutas URL están correctamente mapeadas a los métodos de los controladores y, por otro, asegurar que estos métodos preparan adecuadamente los datos que se enviarán a las vistas.

  La metodología de prueba se ha basado en el uso de **MockMvc** para simular peticiones HTTP y realizar aserciones sobre la respuesta. Es importante destacar que, para estas pruebas, se han **desactivado los filtros de seguridad** de Spring (`addFilters = false`) con el fin de aislar el comportamiento del controlador y no depender del estado de autenticación. Asimismo, las dependencias de la capa de servicio (como `UsuarioService` o `DestacadoService`) han sido reemplazadas por simulacros (*mocks*) mediante la anotación `@MockBean`.

  Los tests se pueden agrupar en dos categorías principales según su alcance:

  - **Validación de Navegación y Vistas:** [4.7] La mayoría de las pruebas, como las de las rutas `/`, `/login` o `/product`, se centran en confirmar que una petición `GET` a una URL específica devuelve un código de estado '200 OK' y resuelve el nombre de la vista JSP correcta. Estas pruebas garantizan que el enrutamiento básico de la aplicación funciona como se espera.

    Código 4.7: Test de navegación de vistas

```
1  @Test
2  void shouldAllowAccessToPublicRoutes() throws Exception {
3      mockMvc.perform(get("/")).andExpect(status().isOk());
4      mockMvc.perform(get("/login")).andExpect(status().isOk());
5      mockMvc.perform(get("/product")).andExpect(status().isOk());
6  }
```

  - **Validación del modelo de datos:**[4.8] Pruebas más complejas, como la de la página de inicio o la de registro, van un paso más allá. Además de verificar el estado y la vista, comprueban que el controlador pobla correctamente el objeto `Model`. Por ejemplo, en el test de la página de inicio, se simula la respuesta del `destacadoService` y se verifica que el controlador añade correctamente el atributo "destacados" al modelo antes de enviarlo a la vista. Esto confirma que la comunicación entre el controlador y la capa de servicio es correcta y que la vista recibirá los datos que necesita para renderizarse.

Código 4.8: Test página de inicio

```
@Test
void getWelcomePage() throws Exception {
    // Creamos algunos Productos de ejemplo
    Producto p1 = new Producto();
    p1.setId(BigDecimal.valueOf(1));
    p1.setNombre("Producto A");
    Producto p2 = new Producto();
    p2.setId(BigDecimal.valueOf(2));
    p2.setNombre("Producto B");
    List<Producto> fakeDest = List.of(p1, p2);

    when(destacadoService.getAllDestacados()).thenReturn(fakeDest);

    mockMvc.perform(get("/"))
        .andExpect(status().isOk())
        .andExpect(view().name("main/welcome"))
        .andExpect(model().attribute("destacados", fakeDest));
}
```

- **Security** [4.9]: se ha verificado que las rutas públicas son accesibles sin autenticación y que las rutas protegidas redirigen correctamente a los usuarios no autenticados. Esto asegura que la configuración de seguridad funciona según lo esperado.

Código 4.9: Test redireccion no autenticado

```
@Test
void shouldBlockAccessToProtectedRoutesWithoutAuth() throws Exception {
    mockMvc.perform(get("/profile")).andExpect(status().is3xxRedirection());
    mockMvc.perform(get("/admin/home")).andExpect(status().is3xxRedirection());
}
```

En conjunto, las pruebas de integración complementan a las pruebas unitarias verificando que los distintos componentes colaboran correctamente en el contexto de ejecución real.

4.5.3. Pruebas de API

Las pruebas de API permiten validar el comportamiento de los endpoints REST de forma directa, sin depender de la interfaz de usuario. En este proyecto, se llevaron a cabo con la herramienta **Postman** [4.31], que facilita la automatización y organización de pruebas HTTP. Aunque existen soluciones más avanzadas como RESTAssured, se optó por pruebas manuales en Postman debido al alcance y plazos del proyecto.

Estas pruebas han sido fundamentales para comprobar que la capa de aplicación expone correctamente su funcionalidad y responde adecuadamente a distintos escenarios, incluyendo peticiones válidas, errores y accesos restringidos.

Aspectos cubiertos:

- Operaciones CRUD sobre recursos clave del sistema, como usuarios, productos o pedidos.
- Validación del formato y contenido de las respuestas JSON, así como de los códigos de estado HTTP.
- Comprobación del sistema de autenticación con JWT, incluyendo generación y validación de tokens.
- Verificación del control de acceso según los distintos roles definidos en la aplicación (usuario, empleado, administrador).

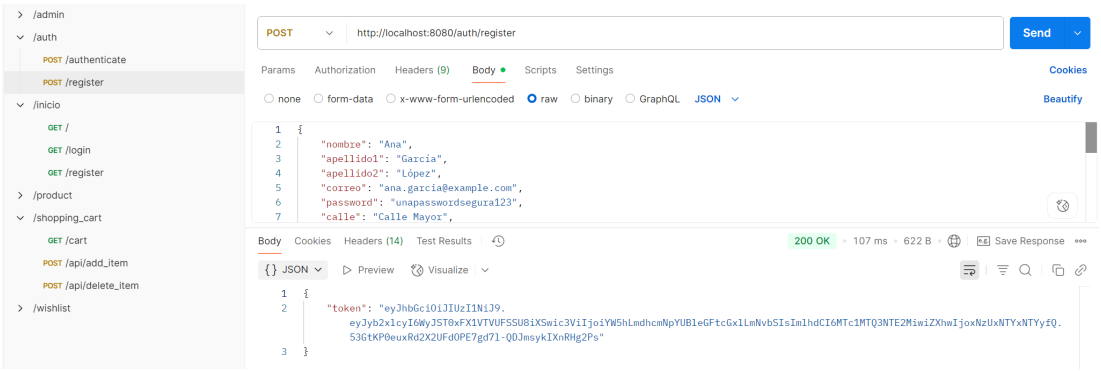


Figura 4.31: Captura de pantalla de Postman con la organización de carpetas por cada bloque de pruebas.

4.5.4. Pruebas de Sistema (End-to-End)

Las pruebas de sistema, o End-to-End (E2E), son la fase final de validación y permiten verificar el comportamiento completo de la aplicación desde la perspectiva del usuario final. Para este proyecto, se han implementado utilizando **Selenium WebDriver**, una herramienta que automatiza un navegador web real para simular las interacciones del usuario y comprobar que todos los componentes del sistema —frontend, backend y base de datos— funcionan de forma coordinada.

Escenarios de Prueba Cubiertos

Se ha desarrollado una suite de pruebas E2E que cubre los flujos de usuario más críticos para el negocio, asegurando que las funcionalidades principales operan correctamente de principio a fin. Los flujos cubiertos son:

- **Proceso de Compra Completo:** Se simula el ciclo de compra íntegro, desde la adición de productos al carrito, pasando por el proceso de "checkout", hasta la confirmación final del pedido.

- **Registro de un Nuevo Usuario:** Se ha validado el flujo completo de registro a través de sus múltiples pasos. Por su complejidad y criticidad para el negocio, este test se utiliza como el principal caso de estudio para detallar la metodología de prueba.

Análisis Detallado del Test de registro

El test "testRegistroExitoso" [4.10] es el ejemplo más representativo de la metodología E2E seguida. Automatiza el proceso de registro de principio a fin, simulando con alta fidelidad las acciones de un usuario real a través de los tres pasos del formulario.

El flujo de la prueba comienza en el **Paso 1**, donde simula la introducción de los datos básicos de la cuenta, como el nombre, correo y contraseña, utilizando el método "sendKeys". También se automatiza la interacción con otros elementos, como el clic en las casillas de verificación obligatorias ("aceptaTerminosz "aceptaPrivacidad"). Una vez completados los campos, se pulsa el botón de "Siguiente". En este punto, para garantizar la robustez del test, se utiliza una **espera explícita ("wait.until")** que detiene la ejecución hasta que la interfaz confirma, mediante un texto en pantalla, que se ha cargado el "Paso 2 de 3". Esta técnica es fundamental para evitar errores por tiempos de carga variables.

En el **Paso 2**, el proceso se repite para los datos personales y de envío, interactuando con distintos tipos de campos como fechas y menús desplegables, para los que se utiliza la clase "Select"de Selenium. De nuevo, se finaliza con un clic y una espera explícita para la confirmación del "Paso 3 de 3".

En el **Paso 3**, el test demuestra la capacidad de interactuar con elementos más complejos, como la selección de intereses, donde puede ser necesario hacer scroll. Para ello, se utilizan herramientas avanzadas de Selenium como "JavascriptExecutorz la clase "Actions"para asegurar que el elemento es visible y clicable antes de la interacción.

Código 4.10: Test registro

```
1  @Test
2  public void testRegistroExitoso() {
3      /*PASO 1 */
4      driver.findElement(By.id("nombre")).sendKeys("Juan");
5      driver.findElement(By.id("apellido1")).sendKeys("Perez");
6      driver.findElement(By.id("apellido2")).sendKeys("Gomez");
7      driver.findElement(By.id("aceptaTerminos")).click();
8      /* otros campos */
9
10     driver.findElement(By.id("next-btn")).click();
11     wait.until(ExpectedConditions.textToBe(By.id("step-label"), "Paso 2 de 3"));
12
13     /*PASO 12*/
14     driver.findElement(By.id("fechaNac")).sendKeys("1990-05-15");
15     new Select(driver.findElement(By.id("genero"))).selectByValue("1");
16     driver.findElement(By.id("telefono")).sendKeys("600123456");
```

```
17      /* otros campos */
18
19      driver.findElement(By.id("next-btn")).click();
20      wait.until(ExpectedConditions.textToBe(By.id("step-label"), "Paso 3 de 3"));
21
22      /* PASO 3 */
23      List<WebElement> intereses =
            driver.findElements(By.cssSelector(".select-interes"));
24      for (int idx : new int[]{0, 2}) {
25          WebElement interes = intereses.get(idx);
26
27          wait.until(ExpectedConditions.elementToBeClickable(interes));
28
29          ((JavascriptExecutor) driver)
30              .executeScript("arguments[0].scrollIntoView({block:'center'});",
                    interes);
31
32          new Actions(driver).moveToElement(interes).click().perform();
33      }
34
35      /* comprobar redireccion */
36      driver.findElement(By.id("next-btn")).click();
37      wait.until(ExpectedConditions.urlToBe("http://localhost:" + port +"/"));
38      assertEquals("http://localhost:" + port + "/", driver.getCurrentUrl());
39  }
```

Metodología y aspectos clave

Para ilustrar la profundidad de estas pruebas, se toma como referencia el flujo de registro. La metodología se centra en validar varios aspectos clave que garantizan la robustez y el correcto funcionamiento de la aplicación:

- **Simulación de flujos de múltiples pasos:** El test automatizado replica la secuencia completa que un usuario seguiría, como la navegación a través de los tres pasos del formulario de registro, asegurando que la transición entre ellos funciona correctamente.
- **Interacción y validación de datos:** Se automatiza la interacción con diversos componentes de la interfaz (campos de texto, casillas de verificación, etc.). Además, se realizan validaciones visuales, comprobando que la aplicación reacciona adecuadamente [??] ante entradas inválidas, mostrando los mensajes de error esperados.
- **Sincronización y robustez:** Para evitar tests inestables (*flaky tests*) debido a variaciones en los tiempos de carga de la página y elementos asíncronos, se utilizan esperas explícitas mediante `WebDriverWait`. El script no avanza ciegamente, sino que espera activamente a que los elementos de la interfaz estén presentes y sean interactuables antes de continuar.
- **Verificación del estado final:** La prueba concluye validando el resultado final del flujo. Esto incluye la comprobación de redirecciones, verificando que el usuario es dirigido a la página esperada (ej. la página de inicio) tras

completar un proceso con éxito. Esta aserción final confirma implícitamente que todo el proceso en el backend se ha completado correctamente.

Toda la suite de pruebas se ha configurado para poder ejecutarse en modo `headless`, es decir, sin lanzar una interfaz gráfica de navegador, lo que optimiza su ejecución en entornos de integración continua.

4.6. Distribución y Despliegue

El ciclo de vida del software, desde la escritura del código hasta su puesta a disposición del usuario final, se ha gestionado mediante un ecosistema de herramientas modernas que garantizan un proceso robusto, reproducible y altamente automatizado. Este capítulo detalla la estrategia de control de versiones, el pipeline de Integración y Despliegue Continuo (CI/CD) implementado con GitHub Actions, y la arquitectura de la infraestructura de producción en la nube de AWS.

4.6.1. Control de versiones y activación del pipeline

El punto de partida de todo el proceso es el control de versiones con Git. Se ha seguido la convención **Conventional Commits** para estructurar los mensajes de commit, utilizando prefijos como `feat:`, `fix:` o `refactor:`. Esta estandarización es fundamental para la automatización, ya que permite que el pipeline de CI/CD interprete la naturaleza de los cambios.

El workflow de despliegue se activa automáticamente solo cuando se cumplen dos condiciones: la suite de pruebas de un workflow previo (`RunTests`) ha finalizado con éxito y los cambios se han integrado en la rama `main`. Una vez activado, el primer paso del pipeline es determinar la nueva versión de la aplicación. Para ello, un script localiza la última etiqueta Git existente (p. ej., `v1.2.5`) y, analizando el mensaje del último commit, incrementa la versión de forma semántica: la palabra *release* incrementa la versión `minor` (ej. `v1.3.0`), mientras que la palabra *fix* incrementa la de *patch* (ej. `v1.2.6`).

4.6.2. Construcción y empaquetado del artefacto (CI)

Una vez determinada la nueva versión, la fase de Integración Continua se encarga de construir el artefacto desplegable. Este proceso consta de varios pasos automatizados:

1. **Construcción del ejecutable:** El workflow compila la aplicación Java con

Maven, generando el archivo `.war`. Se utiliza un sistema de caché para las dependencias de Maven para optimizar los tiempos de ejecución.

2. **Containerización con Docker:** A continuación, se construye una imagen **Docker** a partir de una base optimizada (`eclipse-temurin:17-jdk-alpine`). Esta imagen encapsula la aplicación y todas sus dependencias, garantizando que se ejecute de forma coherente en cualquier entorno.

3. **Publicación del artefacto:** La imagen Docker se etiqueta con la nueva versión semántica y como `latest`. Ambas etiquetas se publican en un registro de contenedores (**Docker Hub**), dejando un artefacto inmutable y versionado, listo para ser desplegado.

4. **Etiquetado en Git:** Como paso final de esta fase, se crea y publica una nueva etiqueta (*tag*) en el repositorio de Git con la versión generada, marcando de forma permanente el estado del código para esa *release*.

4.6.3. Despliegue en la infraestructura de AWS (CD)

La fase de Despliegue Continuo se encarga de poner en producción el artefacto generado. La infraestructura de destino se compone de varios servicios de **Amazon Web Services (AWS)**:

- **Entorno de ejecución (Amazon EC2):** Se utiliza una instancia EC2 como servidor para ejecutar el contenedor Docker. El workflow de GitHub Actions se conecta de forma segura a esta instancia mediante una clave SSH almacenada como secreto. Una vez conectado, un script detiene y elimina el contenedor anterior, descarga la nueva imagen desde Docker Hub y lanza un nuevo contenedor, inyectando todas las variables de entorno necesarias (credenciales, claves, etc.) de forma segura.

- **Base de Datos gestionada (Amazon RDS):** Para la capa de persistencia, se ha optado por desacoplar la base de datos utilizando Amazon RDS con un motor MySQL. Este servicio gestionado garantiza alta disponibilidad, backups automáticos y seguridad, ya que la instancia se ejecuta en una subred privada accesible únicamente desde la instancia EC2 de la aplicación.

4.6.4. Configuración de red y acceso público

La conexión final entre internet y la aplicación desplegada en AWS se gestiona mediante dos componentes clave:

- **Gestión del dominio con Plesk:** El dominio principal, `laboticademar.com`, se administra desde un panel **Plesk**, el servicio preexistente de la farmacia. Su rol en esta arquitectura es el de gestor de DNS. Se han configurado los registros de tipo **'A'** del dominio para que apunten a la dirección IP pública de la instancia EC2, redirigiendo así todo el tráfico web hacia el servidor correcto.
- **Proxy inverso y HTTPS con Nginx:** En la instancia EC2, se ha configurado **Nginx** como proxy inverso. Este recibe todo el tráfico del dominio y lo redirige internamente al contenedor Docker, que se ejecuta de forma aislada en `localhost:8080`. Nginx también se encarga de la seguridad, gestionando el certificado SSL obtenido gratuitamente con **Certbot (Let's Encrypt)** y forzando todo el tráfico a usar **HTTPS**. Certbot, además, ha configurado una tarea `cron` para la renovación automática de los certificados.

5

Conclusiones y trabajos futuros

Llegados a la fase final de este documento, el presente capítulo cumple una doble función esencial. En primer lugar, se realiza un ejercicio de introspección para destilar las conclusiones más significativas extraídas del proceso de desarrollo, abarcando tanto los logros técnicos como el crecimiento personal y metodológico. En segundo lugar, se mira hacia el horizonte, trazando un mapa de futuras mejoras que no solo demuestran el potencial de crecimiento de la plataforma, sino que también sientan las bases para su evolución hacia un producto de mercado completo y competitivo.

5.1. Conclusiones

Este primer apartado tiene como finalidad recapitular la experiencia y los resultados obtenidos a lo largo del desarrollo del proyecto. Ofrece una reflexión sobre el valioso aprendizaje adquirido, tanto en el plano técnico y metodológico como en el personal, consolidando el conocimiento práctico obtenido tras la finalización de este Trabajo de Fin de Grado.

La finalización de este proyecto representa no solo el cumplimiento de los objetivos académicos propuestos, sino también la culminación de un intenso proceso de aprendizaje que ha aportado un valor incalculable. Desde la perspectiva técnica, el proyecto ha sido una inmersión profunda en el ciclo de vida completo del desarrollo de software, trascendiendo la simple codificación. El mayor aprendizaje se ha centrado en el proceso de despliegue y la gestión de infraestructuras en la

nube. Se ha obtenido una competencia práctica en la administración de servidores a través de la línea de comandos (`CLI`), realizando tareas esenciales como la configuración de un firewall (`ufw`), la monitorización de recursos (`htop`) y la edición de ficheros de configuración críticos, como los de Nginx. La gestión de contenedores Docker en producción, inspeccionando logs (`docker logs`) y gestionando su ciclo de vida, ha sido una lección práctica de DevOps. Asimismo, se ha materializado el concepto abstracto de «la nube», configurando grupos de seguridad en AWS, comprendiendo la interacción entre servicios como EC2 y RDS dentro de una VPC, y utilizando variables de entorno para una gestión segura de las credenciales.

A nivel metodológico, se ha validado la importancia crítica de una fase de planificación inicial robusta. La elección de la arquitectura y la selección del `stack` tecnológico son decisiones fundacionales que evitan una futura deuda técnica. Sin embargo, la lección más profunda ha sido aceptar la naturaleza incompleta de toda planificación. Por exhaustivo que sea el análisis, siempre surgirán imprevistos que obligan a desviarse del plan original. Este proyecto enseña que el objetivo de la planificación no es crear un camino rígido, sino construir una base sólida desde la cual poder adaptarse con agilidad y resiliencia.

Quizás la lección más madura y transferible a cualquier ámbito de la vida ha sido la de carácter estratégico. El desarrollo de software a gran escala pone a prueba la capacidad para tomar decisiones difíciles y combatir la «falacia del coste hundido»: la tendencia a seguir por un camino erróneo solo por el tiempo ya invertido. En varias ocasiones, fue crucial detenerse, evaluar el progreso con perspectiva y tener el coraje de pivotar. La decisión de descartar trabajo para adoptar un enfoque mejor, aunque a corto plazo parezca un retroceso, se convierte en el mayor avance a largo plazo, resultando en un producto más limpio, escalable y mantenible. Esta capacidad para la autocrítica y la reorientación estratégica es una de las competencias más valiosas que se han desarrollado.

En definitiva, este Trabajo de Fin de Grado ha sido una experiencia enriquecedora que ha permitido solidificar los conocimientos teóricos de la carrera en un producto tangible y útil, sentando unas bases sólidas para el futuro profesional.

5.2. Líneas de Trabajo Futuro

Si bien el proyecto en su estado actual es una plataforma funcional que cumple con los objetivos iniciales, su arquitectura ha sido diseñada con la escalabilidad en mente. Este apartado traza una hoja de ruta con una serie de futuras implementaciones de alto impacto que enriquecerían enormemente la plataforma, mejorando la experiencia de usuario, la seguridad y el potencial de negocio.

5.2.1. Implementación de Pasarela de Pago Profesional con Stripe

La integración de una pasarela de pago es el paso más crítico para evolucionar la plataforma de un prototipo funcional a una entidad comercial operativa. La elección de Stripe se fundamenta en su seguridad, confianza y facilidad de uso. Su implementación aporta un valor triple. Para el negocio, transfiere la responsabilidad del cumplimiento de la normativa PCI-DSS a Stripe, mitiga riesgos y ofrece un potente panel de control para analíticas de ventas. Para el usuario, proporciona un proceso de pago familiar y seguro, aumentando la confianza y la tasa de conversión. Técnicamente, la integración implicaría añadir la dependencia de `stripe-java` al proyecto. En el `backend`, se crearía un `endpoint` que genere un `PaymentIntent` a partir del importe del carrito, devolviendo un `client_secret` al `frontend`. En el cliente, se utilizaría `Stripe.js` para renderizar los elementos de la tarjeta de forma segura, garantizando que los datos sensibles nunca toquen el servidor. Finalmente, se configuraría un `webhook` para recibir eventos de Stripe (ej. `payment_intent.succeeded`) y actualizar el estado del pedido en la base de datos de forma fiable.

5.2.2. Integración de Inicio de Sesión Social con OAuth 2.0

Reducir la fricción en el registro es clave para la captación de usuarios. El inicio de sesión con una cuenta de Google, mediante el protocolo estándar `OAuth 2.0`, elimina la necesidad de crear y recordar nuevas credenciales. Esto mejora drásticamente la experiencia de usuario y aumenta las tasas de registro. Técnicamente, esta funcionalidad se puede implementar de forma nativa con Spring Security, añadiendo la dependencia `spring-boot-starter-oauth2-client`. El proceso requeriría configurar el fichero `application.properties` con el `client-id` y `client-secret` obtenidos desde la Google Cloud Platform Console. Sería necesario, además, personalizar la lógica del servicio de usuario (`OAuth2UserService`) para procesar la información devuelta por Google, verificando si el usuario ya existe en la base de datos local por su email o creando una nueva cuenta asociada.

5.2.3. Refuerzo de la Seguridad del Registro con reCAPTCHA

A medida que una aplicación gana visibilidad, se convierte en objetivo de bots que automatizan la creación de cuentas de spam. Para mitigar este riesgo, es fundamental implementar un sistema como Google `reCAPTCHA` v3. Esta versión, al

ser invisible, no interrumpe la experiencia del usuario legítimo. Su implementación técnica consiste en integrar el `script` de Google en la página de registro para obtener un token en el lado del cliente. Este token se envía al `backend` junto con los datos del formulario. El servidor, a su vez, debe realizar una petición a la API de Google `siteverify`, enviando su clave secreta y el token del usuario. Solo si la respuesta de Google valida la interacción como humana, se procederá a crear la cuenta, protegiendo así la integridad de la base de datos.

5.2.4. Publicación en Servidor de Producción con Plesk

El despliegue final de la aplicación se realizaría en el servidor de producción gestionado con el panel Plesk, donde la farmacia ya tiene contratado su dominio y alojamiento. Este paso es crucial para que el proyecto pase a un entorno operativo real. El proceso técnico consistiría en ejecutar la aplicación dentro de un contenedor Docker, gestionado a través de la extensión de Plesk o por acceso SSH. A continuación, se configuraría el servidor web del panel (Nginx o Apache) como proxy inverso para redirigir el tráfico del dominio al contenedor. Plesk facilitaría la gestión de los certificados SSL y, finalmente, se reconfiguraría la aplicación para conectarse a la base de datos proporcionada por el servicio de hosting.

5.2.5. Desarrollo de un Ecosistema de Notificaciones por Correo Electrónico

Una comunicación fluida y profesional es esencial tras una compra. Se propone implementar un sistema de correos automáticos tanto transaccionales (confirmación de pedido, notificación de envío) como de marketing (promociones, newsletters), siempre bajo consentimiento del usuario (RGPD). La implementación se realizaría aprovechando la infraestructura de correo ya contratada y gestionada a través del panel Plesk. Se configurarían direcciones de correo personalizadas (ej. pedidos@laboticademar.com), lo que refuerza la confianza y la imagen de marca. La aplicación Spring Boot utilizaría la dependencia JavaMailSender para conectarse al servidor SMTP proporcionado por el hosting.

5.2.6. Implementación de la Compra como Invitado (Guest Checkout)

La obligatoriedad de crear una cuenta es una de las principales causas de abandono del carrito. Habilitar una opción de çomprar como invitadoçaptura ventas de clientes que buscan una transacción rápida. Esta implementación requiere una adaptación en el modelo de datos. La entidad `Order`, que probablemente tenga

una relación con `User`, debería permitir una asociación nula. En su lugar, se almacenarían los datos del cliente (email, dirección) directamente en la orden. Para que el invitado pueda consultar el estado de su pedido, se le enviaría un correo con un enlace único y seguro a una página de seguimiento.

5.2.7. Adaptación de la Plataforma a Dispositivos Móviles

Dado que la mayoría del tráfico y las ventas en e-commerce se originan en móviles, una experiencia optimizada es fundamental para la viabilidad del proyecto. Un diseño que no sea adaptable resulta en una alta tasa de rebote y pérdida de clientes. Técnicamente, la solución consiste en aplicar un diseño *Responsive* utilizando CSS Media Queries para adaptar los estilos al tamaño de la pantalla, junto con sistemas de maquetación flexibles como Flexbox y CSS Grid para que la estructura se reorganice fluidamente. La correcta implementación de la metaetiqueta viewport sería igualmente esencial para garantizar una visualización adecuada en los navegadores móviles.

5.2.8. Creación de Contenido Legal y de Soporte

La transparencia y el cumplimiento legal son fundamentales en el `e-commerce`. Es imprescindible desarrollar y publicar páginas estáticas para los "Términos y Condicionesz la "Política de Privacidad", cumpliendo con normativas como el RGPD y la LSSI-CE. Paralelamente, la creación de una sección de "Preguntas Frecuentes"(`FAQ`) bien estructurada actúa como una primera línea de soporte. Esto reduce la carga de trabajo del personal de la farmacia al resolver dudas comunes de forma instantánea y empodera al usuario, mejorando su satisfacción. Técnicamente, implicaría crear vistas estáticas y enlazarlas de forma visible en el `footer` de la aplicación.

5.2.9. Integración de un Asistente Virtual con Inteligencia Artificial

Esta es la línea de trabajo más innovadora, capaz de posicionar la plataforma a la vanguardia tecnológica. Un chatbot entrenado con los datos de los productos, las políticas de la tienda y las `FAQ` podría ofrecer soporte 24/7. Su implementación podría realizarse utilizando plataformas como Google Dialogflow o directamente la API de OpenAI.

En conjunto, estas líneas de trabajo futuro no solo representan una lista de funcionalidades deseables, sino que conforman un testimonio del potencial y la es-

calabilidad con la que el proyecto fue concebido desde sus inicios. La plataforma actual es un cimiento sólido, una prueba tangible de que los objetivos planteados se han alcanzado con éxito. Sin embargo, su verdadero valor reside en su capacidad para evolucionar. El camino recorrido ha sido una intensa y gratificante experiencia de aprendizaje; el camino por recorrer está lleno de oportunidades para transformar este proyecto académico en un ecosistema digital de impacto real, demostrando que la ingeniería de software es, en esencia, un viaje continuo de construcción y mejora.

Bibliografía

[1] O. Corporation, "Java," Online, n.d. [Online]. Available: https://www.oracle.com/java/

[2] I. Pivotal Software, "Spring boot," Online, n.d. [Online]. Available: https://spring.io/projects/spring-boot

[3] A. S. Foundation, "Apache maven," Online, n.d. [Online]. Available: https://maven.apache.org/

[4] ——, "Apache tomcat," Online, n.d. [Online]. Available: https://tomcat.apache.org/

[5] I. Pivotal Software, "Spring security," Online, n.d. [Online]. Available: https://spring.io/projects/spring-security

[6] S. S. Team, "Spring security taglibs," Online, n.d. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/integrations/jsp-taglibs.html

[7] JSON.org, "Json," Online, n.d. [Online]. Available: https://www.json.org/json-en.html

[8] J. Project, "Java jwt: Json web token for java and android," Online, n.d. [Online]. Available: https://github.com/jwtk/jjwt

[9] O. Corporation, "Mysql," Online, n.d. [Online]. Available: https://www.mysql.com/

[10] I. Red Hat, "Hibernate orm," Online, n.d. [Online]. Available: https://hibernate.org/orm/

[11] H. D. Engine, "H2 database," Online, n.d. [Online]. Available: https://www.h2database.com

[12] J. Team, "Junit 5," Online, n.d. [Online]. Available: https://junit.org/junit5/

[13] M. Team, "Mockito," Online, n.d. [Online]. Available: https://site.mockito.org/

[14] S. Project, "Selenium," Online, n.d. [Online]. Available: https://www.selenium.dev/

[15] J. Team, "Jacoco," Online, n.d. [Online]. Available: https://www.jacoco.org/jacoco/trunk/doc/

[16] M. Foundation, "Javascript," Online, n.d. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[17] W. W. W. C. (W3C), "Html," Online, n.d. [Online]. Available: https://www.w3.org/TR/html52/

[18] ——, "Css," Online, n.d. [Online]. Available: https://www.w3.org/Style/CSS/

[19] E. Foundation, "Jakarta servlet api," Online, n.d. [Online]. Available: https://jakarta.ee/specifications/servlet/

[20] A. S. Foundation, "Tomcat embed jasper," Online, n.d. [Online]. Available: https://tomcat.apache.org/tomcat-9.0-doc/api/org/apache/jasper/servlet/JspServlet.html

[21] O. Corporation, “Javaserver pages standard tag library,” Online, n.d. [Online]. Available: https://javaee.github.io/jstl-api/

[22] I. Twitter, “Bootstrap,” Online, n.d. [Online]. Available: https://getbootstrap.com/

[23] T. Labs, “Tailwind css,” Online, n.d. [Online]. Available: https://tailwindcss.com/

[24] ModelMapper, “Modelmapper,” Online, n.d. [Online]. Available: https://modelmapper.org/

[25] P. Lombok, “Project lombok,” Online, n.d. [Online]. Available: https://projectlombok.org/

[26] A. S. Foundation, “Apache poi,” Online, n.d. [Online]. Available: https://poi.apache.org/

[27] dotenv java, “dotenv-java,” Online, n.d. [Online]. Available: https://github.com/cdimascio/dotenv-java

[28] I. Docker, “Docker,” Online, n.d. [Online]. Available: https://www.docker.com/

[29] M. Corporation, “Visual studio code,” Online, n.d. [Online]. Available: https://code.visualstudio.com/

[30] L. Torvalds, “Git,” Online, n.d. [Online]. Available: https://git-scm.com/

[31] O. Corporation, “Mysql workbench,” Online, n.d. [Online]. Available: https://www.mysql.com/products/workbench/

[32] I. GitHub, “Github,” Online, n.d. [Online]. Available: https://github.com/

[33] Postman, “Postman api platform,” Online, n.d. [Online]. Available: https://www.postman.com/

[34] I. Docker, “Docker hub,” Online, n.d. [Online]. Available: https://hub.docker.com/

[35] A. W. Services, “Amazon ec2,” Online, n.d. [Online]. Available: https://aws.amazon.com/ec2/

[36] ——, “Amazon rds,” Online, n.d. [Online]. Available: https://aws.amazon.com/rds/

[37] I. NGINX, “Nginx,” Online, n.d. [Online]. Available: https://www.nginx.com/

[38] L. Encrypt, “Let’s encrypt - free ssl/tls certificates,” Online, n.d. [Online]. Available: https://letsencrypt.org/

[39] P. I. GmbH, “Plesk,” Online, n.d. [Online]. Available: https://www.plesk.com/

[40] S. Tatham, “Putty,” Online, n.d. [Online]. Available: https://www.putty.org/

[41] N. L. Inc., “Notion,” Online, n.d. [Online]. Available: https://www.notion.so/