
Grado en Ingeniería de Tecnologías de Telecomunicación

Trabajo Fin de Grado

Desarrollo de e-Commerce aplicando procesos Devops

Development of e-Commerce applying DevOps processes

Alfredo Crespo Fernández-Tostado
Cuenca, julio, 2024

ACTA DE TRABAJO DEL FIN DE GRADO (TRIBUNAL COLEGIADO)

Datos del título

Denominación: Grado en Ingeniería de Tecnologías de Telecomunicación

Curso académico:

Convocatoria:

Datos del estudiante y del trabajo de fin de grado

Documento:

Apellidos:

Nombre:

Título del trabajo de fin de grado (español):

Título del trabajo de fin de grado (inglés):

Tutores

- Tutor principal (Universidad de Castilla-La Mancha):
- Cotutor (Universidad de Castilla-La Mancha o Nombre de Empresa/Institución):

Miembros del tribunal

- Nombre y Apellidos (Presidente/a):
- Nombre y Apellidos (Secretario/a):
- Nombre y Apellidos (Vocal):

Reunido el Tribunal de Evaluación con fecha _____ de _____ se declara abierta la sesión pública, el Secretario comprueba la identidad de los participantes (estudiante y miembros del tribunal) y

ACUERDA otorgar al alumno la CALIFICACIÓN GLOBAL de _____.

Justificación en el caso de calificación global suspensa:

Justificación matrícula de honor:

Justificación premio:

Presidente/a

Secretario/a

Vocal

Resumen

Las Tecnologías de la Información y la Comunicación (*TIC*) desempeñan un papel fundamental en la sociedad actual, impactando en todos los aspectos de nuestras vidas. Especialmente en el ámbito rural, estas tecnologías ofrecen una serie de beneficios significativos. Las plataformas en línea proporcionan a los residentes rurales la capacidad de establecer conexiones directas con los consumidores, lo que resulta en la eliminación de intermediarios y un incremento en sus ganancias.

El trabajo de fin de grado se enfoca en desarrollar una aplicación de comercio electrónico (*e-Commerce*) utilizando la tecnología *React* y aplicando procesos *DevOps*. La elección de este tipo de aplicación se justifica por las múltiples ventajas que ofrece para las empresas rurales. Por un lado, el comercio electrónico puede representar una fuente adicional de ingresos, reduciendo su dependencia de las ventas locales. Además, les brinda la oportunidad de alcanzar a clientes en todo el mundo, ampliando así su base de clientes y aumentando las oportunidades de venta.

Para el desarrollo de la aplicación, se ha optado por la tecnología *React*, reconocida por su capacidad para crear interfaces de usuario interactivas y eficientes. *React*, una biblioteca de JavaScript mantenida por *Facebook*, simplifica la creación de componentes reutilizables, lo que agiliza el proceso de desarrollo y mejora la mantenibilidad del código. Su popularidad y el respaldo de una gran comunidad de desarrolladores la convierten en una opción sólida para proyectos de este tipo.

Además, se explora el concepto de *DevOps* y su importancia en el desarrollo de software actual. La adopción de prácticas *DevOps* permite reducir tiempos de desarrollo, mejorar la calidad del *software* y aumentar la eficiencia operativa, lo que se traduce en una mayor capacidad de respuesta a las demandas del mercado y una mayor competitividad para las empresas.

En resumen, este trabajo de fin de grado investiga la creación de una aplicación de comercio electrónico utilizando *React* y la implementación de procesos *DevOps*, resaltando las ventajas y relevancia de estas tecnologías en el contexto actual de desarrollo de *software*, especialmente en el ámbito rural.

Abstract

Information and Communication Technologies (*ICT*) play a fundamental role in today's society, impacting every aspect of our lives. Particularly in rural areas, these technologies offer a range of significant benefits. Online platforms provide rural residents with the ability to establish direct connections with consumers, thereby eliminating intermediaries and increasing their profits.

The undergraduate thesis focuses on developing an *e-Commerce* application using *React* technology and applying *DevOps* processes. The choice of this type of application is justified by the multiple advantages it offers for rural businesses. Firstly, *e-Commerce* can serve as an additional source of income, reducing dependence on local sales. Furthermore, it provides them with the opportunity to reach customers worldwide, thus expanding their customer base and increasing sales opportunities.

For the application development, *React* technology has been chosen for its ability to create interactive and efficient user interfaces. *React*, a *JavaScript* library maintained by Facebook, simplifies the creation of reusable components, speeding up the development process and enhancing code maintainability. Its popularity and strong developer community support make it a solid choice for such projects.

Additionally, the concept of *DevOps* and its importance in current software development is explored. Adopting *DevOps* practices helps reduce development times, improve software quality, and increase operational efficiency, resulting in better responsiveness to market demands and greater competitiveness for businesses.

In summary, this undergraduate thesis investigates the creation of an *e-Commerce* application using React and the implementation of *DevOps* processes, highlighting the advantages and relevance of these technologies in the current software development context, especially in rural areas.

Agradecimientos

En primer lugar, deseo expresar mi sincero agradecimiento a mi familia, en particular a mis padres, quienes siempre han brindado su apoyo incondicional en los momentos más importantes de mi vida. Quiero reconocer y agradecer a mis compañeros de carrera, quienes han sido leales compañeros a lo largo de estos años, compartiendo tanto tiempo como experiencias conmigo. Además, deseo expresar mi agradecimiento a los directores de mi Trabajo de Fin de Grado por su generosa dedicación a este proyecto. No puedo dejar de expresar mi gratitud al resto del cuerpo docente de la Escuela Politécnica de Cuenca, quienes han desempeñado un papel fundamental en mi desarrollo académico. Les agradezco sinceramente a todos por su apoyo y contribución.

Alfredo Crespo Fernández-Tostado
Cuenca, 2024

Notación y acrónimos

LISTA DE ACRÓNIMOS

Lista *ordenada alfabéticamente* con los acrónimos empleados en el TFG.

API	Interfaz de Programación de Aplicaciones
AWS	Amazon Web Services
BBDD	Base de datos
CI	Integración continua
CORS	Compartición de recursos entre orígenes
CSS	Hojas de estilo en cascada
DOM	Modelo de objeto de documento
ES6	ECMAScript 6
FTP	Protocolo de transferencia de archivos
HTML	Lenguaje de marcado de hipertexto
HTTP	Protocolo de transferencia de hipertexto
IDE	Entorno de desarrollo integrado
JSON	JavaScript Object Notation
JWT	JSON Web Token
MVC	Modelo-Vista-Controlador
Mern	MongoDB, Express.js, React.js, Node.js
PC	Computadora personal
REST	Transferencia de estado representacional
SDK	Kit de desarrollo de software
SDLC	Ciclo de vida de desarrollo de software
SQL	Lenguaje de consulta estructurado
SSH	Secure Shell
SVN	Subversion
TCP/IP	Protocolo de control de transmisión / Protocolo de Internet
UDP	Protocolo de datagramas de usuario
URL	Localizador uniforme de recursos
URI	Identificador de recursos uniforme
W3C	Consorcio World Wide Web
XML	Lenguaje de marcado extensible

YAML	YAML Ain't Markup Language
B2B	Business-to-Business
B2C	Business-to-Consumer
B2E	Business-to-Employee
C2B	Consumer-to-Business
C2C	Consumer-to-Consumer
CNMC	Comisión Nacional de los Mercados y la Competencia
DEVOPS	Desarrollo y operaciones
E2E	End-to-End
QA	Assurance de calidad
IVA	Impuesto sobre el Valor Añadido

Índice general

Resumen	v
Abstract	vii
Agradecimientos	ix
Notación y acrónimos	xI
Índice de figuras	xv
Índice de tablas	xvII
Índice de Bloques de código	xix
Parte I: Memoria	1
1. Introducción	3
1.1. Motivación	3
1.2. Objetivos	3
1.3. Estructura del documento	4
2. Marco Teórico	5
2.1. Introducción a las Tecnologías Web	5
2.2. Arquitectura Actual de Aplicaciones Web	9
3. Estado del Arte	13
3.1. <i>e-Commerce</i>	13
3.2. <i>Frameworks</i> y Tecnologías <i>Frontend</i> Relevantes	15
3.3. <i>DevOps</i> en el desarrollo de <i>software</i>	18
3.4. Herramientas Populares en <i>DevOps</i>	22
4. Metodología	27
4.1. Desarrollo iterativo e incremental	27
4.2. El repositorio <i>GitHub</i>	27
4.3. Lenguajes de programación en React	30
4.4. <i>Backend</i> de la aplicación	32
4.5. El <i>testing</i> en <i>DevOps</i>	35
4.6. Entorno de desarrollo	37
5. Desarrollo de la Aplicación	39
5.1. Arquitectura de la Aplicación	39
5.2. Configuración Inicial	40

5.3. Estructura del Proyecto	40
5.4. Desarrollo del Frontend	41
5.5. Desarrollo del Backend	62
5.6. Otras integraciones	66
6. Aplicación de Procesos DevOps	69
6.1. GitFlow	69
6.2. Despliegue del e-Commerce en Render	71
6.3. Pruebas en el <i>e-Commerce</i>	73
6.4. Desarrollo del <i>CI/CD</i> en la aplicación	78
7. Resultados obtenidos	83
7.1. laManchaCommerce. Descripción y características	83
7.2. Resultado final	83
8. Conclusiones y líneas futuras	91
8.1. Conclusiones	91
8.2. Líneas futuras	91
Parte II: Planos	93
Parte III: Pliego de condiciones	99
9. Pliego de Condiciones	101
9.1. Hardware	101
Parte IV: Presupuesto	105
10. Presupuesto	107
10.1. Hardware	107
10.2. Costes derivados del proyecto	107
10.3. Presupuesto Final	108
Bibliografía	109
A. Repositorio de GitHub	115

Índice de figuras

2.1. Conjunto de protocolos TCP/IP. Fuente: Página IBM [21]	5
2.2. Comunicación entre capas de nodos. Fuente: Temario Aplicaciones Web UCM [28]	6
2.3. Esquema de Comunicación en el Modelo TCP/IP. Fuente: Temario Aplicaciones Web UCM [28]	6
2.4. Esquema General Protocolo <i>HTTP</i> . Fuente: Temario Aplicaciones Web UCM [28]	8
2.5. Mensajes <i>HTTP</i> . Fuente: Página Mozilla Developers [29]	9
 3.1. Ramas de actividad con mayor porcentaje de volumen de negocio del comercio electrónico. Fuente: CNMC [6]	14
3.2. <i>e-Commerce</i> en España 2023. Fuente: Data Comunicación[7]	14
3.3. Logo React. Fuente: Página oficial de React [36]	15
3.4. Logo de <i>Angular</i> . Fuente: Página oficial de <i>Angular</i> [3]	16
3.5. Logo de <i>Vue.js</i> . Fuente: Página oficial de <i>Vue.js</i> [43]	17
3.6. Intersección de Desarrollo, Operaciones y Calidad en el marco de <i>DevOps</i> . Fuente: Página Xeridia Devops[48]	18
3.7. Ciclo de vida del desarrollo de software. Fuente: Página Xeridia Devops[48]	19
3.8. Logo <i>Git</i> . Fuente: Página oficial de <i>Git</i> [17]	22
3.9. Logo SVN. Fuente: Wikipedia [47]	23
3.10. Logo Heroku. Fuente: Página oficial Heroku [20]	24
3.11. Logo <i>Render</i> .Fuente Página oficial <i>Render</i> [41]	25
 4.1. Logo GitHub. Fuente: Página oficial de GitHub [18]	27
4.2. Representación GitFlow. Fuente: Elaboración propia	28
4.3. Logo <i>JavaScript</i> . Fuente: Wikipedia [46]	30
4.4. Logo HTML. Fuente: Wikipedia [45]	31
4.5. Logo CSS. Fuente: Wikipedia [44]	31
4.6. Tecnologías más populares en 2023. Fuente: Página Stack Overflow [42]	32
4.7. Logo <i>Node.js</i> . Fuente: Página oficial de <i>Node.js</i> [31]	33
4.8. Número de módulos en varios idiomas. Fuente: Página Module Counts [27]	33
4.9. <i>MERN Stack</i> . Fuente: Página Devtechnosys [9]	34
4.10. Pirámide niveles de prueba. Fuente Página Qanewsblog [35]	36
4.11. Logo de <i>Cypress</i> . Fuente Página Oficial de Cypress [8]	37
4.12. Visual Studio Code. Fuente Página Oficial de <i>Visual Studio Code</i> [26]	38
 5.1. Arquitectura aplicación <i>MERN Stack</i> . Fuente: Elaboración propia	39
5.2. Consola mostrando versión de <i>Node.js</i> . Fuente: Elaboración propia	40
5.3. Estructura del Proyecto. Fuente: Elaboración propia	41
5.4. Página Web Piloto de <i>React</i> . Fuente: Elaboración propia	42
5.5. Estructura Proyecto <i>React</i> . Fuente: Elaboración propia	43
5.6. Diseño estructura de <i>e-Commerce</i> . Fuente: Elaboración propia	46
5.7. Diseño UI de <i>HomeScreen</i> . Fuente: Elaboración propia	50

5.8. Diseño UI de <i>SigninScreen</i> . Fuente: Elaboración propia	51
5.9. Diseño UI de <i>SignupScreen</i> . Fuente: Elaboración propia	52
5.10. Diseño UI de <i>ProductScreen</i> . Fuente: Elaboración propia	54
5.11. Diseño UI de <i>CartScreen</i> . Fuente: Elaboración propia	55
5.12. Diseño UI de <i>ShippingAddressScreen</i> . Fuente: Elaboración propia	56
5.13. Diseño UI de <i>PaymentMethodScreen</i> . Fuente: Elaboración propia	57
5.14. Diseño UI de <i>PlaceOrderScreen</i> . Fuente: Elaboración propia	58
5.15. Diseño UI de <i>OrderScreen</i> . Fuente: Elaboración propia	58
5.16. Estructura del Backend. Fuente: Elaboración propia	63
5.17. <i>PayPal Developer</i> . Fuente: Página de PayPal Developer	66
6.1. Crear repositorio en <i>GitHub</i> . Fuente: Elaboración propia	69
6.2. Crear <i>Pull Request</i> en <i>GitHub</i> . Fuente: Elaboración propia	71
6.3. Cambios en la <i>Pull Request</i> . Fuente: Elaboración propia	71
6.4. Configuración de <i>Render Web Service</i> . Fuente: Elaboración propia	72
6.5. Pantalla de bienvenida Cypress. Fuente: Elaboración propia	74
6.6. Estructura proyecto tests de Cypress. Fuente: Elaboración propia	75
6.7. Esquema CI CD GitHub Actions. Fuente: Elaboración propia	81
6.8. Definición de <i>Secrets GitHub</i> . Fuente: Elaboración propia	81
7.1. Pantalla Inicial <i>laManchaCommerce</i> . Fuente: Elaboración propia	84
7.2. Pantalla de Iniciar Sesión <i>laManchaCommerce</i> . Fuente: Elaboración propia	85
7.3. Pantalla de Registro <i>laManchaCommerce</i> . Fuente: Elaboración propia	85
7.4. Pantalla de Información del Producto <i>laManchaCommerce</i> . Fuente: Elaboración propia	86
7.5. Pantalla de Carro de Compras de <i>laManchaCommerce</i> . Fuente: Elaboración propia	86
7.6. Pantalla de Dirección de Envío de <i>laManchaCommerce</i> . Fuente: Elaboración propia	87
7.7. Pantalla de Método de Pago de <i>laManchaCommerce</i> . Fuente: Elaboración propia	87
7.8. Pantalla de Realizar Pedido de <i>laManchaCommerce</i> . Fuente: Elaboración propia	88
7.9. Pop-up de PayPal de <i>laManchaCommerce</i> . Fuente: Elaboración propia	88
7.10. <i>Pop-up de PayPal</i> de <i>laManchaCommerce</i> . Fuente: Elaboración propia	89
9.1. Ordenador HP Elitebook 840 G5. Fuente: PCComponentes	101
9.2. Monitor Philips V Line. Fuente: Amazon	102

Índice de tablas

2.1. Descripción de Métodos HTTP. Fuente: Página Mozilla Developers [29]	8
9.1. Especificaciones técnicas	101
9.2. Especificaciones del Monitor Philips	102
10.1. Costos de Equipos	107
10.2. Costes derivados del proyecto	107
10.3. Presupuesto Final	108

Índice de Bloques de código

4.1.	Ejemplo de código pruebas unitarias	35
4.2.	Ejemplo de código pruebas integración	35
4.3.	Ejemplo de código pruebas e2e	36
5.1.	Comando ver versión de Node	40
5.2.	Comando crear proyecto Node	41
5.3.	Comando crear proyecto React	41
5.4.	Comando crear proyecto React	41
5.5.	Comandos para instalar librerías React	42
5.6.	Header de App.js	44
5.7.	Main de App.js	45
5.8.	Main de App.js	46
5.9.	Spinner del componente LoadingBox	48
5.10.	Alert del componente MessageBox	48
5.11.	Ejemplo de Raiting	49
5.12.	Parte de código de HomeScreen	49
5.13.	Código fuente en JavaScript	50
5.14.	Ejemplo de COL	52
5.15.	Ejemplo de img	52
5.16.	Ejemplo deListGroup	53
5.17.	Ejemplo deListGroup.Item	53
5.18.	Ejemplo de sRating	53
5.19.	Ejemplo de Badge	54
5.20.	Ejemplo de Button	54
5.21.	Ejemplo de Form.Select	55
5.22.	Ejemplo de Form.Check	56
5.23.	Ejemplo de Card	57
5.24.	Ejemplo de useState	59
5.25.	Ejemplo de useContext	59
5.26.	Ejemplo de librería Axios GET	60
5.27.	Ejemplo de librería Axios POST	60
5.28.	Ejemplo de CSS	61
5.29.	Comandos para instalar Express y Mongoose	62
5.30.	Ejemplo de como dotenv carga variables de entorno	63
5.31.	Ejemplo de uso de mongoose	63
5.32.	Ejemplo de CORS en server.js	63
5.33.	Ejemplo de rutas en Serve.js	64
5.34.	Ejemplo de como dotenv carga variables de entorno	64
5.35.	Ejemplo de como mongoose	64
5.36.	Código de productModel	65
5.37.	Ejemplo de script en Matlab	66
5.38.	Ejemplo de Axios y Paypal	67

5.39. Ejemplo de usePayPalScriptReducer	67
5.40. Ejemplo de uso de JWT en Utils.js	67
6.1. Comando de Git para clonar	69
6.2. Comando push de Git	69
6.3. Comandos de commit en Git	70
6.4. Comando para instalar Cypress	73
6.5. Comando para abrir Cypress	73
6.6. Ejemplo de tests en Cucumber	76
6.7. Steps de Cucumber	77
6.8. Parte 1 del código del archivo YAML	78
6.9. Parte 2 del código del archivo YAML	79
6.10. Parte 3 del código del archivo YAML	80

Parte I: Memoria

CAPÍTULO 1

Introducción

1.1. MOTIVACIÓN

La motivación para la elección de este tema para mi Trabajo de Fin de Grado surge de una combinación de diversos factores. En primer lugar, el comercio electrónico (*e-Commerce*) se ha convertido en un fenómeno de relevancia en la sociedad actual, dado que ofrece una forma cómoda y eficiente de comprar. Se intensifica especialmente el uso de estas aplicaciones *web* durante la pandemia de *COVID-19*, ya que las restricciones de movilidad y las preferencias de compra cambiaron drásticamente.

Como “castellanomanchego”, me siento particularmente motivado a explorar la creación de un *e-Commerce* centrado en productos locales. Ya que el mundo rural se enfrenta a desafíos significativos en su desarrollo económico y social, y el comercio electrónico emerge como una herramienta poderosa para abordar estas dificultades.

La decisión de utilizar *React* para desarrollar este *e-Commerce* se basa en su posición destacada en el ámbito de las tecnologías web. *React*, una librería *JavaScript* mantenida por *Facebook*, destaca por su rendimiento excepcional, su flexibilidad y su facilidad de uso. Estas características son fundamentales para garantizar una experiencia de usuario fluida y atractiva, aspectos cruciales en el éxito de cualquier plataforma de comercio electrónico.

Además, este TFG también se centrará en la introducción y aplicación de los principios de la cultura *DevOps* aplicados en el desarrollo de nuestra aplicación *web e-Commerce*. El interés de estos procesos *DevOps* en estos últimos tiempos ha ido en aumento hasta convertirse en uno de los anhelos más profundos del negocio digital. Debido a que representa un conjunto de prácticas para mejorar la eficiencia y la colaboración en el ciclo de vida del desarrollo de *software*. Empresas líderes como *Amazon* y *Netflix* han demostrado el impacto positivo de estas prácticas, lo que nos inspira a implementarlas en nuestra aplicación web.

1.2. OBJETIVOS

Este TFG tiene como objetivo desarrollar un *e-Commerce* funcional para comprar y vender productos locales, aplicando principios *DevOps* para garantizar la eficiencia en el desarrollo, implementación y mantenimiento del sistema. Para lograr este objetivo general, se plantean los siguientes objetivos específicos:

- **Diseñar la arquitectura del *e-Commerce*:** Definir una arquitectura robusta y escalable para el desarrollo del *e-Commerce*.
- **Desarrollar el *e-Commerce* utilizando *React.js*:** Implementar la interfaz de usuario y la lógica de negocio utilizando el *framework* de *React.js*.
- **Aplicar procesos *DevOps*:** Implementar prácticas *DevOps* para automatizar y optimizar el ciclo de vida del desarrollo de *software*, incluyendo integración continua, entrega continua y despliegue continuo (*CI/CD*).

- **Evaluar la efectividad del e-Commerce:** Realizar pruebas exhaustivas para evaluar la funcionalidad, usabilidad y eficacia del e-Commerce desarrollado.

1.3. ESTRUCTURA DEL DOCUMENTO

Esta memoria se estructura de la siguiente manera:

1. **Introducción.** Se explica, de manera introductoria, los motivos que han llevado a escoger este proyecto como Trabajo Fin de Grado y qué es lo que se esperaba obtener de él.
2. **Marco Teórico.** Presenta conceptos fundamentales como el modelo *TCP/IP*, la arquitectura cliente/servidor, el protocolo *HTTP*, y la estructura de las aplicaciones *web*, incluyendo el *frontend*, *backend*, base de datos y *APIs*.
3. **Estado del Arte.** El estado del arte examina la evolución y las tendencias actuales del e-Commerce, destacando su impacto y transformación en el mercado actual. Además, se exploran los frameworks y tecnologías *frontend* más relevantes para la creación de interfaces dinámicas y efectivas. Finalmente, se analiza la integración de prácticas *DevOps* en el desarrollo de *software*, subrayando su papel en la eficiencia y la entrega continua.
4. **Metodología.** En esta sección, se detalla la metodología empleada en este TFG. Incluye el uso estratégico del repositorio *GitHub*, la implementación de lenguajes de programación en *React* para la interfaz de usuario, el desarrollo del *backend* de la aplicación, y prácticas de *testing* integradas en un entorno *DevOps*.
5. **Desarrollo de la Aplicación.** Esta parte abarca la implementación práctica de la aplicación e-Commerce utilizando las tecnologías y herramientas especificadas. Describe el proceso de desarrollo paso a paso, desde la configuración inicial del entorno de desarrollo hasta la construcción de funcionalidades clave del e-Commerce, asegurando coherencia con los principios de diseño y los requisitos del proyecto.
6. **Aplicación de Procesos DevOps.** Detalla cómo se implementan prácticas *DevOps* en el proyecto, desde la configuración de entornos hasta el uso de herramientas como *Git/GitHub* para control de versiones y la automatización de pruebas y despliegues.
7. **Resultados obtenidos.** Explica, analiza los resultados obtenidos y presenta el *feedback* recibido de los usuarios finales.
8. **Conclusiones y Líneas Futuras.** Resume las conclusiones clave derivadas del trabajo, las lecciones aprendidas, y sugiere posibles áreas para futuras investigaciones y mejoras en el ámbito del e-Commerce y *DevOps*.
9. **Bibliografía.** Lista de las referencias bibliográficas citadas en el texto.
10. **Anexos.** Contenidos auxiliares que complementan del trabajo.

CAPÍTULO 2

Marco Teórico

2.1. INTRODUCCIÓN A LAS TECNOLOGÍAS WEB

2.1.1. Modelo TCP/IP

El modelo *TCP/IP* (Protocolo de Control de Transmisión/Protocolo de Internet) es una explicación de los protocolos de red. Estos protocolos son un conjunto de normas para los formatos de mensaje y los procedimientos que permiten la comunicación entre equipos en la misma red. Los protocolos *TCP/IP* desempeñan un papel importante en el funcionamiento de Internet, ya que muchos otros servicios (como la *World Wide Web*, *FTP*, *SSH*, entre otros) que se encuentran en la red están sustentados por ellos [21].

Como se muestra en la Figura 2.1, estos modelos están basados en cuatro capas:

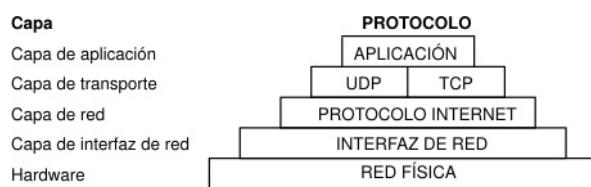


Figura 2.1: Conjunto de protocolos TCP/IP. Fuente: Página IBM [21]

- *Capa de Interfaz de Red* : La capa de interfaz de red, también conocida como capa de enlace de datos, define el método para enviar y recibir información a través de la red a la que se encuentra físicamente conectada. Esta capa es responsable de enviar paquetes *TCP/IP* en la red y de recibir paquetes *TCP/IP* desde fuera de la red [13].
- *Capa de Red* : La capa de red, también conocida como capa de Internet es la responsable de las funciones de direccionamiento, empaquetado y enrutamiento. Proporciona las funciones y los procedimientos necesarios para transferir secuencias de datos entre aplicaciones y dispositivos a través de las redes. En esta capa se encuentran los protocolos de más bajo nivel, destacando el protocolo *IP*.
- *Capa de Transporte* La capa de transporte es responsable de proporcionar una conexión de datos sólida y confiable entre la aplicación o el dispositivo original y su destino previsto. En este nivel, los datos se dividen en paquetes y se numeran para crear una secuencia. Este flujo de datos puede ser fiable (*Transmission Control Protocol, TCP*) o no fiable (*User Datagram Protocol, UDP*).
- *Capa de Aplicación* La capa de aplicación se refiere a los programas que utilizan *TCP/IP* para comunicarse entre sí. Es la encargada de manejar los detalles específicos de las diferentes aplicaciones que utilizará el usuario (*WWW, TELNET, FTP, etc.*). Este es el nivel con el cual los usuarios interactúan normalmente, como sistemas de correo electrónico y plataformas de mensajería.

Cada capa en el modelo *TCP/IP* se comunica con su homóloga en otro nodo (dispositivo) utilizando un protocolo como lenguaje común, como se muestra en la Figura 2.2. Es decir, cada capa en un dispositivo interactúa con la capa equivalente en otro dispositivo usando protocolos específicos que funcionan como lenguajes comunes, asegurando una comunicación efectiva y coherente a través de la red.

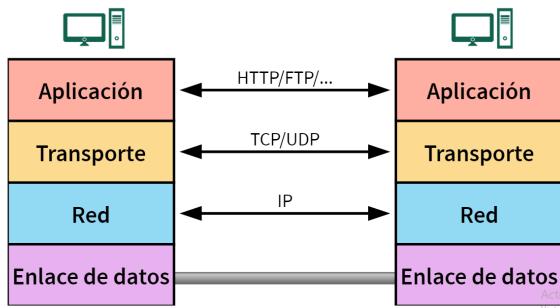


Figura 2.2: Comunicación entre capas de nodos. Fuente: Temario Aplicaciones Web UCM [28]

La comunicación entre capas homólogas no es directa, como se muestra en la Figura 2.3. Cada capa solicita a la capa inferior que transmita el mensaje al destino, agregando su propia información para cumplir con su función específica. Luego, la capa de enlace de datos se encarga del envío físico de los datos.

El nodo receptor recibe el mensaje a través de la capa de enlace. Cada capa interpreta la información adicional introducida por su capa homóloga durante el envío, y remite el mensaje original a la capa superior sin dicha información. Finalmente, la capa de aplicación recibe el mensaje de la misma manera en que fue enviado por la capa de aplicación en el origen.

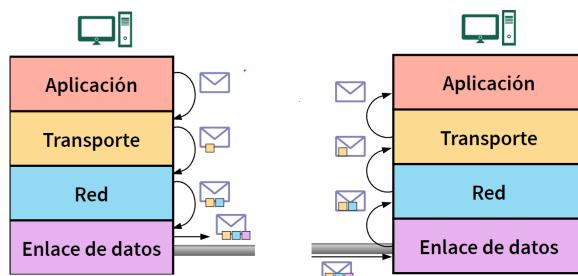


Figura 2.3: Esquema de Comunicación en el Modelo TCP/IP. Fuente: Temario Aplicaciones Web UCM [28]

2.1.2. Modelo Cliente/Servidor

En la arquitectura cliente/servidor, los clientes y servidores son entidades lógicamente separadas que se comunican a través de una red para llevar a cabo una o varias tareas conjuntamente. Un cliente solicita un servicio y recibe una respuesta, mientras que un servidor recibe y procesa la solicitud, devolviendo la respuesta solicitada.

Características de la arquitectura cliente/servidor [11]:

- **Protocolos asimétricos:** Los clientes inician el diálogo solicitando servicios, mientras que los servidores esperan pasivamente por estas solicitudes.
- **Encapsulación de servicios:** Los servidores determinan cómo realizar un trabajo cuando se les solicita un servicio, lo que permite actualizarlos sin afectar a los clientes, siempre que la interfaz pública de mensajes permanezca inalterada.

- **Integridad:** El mantenimiento centralizado de código y datos en el servidor reduce costos y protege la integridad de los datos compartidos, mientras que los clientes mantienen su independencia.
- **Transparencia de localización:** Los servidores pueden residir en la misma máquina que el cliente o en máquinas diferentes, y el *middleware* habitualmente oculta su ubicación a los clientes.
- **Intercambios basados en mensajes:** Los clientes y servidores intercambian solicitudes y respuestas utilizando mensajes, lo que permite una comunicación débilmente acoplada.
- **Modularidad y diseño extensible:** El diseño modular permite que la aplicación sea tolerante a fallos y pueda adaptarse automáticamente a cambios en la carga del sistema.
- **Independencia de la plataforma:** El *software* cliente/servidor ideal es independiente del *hardware* y sistemas operativos, permitiendo la mezcla de plataformas de clientes y servidores.
- **Código reutilizable:** Los servicios implementados pueden ser utilizados en varios servidores.
- **Escalabilidad:** Los sistemas cliente/servidor pueden escalarse horizontal o verticalmente para adaptarse a cambios en la carga o requerimientos de recursos.
- **Separación de la funcionalidad:** El modelo cliente/servidor proporciona una clara separación de funciones entre los procesos que ejecutan los servicios y los que los consumen.
- **Recursos compartidos:** Un servidor puede proporcionar servicios a múltiples clientes simultáneamente, gestionando el acceso a recursos compartidos de forma regulada.

2.1.3. Protocolo HTTP

El protocolo de transferencia de hipertexto (*HTTP*) es esencial en la comunicación web, permitiendo la transferencia de información a través de archivos como *XML*, *HTML*, entre otros, en la *World Wide Web*, como se muestra en la Figura 2.4. Constituye los cimientos de *Internet* y se utiliza para cargar páginas web mediante enlaces de hipertexto.

HTTP opera en la capa de aplicación de la arquitectura de red y facilita la transferencia de datos entre dispositivos conectados. Su funcionamiento se basa en un modelo de solicitud y respuesta entre un cliente y un servidor. En una interacción típica, el cliente envía una solicitud al servidor, que luego responde con un mensaje.

Este protocolo, orientado a transacciones, permite una comunicación eficiente y confiable entre sistemas distribuidos en la web, lo que lo convierte en una pieza fundamental para el funcionamiento de Internet.

Además, *HTTP* define un conjunto predefinido de métodos de petición para indicar la acción deseada sobre un recurso específico. Cada método representa una acción a ser realizada sobre el recurso identificado, cuya naturaleza depende de la aplicación del servidor. En la Tabla 2.1 se pueden observar algunos de los métodos. Por ejemplo, el recurso puede corresponder a un archivo que reside en el servidor.

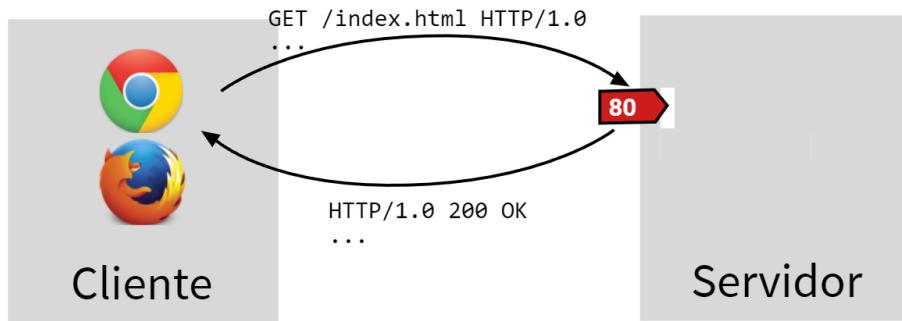


Figura 2.4: Esquema General Protocolo *HTTP*. Fuente: Temario Aplicaciones Web UCM [28]

Método	Descripción
<i>GET</i>	Solicita una representación de un recurso específico.
<i>HEAD</i>	Pide una respuesta idéntica a <i>GET</i> , pero sin el cuerpo de la respuesta.
<i>POST</i>	Envía una entidad a un recurso específico, causando cambios en el servidor.
<i>PUT</i>	Reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
<i>DELETE</i>	Borra un recurso específico.
<i>CONNECT</i>	Establece un túnel hacia el servidor identificado por el recurso.
<i>OPTIONS</i>	Utilizado para describir las opciones de comunicación para el recurso de destino.
<i>TRACE</i>	Realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino.
<i>PATCH</i>	Aplica modificaciones parciales a un recurso.

Tabla 2.1: Descripción de Métodos *HTTP*. Fuente: Página Mozilla Developers [29]

La estructura de las peticiones y respuestas *HTTP* sigue un patrón similar a la Figura 2.5, compuesto por:

- Una línea de inicio (*start-line*), que en la petición se conoce como línea de solicitud y en la respuesta como línea de estado. Esta línea tiene el siguiente formato:

[método] [recurso] [versión_HTTP]

Donde:

- [método]: Indica el método a realizar (*GET*, *POST*, *HEAD*, *PUT*, etc.).
- [recurso]: Es el nombre del recurso (*URI*) al que se accederá.
- [versión_HTTP]: Indica la versión de *HTTP* utilizada (*HTTP/1.0 o HTTP/1.1*).

- Cabeceras *HTTP*, que pueden indicar información sobre la petición o describir el cuerpo del mensaje incluido en el mensaje.
- Un campo de cuerpo de mensaje opcional (*body*), que lleva los datos asociados con la petición (como el contenido de un formulario *HTML*) o los archivos o documentos asociados a una respuesta (como una página *HTML* o un archivo de audio, vídeo, etc.). La presencia del cuerpo y su tamaño se indican en la línea de inicio y en las cabeceras *HTTP*.

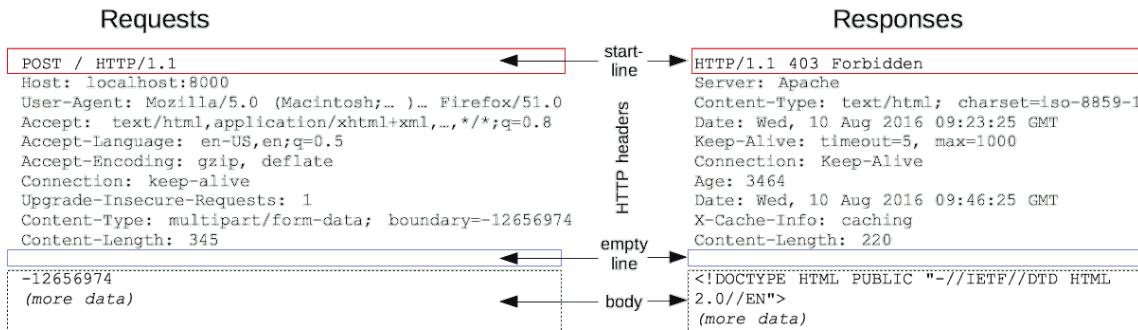


Figura 2.5: Mensajes HTTP. Fuente: Página Mozilla Developers [29]

2.2. ARQUITECTURA ACTUAL DE APLICACIONES WEB

2.2.1. Introducción

Una aplicación web es un sistema proporcionado por un servidor web que permite a los usuarios conectarse desde cualquier lugar a través de clientes web, como navegadores. El servidor web distribuye páginas de información formateada a los clientes que las solicitan mediante el protocolo *HTTP*. Cuando un cliente realiza una solicitud *HTTP*, el servidor web la recibe, localiza la página web correspondiente en su sistema de archivos o base de datos, y la envía de vuelta al navegador que la solicitó.

Estas aplicaciones se basan en la arquitectura cliente/servidor, donde el servidor web actúa como el centro de operaciones, gestionando y distribuyendo el contenido a los clientes. Las páginas web sirven como la interfaz de usuario para interactuar con la aplicación.

Una arquitectura de aplicaciones describe los patrones y las técnicas que se utilizan para diseñar y desarrollar aplicaciones. La arquitectura le proporciona un plan y las prácticas recomendadas que debe seguir para diseñar una aplicación bien estructurada. En una arquitectura de aplicaciones, habrá servicios de *frontend* y de *backend*. El desarrollo de *frontend* se refiere a la experiencia del usuario con la aplicación, mientras que el de *backend* implica proporcionar acceso a los datos, los servicios y otros sistemas que permiten el funcionamiento de la aplicación [38].

En el desarrollo de aplicaciones web modernas, es fundamental estructurar el sistema de manera que separe claramente las responsabilidades de cada componente. Esta separación no solo mejora la mantenibilidad y la escalabilidad de la aplicación, sino que también facilita el trabajo en equipo y la integración de nuevas funcionalidades. Una de las arquitecturas más comunes y efectivas para lograr esto es la arquitectura de tres capas, que se compone de las siguientes partes:

- **Capa de Presentación (*Frontend*):** La capa de presentación es la interfaz de usuario de la aplicación. Incluye todos los elementos visuales y controles con los que interactúan los usuarios finales. Esta capa se encarga de mostrar información al usuario y de interpretar sus comandos, transformándolos en acciones que se enviarán al *backend*.
- **Capa de Lógica de Negocio (*Backend*):** La capa de lógica de negocio gestiona la lógica de la aplicación y las reglas de negocio. Procesa las solicitudes del *frontend*, realiza cálculos necesarios, valida los datos y aplica las reglas de negocio. Esta capa actúa como intermediaria entre la capa de presentación y la base de datos, asegurando que las solicitudes del usuario se procesen correctamente y se apliquen las reglas de negocio.
- **Capa de Datos (Base de Datos):** La capa de datos es responsable del almacenamiento, recuperación y gestión de los datos. Almacena de manera persistente los datos y proporciona acceso a ellos cuando el *backend* lo solicita [15].

2.2.2. Frontend

El término *frontend* se refiere a la parte de una aplicación o sitio web con la que los usuarios interactúan directamente. Esta interfaz gráfica de usuario (*GUI*) incluye elementos como menús de navegación, diseño visual, botones, imágenes y gráficos. En términos técnicos, la página o pantalla que los usuarios ven, con todos sus componentes interactivos, se llama *Modelo de Objetos del Documento (DOM)* [1].

Existen tres lenguajes principales que influyen en la forma en que se desarrolla y se ve el *frontend*:

- **HTML (HyperText Markup Language):** Define la estructura de la página web, especificando los diferentes elementos que la componen, como títulos, párrafos, enlaces, imágenes, etc. El *HTML* es fundamental para crear la base de la interfaz de usuario.
- **CSS (Cascading Style Sheets):** Este lenguaje controla la presentación visual de la página web, incluyendo aspectos como el diseño, colores, fuentes, espaciado y más. *CSS* permite que los desarrolladores definan el estilo y la apariencia de los elementos *HTML*, mejorando así la experiencia del usuario.
- **JavaScript:** Agrega interactividad y dinamismo a la página web al permitir la manipulación del *DOM* y la implementación de funciones y comportamientos específicos. Con *JavaScript*, los desarrolladores pueden crear efectos visuales, animaciones, validar formularios, cargar contenido dinámicamente y mucho más.

Los desarrolladores *frontend* trabajan con una variedad de tecnologías, herramientas y lenguajes, incluyendo *HTML*, *CSS* y *JavaScript*, para crear interfaces de usuario atractivas y funcionales. Además, suelen utilizar bibliotecas y marcos de trabajo como *React*, *Angular* o *Vue.js* para simplificar y optimizar el desarrollo de aplicaciones web más complejas.

2.2.3. Backend

El *backend*, también conocido como el lado del servidor, se refiere a la parte de una aplicación web o sitio web que no es visible para el usuario final, pero que es esencial para su funcionamiento. Configura todos los aspectos lógicos de un sitio web o aplicación; abarca las funciones de lógica, almacenamiento de datos y seguridad necesarias para que una aplicación funcione de manera correcta y confiable, de modo que todas las acciones solicitadas en la página web se ejecuten correctamente [4].

Algunos ejemplos de tecnologías y lenguajes comunes utilizados en el *backend* incluyen:

- **Python:** Ampliamente utilizado por su simplicidad y versatilidad, con *frameworks* como *Django* y *Flask*.
- **JavaScript:** Con *Node.js* como entorno de ejecución, permite el desarrollo de aplicaciones *backend* y *frontend* con un lenguaje unificado.
- **Java:** Con *frameworks* como *Spring* e *Hibernate*, es popular en empresas para aplicaciones empresariales escalables.
- **PHP:** Principalmente utilizado en el desarrollo web, con *frameworks* como *Laravel* y *Symfony*.

2.2.4. Base de Datos

Una Base de Datos (*BBDD*) es una herramienta que permite capturar, organizar y relacionar datos, facilitando el acceso, análisis y explotación de estos datos por parte de usuarios, ya sean personas u otros sistemas. En esencia, una base de datos es un almacén que nos permite guardar grandes cantidades de información para su posterior recuperación, análisis y transmisión. Existen diversos tipos de bases de datos, clasificables de múltiples formas según el contexto, la utilidad o las necesidades que satisfacen. Una de las clasificaciones más utilizadas en el ámbito de *TI* es según el modelo de administración de datos. Las categorías más destacadas son:

- **Relacionales:** Los datos se almacenan y se acceden mediante relaciones preestablecidas, organizados en tablas con filas y columnas. El lenguaje más común en este tipo de bases de datos es *SQL (Structured Query Language)*. Ejemplos populares incluyen *PostgreSQL*, *MySQL*, *Oracle* y *MariaDB*.
- **No Relacionales o NoSQL:** Estas bases de datos proporcionan mecanismos para el almacenamiento y recuperación de datos en formatos distintos a las relaciones tabulares utilizadas en las bases de datos relacionales. Esta categoría incluye varias subcategorías:
 - **De tablas** (columna ancha o *BigTables*): Los datos se almacenan en tablas con numerosas columnas.
 - **Orientadas a Grafos:** La información se representa como nodos de un grafo y sus relaciones mediante aristas, permitiendo el uso de teoría de grafos para recorrer la base de datos. Ejemplos incluyen *Neo4j* y *Sparksee*.
 - **Clave-Valor:** Los datos se almacenan como pares clave-valor, donde una clave actúa como un identificador único. Tanto claves como valores pueden ser objetos simples o compuestos. Ejemplos incluyen *Redis* y *Dynamo*.
 - **Documentales:** Permiten almacenar y consultar datos como documentos en formatos *JSON*, *YAML*, *XML* y otros. Facilitan a los desarrolladores el uso del mismo formato de modelo de documentos que emplean en el código de aplicación. Ejemplos incluyen *MongoDB*, *CouchDB*, *SimpleDB* y *Lucene*.

Cada tipo de base de datos está diseñado para satisfacer diferentes necesidades y ofrece ventajas específicas según el uso y el contexto en que se emplea.

2.2.5. API

Una *API* (Interfaz de Programación de Aplicaciones) es un intermediario de *software* que permite que dos aplicaciones se comuniquen entre sí. Es un conjunto de definiciones y protocolos para construir e integrar *software*. Las *API* permiten que los productos y servicios se comuniquen con otros, sin necesidad de conocer los detalles de su implementación. Esto simplifica el desarrollo de aplicaciones y permite ahorrar tiempo y dinero. Las *API* otorgan flexibilidad, simplifican el diseño, la administración y el uso de las aplicaciones, y ofrecen oportunidades de innovación, lo cual es ideal al diseñar herramientas y productos nuevos [37].

Las *API* son un medio eficiente para conectar su propia infraestructura a través del desarrollo de aplicaciones nativas de la nube, pero también permiten compartir datos con clientes y otros usuarios externos. Las *API* públicas aportan un valor comercial único porque simplifican y amplían las conexiones con los socios y, además, pueden rentabilizar los datos (un ejemplo conocido es la *API* de *Google Maps*).

Por lo general, las *API* web usan *HTTP* para los mensajes de solicitud y proporcionan una definición de la estructura de los mensajes de respuesta. Estos mensajes suelen tener la forma de un archivo *XML* o *JSON*, que son formatos preferidos porque presentan los datos de manera que otras aplicaciones pueden manejar fácilmente.

Entre los tipos de *API* de servicios web más conocidos encontramos:

- **SOAP (Simple Object Access Protocol):** Las *API* diseñadas con *SOAP* usan *XML* para el formato de sus mensajes y reciben solicitudes a través de *HTTP* o *SMTP*. Con *SOAP*, es más fácil que las aplicaciones que funcionan en entornos distintos o están escritas en diferentes lenguajes comparten información.
- **REST (Representational State Transfer):** Las *API* web que funcionan con las restricciones de arquitectura *REST* se llaman *API RESTful*. Es un estilo de arquitectura para desarrollar aplicaciones en internet. Uno de los principios más importantes en *REST* es la separación entre cliente y servidor: el cliente no debe modificar directamente la lógica del servidor, solo puede

hacer consultas que el servidor decide cómo procesar; y el servidor no debe interferir con el funcionamiento del cliente, solo proporcionar información cuando el cliente la requiera. Las *API REST* usan interfaces para comunicar sistemas a través del protocolo *HTTP*.

CAPÍTULO 3

Estado del Arte

3.1. E-COMMERCE

3.1.1. Introducción

El término *e-Commerce* o comercio electrónico se podría definir como una tienda virtual que utiliza Internet como medio para llevar a cabo transacciones comerciales y conectar con los consumidores. Esta modalidad de compra y venta ha revolucionado el panorama empresarial, permitiendo a pequeñas empresas expandirse y acceder a nuevos mercados de manera eficiente. En esencia, un *e-Commerce* representa una aplicación web diseñada específicamente para facilitar la interacción entre vendedores y compradores, ofreciendo productos o servicios de forma accesible y conveniente a través de la red.

3.1.2. Evolución y tendencias actuales del e-Commerce

Aunque para muchos la compra online es una experiencia reciente, el concepto de *e-Commerce* se remonta a principios del siglo XX, con el surgimiento de la venta por catálogo en Estados Unidos alrededor de 1920. Esta innovación permitía adquirir productos sin necesidad de verlos físicamente ni de desplazarse a un establecimiento.

El verdadero boom del *e-Commerce* ocurrió en la década de 1990 con la popularización de Internet. Desde entonces, el término *e-Commerce* engloba todas las actividades de venta online, desde la exposición del producto hasta la compra, el uso y la fidelización del cliente. En esencia, el *e-Commerce* es el motor que impulsa el comercio digital, facilitando a los consumidores la adquisición de bienes y servicios a través de la red.

La pandemia del coronavirus, que comenzó a principios de 2020, ha acelerado aún más el crecimiento del *e-Commerce* a nivel global. Según un Estudio Anual realizado por el IEBS (Escuela de Negocios de la Innovación y los Emprendedores) sobre el *e-Commerce* en 2021, el 83,7 % de los internautas españoles indicaron que aumentaron su uso de Internet para realizar compras en comparación con el comercio tradicional [12]. Este dato resalta la creciente relevancia del *e-Commerce* en la sociedad contemporánea y su papel esencial en la economía digital.

El *e-Commerce* no solo ha representado una alternativa conveniente durante la pandemia, sino que también ha demostrado ser una solución integral para mantener las actividades cotidianas mientras se prioriza la seguridad y el bienestar de la sociedad.

Su capacidad para adaptarse a las necesidades cambiantes del consumidor y para proporcionar acceso a una amplia gama de productos y servicios de manera conveniente y segura ha reforzado su posición como pilar fundamental en la economía digital actual.

Esto ha reforzado su posición como un pilar fundamental en la economía digital actual. Según el CNMC(Comision nacional de los mercados y la competencia) 'el comercio electrónico superó en España los 20 mil millones de euros en el segundo trimestre de 2023, un 12,7 % más que el año anterior' [6].

Como se observa en la Figura 3.1, los sectores de actividad con mayores ingresos han sido las agencias de viajes y operadores turísticos, con el 11,2 % de la facturación total; el transporte aéreo, con el 6,2 %; y las prendas de vestir, en tercer lugar, con el 5,6 % .

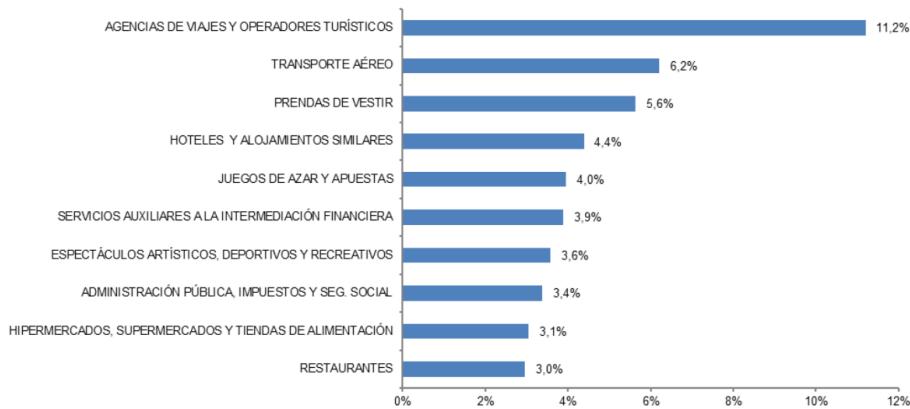


Figura 3.1: Ramas de actividad con mayor porcentaje de volumen de negocio del comercio electrónico.

Fuente: CNMC [6]

En el año 2023 en España, diversos sectores destacaron por su notable desempeño en ventas a través de plataformas online. A continuación, en la Figura 3.2 se presenta una gráfica que ilustra los principales e-Commerce que lideraron el mercado español en términos de volumen de ventas durante ese año. Esta representación visual nos ofrece una perspectiva clara sobre las preferencias de los consumidores y el impacto de ciertos sectores en el panorama del comercio electrónico en España.

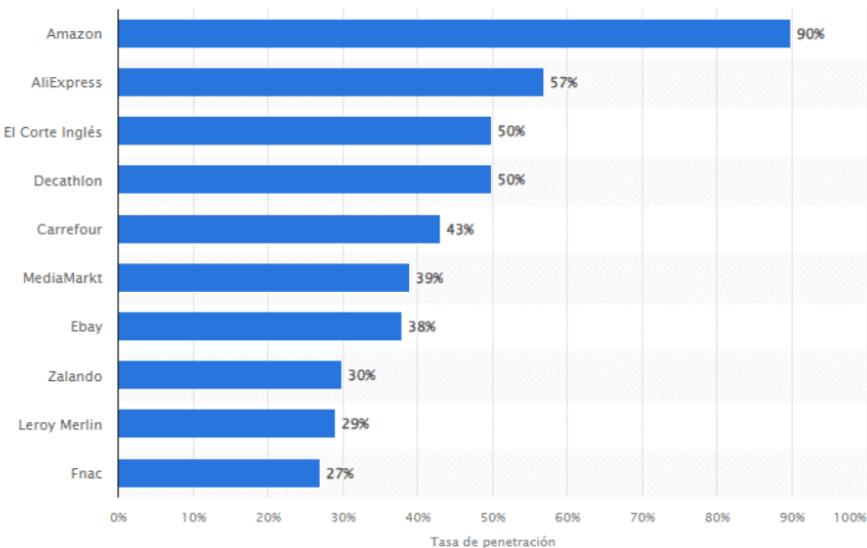


Figura 3.2: e-Commerce en España 2023. Fuente: Data Comunicación[7]

3.1.3. Tipos de e-Commerce

En la actualidad, el comercio electrónico se clasifica según los actores involucrados:

1. e-Commerce B2B (*Business to Business*):

El e-Commerce B2B implica transacciones comerciales realizadas entre empresas en entornos digitales, sin la participación directa de consumidores finales. Este modelo es común en negocios de venta al por mayor, distribución y servicios entre empresas. Algunos ejemplos de este tipo de e-Commerce son: *Alibaba*, *Solostocks* y *Tradekey*.

2. e-Commerce B2C (*Business to Consumer*):

El e-Commerce B2C se refiere a las transacciones comerciales entre una empresa y los consumidores finales. En este caso, los consumidores acceden a plataformas en línea para adquirir productos o servicios para uso personal. Algunos ejemplos de este tipo de e-Commerce son: *Amazon*, *Linio* y *MercadoLibre*.

3. e-Commerce C2C (*Consumer to Consumer*):

En el e-Commerce C2C, los agentes que participan en las transacciones son consumidores individuales. Este modelo se destaca por ser una plataforma para la compra y venta de productos de segunda mano entre particulares, fomentando la reutilización de productos y la economía circular. Algunos ejemplos de este tipo de e-Commerce son: *Facebook Marketplace*, *eBay* y *Airbnb*.

4. e-Commerce C2B (*Consumer to Business*):

El e-Commerce C2B involucra transacciones donde los consumidores son los proveedores de productos o servicios y las empresas son los compradores. Un ejemplo notable de este modelo es el programa de afiliados, donde los consumidores reciben compensación por promover productos o servicios de empresas en línea.

5. e-Commerce B2E (*Business to Employee*):

El e-Commerce B2E implica transacciones comerciales entre una empresa y sus empleados. Esta modalidad se utiliza para ofrecer beneficios como descuentos y ofertas especiales a los empleados, contribuyendo así a la motivación y satisfacción laboral.

6. e-Commerce G2C (*Government to Customer*):

El e-Commerce G2C involucra transacciones entre entidades gubernamentales y los consumidores. Permite a los ciudadanos realizar pagos de impuestos, multas y otros trámites administrativos en línea, agilizando y simplificando los procesos gubernamentales.

3.2. FRAMEWORKS Y TECNOLOGÍAS FRONTEND RELEVANTES

A la hora del desarrollo de un e-Commerce, la elección del *framework frontend* adecuado con el que se desarollara, desempeña un papel crucial en el desarrollo de aplicaciones web dinámicas y eficientes. Los *frameworks frontend* no solo proporcionan estructura y organización al desarrollo de interfaces de usuario, sino que también influyen significativamente en la experiencia del usuario final y en la escalabilidad del sistema.

En este apartado, se explorarán y compararán varios *frameworks frontend* populares utilizados en el desarrollo de plataformas e-Commerce. Se enfocará especialmente en destacar las características, ventajas y consideraciones clave de cada uno, con el objetivo de proporcionar una guía sólida para la selección y aplicación efectiva de tecnologías en proyectos e-Commerce.

3.2.1. React

React es una biblioteca de *JavaScript* de código abierto mantenida por *Facebook*, utilizada para crear interfaces de usuario que se renderizan en *HTML* (en la Figura 3.3 se muestra el logo de *React*).

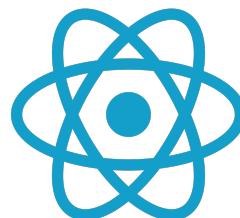


Figura 3.3: Logo React. Fuente: Página oficial de React [36]

Además de *Facebook*, muchas otras empresas utilizan *React* en producción, como *Airbnb*, *Atlassian*,

Bitbucket, Disqus, y Walmart. Facebook creó *React* para su propio uso y posteriormente lo liberó como software de código abierto.

Según encuestas anteriores de *Stack Overflow*, *React* y *Node.js* son dos de las tecnologías más utilizadas por los desarrolladores profesionales, con un 42,87 % de ellos utilizando *React* [42].

Algunas de las características destacadas de *React* son:

- **Declarativo:** Las vistas en *React* son declarativas, lo que significa que los programadores no necesitan gestionar los efectos de los cambios en el estado de la vista o los datos. Esto implica que no te preocupas por las transiciones o mutaciones en el *DOM* causadas por cambios en el estado de la vista. Ser declarativo hace que las vistas sean consistentes, predecibles, más fáciles de mantener y de entender.
- **Basado en componentes:** En *React*, se construyen componentes que luego se combinan para crear otros componentes que representan una vista o una página completa. Un componente encapsula el estado de los datos y la vista, o cómo se representa. Esto facilita la escritura y el razonamiento sobre toda la aplicación, dividiéndola en componentes y centrándose en una cosa a la vez.
- **Sin plantillas:** Muchos frameworks de aplicaciones web se basan en plantillas para automatizar la tarea de crear elementos *HTML* o *DOM* repetitivos. *React* utiliza un lenguaje de programación completo para construir elementos *DOM* repetitivos o condicionales.
- **Isomórfico:** React también puede ejecutarse en el servidor, lo que significa que el mismo código puede ejecutarse tanto en el servidor como en el navegador.

A pesar de sus numerosas ventajas, *React* también presenta algunas limitaciones a considerar:

- **Curva de aprendizaje inicial:** Puede requerir tiempo para familiarizarse con su sintaxis *JSX* y su enfoque en el flujo de datos unidireccional.
- **Elección de herramientas:** Dado que es una librería y no un framework completo, la selección de herramientas y bibliotecas adicionales para el desarrollo puede resultar abrumadora.

3.2.2. Angular

Angular es una plataforma y framework diseñado para crear aplicaciones de una sola página en el lado del cliente, utilizando *HTML* y *TypeScript* (en la Figura 3.4 se muestra el logo de *Angular*). Desarrollado por *Google*, *Angular* está completamente escrito en *TypeScript* e implementa funcionalidades básicas y opcionales como un conjunto de bibliotecas *TypeScript* que se pueden importar en tus aplicaciones. Su objetivo principal es potenciar las aplicaciones basadas en navegador con el patrón Modelo Vista Controlador (*MVC*), facilitando así el desarrollo y la prueba de aplicaciones web modernas [33].



Figura 3.4: Logo de *Angular*. Fuente: Página oficial de *Angular* [3]

Entre las características destacadas de *Angular* se encuentran:

- **Coherencia y reutilización del código:** *Angular* utiliza una estructura basada en componentes que facilita la reutilización y simplifica el desarrollo.
- **Fácil de aprender, usar y probar:** *Angular* está diseñado para ser accesible tanto para nuevos desarrolladores como para expertos, con herramientas robustas para la realización de pruebas.

- **Soporte de Google y comunidad activa:** Respaldado por *Google*, *Angular* cuenta con un sólido soporte técnico y una comunidad vibrante que contribuye con bibliotecas y recursos.
- **Integración perfecta y alta productividad:** *Angular* ofrece una integración fluida con otras herramientas y servicios, promoviendo una alta productividad en el desarrollo de aplicaciones web complejas.

Sin embargo, a pesar de sus numerosas ventajas, *Angular* también presenta algunas limitaciones y consideraciones que deben tenerse en cuenta:

- **Complejidad inicial:** Puede tener una curva de aprendizaje más pronunciada debido a su estructura completa y el uso de *TypeScript*.
- **Rendimiento:** Algunos desarrolladores argumentan que *Angular* puede ser más pesado en términos de carga inicial y rendimiento en comparación con otros *frameworks* más ligeros como *React*.

3.2.3. *Vue.js*

Vue es un *framework* progresivo diseñado para la construcción de interfaces de usuario (en la Figura 3.5 se muestra el logo de *Vue*). A diferencia de otros *frameworks* monolíticos, *Vue* está concebido para ser utilizado de forma incremental desde cero. La biblioteca central se centra exclusivamente en la capa de visualización, lo que facilita su integración con otras bibliotecas o proyectos existentes. Además, *Vue* es capaz de impulsar aplicaciones sofisticadas de una sola página cuando se combina con herramientas modernas y bibliotecas de apoyo [34].



Figura 3.5: Logo de *Vue.js*. Fuente: Página oficial de *Vue.js* [43]

Algunas de las características destacadas de *Vue* son:

- **Curva de aprendizaje sencilla:** Considerada la más fácil entre los *frameworks* más populares, *Vue* permite a los desarrolladores aprender y comenzar a usar rápidamente.
- **Uso directo de HTML:** *Vue* privilegia el uso directo de *HTML*, lo cual simplifica la familiarización y adopción para desarrolladores acostumbrados a tecnologías estándar como *HTML*, *CSS* y *JavaScript*.
- **Enfoque en HTML tradicional:** A diferencia de otras bibliotecas como *React*, que se centran más en la programación en *JavaScript* puro, *Vue* se destaca por su enfoque en el uso tradicional de *HTML*.

A pesar de sus numerosas ventajas, *Vue* también presenta algunas limitaciones a considerar:

- **Ecosistema más pequeño:** Aunque está creciendo rápidamente, *Vue* puede tener menos bibliotecas y herramientas de apoyo en comparación con *React* y *Angular*.
- **Menos respaldo corporativo:** Aunque tiene un creciente respaldo y una comunidad activa, *Vue* puede percibirse como menos respaldado por grandes empresas en comparación con *Angular* (*Google*) y *React* (*Facebook*).

3.3. DEVOPS EN EL DESARROLLO DE SOFTWARE

3.3.1. Introducción a DevOps

DevOps es más que una simple herramienta; es una filosofía que impulsa la colaboración entre equipos de desarrollo, operaciones e infraestructura para optimizar el proceso de creación de *software*. Se enfoca en la automatización, la integración continua y la entrega continua, con el objetivo de mejorar la eficiencia y la calidad del producto final. Esta práctica busca romper las barreras tradicionales entre desarrollo y operaciones, fomentando una cultura de colaboración y responsabilidad compartida. En esencia, *DevOps* es una forma inteligente de abordar el desarrollo de *software*, donde el desarrollo, las pruebas y el despliegue se consideran parte integral de la metodología. El resultado es una organización más ágil, capaz de entregar productos de alta calidad en menos tiempo.

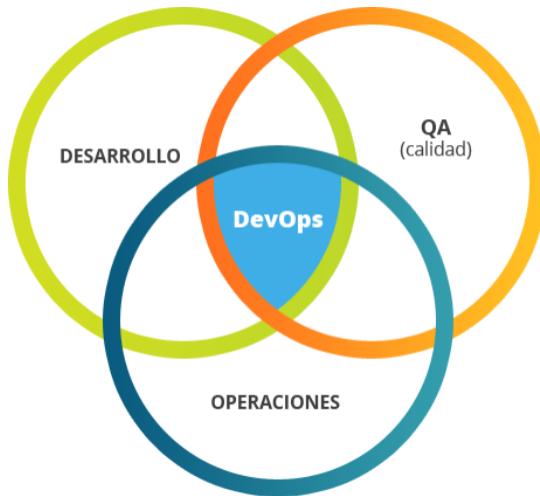


Figura 3.6: Intersección de Desarrollo, Operaciones y Calidad en el marco de *DevOps*. Fuente: Página Xeridia Devops[48]

3.3.2. Ciclo de vida del desarrollo de software

El ciclo de vida del desarrollo de *software*, conocido por sus siglas en inglés *SDLC* (*Software Development Lifecycle*), es un proceso estructurado y planificado que los equipos de desarrollo utilizan para crear y diseñar *software* de manera eficiente y con alta calidad. Este proceso abarca todas las etapas, desde la concepción hasta la finalización del *software*.

El *SDLC* describe una serie de tareas esenciales para desarrollar una aplicación de *software*. Este proceso atraviesa varias etapas, durante las cuales los desarrolladores agregan nuevas características y corrigen errores del *software*. A continuación, se ofrece una descripción general de las etapas del *SDLC* como muestra la Figura 3.7:

1. **Planificar:** En esta etapa inicial, se define el alcance del proyecto y se analizan los requisitos necesarios. El objetivo es comprender el propósito del proyecto, las necesidades del cliente y los resultados esperados.
2. **Diseño:** Durante la fase de diseño, los ingenieros de *software* evalúan los requisitos y determinan las mejores soluciones para desarrollar el *software*.
3. **Implementación:** En esta fase se lleva a cabo la construcción del *software*, generando el código fuente en el lenguaje de programación más adecuado para el proyecto.
4. **Pruebas:** El equipo de desarrollo realiza pruebas automáticas y manuales para detectar y corregir errores en el *software*. Las pruebas de calidad aseguran que el *software* cumple con los requisitos del cliente y no presenta fallos. Esta etapa a menudo se solapa con la fase de desarrollo, ya que muchos equipos prueban el código inmediatamente después de escribirlo.

5. **Despliegue:** Una vez que el *software* ha superado las pruebas, se implementa en el entorno elegido, ya sea pre-productivo o productivo, según las necesidades del cliente y el flujo de trabajo establecido.
6. **Mantenimiento:** En la fase de mantenimiento, el equipo se encarga de corregir errores, resolver problemas reportados por los usuarios y gestionar los cambios realizados en el *software*. Además, supervisa el rendimiento del sistema, la seguridad y la experiencia del usuario para identificar posibles mejoras.

En resumen, el proceso comienza con una idea de mejora o una nueva funcionalidad para el *software*. Después de un proceso de descubrimiento, análisis y refinamiento, los desarrolladores programan la funcionalidad. Una vez completada la programación y realizadas las pruebas en el entorno local del desarrollador, los cambios se despliegan en distintos entornos, que incluyen:

- **Desarrollo:** El entorno local del desarrollador, generalmente en su propio PC.
- **Integración:** Donde los programadores verifican los cambios realizados.
- **Pre-Producción:** Un entorno similar a producción.
- **Producción:** Donde se materializa el desarrollo y los usuarios finales pueden utilizar el software

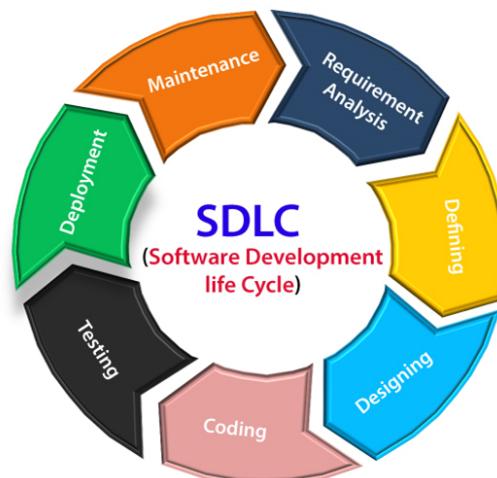


Figura 3.7: Ciclo de vida del desarrollo de software. Fuente: Página Xeridia Devops[48]

Los modelos más comunes del ciclo de vida del desarrollo de *software* en dos categorías principales:

- **Modelos de desarrollo secuencial:** Un modelo de desarrollo secuencial describe el proceso de desarrollo de *software* como un flujo lineal y ordenado de actividades. En este enfoque, cada fase del proceso de desarrollo debe comenzar solo después de que la fase anterior se haya completado. Aunque teóricamente no hay superposición de fases, en la práctica es beneficioso obtener retroalimentación temprana de la fase siguiente [22].
 - **Modelo Waterfall:** En este modelo, las actividades de desarrollo, como el análisis de requisitos, diseño, codificación y pruebas, se completan de manera secuencial. Las pruebas se realizan únicamente después de que todas las demás actividades de desarrollo hayan finalizado.
 - **Modelo V:** A diferencia del modelo Waterfall, el modelo V integra las pruebas a lo largo del proceso de desarrollo, aplicando el principio de pruebas tempranas. Además, asocia niveles de prueba con cada fase correspondiente del desarrollo, lo que refuerza la importancia de las pruebas tempranas. Aunque las pruebas se ejecutan de manera secuencial, puede haber cierta superposición en algunos casos.

Los modelos de desarrollo secuencial suelen entregar software con un conjunto completo de características, pero generalmente requieren varios meses o años para su entrega a las partes interesadas y usuarios.

- **Modelos de desarrollo iterativo e incremental:** El desarrollo incremental se basa en establecer requisitos, diseñar, construir y probar el sistema en partes. Esto significa que las características del *software* se desarrollan y entregan de manera incremental. El tamaño de estos incrementos puede variar, con algunos métodos produciendo incrementos más grandes y otros más pequeños. Los incrementos pueden ser tan pequeños como un único cambio en la interfaz de usuario o una nueva opción de consulta[22].

3.3.3. SDLC y DevOps

DevOps y los modelos de *SDLC* son altamente compatibles, ya que ambos se enfocan en mejorar la eficiencia, calidad y velocidad del desarrollo y entrega de software. *DevOps* provee herramientas y prácticas para integrar estas mejoras a lo largo de todo el ciclo de vida del software, independientemente del modelo específico de *SDLC* utilizado (*Agile*, *Waterfall*, etc.).

Los ingenieros de *DevOps* desempeñan un papel crucial en fases clave como la implementación, prueba y despliegue del *software*. Su enfoque principal se centra en la automatización y eficiencia. Automatizan estas fases para reducir la intervención manual, mejorando así la velocidad y calidad en la entrega del producto.

Es fundamental que todos los miembros de un equipo de desarrollo comprendan el *SDLC* y el papel de *DevOps* en él. Ya sean desarrolladores, *testers* o entusiastas de *DevOps*, todos contribuyen colectivamente para asegurar una entrega eficiente y de alta calidad de los productos.

- Las metodologías *DevOps* han revolucionado el desarrollo de *software* con mejoras significativas:
- **Colaboración:** Promueven la cooperación entre equipos y departamentos, involucrando a todos los *stakeholders* en cada etapa del ciclo de vida del *software*.
 - **Automatización:** Elimina tareas manuales repetitivas, mejorando la fiabilidad en procesos críticos como el despliegue del *software*. Sin automatización, los cambios se suben manualmente a diferentes entornos, lo que aumenta significativamente el riesgo de errores y fallos en el sistema.
 - **Integración continua (CI):** Automatiza la integración de cambios en el código fuente, asegurando la fusión regular de modificaciones y la detección temprana de errores.
 - **Entrega continua (CD):** Extiende la integración continua al automatizar el despliegue de aplicaciones en entornos de producción tras pasar pruebas satisfactorias, garantizando un control riguroso sobre la calidad del software desplegado.
 - **Escalabilidad y flexibilidad:** Adaptan el desarrollo a entornos ágiles y sistemas distribuidos, facilitando la adaptabilidad a las necesidades cambiantes y el despliegue en entornos complejos.
 - **Monitorización:** Fomentan la supervisión continua del rendimiento de aplicaciones y servidores para anticipar y resolver problemas rápidamente, reduciendo los tiempos de respuesta ante incidencias.

Estas metodologías no solo optimizan el proceso de desarrollo de *software*, sino que también fortalecen la colaboración interdepartamental, mejoran la eficiencia operativa y garantizan la calidad del producto final.

3.3.4. Integración Continua (CI)

Una vez que los desarrolladores completan su funcionalidad, es crucial cargarla en el repositorio de código. Este proceso no solo podría provocar errores, sino que también podría afectar lo que ya está implementado al agregar nuevas funcionalidades. En proyectos con múltiples desarrolladores

trabajando en el mismo repositorio, la coordinación y la integración cuidadosa son fundamentales para mantener la estabilidad y el funcionamiento correcto del *software*. De ahí la importancia del proceso de integración continua (*CI*), según Martin Fowler, un destacado autor, consultor y orador en el desarrollo de software. Él lo define como:

"La integración continua es una práctica en el desarrollo de software donde los miembros del equipo integran su trabajo frecuentemente, a menudo varias veces al día. Cada persona incorpora sus cambios al menos una vez al día, lo que resulta en múltiples integraciones diarias. Cada integración se verifica mediante una compilación automatizada que incluye pruebas y análisis para detectar errores de integración lo más rápido posible." [14]

Este enfoque no solo mejora la calidad del *software* al identificar problemas temprano, sino que también fomenta una colaboración eficiente y una entrega más rápida y confiable de nuevas funcionalidades.

Una vez que un desarrollador sube sus cambios al repositorio de código, se activa un proceso de integración continua que incluye la compilación, ejecución de pruebas automáticas y generación de artefactos. Si este proceso es exitoso, los cambios se integran en el repositorio central del software.

A continuación, se presentan algunos de los principales beneficios que aporta la integración continua:

- **Detección rápida de errores:** Permite identificar y corregir errores de manera inmediata, ya que el código se integra y se prueba frecuentemente, lo que reduce el costo y el tiempo de corrección.
- **Aumento de la productividad del equipo:** Automatización y ejecución inmediata de procesos, lo que permite a los desarrolladores enfocarse en tareas de mayor valor.
- **Monitorización continua de las métricas de calidad del proyecto:** Permite un seguimiento constante de la calidad del proyecto, asegurando que se cumplan los estándares establecidos.
- **Mejora en la calidad del código:** Facilita la aplicación de pruebas automatizadas en cada integración, garantizando que el código nuevo no introduzca fallos y cumpla con los estándares de calidad establecidos.
- **Despliegue más rápido y fiable:** Facilita el camino hacia la entrega continua y el despliegue continuo, permitiendo que las nuevas características y correcciones lleguen al usuario final más rápidamente y con menos riesgos.
- **Ciclo de desarrollo más rápido:** Acelera el ciclo de desarrollo al automatizar las pruebas y la integración, permitiendo que los desarrolladores se concentren en agregar valor y mejorar el producto.
- **Documentación automática:** Genera documentación automáticamente, manteniendo un registro de los cambios y las pruebas realizadas, lo que mejora la mantenibilidad del proyecto.

Este entorno siempre contiene la última versión del código (no necesariamente la última versión productiva) y es vital para verificar la funcionalidad entregada y prevenir errores antes de cualquier despliegue en producción.

3.3.5. Despliegue Continua (*CD*)

Si se implementa correctamente, se puede ejecutar un proceso de despliegue continuo (*Continuous Deployment*), de modo que la nueva versión de la aplicación, con los cambios recientes incluidos, esté disponible para ser probada en un entorno de integración.

El despliegue continuo es una práctica que busca entregar o desplegar de manera ágil, fiable y automática todos los cambios nuevos subidos al repositorio central en un entorno específico. Este proceso es una extensión de la integración continua y, en *DevOps*, ambos suelen ir de la mano.

Una buena práctica en CD es realizar despliegues predecibles, automáticos y rutinarios. Aunque se rumorea que la entrega frecuente disminuye la calidad del *software*, esto no es cierto. No es necesario renunciar a la calidad y estabilidad en nuestros proyectos al hacer despliegues frecuentes. Automatizando estos procesos, logramos un sistema de despliegue que siempre funcionará de manera consistente, manteniendo la calidad mientras mejora los tiempos de entrega.

3.3.6. Casos Reales de Implementación de DevOps

En la actualidad, podemos observar que numerosas industrias y sectores diferentes están adoptando procesos *DevOps*, lo que les ha proporcionado una multitud de beneficios y un gran impacto positivo en sus objetivos comerciales. Varias grandes empresas exitosas han implementado prácticas *DevOps* con notable éxito y algunas de ellas han compartido sus experiencias al respecto.

Por ejemplo, *Netflix* ha sido un pionero en la implementación de *DevOps* a gran escala, utilizando herramientas como Spinnaker para la entrega continua en múltiples nubes. Esto les permite mantener una alta disponibilidad y escalar eficientemente sus servicios de streaming globalmente, como se detalla en el artículo '*Global Continuous Delivery with Spinnaker*' disponible en el blog técnico de Netflix [30].

Amazon Web Services (AWS) ha transformado la industria con su enfoque en la nube y la adopción de prácticas *DevOps*. Utilizan herramientas como *AWS CloudFormation* y *Amazon ECS* para implementar y gestionar aplicaciones de manera ágil y segura. Más información se puede encontrar en el artículo de *Amazon* [2].

Etsy, un mercado en línea, utiliza *DevOps* para habilitar el despliegue frecuente y seguro de cambios. Utilizan prácticas de integración y entrega continua para mejorar la eficiencia operativa y la experiencia del usuario, como se menciona en el artículo disponible en [16].

3.4. HERRAMIENTAS POPULARES EN DEVOPS

3.4.1. Control de Versiones

Los sistemas de control de versiones son herramientas de *software* que ayudan a los equipos de *software* a gestionar los cambios en el código fuente a lo largo del tiempo.

Existen varias herramientas de control de versiones disponibles, cada una con sus características y funcionalidades únicas.

3.4.1.1. Git

Git es un sistema de control de versiones distribuido y gestor de código fuente diseñado para manejar proyectos de cualquier tamaño con rapidez y eficiencia (en la Figura 3.8 se muestra el logo de *Git*). Permite a múltiples desarrolladores trabajar simultáneamente en un mismo proyecto sin sobrescribir los cambios de los demás, fomentando así la colaboración, la integración continua y la implementación continua. *Git* registra los cambios en archivos de código fuente y otros documentos, permitiendo a los equipos revertir a versiones anteriores, comparar cambios a lo largo del tiempo y coordinar el trabajo entre diferentes ramas de desarrollo [25].



Figura 3.8: Logo *Git*. Fuente: Página oficial de *Git* [17]

Algunas características clave de *Git* incluyen:

- **Control de versiones:** *Git* ayuda a rastrear los cambios, permitiéndote revertir a estados anteriores si es necesario.

- **Colaboración:** Permite a varios desarrolladores trabajar en un proyecto simultáneamente sin interferir entre sí.
- **Copia de seguridad:** Todo el historial del proyecto se guarda en un repositorio *Git*, proporcionando una copia de seguridad completa de todas las versiones.
- **Ramificación y fusión:** El modelo de ramificación de *Git* permite experimentar con nuevas funciones o correcciones de errores de forma independiente al proyecto principal.
- **Proyectos de código abierto:** La mayoría de los proyectos de código abierto utilizan *Git* para el control de versiones, facilitando contribuciones externas.
- **Estándar de la industria:** *Git* es ampliamente adoptado en la industria del *software*, siendo una habilidad esencial para los desarrolladores.

En la actualidad, *Git* es el sistema de control de versiones más ampliamente utilizado a nivel mundial. Es un proyecto de código abierto con un desarrollo activo, iniciado por *Linus Torvalds* en 2005, conocido por crear el *kernel* del sistema operativo *Linux* [5].

A pesar de sus numerosas ventajas y amplia adopción, *Git* presenta ciertas desventajas que pueden afectar su idoneidad para algunos proyectos. Es importante considerar estos aspectos al evaluar su uso en el entorno de desarrollo. A continuación, se detallan algunas de las principales desventajas de usar *Git*:

- **Curva de Aprendizaje:** *Git* puede ser complicado para los principiantes debido a su terminología y comandos no intuitivos, como *rebase*, *merge*, *branch*, *commit*, y *stash*.
- **Complejidad en la Gestión de Historias:** La flexibilidad de *Git* para reescribir el historial puede causar problemas si no se maneja correctamente, como la pérdida de cambios o conflictos complejos durante la fusión de ramas.
- **Rendimiento en Repositorios Grandes:** La gestión de repositorios muy grandes puede volverse lenta, especialmente en operaciones como *checkout* o *status*.
- **Requiere Configuración y Mantenimiento:** La configuración inicial y la gestión de *hooks* y servidores *Git* pueden requerir conocimientos técnicos significativos y mantenimiento continuo.

3.4.1.2. SVN

Apache Subversion, frecuentemente abreviado como *SVN* debido a su comando *svn*, es una herramienta de control de versiones de código abierto (en la Figura 3.9 se muestra el logo de *SVN*). Se comporta de manera similar a un sistema de archivos y se distribuye bajo una licencia de tipo *Apache/BSD*. Subversion utiliza un sistema de revisiones para almacenar los cambios en el repositorio. Solo guarda las diferencias entre revisiones (deltas), optimizando así el uso de espacio en disco. Permite a los usuarios crear, copiar y eliminar carpetas con la misma facilidad que en un disco duro local. Debido a su flexibilidad, es crucial aplicar buenas prácticas para una gestión adecuada de las versiones del *software*.



Figura 3.9: Logo SVN. Fuente: Wikipedia [47]

Algunas de las funcionalidades clave de *SVN* son:

- **Gestión de Archivos y Directorios:** *Subversion* rastrea directorios igual que archivos.

- **Operaciones Versionadas:** Las copias, eliminaciones y renombrados de archivos son versionados.
- **Metadatos:** Puede almacenar metadatos sobre archivos y directorios.
- **Revisión por Commit:** La revisión del *software* se hace por *commit*, no por archivo individual.
- **Creación de Ramas y Tags:** Permite la fácil creación de ramas y etiquetas, y la fusión de ramas con un sistema de seguimiento de cambios.
- **Bloqueo de Archivos:** Aunque no es necesario bloquear archivos para editarlos, es posible hacerlo.
- **Resolución de Conflictos:** Ofrece una resolución interactiva de conflictos, permitiendo mezclar el código de manera sencilla tanto desde la línea de comandos como desde interfaces gráficas.

Aunque *Subversion* es una herramienta robusta y confiable para el control de versiones, también tiene algunas limitaciones que pueden influir en su elección para ciertos entornos y proyectos. A continuación, se presentan algunas de las principales desventajas de usar *Subversion*:

- **Modelo Centralizado:** *Subversion* requiere una conexión constante al servidor central para realizar la mayoría de las operaciones, lo que puede ser problemático en entornos distribuidos o con conectividad limitada.
- **Rendimiento en Operaciones de Red:** Las operaciones en *Subversion* pueden ser más lentas debido a la necesidad de comunicación constante con el servidor central, afectando *commit*, *update*, y *log*.
- **Menos Flexibilidad en el Manejo de Ramas:** La gestión de ramas y fusiones en *Subversion* es menos flexible y más propensa a errores en comparación con *Git*, haciendo las operaciones de *branching* y *merging* más tediosas y complicadas.
- **Espacio de Almacenamiento:** *Subversion* puede consumir más espacio en disco debido a su manejo de metadatos y revisiones, ya que cada copia de trabajo incluye una carpeta *.svn* con metadatos.
- **Menor Soporte para Trabajo Sin Conexión:** Dado que *Subversion* es un sistema centralizado, su soporte para trabajo sin conexión es limitado, impidiendo a los desarrolladores realizar muchas operaciones críticas sin una conexión activa al servidor central.

3.4.2. Plataformas de Despliegue

3.4.2.1. Heroku

Heroku es uno de los servicios de Plataforma como Servicio (*PaaS*) más utilizados en la actualidad, especialmente en entornos empresariales, debido a su fuerte enfoque en simplificar el despliegue de aplicaciones (en la Figura 3.10 se muestra el logo de *Heroku*). En *Heroku*, solo necesitas especificar el lenguaje de *backend* que estás utilizando o la base de datos que necesitas, permitiéndote concentrarte únicamente en el desarrollo de tu aplicación.



Figura 3.10: Logo Heroku. Fuente: Página oficial Heroku [20]

Algunas de las características de Heroku son:

- **Soporte para múltiples lenguajes de programación:** *Node.js, Ruby, Java, Clojure, Scala, Go, Python, PHP.*
- **Ejecución mediante contenedores:** Las aplicaciones se ejecutan en contenedores conocidos como *Dynos*. Tipos de *Dynos*: *Web, worker o cron*.
- **Complementos:** Ofrece más de 200 complementos para ampliar las aplicaciones al instante.
- **Seguridad:** Incluye características de seguridad como *SSL*, autenticación y cumplimiento de *PCI*.

Y algunas desventajas de usar *Heroku* son:

- **Costo:** Es un servicio de pago.
- **Requisitos previos:** Para utilizar Heroku se requiere un nivel profesional de conocimientos en varias tecnologías y conceptos, incluyendo *Linux, Maven, Git, redes, HTML* y lenguajes de programación.
- **Documentación:** Fuera de la documentación oficial, existen pocas opciones de soporte relacionadas con *Heroku*. Además, toda la documentación oficial está en inglés y generalmente orientada a desarrolladores con experiencia.
- **Costo de los add-ons:** La mayoría de los *add-ons* son servicios empresariales escalables, pero suelen ser costosos, incluso en los paquetes iniciales.
- **Falta de control sobre el sistema operativo:** El control sobre el sistema operativo es prácticamente nulo.
- **Almacenamiento:** Debido a la arquitectura y ciclo de vida de los *Dynos*, es imposible almacenar archivos de forma persistente en el servidor.

3.4.2.2. *Render*

Render es una plataforma de alojamiento en la nube diseñada para desarrolladores y equipos de desarrollo de software (en la Figura 3.11 se muestra el logo de *Render*). *Render* facilita la construcción, implementación y escalado de aplicaciones [40].



Figura 3.11: Logo *Render*.Fuente Página oficial *Render* [41]

Entre las características destacadas de *Render* se encuentran [39]:

- **Facilidad de uso:** *Render* proporciona una experiencia optimizada para los desarrolladores, eliminando la complejidad de la infraestructura y permitiéndoles enfocarse en la creación de aplicaciones y productos.
- **Soporte para múltiples servicios:** La plataforma admite una variedad de servicios, incluyendo sitios estáticos, servicios web, trabajadores en segundo plano, trabajos cron y más.
- **Compatibilidad con múltiples entornos de ejecución:** Soporta múltiples entornos de ejecución como *Node.js, Docker, Python, Ruby, Elixir, Go, Rust* y *PHP*.
- **Fiabilidad y tiempo de actividad:** *Render* se distingue por su compromiso con la fiabilidad y el tiempo de actividad, y es utilizada por desarrolladores y empresas para gestionar aplicaciones, desde pequeños prototipos hasta grandes aplicaciones con cientos de servicios.

- **Costo:** *Render* proporciona un servicio gratuito para desplegar aplicaciones, lo que es beneficioso para desarrolladores y pequeños equipos que buscan probar la plataforma sin incurrir en costos.

Sin embargo, a pesar de sus numerosas ventajas, *Render* también presenta algunas limitaciones y consideraciones que deben tenerse en cuenta:

- **Dependencia de la plataforma:** Al utilizar *Render*, se depende de su infraestructura y servicios, lo que puede limitar el control sobre ciertos aspectos del entorno y la configuración.
- **Limitaciones de personalización:** A pesar de su flexibilidad, puede haber ciertas limitaciones en cuanto a la personalización y configuración avanzada que algunos desarrolladores podrían necesitar para aplicaciones muy específicas.
- **Requisitos técnicos:** Para aprovechar al máximo *Render*, es necesario tener conocimientos previos en desarrollo de software y en los lenguajes y entornos de ejecución que la plataforma soporta.

3.4.3. Selección de Tecnologías y Plataformas

Después de analizar el estado del arte y evaluar las características positivas y negativas de las tecnologías más populares disponibles en el mercado, procedemos a discutir la elección de cada una. En esta sección, definimos las tecnologías seleccionadas para el desarrollo de nuestra aplicación web de comercio electrónico, explicando las razones detrás de cada decisión.

Por un lado, nos inspiramos en el éxito de *Amazon*, uno de los *e-Commerce* más destacados a nivel global y en España, como referencia para el desarrollo de nuestro propio *e-Commerce*, orientado al modelo *B2C*. El objetivo principal de esta aplicación es potenciar el comercio en las zonas rurales, donde los productores locales actuarán como pequeñas empresas, ofreciendo directamente sus productos a los consumidores finales.

Para el desarrollo del *frontend*, hemos optado por *React*. Según encuestas recientes de *Stack Overflow*, *React* es la elección del 42,87 % de los desarrolladores profesionales [42], superando a *Angular* y *Vue*, gracias a su eficiencia y flexibilidad en la creación de interfaces de usuario dinámicas y rápidas.

En cuanto a las herramientas para los procesos *DevOps*, hemos seleccionado *Git* como repositorio principal. *Git* es ampliamente preferido sobre *SVN* por la comunidad de desarrolladores debido a su capacidad para gestionar versiones de código de manera eficiente y facilitar la colaboración en equipos distribuidos. Además, muchas herramientas que utilizan *Git* proporcionan funcionalidades para integración continua y despliegue continuo (*CI/CD*), elementos clave para implementar prácticas *DevOps* efectivas.

Finalmente, hemos decidido desplegar nuestra aplicación en *Render*, una plataforma de hospedaje que ofrece un servicio gratuito robusto y una interfaz intuitiva. Optamos por *Render* debido a su facilidad de uso en comparación con plataformas como *Heroku*, que además es de pago. Esto nos permitirá escalar la aplicación según sea necesario y mantenerla accesible para nuestros usuarios de manera sencilla y eficiente.

CAPÍTULO 4

Metodología

4.1. DESARROLLO ITERATIVO E INCREMENTAL

Como se ha mencionado anteriormente, el desarrollo incremental es un enfoque de desarrollo de *software* en el que la aplicación se construye y entrega en pequeñas partes o incrementos. Cada incremento proporciona una parte funcional del software y, con el tiempo, se combinan para formar un producto completo.

En este TFG se utilizará este modelo de desarrollo para llevar a cabo la implementación de la aplicación de *e-Commerce*. El *software* se irá entregando en partes funcionales denominadas incrementos.

El desarrollo incremental de una aplicación de comercio electrónico implica construir y entregar la aplicación en pequeñas partes funcionales o incrementos, con cada incremento añadiendo nuevas características o mejoras.

4.2. EL REPOSITORIO GITHUB

Como se ha explicado en este TFG se utilizará *Git* como repositorio principal donde estará subido todo el código del *e-Commerce*.

Dentro del universo de *Git* se encuentra *GitHub* (en la Figura 4.1 se muestra el logo de *GitHub*), una plataforma extremadamente popular y ampliamente adoptada por desarrolladores de todo el mundo.



Figura 4.1: Logo GitHub. Fuente: Página oficial de GitHub [18]

GitHub, como indica su propia página, es ‘una plataforma donde puedes almacenar, compartir y colaborar con otros usuarios en la escritura de código’ [19]. Fue adquirida por *Microsoft* en junio de 2018.

Almacenar tu código en un repositorio en *GitHub* te permite [19]:

- Presentar o compartir tu trabajo.
- Seguir y administrar los cambios en el código a lo largo del tiempo.
- Permitir que otros usuarios revisen el código y hagan sugerencias para mejorarlo.
- Colaborar en un proyecto compartido, sin preocuparte de que los cambios afecten el trabajo de los colaboradores antes de que esté listo para integrarlos.

- El trabajo colaborativo, una de las características fundamentales de *GitHub*, es posible gracias al software de código abierto *Git*, en el cual se basa *GitHub*.

4.2.1. Git Flow

Git Flow es un modelo de desarrollo basado en *Git* que establece una serie de convenciones para el nombrado y uso de las ramas. Este enfoque define claramente cómo y cuándo crear y fusionar ramas, facilitando la gestión de versiones y el desarrollo colaborativo en proyectos de software. Este modelo surge de un artículo de *Vincent Driessen* en 2010.

En el repositorio *Git*, se tendrá un repositorio central llamado *origin* que contendrá todos los desarrollos realizados por los miembros del equipo. Los desarrolladores deben realizar una *pull* para traer todos los cambios desde "*origin*".^a su repositorio local. Una vez que terminen sus desarrollos, harán un *push* para subir los cambios de su repositorio local al repositorio "*origin*". Como se puede apreciar en la Figura 4.2.

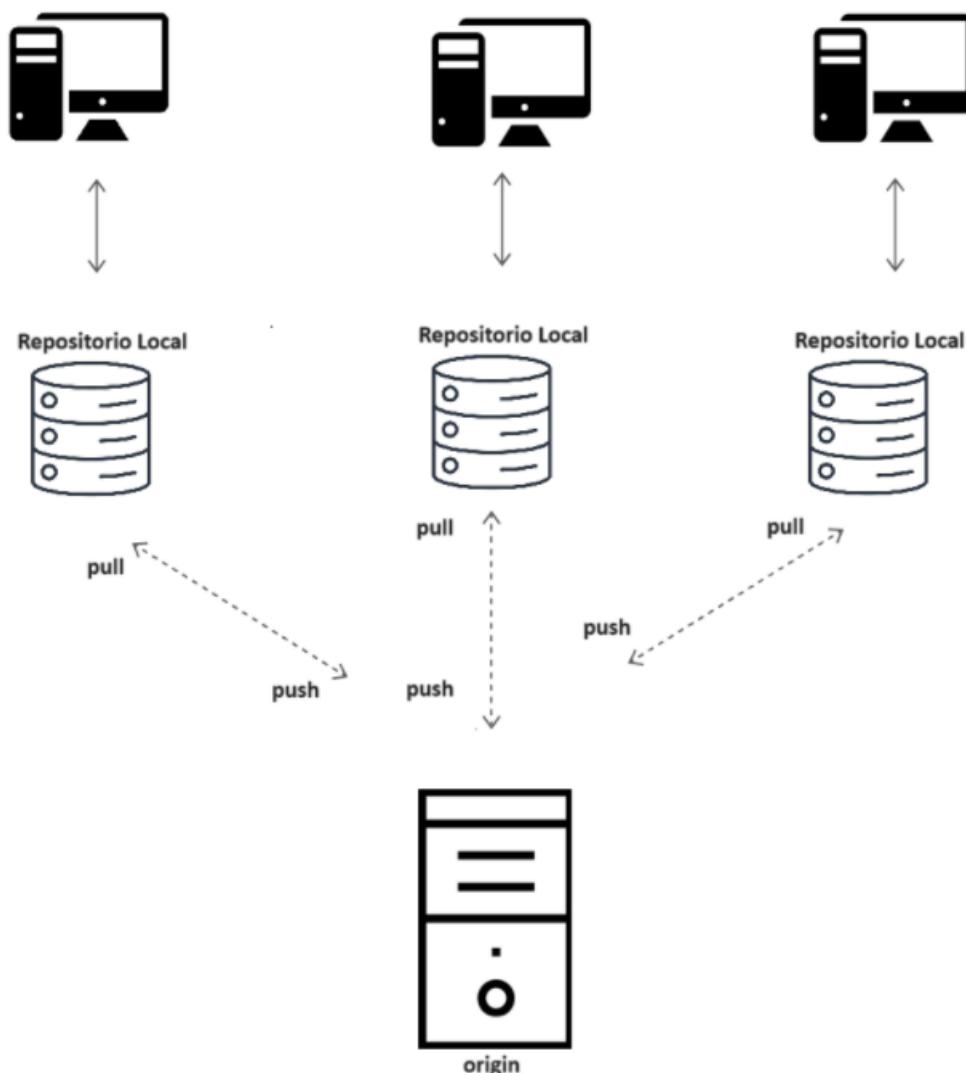


Figura 4.2: Representación GitFlow. Fuente: Elaboración propia

Las principales ramas en Git Flow son:

- **Master:** Rama que contiene el código de producción. Según esta metodología, todo el código en la rama master debe estar siempre listo para ser desplegado en producción sin problemas.

Esto asegura que la versión en producción sea estable y funcional en cualquier momento.

- **Develop:** Rama que recoge el trabajo de desarrollo y es la base para nuevas funcionalidades. En ella estarán los cambios listos que serán entregados. Cuando el código en la rama *develop* está en un estado estable y preparado para una nueva versión, todos los cambios se fusionan en la rama *master* y se etiquetan con un número de versión. Esto garantiza que cada vez que se fusionan cambios en *master*, se crea una nueva versión de producción. Además, se puede configurar un sistema de integración y despliegue continuo que compile y despliegue automáticamente el *software* en producción cada vez que se realice un *commit* en la rama *master*.
- **Feature:** Ramas dedicadas al desarrollo de nuevas funcionalidades. Estas ramas se crean a partir de *develop* y existen mientras se desarrolla la funcionalidad. Una vez finalizada, se fusionan de nuevo en *develop* y se eliminan. La convención del nombre de estas ramas es *feature/nombre-de-la-funcionalidad*.
- **Release:** Ramas usadas para preparar versiones antes de su lanzamiento a producción. Estas ramas se crean a partir de *develop* cuando el código en *develop* está listo para ser desplegado como una nueva versión. En las ramas de *release* se pueden realizar ajustes finales y preparar la versión para el despliegue. Una vez finalizada la preparación, la rama de *release* se fusiona tanto en *master* como en *develop*, y se etiqueta con un número de versión.
- **Hotfix:** Ramas utilizadas para corregir rápidamente errores críticos en producción. Estas ramas se crean a partir de *master* y permiten abordar problemas urgentes en la versión en producción. Una vez aplicadas las correcciones, se fusionan tanto en *master* como en *develop* para mantener la consistencia entre ambas ramas.

GitHub proporciona un flujo de trabajo estandarizado a través de las *pull requests* (peticiones de fusión). Este proceso está diseñado para equipos y proyectos que realizan despliegues continuos. Una vez que un desarrollador completa su trabajo en una rama *feature*, crea una *pull request* para fusionar su trabajo con la rama *develop*. Antes de que esta *pull request* sea fusionada con *develop*, debe someterse a una revisión de código por parte del equipo. Esta revisión no solo asegura la calidad del código, sino que también facilita la transferencia de conocimientos y promueve la cohesión dentro del equipo.

Durante la revisión de código, otros miembros del equipo pueden hacer comentarios, sugerencias o solicitar cambios en el código propuesto. Esto permite detectar y corregir problemas antes de integrar el código en la rama principal de desarrollo. Además, *GitHub* proporciona herramientas para discutir los cambios propuestos, lo que facilita la comunicación y colaboración entre los desarrolladores.

Una vez aprobada la *pull request* y realizados los ajustes necesarios, se procede a fusionar los cambios en *develop*. Este proceso garantiza que el código integrado cumpla con los estándares de calidad establecidos por el equipo, asegurando así la estabilidad y fiabilidad del *software*.

4.2.2. GitHub Actions

GitHub Actions es una plataforma de integración y entrega continua (*CI/CD*) que facilita la automatización de la compilación, prueba y despliegue de código. Permite configurar flujos de trabajo personalizados para verificar cada solicitud de cambio en tu repositorio y desplegar las actualizaciones fusionadas en producción [10].

GitHub ofrece máquinas virtuales con sistemas operativos *Linux*, *Windows* y *macOS* para ejecutar estos flujos de trabajo, además de permitir el uso de tus propios servidores auto-hospedados en tu infraestructura local o en la nube.

Algunas de las características principales de *GitHub Action* son:

- **Automatización de Flujos de Trabajo:** Define flujos de trabajo automatizados que responden

a eventos en el repositorio (*commits, pull requests, issues*) usando archivos *YAML* (*.github/workflows*).

- **Ejecutores y Entornos:** Ejecuta trabajos en máquinas virtuales proporcionadas por *GitHub* (*Linux, Windows, macOS*) o en servidores propios (auto-hospedados).
- **Acciones:** Componentes básicos de los flujos de trabajo que realizan tareas específicas (ejecutar *scripts*, instalar dependencias, compilar código, realizar pruebas, desplegar aplicaciones) con un *marketplace* de acciones predefinidas.
- **Integración con GitHub:** Acceso fácil a eventos del repositorio, secretos y servicios externos, mejorando la colaboración y automatizando tareas comunes de desarrollo.
- **Monitorización y Seguridad:** Herramientas para monitorear y solucionar problemas en los flujos de trabajo, manejo seguro de credenciales y datos sensibles.

GitHub Actions proporciona una solución flexible y completa para la automatización de tareas de desarrollo y despliegue, con una integración perfecta en el ecosistema de *GitHub*.

4.3. LENGUAJES DE PROGRAMACIÓN EN REACT

4.3.1. Introducción

Después de revisar las distintas tecnologías *frontend* en el mercado actual, se ha decidido centrar el TFG en la tecnología *React*. En este contexto, es crucial hablar de los diferentes lenguajes utilizados con esta tecnología. *React* es una biblioteca *JavaScript*, por lo que este será el lenguaje principal. Además, *CSS* y *HTML* son fundamentales, ya que constituyen la base estándar y los bloques de construcción básicos de cualquier aplicación web, incluidas aquellas desarrolladas con *React*.

4.3.2. JavaScript

JavaScript, ideado como un lenguaje de servidor por *Brendan Eich* mientras trabajaba en *Netscape Corporation*, debutó en *Netscape Navigator 2.0* en septiembre de 1995. Su éxito fue inmediato, con *Internet Explorer 3.0* adoptando la compatibilidad en agosto de 1996 bajo el nombre de *JScript*.

En noviembre de 1996, *Netscape* colaboró con *ECMA International* para estandarizar *JavaScript*. Desde entonces, *JavaScript* estandarizado es conocido como *ECMAScript* y su especificación, *ECMA-262*, se ha actualizado regularmente, alcanzando su décima edición (*ES2019*) en junio de 2019. Se puede encontrar toda información detallada y actualizada de este lenguaje hasta fecha en <https://ecma-international.org/>.

JavaScript, frecuentemente abreviado como *JS* (ver Figura 4.3), es un lenguaje ligero, interpretado y orientado a objetos, con funciones de primera clase. Es dinámico, multiparadigma y basado en prototipos, admitiendo estilos de programación orientados a objetos, imperativos y funcionales.

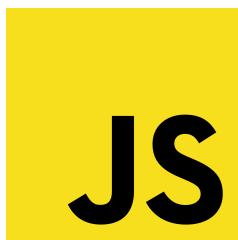


Figura 4.3: Logo *JavaScript*. Fuente: Wikipedia [46]

Aunque inicialmente sobresale como el lenguaje principal en el desarrollo *frontend*, donde se utiliza para manipular el contenido de las páginas web a través del *DOM*, *JavaScript* también desempeña un papel crucial en el lado del servidor.

Para interpretar *JavaScript* en el lado del cliente, los navegadores modernos integran motores como *V8* (*Chrome*, *Opera*, *Edge*) y *SpiderMonkey* (*Firefox*). Mientras que en el lado del servidor, *Node.js* se destaca como el entorno preferido para ejecutar *JavaScript* fuera del navegador.

4.3.3. *HTML*

HTML (*HyperText Markup Language*) (en la Figura 4.4 se muestra el logo de *HTML*) es el componente fundamental de la Web, ya que define la estructura y el significado del contenido en las páginas web [23].



Figura 4.4: Logo HTML. Fuente: Wikipedia [45]

El *Lenguaje de Marcado de Hipertexto* (*HTML*) es el código utilizado para estructurar y desplegar una página web y sus contenidos. Este estándar es gestionado por el World Wide Web Consortium (*W3C*), una organización dedicada a la estandarización de la mayoría de las tecnologías asociadas a la web.

Un elemento *HTML* se distingue de otro texto en un documento mediante etiquetas, que consisten en el nombre del elemento rodeado por '<' y '>'. El nombre de un elemento dentro de una etiqueta no distingue entre mayúsculas y minúsculas, por lo que se puede escribir en mayúsculas, minúsculas o una mezcla de ambas. Por ejemplo, la etiqueta <title> se puede escribir como <Title>, <TITLE> o de cualquier otra forma.

4.3.4. *CSS*

CSS (*Cascading Style Sheets*) (en la Figura 4.5 se muestra el logo de *CSS*) es el lenguaje de estilos utilizado para definir la presentación de documentos *HTML* o *XML* [24]. *CSS* especifica cómo debe mostrarse el contenido estructurado en diferentes medios, ya sea en pantalla, en papel, a través del habla o en otros formatos.



Figura 4.5: Logo CSS. Fuente: Wikipedia [44]

CSS se emplea para diseñar y estilizar páginas web, permitiendo modificar la fuente, el color, el tamaño y el espaciado del contenido, dividirlo en múltiples columnas, o agregar animaciones y otros efectos decorativos. Al igual que *HTML*, *CSS* es gestionado por el World Wide Web Consortium (*W3C*), una organización dedicada a la estandarización de tecnologías web.

4.4. BACKEND DE LA APLICACIÓN

4.4.1. Introducción

Una vez seleccionada la tecnología *frontend* ahora se procede a seleccionar el lenguaje de programación del *backend*. En el mercado actual se encuentran diversos lenguajes como los siguientes:

- **Python:** Python es conocido por su sintaxis clara y legible, lo que facilita el desarrollo rápido y la mantenibilidad del código. Es ampliamente utilizado en aplicaciones web y tiene una gran cantidad de frameworks como *Django* y *Flask* que facilitan la creación de *APIs* y servicios web.
- **JavaScript:** *Node.js* permite ejecutar *JavaScript* en el servidor. Es conocido por su alto rendimiento y escalabilidad, ideal para aplicaciones de tiempo real y *APIs*. Es particularmente beneficioso cuando se utiliza junto con frameworks como *Express.js*.
- **Java:** Java es conocido por su portabilidad y robustez. Es ampliamente utilizado en aplicaciones empresariales y sistemas distribuidos debido a su rendimiento y seguridad. Frameworks como *Spring* facilitan el desarrollo de *APIs RESTful* y servicios web.
- **C# :** C# es el lenguaje principal para el desarrollo en el framework *.NET* de Microsoft. Es ampliamente utilizado en aplicaciones empresariales y ofrece robustez, escalabilidad y seguridad. *ASP.NET* facilita la creación de *APIs* y servicios web.

Dado que el *frontend* de la aplicación se desarrollará con *React*, una biblioteca de *JavaScript* ampliamente reconocida, se ha decidido utilizar *JavaScript* también para el *backend*. Esta elección permite mantener un ecosistema coherente y consistente. Al utilizar *JavaScript* tanto en el *frontend* (*React*) como en el *backend* (*Node.js* con *Express*), simplificamos el desarrollo, la configuración y la gestión del proyecto.

Además, *JavaScript* es uno de los lenguajes más populares y ampliamente utilizados en el mundo actualmente. Su adopción ha crecido significativamente gracias a mejoras en el rendimiento y la disponibilidad de *APIs* modernas en los navegadores. Como muestra la Figura 4.6 *Stack Overflow*, la principal comunidad de desarrolladores, *JavaScript* fue elegido como la tecnología más popular en 2023.

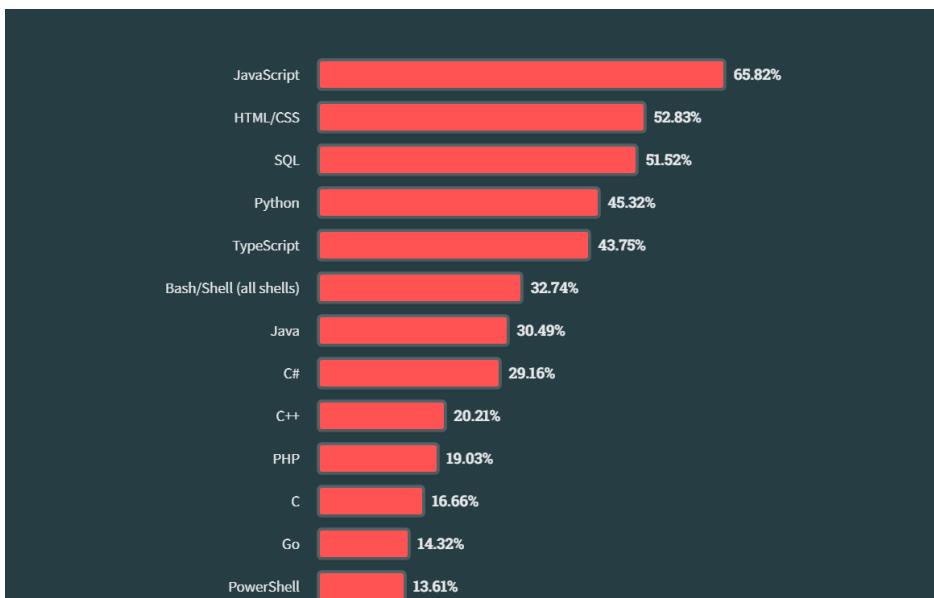


Figura 4.6: Tecnologías más populares en 2023. Fuente: Página Stack Overflow [42]

4.4.2. Node.js

Node.js (en la Figura 4.7 se muestra el logo de *Node*) es un entorno de tiempo de ejecución de *JavaScript* (de ahí su terminación en *.js*, haciendo alusión al lenguaje *JavaScript*). Este entorno de tiempo de ejecución en tiempo real incluye todo lo necesario para ejecutar un programa escrito en *JavaScript* [32]. *Node.js* está basado en el motor de *JavaScript V8* de *Chrome*. Aunque algunas personas creen que *Node.js* fue diseñado solo para el navegador y no es apto para el lado del servidor, líderes de la industria como *Netflix*, *Uber* y *LinkedIn* lo han elegido para ejecutar el backend de sus aplicaciones.



Figura 4.7: Logo *Node.js*. Fuente: Página oficial de *Node.js* [31]

En un navegador, puedes cargar varios archivos *JavaScript*, pero necesitas una página *HTML* para hacerlo. No puedes hacer referencia a otro archivo *JavaScript* desde un archivo *JavaScript* directamente. En cambio, con *Node.js*, no existe una página *HTML* que lo inicie todo. En ausencia de una página *HTML* adjunta, *Node.js* utiliza su propio sistema de módulos basado en *CommonJS* para reunir múltiples archivos *JavaScript*.

npm es el administrador de paquetes predeterminado para *Node.js*. Puedes utilizar *npm* para instalar bibliotecas (paquetes) de terceros y gestionar las dependencias entre ellas. El registro *npm* (www.npmjs.com) es un depósito público de todos los módulos publicados por personas con el fin de compartirlos. Como se muestra en la gráfica de la Figura 4.8, en 2023, *npm* lidera la lista de módulos o repositorios de paquetes, por lo que es el repositorio más grande.

Module Counts

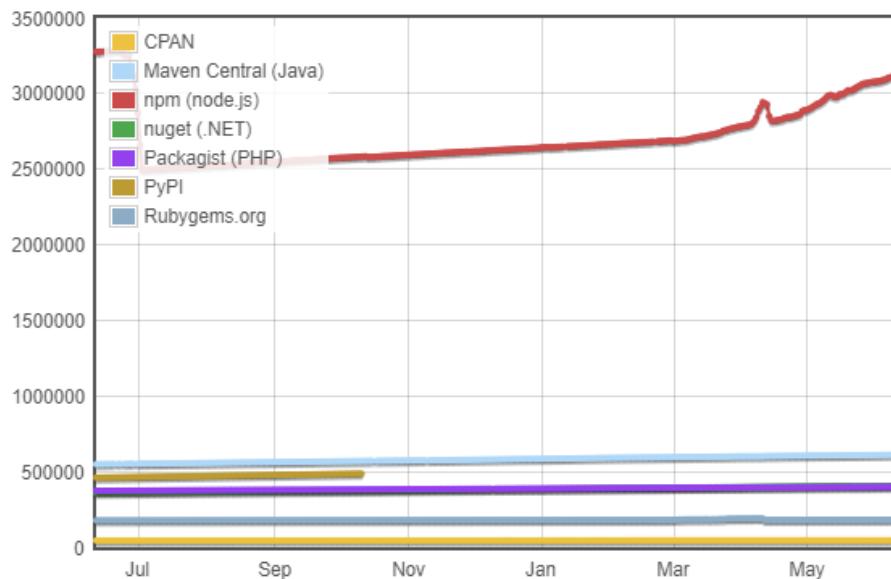


Figura 4.8: Número de módulos en varios idiomas. Fuente: Página Module Counts [27]

4.4.3. MERN Stack

Una aplicación web se construye utilizando una combinación de tecnologías, que se conoce como *stack*. *MERN* (ver Figura 4.9) es una combinación de tecnologías de *JavaScript* de código abierto que

ha ganado popularidad en los últimos años. Este *stack* incluye:

- **Node.js y Express:** Proporcionan un entorno de ejecución de *JavaScript* en el lado del servidor y un servidor web creado en *Node.js* respectivamente. Estas tecnologías forman la base del lado del servidor de la aplicación web.
- **React:** Una biblioteca *frontend* que se utiliza para construir interfaces de usuario interactivas y dinámicas. *React* es una parte fundamental del *stack MERN* y se encarga de la parte *frontend* de la aplicación.
- **MongoDB:** Una base de datos *NoSQL* muy popular que se utiliza para el almacenamiento persistente de datos. *MongoDB* ofrece un esquema flexible y escalabilidad horizontal, lo que lo convierte en una opción atractiva para aplicaciones web modernas.



Figura 4.9: MERN Stack. Fuente: Página Devtechnosys [9]

Este *stack*, *MERN*, ha sido una opción muy popular para el desarrollo de nuevas aplicaciones web en los últimos años. Sin embargo, es importante tener en cuenta que estas tecnologías por sí solas no son suficientes para crear una aplicación web completa. Se requieren otras herramientas y bibliotecas para ayudar en el proceso de desarrollo y complementar las funcionalidades de *React*. Estas herramientas pueden incluir sistemas de gestión de estado como *Redux*, herramientas de enrutamiento como *React Router*, y muchas otras bibliotecas que proporcionan funcionalidades específicas para la aplicación.

4.4.3.1. Express

Node.js, por sí solo, proporciona un entorno de ejecución para *JavaScript*, aunque escribir un servidor web completo en *Node.js* directamente puede ser complicado y no siempre necesario. Para simplificar esta tarea, existe *Express*, un marco que permite definir rutas y especificar acciones a realizar cuando llega una solicitud *HTTP* que coincide con un patrón determinado. *Express* utiliza expresiones regulares para las especificaciones de coincidencia, lo que lo hace muy flexible.

El marco *Express* analiza automáticamente la *URL*, los encabezados y los parámetros de la solicitud *HTTP*. En la respuesta, proporciona funciones para determinar códigos de respuesta, configurar *cookies*, enviar encabezados personalizados, entre otros. Además, *Express* permite escribir *middleware* personalizado, que son fragmentos de código que se pueden insertar en cualquier ruta de procesamiento de solicitud/respuesta para implementar funciones comunes, como el registro o la autenticación.

4.4.3.2. MongoDB

MongoDB es una base de datos *NoSQL* orientada a documentos con un esquema flexible y un lenguaje de consulta basado en *JSON*. Muchas empresas modernas, incluidas *Facebook* y *Google*, utilizan *MongoDB* en producción debido a su capacidad de escalar horizontalmente y su flexibilidad en el modelado de datos.

MongoDB difiere de las bases de datos relacionales en varios aspectos clave. Es una base de datos *NoSQL*, lo que significa que no sigue el modelo relacional tradicional de tablas con filas y columnas.

En su lugar, almacena los datos en forma de documentos, que se organizan en colecciones. Esto permite un modelado de datos más flexible y evita la necesidad de una estricta estructura de esquema.

MongoDB también es conocido por su capacidad de escalar horizontalmente y su facilidad de uso gracias a su lenguaje de consulta basado en *JSON* y su compatibilidad con *JavaScript*. Además, su almacenamiento no requiere un esquema predefinido, lo que permite una flexibilidad adicional durante el desarrollo.

En resumen, *MongoDB* es una base de datos *NoSQL* orientada a documentos, conocida por su flexibilidad y capacidad de escala horizontal, que se utiliza comúnmente en aplicaciones modernas junto con *Express*, *React* y *Node.js*.

4.5. EL TESTING EN DEVOPS

4.5.1. Introducción

En el contexto de *DevOps*, las pruebas desempeñan un papel crucial para garantizar la calidad del *software* en todas las etapas del ciclo de vida del desarrollo y entrega. A menudo subestimadas en la documentación centrada en el desarrollo y operaciones, las pruebas son fundamentales para validar tanto la funcionalidad como los atributos no funcionales del sistema.

Según el *ISTQB* [22], las pruebas se clasifican en varios tipos, incluyendo:

- **Pruebas Funcionales:** Evalúan si el sistema realiza las funciones esperadas. Verifican la completitud, corrección y adecuación funcional del *software*.
- **Pruebas No Funcionales:** Evalúan atributos como eficiencia, usabilidad y seguridad, entre otros.

Además, se distinguen dos enfoques principales:

- **Pruebas de Caja Negra:** Centradas en el comportamiento externo del *software* sin considerar su estructura interna.
- **Pruebas de Caja Blanca:** Enfocadas en el código y la arquitectura interna del *software*.

En términos generales, las pruebas se dividen en tres niveles clave:

- **Unit Testing:** Pruebas aisladas que verifican funciones o métodos individuales. Como ejemplo se muestra el Bloque de Código 4.1.

Bloque de código 4.1: Ejemplo de código pruebas unitarias

```

1  // Función a testear
2  function sum(a, b) {
3      return a + b;
4  }
5
6  // Prueba unitaria utilizando Jest
7  test('sumar 1+2 es igual a 3', () => {
8      expect(sum(1, 2)).toBe(3);
9  });

```

- **Integration Testing:** Pruebas que verifican la interacción entre componentes del sistema. Como ejemplo se muestra el Bloque de Código 4.2.

Bloque de código 4.2: Ejemplo de código pruebas integración

```

1  describe('GET /api/users', () => {
2      it('devuelve todos los usuarios', async () => {
3          const response = await request(app)
4              .get('/api/users')
5              .expect('Content-Type', /json/)
6              .expect(200);
7      });

```

```

8         expect(response.body.length).toBeGreaterThanOrEqual(1);
9     });
10    });

```

- **End-to-End Testing (E2E):** Pruebas que validan la funcionalidad completa del sistema desde la perspectiva del usuario. Como ejemplo se muestra el Bloque de Código 4.3.

Bloque de código 4.3: Ejemplo de código pruebas e2e

```

1 describe('Interacción_básica_del_usuario', () => {
2   it('puede_iniciar_sesión', () => {
3     cy.visit('/login');
4     cy.get('input[name="email"]').type('usuario@example.com');
5     cy.get('input[name="password"]').type('password123');
6     cy.get('button[type="submit"]').click();
7     cy.url().should('include', '/home');
8   });
9 });

```

La Figura 4.10 se denomina la pirámide de pruebas, esta es una representación visual que enfatiza la distribución ideal de pruebas en diferentes niveles para asegurar la calidad del *software* de manera eficiente y efectiva. Estos niveles se representan en forma de pirámide, para destacar que las pruebas unitarias, siendo las más numerosas y rápidas, deben formar la base sólida. A medida que subimos en la pirámide, las pruebas se vuelven menos numerosas pero más amplias en su alcance y complejidad.

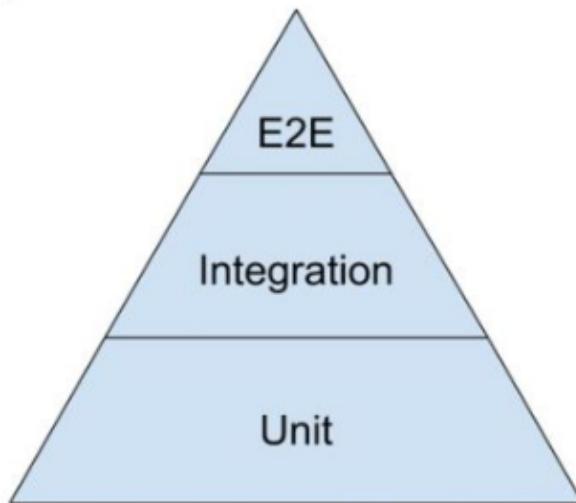


Figura 4.10: Pirámide niveles de prueba. Fuente Página Qanewsblog [35]

4.5.2. Herramienta de *Testing Cypress*

Cypress (ver Figura 4.11) es una herramienta de *testing* de última generación diseñada específicamente para la web moderna [8]. Esta plataforma permite la creación y ejecución de diversos tipos de pruebas, incluyendo *end-to-end*, integración y unitarias, para aplicaciones que operan en navegadores web.

Cypress se distingue por su enfoque en el *frontend*, especialmente orientado a aplicaciones web construidas con *frameworks* modernos de *JavaScript*. Es reconocida por su facilidad de uso y rápida curva de aprendizaje, registrando las acciones realizadas durante las pruebas y permitiendo una navegación temporal para depuración. Además, genera reportes detallados y cuenta con una documentación extensa y actualizada.



Figura 4.11: Logo de Cypress. Fuente Página Oficial de Cypress [8]

Entre sus principales ventajas se encuentran:

- **Versatilidad en tipos de pruebas:** Soporta pruebas *end-to-end*, de integración, de componentes y unitarias.
- **Facilidad de uso:** Fácil instalación y aprendizaje rápido.
- **Funcionalidades avanzadas:** Permite interactuar directamente con aplicaciones web sin la necesidad de drivers adicionales, y es compatible con múltiples navegadores como *Chrome*, *Firefox* y *Edge*. Incluso tiene una versión experimental para simular *Safari* en *iOS*.
- **Eficiencia y fiabilidad:** Proporciona pruebas más rápidas, sencillas y confiables.

En conclusión, en este TFG se utiliza Cypress en las pruebas *e2e* del *e-Commerce*. Ya que no solo se está optando por una herramienta de *testing* avanzada y efectiva, sino que también estamos asegurando la calidad y fiabilidad de nuestra aplicación web desde el inicio del desarrollo hasta su despliegue final. Cypress no solo simplifica las pruebas, sino que también eleva el estándar de calidad de nuestro producto, asegurando una experiencia óptima para nuestros usuarios.

4.6. ENTORNO DE DESARROLLO

4.6.1. Introducción

Una vez seleccionadas las distintas tecnologías que se utilizarán para desarrollar el *e-Commerce*, se procede al uso de herramientas para escribir el código. Este conjunto de configuraciones y herramientas se conoce como entorno local del desarrollador. En este entorno, un *IDE* (*Integrated Development Environment*) actúa como interfaz de desarrollo que facilita la codificación de la aplicación.

Entre los *IDE* más populares para trabajar con JavaScript se encuentran:

- **WebStorm:** *WebStorm* es un potente *IDE* desarrollado por *JetBrains* para trabajar con *JavaScript*, *HTML* y *CSS*. Ofrece un conjunto de herramientas avanzadas para desarrollar aplicaciones, depurar errores, editar código y realizar pruebas. Es un *software* de pago, pero es conocido por su robustez y funcionalidades completas.
- **Visual Studio Code:** *Visual Studio Code* es un *IDE* creado por *Microsoft* que se ha convertido en uno de los más populares del mercado para trabajar con *JavaScript*, *HTML* y *CSS*. Es de código abierto y su licencia es gratuita, lo que permite a los desarrolladores personalizarlo y extender sus funcionalidades mediante la instalación de plugins de terceros.
- **Atom:** *Atom*, desarrollado por *GitHub*, es otro *IDE* popular, aunque menos utilizado que los anteriores. *GitHub* ha anunciado que dejará de mantener *Atom*, lo que ha reducido su popularidad y uso.

En este TFG, se utilizará *Visual Studio Code* como *IDE* principal debido a su gratuidad y a que, al ser de código abierto, permite extender su funcionalidad mediante la instalación de plugins de terceros.

4.6.2. Visual Studio Code

Visual Studio Code es un editor de código fuente ligero pero poderoso diseñado para escritorios, compatible con *Windows*, *macOS* y *Linux*. Destaca por su soporte integrado para *JavaScript*, *TypeScript* y *Node.js*, y su extenso ecosistema de extensiones que cubre otros lenguajes y entornos de ejecución como *C++*, *C*, *Java*, *Python*, *PHP*, *Go* y *.NET*.

En la Figura 4.12 se puede apreciar la apariencia de Visual Studio Code.

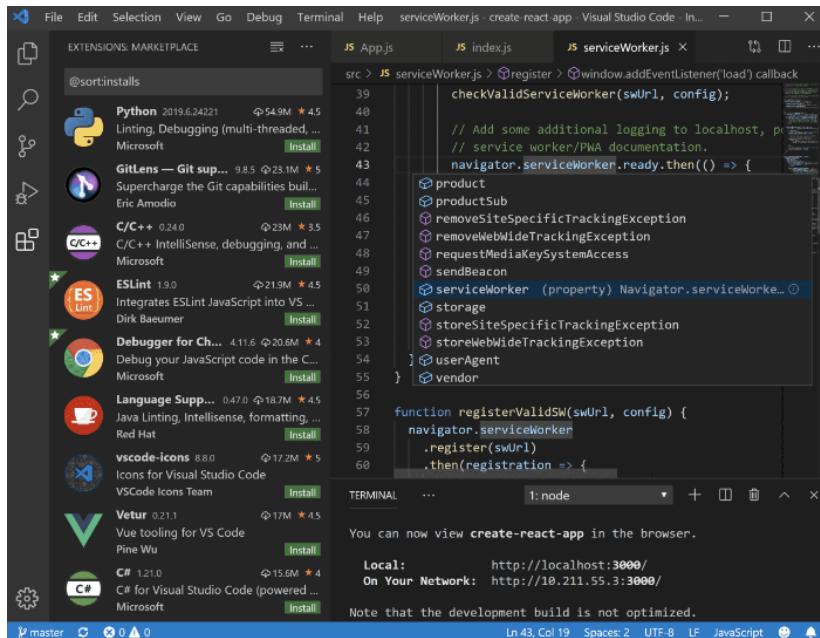


Figura 4.12: Visual Studio Code. Fuente Página Oficial de *Visual Studio Code* [26]

Algunas características clave de *Visual Studio Code* son:

- **Eficiencia y agilidad:** *Visual Studio Code* se distingue por su rapidez y eficiencia, incluso en equipos con recursos limitados. Su interfaz intuitiva permite comenzar a trabajar sin complicaciones.
- **Compatibilidad amplia:** Es multiplataforma, funcionando sin problemas en *Windows*, *Linux* y *MacOS*, lo que lo hace versátil y accesible para desarrolladores en diferentes entornos.
- **Herramientas integradas:** Ofrece potentes herramientas integradas para depuración y pruebas, facilitando un proceso de desarrollo fluido sin necesidad de salir del editor.
- **Integración con Git:** Su integración nativa con *Git* simplifica el control de versiones y optimiza el flujo de trabajo, permitiendo sincronización y gestión de ramas de manera eficiente.
- **Personalización y extensibilidad:** *Visual Studio Code* es altamente personalizable mediante numerosas extensiones que mejoran la integración con diversas tecnologías, adaptándose a las necesidades específicas de los desarrolladores.

Estos son algunos de los plugins que se utilizarán en *Visual Studio Code* para mejorar el trabajo de desarrollo:

- ***ESLint*:** Ayuda a mantener un código limpio y consistente, proporcionando advertencias y errores según las reglas de linting configuradas.
- ***Prettier*:** Formatea automáticamente el código para cumplir con un estilo predefinido, mejorando la legibilidad y consistencia del mismo.
- ***ES7 React/Redux/GraphQL/React-Native snippets*:** Provee snippets útiles para la creación rápida de diferentes tipos de componentes en *JavaScript*.
- ***Javascript (ES6) code snippets*:** Ofrece pequeños snippets para crear elementos *ES6* de manera eficiente.

CAPÍTULO 5

Desarrollo de la Aplicación

5.1. ARQUITECTURA DE LA APLICACIÓN

Como toda aplicación web moderna, el *e-commerce* tendrá su parte *frontend* y su parte *backend*. En el lado del *frontend*, que corresponde a la interfaz del cliente, se utilizará *React*. Aquí es donde el usuario interactúa con la aplicación a través del navegador. A medida que el usuario navega e interactúa con la aplicación, *React* envía peticiones al *backend* para obtener datos y realizar acciones. Estas peticiones usan el protocolo *HTTP*.

En el lado del *backend*, se utilizará una *API REST* implementada con *Node.js* y *Express*. Esta *API* se encarga de procesar las peticiones que recibe del *frontend* y realizar las consultas necesarias a la base de datos, que en este caso será *MongoDB*. Una vez que la base de datos responde a las consultas, la *API REST* se encarga de devolver los datos al *frontend* para que el usuario pueda ver los resultados de sus acciones.

Gracias a que todas las tecnologías utilizadas trabajan con *JavaScript*, todos los datos se enviarán en formato *JSON* (*JavaScript Simple Object Notation*). *JSON* permite representar objetos y estructuras complejas de una manera que es fácil de interpretar y manipular tanto en el *frontend* como en el *backend*.

El flujo descrito anteriormente se representa en la Figura 5.1.

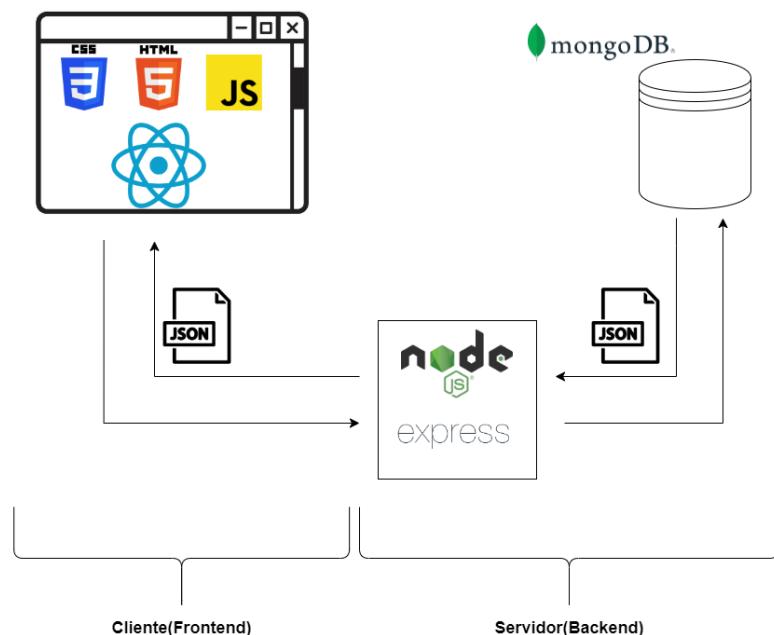


Figura 5.1: Arquitectura aplicación *MERN Stack*. Fuente: Elaboración propia

5.2. CONFIGURACIÓN INICIAL

Para desarrollar la aplicación se requiere la instalación previa de una serie de herramientas que se detallan a continuación:

- **El IDE:** Como se ha discutido anteriormente, el *IDE* seleccionado para el desarrollo será *Visual Studio Code*. Este se puede descargar desde su página web *Visual Studio Code*. En el sitio web, hay varias opciones disponibles según el sistema operativo de tu equipo. Una vez completada la descarga, procede a instalarlo en tu equipo.
- **Node.js:** Para descargar *Node.js*, visita su página oficial. Allí puedes seleccionar la versión de *Node.js* que deseas utilizar y el sistema operativo correspondiente. Se recomienda utilizar la versión más estable y recomendada. Una vez finalizada la descarga, instala *Node.js* en tu equipo. *Node.js* incluye *npm* por defecto, lo cual será de gran ayuda para manejar las dependencias de nuestros proyectos tanto en el *frontend* como en el *backend*.

Para comprobar que se ha instalado todo correctamente, ejecuta en la consola el siguiente comando del Bloque de Código 5.1.

Bloque de código 5.1: Comando ver versión de Node

```
1 node -v
```

Este comando devolverá la versión de *Node.js* instalada en el equipo. La figura 5.2 muestra la versión de *Node.js* que tiene instalada el equipo.

```
Usuario@DESKTOP-29GT8AM MINGW64 ~
$ node -v
v18.18.1
```

Figura 5.2: Consola mostrando versión de *Node.js*. Fuente: Elaboración propia

5.3. ESTRUCTURA DEL PROYECTO

El proyecto estará organizado en dos carpetas principales: *frontend* y *backend*. La carpeta *frontend* contendrá todo lo relacionado con la interfaz de usuario y la parte visible del proyecto, mientras que la carpeta *backend* albergará toda la lógica de servidor y la manipulación de datos.

Para comenzar el proyecto, se utiliza *npm*, el gestor de paquetes de *Node.js*. Primeramente, ejecutaremos el comando *npm init*, el cual nos guiará para crear el archivo *package.json*. Este archivo es fundamental, ya que en él se almacenará información clave del proyecto, como el nombre del mismo, la versión actual, la descripción, y las dependencias que se utilizarán, entre otros detalles.

El archivo *package.json* también nos permite configurar *scripts* personalizados que facilitan tareas como iniciar el servidor, compilar el *frontend*, o ejecutar pruebas, entre otras funciones. Es una parte esencial del ecosistema de desarrollo de *Node.js* y *JavaScript* en general.

Además, se crea un archivo llamado *.gitignore*. Este archivo de texto le indica a *Git* qué archivos o carpetas debe ignorar en el control de versiones del proyecto. Es importante incluirlo para evitar subir archivos innecesarios o sensibles al repositorio *Git*.

Con esta estructura y configuración básica, se estará listo para desarrollar y gestionar el proyecto de manera organizada y eficiente, asegurando que tanto el *frontend* como el *backend* funcionen de manera integrada y satisfactoria.

En resumen, la estructura del proyecto será como la que se muestra en la Figura 5.3.

```
 proyecto/
| -- frontend/
|   | -- ...
|   | -- ...
|
| -- backend/
|   | -- ...
|   | -- ...
|
| -- package.json
| -- .gitignore
```

Figura 5.3: Estructura del Proyecto. Fuente: Elaboración propia

5.4. DESARROLLO DEL FRONTEND

5.4.1. Configuración Inicial

Para comenzar un proyecto *frontend*, primero se ejecuta el comando del Bloque de Código 5.2 , lo cual genera el archivo *package.json*. Este archivo nos ayuda a gestionar todos los paquetes instalados en el proyecto, especialmente útil cuando se trabaja con muchas bibliotecas.

Bloque de código 5.2: Comando crear proyecto Node

```
1 npm init
```

Para iniciar un proyecto *React*, se ejecuta el comando del Bloque de Código 5.3. Este proceso descarga todas las dependencias necesarias y configura la estructura inicial del proyecto *React*. El proceso puede tardar unos segundos.

Bloque de código 5.3: Comando crear proyecto React

```
1 npx create-react-app nombre-del-proyecto
```

Para verificar que la configuración ha sido exitosa, se ejecuta el comando del Bloque de Código 5.4 . Este comando inicia el servidor de desarrollo de *React* y abre la nueva aplicación en el navegador. La Figura 5.4 muestra cómo lucirá la aplicación predefinida inicialmente.

Bloque de código 5.4: Comando crear proyecto React

```
1 npm start
```

Al guardar cambios en los archivos, la aplicación se actualizará automáticamente, facilitando así el desarrollo.

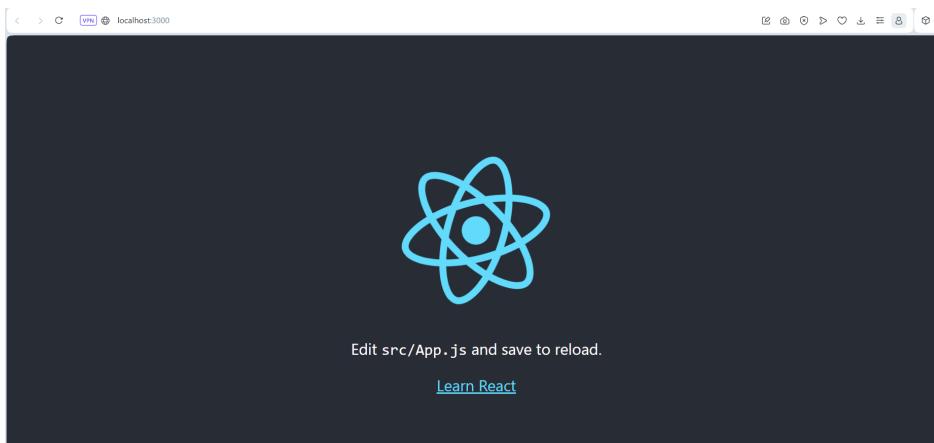


Figura 5.4: Página Web Piloto de React. Fuente: Elaboración propia

También se necesitarán las siguientes librerías para el correcto desarrollo del *frontend* del e-Commerce:

- **React Bootstrap:** *React Bootstrap* proporciona componentes listos para usar con un diseño elegante y consistente, basado en *Bootstrap*. Esto te permite construir interfaces de usuario atractivas rápidamente sin tener que diseñar desde cero.
- **Axios:** *Axios* es una librería de cliente *HTTP* basada en promesas que te facilita hacer solicitudes *HTTP* asíncronas (*GET*, *POST*, etc.) a servidores. Es más fácil de usar y tiene más funcionalidades que el *fetch API* nativo de *JavaScript*.
- **React Router DOM:** *React Router DOM* te permite gestionar la navegación y las rutas en tu aplicación de manera sencilla y efectiva. Es fundamental para crear aplicaciones de una sola página (*SPA*) donde la navegación entre diferentes vistas es fluida y manejada en el lado del cliente.

En el Bloque de Código 5.5 se muestran los comandos para instalar estas librerías.

Bloque de código 5.5: Comandos para instalar librerías React

```
1 npm install react-bootstrap bootstrap
2 npm install axios
3 npm install react-router-dom
```

5.4.2. Estructura del Proyecto

Después de ejecutar el comando para crear el proyecto *React*, este generará una estructura de directorios y archivos pre configurados (ver Figura 5.5) que facilitan el desarrollo de la aplicación. Aquí se tiene una descripción general de los directorios y archivos más importantes:

- **node_modules:** Este directorio contiene todas las dependencias del proyecto, incluidas las bibliotecas de *React* que *create-react-app* añade automáticamente y otras que se irán añadiendo conforme se avance en el desarrollo.
- **public:** Es el directorio público donde se colocan los archivos estáticos. En este directorio se encuentra el archivo *index.html* es el punto de entrada de la aplicación *React* y es aquí donde se monta la aplicación usando *<div id=root></div>*.
- **src:** Contiene todos los archivos fuente de tu aplicación *React*. Aquí es donde se desarrollará la mayor parte del código.
 - **components:** Este directorio contiene componentes reutilizables que pueden ser utilizados en diferentes partes de la aplicación.

- **screens:** Aquí se encuentran las pantallas o vistas principales de la aplicación. Cada archivo, como *Home.js*, *About.js*, y otros, representa una pantalla individual de la aplicación.
- **index.js:** Punto de entrada de la aplicación *React*. Este archivo importa *ReactDOM* y el componente principal *App* y lo renderiza en el elemento con id *root* en *index.html*.
- **App.js:** Es el componente principal de la aplicación *React*. Define la estructura inicial de tu aplicación y puede importar y renderizar otros componentes.
- **App.css:** Archivo *CSS* donde puedes definir estilos específicos para el componente *App*.
- **index.css:** Estilos globales que se aplican a toda la aplicación.
- **logo.svg:** Una imagen de ejemplo que viene con el proyecto.
- **.gitignore:** Archivo que le dice a *Git* qué archivos y directorios ignorar. Esto asegura que no se incluyan archivos innecesarios o sensibles en el repositorio *Git*.
- **package.json :** Como se ha visto antes, es el archivo de configuración del proyecto *npm*. Contiene metadatos del proyecto, listado de dependencias, *scripts* de *npm* para tareas comunes como iniciar el servidor de desarrollo, construir el proyecto para producción, ejecutar pruebas, entre otros.
- **README.md:** Archivo donde puedes proporcionar información sobre tu proyecto, cómo configurarlo, cómo ejecutarlo, y otra documentación relevante.

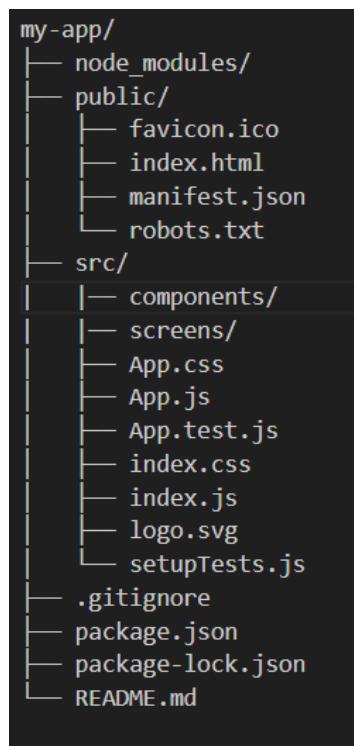


Figura 5.5: Estructura Proyecto React. Fuente: Elaboración propia

5.4.3. Componentes principal App.js

Como se ha mencionado anteriormente, el archivo *App.js* es el componente principal de una aplicación *React*. Este archivo define la estructura general de la aplicación y puede contener la lógica de enrutamiento y la gestión del estado global.

En *App.js*, se establecerá la estructura *HTML* del *e-Commerce*. En la Figura 5.6 se pueden observar las distintas partes en las que se estructurará la aplicación. A continuación, se explica cada una de ellas:

- **Header:** El encabezado se implementará utilizando el componente `Navbar` de `react-bootstrap`. Como muestra el Bloque de Código 5.6, este será el responsable de:

- **Definir el logo y el nombre de la aplicación:** El logo y el nombre del *e-Commerce*, ‘*laManchaCommerce*’, se definen utilizando el componente `<Navbar.Brand>` de `react-bootstrap`.
- **Navegación principal de la aplicación:** Incluye enlaces al carro de compras y a la opción de iniciar sesión, utilizando los componentes `NavDropdown` y `LinkContainer` de `react-bootstrap`. Si el usuario está autenticado, se muestra un menú desplegable con opciones adicionales como “*User Profile*”, “*Order History*” “*Sign Out*”.
- **Mostrar Badge de carrito:** Se utiliza el componente `<Badge>` de `react-bootstrap` para mostrar la cantidad de elementos en el carro de compras.

Bloque de código 5.6: Header de App.js

```

1  <header>
2      <Navbar variant="dark" expand="lg">
3          <Container>
4              <LinkContainer to="/" >
5                  <Navbar.Brand>
6                      {' '}
13                     laManchaCommerce
14                 </Navbar.Brand>
15             </LinkContainer>
16
17             <Navbar.Toggle <-->
18                 <-- aria-controls="basic-navbar-nav" />
19             <Navbar.Collapse id="basic-navbar-nav">
20                 <Nav className="me-auto w-100 <-->
21                     <-- justify-content-end">
22                     <Link to="/cart" className="nav-link" >
23                         Cart
24                         {cart.cartItems.length > 0 && (
25                             <Badge pill bg="danger">
26                             {cart.cartItems.reduce((a, c) => a <-->
27                                 + c.quantity, 0)}
28                         </Badge>
29                     )}
30                 </Link>
31                 {userInfo ? (
32                     <NavDropdown title={userInfo.name} <-->
33                         <-- id="basic-nav-dropdown">
34                         <LinkContainer to="/profile" >
35                             <NavDropdown.Item>User <-->
36                             <-- Profile</NavDropdown.Item>
37                         </LinkContainer>
38                         <LinkContainer to="/orderhistory" >
39                             <NavDropdown.Item>Order <-->
40                             <-- History</NavDropdown.Item>
41                         </LinkContainer>
42                         <NavDropdown.Divider />
43                         <Link
44                             className="dropdown-item"
45                             to="#signout"
46                             onClick={signoutHandler}
47                         >

```

```

42           Sign Out
43             </Link>
44           </NavDropdown>
45         ) : (
46           <Link className="nav-link" to="/signin">
47             Sign In
48           </Link>
49         )
50       </Nav>
51     </Navbar.Collapse>
52   </Container>
53 </Navbar>
54 </header>

```

- **Main:** La sección principal se implementará con la etiqueta `<main>`. Esta sección se encargará de gestionar el enrutamiento de la aplicación, determinando qué componente debe renderizarse según la *URL* actual gracias a la dependencia *react-router-dom*. Con esto se conseguirá una navegación fluida entre diferentes pantallas, como la pantalla de inicio, el producto, el carrito, etc. Como se puede observar en el Bloque de Código 5.7, este funcionamiento se conseguirá gracias a:

- **Routes y Route:** Utiliza los componentes *Routes* y *Route* de *react-router-dom* para definir las rutas de la aplicación y sus correspondientes componentes.
- **Definición de Rutas:** Cada *Route* mapea una ruta *URL* a un componente específico, como *HomeScreen*, *ProductScreen*, *CartScreen*, etc.
- **Container:** Usa el componente *Container* de *react-bootstrap* para centrar y dar formato al contenido principal.

Bloque de código 5.7: Main de App.js

```

1   <main className="mt-3 mb-3">
2     <Container>
3       <Routes>
4         <Route path="/product/:slug" ↵
5           ↵ element={<ProductScreen />} />
6         <Route path="/" element={<HomeScreen />} />
7         <Route path="/cart" element={<CartScreen />} />
8         <Route path="/signin" element={<SigninScreen />} />
9         <Route path="/signup" element={<SignupScreen />} />
10        <Route path="/payment" ↵
11          ↵ element={<PaymentMethodScreen />} /></Route>
12        <Route
13          path="/shipping"
14          element={<ShippingAddressScreen />}
15        ></Route>
16        <Route path="/placeorder" ↵
17          ↵ element={<PlaceOrderScreen />} />
18        <Route path="/order/:id" element={<OrderScreen ↵
19          ↵ />} /></Route>
20        <Route
21          path="/orderhistory"
22          element={<OrderHistoryScreen />}
23        ></Route>
24        <Route path="/profile" element={<ProfileScreen ↵
25          ↵ />} />
26      </Routes>
27    </Container>
28  </main>

```

- **Footer:** El pie de página se implementará con la etiqueta *footer*. Esta sección proporcionará la información de contacto y enlaces a redes sociales del *e-Commerce*. Como se observa en

el Bloque de Código 5.8, dentro de esta información se incluyen detalles importantes como dirección, correo electrónico y teléfono, que serán implementados con la etiqueta *HTML* `<p>` para crear párrafos, y enlaces a plataformas de redes sociales gracias a la etiqueta *HTML* `<a>`. Se utilizan íconos de *FontAwesome* para dar una apariencia más moderna y visual.

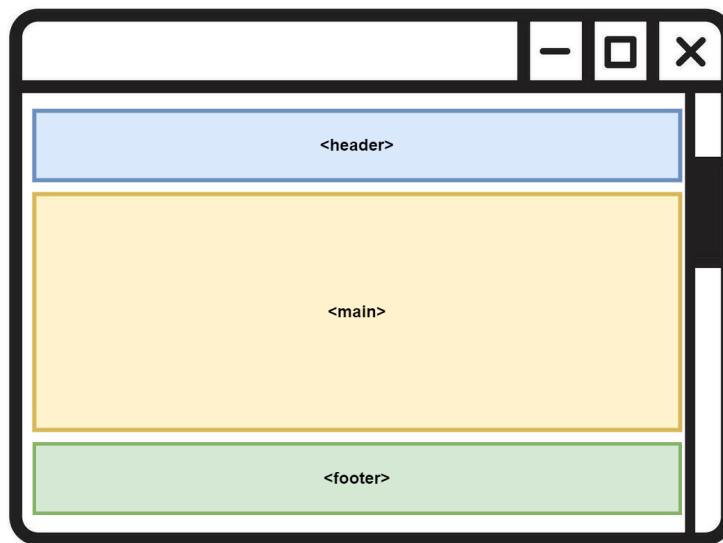


Figura 5.6: Diseño estructura de *e-Commerce*. Fuente: Elaboración propia

Bloque de código 5.8: Main de App.js

```

1 <footer className="footer mt-auto py-3">
2   <div className="container">
3     <div className="row">
4       <div className="col-md-6">
5         <div className="footer-info">
6           <p className="mb-0">
7             <FontAwesomeIcon
8               icon={faHome}
9               style={{ marginRight: '5px' }}>
10            />
11            laManchaCommerce
12          </p>
13          <p className="mb-0">
14            <FontAwesomeIcon
15              icon={faMapMarker}
16              style={{ marginRight: '5px' }}>
17            />
18            Dirección: Sierra del Gollino, Toledo, ↪
19              ↪ España, 45000
20          </p>
21          <p className="mb-0">
22            <FontAwesomeIcon
23              icon={faEnvelope}
24              style={{ marginRight: '5px' }}>
25            />
26            Correo electrónico: laManchaCommerce@xxxxx.com
27          </p>
28          <p className="mb-0">
29            <FontAwesomeIcon
30              icon={faPhone}>
31            />
32            Teléfono: +34 925 123 456
33          </p>
34        </div>
35      </div>
36    </div>
37  </div>
38</footer>

```

```

29           icon={faPhone}
30             style={{ marginRight: '5px' }}
31         />
32         Teléfono: +123456789
33       </p>
34     </div>
35   </div>
36   <div className="social-icons col-md-6 d-flex" style="justify-content-align: center; justify-content-end; justify-content-center">
37     <a href="https://www.facebook.com/" target="_blank" style={{ marginRight: '10px' }}>
38       <FontAwesomeIcon icon={faFacebookF} size="2x" />
39     </a>
40     <a href="https://twitter.com/" target="_blank" style={{ marginRight: '10px' }}>
41       <FontAwesomeIcon icon={faTwitter} size="2x" />
42     </a>
43     <a href="https://www.instagram.com/" target="_blank" style={{ marginRight: '10px' }}>
44       <FontAwesomeIcon icon={faInstagram} size="2x" />
45     </a>
46   </div>
47   <div className="row mt-3">
48     <div className="col text-center">
49       <p className="mb-0">
50         © 2024 Todos los derechos reservados
51       </p>
52     </div>
53   </div>
54 </div>
55 </footer>
56
57
58
59
60
61
62
63
64
65
66
67
68

```

Además de todo lo expuesto anteriormente sobre *App.js*, también es importante destacar los siguientes aspectos adicionales:

- **Uso del Contexto para el Estado Global:** Aquí se utiliza el contexto *Store* para manejar el estado global de la aplicación, incluyendo el carrito de compras y la información del usuario.
- **Función para Manejar la Desconexión del Usuario:** Esta función gestiona la desconexión del usuario, limpiando el estado y redirigiendo a la pantalla de inicio de sesión.

5.4.4. Pantallas (*Screens*)

React ofrece numerosas ventajas para la programación del *frontend* gracias a su arquitectura basada en componentes. En este proyecto, se ha decidido estructurar la aplicación utilizando *screens* para mantener un código organizado, modular y fácilmente mantenible. Cada *screen* representará una vista específica de la aplicación de *e-Commerce*, permitiendo una mejor gestión del flujo de trabajo y una experiencia de usuario más coherente. Las *screens* que compondrán la aplicación de *e-Commerce* son las siguientes:

- **HomeScreen.js:** Es la pantalla principal del *e-Commerce* donde se muestran los productos más destacados. Sirve como punto de entrada para los usuarios.

- ***SigninScreen.js***: Es la pantalla que utilizan los usuarios para iniciar sesión en el e-Commerce.
- ***SignupScreen.js***: La pantalla *SignupScreen* permite a los usuarios registrarse en el e-Commerce "laManchaCommerce".
- ***ProductScreen.js***: La pantalla *ProductScreen* representa la página detallada de un producto en el e-Commerce. Es fundamental para mostrar información específica del producto y permitir a los usuarios interactuar con él.
- ***CartScreen.js***: La pantalla del carro de compras (*CartScreen*) se encarga de gestionar y mostrar los productos que el usuario ha añadido a su carrito.
- ***OrderScreen.js***: La pantalla *OrderScreen* está diseñada para proporcionar a los usuarios una visión clara y detallada de sus pedidos, incluyendo información de envío, detalles de pago, y una lista de los artículos en el pedido.
- ***PaymentMethodScreen.js***: La pantalla *PaymentMethodScreen* es responsable de permitir al usuario seleccionar el método de pago durante el proceso de *checkout*.
- ***ShippingAddressScreen.js***: La pantalla *ShippingAddressScreen* es responsable de permitir a los usuarios ingresar y guardar su dirección de envío.
- ***ProfileScreen.js***: La pantalla *ProfileScreen* está diseñada para permitir a los usuarios actualizar su perfil, incluyendo nombre, correo electrónico y contraseña.
- ***PlaceOrderScreen.js***: La pantalla *PlaceOrderScreen* es crucial en el flujo de compra de un usuario, permitiendo confirmar y realizar un pedido con los productos seleccionados y la información de envío y pago.

5.4.5. Desarrollo de Componentes

Los componentes en *React* son fundamentales para construir interfaces de usuario flexibles, mantenibles y escalables.

Un componente es una parte autónoma y reutilizable de la interfaz de usuario que encapsula una parte específica de la funcionalidad y el diseño. Los componentes pueden ser simples (como un botón) o complejos (como una barra lateral completa de navegación), y se componen entre sí para construir la estructura completa de la aplicación.

Algunas bibliotecas como *react-bootstrap* proporcionan una serie de componentes predefinidos y estilizados que puedes usar directamente en tus aplicaciones *React*. Estos componentes están diseñados para facilitar la creación de interfaces de usuario atractivas y responsivas, siguiendo los principios de diseño y la estética de *Bootstrap*.

Algunos de los componentes que se han desarrollado son:

- ***Product***: El componente *Product* está estructurado para mostrar los detalles del producto y manejar la adición de productos al carrito.
- ***LoadingBox***: El componente *LoadingBox* en *React* utiliza el componente *<Spinner>* de *react-bootstrap* para mostrar un indicador de carga. Como se observa en el Bloque de Código 5.9.

Bloque de código 5.9: Spinner del componente LoadingBox

```
1  <Spinner animation="border" role="status">
2    <span className="visually-hidden">Loading...</span>
3  </Spinner>
```

- ***MessageBox***: El componente *MessageBox* utiliza *<Alert>* de *react-bootstrap* para mostrar mensajes con diferentes variantes. Como se observa en el Bloque de Código 5.10.

Bloque de código 5.10: Alert del componente MessageBox

```
1  <Alert variant={props.variant || ←
  ↩ 'info'}>{props.children}</Alert>
```

- **Rating:** El componente *Rating* está diseñado para mostrar una calificación de estrellas basada en el *rating* pasado como *prop* y el número de valoraciones (*numReviews*). Como se observa en el Bloque de Código 5.11

Bloque de código 5.11: Ejemplo de Rating

```

1  function Rating(props) {
2    const { rating, numReviews } = props;
3    return (
4      <div className="rating">
5        <span>
6          <i
7            className={
8              rating >= 1
9                ? 'fas fa-star'
10               : rating >= 0.5
11                 ? 'fas fa-star-half-alt'
12                   : 'far fa-star'
13             }
14           />
15         </span>
16         <span>
17           <i
18             className={
19               rating >= 2
20                 ? 'fas fa-star'
21                   : rating >= 1.5
22                     ? 'fas fa-star-half-alt'
23                       : 'far fa-star'
24             }
25           />
26         </span>

```

5.4.6. Diseño de la Interfaz de Usuario (UI)

El diseño de la Interfaz de Usuario (*UI*) desempeña un papel fundamental en la creación de experiencias digitales que no solo sean funcionales, sino también intuitivas y atractivas para los usuarios. En el contexto de este TFG, el diseño *UI* se centra en la aplicación web de *e-Commerce* desarrollada con *React*, donde cada pantalla y componente están diseñados con el objetivo de facilitar la navegación, mejorar la usabilidad y promover la interacción efectiva con los usuarios.

En esta sección, se explora el diseño de cada pantalla, que asegura una experiencia coherente y satisfactoria a lo largo de toda la aplicación *e-Commerce*.

5.4.6.1. HomeScreen

En la Figura 7.1 se presenta el diseño y la estructura de la pantalla de inicio. Para implementar este diseño y lograr una organización responsiva de los productos, se utilizan los elementos *<Row>* y *<Col>* de *react-bootstrap*, como se puede apreciar en el Bloque de Código 5.12. Esto implica la disposición de los productos en filas horizontales (*<Row>*) y columnas (*<Col>*) dentro de una cuadrícula adaptable, diseñada para diferentes tamaños de pantalla. Cada producto se presenta mediante el componente *<Product>*.

Bloque de código 5.12: Parte de código de HomeScreen

```

1   <Row>
2     {products.map((product) => (
3       <Col key={product.slug} sm={6} md={4} lg={3} ←
4         className="mb-3"
5           <Product product={product}></Product>
6         </Col>
7     )
8   )
9 
```

```

6           })}]
7       </Row>

```

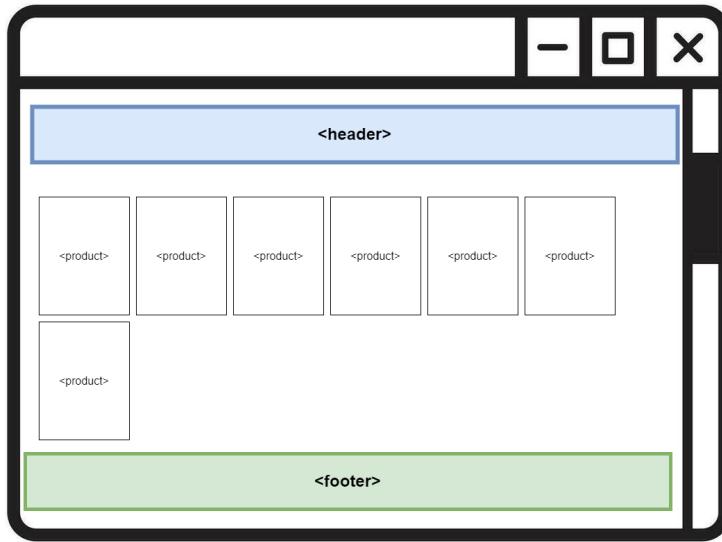


Figura 5.7: Diseño UI de *HomeScreen*. Fuente: Elaboración propia

5.4.6.2. *SigninScreen*

La Figura 5.8 ilustra el diseño y la estructura de la pantalla de inicio de sesión de usuario. Para implementar este diseño, como se muestra en el Bloque de Código 5.13 se emplea el componente *<Form>* de *react-bootstrap* para formularios. Este componente permite agrupar y organizar elementos relacionados dentro del formulario. Dentro de *<Form>*, se utiliza:

- **<Form.Control>**: Este componente se utiliza para crear varios tipos de campos de entrada, como los campos de texto para las credenciales de correo electrónico y contraseña.
- **<Form.Group>**: Este componente se utiliza para agrupar campos relacionados dentro de formularios, mejorando la organización y accesibilidad al proporcionar un contenedor semántico para etiquetas y controles.
- **<Form.Label>**: Este componente se emplea para etiquetar campos específicos dentro de formularios, asegurando una asociación visual clara entre la etiqueta y el campo correspondiente, lo que mejora la usabilidad y la accesibilidad de la interfaz de usuario.
- **<Button>**: Este componente se utiliza para generar botones con estilos predefinidos que facilitan el envío del formulario.

Dentro del *<Form>* se añade el atributo *onSubmit=submitHandler* que se utiliza para especificar que la función *submitHandler* se ejecutará al enviar el formulario. Esto se produce al hacer clic en el componente *<Button>*, se activa *submitHandler*, el cual envía la solicitud de inicio de sesión al servidor.

También en la parte inferior del formulario, se utiliza el componente *<Link>* de *react-router* para redirigir a los nuevos usuarios a la página de registro en caso de que aún no tengan una cuenta.

Bloque de código 5.13: Form de SigninScreen

```

1           <Form className="login-form" onSubmit={submitHandler}>
2

```

```
3      <h1 className="my-3">Iniciar Sesión</h1>
4      <div className="input-group">
5          <Form.Group controlId="email">
6              <Form.Label>Correo electrónico</Form.Label>
7              <Form.Control
8                  type="email"
9                  required
10                 onChange={(e) => setEmail(e.target.value)}
11             />
12         </Form.Group>
13     </div>
14     <div className="input-group">
15         <Form.Group controlId="password">
16             <Form.Label>Contraseña</Form.Label>
17             <Form.Control
18                 type="password"
19                 required
20                 onChange={(e) => setPassword(e.target.value)}
21             />
22         </Form.Group>
23     </div>
24     <div className="mb-3">
25         <Button className="button" type="submit">
26             Iniciar Sesión
27         </Button>
28     </div>
29     <div className="mb-3 bottom-text">
30         ¿Eres nuevo aquí?{' '}
31         <Link to={`/signup?redirect=${redirect}`}>Crea tu ←
32             cuenta</Link>
33     </div>
34 </Form>
```

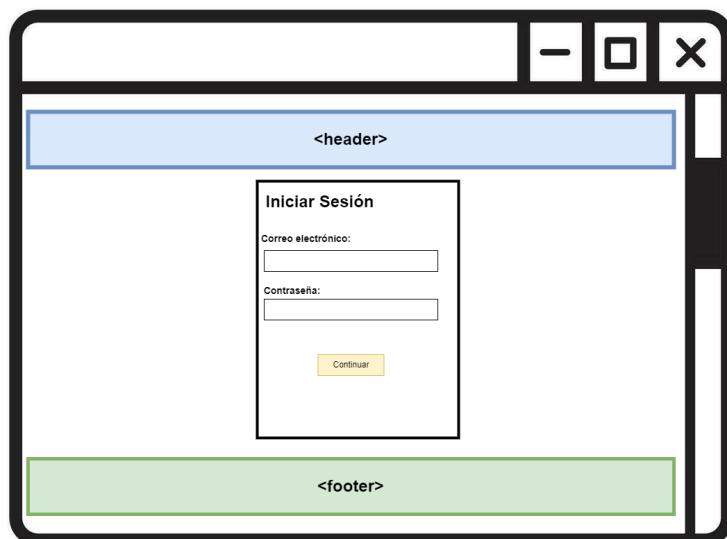


Figura 5.8: Diseño UI de *SigninScreen*. Fuente: Elaboración propia

5.4.6.3. SignupScreen

La Figura 5.9 ilustra el diseño y la estructura de la pantalla crear cuenta de usuario. Para implementar este diseño, tambien se utiliza el componente de formulario `<Form>`. Este formulario incluye campos para nombre, correo electrónico, contraseña y confirmación de contraseña. Para el diseño tambien se utilizas los componentes `<Form.Group>`, `<Form.Label>`, `<Form.Control>`, y `<Button>` para estructurar y estilizar el formulario de manera ordenada y funcional.

En este pantalla cuando se hace clic en el componente `<Button>` tambien se activa la función `submitHandler` que se encarga de gestionar el formulario de registro.

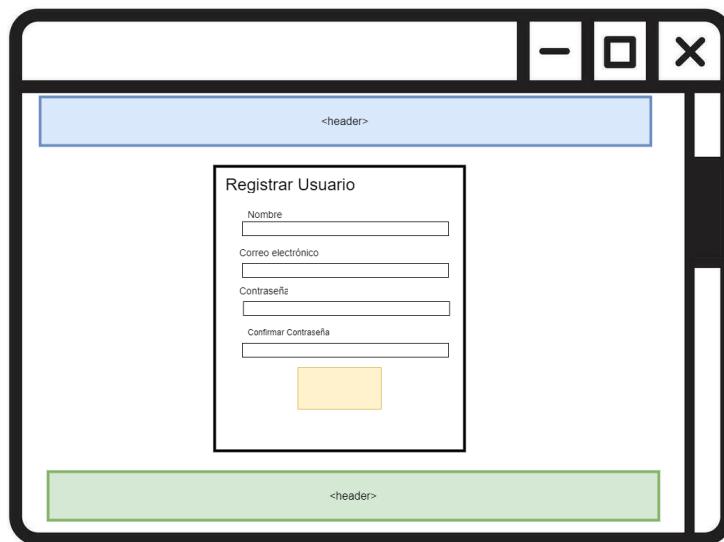


Figura 5.9: Diseño UI de *SignupScreen*. Fuente: Elaboración propia

5.4.6.4. ProductScreen

La Figura 5.10 ilustra el diseño y la estructura de la pantalla de detalle de un producto. Para implementar este diseño, se emplea los siguientes componentes:

- `<Row>` y `<Col>` : Estos componentes se utilizan para darle a la pagina un diseño de filas y columnas. Como se muestra en el Bloque de Código 5.14.

Bloque de código 5.14: Ejemplo de COL

```

1  <Col md={6}>
2    /* Contenido de la primera columna */
3  </Col>
4  <Col md={3}>
5    /* Contenido de la segunda columna */
6  </Col>
7  <Col md={3}>
8    /* Contenido de la tercera columna */
9  </Col>

```

- `` : Se utiliza la etiqueta `` de *HTML* para representar la imgagen del producto. Como se muestra en el Bloque de Código 5.15.

Bloque de código 5.15: Ejemplo de img

```
1 | <img
```

```

2     className="img-large"
3     src={product.image}
4     alt={product.name}
5 ></img>

```

- **<ListGroup>**: Componente de *react-bootstrap* que ayuda a organizar y mostrar información en forma de lista. Dentro de este componente se encuentra **<ListGroup.Item>** que representa cada elemento de la lista. Como se muestra en el Bloque de Código 5.16.

Bloque de código 5.16: Ejemplo de ListGroup

```

1 <ListGroup variant="flush">
2   {/* ListGroup.Items dentro de este ListGroup */}
3 </ListGroup>

```

- **<ListGroup.Item>**: Como se observa en el Bloque de Código 5.17 este componente muestra diferentes detalles del producto como el precio, origen, certificaciones y descripción. Usar las etiquetas *HTML* **** y **<p>** hacen que los detalles sean claros y bien organizados. Ya **<p>** sirve para definir un párrafo de texto en *HTML* y **** se utiliza para enfatizar un texto y hacerlo visualmente más destacado.

Bloque de código 5.17: Ejemplo de ListGroup.Item

```

1 <ListGroup.Item>
2   <strong>Precio: </strong> {product.price}
3 </ListGroup.Item>
4 <ListGroup.Item>
5   <strong>Origen: </strong> {product.origen}
6 </ListGroup.Item>
7 <ListGroup.Item>
8   <strong>Certificaciones: </strong> ↵
9     ↵ {product.certificaciones}
10 </ListGroup.Item>
11 <ListGroup.Item>
12   <strong>Descripción:</strong>
13   <p>{product.descripcion}</p>
14 </ListGroup.Item>

```

- **<Card>**: Es un componente de *react-bootstrap* que muestra contenido en una tarjeta con borde y sombra. Como se muestra en el Bloque de Código 5.23.

```

1
2   \begin{lstlisting}[style=ruled,language=html,caption={Ejemplo ↵
3     ↵ de Card},label=lst:card]
4 <Col md={3}>
5   <Card>
6     /* Contenido del Card */
7   </Card>
8 </Col>

```

- **<Rating>**: Es un componente personalizado que muestra la calificación del producto (*product.rating*) y el número de revisiones (es un componente personalizado que muestra la calificación del producto (*product.rating*) y el número de revisiones (*product.numReviews*)). Como se muestra en el Bloque de Código 5.18.

Bloque de código 5.18: Ejemplo de sRating

```

1 <ListGroup.Item>
2   <Rating
3     rating={product.rating}
4     numReviews={product.numReviews}
5   ></Rating>
6 </ListGroup.Item>

```

- **<Badge>**: Es un componente de *react-bootstrap* que muestra un distintivo visual (*success* o *danger*) dependiendo del estado de *product.countInStock*. Como se muestra en el Bloque de Código 5.19.

Bloque de código 5.19: Ejemplo de Badge

```

1 <Col>
2   {product.countInStock > 0 ? (
3     <Badge bg="success">Disponible</Badge>
4   ) : (
5     <Badge bg="danger">Sin Stock</Badge>
6   )}
7 </Col>

```

- **<Button>**: Es un componente de *react-bootstrap* que muestra un botón *Añadir al carrito*. *addCartHandler* es la función que maneja la acción de añadir al carrito cuando se hace clic en este botón. Como se muestra en el Bloque de Código 5.20.

Bloque de código 5.20: Ejemplo de Button

```

1 <Button onClick={addCartHandler} variant="primary">
2   {' '}
3   Añadir al carrito
4 </Button>

```

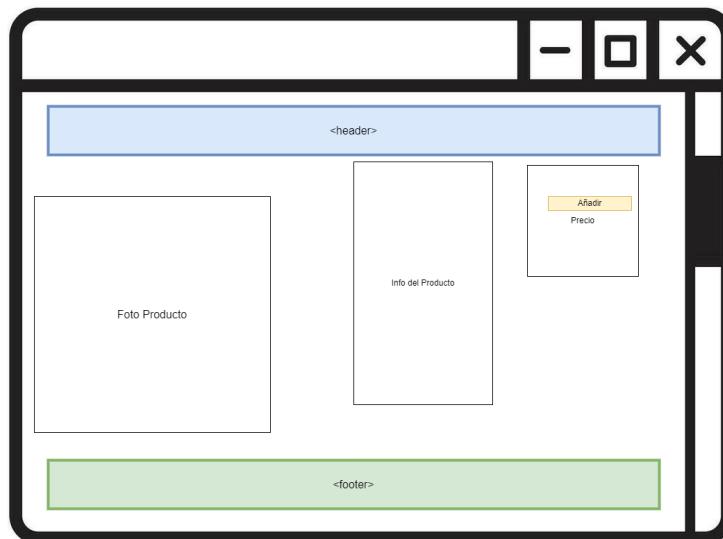


Figura 5.10: Diseño UI de *ProductScreen*. Fuente: Elaboración propia

5.4.6.5. CartScreen

La Figura 5.12 ilustra el diseño y la estructura de la pantalla de carro de compras del *e-Commerce*. Para implementar este diseño, se ha utilizado un diseño de página basado en filas y columnas (*<Row>* y *<Col>*) y también se han utilizado los componentes explicados anteriormente *<Card>*, *<ListGroup>* y *<Button>*.



Figura 5.11: Diseño UI de *CartScreen*. Fuente: Elaboración propia

5.4.6.6. ShippingAddressScreen

La Figura 5.8 ilustra el diseño y la estructura de la pantalla de la dirección de envío de la compra del *e-Commerce*. El diseño de esta pantalla es de tipo formulario, por lo que se utiliza el componente `<Form>` y sus elementos relacionados como `<Form.Group>`, `<Form.Label>` y `<Form.Control>`. En este formulario también se utiliza otro componente de *react-bootstrap* que es `<Form.Select>`, este se utiliza para crear y manejar selectores (elementos `<select>` en *HTML*) de una manera más integrada y estilizada dentro de aplicaciones React. Como se muestra en el Bloque de Código 5.21 se utilizará para crear un selector de países dentro de un formulario.

Bloque de código 5.21: Ejemplo de Form.Select

```

1 <Form.Select
2   value={country}
3   onChange={(e) => setCountry(e.target.value)}
4   required
5 >
6   <option value="">-- Selecciona una opción--</option>
7   <option value="espana">España</option>
8   <option value="portugal">Portugal</option>
9   <option value="chile">Chile</option>
10  <option value="argentina">Argentina</option>
11  <option value="bolivia">Bolivia</option>
12  <option value="colombia">Colombia</option>
13 </Form.Select>

```



Figura 5.12: Diseño UI de *ShippingAddressScreen*. Fuente: Elaboración propia

5.4.6.7. PaymentMethodScreen

La Figura 5.13 ilustra el diseño y la estructura de la pantalla de la selección del método de pago del *e-Commerce*. El diseño de esta pantalla es de tipo formulario, por lo que se utiliza el componente *<Form>* y dentro de este se utilizará el componente de *react-bootstrap* *<Form.Check>*. Este se utiliza para crear *checkboxes* (casillas de verificación) dentro de formularios en aplicaciones React. Como se muestra en el Bloque de Código 5.22 tendrá una única forma de pago que será *PayPal*.

Bloque de código 5.22: Ejemplo de Form.Check

```

1 <Form.Check
2   type="radio"
3   id="PayPal"
4   label={
5     <>
6       {' '}
11      </>
12    }
13   value="PayPal"
14   checked={paymentMethodName === 'PayPal'}
15   onChange={(e) => setPaymentMethod(e.target.value)}
16 />

```

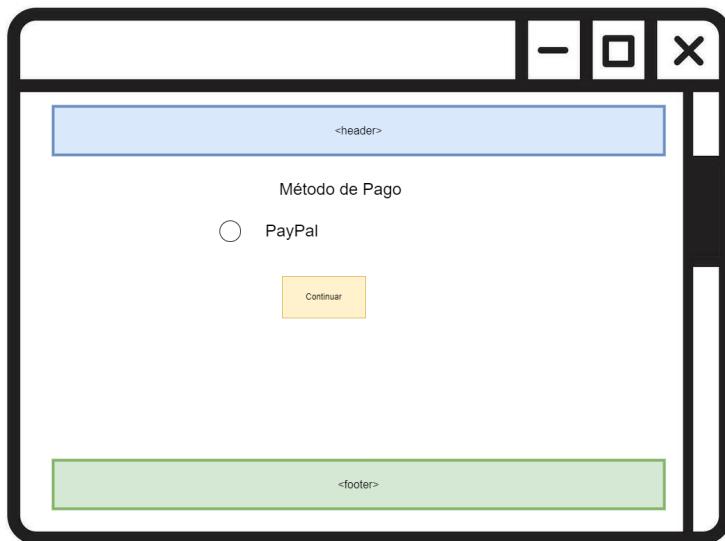


Figura 5.13: Diseño UI de *PaymentMethodScreen*. Fuente: Elaboración propia

5.4.6.8. PlaceOrderScreen

La Figura 5.14 ilustra el diseño y la estructura de la pantalla de realizar pedido del *e-Commerce*. El diseño de esta pantalla se basa en filas y columnas mostrando la información en tarjetas (*<Card>*), dicha información es datos del envío, datos del pago, productos en el carro y resumen del pedido. Para estructurar y mostrar contenido dentro de una tarjeta de forma organizada y estilizada se utilizan los componentes *<Card.Body>*, *<Card.Title>* y *<Card.Text>*. *<Card.Body>* es un componente que se utiliza para agrupar y organizar el contenido principal dentro de una tarjeta (*<Card>*). *<Card.Title>* es un componente que se utiliza dentro de *<Card.Body>*. *<Card.Text>* es un componente que se utiliza dentro de *<Card.Body>* para colocar texto descriptivo o informativo debajo del título de la tarjeta.

El Bloque de Código 5.23 es un ejemplo de cómo sería la tarjeta de datos del envío

Bloque de código 5.23: Ejemplo de Card

```

1 <Card className="mb-3">
2   <Card.Body>
3     <Card.Title>Envío:</Card.Title>
4     <Card.Text>
5       <strong>Nombre:</strong> ↵
6         ↵ {cart.shippingAddress.fullName} <br />
7       <strong>Dirección:</strong> ↵
8         ↵ {cart.shippingAddress.address},
9         {cart.shippingAddress.city}, ↵
10        ↵ {cart.shippingAddress.postalCode},
11        {cart.shippingAddress.country}
12      </Card.Text>
13      <Link to="/shipping">Editar</Link>
14    </Card.Body>
15  </Card>

```

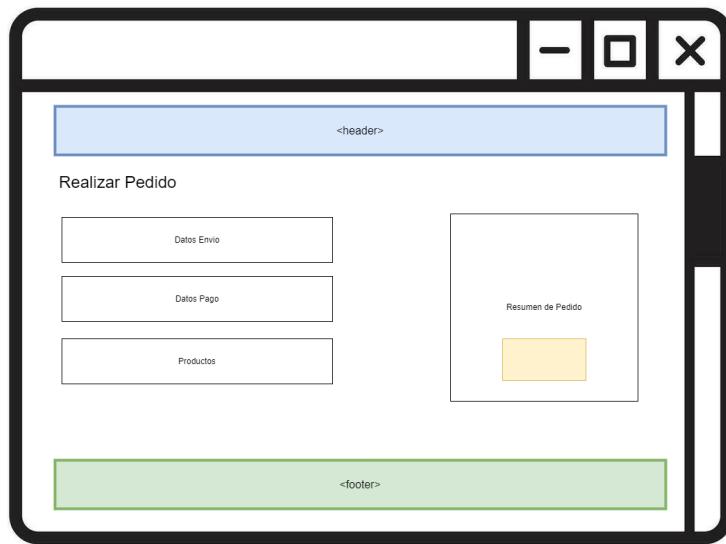


Figura 5.14: Diseño UI de *PlaceOrderScreen*. Fuente: Elaboración propia

5.4.6.9. OrderScreen

La Figura 5.15 ilustra el diseño y la estructura de la pantalla del pedido del *e-Commerce*. El diseño de esta pantalla se basa en filas y columnas mostrando la información en tarjetas (*<Card>*). Dicha información es datos del envío, datos del pago, productos y resumen del pedido. Dentro de la tarjeta de resumen del pedido destacar el componente *<PayPalButtons>*. Este es un componente proporcionado por la biblioteca *@paypal/react-paypal-js* que facilita la integración de botones de *PayPal* en aplicaciones *React*. Este componente está diseñado para simplificar el proceso de aceptación de pagos utilizando *PayPal*.

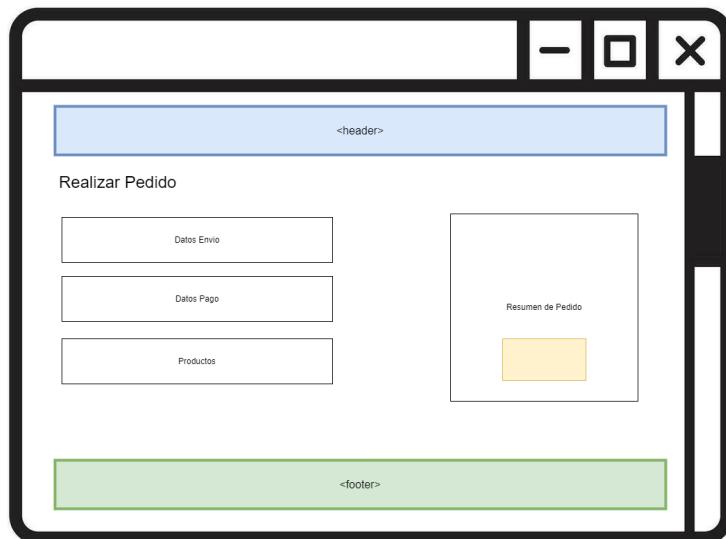


Figura 5.15: Diseño UI de *OrderScreen*. Fuente: Elaboración propia

5.4.7. Estado y *Hooks* en las Pantallas

Cuando se habla de estado o gestión de estado en una aplicación, se refiere a cómo se manejan y actualizan los datos que cambian con el tiempo, como la información de usuario, artículos en un carrito de compras, etc. Esto se consigue gracias a lo que *React* se llama *Hooks*. Un Hook es una función especial que permite conectarse a características de *React*. En esta aplicación, dos de los *Hooks* de *React* clave para conseguir esta gestión de estado son:

- ***useState***: *useState* es un Hook que te permite añadir el estado de *React* a un componente de función. Un estado es básicamente un dato que pertenece solo a ese componente en particular y puede cambiar a lo largo del tiempo (por ejemplo, un valor en un formulario que el usuario puede editar). Como ejemplo se puede observar el Bloque de Código 5.24, primero se inicializa el estado con un valor vacío (*useState('')*). *setEmail* y *setPassword* son funciones que se usan para actualizar los valores, de *email* y *password* respectivamente. Por lo que cada vez que el usuario escribe en los campos del formulario, *setEmail* y *setPassword* actualizan los estados *email* y *password* con los nuevos valores como muestra el Bloque de Código 5.24.
- ***useContext***: *useContext* es otro *hook* de *React* que te permite acceder al estado global de la aplicación. Un *estado global* es un dato que puede ser compartido entre múltiples componentes sin tener que pasarlo manualmente como propiedades (*props*). Es útil para cosas como la autenticación de usuarios, donde varios componentes necesitan saber si el usuario ha iniciado sesión o no.

Bloque de código 5.24: Ejemplo de *useState*

```

1  const [email, setEmail] = useState(''); // Define el ←
   ↵ estado 'email' y la función 'setEmail' para ←
   ↵ actualizarlo
2  const [password, setPassword] = useState(''); // Define el ←
   ↵ estado 'password' y la función 'setPassword' para ←
   ↵ actualizarlo
3
4  return (
5    <form>
6      <input
7        type="email"
8        value={email}
9        onChange={(e) => setEmail(e.target.value)}
10     />
11     <input
12       type="password"
13       value={password}
14       onChange={(e) => setPassword(e.target.value)}
15     />
16   </form>
17 );

```

Como ejemplo, se toma la pantalla de *SigninScreen* el trozo de código que muestra el Bloque de Código 5.25. Se puede observar cómo se utiliza *useContext*. *useContext(Store)* accede al estado global definido en *Store*. *state* contiene todo el estado global de la aplicación, incluyendo *userInfo*, que es la información del usuario. *ctxDispatch* es una función que se usa para enviar acciones y actualizar el estado global.

Bloque de código 5.25: Ejemplo de *useContext*

```

1  const { state, dispatch: ctxDispatch } = useContext(Store); ←
   ↵ // Accede al estado global y al dispatcher desde el ←
   ↵ contexto
2
3  const { userInfo } = state; // Obtén la información del ←
   ↵ usuario del estado global

```

```

4
5  const handleLogin = () => {
6    ctxDispatch({ type: 'USER_LOGIN', payload: { email: ↵
7      ↵ 'usuario@example.com' } }); // Envía una acción para ↵
8      ↵ actualizar el estado global
9  };
10
11 return (
12   <div>
13     {userInfo ? (
14       <p>Bienvenido, {userInfo.email}</p>
15     ) : (
16       <button onClick={handleLogin}>Iniciar Sesión</button>
17     )}
18   </div>
19 );

```

5.4.8. Integración con Backend

Para integrar el *backend* en la aplicación *React* se utiliza la librería *Axios*. *Axios* es una librería *JavaScript* que permite hacer sencillas las operaciones como cliente *HTTP*. Un ejemplo del uso de esta librería es en la pantalla *HomeScreen* cuando hace una llamada *GET* al *backend* para traer todos los productos destacados. Como se observa en el Bloque de Código 5.26 se utiliza el *Hook useEffect* para lograr hacer la petición *HTTP* al *backend* y traer los productos. *useEffect* es un *hook* en *React* que te permite realizar efectos secundarios en tus componentes funcionales. En la aplicación también se definirá una variable de entorno, la cual será la la *URL* del *backend*(*process.env.REACT_APP_API_URL*).

Bloque de código 5.26: Ejemplo de librería Axios GET

```

1     useEffect(() => {
2   const fetchData = async () => {
3     dispatch({ type: 'FETCH_REQUEST' });
4     try {
5       const result = await axios.get(
6         `${process.env.REACT_APP_API_URL}/api/products`
7       );
8       dispatch({ type: 'FETCH_SUCCESS', payload: result.data });
9     } catch (err) {
10       dispatch({ type: 'FETCH_FAIL', payload: err.message });
11     }
12     //setProducts(result.data);
13   };
14   fetchData();
15 }, []);

```

Otro ejemplo de una llamada *POST* al *backend* es cuando se registra en el *e-Commerce*. Como se observa en el Bloque de Código 5.27 , que es un parte del código de la pantalla *SignupScreen*, la llamada *HTTP* está dentro de una función creada llamada *submitHandler*. Esta se ejecuta cuando se hace clic al *Button* del formulario, que es cuando el usuario se registra.

Bloque de código 5.27: Ejemplo de librería Axios POST

```

1 const submitHandler = async (e) => {
2   e.preventDefault();
3   console.log(password);
4   console.log(confirmPassword);
5
6   if (password !== confirmPassword) {
7     toast.error('Passwords do not match');
8     return;
9   }

```

```

10  try {
11    const { data } = await Axios.post(
12      `${process.env.REACT_APP_API_URL}/api/users/signup`,
13      {
14        name,
15        email,
16        password,
17      }
18    );
19    ctxDispatch({ type: 'USER_SIGNIN', payload: data });
20    localStorage.setItem('userInfo', JSON.stringify(data));
21    navigate(redirect || '/');
22  } catch (err) {
23    toast.error(getError(err));
24  }
25};

```

5.4.9. Estilos y Tematización

Como se ha mencionado antes, para el desarrollo de la *frontend* del *e-Commerce* se ha utilizado *react-bootstrap*. Este *framework* de *frontend* viene con una serie de componentes de interfaz de usuario predefinidos como puede ser *Button*, *Navbar*, *Form*, etc.

Aunque se han utilizado elementos predefinidos, en alguna ocasión se han tenido que modificar el estilo de estos componentes modificando el archivo de estilo *CSS* *index.css*.

Una de las cosas que se ha intentado es que el *e-Commerce* tenga un aspecto y un estilo similar a un *e-Commerce* mundialmente conocido como es *Amazon*. Como ejemplo, uno de los elementos que se ha modificado son los botones para que se parezcan a los de *Amazon*, en el Bloque de Código 5.28 se pueden observar las propiedades *CSS* que van a tener los botones para que se parezcan a los de *Amazon*. A continuación, se detallan las propiedades y sus funciones:

- ***width***: Establece el ancho del botón.
- ***padding***: Añade un relleno interno de 10 píxeles en todos los lados del botón.
- ***background-color***: Establece el color de fondo del botón en un tono de naranja.
- ***color***: Establece el color del texto del botón en blanco.
- ***border***: Elimina cualquier borde predeterminado del botón.
- ***border-radius***: Redondea las esquinas del botón con un radio de 4 píxeles.
- ***font-size***: Establece el tamaño de la fuente del texto del botón a 16 píxeles.
- ***cursor***: Cambia el cursor a una mano apuntando cuando se pasa el ratón sobre el botón.
- ***font-weight***: Hace que el texto del botón sea negrita.
- ***text-transform***: Transforma el texto del botón a mayúsculas.

Bloque de código 5.28: Ejemplo de CSS

```

1 .shipping-form .button {
2   width: 100%;
3   padding: 10px;
4   background-color: #ffa41c;
5   color: white;
6   border: none;
7   border-radius: 4px;
8   font-size: 16px;
9   cursor: pointer;
10  font-weight: bold;
11  text-transform: uppercase;
12}

```

5.5. DESARROLLO DEL BACKEND

5.5.1. Configuración Inicial

Para iniciar un proyecto en *Node.js*, se debe ejecutar el comando de *npm* del Bloque de Código 5.2. Este comando creará un proyecto de *Node.js* desde cero, generando un archivo *package.json* que contendrá la configuración del proyecto.

Las principales librerías necesarias para el desarrollo del *backend* son *Express* y *Mongoose*. Para añadirlas, se deben ejecutar los comandos del Bloque de Código 5.29. La librería *Express* se utiliza para crear una *API*, mientras que *Mongoose* se emplea para conectar y gestionar la base de datos *MongoDB*.

5.5.2. Estructura del Proyecto

Este proyecto de *backend* está desarrollado utilizando *Node.js* y *Express*, proporcionando una estructura modular y escalable para la creación de aplicaciones web. *Node.js*, basado en *JavaScript*, permite construir aplicaciones rápidas y eficientes en el lado del servidor. *Express* es un *framework* minimalista que facilita la creación de *APIs* y el manejo de rutas.

Bloque de código 5.29: Comandos para instalar Express y Mongoose

```
1 | npm install express
2 | npm install mongoose
```

En este proyecto, también se utiliza *Mongoose* para gestionar la interacción con *MongoDB*, una base de datos *NoSQL*. *Mongoose* proporciona una solución elegante y sencilla para definir esquemas de datos y realizar operaciones *CRUD* (Crear, Leer, Actualizar, Borrar).

Como se aprecia en la Figura 5.16 la estructura del proyecto está organizada de manera que cada componente, como configuraciones, controladores, modelos y rutas, tenga su propio lugar, lo que mejora la mantenibilidad y escalabilidad del código. A continuación, se detalla la configuración inicial y la estructura del proyecto para comenzar con el desarrollo:

- **node_modules/**: Carpeta donde *NPM* instala las dependencias del proyecto.
- **models/**: Modelos de datos definidos con *Mongoose*.
- **routes/**: Define las rutas de la aplicación.
- **utils.js**: Funciones auxiliares y utilidades que pueden ser reutilizadas en diferentes partes del proyecto.
- **package.json**: Información del proyecto, scripts y dependencias.
- **.env**: Archivo para definir variables de entorno, como las credenciales de la base de datos.
- **server.js**: Punto de entrada de la aplicación donde se inicializa *Express* y se configuran las rutas.

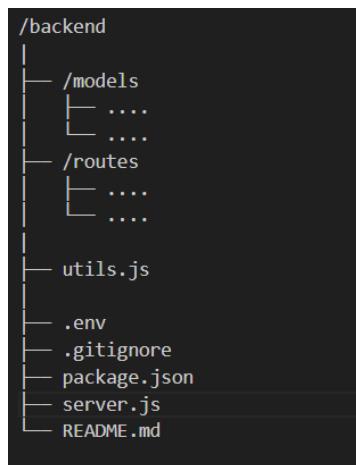


Figura 5.16: Estructura del Backend. Fuente: Elaboración propia

5.5.3. Archivos de Configuración *server.js*

Este archivo es el principal para la configuración y ejecución del *backend*. A continuación, se explican las configuraciones presentes en este archivo:

- **Carga de variables de entorno:** Como se muestra en el Bloque de Código 5.30, se cargan las variables de entorno que se encuentran en el archivo *.env* utilizando el módulo *dotenv*. Esto permite acceder a configuraciones sensibles sin tener que incluirlas directamente en el código fuente.

Bloque de código 5.30: Ejemplo de como dotenv carga variables de entorno

```
1 | dotenv.config();
```

- **Conexión con MongoDB:** La conexión con *MongoDB* se realiza mediante la librería *mongoose*. En el Bloque de Código 5.31, se observa cómo *mongoose.connect(process.env.MONGODB_URI)* conecta a la base de datos *MongoDB* usando la *URI* especificada en las variables de entorno. Además, se maneja la conexión exitosa o cualquier error que pueda ocurrir durante el proceso.

Bloque de código 5.31: Ejemplo de uso de mongoose

```
1 | mongoose
2 |   .connect(process.env.MONGODB_URI)
3 |   .then(() => {
4 |     console.log('Connected to BBDD');
5 |   })
6 |   .catch((err) => {
7 |     console.log(err.message);
8 |   });

```

- **CORS (Cross-Origin Resource Sharing):** CORS es un mecanismo que permite o restringe que recursos de una página web sean solicitados desde un dominio diferente al que originalmente los sirvió (ver Bloque de Código 5.32). Esto se configura para permitir solicitudes de diferentes orígenes, mejorando la seguridad y flexibilidad de la *API*.

Bloque de código 5.32: Ejemplo de CORS en *server.js*

```
1 | app.use(cors());
2 | app.use(express.json());
3 | app.use(express.urlencoded({ extended: true }));
```

- **Definición de rutas:** Se definen rutas para diferentes partes de la *API* (ver Bloque de Código 5.33). Esto organiza las distintas funcionalidades del *backend*, permitiendo un acceso estructurado a los recursos.

Bloque de código 5.33: Ejemplo de rutas en Serve.js

```

1 app.use('/api/seed', seedRouter);
2 app.use('/api/products', productRouter);
3 app.use('/api/users', userRouter);
4 app.use('/api/orders', orderRouter);

```

- **Inicio del servidor:** El servidor se inicia utilizando el puerto especificado en las variables de entorno o el puerto 4000 por defecto, como se muestra en el Bloque de Código 5.34. Esto pone en marcha el *backend* y lo hace accesible para recibir solicitudes.

Bloque de código 5.34: Ejemplo de como dotenv carga variables de entorno

```

1 const port = process.env.PORT || 4000;
2 app.listen(port, () => {
3   console.log('Serve at http://localhost:${port}');
4 });

```

5.5.4. Rutas y Controladores

Las rutas en *Express* permiten definir cómo responde el servidor a las solicitudes *HTTP* en diferentes puntos finales. Como ejemplo, se explicará *productRouter*. A continuación, se explicará cómo se utiliza *productRouter* (ver Bloque de Código 5.35) para manejar solicitudes relacionadas con productos:

- **Configuración del Enrutador:** Primero, se crea un enrutador de *Express*.
- **Definición de Rutas:** Luego, se definen las distintas rutas para el enrutador de productos:
 - **Obtener todos los productos :** La ruta raíz (/) maneja solicitudes *GET* para obtener todos los productos. Utiliza el modelo *Product* para buscar todos los productos en la base de datos y los envía en la respuesta.
 - **Obtener un producto por slug:** Esta ruta maneja solicitudes *GET* para obtener un producto específico basado en su *slug*. Utiliza *findOne* para buscar un producto que coincida con el *slug* proporcionado en los parámetros de la *URL*.
 - **Obtener un producto por ID:** Esta ruta maneja solicitudes *GET* para obtener un producto específico basado en su *id*. Utiliza *findById* para buscar un producto que coincida con el *id* proporcionado en los parámetros de la *URL*.
- **Manejo de Errores:** En todas las rutas, se maneja el caso en que no se encuentra un producto. Si esto ocurre, se responde con un estado 404 y un mensaje indicando que el producto no fue encontrado.
- **Exportación del Enrutador:** Finalmente, *productRouter* se exporta para que pueda ser utilizado en otras partes de la aplicación.

Bloque de código 5.35: Ejemplo de como mongoose

```

1 import express from 'express';
2 import Product from '../models/productModel.js';
3
4 const productRouter = express.Router();
5
6 productRouter.get('/', async (req, res) => {
7   const products = await Product.find();
8   res.send(products);
9 });
10
11 productRouter.get('/slug/:slug', async (req, res) => {
12   const product = await Product.findOne({ slug: req.params.slug });
13   if (product) {
14     res.send(product);
15   } else {

```

```

16     res.status(404).send({ message: 'Product Not Found' });
17   }
18 });
19
20 productRouter.get('/:id', async (req, res) => {
21   const product = await Product.findById(req.params.id);
22   if (product) {
23     res.send(product);
24   } else {
25     res.status(404).send({ message: 'Product Not Found' });
26   }
27 });
28
29 export default productRouter;

```

Para utilizar el enrutador en la aplicación *Express*, se registra en el *server.js* como se muestra en el siguiente Bloque de Código 5.33. Esto asegura que las rutas definidas en *productRouter* estén disponibles bajo un prefijo específico (en este caso, */api/products*).

5.5.5. Modelos

Los modelos son estructuras de datos que representan cómo se organizarán y se almacenarán los datos en una base de datos *MongoDB*.

En el *backend* del *e-Commerce* se tienen tres modelos:

- ***orderModel***: En este modelo se representan y organizan todos los datos relacionados con el pedido que se quiere realizar en él *e-Commerce*.
- ***productModel***: En este modelo se representan y organizan todos los datos relacionados con un producto del *e-Commerce*.
- ***userModel***: En este modelo se representan y organizan todos los datos relacionados con un usuario *e-Commerce*.

Aquí se tiene una explicación detallada de cómo es un archivo modelo:

- Primero se define el esquema. Un esquema en *Mongoose* define la estructura de los documentos dentro de una colección en *MongoDB*. En el Bloque de Código 5.36 se observa la definición del esquema del producto.
- Luego se definen los campos del esquema. En el Bloque de Código 5.36 se observan los campos del producto.
- Y por último, se crea el modelo. Un modelo en *Mongoose* es un constructor compilado a partir de un esquema. Permite crear, leer, actualizar y eliminar documentos en la colección asociada en *MongoDB*. En el Bloque de Código 5.36 se observa cómo se define el modelo *Product* utilizando el esquema *productSchema*.

Bloque de código 5.36: Código de *productModel*

```

1 import mongoose from 'mongoose';
2
3 const productSchema = new mongoose.Schema(
4   {
5     name: { type: String, required: true, unique: true },
6     slug: { type: String, required: true, unique: true },
7     image: { type: String, required: true },
8     brand: { type: String, required: true },
9     category: { type: String, required: true },
10    descripcion: { type: String, required: true },
11    price: { type: Number, required: true },
12    countInStock: { type: Number, required: true },
13    rating: { type: Number, required: true },

```

```

14     numReviews: { type: Number, required: true },
15     origen: { type: String, required: true },
16     certificaciones: { type: String, required: true },
17   },
18   {
19     timestamps: true,
20   }
21 );
22
23 const Product = mongoose.model('Product', productSchema);
24 export default Product;

```

Una vez definido el modelo, ya se puede utilizar en cualquier momento. Por ejemplo el Bloque de Código 5.37 muestra como se usa el modelo *Product* para que devuelve la colección de productos en *MongoDB*.

Bloque de código 5.37: Ejemplo de script en Matlab

```

1 productRouter.get('/', async (req, res) => {
2   const products = await Product.find();
3   res.send(products);
4 });

```

5.6. OTRAS INTEGRACIONES

5.6.1. Integración con Paypal

El método de pago seleccionado para el *e-Commerce* será a través de *PayPal*. *PayPal* es una empresa estadounidense especializada en el comercio electrónico que facilita pagos seguros en línea.

Para integrar *PayPal* en el sitio web, primero se registra en *PayPal Developer*, una plataforma que ofrece herramientas a desarrolladores y empresas para incorporar funcionalidades de pago y otras características de *PayPal* en sus aplicaciones y sitios web.

La herramienta principal utilizada es la *API* de *PayPal*, que permite a los *e-Commerce* integrar capacidades de pago de manera eficiente.

En el *backend* del sitio web, se define una ruta *GET* específica. Cuando el servidor recibe una solicitud *GET* a /api/keys/paypal, ejecuta la función correspondiente. Esta función devuelve el valor de la variable de entorno PAYOUT_CLIENT_ID, la cual es un identificador único utilizado para identificar la aplicación o sitio web durante la integración con los servicios de *PayPal* a través de su *API*. Esta variable se obtiene del panel de *PayPal Developer*, como se ilustra en la Figura 5.17.

App name	Client ID	Secret
Default Application	AX30MF8romQrMCwGtYZUNn...
lamanchacommerce	AdQiZXCLYmqzRN96LQh0Z5...

Figura 5.17: *PayPal Developer*. Fuente: Página de *PayPal Developer*

En el *frontend* de la aplicación, se emplea el componente *PayPalScriptProvider* proporcionado por la biblioteca *@paypal/react-paypal-js*. Este componente facilita la integración de *PayPal* en aplicaciones desarrolladas en *React*, permitiendo la carga eficiente y segura del *SDK* de *JavaScript* de *PayPal*.

El Bloque de Código 5.38 ilustra cómo cargar dinámicamente el *client-id* de *PayPal* desde el *backend* utilizando *Axios*, y cómo gestionar el estado de carga de *PayPal* en la aplicación *React*. Posteriormente, este *client-id* será utilizado en el *PayPalScriptProvider*.

Para llevar a cabo esta integración de manera efectiva, se utiliza el *hook* *usePayPalScriptReducer*. Este *hook* está diseñado específicamente para manejar el estado relacionado con la carga del script de *PayPal*. Como se muestra en el Bloque de Código 5.39, *paypalDispatch* se utiliza para gestionar el *client-id*, el cual será luego utilizado por *PayPalScriptProvider*.

Bloque de código 5.38: Ejemplo de Axios y Paypal

```

1 | const loadPaypalScript = async () => {
2 |   const { data: clientId } = await axios.get(
3 |     `${process.env.REACT_APP_API_URL}/api/keys/paypal`,
4 |     {
5 |       headers: { authorization: 'Bearer ↵
6 |         ↵ ${userInfo.token}' },
7 |     }
8 |   );
9 |   paypalDispatch({
10 |     type: 'resetOptions',
11 |     value: {
12 |       'client-id': clientId,
13 |       currency: 'USD',
14 |     },
15 |   });
16 |   paypalDispatch({ type: 'setLoadingStatus', value: ↵
17 |     ↵ 'pending' });
18 |
19 |   loadPaypalScript();

```

Bloque de código 5.39: Ejemplo de usePayPalScriptReducer

```

1 | const [ { isPending }, paypalDispatch ] = usePayPalScriptReducer();

```

5.6.2. Integración con JWT

JWT (JSON Web Token) es un estándar definido en el *RFC 7519* que proporciona un método seguro y eficiente para transmitir información entre dos partes. Este estándar está diseñado para propagar de manera segura la identidad de un usuario específico, junto con una serie de claims o privilegios, entre sistemas distribuidos.

En un *JWT*, los privilegios y la información del usuario se codifican en forma de objetos *JSON* que se insertan dentro del *payload* o cuerpo del *token*. Este *payload* va firmado digitalmente para garantizar su integridad y autenticidad. La firma digital se realiza utilizando una clave secreta (*JWT_SECRET*), que generalmente se almacena de forma segura en el servidor.

Un ejemplo práctico de su implementación se muestra en el Bloque de Código 5.40 del archivo *Utils.js* del *backend*, donde se definen funciones clave que utilizan este estándar.

Bloque de código 5.40: Ejemplo de uso de JWT en Utils.js

```

1 | import jwt from 'jsonwebtoken';
2 |
3 | export const generateToken = (user) => {
4 |   return jwt.sign(
5 |     {

```

```
6     _id: user._id,
7     name: user.name,
8     email: user.email,
9     isAdmin: user.isAdmin,
10    },
11    process.env.JWT_SECRET,
12    {
13      expiresIn: '30d',
14    }
15  );
16};
```

La función `generateToken` se utiliza para crear un *JWT* cuando un usuario inicia sesión correctamente. Toma como entrada un objeto *user*, que contiene información crucial como *_id*, *name*, *email*, y *isAdmin*. El *token* generado incluye estos datos como parte del *payload* y está firmado digitalmente utilizando la clave secreta especificada (*JWT_SECRET*). Además, se especifica que el *token* expirará en 30 días, lo cual es una medida de seguridad estándar para *tokens* de sesión.

Ejemplo de uso de la función anterior es el que se implementa en la ruta de inicio de sesión (`/signin`), cuando un usuario proporciona su correo electrónico y contraseña, el servidor verifica las credenciales comparando la contraseña almacenada en la base de datos con la proporcionada por el usuario. Si la autenticación es exitosa, el servidor devuelve un objeto *JSON* que incluye información básica del usuario y el *token* *JWT* generado por la función `generateToken(user)`. Este *token* puede ser utilizado por el cliente para autenticar sus solicitudes subsiguientes y acceder a recursos protegidos dentro de la aplicación.

CAPÍTULO 6

Aplicación de Procesos DevOps

6.1. GITFLOW

6.1.1. Configuración inicial

Para el desarrollo del *e-Commerce*, se ha seguido la metodología de trabajo con *Git Flow*. Primero, es necesario tener *Git* instalado en el equipo. Puedes descargar *Git* desde la página oficial: <https://git-scm.com/downloads>. Una vez instalado, se creará un repositorio en *GitHub* donde se ubicará el código del *e-Commerce*. En la Figura 6.1 se muestra la pantalla para crear un repositorio en *GitHub*.

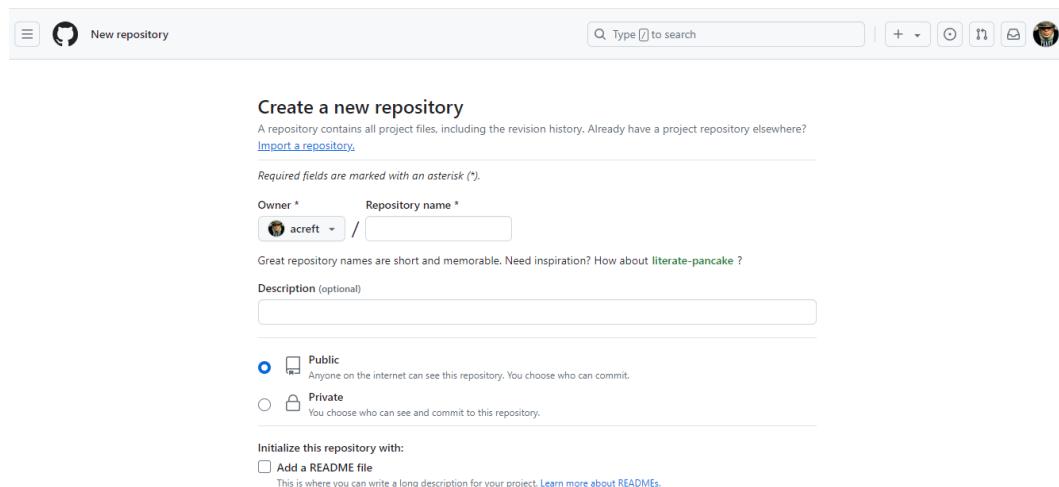


Figura 6.1: Crear repositorio en *GitHub*. Fuente: Elaboración propia

Una vez creado el repositorio, ejecutaremos el comando del Bloque de Código 6.1 en el equipo para clonarlo y poder trabajar de forma local.

Bloque de código 6.1: Comando de Git para clonar

```
1 | git clone https://github.com/acrefit/eCommerceTfg.git
```

Una vez desarrollada la estructura del proyecto, como se ha visto en el apartado 5.3, se procede a subir el código a la rama *develop* para empezar a trabajar sobre esta rama. Para hacerlo se usa el comando *push* como muestra el Bloque de Código 6.2 .

Bloque de código 6.2: Comando push de Git

```
1 | git push -u origin develop
```

6.1.2. GitFlow en el desarrollo del e-Commerce

Una vez definido el punto de partida en la rama *develop*, comenzaremos a crear las ramas de funcionalidades (*features*) para ir dando forma al *e-Commerce*. Estas ramas se crearán a partir de la rama *develop*. A continuación, se muestran todas las ramas de funcionalidades creadas para este proyecto:

- feature/config_v1_ecommerce
- feature/add_integration_paypal
- feature/add_placeorder_screen
- feature/add_payment_method_screen
- feature/add_register_screen
- feature/add_address_screen
- feature/add_mongoose_models
- feature/add_mongodb
- feature/add_login_screen
- feature/add_carreton_screen
- feature/add_react_context
- feature/add_loading_message_component
- feature/add_product_screen
- feature/create_product_component
- feature/add_bootstrap
- feature/manage_state_hook
- feature/fetch_products
- feature/create_node_server
- feature/add_routing
- feature/list_products

Para mantener una gestión organizada del desarrollo, cada una de estas ramas debe integrarse correctamente en la rama *develop*. A continuación, se detalla el proceso para finalizar una rama de funcionalidad e integrarla en *develop* utilizando *Git Flow*:

- Se ha finalizado el desarrollo en la rama *feature/add_integration_paypal*.
- Se realiza *commit* y se hace *push* de la rama, como se muestra en los comandos del Bloque de Código 6.3.

Bloque de código 6.3: Comandos de commit en Git

```

1 | git add .
2 | git commit -m "Finalizar integración de PayPal"
3 | git push origin feature/add_integration_paypal

```

- Una vez subida la rama, se crea una *pull request* en *GitHub*. En la Figura 6.2 se observa la pantalla de creación de la *pull request* en *GitHub*.

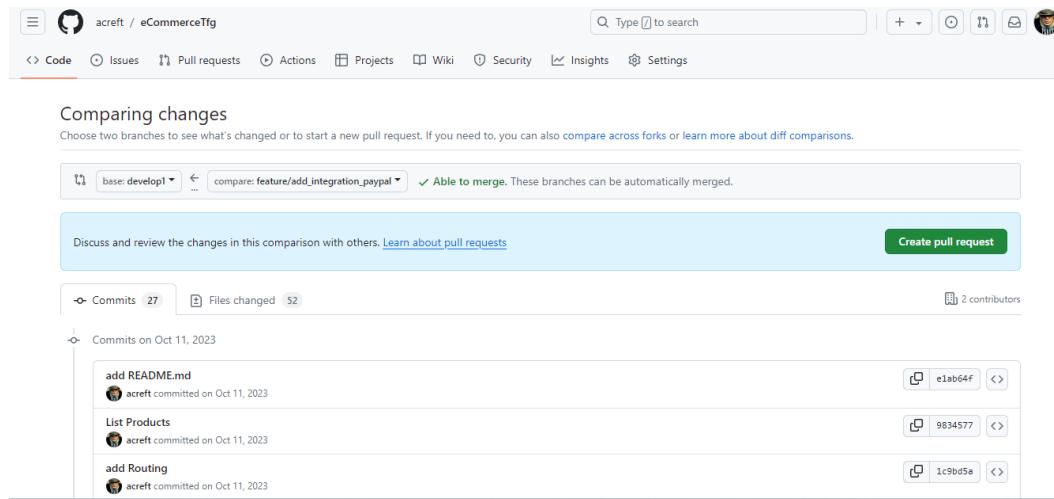


Figura 6.2: Crear *Pull Request* en *GitHub*. Fuente: Elaboración propia

- Un compañero revisa que todo esté correcto y realiza pruebas. La Figura 6.3 muestra todos los cambios de la rama *feature* a *develop*.

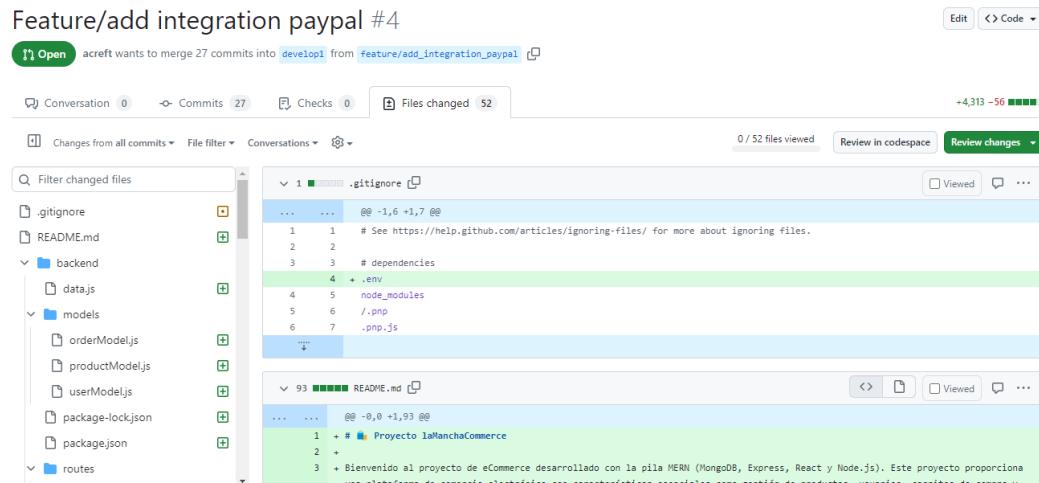


Figura 6.3: Cambios en la *Pull Request*. Fuente: Elaboración propia

- Una vez aprobada, se hace merge con *develop* y se elimina la rama.

Una vez terminadas todas las funcionalidades y teniendo una primera versión del *e-Commerce*, se crea la rama de release llamada *release/1.0.0*.

- Una vez creada la rama de *release*, se realiza una *pull request* con la rama *master*.
- Después de que la *pull request* es aceptada y revisada por los miembros del equipo, se realiza el *merge*.
- Con esto, la primera versión del *e-Commerce* está lista en la rama *master* y lista para ser desplegada en cualquier momento.
- Ahora se dispone de una primera versión funcional del *e-Commerce*.

6.2. DESPLIEGUE DEL E-COMMERCE EN RENDER

6.2.1. Introducción

En este Trabajo de Fin de Grado (TFG) para el *e-Commerce* se dispone de dos entornos: **int** y **pro**. El entorno *int* es previo a producción, donde se verifica que el *e-Commerce* funciona correctamente.

y cumple todos los requisitos antes de ser desplegado en *pro*, que es donde los usuarios finales lo utilizarán.

Las URLs de los entornos son las siguientes:

- **Producción:** <https://ecommercetfg-pro.onrender.com>
- **Integración:** <https://lamanchacommerce-int.onrender.com>

Para la gestión del ciclo de vida del software, se utiliza *Git Flow* como metodología de trabajo. Una vez finalizadas todas las funcionalidades y verificada la primera versión del e-Commerce, se crea la rama de *release*, por ejemplo, *release/1.0.0*. Esta versión se despliega en el entorno de integración donde se realizan pruebas exhaustivas para garantizar su correcto funcionamiento y cumplimiento de requisitos antes de prepararla para su despliegue en producción.

6.2.2. Despliegue en Render

Para llevar a cabo el despliegue en *Render*, es necesario seguir varios pasos esenciales que aseguran la correcta configuración y funcionamiento del e-Commerce:

Inicialmente, se crea un nuevo *Web Service* en la plataforma *Render*. Desde allí, se selecciona el repositorio en *Github* que contiene todo el código del proyecto del e-Commerce.

Durante la configuración del servicio, se definen diversos parámetros cruciales. Esto incluye asignar un nombre al servicio web y especificar que el lenguaje del proyecto es *Node.js*. Además, se elige la rama *master* del repositorio para desplegar la versión principal del e-Commerce. Se opta por el plan gratuito ofrecido por *Render* y se establece varias variables de entorno esenciales. Entre ellas se encuentran la clave *JWT* para la autenticación, la URL de conexión a *MongoDB* para la persistencia de datos, el *ClientID* de *PayPal* para los pagos y la URL del *backend* que conecta todas las partes del sistema.

A continuación en la Figura 6.4 se muestra esta pantalla de configuración.

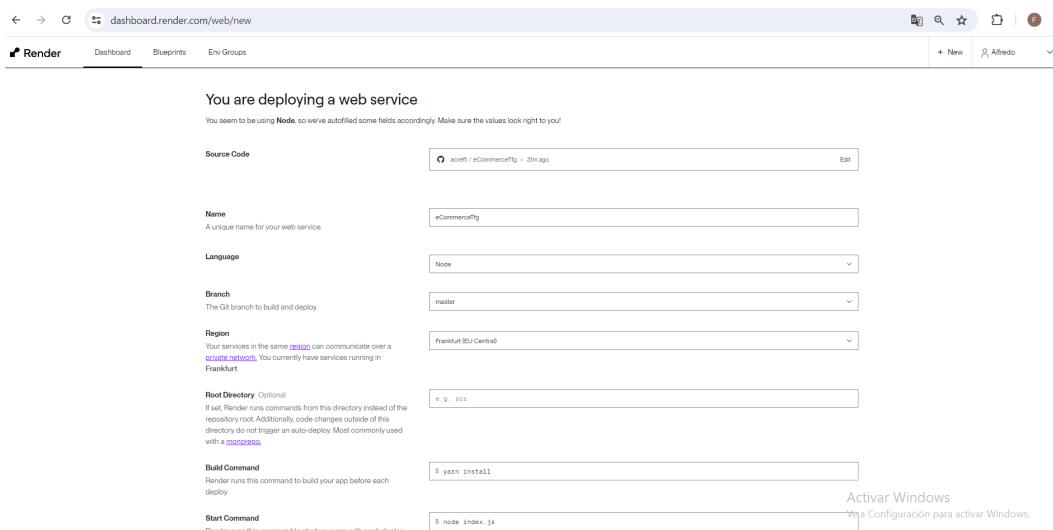


Figura 6.4: Configuración de Render Web Service. Fuente: Elaboración propia

Uno de los pasos más importantes en esta configuración es la definición de los comandos *build* y *start*, que son fundamentales para el proceso de construcción y ejecución del proyecto. Estos comandos, especificados en el archivo *package.json* del proyecto, aseguran que todas las dependencias necesarias se instalen correctamente y que el servidor *Node.js* se inicie adecuadamente para manejar las solicitudes del e-Commerce.

El comando *build* realiza la instalación de todas las dependencias tanto del *backend* como del *frontend*, seguido de la compilación de los recursos necesarios para la ejecución. Por otro lado, el

comando *start* se encarga de iniciar el servidor *Node.js* que gestiona el *backend* del *e-Commerce*, asegurando que esté listo y funcional para manejar las solicitudes de los usuarios.

Este proceso integral garantiza que el *e-Commerce* se despliegue de manera efectiva en *Render*, proporcionando una plataforma estable y eficiente para que los usuarios finales puedan acceder y utilizar la aplicación de manera segura y confiable.

6.3. PRUEBAS EN EL *E-COMMERCE*

6.3.1. Introducción

Las pruebas juegan un papel fundamental en el ciclo de vida del desarrollo de *software*, ya que aseguran que la aplicación tenga la calidad necesaria, cumpla con las necesidades del cliente y no tenga defectos.

En este TFG se implementan diversas técnicas y metodologías de prueba, para que se puedan identificar y corregir errores en etapas previas a producción, como es el entorno de integración, ya que la presencia de defectos en producción puede afectar la imagen del *e-Commerce* o causar pérdidas económicas. Además, las pruebas permiten verificar que todas las funcionalidades del *software* operen de manera adecuada bajo diferentes condiciones, contribuyendo a una experiencia de usuario satisfactoria y a la confiabilidad del producto final.

Siguiendo los principios *DevOps* en este TFG las pruebas son automatizadas. Las pruebas automatizadas son una parte esencial del desarrollo moderno de *software*, especialmente cuando se siguen los principios de *DevOps*. Estas pruebas permiten la ejecución automática de casos de prueba sin intervención humana, lo que facilita la identificación temprana de errores y la verificación continua de la calidad del *software* a lo largo de su ciclo de vida.

Existen varios tipos de pruebas automatizadas, cada una con un propósito específico. En este TFG se desarrollan pruebas de tipo funcionales y *End-to-End*, estas se enfocan en validar que el sistema cumpla con los requisitos especificados y que las funcionalidades clave operen como se espera. Las pruebas de regresión, por su parte, aseguran que las nuevas modificaciones no afecten negativamente a las funcionalidades existentes.

6.3.2. Configuración inicial

Como se ha mencionado en el apartado 4.5.2, para el tipo de pruebas *end-to-end* se opta por la herramienta *Cypress*. *Cypress* es popularmente conocido por automatizar pruebas funcionales en el navegador web de aplicaciones web.

Cypress destaca por su capacidad de ejecutar pruebas de extremo a extremo, lo que implica probar todo el flujo de la aplicación desde el inicio hasta el final, como lo haría un usuario real. Esta herramienta permite simular interacciones complejas con la aplicación, como clics, desplazamientos y entrada de datos, verificando que todas las funcionalidades operen correctamente en conjunto.

Una de las principales ventajas de *Cypress* es su facilidad de configuración y uso.

Primero, se debe instalar *Cypress*. Con el comando que aparece en el Bloque de Código 6.4, se instalará *Cypress* localmente.

Bloque de código 6.4: Comando para instalar Cypress

```
1 | npm install cypress --save-dev
```

Luego, se debe abrir *Cypress* con el comando que aparece en el Bloque de Código 6.5.

Bloque de código 6.5: Comando para abrir Cypress

```
1 | npx cypress open
```

Al abrir *Cypress*, aparece la siguiente pantalla que se muestra en la Figura 6.5. Su función es guiar al usuario a través de las decisiones y las tareas de configuración que debe completar antes de comenzar a escribir su primera prueba.

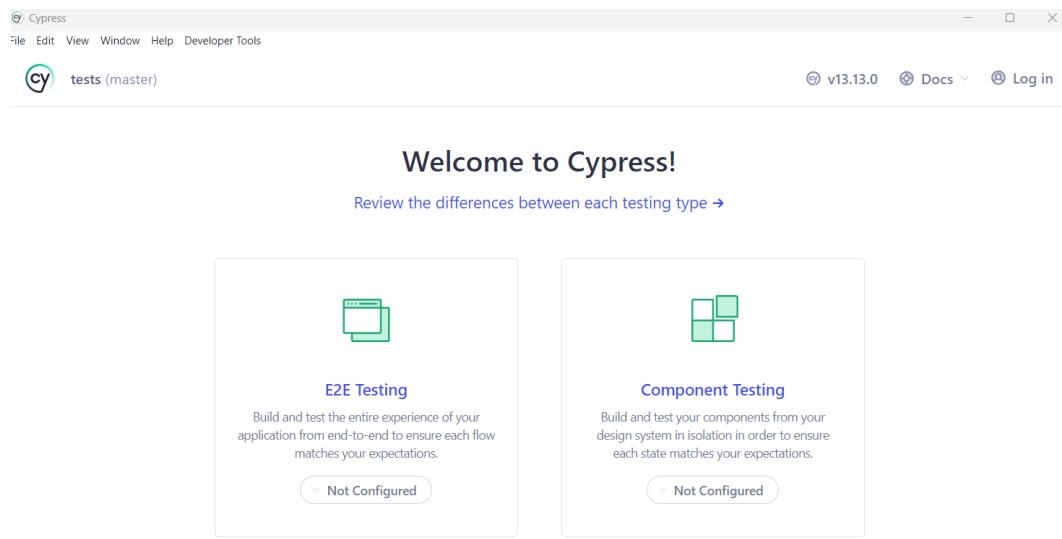


Figura 6.5: Pantalla de bienvenida Cypress. Fuente: Elaboración propia

Esta configuración incluye:

- **Seleccionar el tipo de prueba:** En este caso, se seleccionarán pruebas *end-to-end (e2e)*.
- **Seleccionar el archivo de configuración:** Se seleccionará el archivo *cypress.config.js*.
- **Seleccionar el navegador:** En este caso, se seleccionará el navegador *Chrome*.

Estas opciones permiten a los desarrolladores ajustar *Cypress* a sus necesidades específicas, asegurando que el entorno de pruebas esté optimizado para la aplicación que están desarrollando. Una vez completada la configuración, *Cypress* está listo para que se empiecen a escribir y ejecutar las pruebas, proporcionando una experiencia de prueba fluida y eficiente.

6.3.3. Estructura del proyecto de pruebas

La estructura del proyecto es fundamental para organizar y gestionar eficazmente tus pruebas automatizadas con *Cypress* en una aplicación *e-Commerce*. Una estructura organizada no solo facilita el mantenimiento y la escalabilidad de tus pruebas, sino que también ayuda a los desarrolladores y *testers* a navegar y comprender rápidamente el proyecto. A continuación como se muestra en el Bloque de Código 6.6 se muestra la estructura de carpetas del proyecto *Cypress*.

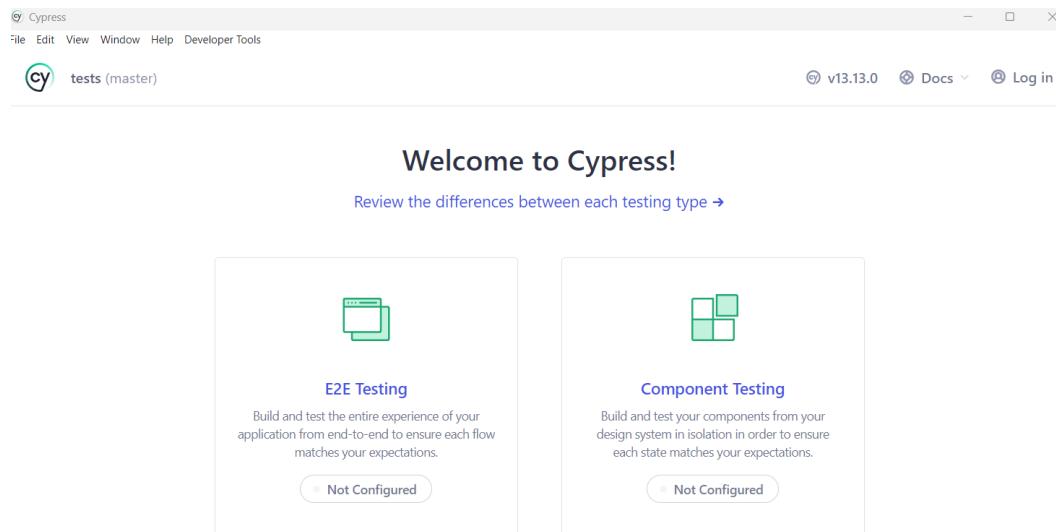


Figura 6.6: Estructura proyecto tests de Cypress. Fuente: Elaboración propia

A continuación se procede a la descripción de los archivos y carpetas:

- **cypress/fixtures/** Esta carpeta contiene archivos de datos de prueba que pueden ser usados en tus pruebas. Por ejemplo, `example.json` puede contener datos estáticos que se reutilizan en múltiples pruebas.
- **cypress/integration/** Esta es la carpeta principal donde se colocan todos los archivos de prueba. Cada subcarpeta dentro de `integration` corresponde a una pantalla o funcionalidad específica de tu aplicación *e-Commerce*. Esto ayuda a mantener las pruebas organizadas y fáciles de localizar.
 - `home/`: Contiene pruebas relacionadas con la pantalla principal del e-Commerce.
 - `signin/`: Contiene pruebas para la pantalla de inicio de sesión.
 - `signup/`: Contiene pruebas para la pantalla de registro de nuevos usuarios.
 - `product/`: Contiene pruebas para la pantalla de detalles de producto.
 - `cart/`: Contiene pruebas para la pantalla del carrito de compras.
 - `order/`: Contiene pruebas para la pantalla de detalles del pedido.
 - `payment/`: Contiene pruebas para la pantalla de método de pago.
 - `shipping/`: Contiene pruebas para la pantalla de dirección de envío.
 - `profile/`: Contiene pruebas para la pantalla de perfil de usuario.
 - `placeOrder/`: Contiene pruebas para la pantalla de confirmación de pedido.
- **cypress/plugins/** Esta carpeta es para la configuración de *plugins* de *Cypress*. El archivo `index.js` se usa para cargar y configurar los *plugins* que tu proyecto necesita.
- **cypress/support/** Esta carpeta contiene archivos de soporte y configuraciones globales para tus pruebas.
 - `commands.js`: Aquí puedes definir comandos personalizados que se pueden reutilizar en tus pruebas.
 - `index.js`: Archivo de configuración que se ejecuta antes de cada archivo de prueba. Es útil para configurar comportamientos globales.

- **cypress.json** Archivo de configuración principal de *Cypress* donde puedes definir opciones de configuración como la *URL* base, el tiempo de espera predeterminado y otras opciones de *Cypress*.

Una estructura organizada en un proyecto de pruebas automatizadas como *Cypress* para una aplicación *e-Commerce* ofrece beneficios significativos:

Facilita la mantenibilidad al organizar las pruebas de manera clara y ordenada, lo que permite realizar actualizaciones y correcciones de manera eficiente.

Promueve la escalabilidad al dividir las pruebas en carpetas que corresponden a diferentes partes de la aplicación (como inicio de sesión, carrito de compras, etc.), facilitando la adición de nuevas pruebas sin generar desorden.

Mejora la colaboración entre equipos al proporcionar una estructura comprensible y coherente, lo que ayuda a los desarrolladores y *testers* a trabajar de manera eficiente y coordinada.

Fomenta la reutilización de recursos al permitir la definición de comandos y configuraciones globales que pueden ser utilizados en múltiples pruebas, reduciendo la duplicación de código y manteniendo la consistencia en las pruebas.

En resumen, una estructura organizada no solo optimiza el proceso de desarrollo de software, sino que también asegura una gestión efectiva de las pruebas, facilitando la calidad y la eficiencia en todo el ciclo de vida del proyecto.

6.3.4. Pruebas en las distintas pantallas

Para crear un conjunto de pruebas para las pantallas de un *e-Commerce* se usará *Cucumber*. Esto implica escribir escenarios de prueba en lenguaje *Gherkin* que cubran diversas funcionalidades y casos de uso.

Gherkin es un lenguaje de especificación de comportamiento legible por humanos utilizado para definir pruebas en *Cucumber*, que es una herramienta que ejecuta esas pruebas automatizadas.

Gherkin y *Cucumber* son herramientas poderosas en el ámbito de las pruebas de *software* y la automatización, especialmente en metodologías ágiles y de desarrollo guiado por comportamiento.

En el Bloque de Código 6.6 se muestran el conjunto de pruebas en Cucumber del e-Commerce.

Bloque de código 6.6: Ejemplo de tests en Cucumber

```

1 Feature: Pruebas de las pantallas de e-Commerce
2
3   Background:
4     Given Estoy en la página de inicio de "laManchaCommerce"
5
6   Scenario: Iniciar sesión en la aplicación
7     Given Estoy en la pantalla de "Iniciar Sesión"
8     When Ingreso credenciales válidas
9     And Hago clic en el botón "Iniciar Sesión"
10    Then Debería iniciar sesión correctamente
11
12  Scenario: Registrar un nuevo usuario
13    Given Estoy en la pantalla de "Registro"
14    When Lleno el formulario de registro con detalles válidos
15    And Hago clic en el botón "Registrarse"
16    Then Debería registrarme correctamente
17
18  Scenario: Ver detalles de un producto
19    Given Estoy en la pantalla de "Inicio"
20    When Hago clic en un producto
21    Then Debería estar en la pantalla de "Detalles del Producto"
22
23  Scenario: Agregar un producto al carrito de compras

```

```

24 Given Estoy en la pantalla de "Detalles del Producto"
25 When Hago clic en "Añadir al Carrito"
26 And Voy a la pantalla de "Carrito"
27 Then Debería ver el producto añadido en mi carrito
28
29 Scenario: Realizar un pedido
30 Given Tengo artículos en mi carrito en la pantalla de "Carrito"
31 And Estoy en la pantalla de "Carrito"
32 When Procedo al checkout
33 And Seleccióno métodos de envío y pago en las pantallas de ↵
   ↵ "Checkout"
34 And Realizo el pedido
35 Then Debería ver la confirmación del pedido en la pantalla ↵
   ↵ de "Confirmación de Pedido"
36
37 Scenario: Actualizar el perfil de usuario
38 Given Estoy conectado en la pantalla de "Perfil"
39 When Actualizo la información de mi perfil
40 And Guardo los cambios
41 Then Mi información de perfil debería actualizarse ↵
   ↵ correctamente

```

Este conjunto de pruebas cubre funcionalidades clave como iniciar sesión, registrarse, ver detalles de productos, gestionar el carrito de compras, realizar pedidos y actualizar el perfil de usuario. Cada escenario puede expandirse según los requisitos específicos y las características de la aplicación.

Aquí tienes un ejemplo de un escenario de pruebas escrito en *Gherkin* y cómo se vería su implementación en código utilizando *Cucumber* con *JavaScript*.

En el archivo de definiciones de pasos de prueba (*step definitions*), que está ubicado en el directorio `cypress/integration/common/step_definitions`, se escribirá el código que se muestra en el Bloque de Código 6.7 que describe cómo ejecutar cada paso del escenario.

Bloque de código 6.7: Steps de Cucumber

```

1 // Definición del paso 'Given'
2 Given('Estoy en la pantalla de {string}', (screenName) => {
3     // Implementación para navegar a la pantalla específica
4     cy.visit('/login'); // Ejemplo de URL de inicio de sesión
5 });
6
7 // Definición del paso 'When'
8 When('Ingreso credenciales válidas', () => {
9     // Implementación para ingresar credenciales válidas
10    cy.get('input[name="username"]').type('usuario@example.com');
11    cy.get('input[name="password"]').type('contraseñaSegura');
12 });
13
14 When('Hago clic en el botón {string}', (buttonText) => {
15     // Implementación para hacer clic en el botón de inicio de sesión
16     cy.contains(buttonText).click();
17 });
18
19 // Definición del paso 'Then'
20 Then('Debería iniciar sesión correctamente', () => {
21     // Implementación para verificar que se haya iniciado sesión ↵
   ↵ correctamente
22     cy.url().should('include', '/dashboard'); // Ejemplo de URL ↵
   ↵ de dashboard después del inicio de sesión
23 });

```

Esta implementación muestra cómo los escenarios de prueba escritos en *Gherkin* se convierten en código ejecutable utilizando *Cucumber* con *Cypress* en *JavaScript*. Cada paso del escenario se

mapea a funciones específicas que realizan acciones y verificaciones en la aplicación bajo prueba, garantizando así el comportamiento esperado según las especificaciones descritas en *Gherkin*.

En el Bloque de Código 6.7 se destacan métodos importantes para interactuar con la aplicación utilizando *Cypress*:

- *cy.visit*: Este método se utiliza para navegar a una *URL* específica, en este caso, la página de inicio de sesión de la aplicación. Esto simula el inicio de sesión del usuario en la aplicación.
- *cy.get* y *type*: *cy.get* se utiliza para seleccionar elementos en la página, como campos de entrada de texto, botones, etc. *type* es un comando que simula la entrada de texto del usuario en esos elementos seleccionados. Esto es útil para llenar formularios o realizar acciones específicas en la interfaz de usuario.
- *cy.url().should*: Este método se utiliza para verificar la *URL* actual en la que se encuentra el navegador durante la ejecución de la prueba. El uso de *.should('include', '/dashboard')* asegura que la *URL* contenga la cadena */dashboard*, indicando que el usuario ha iniciado sesión correctamente y ha sido redirigido a su panel de control o *dashboard*.

Estos métodos son fundamentales para realizar pruebas de interfaz de usuario utilizando *Cypress*, ya que permiten simular las acciones del usuario y verificar que la aplicación responda correctamente a esas acciones, como el inicio de sesión y la navegación a páginas específicas.

6.4. DESARROLLO DEL CI/CD EN LA APLICACIÓN

6.4.1. Introducción

Una vez configurados los entornos de *Render* donde estarán alojadas nuestras aplicaciones web, así como configurado nuestro repositorio *GitHub* con todo el código desarrollado y un conjunto de pruebas para el *e-Commerce*, se procede a seguir uno de los principios de *DevOps*: automatizar todas estas configuraciones. Hasta ahora se ha explicado cómo hacerlo de forma manual.

Para lograr esto, vamos a diseñar una canalización de integración continua y despliegue continuo (*CI/CD*). La idea es que cuando se cree una *pull request* de la rama *release* con respecto a *master*, y esta *pull request* sea aprobada por los integrantes del equipo y aceptada, se integrará con *master*. Esto activará la canalización *CI/CD*, que automáticamente desplegará la nueva versión del *e-Commerce* en el entorno de integración. Posteriormente, después de la integración, se ejecutarán las pruebas *end-to-end* desarrolladas en *Cypress*. Si estas pruebas resultan exitosas, se procederá a desplegar la versión en producción y se volverán a ejecutar las pruebas para asegurar que todo esté funcionando correctamente en producción.

6.4.2. Configuración archivo YAML

Como se explicó en el apartado 4.2.2, *GitHub Actions* es un servicio de automatización integrado directamente en *GitHub* que permite automatizar tareas de desarrollo como la construcción, pruebas y despliegue de tu código desde tu repositorio en *GitHub*.

Para definir este flujo de trabajo, se utiliza un archivo *YAML* (*.yml*). A continuación, se procederá a realizar una descripción detallada del archivo *YAML* utilizado en la configuración de la canalización *CI/CD* de este TFG:

- **Eventos que Activan el Flujo de Trabajo:** Primero, se deben declarar los eventos que activan el flujo de trabajo. Como se muestra en el Bloque de Código 6.8, este flujo de trabajo se activa tanto cuando se realiza un *push* directo a la rama *master* como cuando se abre una *pull request* hacia la misma rama.

Bloque de código 6.8: Parte 1 del código del archivo *YAML*

```

1 | on:
2 |   push:

```

```

3   branches:
4     - master
5   pull_request:
6     branches:
7       - master

```

- **Trabajo *build-and-deploy-int*:** Se define el trabajo *build-and-deploy-int*, el cual construye y despliega la aplicación en un entorno de integración. Como se observa en el Bloque de Código 6.9, este trabajo se ejecuta en Ubuntu y solo si una *pull request* ha sido fusionada en la rama *master*.

El proceso para desplegar una aplicación en *Render* se estructura en varios pasos cruciales:

Primero, clonas el repositorio de tu aplicación dentro del entorno de ejecución de *Render*. Esto garantiza que tienes la última versión del código en el entorno donde será desplegada.

Luego, asegúrate de configurar *Node.js* utilizando la versión 14 específicamente. Esto se logra mediante la instalación adecuada de *Node.js* en *Render* o mediante la configuración del entorno para que utilice la versión correcta.

Después, ejecutas `npm run build` para compilar y construir la aplicación. Este paso es crucial para preparar los archivos estáticos y optimizar la aplicación para el entorno de producción.

Una vez que la aplicación está construida correctamente, utilizas `npm start` para iniciar la aplicación en *Render*. Este comando asegura que la aplicación se ejecute de manera adecuada y esté lista para ser utilizada por los usuarios.

Finalmente, para desplegar la aplicación en *Render* de manera formal, utilizas la *API* proporcionada por *Render*. Es esencial configurar correctamente los secretos en *Render*, asegurando que cualquier información sensible esté protegida y que la aplicación pueda acceder a las configuraciones necesarias sin problemas.

Bloque de código 6.9: Parte 2 del código del archivo YAML

```

1 jobs:
2   build-and-deploy-int:
3     runs-on: ubuntu-latest
4     if: github.event.pull_request.merged == true && ↵
5       ↵ github.event.pull_request.base.ref == 'master'
6     steps:
7       - name: Clonar repositorio
8         uses: actions/checkout@v2
9
10      - name: Configurar Node.js
11        uses: actions/setup-node@v2
12        with:
13          node-version: '14'
14
15      - name: Comando de Build
16        run: npm run build
17
18      - name: Comando de Inicio
19        run: npm start
20
21      - name: Desplegar en Render (Integración)
22        env:
23          RENDER_SERVICE_ID: ${{ ↵
24            ↵ secrets.RENDER_INT_SERVICE_ID }}
25          RENDER_API_KEY: ${{ secrets.RENDER_API_KEY }}
26        run:
27          |
            curl -X POST \
              -H "Authorization: Bearer $RENDER_API_KEY" \
              -H "Accept: application/json" \

```

```

28     -H "Content-Type: application/json" \
29     --data '{"serviceId": "$RENDER_SERVICE_ID", ↩
30       ↩ "clearCache": false}', \
      https://api.render.com/v1/services/$RENDER_SERVICE_ID/deploy

```

- **Trabajo *test-int*:** Una vez completado el despliegue inicial, procede a ejecutar los *tests* de integración para verificar que todas las partes de la aplicación funcionan correctamente en el entorno específico de Render (ver Bloque de Código 6.10). Estos *tests* son fundamentales para asegurar que la aplicación responda adecuadamente a las interacciones esperadas.

Posteriormente, es recomendable esperar al menos 5 minutos (300 segundos) antes de proceder con las siguientes acciones. Este tiempo de espera permite asegurarse de que el despliegue esté completamente finalizado y que la aplicación esté operativa sin problemas.

Finalmente, utiliza *Cypress*, una herramienta especializada en pruebas de extremo a extremo, para ejecutar *tests* automatizados en la *URL* de integración de tu aplicación desplegada en *Render*. Esto te permitirá verificar que todas las funcionalidades importantes de la aplicación funcionen correctamente y que no haya errores evidentes en la experiencia del usuario.

Bloque de código 6.10: Parte 3 del código del archivo YAML

```

1 test-int:
2   runs-on: ubuntu-latest
3   needs: build-and-deploy-int
4
5   steps:
6     - name: Clonar repositorio
7       uses: actions/checkout@v2
8
9     - name: Configurar Node.js
10       uses: actions/setup-node@v2
11       with:
12         node-version: '14'
13
14     - name: Esperar a que termine el despliegue
15       run: sleep 300
16
17     - name: Ejecutar pruebas Cypress
18       run: npm run test
19       env:
20         CYPRESS_BASE_URL: ${{ secrets.CYPRESS_INT_BASE_URL }}

```

- **Trabajo *build-and-deploy-pro*** Si las pruebas en el entorno de integración han sido exitosas, se procede al despliegue en producción. Los pasos son similares a los definidos en *build-and-deploy-int*.
- **Trabajo *test-pro*** Finalmente, para confirmar que todo está correcto en producción, se ejecutan pruebas específicas en el entorno de producción. Los pasos son similares a los definidos en *test-int*.

6.4.3. Configuración GitHub Actions

Una vez finalices la configuración de este archivo YAML, súbelo al repositorio de la aplicación dentro de la carpeta *.github*. Una vez cargado en *GitHub*, el sistema leerá automáticamente el archivo YAML y configurará la canalización *CI/CD* de la aplicación. La Figura 6.7 muestra cómo se ve la canalización *CI/CD* recién configurada en *GitHub Actions* según lo especificado en el archivo YAML.

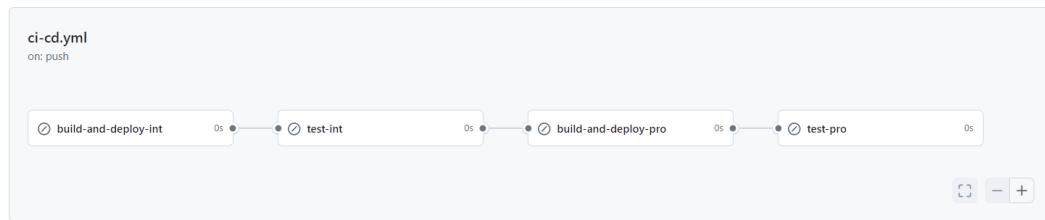


Figura 6.7: Esquema CI CD GitHub Actions. Fuente: Elaboración propia

Es crucial configurar correctamente las variables de entorno en *GitHub Actions* para que puedan ser utilizadas en el archivo *YAML*, por ejemplo, utilizando `$${{secrets.RENDER_API_KEY}}`. Estas variables se añaden en el apartado de secrets del repositorio en *GitHub*.

En la Figura 6.8 se muestran las variables de entorno necesarias para esta canalización, que incluyen la clave de la *API de Render*, los IDs de servicio de *Render* para las versiones de prueba y producción.

Repository secrets		New repository secret
Name	Last updated	
CYPRESS_INT_BASE_URL	1 hour ago	
RENDER_API_KEY	1 hour ago	
RENDER_INT_SERVICE_ID	1 hour ago	
RENDER_PRO_SERVICE_ID	1 hour ago	

Figura 6.8: Definición de Secrets GitHub . Fuente: Elaboración propia

CAPÍTULO 7

Resultados obtenidos

7.1. LAMANCHACOMMERCE. DESCRIPCIÓN Y CARACTERÍSTICAS

La aplicación desarrollada en este TFG se llama *laManchaCommerce*. *laManchaCommerce* es una plataforma de comercio electrónico (*e-Commerce*) de tipo *B2C*, enfocada en la venta de productos procedentes de la Mancha, destacando especialmente el vino.

Se trata de una aplicación visualmente atractiva e intuitiva para su uso, compatible con cualquier navegador web en cualquier dispositivo. La interfaz ha sido diseñada para ofrecer una experiencia de usuario fluida y agradable, facilitando la navegación y la interacción con los productos.

En esta aplicación, los usuarios pueden iniciar sesión o registrarse si aún no tienen una cuenta. Podrán navegar por los productos más destacados, ver sus características, añadirlos al carrito de compras y finalizar el pedido, proporcionando su dirección de envío y seleccionando el método de pago, que es *PayPal*.

7.2. RESULTADO FINAL

7.2.1. Pantalla Inicial

Al entrar en la aplicación, se mostrará la pantalla inicial, donde aparecerán todos los productos más destacados. En la esquina superior, se encuentra la opción de iniciar sesión, y en la parte inferior, se puede ver información sobre *laManchaCommerce* y enlaces a las redes sociales. A continuación, se presenta la pantalla inicial de *laManchaCommerce*, como se puede observar en la Figura 7.1.

7.2.2. Pantalla de Iniciar Sesión

Para poder utilizar el *e-Commerce*, es necesario iniciar sesión; de lo contrario, no podrás realizar ninguna compra. En esta pantalla, se mostrará un formulario para iniciar sesión con tu correo electrónico y contraseña, junto con un enlace que te redirigirá si no tienes una cuenta registrada. A continuación, se presenta la pantalla de iniciar sesión de *laManchaCommerce*, como se puede observar en la Figura 7.2.

The screenshot displays the homepage of the laManchaCommerce website. At the top, there is a header bar with the logo 'laManchaCommerce' and a user icon labeled 'Cart 0 prueba'. Below the header, a section titled 'Productos Destacados' shows a grid of seven wine bottles. The first six bottles have their details visible, while the seventh bottle's details are partially visible or obscured.

Nombre del Vino	Valoración	Precio	Opciones
Tempranillo Vino Tinto	★★★★☆ 10 valoraciones	4 €	Añadir al carrito
Vino Tinto Reserva	★★★★☆ 10 valoraciones	30 €	Añadir al carrito
Vino Tinto Roble Selección	★★★★☆ 10 valoraciones	40 €	Añadir al carrito
Vino Tinto Graciano	★★★★☆ 10 valoraciones	10 €	Añadir al carrito
Vino Tinto Syrah	★★★★☆ 10 valoraciones	12 €	Añadir al carrito
Vino Tinto Cabernet Sauvignon	★★★★☆ 10 valoraciones	17 €	Producto agotado

At the bottom of the page, there is a footer bar containing contact information, social media links (Facebook, Twitter, Instagram), and a copyright notice: '© 2024 Todos los derechos reservados'.

Figura 7.1: Pantalla Inicial *laManchaCommerce*. Fuente: Elaboración propia

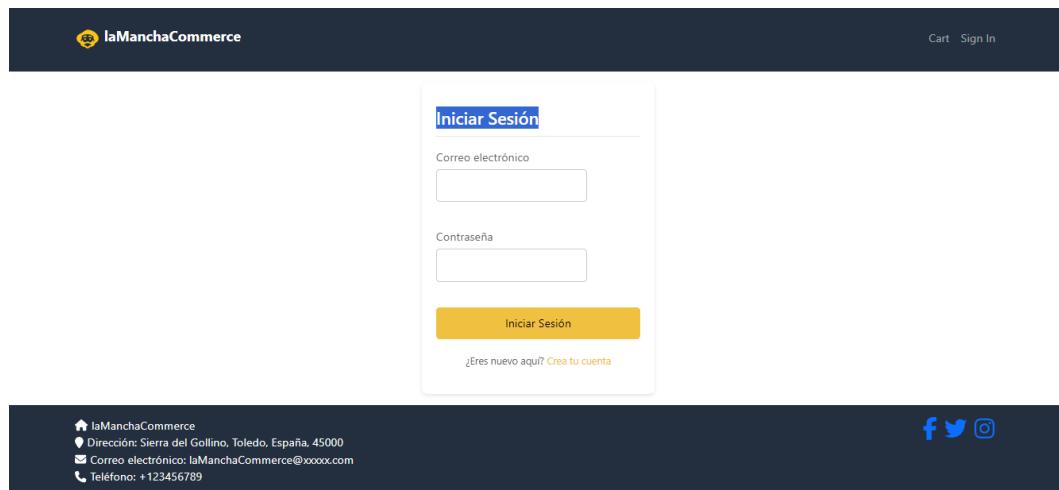


Figura 7.2: Pantalla de Iniciar Sesión *laManchaCommerce*. Fuente: Elaboración propia

7.2.3. Pantalla de Registro

En esta pantalla, se podrá crear una cuenta para *laManchaCommerce* si aún no se dispone de una. Los campos requeridos para crear una cuenta son: nombre, correo electrónico, contraseña y confirmación de contraseña. La contraseña debe tener más de seis caracteres y debe coincidir exactamente con la introducida en el campo de confirmación de contraseña. Además, al crear la cuenta, se aceptará una serie de Condiciones de Uso y Política de Privacidad.

Este proceso asegura que los usuarios puedan registrarse de manera segura y con pleno conocimiento de las condiciones bajo las cuales utilizarán la aplicación *laManchaCommerce*. A continuación, se presenta la pantalla de registro de *laManchaCommerce*, como se puede observar en la Figura 7.3.

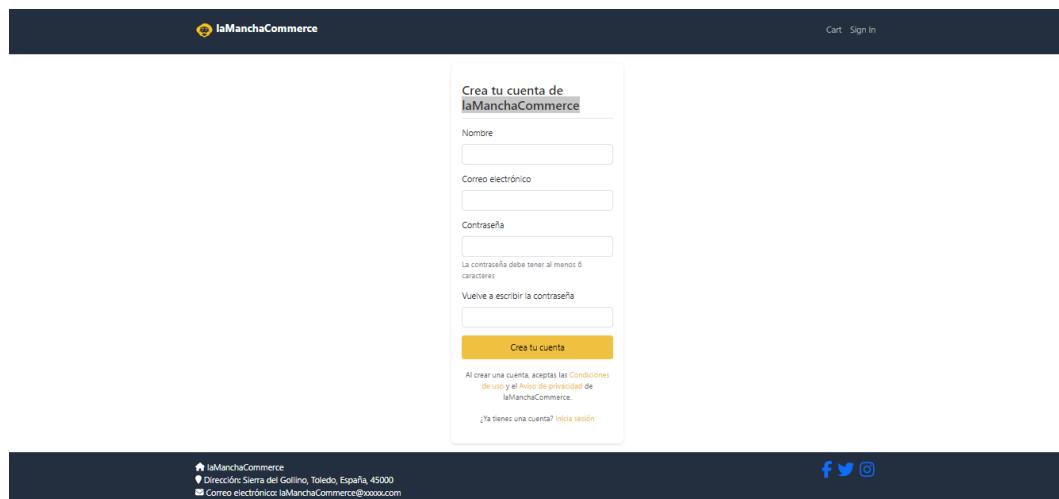


Figura 7.3: Pantalla de Registro *laManchaCommerce*. Fuente: Elaboración propia

7.2.4. Pantalla de Información del Producto

Para acceder a la información detallada de cada producto, los usuarios podrán visitar la página de Información del Producto. En esta sección, podrán encontrar detalles como el nombre del producto, su valoración, precio, origen, certificaciones y una descripción detallada del mismo. Si el producto está disponible en stock, tendrán la opción de añadirlo al carro de compras directamente desde esta página. A continuación, se presenta la pantalla de registro de *laManchaCommerce*, como se puede observar en la Figura 7.4.

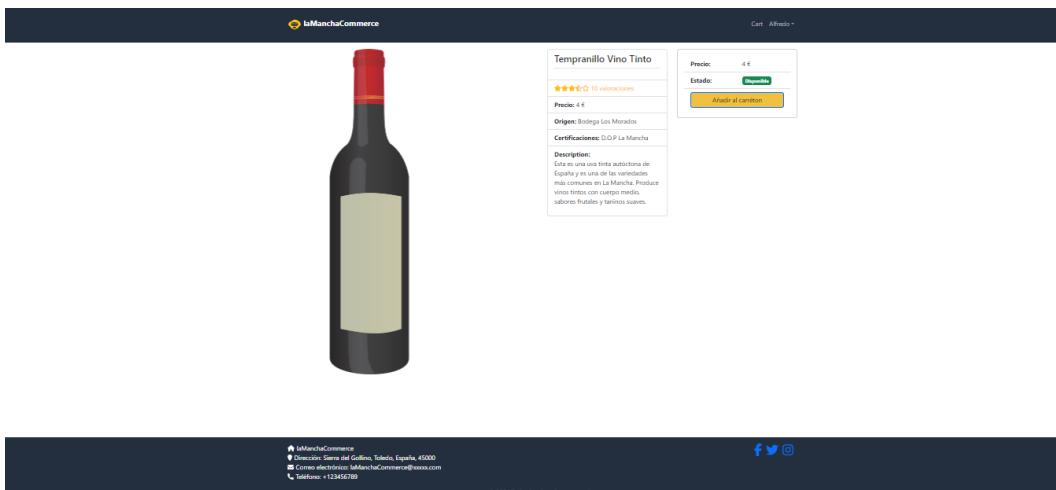


Figura 7.4: Pantalla de Información del Producto *laManchaCommerce*. Fuente: Elaboración propia

7.2.5. Pantalla de Carro de Compras

Una vez que se hayan añadido todos los productos de *laManchaCommerce* que se desean comprar, los usuarios serán dirigidos a la pantalla del Carro de Compras. En esta pantalla, podrán visualizar todos los productos que han sido añadidos al carro junto con la cantidad de cada uno. Tendrán la opción de aumentar o disminuir las unidades de cada producto, así como eliminar productos individuales si así lo desean.

En la tarjeta de la derecha de la pantalla, se mostrará el precio total de los productos seleccionados. También encontrarán un botón para continuar con el proceso de compra y proceder al pago.

Esta pantalla está diseñada para proporcionar a los usuarios un resumen claro y detallado de su selección de productos, permitiéndoles realizar ajustes antes de finalizar la compra.

A continuación, se presenta la pantalla de registro de *laManchaCommerce*, como se puede observar en la Figura 7.5.

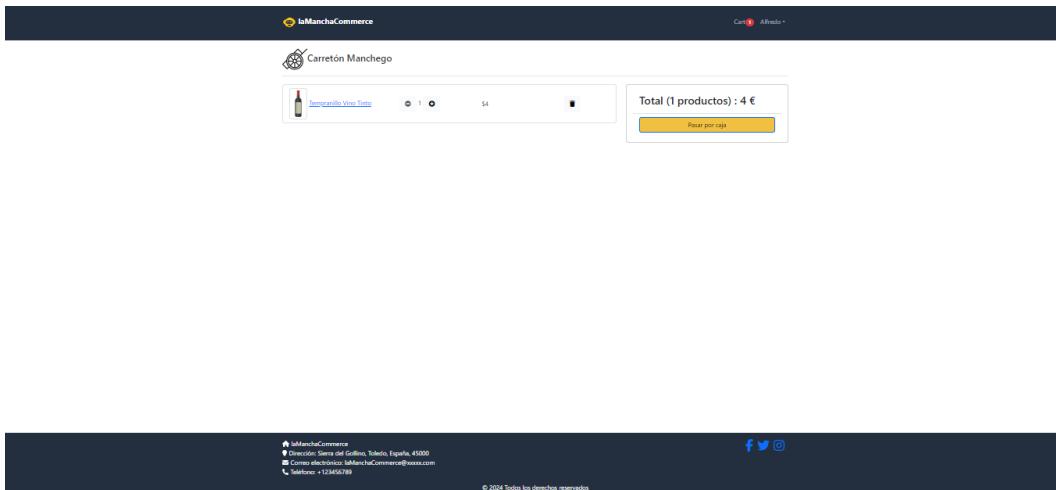


Figura 7.5: Pantalla de Carro de Compras de *laManchaCommerce*. Fuente: Elaboración propia

7.2.6. Pantalla de Dirección de Envío

En esta pantalla se introducirán los datos de envío para completar la compra, incluyendo la dirección completa, ciudad, código postal y país. A continuación, se presenta la pantalla de dirección de envío de *laManchaCommerce*, como se puede observar en la Figura 7.6.

laManchaCommerce

Iniciar Sesión | Entra | Pago | Realizar Pedido

Agregar una nueva dirección de envío

Nombre completo:

Dirección (Línea 1):

Ciudad:

Código postal:

País:

Selección una opción

GUARDAR DIRECCIÓN

© 2024 Todos los derechos reservados

Figura 7.6: Pantalla de Dirección de Envío de *laManchaCommerce*. Fuente: Elaboración propia

7.2.7. Pantalla de Método de Pago

En esta pantalla se selecciona el método de pago, que será a través de *PayPal*. A continuación, se presenta la pantalla de dirección de envío de *laManchaCommerce*, como se puede observar en la Figura 7.7.

Artículo	Cantidad	Precio
Camiseta de algodón	1	100,00 €
Total		100,00 €
Envío		0,00 €
Iva		0,00 €
Total a pagar		100,00 €

Realizar Pedido

Realizar Pedido

© 2024 Todos los derechos reservados

Figura 7.7: Pantalla de Método de Pago de *laManchaCommerce*. Fuente: Elaboración propia

7.2.8. Pantalla de Realizar Pedido

En esta pantalla se mostrará un resumen detallado del pedido antes de confirmarlo. Los usuarios podrán revisar la dirección de envío ingresada previamente, confirmar el método de pago seleccionado (*PayPal* en este caso), y verificar los detalles completos de los productos que están a punto de comprar, incluyendo nombres, cantidades y precios unitarios.

A la derecha, una tarjeta mostrará el precio total del pedido, desglosando claramente el IVA y los gastos de envío si los hubiera. Este resumen proporciona una última oportunidad para que los usuarios revisen y ajusten su pedido antes de proceder. Un botón destacado al final de la pantalla permitirá a los usuarios confirmar y completar la compra de manera segura y sencilla.

A continuación, se presenta la pantalla de realizar pedido de *laManchaCommerce*, como se puede observar en la Figura 7.7.

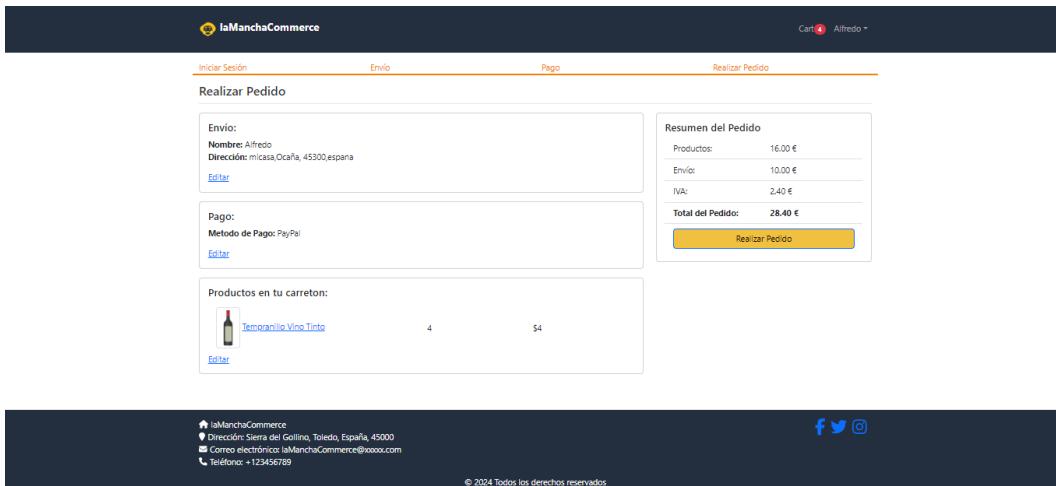


Figura 7.8: Pantalla de Realizar Pedido de *laManchaCommerce*. Fuente: Elaboración propia

7.2.9. Pantalla de Pedido

En esta pantalla, los usuarios pueden ver el estado actual de su pedido, indicando si está entregado y si está pagado. Si el pedido aún no está pagado, a la derecha aparecerá un botón de *PayPal*. Al hacer clic en este botón, serán redirigidos a la aplicación de *PayPal* para completar el pago, como se muestra en la Figura 7.9.

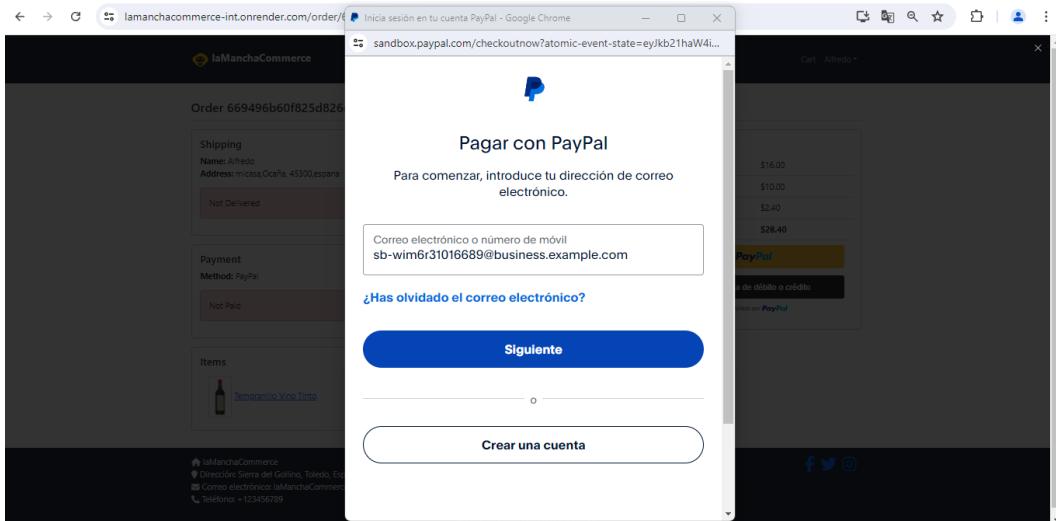


Figura 7.9: Pop-up de PayPal de *laManchaCommerce*. Fuente: Elaboración propia

A continuación, se presenta la pantalla del pedido de *laManchaCommerce*, como se puede observar en la Figura 7.10.

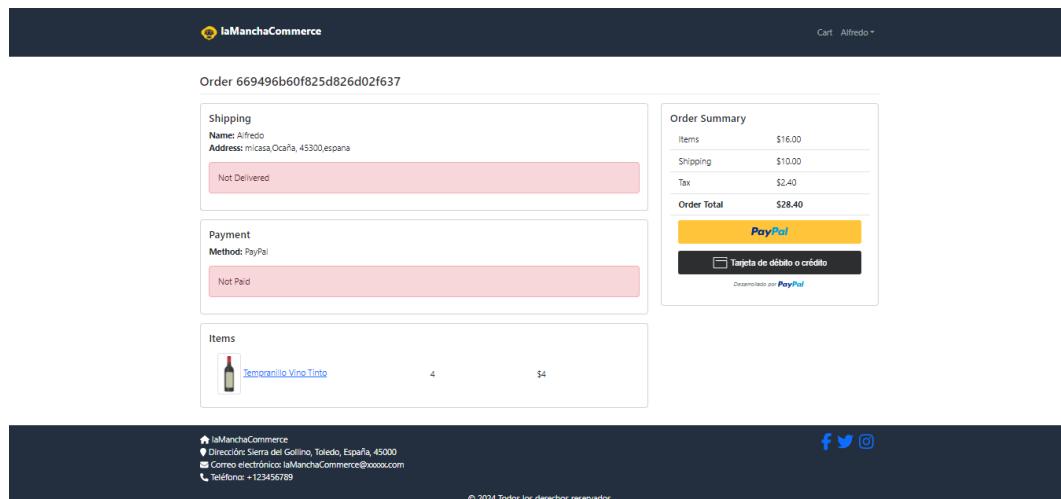


Figura 7.10: Pop-up de PayPal de laManchaCommerce. Fuente: Elaboración propia

CAPÍTULO 8

Conclusiones y líneas futuras

8.1. CONCLUSIONES

En conclusión, el desarrollo del ecommerce *laManchaCommerce* utilizando React y la arquitectura *MERN* ha sido un proceso transformador que ha permitido integrar tecnologías modernas para ofrecer una experiencia de usuario robusta y escalable. La combinación de *MongoDB* para la base de datos, *Express.js* y *Node.js* para el backend, junto con React en el *frontend*, ha facilitado la creación de una aplicación web dinámica y eficiente.

La implementación de la canalización *CI/CD* como parte de los procesos *DevOps* ha sido fundamental para garantizar la entrega continua y la calidad del *software* desarrollado.

En resumen, *laManchaCommerce* no solo representa un logro tecnológico significativo, sino también un compromiso continuo con la innovación y la excelencia en el desarrollo de aplicaciones web basadas en el stack *MERN*. Con un enfoque en la mejora continua y la adaptabilidad a las necesidades del mercado, estamos preparados para seguir evolucionando y ofreciendo una experiencia de usuario excepcional en el entorno competitivo del comercio electrónico actual.

8.2. LÍNEAS FUTURAS

Partiendo del desarrollo de este proyecto y con la intención de implementar futuras mejoras se proponen las siguientes:

En primer lugar, continuar agregando características que enriquezcan la experiencia del usuario, como opciones de personalización avanzada, integración con redes sociales para compartir productos de manera integrada y fluida, y sistemas de recomendación basados en inteligencia artificial para ofrecer sugerencias personalizadas y relevantes.

Por un lado, extender los principios *DevOps* a otras áreas del ciclo de vida del desarrollo del *software*, más allá de la automatización de la integración y entrega (*CI/CD*).

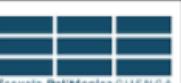
Por otro lado, implementar la automatización de pruebas de seguridad para detectar y mitigar vulnerabilidades potenciales de manera proactiva. Asimismo, automatizar pruebas de rendimiento para optimizar el rendimiento del sistema y garantizar una experiencia fluida para los usuarios, incluso bajo cargas elevadas.

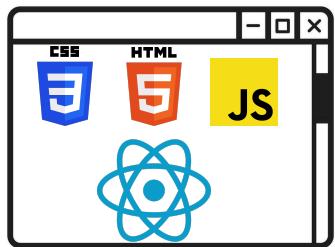
Por último, considerar el despliegue de la aplicación en un entorno más potente en la nube, como *Amazon Web Services (AWS)*, para aprovechar la escalabilidad, seguridad y disponibilidad que ofrecen servicios como *EC2*, *S3* y *AWS Lambda*. Esto permitirá manejar eficientemente el crecimiento del tráfico y la demanda sin comprometer el rendimiento.

Parte II: Planos

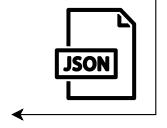
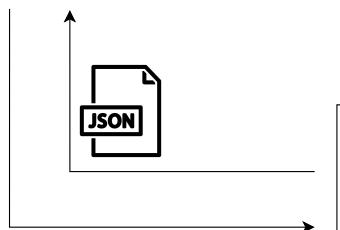
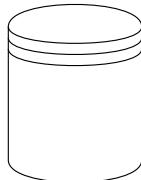


LEYENDA	
	Inicio y fin del proceso
	Actividades o procesos
	Decisiones

Universidad de Castilla – La Mancha Escuela Politécnica de Cuenca  	
Proyecto	Trabajo Fin de Grado
	Desarrollo de e-Commerce con React aplicando procesos DevOps
Autor	Alfredo Crespo Fernández-Tostado
Sin escala	Diagrama de flujo del e-Commerce
	P-01



mongoDB



Cliente(Frontend)

Servidor(Backend)

Universidad de Castilla – La Mancha
Escuela Politécnica de Cuenca



Trabajo Fin de Grado

Proyecto

Desarrollo de e-Commerce con React aplicando procesos DevOps

Autor

Alfredo Crespo Fernández-Tostado

Sin escala

Arquitectura del
e-Commerce

P-02

Parte III: Pliego de condiciones

CAPÍTULO 9

Pliego de Condiciones

9.1. HARDWARE

Durante el desarrollo de la aplicación, se utilizó un ordenador portátil conectado a un monitor externo para disponer de una pantalla más amplia, lo cual facilitó el diseño de la interfaz gráfica.

A continuación, se detallan las especificaciones técnicas de todos los componentes hardware utilizados:

- **Lenovo IdeaPad Gaming 3.** Equipo indispensable para el desarrollo de la aplicación y la elaboración del presente documento. En la Tabla 11 se resumen sus especificaciones técnicas.

Tabla 9.1: Especificaciones técnicas

Sistema operativo	Windows 11 Pro 64-bit
Procesador	AMD Ryzen 7 6800H
Memoria RAM	16 GB
Disco duro	SSD de 512 GB
Tarjeta gráfica	NVIDIA GeForce RTX 3050Ti



Figura 9.1: Ordenador HP Elitebook 840 G5. Fuente: PCComponentes

- **Philips V Line - Monitor.** La pantalla externa se utilizó para diseñar las interfaces gráficas con mayor comodidad, así como para maquetar el presente documento. En la Tabla 12 se resumen sus principales especificaciones técnicas.

Tabla 9.2: Especificaciones del Monitor Philips

Marca	Philips
Tamaño de pantalla	24 Pulgadas
Resolución	FHD 1080p
Relación de aspecto	16:9
Descripción de la superficie de la pantalla	Brillante
Relación de contraste de la imagen	10,000,000:1
Tiempo de respuesta	4 Milisegundos
Frecuencia de actualización	76 Hz
Resolución de pantalla máxima	1080p Full HD

**Figura 9.2:** Monitor Philips V Line. Fuente: Amazon

9.1.1. Software

Para el desarrollo del trabajo se utilizaron las siguientes herramientas software:

- **Sistema operativo:** El sistema operativo de nuestro ordenador portátil es Windows 10.
- **Visual Studio Code** (versión 1.91.1). Visual Studio Code es un editor de código fuente desarrollado por Microsoft para Windows, Linux y macOS. Ha sido el utilizado para realizar el desarrollo de la aplicación web y de las pruebas automatizadas.
- **Google Chrome** (versión 100.0.4896.60): Navegador web utilizado por defecto para visualizar la aplicación.
- **React** (versión 18.0.0): Biblioteca de JavaScript utilizada para la construcción de la interfaz de usuario.
- **Node.js** (versión 17.5.0): Entorno de ejecución de JavaScript utilizado para la construcción del servidor y el manejo del backend.
- **Render**: Plataforma de alojamiento y despliegue de aplicaciones utilizada para hospedar la aplicación web desarrollada.
- **GitHub**: Plataforma de desarrollo colaborativo para almacenar el código y gestionar versiones.
- **Latex** (versión 2e): Sistema de composición de textos utilizado para la redacción de documentos científicos. Esta herramienta será empleada para redactar este documento.

- **Draw.io:** Aplicación web gratuita y de código abierto que permite a los usuarios crear diagramas de flujo, esquemas, organigramas y otros gráficos visuales de manera intuitiva y eficiente.

Parte IV: Presupuesto

CAPÍTULO 10

Presupuesto

A continuación se detallan los gastos económicos asociados al desarrollo del proyecto:

- **Hardware:** Costos del equipo especificado en el Pliego de Condiciones.
- **Software:** El software utilizado para el desarrollo del proyecto es de libre uso y gratuito, por lo tanto, no se incluirá en el Presupuesto.
- **Costos derivados del proyecto:** Honorarios y gastos del Ingeniero.
- **Presupuesto Final:** Cálculo realizado sumando los subtotales anteriores y añadiendo el 21 % de IVA.

10.1. HARDWARE

El coste de los equipos se ha calculado según la siguiente fórmula de amortización:

$$\text{Coste (€)} = \frac{\text{Precio del equipo}}{\text{Vida útil}} \times \frac{\text{Uso del equipo}}{365}$$

Tabla 10.1: Costos de Equipos

Concepto	Precio(€)	Vida útil (años)	Uso (días)	Total (€)
Lenovo IdeaPad Gaming 3	1029.00	5	160	97,61
Monitor Philips	129.1	5	160	11.30
Subtotal (€)				108.91

10.2. COSTES DERIVADOS DEL PROYECTO

Tabla 10.2: Costes derivados del proyecto

Concepto	Horas	Precio / Hora Desarrollo (€)	Coste Total Desarrollo (€)
Desarrollador Frontend	200	40	8000
Desarrollador Backend	100	40	4000
Ingeniero DevOps	50	50	2500
Subtotal (€)			14500

10.3. PRESUPUESTO FINAL**Tabla 10.3:** Presupuesto Final

Concepto	Coste (€)
Hardware	188,91
Costes Derivados	14500,00
Subtotal	14608,91
IVA (21 %)	3067,9
Total	17676,8

Bibliografía

- [1] Amazon Web Services. The difference between frontend and backend. <https://aws.amazon.com/es/compare/the-difference-between-frontend-and-backend/>. Accedido en fecha de 2024.
- [2] Amazon Web Services. AWS CloudFormation. <https://aws.amazon.com/es/cloudformation/>, 2024. Accedido en julio 2024.
- [3] Angular. Página oficial de angular. <https://angular.dev/>. Consultado junio de 2024.
- [4] Arizbé Ken . Backend: ¿Qué es y para qué sirve? <https://www.gluo.mx/blog/backend-que-es-y-para-que-sirve>, 2023. Accedido en julio 2024.
- [5] Atlassian. What is git. <https://www.atlassian.com/es/git/tutorials/what-is-git>. Accedido junio de 2024.
- [6] CNMC. El comercio electrónico superó en España los 20 mil millones de euros en el segundo trimestre de 2023, un 12,7 % más que el año anterior, 2024. Consultado junio de 2024.
- [7] Data Comunicación. Marketplaces en España 2024. <https://datacomunicacion.com/2024/03/25/marketplaces-en-espana-2024/>, 2024.
- [8] Cypress Documentation. Who uses Cypress. <https://docs.cypress.io/guides/overview/why-cypress#Who-uses-Cypress>, 2024. Accedido en julio 2024.
- [9] Dev Technosys. MERN Stack: The Complete Guide. <https://devtechnosys.com/insights/mern-stack/>, 2024. Accedido en julio 2024.
- [10] GitHub Docs. Understanding GitHub Actions. <https://docs.github.com/es/actions/learn-github-actions/understanding-github-actions>, 2024. Consultado junio de 2024.
- [11] F. Díaz. Documento técnico sobre sistemas distribuidos. https://www.infor.uva.es/~fdiaz/sd_2005_06/doc/SD_TE02_20060305.pdf, 2006. Accedido en julio 2024.
- [12] Elena Bello. El 83,7% <https://www.iebschool.com/blog/comercio-online-ecommerce/>, 2021. Accedido en julio 2024.
- [13] Fortinet. TCP/IP. <https://www.fortinet.com/lat/resources/cyberglossary/tcp-ip>, 2024. Accedido en julio 2024.
- [14] Martin Fowler. Continuous integration. <https://www.martinfowler.com/articles/continuousIntegration.html>. Consultado junio de 2024.
- [15] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, Boston, 2002.
- [16] Gart Solutions. What Turned Etsy into a DevOps Unicorn. <https://gartsolutions.medium.com/what-turned-etsy-into-a-devops-unicorn-5d646eaf54c8>, 2021. Accedido en julio 2024.
- [17] Git. Logo de git. Accedido el 14 de junio de 2024, 2024. <https://www.git-scm.com/>.
- [18] GitHub. Página oficial de github. <https://github.com/>. Consultado junio de 2024.
- [19] GitHub. About GitHub and Git, 2024. Accedido el 14 de junio de 2024.

- [20] Heroku. Heroku: Cloud Application Platform. <https://www.heroku.com/>, 2024. Accedido en julio 2024.
- [21] IBM. TCP/IP Protocols. <https://www.ibm.com/docs/es/aix/7.2?topic=protocol-tcip-protocols>, 2024. Accedido en julio 2024.
- [22] International Software Testing Qualifications Board (ISTQB). Istqb certified tester foundation level syllabus. <https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFLSyllabus2018v3,1,1.pdf>, 2018. Consultado junio de 2024.
- [23] MDN Web Docs. Css: Cascading style sheets. <https://developer.mozilla.org/es/docs/Web/CSS>, 2023. Consultado junio de 2024.
- [24] MDN Web Docs. Html: Hypertext markup language. <https://developer.mozilla.org/es/docs/Web/HTML>, 2023. Consultado junio de 2024.
- [25] Microsoft. What is git. <https://learn.microsoft.com/es-es/devops/develop/git/what-is-git>. Accedido de junio de 2024.
- [26] Microsoft. Visual Studio Code. <https://code.visualstudio.com/>, 2024. Accedido en julio 2024.
- [27] Module Counts. Module counts. <http://www.modulecounts.com/>, 2023. Consultado junio de 2024.
- [28] Manuel Montenegro. Apuntes de la asignatura aplicaciones. <https://github.com/manuelmontenegro/AW-2017-18>, 2024. Dpto. de Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid.
- [29] Mozilla Developer Network. HTTP methods. <https://developer.mozilla.org/es/docs/Web/HTTP/Methods>, 2024. Accedido en julio 2024.
- [30] Netflix Tech Blog. Global Continuous Delivery with Spinnaker. <https://netflixtechblog.com/global-continuous-delivery-with-spinnaker-2a6896c23ba7>, 2017. Accedido en julio 2024.
- [31] Página oficial de Node. Logo de node.js. <https://nodejs.org/en>. Consultado junio de 2024.
- [32] OpenWebinars. ¿qué es node.js? <https://openwebinars.net/blog/que-es-nodejs/>, 2023. Consultado junio de 2024.
- [33] Página de Angular. Introducción a los conceptos de Angular.
- [34] Página de Vue. ¿Qué es Vue.js?
- [35] QA News Blog. El verdadero valor del testing. <https://qanewsblog.com/2018/03/14/el-verdadero-valor-del-testing/>, 2018. Accedido en julio 2024.
- [36] React. Página oficial de react. <https://es.react.dev/>. Consultado junio de 2024.
- [37] Red Hat. What are Application Programming Interfaces (APIs). <https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces>, 2024. Accedido en julio 2024.
- [38] RedHat. ¿qué es una arquitectura de aplicaciones? URL: <https://www.redhat.com/es/topics/cloud-native-apps/what-is-an-application-architecture/>, 2023.
- [39] Render. About render. <https://render.com/about>. Consultado junio de 2024.
- [40] Render. Cloud application hosting for developers. <https://render.com/>. Consultado junio de 2024.
- [41] Render. Render: Modern Cloud to Build and Run Apps. <https://dashboard.render.com/>, 2024. Accedido en julio 2024.
- [42] Stack Overflow. Stack overflow developer survey 2023. <https://survey.stackoverflow.co/2023/most-popular-technologies-language-prof>, 2023. Consultado junio de 2024.
- [43] Vue. Página oficial de vue.js. <https://vuejs.org/>. Consultado junio de 2024.

- [44] Wikipedia. Logo css. https://es.m.wikipedia.org/wiki/Archivo:CSS3_logo_and_wordmark.svg. Consultado junio de 2024.
- [45] Wikipedia. Logo html. https://es.m.wikipedia.org/wiki/Archivo:HTML5_logo_and_wordmark.svg. Consultado junio de 2024.
- [46] Wikipedia. Logo js. <https://es.wikipedia.org/wiki/Archivo:JavaScript-logo.png>. Consultado junio de 2024.
- [47] Wikipedia. Logo svn. https://es.m.wikipedia.org/wiki/Archivo:Apache_Subversion_logo.svg. Consultado junio de 2024.
- [48] Xeridia. ¿Sabes realmente qué es DevOps? <https://www.xeridia.com/blog/sabes-realmente-que-es-devops>, 2024. Accedido en julio 2024.

ANEXOS

ANEXO A

Repositorio de GitHub

El código completo del TFG *Desarrollo de e-Commerce con React, aplicando procesos DevOps*, se encuentra disponible en el siguiente repositorio de GitHub:

- <https://github.com/acreft/eCommerceTfg>