Universidad de Cantabria

Facultad
de
Ciencias

Desarrollo de una Aplicación Web para una Tienda de Artículos de Coleccionismo Deportivo

Development of a web application for a sports collectibles store

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Luis Álvarez Conde

Director: Marta Zorrilla Pantaleón

Septiembre - 2024

RESUMEN

En la actualidad, la digitalización de las empresas se ha convertido en un pilar esencial para asegurar su competitividad en el mercado. La transformación digital no solo mejora la eficiencia interna, sino que permite a las compañías aumentar su alcance, ofreciendo sus productos y servicios en todo el mundo.

Es por ello, que se propone el desarrollo de una aplicación web que va a posibilitar a una tienda de artículos de coleccionismo deportivo vender sus productos a nivel internacional, permitiendo a los clientes explorar los artículos ofertados por la tienda, identificarse con su cuenta, meter productos al carro de la compra y realizar pedidos. Con el desarrollo de esta aplicación, adquiriré una experiencia muy importante en las tecnologías más utilizadas en el ámbito del desarrollo web.

La parte del *frontend*, que contiene todas las vistas a través de las cuales el usuario puede interactuar con la aplicación, está desarrollada en React, mientras que el *backend*, que contiene la capa de negocio de la aplicación, está desarrollada en Node.js utilizando un patrón arquitectural REST. Esta capa funciona como un intermediario entre el *frontend* y la base de datos, que se implementó utilizando MySQL, junto con el ORM Sequelize para agilizar la interacción conla base de datos.

Palabras clave: Aplicación Web, React, Node.js, JavaScript, Tienda virtual.

ABSTRACT

Nowadays, the companies digitalization has become crucial to ensure their competitiveness in the market. Digital transformation not only improves internal efficiency, but also allows companies to expand their reach by offering their products and services worldwide.

Therefore, the development of a web application is proposed, which will enable a sports collectibles store to sell its products internationally, allowing customers to explore the items offered by the store, log in with their account, add products to the shopping cart, and place orders. With the development of this application, I will gain valuable experience in the most used technologies in web development.

The *frontend*, which contains all the views through which the user can interact with the application, is developed in React, while the *backend*, which contains the business logic of the application, is developed in Node.js using a REST architectural pattern. This layer acts as the intermediary between the frontend and the database, which is implemented using MySQL, along with the Sequelize ORM to make it easier to interact with the database.

Keywords: Web Application, React, Node.js, JavaScript, Online store.

Índice general

1. INTRODUCCIÓN Y OBJETIVOS

En este apartado, se va a exponer el contexto de la aplicación, y los objetivos que se persiguen de cara al desarrollo de la misma. Además, se describirá de manera breve la estructura de este documento

1.1. Contexto

En el mundo empresarial a día de hoy, es fundamental para las empresas disponer de una plataforma online que les permita ofrecer sus productos o servicios a clientes de todo el mundo. Entre otras ventajas que aportan este tipo de plataformas a las empresas están la capacidad de alcanzar un mayor número de clientes, superando las limitaciones físicas impuestas por la ubicación geográfica de un establecimiento, facilidad de la gestión del stock y además permite una contabilidad de la empresa más automatizada.

Debido a la alta demanda de tiendas online en el mercado actual, es esencial que los profesionales del desarrollo de software adquieran experiencia utilizando las últimas tecnologías y metodologías de desarrollo. Es por ello que se propone esta idea de desarrollar una aplicación web para una tienda ficticia de coleccionismo deportivo, que no solo sirve como una posible solución real para un hipotético negocio de coleccionismo, sino que aporta un aprendizaje que me resultará muy útil para mi futuro laboral como desarrollador.

1.2. Objetivos

A continuación, se detallarán los objetivos que se persiguen con el desarrollo de esta aplicación.

- Desarrollar una plataforma web responsiva y accesible que permita a los usuarios interactuar con la tienda desde cualquier dispositivo.
- Diseñar e implementar una base de datos para la gestión del inventario de la tienda y de los productos del carrito de la compra de los usuarios, proporcionando una administración clara y estructurada de los productos disponibles.
- Realizar un prototipado y diseño de la aplicación enfocado en la experiencia del usuario, para que la interacción persona-computador sea intuitiva y sencilla.
- Adquirir experiencia en la implementación de sistemas de autenticación de usuarios, asegurando la protección de datos y la seguridad en el acceso a la aplicación.
- Manejar correctamente el contexto de un carrito en una aplicación web, de manera que se mantenga la información de este entre sesiones.
- Desarrollar una API completa que cumpla con el estilo arquitectural REST.

- Realizar pruebas unitarias, de integración y de aceptación tanto en la capa de negocio como en la de presentación.
- Desplegar la aplicación una vez acabada la implementación completa como parte de un portfolio personal para mostrar la experiencia adquirida en el campo.

1.3. Estructura

En este apartado, se explica brevemente la estructura y contenido de esta memoria:

Primeramente, en el capítulo 2 se muestran las metodologías que se han seguido para desarrollar este proyecto, así como el conjunto de herramientas y tecnologías utilizadas y que han hecho mucho más eficiente la implementación de esta aplicación web.

El capítulo 3, contiene una descripción general del sistema en la que se detalla a grandes rasgos las funcionalidades principales de la aplicación. Después, se detalla la fase de análisis de requisitos con los actores del sistema, requisitos funcionales y no funcionales en formas de tablas, y los casos de uso, de los que se muestran los diagramas y la especificación. Por último, se muestran una serie de prototipos que se han diseñado para poder ver como tendría que quedar la aplicación una vez desarrollada.

Posteriormente, se explica en el capítulo 4 el diseño de la aplicación, mostrando algunos diagramas para ofrecer más claridad, así como el diseño de cada uno de sus componentes, base de datos, *backend* y *frontend*.

En el capítulo 5, se explican las decisiones que se han tomado en lo que respecta al desarrollo del código en cada uno de los componentes de la aplicación, explicando como se han utilizado cada una de las tecnologías y apoyando esta explicación con capturas de pantalla del código.

Por último, en el capítulo 6 se muestran las pruebas que se han realizado para comprobar el funcionamiento de los componentes y el capítulo 7 se utiliza a modo de conclusión del proyecto y se adjuntan los enlaces del código, prototipos y diagramas. Finalmente, en el capítulo 8 se muestra la bibliografía.

Las figuras y tablas que no se han podido mostrar a lo largo de la memoria, se encuentran en el Anexo, al final del documento.

2. TECNOLOGÍAS Y MÉTODOS

En este apartado se comentará la planificación con el detalle de tareas y su duración, a través de un diagrama de Gantt, las metodologías de desarrollo que se han seguido a lo largo de este proyecto y las tecnologías utilizadas.

2.1. Planificación del proyecto

En la Figura 1 podemos ver la planificación que se ha diseñado del proyecto. Observamos, como la fase que más tiempo lleva se trata de la fase de desarrollo, la cual suele ocupar alrededor del 40 % del tiempo empleado en el desarrollo de u proyecto software, y más aún en el este caso en el que se tienen que aprender muchas tecnologías.

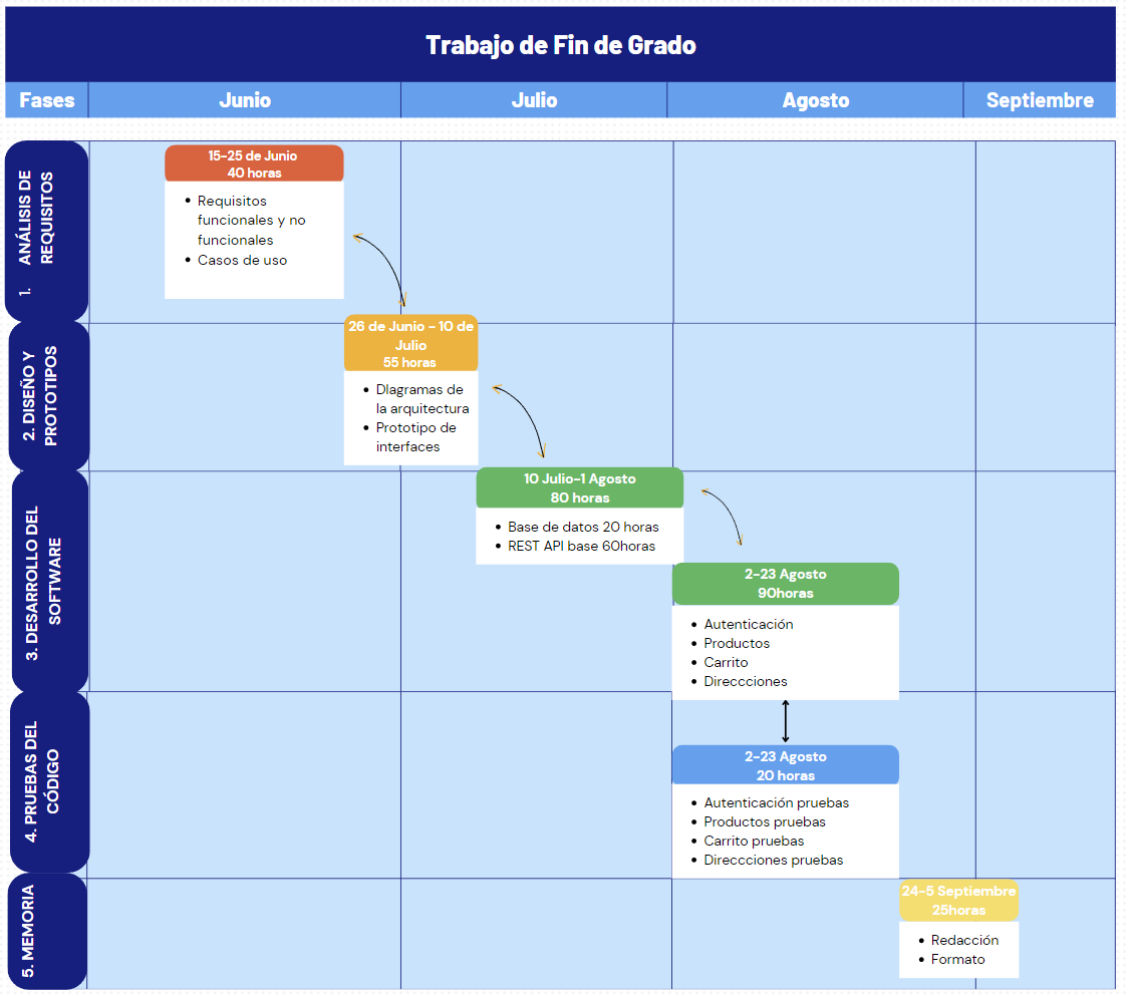

Figura 1: Planificación del proyecto

2.2. Metodología de desarrollo utilizada

Para este proyecto, como se ha podido observar en la Figura 1, se ha utilizado una combinación entre la metodología en cascada y la metodología incremental. Por una parte, se ha utilizado una metodología en cascada para las fases de análisis de requisitos, diseño, y desarrollo de la base de datos y parte de la REST API, es decir, se han desarrollado en orden una tarea detrás de otra. Por otro lado, una vez se acabó esta fase, se empezó a seguir una metodología incremental, en la que por cada caso de uso que se iba desarrollando, se desarrollaba el código junto con las pruebas correspondientes. De esta manera, se iban desarrollando pequeños incrementos que se integraban con el resto de la aplicación.

2.3. Tecnologías y herramientas

En este apartado, se citan todas las tecnologías que se han utilizado para el desarrollo del *backend* y del *frontend* así como para realizar el control de versiones.

2.3.1. Node.js

Node.js [1] es un entorno de ejecución de JavaScript de código abierto y multiplataforma que es muy popular hoy en día debido a las siguientes características

- Ejecuta el motor V8 de JavaScript, que es el núcleo de Google Chrome, por lo que en términos de rendimiento es muy eficiente.
- Las aplicaciones Node.js se ejecutan en un solo proceso, sin crear un nuevo hilo por cada solicitud. Además, proporciona un conjunto de primitivas en su librería estándar con las que se puede evitar que el código sea bloqueante, lo que permite manejar miles de conexiones simultáneas sin la complejidad de tener que manejar la concurrencia de hilos.
- Permite escribir tanto el *frontend* como el *backend* en el mismo lenguaje, JavaScript, sin la necesidad de aprender un lenguaje completamente diferente.
- Cuenta con una amplia comunidad y con un ecosistema repleto de módulos y paquetes disponibles a través del NPM (Node Package Manager) con los que se facilita añadir funcionalidades a las aplicaciones.

2.3.2. RESTful API

La arquitectura REST (Representational State Transfer) [2] es un estilo arquitectural para la programación de aplicaciones que utiliza peticiones HTTP para el acceso y manipulación de datos. Además, se caracteriza por no almacenar el estado entre peticiones, lo que significa que estas son independientes entre sí y cada una de estas solicitudes debe contener la información necesaria para procesar la petición. Además, cada recurso en esta arquitectura, está identificado por un URI (Uniform Resource Identifier) para poder distinguirlos de manera inequívoca.

2.3.3. Express.js

Consiste en un *framework* [3] de Node.js que proporciona una serie de librerías y herramientas que facilitan la construcción de aplicaciones web. En particular, nos hace mucho más sencilla la gestión de las rutas para distintas peticiones HTTP (GET, POST, PUT, DELETE, etc.). Otra característica principal que tiene este *framework* es que nos permite introducir *middlewares*, que consisten en capas intermedias entre la recepción de la petición y el envío de la respuesta. Estas capas nos permiten añadir funcionalidades como autenticación y autorización. Entre los *middlewares* que se han implementado en este proyecto y que se comentarán más adelante se encuentran, un gestor de errores, un control de la autenticación y un control de la autorización.

2.3.4. Sequelize

Sequelize es un ORM [4] (Object-Relational Mapper), que nos permite trabajar con bases de datos desde el paradigma de la programación orientada a objetos. En un ORM, se mapean las clases a las tablas y los atributos a las columnas, de esta manera conseguimos abstraer la base de datos y facilitar el trabajo del desarrollador.

Esta herramienta ha permitido interactuar con la base de datos a través de modelos que se han definido, y que han permitido realizar las consultas y operaciones con la base de datos sin la necesidad de escribir directamente el código. Esto un ahorro importante de tiempo.

2.3.5. Zod

Zod [5] es una biblioteca de validación de esquemas para TypeScript y JavaScript. Esta permite definir esquemas para validar los datos que se pasan a la aplicación, así como adaptarlos a otros tipos. Es muy útil porque permite realizar un manejo de errores de sencillo y estructurado.

2.3.6. MySQL

MySQL [6] es un sistema de gestión de base de datos relacional de código abierto, que se ha convertido en uno de los estándares más utilizados por la mayoría de aplicaciones empresariales gracias al rendimiento y escalabilidad, los mecanismos que proporciona para garantizar la integridad, seguridad (autenticación de usuarios, control de accesos, etc.), una gran comunidad y soporte para resolver los problemas que les surgen a los programadores y multiplataforma, compatible con Windows, Linux y macOS.

2.3.7. Jest

Jest [7] es un *framework* creado por Facebook que proporciona una serie de herramientas para realizar tests en JavaScript. Muy utilizado en el entorno del desarrollo web por la capacidad de ofrecer un entorno de pruebas con muy poca configuración.

Permite la ejecución de pruebas en paralelo por lo que es realmente útil en aplicaciones donde hay muchos casos de prueba, proporciona herramientas para medir

la cobertura del código, es decir, qué porcentaje del código se está probando y permite tanto pruebas unitarias (con el uso de mocks) como pruebas de integración con una configuración mínima.

2.3.8. Figma

Figma [8] es una herramienta utilizada para realizar prototipos web y aplicaciones móviles o de escritorio, que permite a los equipos de diseñadores trabajar en el diseño de la interfaz de una aplicación de manera concurrente.

Es una herramienta muy potente que permite generar prototipos interactivos, es decir, permite crear transiciones y enlazar vistas a través de botones, simulando el comportamiento que tendrá la web o aplicación que se está diseñando. Proporciona, además, una gran variedad de *plugins* para integrarlo con otras aplicaciones y soporta la reutilización de componentes, lo que aumenta mucho la productividad de los diseñadores.

2.3.9. React

React [9] es una biblioteca de JavaScript de código abierto, actualmente mantenida por Facebook y una comunidad de desarrolladores individuales. Se utiliza para construir la capa del *frontend* de las aplicaciones, en concreto es muy popular en la creación de aplicaciones SPA (Single Page Applications) proporcionando al usuario una experiencia interactiva y dinámica. Entre las características que proporciona esta biblioteca se encuentran:

- Componentes reutilizables: React está basado en estos componentes, que son bloques de construcción que se puede reutilizar en la interfaz y que cada uno encapsula la parte gráfica, el estilo y la lógica de negocio de ese componente.
- JSX: Es una extensión de la sintaxis de JavaScript que permite escribir HTML dentro de JavaScript, lo cual, facilita enormemente la creación de interfaces de usuario complejas de una manera más sencilla y clara.
- Flujo de datos unidireccional: Los datos en React se transmiten de los componentes principales a los secundarios, haciendo que el estado sea más predecible y sencillo a la hora de depurar.
- Virtual DOM: Se mejora enormemente el rendimiento, ya que solo se actualiza el DOM con los cambios que se han realizado y no todo el conjunto.
- Gran comunidad: React cuenta con una amplia comunidad y un ecosistema robusto que incluye numerosas herramientas como Redux o React Router.
- Compatibilidad: Además de aplicaciones web, se puede utilizar React Native para aplicaciones móviles.

2.3.10. Git

Git [10] es un sistema de control de versiones distribuido de código abierto, que permite controlar los cambios que se realizan en un proyecto durante el desarrollo software. Git se ha convertido en la herramienta más utilizada para el control de

versiones en el ámbito de los proyectos de desarrollo software. El código del proyecto completo se encuentra subido en el siguiente repositorio:
https://github.com/alvarezcondeluis/FanBox

2.3.11. Visual Studio Code

Visual Studio Code [11] es un editor de código desarrollado por Microsoft, gratuito, de código abierto y que es una de las herramientas más utilizadas por los desarrolladores de todo el mundo debido a la gran variedad de características que ofrece.

Una de las principales ventajas, es la capacidad de personalización y el amplio mercado de extensiones, donde los desarrolladores pueden añadir desde distintos soportes para diferentes lenguajes de programación, herramientas de depuración, herramientas de estilo, que muestran el código bien justificado y con distintos colores para que sea lo más legible posible para el programador. Destacar además su compatibilidad con distintos sistemas operativos (Windows, Linux y macOS), soporte para una gran variedad de lenguajes de programación y la integración con Git para aumentar la productividad del control de versiones directamente en el editor de código.

2.3.12. Postman

Postman [12] es una herramienta muy utilizada para la documentación y las pruebas de APIs, ampliamente utilizada en el mundo del desarrollo, ya que permite a los desarrolladores diseñar, documentar, depurar y probar APIs de manera eficiente y cómoda.

Una de las claves de esta herramienta es que nos facilita la creación y ejecución de peticiones HTTP a nuestra API mediante su interfaz gráfica sin necesidad de escribir código adicional. Permite organizar estas peticiones en distintas colecciones para agrupar, reutilizar y ejecutar las pruebas. Además, nos da la posibilidad de generar la documentación de la API de manera automática a partir de estas colecciones, especialmente útil en equipos de desarrollo donde la documentación es esencial para la colaboración e integración continua.

3. DESCRIPCIÓN DEL SISTEMA Y ANÁLISIS DE REQUISITOS

En este apartado, se realiza una descripción general de la aplicación y se detallan los requisitos funcionales y no funcionales que se deben satisfacer. Además, exponen los casos de uso y algunos de los prototipos de las vistas de la aplicación web.

3.1. Descripción del Sistema

La aplicación dispondrá de una página principal, donde el cliente podrá consultar los productos ofertados que se encuentran en la página, de manera que si quiere añadir alguno al carrito de la compra tendrá que iniciar sesión o registrarse si no lo ha hecho todavía. Una vez iniciada la sesión, el sistema detectará el rol del usuario y en función de este le permitirá desbloquear acciones de administrador o limitarlas a las de un cliente normal.

En el caso de que sean administradores, podrán abrir un menú para realizar la gestión de los productos en la base de datos. Este permitirá a los administradores ir actualizando la página con nuevos productos, variación de precios para aprovechar tiempos como las rebajas donde puede haber un aumento en las ventas y añadir descuentos para atraer nuevos clientes. Además, el objetivo de la página es que en un futuro se puedan mostrar al administrador los datos de las ventas y estadísticas acerca de los productos vendidos de cara a poder realizar un análisis de los mejores y peores productos. De esta manera, el administrador podrá analizar estos datos para ayudarle en las decisiones estratégicas de la empresa.

Por otro lado, a los clientes autenticados, se les mostrará la página principal en la que van a poder consultar los distintos productos y filtrarlos por las distintas categorías, precios, etc. Además, una vez hayan encontrado el producto que les interesa, si pinchan en él, accederán a una vista que contiene los detalles del producto seleccionado, donde se les mostrarán las fotos, la descripción detallada y alguna que otra característica del producto. En esta misma página tendrán la opción de añadir el producto al carrito, para proceder a su compra. Una vez se dirijan a la página de compra tendrán que seleccionar una dirección de envío, si no tienen una establecida tendrán que añadirla mediante un formulario, elegirán el método de pago que deseen y podrán aplicar algún tipo de descuento para reducir el precio de su compra. Por otro lado, podrán realizar la consulta del histórico de pedidos y de pedidos actuales, así como ver el estado de los mismos. Además, se les permitirá modificar los datos de su cuenta como el contacto o la dirección de envío de cara a los nuevos pedidos.

Todas estas funciones se podrán realizar a través de una interfaz intuitiva para que cualquier cliente sea capaz de realizar compras y consultar información de manera fácil y comprensible.

3.2. Análisis de requisitos

Los requisitos de un sistema [13] deben recoger la descripción de los servicios proporcionados por este y sus restricciones operativas. En primer lugar, se identifican los actores del sistema para posteriormente especificar los requisitos que se van a satisfacer.

3.2.1. Actores del sistema

Los actores del sistema son las personas que van a interactuar con nuestra aplicación web. En este caso, los actores de nuestra tienda de coleccionismo deportivo serán:

- **Cliente no registrado:** Este actor representa a todos aquellos clientes que acceden a la aplicación por primera vez y que no se han creado una cuenta todavía. Estos usuarios tienen permitido navegar a través del catálogo de productos ofertados, filtrándolos según sus prioridades y acceder al detalle de estos.
- **Cliente registrado:** Este actor representa a un usuario que ya ha accedido previamente a la página y que al intentar añadir un producto en el carrito de la compra se le ha redirigido al registro de la cuenta. Este usuario tiene permitidas todas las funcionalidades del cliente no autenticado en adición a la gestión de su cuenta, gestión de su carrito de compra y realizar pedidos.
- **Administrador:** Este actor representa a un administrador del sistema, es decir, una persona con autoridad dentro del negocio y que tiene la posibilidad de realizar distintas gestiones sobre los productos ofertados en la página. El administrador, al autenticarse, dispone de un menú mediante el cual puede añadir, eliminar o actualizar productos, de manera que se puedan realizar las gestiones sobre los productos ofertados en la base de datos de manera más cómoda. Además, dispone de la información de todos los usuarios de la aplicación.

3.2.2. Requisitos funcionales

Los requisitos funcionales [13] definen todas las funcionalidades y comportamientos que el sistema debe de realizar para poder cumplir con las necesidades del usuario y los objetivos del negocio, es decir, describen qué debe de hacer el sistema sin entrar en detalles de la especificación sobre como se va a implementar.

A continuación en la Tabla 1 se muestran los requisitos funcionales que debe satisfacer la aplicación.

Tabla 1: Requisitos funcionales del Sistema

ID	Nombre	Descripción
Autenticación y Gestión del Perfil		
RF01	Registro de Usuario	El sistema debe permitir a los nuevos usuarios registrarse mediante un formulario. Una vez se hayan registrado se les redirigirá a la vista de inicio de sesión.
RF02	Inicio de Sesión	Los usuarios registrados deben poder iniciar sesión utilizando su correo electrónico y contraseña. El sistema generará un token JWT para manejar la sesión del usuario autenticado.
RF03	Recuperación de Contraseña	Los usuarios deben poder recuperar su contraseña a través de un proceso de recuperación.
RF04	Modificación de perfil	El usuario puede modificar datos de perfil como dirección de envío y contacto.
RF05	Cerrar sesión	El usuario una vez autenticado puede cerrar sesión para autenticarse con otra cuenta.
Gestión de los pedidos		
RF06	Creación de pedido	El cliente puede añadir artículos al carrito, elegir dirección y forma de pago y posteriormente confirmar el envío.
RF07	Generación de código de pedido	Cada pedido recibe un código único para su seguimiento.
RF08	Confirmación de pedido	El sistema envía un correo de confirmación tras la creación del pedido.
RF09	Consulta de pedidos	El usuario puede consultar su histórico de pedidos, incluyendo detalles como artículos, precio, y estado del pedido.
Procesamiento de Pagos		
RF10	Método de pago	El sistema permite elegir entre varios métodos de pago, como PayPal, tarjetas de crédito o transferencia bancaria.
RF11	Confirmación de pago	El sistema muestra una confirmación visual tras el pago exitoso y le redirige a la confirmación del pedido.
Búsqueda y Filtrado de Productos		
RF12	Búsqueda de productos	El sistema permite buscar productos por el nombre del producto.
RF13	Filtrado de productos	El sistema permite filtrar productos por precio, categoría, deporte y talla.
RF14	Ver detalles del producto	El usuario puede acceder a la vista del detalle de un producto concreto, donde se le mostrarán las imágenes del producto, las tallas disponibles, el precio, nombre y descripción detallada de este.
Gestión del Carro de la Compra		

ID	Nombre	Descripción
RF15	Añadir al carrito	El usuario puede añadir productos al carrito, recalculando el precio total.
RF16	Eliminar del carrito	El usuario puede eliminar productos del carrito, actualizando el total.
RF17	Modificar cantidad	El usuario puede cambiar la cantidad de productos en el carrito, actualizando el precio total.
RF18	Aplicar descuento	El sistema permite aplicar un código de descuento, ajustando el precio total del carrito.
Administración de los Productos		
RF19	Añadir productos	Los administradores pueden agregar nuevos productos introduciendo la información necesaria: nombre, descripción, deporte, marca, categoría y equipo en caso de haberlo. Además, se añadirán las imágenes y las unidades del producto.
RF20	Editar productos	Los administradores pueden editar la información de los productos existentes.
RF21	Eliminar productos	Los administradores pueden eliminar productos de la tienda.

3.2.3. Requisitos no nuncionales

Los requisitos no funcionales [13] son las especificaciones que establecen cómo va a funcionar un sistema, no qué funciones va a realizar. Estos requisitos definen las cualidades y propiedades para cumplir con las expectativas de calidad, seguridad, rendimiento, entre otros aspectos.

En la Tabla 2 se pueden observar los requisitos no funcionales que debe de cumplir esta aplicación.

Tabla 2: Requisitos funcionales del Sistema

ID	Nombre	Descripción
Seguridad		
RNF01	Autenticación y autorización	Solo se podrá acceder al sistema mediante un usuario y contraseña válidos. Se deben diferenciar los roles de usuario y administrador, permitiendo a estos últimos realizar gestiones adicionales.
RNF02	Encriptación de contraseñas	Las contraseñas de los usuarios deben almacenarse en la base de datos utilizando un algoritmo de encriptación seguro.
RNF03	Protección de datos	Los datos sensibles, como contraseñas y detalles de pago, serán ocultados de manera parcial o total al usuario para evitar posibles ataques.
RNF04	Uso de *tokens* JWT	El sistema debe utilizar *tokens* JWT para manejar la autenticación de sesiones de manera segura y sin estado.

ID	Nombre	Descripción
RNF05	*Middlewares*	El sistema, mediante una serie de *middlewares*, vetará el acceso tanto a las vistas de la web como a las llamadas de la API, para evitar que un usuario sin acceso pueda obtener datos de otra persona. Además, se utilizará un gestor de errores para evitar que a través de los mensajes de error se muestre información sensible acerca de la implementación.
Usabilidad		
RNF06	Interfaz intuitiva	La aplicación debe proporcionar una interfaz de usuario intuitiva que permita a los usuarios realizar operaciones de manera sencilla y eficiente.
RNF07	Interfaz estilizada y consistente	La interfaz de la aplicación debe ser visualmente atractiva para los usuarios, con un estilo moderno y que invite a los usuarios a navegar por esta. Además, tiene que ser coherente con todos sus elementos y vistas, es decir, utilizar una combinación de colores parecida, coherencia en los componentes, etc.
RNF08	Retroalimentación de las acciones	El usuario debe de recibir retroalimentación por parte del sistema de manera inmediata tras haber realizado alguna acción importante.
Accesibilidad		
RNF09	Adaptabilidad a tamaños de pantalla	La aplicación debe adaptarse correctamente a distintos tamaños de pantalla y navegadores, garantizando la accesibilidad en dispositivos móviles y de escritorio.
RNF10	Compatibilidad con navegadores	La aplicación debe ser compatible con los navegadores más utilizados, como Chrome, Firefox, Safari, y Edge.
RNF11	Contraste de colores adecuado	La aplicación debe de utilizar combinaciones de colores que proporcionen un contraste adecuado para poder diferenciar entre el fondo y los distintos elementos.

3.2.4. Casos de uso

En este apartado, se mostrarán los distintos casos de uso que tendrá la aplicación, primero se mostrará un diagrama para los distintos actores que interaccionen con el sistema y posteriormente se especificarán aquellos más relevantes. Un caso de uso [13] consiste en una descripción detallada sobre cómo los actores interactúan con el sistema.

3.2.4.1. Diagramas de Casos de Uso

En este apartado se muestran los diagramas de los casos de uso para cada uno de los actores del sistema. En la Figura 2 se muestran los casos del uso del usuario

no registrado, en la Figura 3 y la Figura 4 se muestran los del usuario registrado, y en la Figura 5 se muestran los correspondientes al administrador.

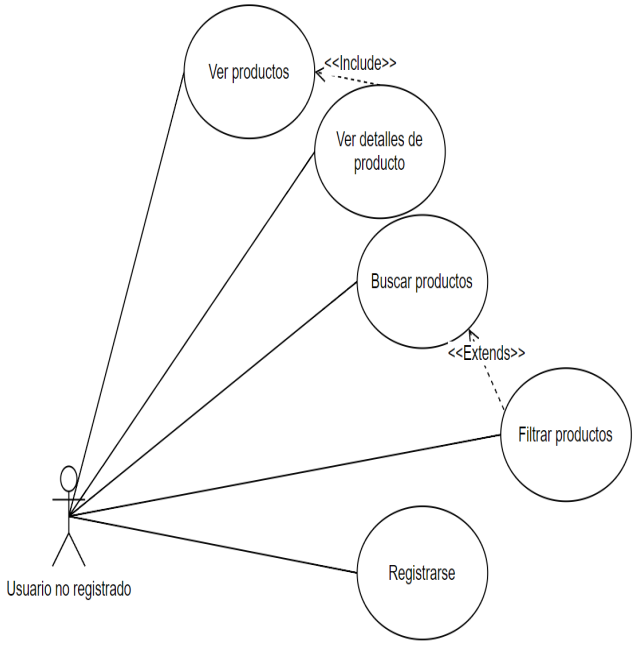

Figura 2: Usuario No Registrado

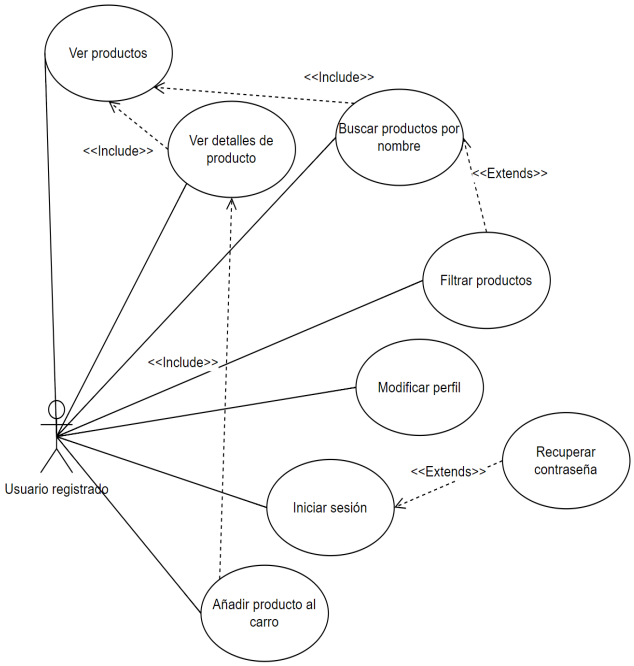

Figura 3: Usuario Registrado

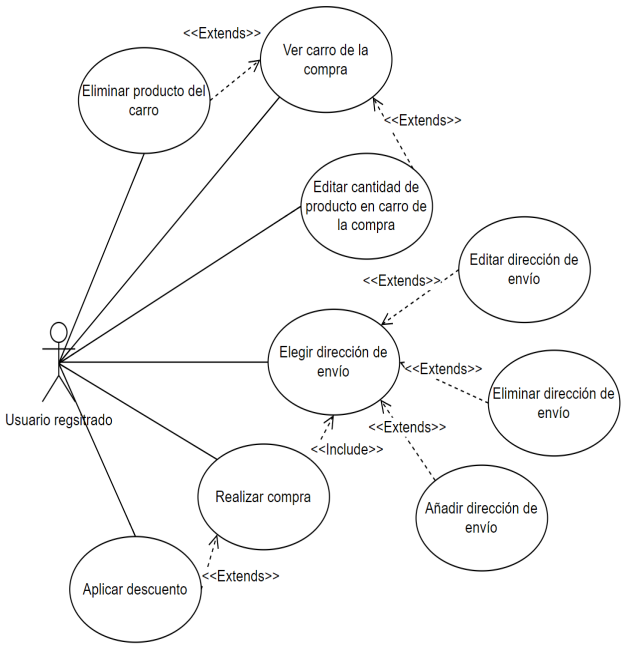

Figura 4: Usuario Registrado

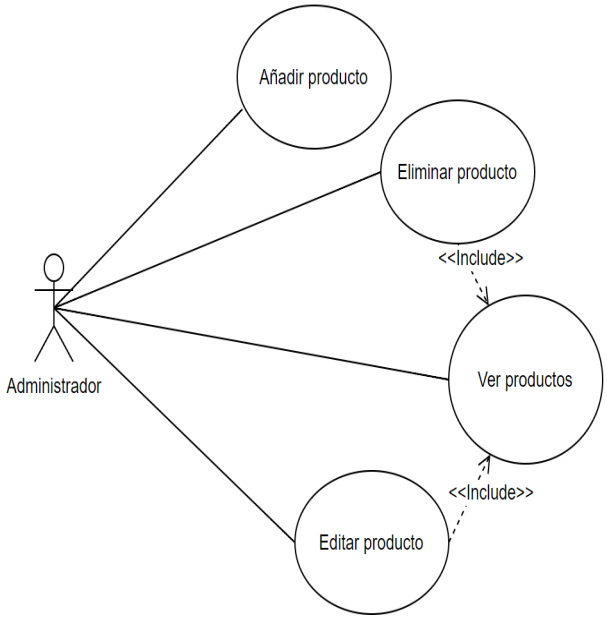

Figura 5: Administrador

3.2.4.2. Especificación de los casos de uso

A continuación, se muestran en forma de tablas la especificación de los casos de uso que se han considerado más relevantes. En la Tabla 3 se muestra la especificación

del caso de uso 'Iniciar Sesión' y en la Tabla 4 la especificación del caso de uso 'Registrarse'. El resto de las especificaciones se encuentra en el Anexo 9.1.

Tabla 3: Especificación del Caso de Uso: Iniciar Sesión

Caso de Uso	Iniciar Sesión
ID	CU03
Descripción	El usuario accede a su cuenta introduciendo el correo electrónico y la contraseña.
Actores Principales	Cliente registrado
Precondiciones	▪ El usuario debe tener una cuenta registrada en el sistema. ▪ El sistema debe tener una conexión estable con la base de datos.
Flujo Principal	1. El usuario accede a la página principal de la aplicación. 2. El usuario pulsa el botón de Iniciar Sesión. 3. El sistema dirige al usuario al formulario de autenticación. 4. El usuario introduce su correo electrónico y contraseña. 5. El sistema valida las credenciales. 6. El sistema genera un token de sesión y autentica al usuario. 7. El sistema redirige al usuario a la página principal con la sesión autenticada.

Flujos Alternativos	**4.1 Contraseña incorrecta:** 4.1a El sistema detecta que la contraseña introducida es incorrecta. 4.1b El sistema muestra un mensaje de error indicando que la contraseña no es la correcta. 4.1c El usuario vuelve a introducir la contraseña de nuevo. **4.2 Correo electrónico no existe:** 4.2a El sistema detecta que el correo electrónico no coincide con ninguna cuenta. 4.2b El sistema muestra un mensaje de error indicando que el correo introducido no existe. 4.2c El usuario tiene la opción de reintentar la autenticación. **5.1 Error en la conexión con la base de datos:** 5.3a El sistema intenta validar las credenciales, pero hay un error en la conexión con la base de datos. 5.3b El sistema muestra un mensaje informativo indicando un error en la conexión.

Tabla 4: Registrarse

Caso de Uso	Registrarse
ID	CU04
Descripción	Un nuevo usuario puede registrarse en el sistema proporcionando sus datos personales para crear una cuenta.
Actores Principales	Cliente No Registrado
Precondiciones	▪ El sistema debe estar conectado a la base de datos para almacenar los datos de registro.

Flujo Principal	1. El usuario accede a la página principal de la aplicación. 2. El usuario pulsa el botón de Iniciar Sesión. 3. El sistema dirige al usuario al formulario de autenticación. 4. El usuario selecciona el botón para crear una nueva cuenta. 5. El sistema muestra el formulario de registro. 6. El usuario completa el formulario de registro con su nombre, apellidos, correo electrónico, fecha de nacimiento, número de teléfono y contraseña. 7. El usuario pulsa en Crear Cuenta. 8. El sistema valida los datos. 9. El sistema almacena los datos del usuario en la base de datos. 10. El sistema redirige al usuario a la página principal de inicio de sesión para su autenticación.
Flujos Alternativos	**8.1 Correo Electrónico ya Registrado:** 8.1a El sistema detecta que el correo ya está registrado. 8.1b El sistema muestra un mensaje de error indicando que el correo electrónico ya está en uso. 8.1c El usuario tiene la opción de usar un correo electrónico diferente o iniciar sesión. **8.2 Datos introducidos incorrectos:** 8.2a El sistema detecta un error en los datos introducidos. 8.2b El sistema muestra un mensaje indicando el campo que está generando el error. **8.3 Error en la conexión con la base de datos:** 8.3a El sistema intenta registrar al usuario, pero hay un error en la conexión con la base de datos. 8.3b El sistema muestra un mensaje informativo indicando un error en la conexión.

3.3. Prototipos de la aplicación

A continuación, se muestran algunas imágenes de los prototipos diseñados para la aplicación web. Se adjunta el link del proyecto completo donde se encuentran todas las vistas de la aplicación: https://www.figma.com/proyecto

Primeramente, al entrar en la aplicación web, se abrirá una página como la que se muestra en la Figura 6, en donde se observa el logo principal de la página, alguna información adicional y los productos recomendados.

Figura 6: Página principal

Al seleccionar un producto, el sistema redirigirá al usuario a una página como la que se muestra en la Figura 7, en la que se mostrarán las imágenes del producto, las tallas disponibles y el precio, junto con la información general de este.

Figura 7: Vista de detalle del producto

Como se ha podido observar, se han elegido una combinación de colores que den un estilo profesional y moderno a la aplicación. Se ha diseñado de manera que la interfaz sea lo más profesional posible, simulando los detalles de cada página como si fuese una tienda real.

Se ha realizado este diseño estudiando comercios similares de los que se han sacado ideas sobre los detalles que mostrar, como es el caso del pie de página. Estos diseños han ayudado mucho a realizar la implementación, se han utilizado como guía para realizar la implementación real de la interfaz del usuario, por lo que no son exactamente iguales.

4. DISEÑO SOFTWARE

En este apartado, se describe el diseño arquitectural de la aplicación y el diseño detallado de cada uno de los componentes que componen la aplicación.

4.1. Diseño arquitectural

En este apartado se presenta la arquitectura general de la aplicación. En este caso se ha optado por una arquitectura en tres capas, lo que facilita el mantenimiento, organización y la escalabilidad del sistema. Las capas son las siguientes:

- **Capa de Presentación:** Esta capa se encarga la interfaz del usuario. Es la capa con la que el usuario interactúa directamente y a través de la cual puede ejecutar los diferentes casos de uso. Para ello, lanza peticiones a los métodos de la capa de negocio y muestra al usuario los resultados.
- **Capa de Negocio:** Capa encargada de manejar toda la lógica de la aplicación, recibe las peticiones de la capa de presentación y si es necesario, interactúa con la base de datos para recuperar, almacenar, actualizar o eliminar información. Es la capa intermediaria entre la capa de presentación y la capa de datos.
- **Capa de Datos:** Responsable del almacenamiento y gestión de los datos de la aplicación. Procesa las solicitudes que llegan de la capa de negocio y garantiza un acceso seguro y eficiente a los datos. Se utiliza un ORM para minimizar el desacople de impedancia entre el modelo relacional y el objetual. Facilitando, la gestión de sesiones, transacciones y posibles migraciones a otro gestor, al tiempo que se reutiliza código reduciendo los costes. En contra partida, el rendimiento y la flexibilidad en las consultas se ven afectados.

Este patrón de diseño nos ofrece una serie de ventajas como un mantenimiento sencillo, escalabilidad, mayor seguridad y facilidad en las pruebas y la integración continua. En la Figura 8 se puede observar con mayor claridad como se distribuyen cada una de estas capas.

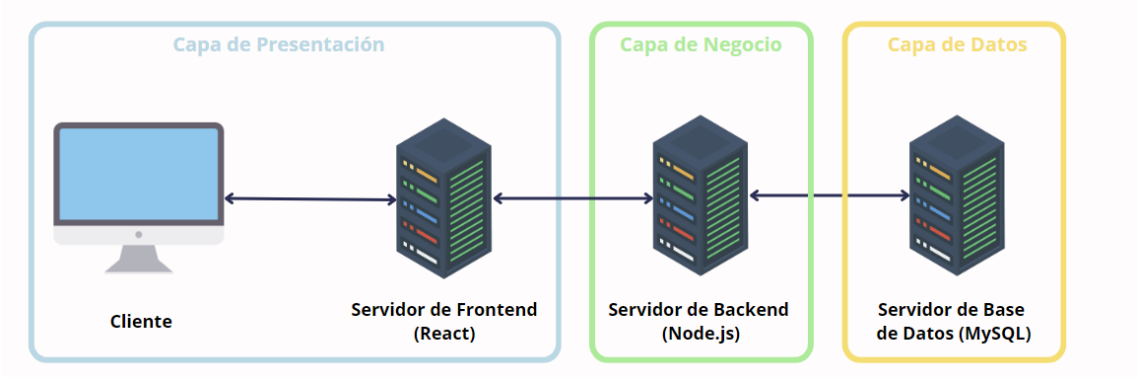

Figura 8: Arquitectura en tres capas

4.2. Diseño de la base de datos

Para abordar el diseño de la base de datos, se ha realizado el diseño conceptual utilizando la herramienta draw.io, desarrollando el diagrama conceptual del sistema de información usando la anotación ER. Se proporciona a continuación el link del diseño en el Anexo 9.2.

Un esquema de Entidad-Relación es muy útil en el proceso de diseño de una base de datos, ya que representa de manera muy visual las entidades y las relaciones entre ellas. A continuación, se comentarán las decisiones tomadas en el desarrollo de este diagramma.

La entidad débil 'productunit' se añade para separar la información general de un producto, que no suele cambiar, de la información del stock, precio, tamaño y peso. Las entidades Categoría y Equipo se añaden para limitar los valores que pueden tomar estos campos del producto. Las relaciones entre las tablas se han diseñado de manera que haya la menor duplicidad posible. Además, se añade la entidad de CartItem con el objetivo de poder persistir los productos que el usuario introduzca en el carrito para poder recuperarlos entre sesiones.

A continuación, se realizó el paso a tablas de manera manual, ya que no se disponía de una licencia para una herramienta CASE que generase el código automáticamente. Una vez definidas las tablas y las restricciones de integridad, se sintonizó la BD en MySQL. Posteriormente, se programaron disparadores para:

1. Controlar que un producto tenga una única imagen principal.
2. Validar las fechas de un descuento
3. Verificar que haya stock de un producto antes de añadirlo al carrito

4.3. Diseño del backend

En este apartado se va a describir el proceso de diseño para el desarrollo de la capa de negocio de la aplicación. En el Anexo 9.4, se muestra una tabla que contiene una descripción de las rutas que se han desarrollado en esta API.

Se ha dividido las rutas en función de los recursos identificados en la aplicación, usuarios, productos, pedidos, etc. En cada una de ellas, se puede realizar las principales operaciones CRUD. Además, se ha añadido una ruta extra para manejar el registro y autenticación del usuario. También se ha añadido una ruta para poder recuperar directamente una imagen principal y para el estado de un pedido, ya que son recursos que se van a consultar frecuentemente.

Por otro lado, para no tener que realizar para optimizar el rendimiento y realizar un menor número de peticiones, en la ruta `/api/products/`, se devuelve la lista de los objetos junto con las imágenes y el precio de cada uno.

Podemos observar que es una API completa que cumple con el paradigma REST. Cada recurso está identificado por una URI única y no se mantiene el estado entre las peticiones HTTP.

En cuanto a la arquitectura elegida para desarrollar esta API, se ha seguido un enfoque modular y escalable, en el que se han dividido las responsabilidades en tres componentes principales:

- **Routers:** Son responsables de mapear las solicitudes HTTP entrantes a cada controlador, es decir, se encargan de definir las rutas de la API. Además, gestionan los *middlewares* que se aplican a cada ruta, de manera que se garanticen los requisitos de seguridad y autenticación mediante un control de acceso por token JWT (JSON Web Token), que se utilizan para transmitir información entre el cliente y el servidor de manera segura.
- **Controllers:** Los controladores contienen los métodos que procesan cada petición. Estos métodos se encargan de extraer los argumentos necesarios, ya sean en la ruta, parámetros de consulta y el cuerpo de la petición y se les pasa a los servicios. Además, se encargan de devolver las respuestas correspondientes al cliente en función de los resultados de los servicios.
- **Services:** Encapsulan la lógica de negocio de la aplicación. Interactúan con la base de datos utilizando modelos ORM, definidos con `Sequelize` y que nos permiten interaccionar con la base de datos de manera segura y sin código SQL. Además, realizan la validación de los datos de entrada, ya sean parámetros de consulta, de ruta o cuerpos de solicitudes HTTP.

Esta modularización nos ofrece varias ventajas, como una mayor comprensión del código gracias a la modularización que aporta una mayor claridad y legibilidad sobre este. Además, aumenta la mantenibilidad del código debido a la facilidad de localización de los errores. Reducimos además la duplicidad del código y aumentamos la mantenibilidad.

4.4. Diseño del frontend

En esta fase del diseño de la capa de presentación, se ha escogido un criterio que garantiza la modularidad, mantenibilidad y escalabilidad del código. A continuación se explica el diseño que se ha llevado a cabo para esta capa.

La aplicación es un SPA (Single Page Application) en donde existe un único fichero HTML que sirve de contenedor de la interfaz del usuario. Esto permite tener una página que renderiza nuevo contenido en función de las interacciones del usuario. Este enfoque no solo mejora la fluidez y la experiencia del usuario evitando recargas de página, sino que también mejora el rendimiento al minimizar la cantidad de datos transferidos en cada interacción. Se ha dividido en diferentes carpetas que son las páginas, componentes y servicios, cada una con una función distinta.

En la Figura 9 se observa como se relacionan cada uno de estos elementos para formar la arquitectura del *frontend* de la aplicación, se adjuntan los ejemplos de dos

de las páginas desarrolladas.

La aplicación tiene un único HTML que se va renderizando con nuevo contenido a través del Router, que es el que se encarga de gestionar las rutas de la aplicación y de mostrar la página correspondiente. Cada una de estas páginas utiliza componentes, que se pueden reutilizar, como el caso del componente Layout, que contiene los componentes Header y Footer. Estos componentes se comunican y cambian el estado de las páginas en función de las interacciones del usuario.

Por otro lado, tanto los componentes como las páginas, utilizan los servicios desarrollados para realizar peticiones a la API. En el caso de la página 'LandingPage' utiliza directamente el servicio y le pasa los resultados a los componentes hijos, mientras que en el caso de la 'CheckoutPage', son los componentes quienes utilizan el servicio, en este caso a través del contexto que se explicará más adelante.

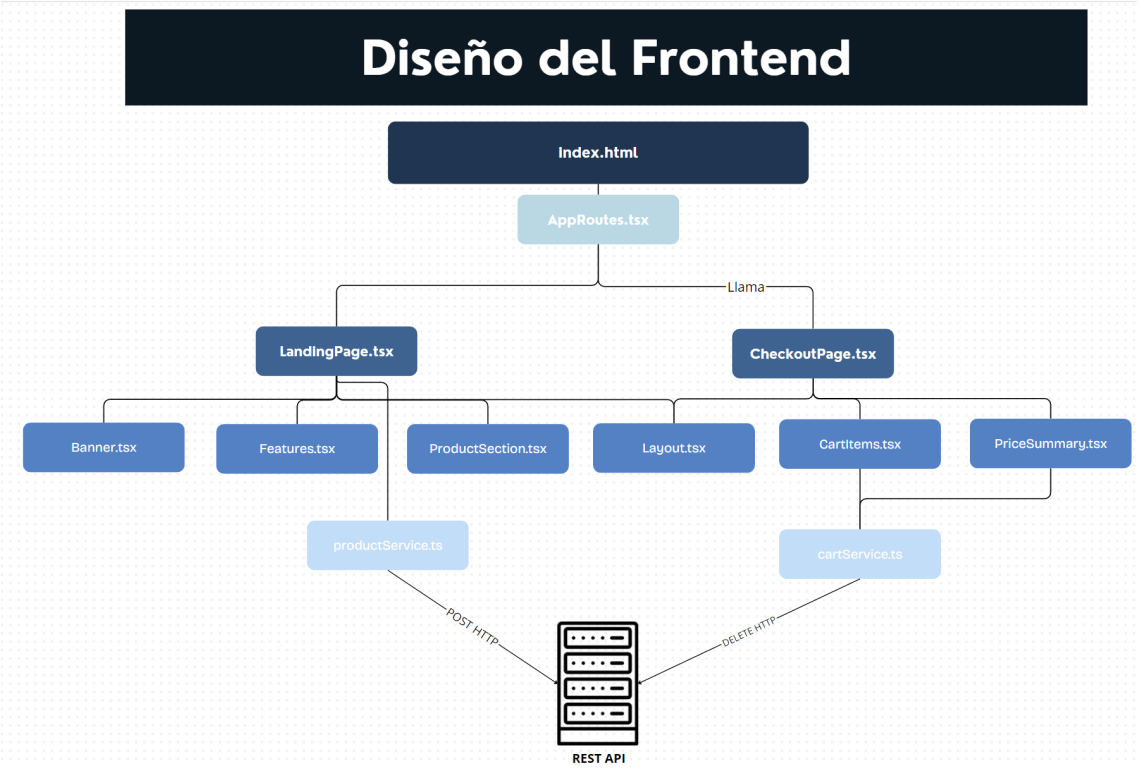

Figura 9: Diseño Frontend

5. DESARROLLO DEL SOFTWARE

En este apartado, se detallarán las decisiones tomadas y explicaciones necesarias sobre el código implementado. Esta etapa ha sido la más duradera de todas debido a que ha supuesto una dificultad añadida el aprendizaje de las tecnologías utilizadas.

5.1. Base de Datos

Comenzando con la Base de Datos, utilicé DBeaver, herramienta de gestión de bases de datos, para crear los scripts que generan las tablas y disparadores. Estos códigos están disponibles en el Anexo 9.3.2.

Se ha tomado la decisión, de crear una tabla intermedia para la relación, *many to many*, entre las unidades de producto y los pedidos. En la Figura 10 se adjunta el fragmento de código que representa la estructura de esa tabla.

```
DROP TABLE IF EXISTS OrderProductUnit;
CREATE TABLE OrderProductUnit (
    quantity INT NOT NULL,
    price DECIMAL(6,2) NOT NULL,
    orderID CHAR(36) NOT NULL,
    productID CHAR(36) NOT NULL,
    productNumber INT NOT NULL,
    PRIMARY KEY (orderID, productID, productNumber),
    FOREIGN KEY (orderID) REFERENCES `Order`(orderID) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (productID, productNumber) REFERENCES ProductUnit(productID, productNumber) ON DELETE CA
    CHECK (quantity > 0)
) ENGINE=InnoDB;
```

Figura 10: Tabla OrderProductUnit

Se puede observar como se relaciona tanto con la unidad de producto, que tiene una llave primaria compuesta por dos campos, y con los pedidos a través de `orderID`. Esta relación almacena la cantidad de unidades y el precio por unidad. Además, se ha tomado la decisión de utilizar un `UUID` para las claves primarias de algunas de las tablas, por eso `productID` es de tipo char(36).

Un `UUID` es un identificador único universal que nos permite ahorrarnos colisiones entre datos y nos proporcionan mucha seguridad al ser tan difíciles de adivinar. Se ha tomado esta decisión únicamente en aquellas tablas que contienen información sensible como productos, pedidos, usuarios y descuentos. No se utiliza en el resto, ya que a pesar de tener estas ventajas, hacer consultas y actualizaciones es muy costoso.

Además, en la parte final del script se han introducido una serie de índices no agrupados en las tablas que se van a consultar en mayor medida. Con esto optimizamos el rendimiento de las consultas a tablas a las que se accede con frecuencia.

Por otro lado, se han implementado ciertas restricciones básicas para garantizar la integridad y consistencia de los datos, como restricciones para el código postal o

el número de teléfono.

Por último, he implementado algunos disparadores para controlar los comportamientos a la hora de insertar o actualizar. En el Anexo 9.3.2.2 se puede ver el SQL que los contiene.

- Disparador imagen principal: Este disparador se encarga de controlar que cuando se añade una imagen a un producto, que si es principal, solo se añada si no hay ninguna que lo sea. De esta manera me aseguro de que solo haya una imagen principal por producto.
- Disparador descuento: Este disparador se encarga de verificar que la fecha actual esté dentro del intervalo entre la fecha de inicio y fin del descuento. Con esto evitamos la inserción de descuentos no válidos.
- Disparadores inventario: Los dos últimos disparadores controlan la inserción y actualización de la tabla cartItem. Controlan que no se puedan añadir productos al carrito, que no tengan suficiente stock.

5.2. Backend

En este apartado se comentarán las decisiones tomadas y se mostrarán algunos ejemplos del código para respaldarlas. El *backend* se divide en varios elementos:

- Modelos: Representan las distintas entidades representadas utilizando Sequelize, donde se definen los atributos de cada uno y las asociaciones con las demás entidades.
- Controladores: Responsables de manejar las solicitudes HTTP, obteniendo las cabeceras y pasar los parámetros correspondientes a la capa de negocio, y posteriormente gestionar la respuesta.
- Servicios: Encapsulan la lógica de negocio de la API, utilizando los modelos para interaccionar con la base de datos.
- Routers: Definen las rutas de la aplicación y asignan a cada una de ellas un método del controlador específico.
- `Middlewares`: Funciones que se ejecutan antes de llegar al controlador utilizado para la autenticación y manejo de errores.
- Schemas: Son esquemas que se utilizan para validar la estructura y contenido de los datos que se reciben en las solicitudes HTTP.

5.2.1. Modelos

A continuación se muestran unos ejemplos de los modelos que se han implementado con `Sequelize` para hacer más sencilla la interacción con la base de datos. Primero de todo, he configurado la conexión con la base de datos utilizando variables de entorno en un fichero `.env`. Esto se ha implementado en el fichero `/config/database.js`.

Primeramente, definimos el modelo especificando el nombre de la tabla y posteriormente los campos definiendo las claves primarias, los tipos de dato y si puede tomar valores nulos o no. En la Figura 11 se puede observar el modelo definido para la entidad de los productos.

```
export default function(sequelize, DataTypes) {
  const Product = sequelize.define('product', {
    productID: {
      type: DataTypes.CHAR(36),
      allowNull: false,
      primaryKey: true,
      defaultValue: DataTypes.UUIDV4
    },
    name: {
      type: DataTypes.STRING(255),
      allowNull: false
    },
    description: {
      type: DataTypes.TEXT,
      allowNull: false
    },
    sport: {
      type: DataTypes.STRING(50),
      allowNull: false
    },
    brand: {
      type: DataTypes.STRING(50),
      allowNull: false
    },
    releaseDate: {
      type: DataTypes.DATE,
      allowNull: false,
      defaultValue: Sequelize.NOW
    },
    isActive: {
      type: DataTypes.BOOLEAN,
      allowNull: false,
      defaultValue: true
    },
    categoryID: {
      type: DataTypes.INTEGER,
      allowNull: false,
```

Figura 11: Modelo de Producto

Además, he añadido las asociaciones a los modelos, como podemos observar en la Figura 12, que nos permiten realizar consultas en varias tablas a la vez de manera sencilla. En cada asociación, le indicamos el tipo (hasMany, belongsTo), el campo que referencia a la tabla y por último, en algunos casos, el borrado en cascada, es decir, en caso de que se borre el producto, se borren las tablas que lo referencian.

Figura 12: Asociaciones del modelo producto

5.2.2. Servicios

En cada servicio se han añadido los métodos necesarios para realizar las operaciones CRUD. Con la ayuda de los modelos nos ahorramos escribir el código SQL.

En la Figura 13, podemos observar como se han utilizado las anteriormente comentadas asociaciones para obtener los productos que se van a mostrar en la página principal. De cada producto se obtiene, la información del propio producto, la imagen principal y el precio, que se obtiene de una unidad del producto. De esta manera, nos ahorramos tener que hacer tres peticiones al servicio por cada producto que se muestre.

En la Figura 14, podemos ver el método que gestiona las creaciones de los productos. En primer lugar, validamos la estructura y datos, en caso de error, se lanza un error al controlador indicando el campo que ha fallado. Después, generamos el `UUID`, asignamos la fecha de hoy y creamos el producto. Podemos observar también, como se utiliza una clase de error que he creado que tiene como atributos el código del error y el mensaje, esto se hace para facilitar la gestión de errores en el *middleware*.

Por último, en la Figura 15 vemos como esta vez, he realizado la consulta SQL a mano, debido a que tuve unos problemas con las asociaciones. En este caso, Sequelize nos proporciona un control de inyección SQL a través del argumento `replacements`.

```
async getAll(filters) {
  try {

    // Incluir la tabla ProductUnit en la consulta y filtrar por precio si es necesario
    const products = await this.ProductModel.findAll({
      where,
      include: [
        {
          association: 'images',  // Uso directo de la asociación 'images'
          where: { isMain: true },
          required: false
        },
        {
          association: 'units',   // Uso directo de la asociación 'units'
          attributes: ['price'],
          limit: 1,
          required: false
        }
      ]
    });
    return products;
  } catch (error) {
    throw new Error('Error fetching products: ' + error.message);
  }
}
```

Figura 13: Método obtener productos

```
async create(product) {

  const { success, error, data } = validatePartialProduct(product);

  if (!success) {
    const errorMessages = error.errors.map(err => err.message);

    throw new ServiceError(`Validation error: ${errorMessages.join(', ')}`, 'VALIDATION_ERROR');
  }

  try {
    data.productID = uuidv4();
    data.releaseDate = new Date();
    return await this.ProductModel.create(data);
  } catch (error) {
    throw error;
  }
}
```

Figura 14: Método añadir producto

```
async getCartItems(userID) {
  try {

    const user = await this.UserModel.findByPk(userID);
    if (!user) {
      throw new ServiceError('User not found', 'NOT_FOUND');
    }


    const cartItems = await sequelize.query(`
      SELECT ci.*, pu.size, pu.price, p.name, pi.url as imageUrl
      FROM CartItem AS ci
      JOIN ProductUnit AS pu
      ON ci.productID = pu.productID AND ci.productNumber = pu.productNumber
      JOIN Product AS p
      ON pu.productID = p.productID
      LEFT JOIN ProductImage AS pi
      ON p.productID = pi.productID AND pi.isMain = 1
      WHERE ci.userID = :userID
    `, {
      replacements: { userID },
      type: QueryTypes.SELECT
    });
    return cartItems;
  } catch (error) {
    throw error;
  }
}
```

Figura 15: Método obtener productos del carrito

5.2.2.1. Autenticación y encriptación

En lo que respecta a la información sensible del usuario, tal como se especifica en los requisitos, se utiliza la librería `bcrypt` para cifrar la contraseña del usuario. Se implementó un servicio para gestionar la autenticación y registro de un usuario.

En la Figura 16 se puede ver el método que gestiona la creación de un usuario. Una vez se validan los datos del usuario, se cifra la contraseña con el método `hash()`, al cual se le pasa la contraseña y el número de *rounds* que va a aplicar el algoritmo de cifrado, se le asigna un `UUID` y se crea el usuario. Además, se autentica al usuario generando el *token* JWT correspondiente.

En la Figura 17 se puede ver el método de autenticación del usuario. Una vez se valida que el usuario existe, para mantener la seguridad de las contraseñas de los usuarios, se emplea el método `compare()`, que genera el *hash* de la contraseña introducida y lo compara con el *hash* de la contraseña almacenada. En caso de que coincidan, se genera un token JWT que se le devuelve al usuario junto con su información, excluyendo la contraseña. En caso contrario se lanza un error.

```
async register(user) {
  const { success, error, data } = validateUser(user);

  if (!success) {
    const errorMessages = error.errors.map(err => err.message);
    throw new ServiceError(`Validation error: ${errorMessages.join(', ')}`, 'VALIDATION_ERROR');
  }

  data.passwd = await bcrypt.hash(data.passwd, 10);
  data.userID = uuidv4();
  data.userRole = 'user';

  try {
    const newUser = await this.UserModel.create(data);

    // Generar token JWT
    const token = jwt.sign({ userId: newUser.userID, role: newUser.role }, SECRET_KEY, { expiresIn: '1h' });
    // Devolvemos el usuario sin la contraseña para evitar problemas de seguridad
    const {passwd, ...userWithoutPasswd} = newUser.dataValues;
    return { token, user: userWithoutPasswd };;
  } catch (error) {

    throw error;
  }
}
```

Figura 16: Método de registro

```
async login(email, passwd) {
  const user = await this.UserModel.findOne({ where: { email } });
  if (!user) {
    throw new ServiceError('User not found', 'NOT_FOUND');
  }

  const isPasswordValid = await bcrypt.compare(passwd, user.passwd);
  if (!isPasswordValid) {
    throw new ServiceError('Invalid password', 'INVALID_PASSWORD');
  }
  const userPlain = user.get({ plain: true });
  // No le enviamos la contraseña al cliente para evitar problemas de seguridad
  const {passwd: password, ...userWithoutPasswd} = userPlain;
  console.log(user);
  const token = jwt.sign({ userId: user.userID, userRole: user.userRole }, SECRET_KEY, { expiresIn: '3h' });
  return { token, user:  userWithoutPasswd };
}
```

Figura 17: Método de inicio de sesión

5.2.3. Controladores

Estos archivos se encargan de extraer los argumentos de la petición y llamar a los métodos del servicio correspondiente, gestionando la respuesta que se envía al cliente dependiendo de lo que haya ocurrido en el servicio.

La Figura 18 contiene un ejemplo de un método del controlador de productos que añade una unidad a un producto. Obtiene los datos de la unidad del cuerpo y el ID del producto de los parámetros de la ruta. Posteriormente, llama al servicio y si todo ha salido bien, devuelve un estado '201' que significa que se ha creado correctamente el recurso. En caso contrario, con el `next` pasamos el error al *middleware* de gestión errores.

```
async addUnit(req, res, next) {
  try {
    const productUnit = req.body;
    const { productID } = req.params;
    const newProductUnit = await this.productUnitService.addUnit(productID, productUnit);
    res.status(201).json(newProductUnit);
  } catch (error) {
    next(error);
  }
}
```

Figura 18: Token JWT

5.2.4. Routers

Encargados de asignar las rutas del servicio REST a cada método del controlador. En la Figura 19 se muestra el router encargado de las gestiones de los usuarios.

```
export const createUserRouter = (userModel, orderModel, addressModel, ProductUnitModel, cartModel, ProductModel, ProductoImageModel)


  const usersRouter = Router();
  const userController = new UserController(userModel, addressModel);
  const orderController = new OrderController(orderModel);
  const cartController = new CartController(cartModel, userModel, ProductUnitModel);


  usersRouter.post('',authMiddleware, roleMiddleware(['admin']), userController.create.bind(userController));
  usersRouter.get('',authMiddleware, roleMiddleware(['admin']), userController.getAll.bind(userController));
  usersRouter.get('/:userID',authMiddleware, authorizationMiddleware, userController.getById.bind(userController));
  usersRouter.put('/:userID', authMiddleware, authorizationMiddleware, userController.update.bind(userController));
  usersRouter.delete('/:userID', authMiddleware, roleMiddleware(['admin']), userController.delete.bind(userController));


  usersRouter.get('/:userID/addresses',authMiddleware, authorizationMiddleware, userController.getUserAddresses.bind(userController)
  usersRouter.post('/:userID/addresses',authMiddleware, authorizationMiddleware, userController.addUserAddress.bind(userController))
  usersRouter.put('/:userID/addresses/:addressID',authMiddleware, authorizationMiddleware, userController.updateUserAddress.bind(user
  usersRouter.get('/:userID/addresses/:addressID',authMiddleware, authorizationMiddleware, userController.getUserAddress.bind(userCo
  usersRouter.delete('/:userID/addresses/:addressID',authMiddleware, authorizationMiddleware, userController.deleteUserAddress.bind(

  usersRouter.get('/:userID/cart', authMiddleware, authorizationMiddleware, cartController.getCartItems.bind(cartController));
  usersRouter.post('/:userID/cart', authMiddleware, authorizationMiddleware, cartController.addCartItem.bind(cartController));
  usersRouter.put('/:userID/cart/:cartItemID', authMiddleware, authorizationMiddleware, cartController.updateCartItem.bind(cartContr
  usersRouter.delete('/:userID/cart/:cartItemID', authMiddleware, authorizationMiddleware, cartController.deleteCartItem.bind(cartCo


  usersRouter.get('/:userID/orders', authMiddleware, authorizationMiddleware, orderController.getUserOrders.bind(orderController));
  return usersRouter
}
```

Figura 19: Router de usuarios

Se pueden ver las diferentes rutas que gestionan los usuarios, las direcciones y el carrito. En cada ruta, se especifica el método HTTP (POST, GET, DELETE,

etc.), el `authmiddleware`, que se encarga de permitir utilizar esta ruta solo a los usuarios autenticados. Por otro lado, el `roleMiddleware` encargado de asegurarse de que solo accedan ciertos roles. Por último, le indicamos a cada ruta que método del controlador tiene que llamar al recibir una petición.

5.2.5. Middlewares

En este apartado, se muestran los distintos *middlewares* que se han implementado en el servicio REST y se detalla su función.

5.2.5.1. AuthMiddleware

En la Figura 20 se muestra el primer *middleware* que se encarga de la autenticación. Se encarga de extraer el token de la petición, verifica que es válido y extrae del token tanto el usuario como el rol de este. Una vez extraídos, se añaden al objeto `req` que contiene la petición y con `next()` pasamos la petición al siguiente *middleware*.

Una vez extraemos el `authHeader`, para obtener el token tengo que dividir la cadena que suele comenzar con un 'Bearer miToken' y de esta manera obtenemos la segunda posición del *array* que contiene el token del usuario.

Al asignar este *middleware* a una ruta, conseguimos que el usuario solo pueda realizar peticiones en el caso de que en el encabezado de autorización incluya el token. Si no está autenticado se devuelve el error 'Unauthorized'.

```
export const authMiddleware = (req, res, next) => {
  const authHeader = req.headers.authorization;

  if (!authHeader) {
    return res.status(401).json({ error: 'No token provided', code: 'UNAUTHORIZED'
  }

  const token = authHeader.split(' ')[1];

  try {
    const decoded = jwt.verify(token, SECRET_KEY);
    req.userId = decoded.userId;
    req.userRole = decoded.userRole;

    next();
  } catch (error) {
    console.log(error);
    return res.status(401).json({ error: 'Invalid token', code: 'INVALID_TOKEN' })
  }
};
```

Figura 20: Control de Autenticación

5.2.5.2. AuthorizationMiddleware

En este apartado se muestra el *middleware* que se encarga de controlar que el usuario autenticado sea el mismo que el de la ruta.

Como podemos observar en la Figura 21 se extrae el ID del usuario tanto del `req`, que ha sido extraído de la cabecera en el *middleware* anterior, como el de la ruta a la que se hace la petición. Estos IDs, se comparan y si coinciden se pasa con el siguiente *middleware*, en caso contrario se lanza un error '403 Forbidden'.

```
export const authorizationMiddleware = (req, res, next) => {
  const userIdFromToken = req.userId;
  const userIdFromParams = req.params.userID;

  if (userIdFromToken !== userIdFromParams) {
    return res.status(403).json({ error: 'Forbidden: You do not have access to this resource', code: 'FORBIDDEN' });
  }

  next();
};
```

Figura 21: Control de autorización

5.2.5.3. RoleMiddleware

Por otro lado, en el siguiente *middleware*, se gestiona que el usuario que está autenticado tenga el rol requerido. En caso de que no se cumpla esta condición, se lanza un error '403 Forbidden'. En caso afirmativo, se pasa la petición al controlador correspondiente para que se encargue de ella. En la Figura 22 se observa la implementación de esta parte del código.

```
export const roleMiddleware = (roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.userRole)) {
      return res.status(403).json({ error: 'Forbidden: You do not have access to this resource', code: 'UNAUTHORIZED' });
    }
    next();
  };
};
```

Figura 22: RoleMiddleware

5.2.5.4. CORS

Este *middleware* sirve para no tener problemas con el *Cross Origin Resource Sharing*, que controla que dominios pueden hacer solicitudes al servicio REST. En este caso, se permite el puerto local donde se despliega el *frontend*. En la Figura 23 podemos observar la implementación. Utilizando la función `cors()` la pasamos la función que verifica los orígenes de la petición. En caso de que la ruta de origen de la petición se encuentre entre los orígenes aceptados, definidos en una constante, se llama a `callback(null, true)` para indicar que no hubo errores y que se puede

proceder con la solicitud. En caso de que no haya un origen establecido, es decir, solicitudes internas o en el modo de desarrollo, también se procede con la solicitud. En caso contrario, se lanza un error especificando el mensaje de error.

```
export const corsMiddleware = ({ acceptedOrigins = ACCEPTED_ORIGINS } = {}) => cors({
  origin: (origin, callback) => {
    if (acceptedOrigins.includes(origin)) {
      return callback(null, true)
    }

    if (!origin) {
      return callback(null, true)
    }

    return callback(new Error('Not allowed by CORS'))
  }
})
```

Figura 23: CORS

5.2.5.5. Gestor de errores

Por último, se ha implementado una capa intermedia que gestiona los errores. En el Anexo 9.5 el código que se encarga de esta gestión. Los controladores pasan los errores que reciben de los servicios a esta capa, la cual, mediante los códigos de error, devuelve mensajes predeterminados para cada caso. Con esto logramos evitar la duplicidad de código, centralizando el manejo de errores. Se encarga de devolver respuestas con diferentes estados en función del código de error que se haya recibido.

5.2.6. Archivo principal

El archivo principal del servicio REST es el `app.js`. En este archivo, primeramente se inicia la conexión con la base de datos y se sincronizan los modelos. Posteriormente, utilizando la librería `Express`, se crea una instancia que sirve como núcleo del servidor, y a partir de este se habilitan una serie de *middlewares*. A continuación, se definen las rutas correspondientes para cada uno de los *Routers*, a los que se les pasan los modelos correspondientes que van a utilizar. Abstrayendo el uso de los modelos a este archivo, conseguimos hacer mucho más mantenible el código, ya que se hace mucho más sencillo el cambio de modelos. Por último, añadimos el gestor de errores y desplegamos la aplicación en el puerto correspondiente.

Se adjunta en la Figura 24 el código implementado. Como se puede observar se ha añadido la ruta `/api/geonames/` que se utiliza para obtener información sobre los países, provincias y ciudades para ofrecer las opciones a los usuarios en el formulario de direcciones.

```
db.sequelize.authenticate().then(() => {
  console.log('Connection has been established successfully.');
}).catch((error) => {
  console.error('Unable to connect to the database:', error);
});

// Sincronizar los modelos
db.sequelize.sync().then(() => {
  console.log('Models synchronized successfully.');
}).catch((error) => {
  console.error('Error synchronizing models:', error);
});

const app = express()

app.use(morgan('combined', {
  stream: {
    write: message => logger.info(message.trim())
  }
}));

app.use(express.json());
app.use(corsMiddleware())
app.disable('x-powered-by')
app.use('/api/auth', createAuthRouter( db.User));
app.use('/api/products', createProductRouter( db.Product, db.ProductUnit, db.ProductImage, db.Category));
app.use('/api/categories', createCategoryRouter( db.Category));
app.use('/api/teams', createTeamRouter( db.Team));
app.use('/api/users', createUserRouter( db.User, db.Order, db.Address, db.ProductUnit, db.CartItem, db.Product, db.ProductImage));
app.use('/api/orders', createOrderRouter( db.Order, db.OrderProductUnit, db.OrderHistory));
app.use('/api/geonames', createGeoNamesRouter());
app.use('/api/discounts', createDiscountRouter(db.Discount))
app.use(errorHandler);

const PORT = process.env.PORT ?? 1234

app.listen(PORT, () => {
  console.log(`server listening on port http://localhost:${PORT}`)
})
```

Figura 24: app.js

5.3. Capa de presentación

Una vez finalizado el servicio REST, siguiendo la metodología escogida se fueron añadiendo las distintas vistas y funcionalidades a la interfaz del usuario. En este apartado se comentarán las decisiones tomadas acerca del código de la capa de presentación.

Se ha implementado utilizando TypeScript, que funciona como JavaScript pero utilizando tipos en las variables. En la figura 25 se muestra la estructura que está dividida en varias carpetas que se explicarán con más detalle a continuación.

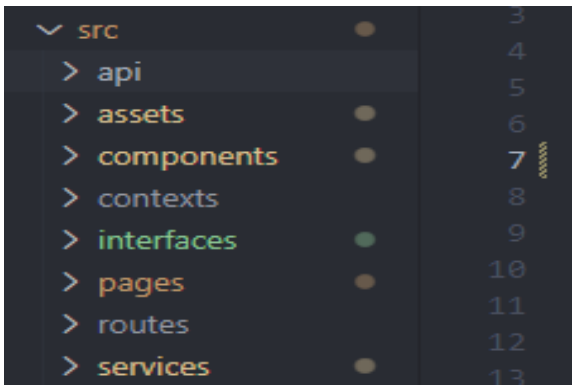

Figura 25: Estructura de carpetas

Empezamos comentando por separado las dos primeras carpetas debido a su simplicidad. La primera carpeta contiene el archivo de configuración de `Axios` y se encarga de crear una conexión con la URL del servicio REST. Esta conexión se utiliza en los servicios de la capa de presentación para interactuar con el *backend*.

Por otro lado, la segunda carpeta que contiene algunos archivos de estilo utilizados en los componentes comunes, como por ejemplo el Header. En la mayoría de componentes, se han utilizado otros componentes más pequeños de la librería de `Material UI`, lo que me ha facilitado el trabajo, pero a la hora de personalizar el estilo he tenido que sobreescribir el estilo que venía por defecto, ya que me generaba conflictos.

5.3.1. Componentes

Esta carpeta contiene los componentes reutilizables de la aplicación, se ha dividido en componentes comunes, como el Header o el Footer, y en componentes específicos de cada página.

Los componentes encapsulan la parte visual de la interfaz que ve el usuario (HTML), el estilo (CSS) y la lógica de negocio (TypeScript). Los componentes pueden recibir `props`, que son parámetros que pueden utilizar en el componente, esto es útil para poder pasar funciones que permiten a componentes hijo, modificar el estado de componentes padre.

En la Figura 26, observamos la lógica de negocio del componente que muestra el resumen de compra del carrito de un usuario. Primeramente, se le pasa el parámetro `isAddress` que se usa en este caso para cambiar el estilo y contenido del sumario de compra, dependiendo de en qué página esté siendo utilizado.

Posteriormente, se definen una serie de constantes utilizando `useState()`. Es un `hook`, que nos permite introducir un estado interno a los componentes funcionales. Cuando se llama a esta función, se retorna un *array* con el estado actual, y la función que actualiza el estado. En este componente, se utilizan para guardar el código de descuento introducido por el cliente, cambiar el mensaje de error en caso de que haya algún fallo o actualizar el descuento que se está aplicando.

Por otro lado, se ha implementado la lógica del botón de aplicar descuento. Lo que hace esta función es inicializar los mensajes de confirmación y de error para que no se muestre nada, validar el código de descuento a través del servicio y en el caso de que el código sea válido, se actualiza el estado del código y se muestra el mensaje de confirmación. En caso contrario, se actualiza el estado del error y se muestra el mensaje.

```
interface PriceSummaryProps {
  isAddress: boolean;
}

const PriceSummary: React.FC<PriceSummaryProps> = ({ isAddress }) => {
  const [coupon, setCoupon] = useState("");
  const [discount, setDiscount] = useState<Discount | null>(null);
  const { getUser } = useAuth();
  const [message, setMessage] = useState<string>("");
  const [open, setOpen] = useState<boolean>(false);
  const { calculateSubtotal, commitCartChanges } = useCart();
  const [error, setError] = useState<string>("");

  const handleApplyCoupon = async () => {
    try {
      setMessage("");
      setOpen(false);
      setError("");
      const result = await DiscountService.validateCoupon(coupon);
      if (!result) {
        setError("Cupón inválido");
        return;
      }

      setDiscount(result);
      setCoupon("");
      setMessage("Cupón aplicado correctamente");
      setOpen(true);
    } catch (error: any) {
      setError(error.message);
    }
  };

  const calculateTotal = () => {
    const subtotal = calculateSubtotal();
    const discountAmount = discount ? discount.discount : 0;
    return (subtotal - discountAmount).toFixed(2);
  };
```

Figura 26: Componente resumen de compra

En la parte final podemos ver otra función que calcula el total del importe del carrito, aplicando el descuento en caso de que se haya introducido.

En lo que respecta a la parte visual del componente, como podemos observar en la Figura 27, el parámetro o `prop` que se le pasa al componente, afecta directamente tanto a lo que se ve como al estilo. Esto es gracias a que podemos introducir TypeScript directamente en el HTML.

Para configurar el estilo de los componentes de `Material UI` se ha realizado de tres maneras, creando un fichero de estilo `.css` y añadiendo las clases a cada elemento, a través de `props` y, por último, usando `sx`, que permite inyectar directamente el CSS.

Se han intentado crear el mayor número de componentes posible que puedan ser reutilizados con el objetivo de evitar la duplicidad de código.

```
return (
  <div className={`price-summary-container ${isAddress ? "address" : ""}`}>
    <Typography variant="h5" color={"black"} className="price-summary-title">
      Resumen de la compra
    </Typography>
    {isAddress ? (
      <></>
    ) : (
      <CouponMenu
        setCoupon={setCoupon}
        handleApplyCoupon={handleApplyCoupon}
        error={error}
        coupon={coupon}
        message={message}
        open={open}
      />
    )}
    <Box flex={1} />

    <Box mt={2} display={"flex"} flexDirection={"column"}>
      <Grid
        container
        className="price-summary-text"
        justifyContent="space-between"
      >
        <Typography>Subtotal</Typography>
        <Typography>{calculateSubtotal().toFixed(2)}€</Typography>
      </Grid>
      <Grid
        className="price-summary-text"
        container
        justifyContent="space-between"
      >
```

Figura 27: Componente resumen de compra HTML

5.3.2. Contextos

Para implementar esta aplicación, se han desarrollado dos ficheros que gestionan el contexto de la aplicación, uno se encarga de la gestión del carrito a lo largo de la sesión y el otro se encarga de la autenticación del usuario.

En la Figura 28 se muestra el ejemplo del contexto de autenticación del usuario. Proporciona al resto de páginas y de componentes de la aplicación un mecanismo para acceder y consultar si el usuario está autenticado.

Podemos observar, como se está utilizando el `useEffect`, que es una función que se ejecuta dependiendo de los valores de las dependencias que se encuentran en el *array*, en este caso al estar vacío, se ejecuta una única vez. Al renderizarse el contexto por primera vez, comprueba si hay un token, en caso afirmativo asigna el usuario actual a la sesión.

```tsx
interface AuthContextType {
  isAuthenticated: boolean;
  login: (email: string, password: string) => Promise<void>;
  logout: () => void;
  userID: string;
  getUser: () => string;
}

const AuthContext = createContext<AuthContextType | undefined>(undefined);

export const AuthProvider: React.FC<{ children: React.ReactNode }> = ({
  children,
}) => {
  const [isAuthenticated, setIsAuthenticated] = useState<boolean>(false);
  const [userID, setUserID] = useState<string>("");

  useEffect(() => {
    const token = AuthService.getToken();
    setIsAuthenticated(!!token);

    const storedUserID = AuthService.getUser();
    if (storedUserID) {
      setUserID(storedUserID);
    }
  }, []);

  const login = async (email: string, password: string) => {
    try {
      await AuthService.login(email, password);
      setIsAuthenticated(true);
      const userID = AuthService.getUser();

      if (userID) {
        setUserID(userID);
      }
    } catch (error) {
      throw error;
    }
  };
```

Figura 28: Contexto de autenticación

5.3.3. Rutas

De una forma similar a como se hace en el servicio REST, como se observa en la Figura 29, se asigna a cada ruta de la aplicación una página que se tiene que renderizar en el HTML.

```tsx
const AppRoutes: React.FC = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<LandingPage />} />
        <Route path="/login" element={<LoginPage />} />
        <Route path="/register" element={<RegisterPage />} />
        <Route path="products/:productID" element={<DetailsPage />} />
        <Route path="/checkout" element={<CheckoutPage />} />
        <Route path="/checkout/address" element={<OrderAddressPage />} />
        <Route path="/checkout/address/new" element={<AddAddressPage />} />
        <Route path="/address/edit/:addressID" element={<EditAddressPage />} />
      </Routes>
    </Router>
  );
};

export default AppRoutes;
```

Figura 29: Gestor de rutas

5.3.4. Servicios

Los servicios utilizan la configuración de `Axios` para comunicarse con la API. En la Figura 30 podemos observar el método de autenticación de un usuario. Se comunica con el *backend*, y si todo ha ido bien, se utilizan las Cookies para almacenar el token de sesión y el usuario.

```
login: async (email: string, password: string) => {
  try {
    const response = await api.post(`/auth/login`, { email, passwd: password });
    const { token, user } = response.data;
    const date = new Date();
    date.setTime(date.getTime() + (60 * 60 * 1000)); // 2 horas en milisegundos

    Cookies.set('token', token, { expires: date });
    Cookies.set('user', user.userID, { expires: date });
    return { token, user};
  } catch (error: any) {
    if (axios.isAxiosError(error)) {
      if (error.response?.status === 404) {
        throw new Error('No existe el usuario introducido');
      } else if (error.response?.status === 403) {
        throw new Error('Contraseña incorrecta');
      } else {
        throw new Error('Error en la conexión con el servidor');
      }
    }

    throw new Error('Error en el inicio de sesión' + error.message);
  }
},
```

Figura 30: Servicio de autenticación

5.3.5. Páginas

A continuación, se muestran algunas de las páginas que se han implementado para satisfacer los requisitos de la primera fase del desarrollo.

En la Figura 31 se puede ver el resultado del componente del carrito de la compra, en la Figura 32 la página del detalle de un producto y en la Figura 33 la página de confirmación de la compra previa al pago.

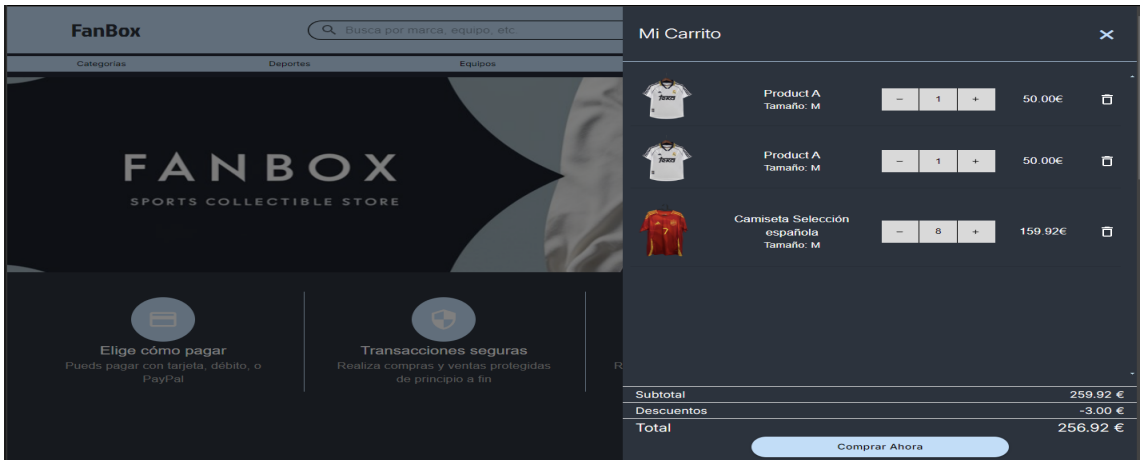

Figura 31: Carrito de la compra

Figura 32: Página detalle de producto

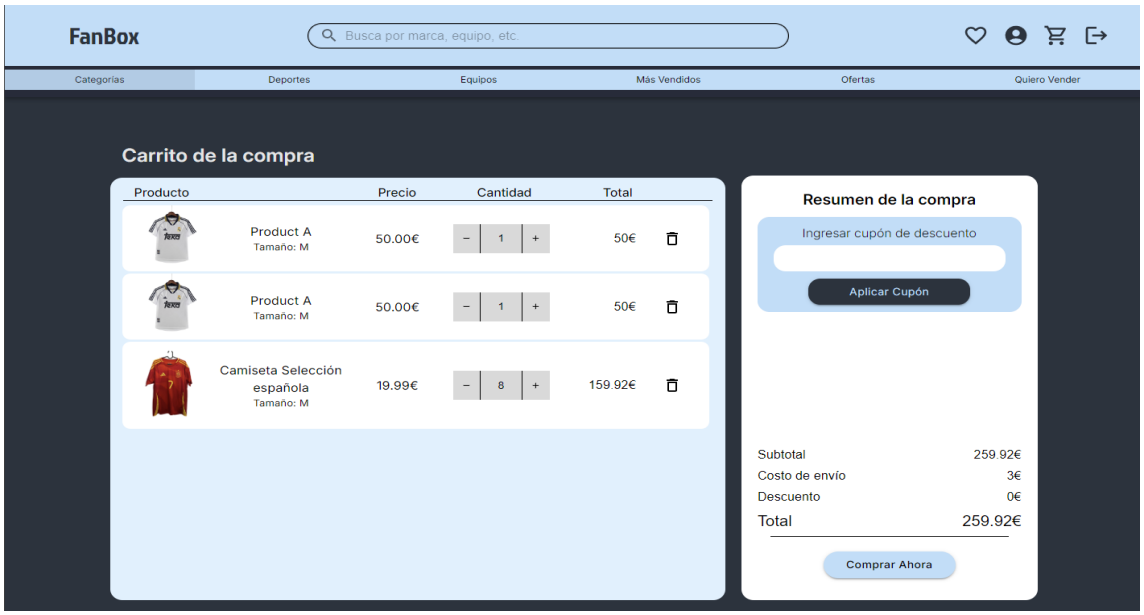

Figura 33: Página confirmación de compra

Como se puede observar, se han seguido firmemente los prototipos diseñados, empleando gran parte del tiempo en modificar la estética de la página.

5.3.6. Otras funcionalidades

Al no ser posible adjuntar una imagen de todas las páginas de la aplicación, voy a explicar otras decisiones fundamentales que se han tomado a la hora de desarrollar esta capa de presentación.

Para evitar la duplicidad del código y tener que añadir en cada página tanto el Header como el Footer, se ha implementado un componente Layout que recibe como `prop` el componente que se va a mostrar y lo envuelve entre los dos anteriores.

Por otro lado, para gestionar la persistencia de las operaciones del carrito, he añadido un estado que sirve como caché para almacenar los cambios que haya hecho el usuario, y en caso de que se salga del carrito o de la sesión, se realicen todas esas peticiones al servicio REST. De esta manera evitamos que haya una petición cada vez que aumenta, disminuye la cantidad o elimina un producto.

Además, se ha utilizado la librería `Formik` para facilitar la creación de los formularios de autenticación, registro e inserción de una nueva dirección. Por otro lado, se han añadido transiciones en la navegación entre páginas para que haya un efecto suave, mejorando la experiencia del usuario.

En la Figura 34 se observa la página del formulario para editar la dirección del usuario. Se utiliza la librería `Yup` para validar los valores introducidos por el usuario, y en este caso, para controlar que los valores de los países, provincias y ciudades sea válido, se hace una petición a través de nuestro servicio REST a otra API que contiene información de los países, provincias y ciudades del mundo. De esta manera, el usuario se limita a seleccionar las opciones que se le muestran.

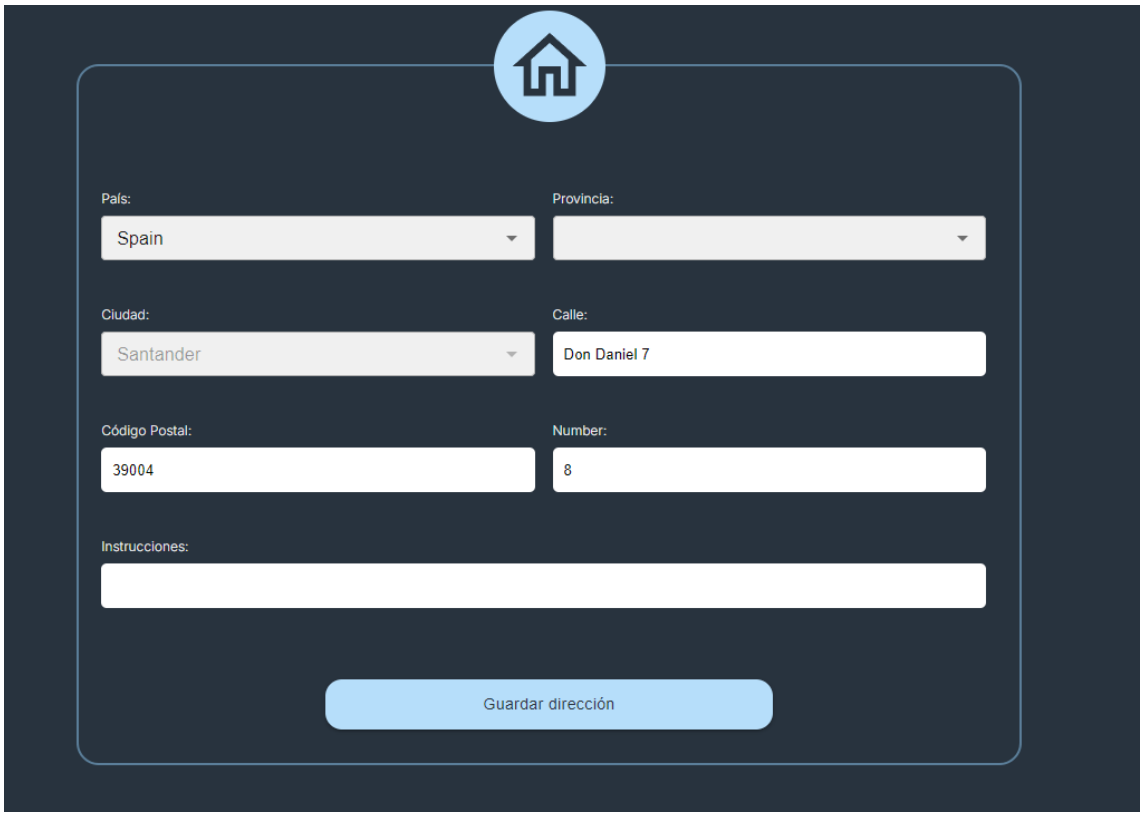

Figura 34: Formulario editar dirección

6. Pruebas

En este apartado se explican las pruebas que se han realizado en el desarrollo de la aplicación. Debido a la escasa lógica de negocio de la aplicación y a la falta de tiempo, la mayoría de pruebas de las funcionalidades añadidas se han realizado utilizando la interfaz como prueba de aceptación.

6.1. Servicio REST

Para realizar las pruebas correspondientes de los *endpoints* de la aplicación, se ha utilizado la herramienta de Postman, que nos permite crear colecciones de peticiones, de manera que podamos estructurar y ejecutar de manera paralela las pruebas para cada *endpoint*.

Se han realizado las pruebas para todos los *endpoints* que se utilizan en la aplicación, además las pruebas de integración se iban realizando al finalizar cada componente. Como podemos ver en la Figura 35, se ha realizado la prueba de la autenticación del usuario que recibe correctamente el token y el usuario sin la contraseña. En la Figura 36 se puede ver como se recuperan los productos del carrito del usuario, siendo necesario introducir el token de autenticación, asegurándonos que los *middlewares* funcionan correctamente. Por último, en la Figura 37, se observa como se obtiene de cada producto sus imágenes y el precio.

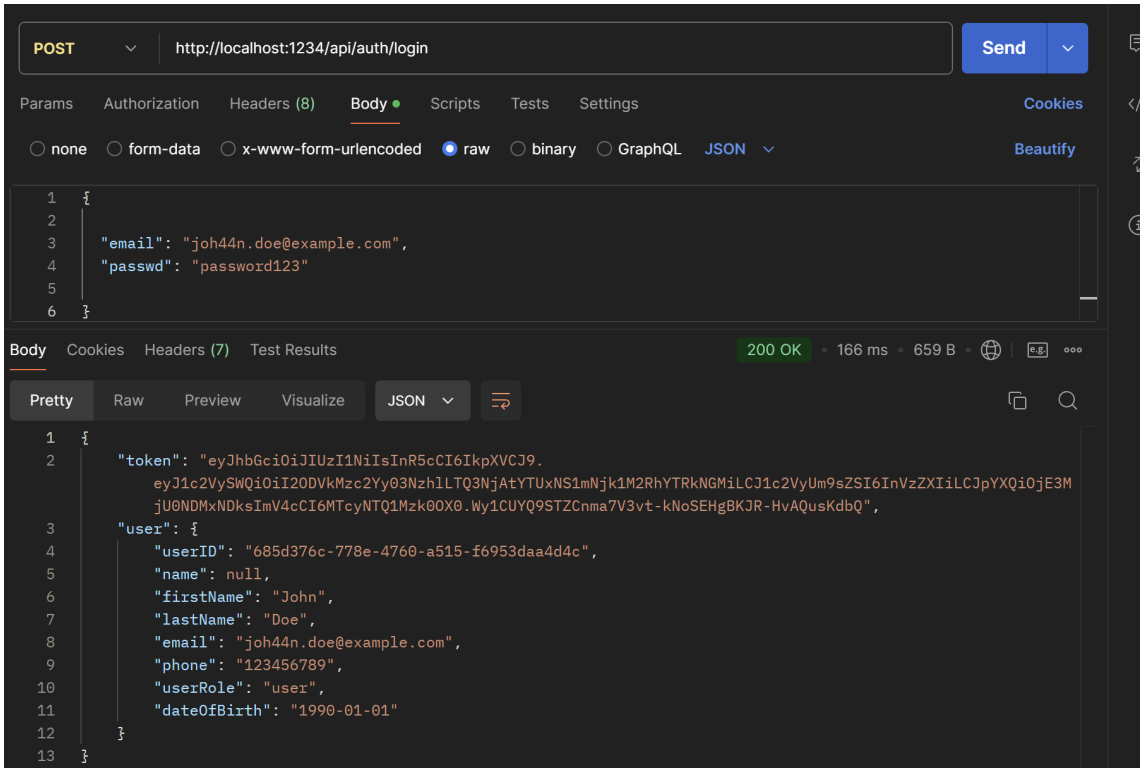

Figura 35: Prueba de login

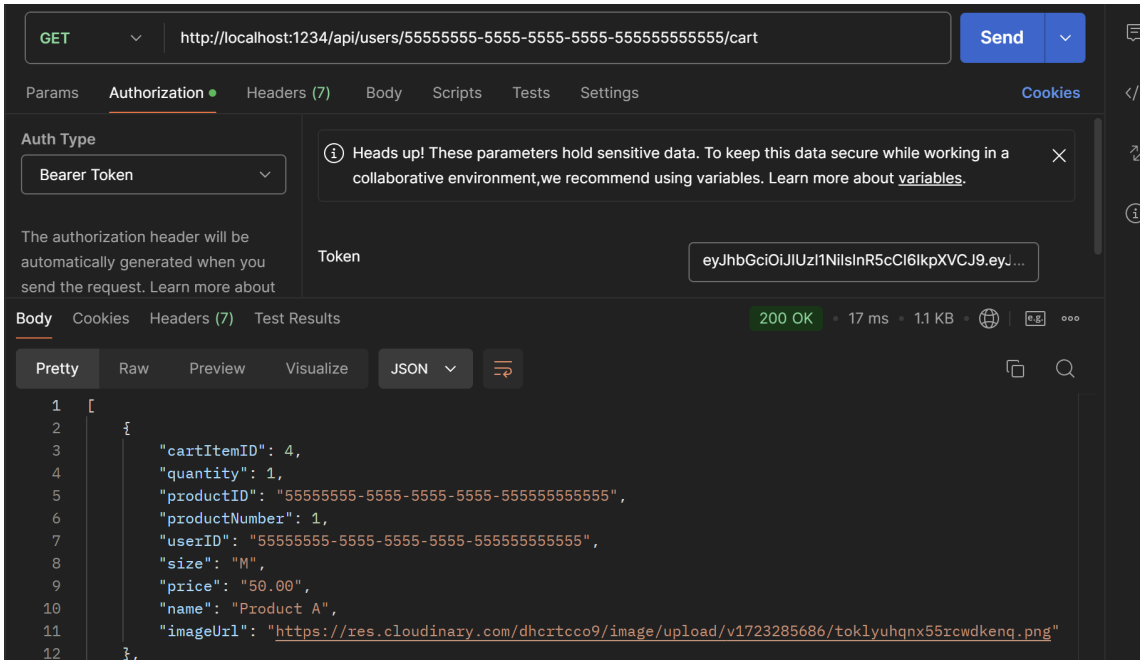

Figura 36: Obtener carrito del usuario

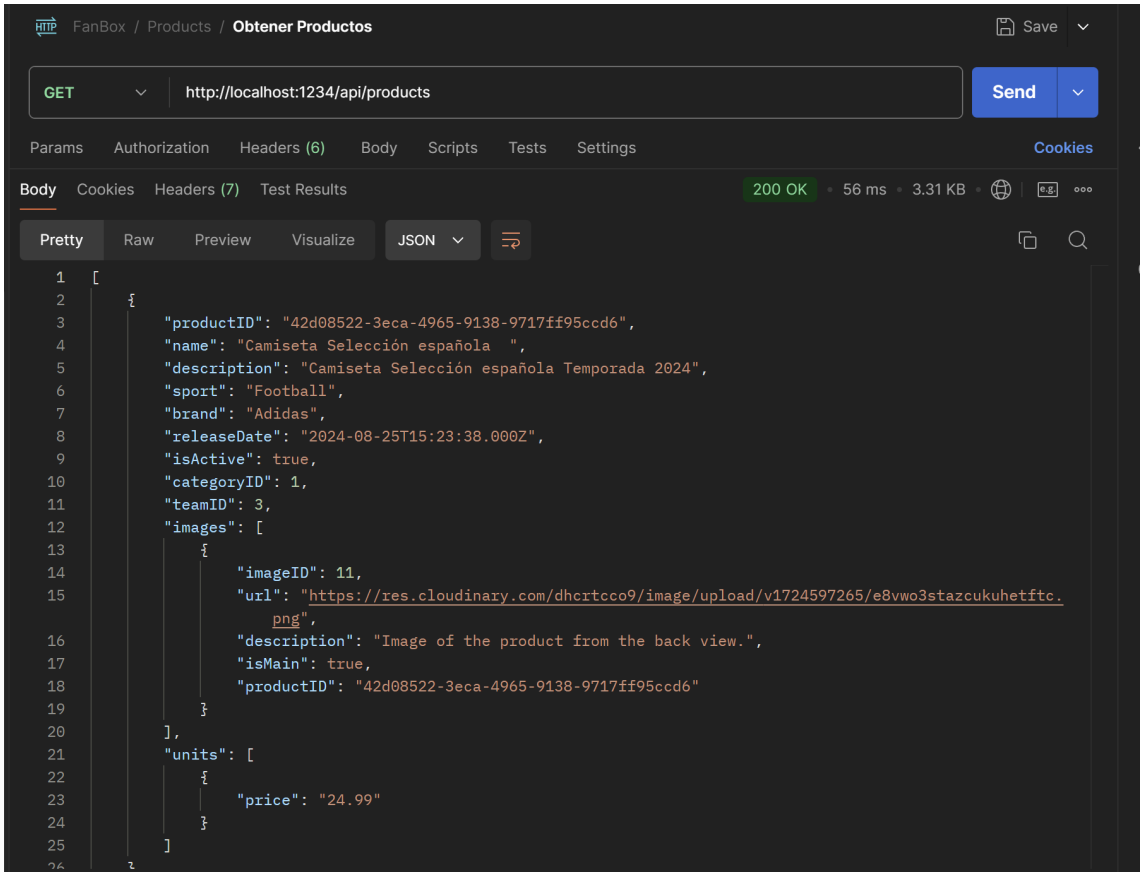

Figura 37: Obtener productos

7. Conclusiones

El objetivo principal de este proyecto era adquirir la mayor experiencia posible en las tecnologías más utilizadas a día de hoy en el desarrollo web. Se han utilizado numerosas herramientas que no se conocían como React, Node.js, Express, Sequelize y componentes de Material UI entre otras. Estas herramientas y tecnologías me han permitido realizar una aplicación funcional de una manera eficiente y en el periodo establecido en la planificación.

Se han intentado desarrollar todos los requisitos funcionales de la aplicación, pero a medida que avanzaba el tiempo, se tuvo que dejar de lado el desarrollo completo de la aplicación y centrarse tanto en las pruebas como en la memoria. Siguiendo la metodología detallada en la primera parte de esta memoria, se han ido realizando los incrementos en la parte de la capa de presentación de manera que han quedado finalizado los casos de uso de ver productos, acceder al detalle del producto, las operaciones del carrito de la compra, las operaciones de las direcciones del usuario, inicio de sesión, registro y por último aplicar descuento. A pesar de que no se hayan implementado en el *frontend* todas las funcionalidades, la API está completamente terminada y se pueden realizar los casos de uso que faltan utilizando Postman.

Debido a la falta de tiempo, no se han podido realizar las pruebas de calidad del código utilizando SonarQube. Es una parte fundamental en el desarrollo software proporcionar un código de calidad, por lo tanto, estas pruebas se realizarán más adelante, una vez completada la aplicación.

Aunque no se hayan logrado terminar todos los casos de uso planteados, estoy contento con las funcionalidades que se han logrado desarrollar, ya que la aplicación que se plantea es bastante amplia y en especial estoy orgulloso de la estética de la aplicación que ha supuesto mucho trabajo. He conseguido obtener experiencia autenticando usuarios, gestionando los *tokens* de sesión y el contexto del carrito de la compra, habilidades que me van a servir mucho a lo largo de mi carrera como desarrollador.

Como comentario personal, quiero decir que voy a seguir desarrollando esta aplicación, completando todos los casos de uso y añadiendo otros casos de uso que den valor a esta aplicación de una tienda ficticia, pero que en un futuro pueda convertirse en una aplicación útil para alguna tienda real.

A continuación, se adjuntan los enlaces del código, prototipos y diagramas implementados:

Código: https://github.com/alvarezcondeluis/FanBox

Diagrama draw.io:: https://app.diagrams.net/

Prototipos: https://www.figma.com/design/

8. Bibliografía

[1] Node js. Recuperado el 26 de agosto de 2024, de https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

[2] IBM. (n.d.). REST APIs. Recuperado el 26 de agosto de 2024, de https://www.ibm.com/es-es/topics/rest-apis

[3] Express.js. (n.d.). *Express.js Documentation*. Recuperado el 26 de agosto de 2024, de https://expressjs.com/

[4] Sequelize. (n.d.). *Sequelize Documentation*. Recuperado el 26 de agosto de 2024, de https://sequelize.org/

[5] Zod (n.d). *Zod Documentation*. Recuperado el 6 de septiembre de 2024, de https://zod.dev/

[6] Santillán, Luis Alberto Casillas, Marc Gibert Ginestà, and Óscar Pérez Mora. "Bases de datos en MySQL.Üniversitat oberta de Catalunya (2014).

[7] Jest. (n.d.). *Jest Documentation*. Recuperado el 27 de agosto de 2024, de https://jestjs.io/

[8] Figma. (n.d.). *Figma for UI/UX Design*. Recuperado el 27 de agosto de 2024, de https://www.figma.com/

[9] Meta Platforms, Inc. (n.d.). *React: A JavaScript library for building user interfaces*. Recuperado el 27 de agosto de 2024, de https://react.dev

[10] Chacon, S. and Straub, B. (n.d.). *Pro Git*. Recuperado el 27 de agosto de 2024, de https://git-scm.com/

[11] Microsoft. (n.d.). *Visual Studio Code Documentation*. Recuperado el 27 de agosto de 2024, dehttps://code.visualstudio.com/docs

[12] Postman. (n.d.). *Postman API Platform*. Recuperado el 27 de agosto de 2024, de https://www.postman.com/

[13] Sommerville, Ian.: 'Ingeniería del Software'. 7ª Edición. PEARSON EDUCACIÓN, S.A. Madrid 2005. [Som05]

9. Anexo

9.1. Especificación de los casos de uso

Tabla 5: Ver Productos

Caso de Uso	Ver Productos
ID	CU01
Descripción	El usuario puede ver una lista con los productos disponibles en la tienda incluyendo la imagen principal y el precio.
Actores Principales	Cliente No Autenticado, Cliente Autenticado
Precondiciones	▪ El usuario debe estar en la página principal o en una categoría específica de productos. ▪ La aplicación tiene que tener una conexión estable con la base de datos.
Flujo Principal	1. El usuario accede a la página principal. 2. El sistema obtiene los productos de la base de datos. 3. El sistema muestra la lista de productos, incluyendo la imagen y el precio del producto.
Flujos Alternativos	**2.1 No hay productos disponibles:** 2.1a El sistema obtiene una lista vacía de la base de datos. 2.1b El sistema muestra un mensaje informativo indicando que no hay productos disponibles. **2.2 Error en la conexión con la base de datos** 2.2a El sistema no recibe respuesta de la base de datos. 2.2b El sistema muestra un mensaje informativo indicando un error en la conexión.

Tabla 6: Ver Carro de la compra

Caso de Uso	Ver Carro de la Compra
ID	CU05

Descripción	El usuario puede ver acceder al carro de la compra y ver los productos que haya añadido, incluyendo el nombre, cantidad y precio total de cada producto.
Actores Principales	Cliente Autenticado
Precondiciones	▪ El usuario debe estar autenticado en el sistema. ▪ La aplicación tiene que tener una conexión estable con la base de datos.
Flujo Principal	1. El usuario accede a su carrito de compras. 2. El sistema obtiene los productos del carrito de la base de datos. 3. El sistema muestra una lista de los productos en el carrito, con el nombre, cantidad y precio total de cada producto.
Flujos Alternativos	**2.1 No hay productos en el carrito:** 2.1a El sistema no encuentra productos en el carrito. 2.1b El sistema muestra un mensaje indicando que el carrito está vacío. **2.2 Error en la conexión con la base de datos:** 2.2a El sistema no recibe respuesta de la base de datos. 2.2b El sistema muestra un mensaje informativo indicando un error en la conexión.

Tabla 7: Eliminar Producto del Carrito

Caso de Uso	Eliminar Producto del Carrito
ID	CU06
Descripción	El usuario puede eliminar un producto concreto que haya en el carro de la compra.
Actores Principales	Cliente Autenticado
Precondiciones	▪ El usuario debe estar autenticado en el sistema. ▪ El usuario debe tener al menos un producto en su carrito. ▪ La aplicación debe estar conectada correctamente a la base de datos.
Flujo Principal	1. El usuario accede a su carrito de compras. 2. El usuario selecciona el producto que desea eliminar. 3. El sistema elimina el producto del carrito y actualiza el estado del carrito. 4. El sistema muestra la lista actualizada de productos en el carrito.
Flujos Alternativos	**2.1 Error en la conexión con la base de datos:** 2.1a El sistema no puede eliminar el producto debido a un error de conexión. 2.1b El sistema muestra un mensaje indicando que ocurrió un error y que el producto no pudo ser eliminado.

Tabla 8: Añadir Dirección de Envío

Caso de Uso	Añadir Dirección de Envío
ID	CU07
Descripción	El usuario puede añadir una nueva dirección de envío en el proceso de compra de un producto.
Actores Principales	Cliente Autenticado
Precondiciones	▪ El sistema debe estar correctamente conectado a la base de datos. ▪ El usuario debe de estar autenticado.

Flujo Principal	1. El usuario pulsa Realizar Compra. 2. El sistema recupera las direcciones almacenadas y se las muestra al usuario. 3. El usuario pulsa en Añadir Dirección. 4. El sistema muestra un formulario para introducir la información de la dirección que se va a añadir. 5. El usuario introduce los datos y pulsa Guardar Dirección. 6. El sistema valida los datos y almacena la nueva dirección. 7. El sistema muestra un mensaje de confirmación y redirige al usuario a la página de las direcciones.
Flujos Alternativos	**6.1 El sistema detecta un error en los datos de la dirección:** 6.1a El sistema detecta un error al validar los datos introducidos. 6.1b El sistema muestra un mensaje de error al usuario indicando el campo que ha fallado. **6.2 No hay conexión con la base de datos:** 6.2a El sistema no recibe respuesta de la base de datos. 6.2b El sistema muestra un mensaje indicando que no hay conexión con la base de datos.

Tabla 9: Eliminar Dirección de Envío

Caso de Uso	Eliminar Dirección de Envío
ID	CU07
Descripción	El usuario puede eliminar una dirección de envío existente en el proceso de compra de un producto.
Actores Principales	Cliente Autenticado

Precondiciones	▪ El sistema debe de estar correctamente conectado a la base de datos. ▪ El usuario debe de estar autenticado. ▪ El usuario debe de tener almacenada al menos una dirección
Flujo Principal	1. El usuario pulsa Realizar Compra. 2. El sistema recupera las direcciones almacenadas y se las muestra al usuario. 3. El usuario pulsa en Eliminar Dirección. 4. El sistema elimina la dirección de la base de datos. 5. El sistema muestra un mensaje confirmando la eliminación de la dirección.
Flujos Alternativos	**2.1 No hay direcciones almacenadas:** 2.1a El sistema muestra un mensaje indicando que el usuario no tiene direcciones almacenadas. 2.1b El usuario añade una dirección y vuelve al paso 1. **2.2 No hay conexión con la base de datos:** 2.2a El sistema no recibe respuesta de la base de datos. 2.2b El sistema muestra un mensaje indicando que no hay conexión con la base de datos. **4.1 No hay conexión con la base de datos:** 4.1a El sistema intenta eliminar la dirección pero la base de datos no responde 4.1b El sistema muestra un mensaje de error al usuario indicando que no hay conexión con la base de datos.

Tabla 10: Editar Dirección de Envío

Caso de Uso	Editar Dirección de Envío
ID	CU07
Descripción	El usuario puede editar una dirección de envío existente en el proceso de compra de un producto.

Actores Principales	Cliente Autenticado
Precondiciones	▪ El sistema debe de estar correctamente conectado a la base de datos. ▪ El usuario debe de estar autenticado. ▪ El usuario debe de tener almacenada al menos una dirección
Flujo Principal	1. El usuario pulsa Realizar Compra. 2. El sistema recupera las direcciones almacenadas y se las muestra al usuario. 3. El usuario pulsa en Editar Dirección. 4. El sistema muestra un formulario con los datos de la dirección seleccionada. 5. El usuario edita los campos que se desea modificar y selecciona Guardar Dirección. 6. El sistema valida los datos y actualiza la dirección en la base de datos. 7. El sistema muestra un mensaje de confirmación y redirige al usuario a la página de las direcciones.

Flujos Alternativos	**2.1 No hay direcciones almacenadas:** 2.1a El sistema muestra un mensaje indicando que el usuario no tiene direcciones almacenadas. 2.1b El usuario añade una dirección y vuelve al paso 1. **2.2 No hay conexión con la base de datos:** 2.2a El sistema no recibe respuesta de la base de datos. 2.2b El sistema muestra un mensaje indicando que no hay conexión con la base de datos. **6.1 El sistema detecta un error en los datos de la dirección:** 4.1a El sistema detecta un error al validar los datos introducidos. 4.1b El sistema muestra un mensaje de error al usuario indicando el campo que ha fallado.

Tabla 11: Ver Producto

Caso de Uso	Ver Producto
ID	CU07
Descripción	El usuario puede acceder a la vista de detalle de un producto y ver sus imágenes, descripción e información sobre las tallas y el stock
Actores Principales	Cliente No Autenticado y Cliente Autenticado
Precondiciones	▪ El sistema debe estar correctamente conectado a la base de datos.
Flujo Principal	1. El usuario accede a la página principal. 2. El sistema recupera los datos de los productos en el inventario. 3. El usuario selecciona un producto accediendo a la vista de detalle. 4. El sistema recupera los datos del producto, los datos de las unidades de cada producto y las imágenes. 5. El sistema muestra estos datos al usuario.

Flujos Alternativos	**2.1 No hay productos en el inventario:** 2.1a El sistema recupera una lista vacía de productos. 2.1b El sistema muestra un mensaje indicando que no hay productos disponibles. **2.2 Error en la conexión con la base de datos:** 2.2a El sistema no puede recuperar los datos del producto debido a que la base de datos no responde. 2.2b El sistema muestra un mensaje indicando que ocurrió un error en la conexión con la base de datos. **4.1 No hay imágenes disponibles** 4.1a El sistema recupera una lista vacía de imágenes. 4.1b El sistema muestra un mensaje indicando que no existen imágenes del producto. **4.2 Error en la conexión con la base de datos:** 4.2a El sistema no pudo recuperar los datos debido a que la base de datos no responde. 4.2b El sistema muestra un mensaje indicando que ocurrió un error en la conexión con la base de datos.

Tabla 12: Modificar Producto del Carrito

Caso de Uso	Modificar Producto del Carrito
ID	CU07
Descripción	El usuario puede modificar la cantidad de un producto que haya añadido previamente a su carrito de compras.
Actores Principales	Cliente Autenticado
Precondiciones	▪ El usuario debe estar autenticado en el sistema. ▪ El usuario debe tener productos en su carrito. ▪ El sistema debe estar correctamente conectado a la base de datos.

Flujo Principal	1. El usuario accede a su carrito de compras. 2. El usuario selecciona el producto que desea modificar. 3. El usuario ajusta la cantidad del producto. 4. El sistema actualiza la cantidad del producto en el carrito. 5. El sistema muestra la lista actualizada con el nuevo total de los productos.
Flujos Alternativos	**2.1 Cantidad no válida:** 2.1a El sistema detecta que la cantidad ingresada no es válida (por ejemplo, menor que 1). 2.1b El sistema muestra un mensaje solicitando una cantidad válida. **2.2 Error en la conexión con la base de datos:** 2.2a El sistema no puede actualizar el producto debido a un error de conexión. 2.2b El sistema muestra un mensaje indicando que ocurrió un error y que el producto no pudo ser modificado.

Tabla 13: Realizar Compra

Caso de Uso	Realizar Compra
ID	CU02
Descripción	El usuario autenticado puede completar la compra de los productos seleccionados en el carrito, seleccionando una dirección de envío y un método de pago, y confirmando el pedido.
Actores Principales	Cliente Autenticado
Precondiciones	▪ El usuario debe estar autenticado y tener productos en su carrito de compra. ▪ El sistema debe estar conectado a la base de datos para validar la disponibilidad de los productos y procesar la compra.

Flujo Principal	1. El usuario accede a su carrito de compra. 2. El selecciona en Realizar Compra. 3. El sistema solicita al usuario seleccionar o ingresar una dirección de envío. 4. El usuario selecciona o ingresa la dirección de envío. 5. El sistema solicita al usuario seleccionar un método de pago. 6. El usuario selecciona el método de pago y confirma la compra. 7. El sistema verifica la disponibilidad de los productos en la base de datos. 8. El sistema procesa el pago utilizando el método seleccionado. 9. El sistema crea el pedido y devuelve el código al usuario. 10. El sistema muestra un mensaje de confirmación.
Flujos Alternativos	**2.1 No hay productos en el carrito:** 2.1a El usuario pulsa Realizar Compra 2.1b El sistema muestra un mensaje informativo indicando que no hay productos en el carrito y no se puede proceder con la compra. **2.2 Productos no disponibles:** 2.2a El sistema detecta que uno o más productos en el carrito ya no están disponibles. 2.2b El sistema muestra un mensaje al usuario indicando qué productos no están disponibles y solicita la revisión del carrito antes de continuar. **2.3 Error en la conexión con la base de datos:** 2.3a El sistema intenta procesar la compra, pero hay un error en la conexión con la base de datos. 2.3b El sistema muestra un mensaje informativo indicando que no se puede procesar la compra en este momento y solicita al usuario intentarlo más tarde.

9.2. Diagrama ER

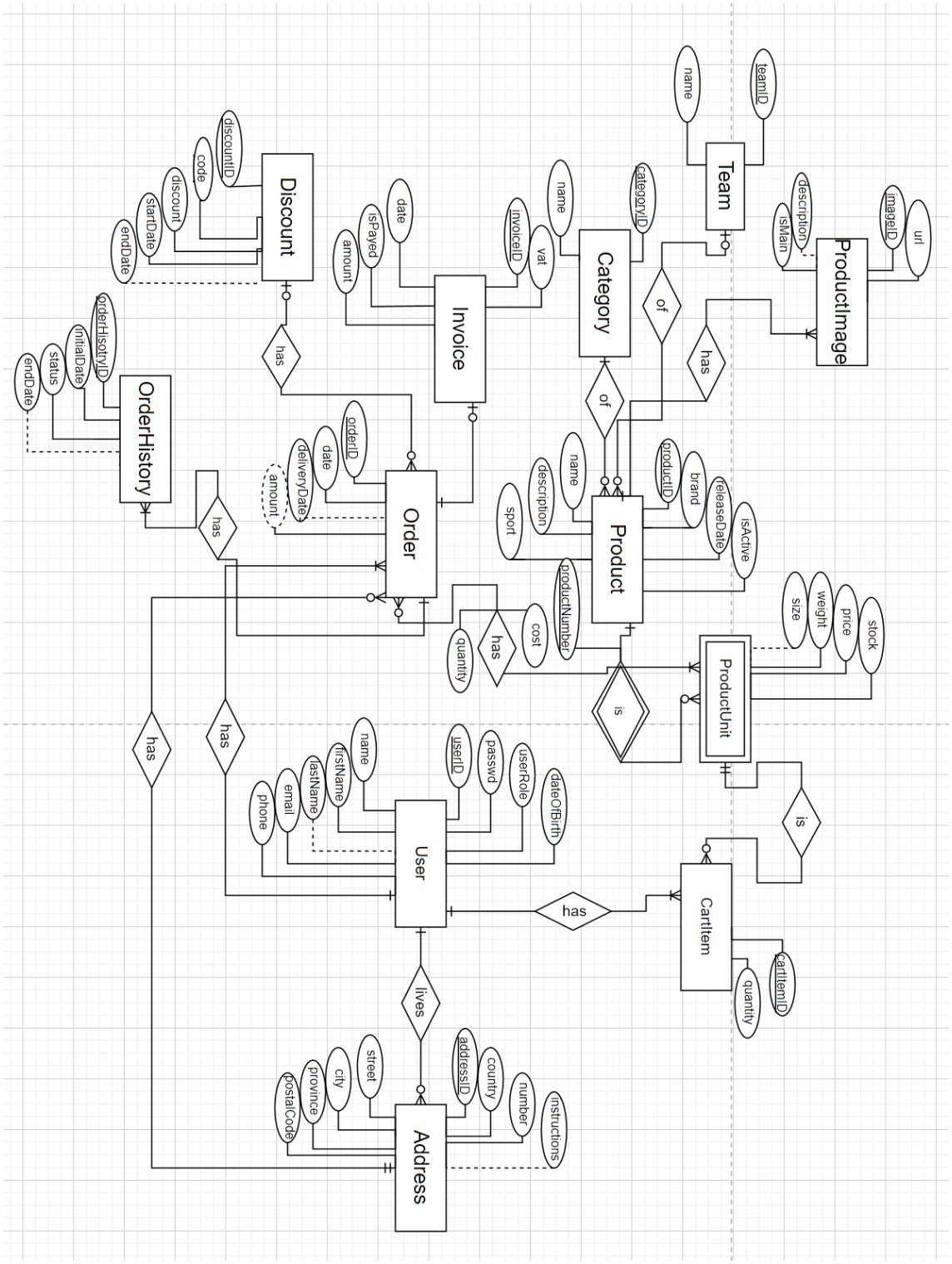

Figura 38: Diagrama ER de la base de datos

9.3. Diseño arquitecural

9.3.1. Esquema de la base de datos

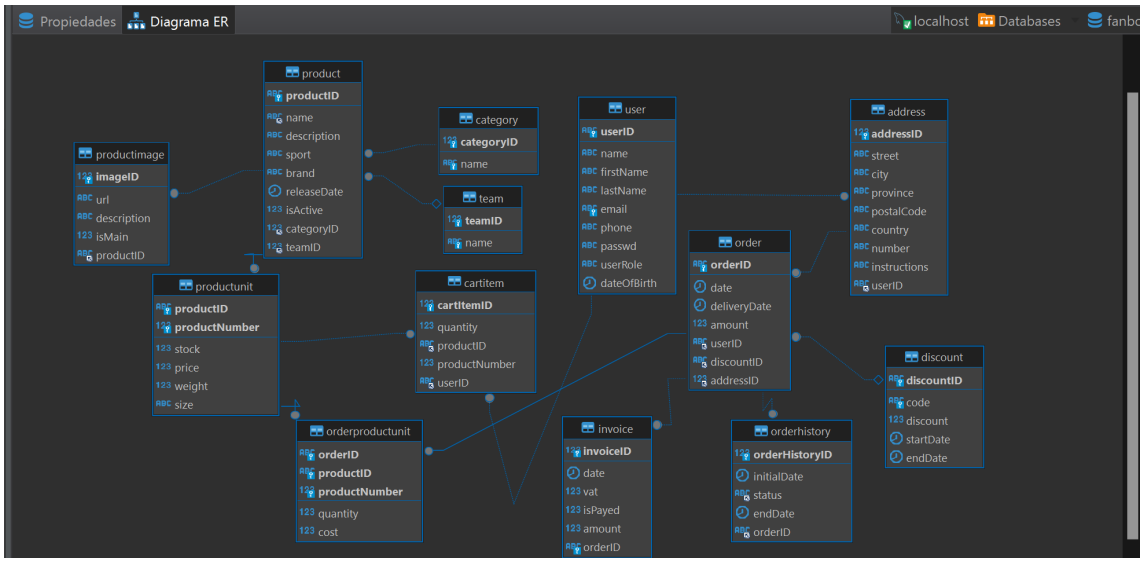

Figura 39: Esquema de la base de datos

9.3.2. Códigos base de datos

9.3.2.1. Script base de datos

```
-- Crear la base de datos si no existe

create database prueba
-- Creaci n de tablas
-- Tabla: Category
DROP TABLE IF EXISTS Category;
CREATE TABLE Category (
    categoryID INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(100) NOT NULL,
    PRIMARY KEY (categoryID),
    UNIQUE (name)
) ENGINE=InnoDB;

-- Tabla: Product
DROP TABLE IF EXISTS Product;
CREATE TABLE Product (
    productID CHAR(36) NOT NULL,
    name VARCHAR(100) NOT NULL,
    description TEXT NOT NULL,
    sport VARCHAR(50) NOT NULL,
    brand VARCHAR(50) NOT NULL,
    releaseDate DATE NOT NULL DEFAULT CURDATE(),
    isActive BIT(1) NOT NULL DEFAULT 1,
    categoryID INT NOT NULL,
    PRIMARY KEY (productID),
    UNIQUE (name, categoryID),
    FOREIGN KEY (categoryID) REFERENCES Category(categoryID)
) ENGINE=InnoDB;

```

```sql
-- Tabla: ProductUnit
DROP TABLE IF EXISTS ProductUnit;
CREATE TABLE ProductUnit (
    productID CHAR(36) NOT NULL,
    productNumber INT AUTO_INCREMENT NOT NULL,
    stock SMALLINT NOT NULL,
    price DECIMAL(6,2) NOT NULL,
    weight DECIMAL(5,2) NOT NULL,
    size CHAR(1),
    PRIMARY KEY (productID, productNumber),
    FOREIGN KEY (productID) REFERENCES Product(productID),
    CHECK (size IN ('XS', 'S', 'M', 'L', 'XL')),
    CHECK (stock > 0),
    CHECK (price > 0)
) ENGINE=InnoDB;

-- Tabla: ProductImage
DROP TABLE IF EXISTS ProductImage;
CREATE TABLE ProductImage (
    imageID INT AUTO_INCREMENT NOT NULL,
    url VARCHAR(255) NOT NULL,
    description TEXT,
    isMain BIT(1) NOT NULL DEFAULT 0,
    productID CHAR(36) NOT NULL,
    PRIMARY KEY (imageID),
    FOREIGN KEY (productID) REFERENCES Product(productID)
) ENGINE=InnoDB;

-- Tabla: User
DROP TABLE IF EXISTS User;
CREATE TABLE User (
    userID CHAR(36) NOT NULL,
    firstName VARCHAR(50) NOT NULL,
    lastName VARCHAR(50),
    email VARCHAR(100) NOT NULL,
    phone VARCHAR(9) NOT NULL,
    passwd VARCHAR(255) NOT NULL,
    userRole CHAR(8) NOT NULL,
    dateOfBirth DATE NOT NULL,
    PRIMARY KEY (userID),
    UNIQUE (email),
    CHECK (email LIKE '%@%.%'),
    CHECK (phone REGEXP '^[0-9]{9}$'),
    CHECK (userRole IN ('admin', 'user'))
) ENGINE=InnoDB;

-- Tabla: Address
DROP TABLE IF EXISTS Address;
CREATE TABLE Address (
    addressID INT AUTO_INCREMENT NOT NULL,
    street VARCHAR(100) NOT NULL,
    city VARCHAR(50) NOT NULL,
    province VARCHAR(50) not null,
    postalCode VARCHAR(10) NOT NULL,
    country VARCHAR(50) NOT NULL,
    number VARCHAR(5) NOT NULL,
    instructions TEXT,
    userID CHAR(36) NOT NULL,
    PRIMARY KEY (addressID),
    FOREIGN KEY (userID) REFERENCES User(userID),
    CHECK (postalCode REGEXP '^[0-9]{5}$'),
```

```
91      CHECK (number REGEXP '^[0-9]{1,5}$')
92  ) ENGINE=InnoDB;
93
94  -- Tabla: Discount
95  DROP TABLE IF EXISTS Discount;
96  CREATE TABLE Discount (
97      discountID CHAR(36) NOT NULL,
98      code VARCHAR(50) NOT NULL,
99      discount DECIMAL(5,2) NOT NULL,
100     startDate DATE NOT NULL,
101     endDate DATE,
102     PRIMARY KEY (discountID),
103     UNIQUE (code)
104 ) ENGINE=InnoDB;
105
106 -- Tabla: Order
107 DROP TABLE IF EXISTS `Order`;
108 CREATE TABLE `Order` (
109     orderID CHAR(36) NOT NULL,
110     date DATE DEFAULT CURDATE() NOT NULL,
111     deliveryDate DATE,
112     amount DECIMAL(10,2) NOT NULL,
113     userID CHAR(36) NOT NULL,
114     discountID CHAR(36),
115     addressID INT NOT NULL,
116     PRIMARY KEY (orderID),
117     FOREIGN KEY (userID) REFERENCES User(userID),
118     FOREIGN KEY (discountID) REFERENCES Discount(discountID),
119     FOREIGN KEY (addressID) REFERENCES Address(addressID)
120 ) ENGINE=InnoDB;
121
122 -- Tabla: OrderHistory
123 DROP TABLE IF EXISTS OrderHistory;
124 CREATE TABLE OrderHistory (
125     orderHistoryID INT AUTO_INCREMENT NOT NULL,
126     initialDate DATE NOT NULL,
127     status CHAR(20) NOT NULL,
128     endDate DATE,
129     orderID CHAR(36) NOT NULL,
130     PRIMARY KEY (orderHistoryID),
131     UNIQUE (status, orderID),
132     FOREIGN KEY (orderID) REFERENCES `Order`(orderID),
133     CHECK (status IN ('Pendiente', 'Procesado', 'Enviado', 'Entregado', '
            Cancelado', 'Devuelto'))
134 ) ENGINE=InnoDB;
135
136 -- Tabla: OrderProductUnit
137 DROP TABLE IF EXISTS OrderProductUnit;
138 CREATE TABLE OrderProductUnit (
139     quantity INT NOT NULL,
140     price DECIMAL(6,2) NOT NULL,
141     orderID CHAR(36) NOT NULL,
142     productID CHAR(36) NOT NULL,
143     productNumber INT NOT NULL,
144     PRIMARY KEY (orderID, productID, productNumber),
145     FOREIGN KEY (orderID) REFERENCES `Order`(orderID) ON DELETE CASCADE ON
            UPDATE CASCADE,
146     FOREIGN KEY (productID, productNumber) REFERENCES ProductUnit(productID
            , productNumber) ON DELETE CASCADE ON UPDATE CASCADE,
147     CHECK (quantity > 0)
148 ) ENGINE=InnoDB;
```

```sql

-- Tabla: CartItem
DROP TABLE IF EXISTS CartItem;
CREATE TABLE CartItem (
    cartItemID INT AUTO_INCREMENT NOT NULL,
    quantity SMALLINT NOT NULL,
    productID CHAR(36) NOT NULL,
    productNumber INT NOT NULL,
    userID CHAR(36) NOT NULL,
    PRIMARY KEY (cartItemID),
    FOREIGN KEY (productID, productNumber) REFERENCES ProductUnit(productID
        , productNumber),
    FOREIGN KEY (userID) REFERENCES User(userID),
    CHECK (quantity > 0)
) ENGINE=InnoDB;

-- Tabla: Invoice
DROP TABLE IF EXISTS Invoice;
CREATE TABLE Invoice (
    invoiceID INT AUTO_INCREMENT NOT NULL,
    date DATE NOT NULL,
    vat DECIMAL(5,2) NOT NULL DEFAULT 21.00,
    isPayed BIT(1) DEFAULT 0 NOT NULL,
    amount DECIMAL(10,2) NOT NULL,
    orderID CHAR(36) NOT NULL,
    PRIMARY KEY (invoiceID),
    FOREIGN KEY (orderID) REFERENCES `Order`(orderID),
    UNIQUE (orderID),
    CHECK (vat BETWEEN 0 AND 100)
) ENGINE=InnoDB;

--  ndices  adicionales

--  ndices  Nonclustered en columnas de uso frecuente en consultas
CREATE INDEX idx_Order_userID ON `Order`(userID);
CREATE INDEX idx_OrderProductUnit_productID ON OrderProductUnit(productID);
CREATE INDEX idx_ProductUnit_productID_productNumber ON ProductUnit(
    productID, productNumber);
```

9.3.2.2. Script disparadores

```sql
CREATE TRIGGER `TR_SingleMainImage` BEFORE
INSERT ON `productimage` FOR EACH ROW BEGIN IF NEW.isMain = 1 THEN --
    Verificar si ya existe una imagen principal para este producto
    IF EXISTS (
        SELECT 1
        FROM ProductImage
        WHERE productID = NEW.productID
            AND isMain = 1
    ) THEN SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Solo una imagen principal por producto';
END IF;
END IF;
END;
CREATE TRIGGER `TR_ValidateDiscount` BEFORE
INSERT ON `order` FOR EACH ROW BEGIN
DECLARE discountValid BOOLEAN;
IF NEW.discountID IS NOT NULL THEN
SET discountValid = (
        SELECT COUNT(*)
        FROM Discount
```

```
        WHERE discountID = NEW.discountID
            AND CURRENT_DATE BETWEEN startDate AND endDate
    );
IF discountValid = 0 THEN SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Descuento no v lido o expirado';
END IF;
END IF;
END;
CREATE TRIGGER `TR_CheckStockBeforeInsertCart` BEFORE
INSERT ON `cartitem` FOR EACH ROW BEGIN IF (
        SELECT stock
        FROM ProductUnit
        WHERE productID = NEW.productID
            AND productNumber = NEW.productNumber
    ) < NEW.quantity THEN SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'No hay suficiente stock';
END IF;
END;
CREATE TRIGGER `TR_CheckStockBeforeUpdateCart` BEFORE
UPDATE ON `cartitem` FOR EACH ROW BEGIN IF (
        SELECT stock
        FROM ProductUnit
        WHERE productID = NEW.productID
            AND productNumber = NEW.productNumber
    ) < NEW.quantity THEN SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'No hay suficiente stock';
END IF;
END;
```

9.4. Diseño del backend

URI	Métodos HTTP	Resumen
{baseURL}/api/auth		
/register	POST	Registro de usuario
/login	POST	Autenticación
{baseURL}/api/products		
/	GET, POST	Gestión de productos
/:productID	GET, PUT, DELETE	CRUD de un producto
/:productID/details	GET	Producto, imágenes y unidades
/:productID/images	GET, POST	Gestión de imágenes
/:productID/images/:imageID	GET, PUT, DELETE	CRUD de imagen por ID
/:productID/main-image	GET	Imagen principal
/:productID/units	GET, POST	Gestión de unidades
/:productID/units/:productNumber	GET, PUT, DELETE	CRUD de unidad
{baseURL}/api/categories		
/	GET, POST	Gestión de categorías
/:categoryID	GET, PUT, DELETE	CRUD de una categoría
{baseURL}/api/teams		
/	GET, POST	Gestión de equipos
/:teamID	GET, PUT, DELETE	CRUD de un equipo
{baseURL}/api/orders		

URI	Métodos HTTP	Resumen
/	GET, POST	Gestión de pedidos
/:orderID	GET, PUT, DELETE	CRUD de un pedido
/:orderID/products	GET	Productos de un pedido
/:orderID/currentStatus	GET	Estado del pedido
/:orderID/history	GET, POST	Historial del pedido
{baseURL}/api/users		
/	GET, POST	Gestión de usuarios
/:userID	GET, PUT, DELETE	CRUD de un usuario
/:userID/addresses	GET, POST	Gestión direcciones
/:userID/addresses/:addressID	GET, PUT, DELETE	CRUD de una dirección
/:userID/cart	GET, POST	Gestión carrito de compras
/:userID/cart/:cartItemID	PUT, DELETE	CRUD de un ítem del carrito
/:userID/orders	GET	Pedidos de usuario
{baseURL}/api/discounts		
/	GET,POST	Gestión de descuentos
/:discountID	GET	CRUD de un descuento

Tabla 14: Diseño de las rutas de la API

9.5. Código del errorHandler

```
import { ServiceError } from '../utils/serviceError.js';

export const errorHandler = (err, req, res, next) => {
  console.error(err.stack); // Registra el error en la consola del servidor
  console.log(err + "handler");

  let message = 'Internal Server Error';
  let statusCode = 500;
  let code = 'INTERNAL_ERROR';

  if (err instanceof ServiceError) {
    message = err.message;
    code = err.code;

    switch (err.code) {
      case 'VALIDATION_ERROR':
      case 'INVALID_QUANTITY':
        statusCode = 400;
        break;
      case 'NOT_FOUND':
        statusCode = 404;
        break;
      case 'ADDRESS_MISMATCH':
      case 'INVALID_PASSWORD':
      case 'INVALID_TOKEN':
      case 'UNAUTHORIZED':
        statusCode = 403;
        break;
      case 'INSUFFICIENT_STOCK':
```