

INFINITY COMICS

[Subtítulo del documento]

MEMORIA FINAL

TFG

INFINITY COMICS

ÍNDICE

1. Introducción

1.1 Objetivos del proyecto

1.2 Tecnologías

1.2.1 Frontend

1.2.2 Backend

2. Adecuación del entorno de trabajo

2.1 Cómo se creó el proyecto

2.1 Cómo importar el proyecto

3. Arquitectura del proyecto

3.1 Arquitectura y diseño

3.1.1 `.git`

3.1.2 `.vscode`

3.1.3 Backend

3.1.4 `creacionUsuario.sql`

3.1.5 Esquema de carpetas

3.1.6 Nuevo-Front

3.1.7 `README.md`

3.1.8 tarjeta_de_prueba.txt

3.1.9 webcomic_dump.sql

4. Análisis de requisitos

4.1 Login

4.2 Registro

4.3 Navbar

4.4 MainPage

4.5 ProductDetailPage

4.6 ProductDetailPage

4.7 UserProfile

4.8 TramitarPedido

5. Pruebas

6. Desarrollo del proyecto

6.1 Backend

6.2 Frontend

7. Desarrollo de Base de datos

8. Conclusiones

8.1 Dificultades

8.2 Alcance y Limitaciones

8.3 Posibles mejoras de futuro

8.4 Conclusiones

9. Bibliografía

Link al repositorio (ir a la rama: cambios-javi1):
<https://github.com/james200327/Webcomic.git>

1.INTRODUCCIÓN

Este documento es la memoria del Trabajo de Final de Grado (TFG) del ciclo formativo de DAW realizado por Javier Lopez Laureano y James Matos Melo.

Infinity Comics es una web preparada para gestionar la compra y venta de cómics manga.

1.1 OBJETIVOS DEL PROYECTO

Como finalidad el proyecto pretende construir una página web usable en la que se pueda gestionar una tienda virtual de comics

1. Aprender a manejar y afianzar nuevos conocimientos en Spring ya que no hemos llegado a tratar apartados como el Spring Security que creemos que nos será de utilidad aprender de cara a nuestra futura vida laboral.
2. Obtener una aplicación que sea funcional tanto en fines prácticos y demostrativos para quizá en un futuro si se sigue trabajando en ella una posible comercialización de este o algo que sea parecido.
3. Aprender a unir los dos módulos principales de este último curso en un proyecto.
4. Aprender tecnologías muy demandadas y de un uso extendido en el mercado laboral como es React.

1.2 TECNOLOGÍAS

Estas son las tecnologías y entornos que hemos usado para este proyecto

1.2.1 Frontend

React

- Biblioteca principal: React es el corazón de nuestro frontend. Nos permite crear interfaces de usuario interactivas y componentes reutilizables como Navbar, ProductCard y ModifyComicPage.
- Componentización: Los componentes individuales facilitan el mantenimiento y la escalabilidad. Por ejemplo, PointsPage maneja la venta de puntos, mientras que ModifyComicPage funciona como un modal específico para editar cómics.
- Routing: Estamos gestionando rutas con react-router-dom u otra herramienta similar, dado que navegamos entre páginas como PointsPage, MainPage y ProductDetailPage.

Vite

- Herramienta de construcción moderna: Vite reemplaza a herramientas tradicionales como create-react-app, ofreciéndonos tiempos de inicio y compilación mucho más rápidos.
- Optimización: Su enfoque en el "hot module replacement" y en la carga bajo demanda mejora significativamente nuestra experiencia de desarrollo.

JavaScript

Utilizamos JavaScript como lenguaje principal para manejar la lógica de la aplicación: control de estados, interacción con la API del backend, y renderizado dinámico de contenido.

CSS

Personalizamos el diseño de nuestra aplicación. Por ejemplo, preferimos usar botones estilizados para acciones como cargar imágenes o abrir modales.

npm

Controlador de dependencias: Usamos npm para instalar y gestionar paquetes como react-router-dom, axios, o cualquier otra biblioteca que utilizamos en el proyecto.

1.2.2 Backend

Spring Boot

- Framework robusto: Spring Boot facilita el desarrollo de nuestro backend, especialmente para servicios RESTful. Es ideal para nuestro proyecto, ya que necesitamos una API para manejar datos de productos, usuarios y compras.
- Integración sencilla: Ofrece soporte para conectar con bases de datos mediante JPA (Java Persistence API) y herramientas como Hibernate.
- Controladores y servicios: Creamos controladores REST que manejan rutas como /comics, /users y /cart, con métodos para consultar, actualizar y eliminar datos.
- Seguridad: Podríamos usar Spring Security para proteger endpoints y manejar la autenticación de usuarios.
- Sincronización con frontend: A través de endpoints, nuestro frontend en React se conecta al backend para leer o escribir datos, como en la sincronización de descripciones de cómics.

MySQL

- Base de datos relacional: Usamos MySQL para almacenar y gestionar la información estructurada de nuestra aplicación.

Tablas

Productos

id_producto: Identificador único del producto.

titulo: Título del cómic.

autor: Autor del cómic.

genero: Género del cómic.

precio: Precio del cómic.

stock: Cantidad disponible en inventario.

fecha_publicacion: Fecha de publicación del cómic.

Usuarios

id_usuario: Identificador único del usuario.

nombre: Nombre del usuario.

contrasena: Contraseña en formato hash (opcional dependiendo de la implementación).

email: Correo electrónico del usuario.

fecha_registro: Fecha en que el usuario se registró.

rol: Rol del usuario (por ejemplo, cliente o administrador).

direccion: Dirección del usuario.

Pedidos

id_pedido: Identificador único del pedido.

id_usuario: Identificador del usuario que realizó el pedido (relación con la tabla Usuarios).

estado: Estado del pedido (por ejemplo, "pendiente", "enviado", "entregado").

fecha_pedido: Fecha en que se realizó el pedido.

fecha_llegada: Fecha estimada o real de entrega del pedido.

Detalles_Pedido

id_pedido: Identificador del pedido al que pertenece el detalle (relación con la tabla Pedidos).

id_producto: Identificador del producto incluido en el pedido (relación con la tabla Productos).

cantidad: Cantidad del producto pedida.

precio_und: Precio por unidad del producto al momento de la compra.

Metodo_Pago

id_metodo_pago: Identificador único del método de pago.

tipo_metodo: Tipo de método de pago (por ejemplo, "tarjeta", "PayPal").

Transacciones

id_transaccion: Identificador único de la transacción.

id_pedido: Identificador del pedido asociado (relación con la tabla Pedidos).

id_metodo_pago: Identificador del método de pago utilizado (relación con la tabla Metodo_Pago).

fecha_transaccion: Fecha en que se realizó la transacción.

monto: Monto total de la transacción.

estado: Estado de la transacción ("completado", "carrito").

2. ADECUACIÓN DEL ENTORNO DE TRABAJO

2.1 Explicación de la creación del proyecto

Las instalaciones necesarias para llevar a cabo este proyecto han sido mínimas, ya que hemos seleccionado herramientas modernas y eficientes que facilitan el desarrollo. Para ello, hemos configurado un entorno que incluye un IDE de desarrollo para el frontend y backend, además del uso de la terminal de Linux Mint para la gestión de la base de datos MySQL. Asimismo, la configuración del proyecto ha sido optimizada tanto para React como para Spring Boot, permitiéndonos comenzar a programar de manera rápida y eficiente.

Gracias a **Vite**, la configuración del proyecto frontend se ha automatizado, generando una estructura inicial con los archivos y carpetas necesarios. En el backend, Spring Boot nos proporciona un entorno robusto para desarrollar nuestras APIs REST. Por su parte, MySQL, manejado desde la terminal, nos permite gestionar la base de datos relacional de manera directa y eficiente.

Instalación de los IDE y entornos necesarios

1.Frontend

- **Instalación de Node.js y npm:** Descargamos e instalamos Node.js desde su página oficial, lo que incluye npm como gestor de dependencias o podemos instalar Node.js y npm utilizando los siguientes comandos en la terminal:

```
sudo apt update
```

```
sudo apt install -y curl
```

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E  
bash -
```

```
sudo apt install -y nodejs
```

Verificamos la instalación con:

```
node -v
```

```
npm -v
```

- **Configuración de Vite:** Creamos el proyecto ejecutando el siguiente comando en la terminal:

```
npm create vite@latest
```

Seleccionamos React como framework y configuramos el proyecto según nuestras necesidades.

- **Visual Studio Code:** Instalamos este IDE desde su página oficial o utilizando un gestor de paquetes como snap:

```
sudo snap install --classic code
```

Una vez instalado, añadimos extensiones útiles como "ES7+ React/Redux/React-Native snippets" y "Prettier" para mejorar la experiencia de desarrollo.

2.Backend

- **Spring Boot:** Creamos un proyecto Spring Boot utilizando Spring Initializr desde su página oficial o directamente desde el IDE.
- **IDE (IntelliJ IDEA o Eclipse):** Instalamos IntelliJ IDEA (Community Edition) o Eclipse desde sus respectivas páginas oficiales. Configuramos el entorno con un JDK compatible, como OpenJDK 17, con los siguientes comandos:

```
sudo apt update
```

```
sudo apt install -y openjdk-17-jdk
```

Verificamos la instalación con:

```
java -version
```

Base de datos

- **Instalación de MySQL:** Instalamos MySQL utilizando los comandos:

```
sudo apt update
```

```
sudo apt install -y mysql-server
```

```
sudo systemctl start mysql
```

```
sudo systemctl enable mysql
```

- **Configuración de MySQL:** Accedemos al cliente MySQL en la terminal con:

```
sudo mysql
```

Desde allí, configuramos la base de datos y usuarios necesarios para nuestro proyecto.

Este entorno de trabajo bien configurado nos ha permitido iniciar el desarrollo del proyecto de forma ágil y organizada, reduciendo los tiempos de preparación y maximizando la productividad desde el principio.

2.2 Cómo importar el proyecto

Requisitos Previos

1. Node.js (versión 14 o superior).
2. Java Development Kit (JDK) (versión 17).
3. Visual Studio Code con extensiones para Spring Boot.
4. Importar base de datos

Configuración del Frontend

1.Verificar Node.js y npm

comprobaremos las versiones de node.js y de npm para eso usaremos los siguientes comandos:

```
node-v
```

```
npm-v
```

En nuestro caso tenemos estas versiones

```
james@james-VirtualBox:~$ node -v
v18.19.1
james@james-VirtualBox:~$ npm -v
10.2.4
```

Si tienes una versión de Node.js inferior a 14, sigue estos pasos para instalar o actualizar:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.2/install.sh | bash source ~/.bashrc
```

```
nvm install node
```

```
nvm use node
```

Comprobar otra vez la versión con el comando:

```
node -v
```

2.Descargar Visual Studio Code

Para continuar necesitaremos tener instalado Visual Studio Code ya que nosotros lo hemos usado como IDE para nuestro Frontend y nuestro Backend

Instalamos este IDE desde su página oficial o utilizando un gestor de paquetes como snap:

```
sudo snap install --classic code
```

3.Instalar dependencias en Visual Studio Code

Para ser capaces de importar nuestro proyecto necesitaremos algunas dependencias para importarlo desde github

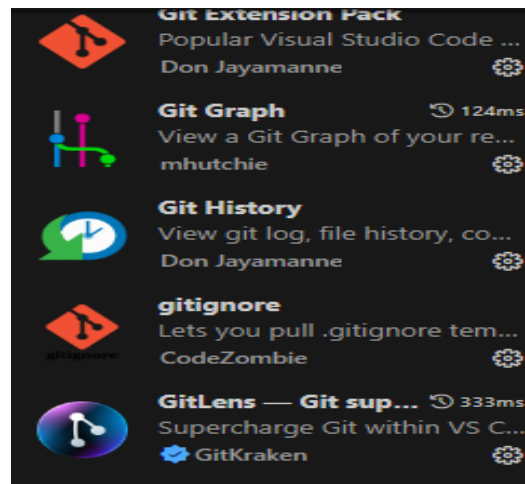
Dependencias de github:

Git-lens,

Git-Graph,

Git-History,

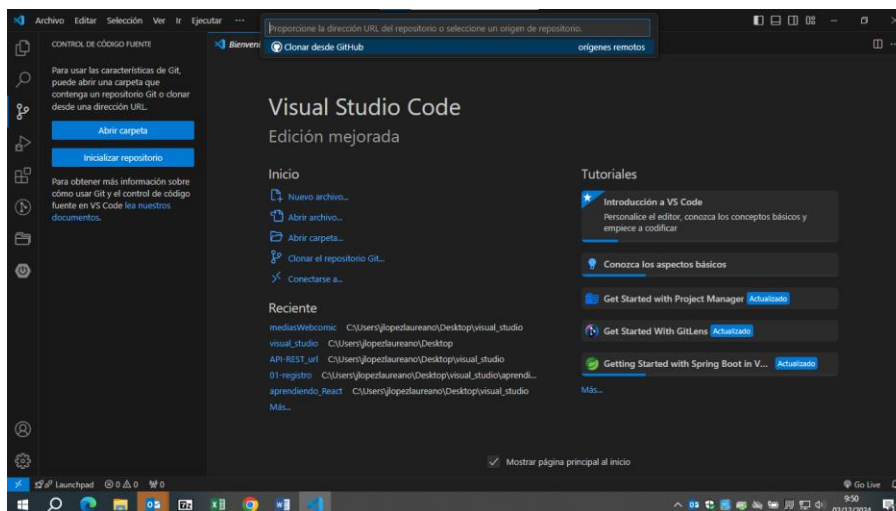
Gitignore,



Open in Github

Git Extension Pack

Una vez descargadas estas dependencias será necesario iniciar sesión en Github después podremos importar la url del repositorio tal como sale en la imagen



y pondremos esta url <https://github.com/james200327/Webcomic.git>

3.2 Instalar jdk 17 y dependencias para Backend

Para poder correr el backend de nuestro proyecto necesitamos instalar jdk 17

Si ya tienes jdk instalado puedes comprobar que versión tienes con el siguiente comando

```
java -version
```

Si no tienes ninguna versión de jdk instalada puedes instalarlo con este comando

```
sudo apt install openjdk-17-jdk -y
```

después de completar la instalación ejecutaremos el siguiente comando que nos dirá la ruta exacta donde se ubica la instalación de jdk

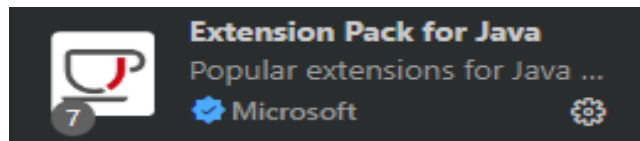
```
readlink -f $(which java)
```

Después de comprobar nuestra instalación de jdk deberemos configurar Visual Studio Code para que detecte código Java para eso deberemos hacer lo siguiente

Setting>JavaHome y dentro del archivo archivo meteremos la ruta que nos dejó el comando anterior

Después Instalaremos las siguientes Dependencias

- Extension Pack for Java

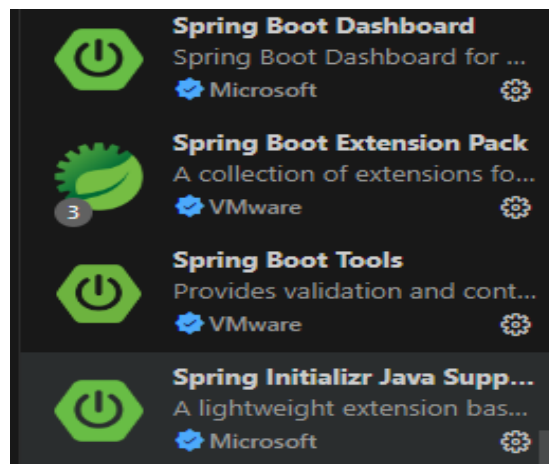


- Spring Boot Extension Pack

- Spring Boot Dashboard

- Spring Boot Tools

- Spring Initializr Java Support



Por último inicializamos el backend

1. Abre Visual Studio Code en la carpeta del backend.
2. Haz clic derecho en el archivo BackEndApplication.java y selecciona Run Java.
3. Después de iniciarlo una vez, puedes gestionar el proyecto desde el Spring Boot Dashboard para iniciar o detener el servidor fácilmente.

Cuando tengamos todo esto deberemos hacer un npm install en la carpeta del proyecto y después un npm run dev

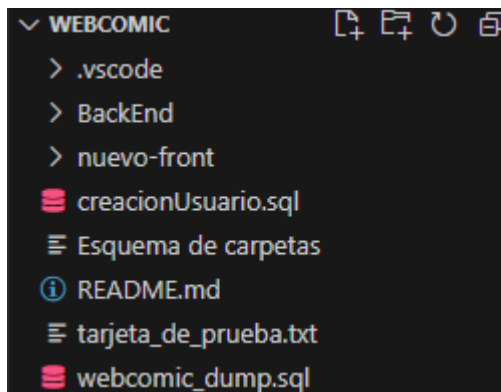
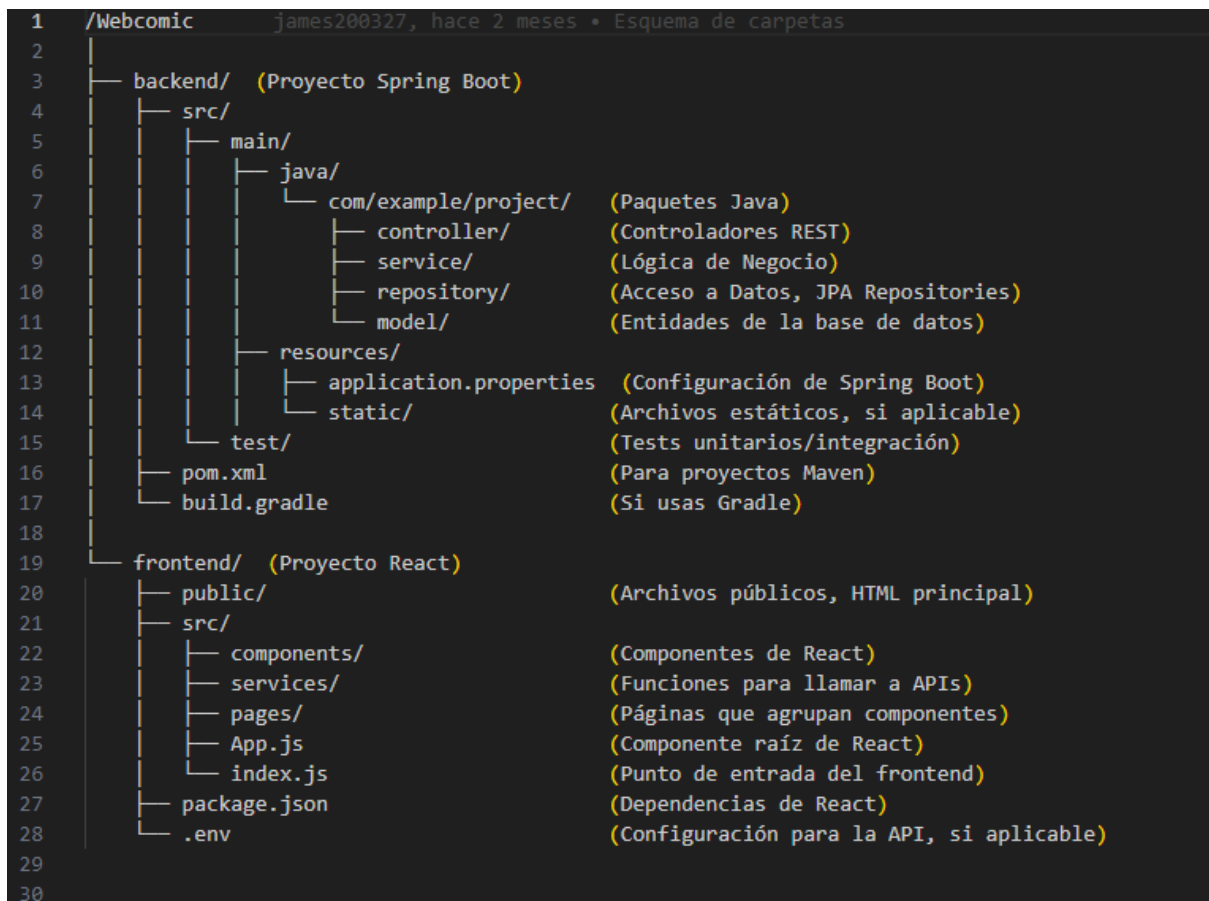
4.Importar Base de datos

1. Primero ejecutar el script "creacionUsuario.sql"
2. Luego ejecutar el script "webcomic_dump.sql"

3.Arquitectura del Proyecto

3.1 Arquitectura y diseño

Este proyecto tiene una estructura simple y ordenada que se divide en varias partes

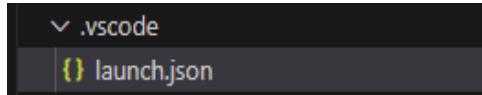


3.1.1 .git

- Indica que el proyecto está bajo control de versiones con Git.

3.1.2 .vscode

- Este archivo configura el entorno de depuración para una aplicación **Spring Boot** llamada `BackEndApplication` que se encuentra en el paquete `com.example.demo`.
- Proporciona a VS Code la información necesaria para compilar, ejecutar y depurar la aplicación Java desde su clase principal.
- Carga variables de entorno desde un archivo `.env`, lo que es útil para separar configuraciones sensibles o específicas del entorno.



3.1.3 Backend

En este archivo se encontrarán las carpetas y el código que da vida al backend de nuestro TFG y las podemos desglosar de la siguiente manera

.gitattributes y .gitignore:

- Archivos relacionados con Git.
- `.gitignore` define los archivos o carpetas que no deben incluirse en el control de versiones.

build.gradle y settings.gradle:

- Archivos de configuración del sistema de construcción Gradle.
- Manejan dependencias, tareas de compilación y configuración del proyecto.

gradle/:

- Contiene configuraciones adicionales y utilidades para Gradle.

gradlew y gradlew.bat:

- Scripts para ejecutar Gradle de manera independiente, sin necesidad de instalarlo globalmente.

src/:

La estructura de la carpeta `src` en el backend está dividida en las siguientes subcarpetas:

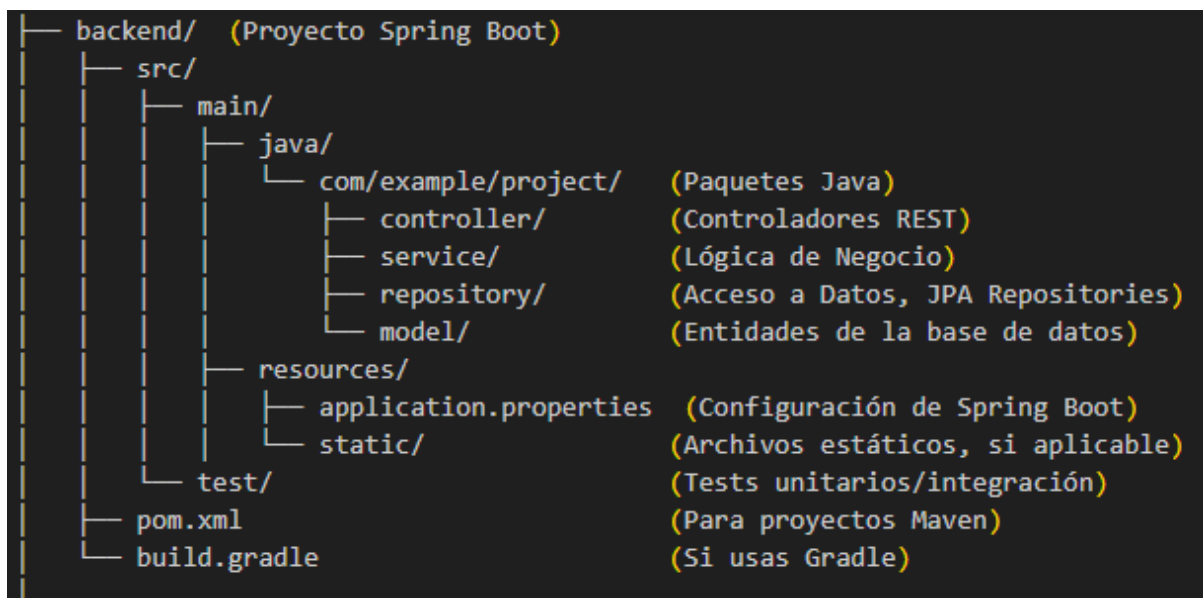
Subcarpetas principales

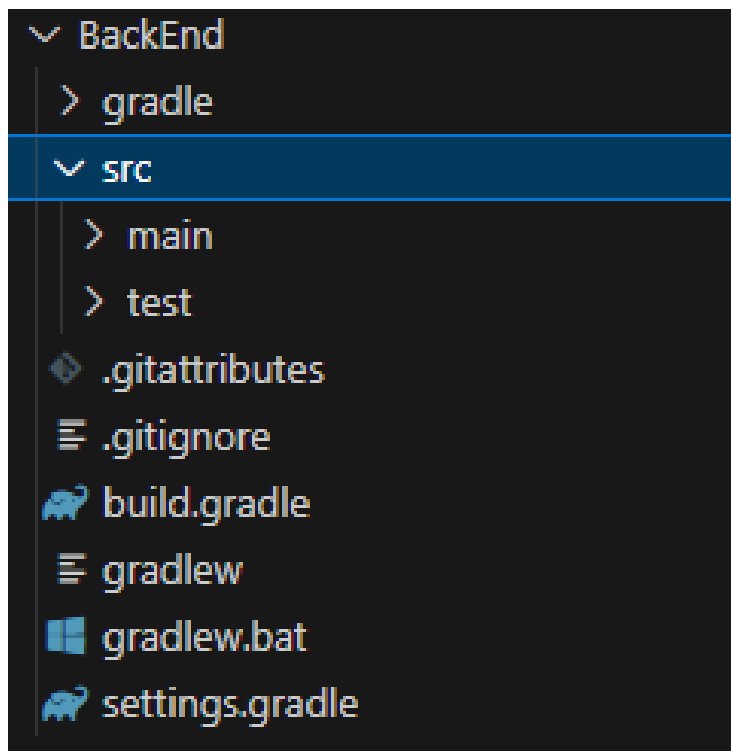
1. **main/**:

- Es la carpeta principal donde reside el código fuente de la aplicación.
- Generalmente contiene las siguientes subcarpetas:
 - **java/**: Código fuente Java.
 - **resources/**: Archivos de configuración, propiedades, y recursos estáticos.
 - **webapp/**: Archivos relacionados con vistas web (si aplica).
- Exploraré más si es necesario.

2. **test/**:

- Contiene las pruebas unitarias y de integración para la aplicación.
- Suele reflejar la estructura de la carpeta **main/**.





3.1.4 creacionUsuario.sql

- Archivo SQL para crear usuarios en la base de datos.

```
james200327, hace 4 semanas | 1 author (james200327)
1 CREATE USER 'james'@'localhost' IDENTIFIED BY '1234';
2 create database webcomic;
3 GRANT ALL PRIVILEGES ON webcomic.* TO 'james'@'localhost';
4 FLUSH PRIVILEGES;
5 use webcomic;
6
```

3.1.5 Esquema de carpetas

- Puede ser un documento o archivo que describe la estructura del proyecto.

3.1.6 Nuevo-Front

.gitignore:

- Define qué archivos o carpetas no se deben incluir en el control de versiones.

index.html:

- Archivo principal de entrada del frontend.
- Probablemente carga los scripts y hojas de estilo del proyecto.

node_modules/:

- Contiene las dependencias instaladas a través de npm o yarn.

package.json y package-lock.json:

- **package.json:**
 - Define las dependencias y scripts del proyecto.
 - Incluye información del proyecto, como el nombre, versión, y comandos npm.
- **package-lock.json:**
 - Registra versiones específicas de las dependencias para garantizar consistencia en instalaciones.

public/:

- Puede contener archivos estáticos como imágenes, fuentes, y el archivo **index.html**.

README.md:

- Archivo Markdown que proporciona información sobre el frontend.

src/:

La carpeta **src** en el frontend incluye los siguientes archivos y carpetas:

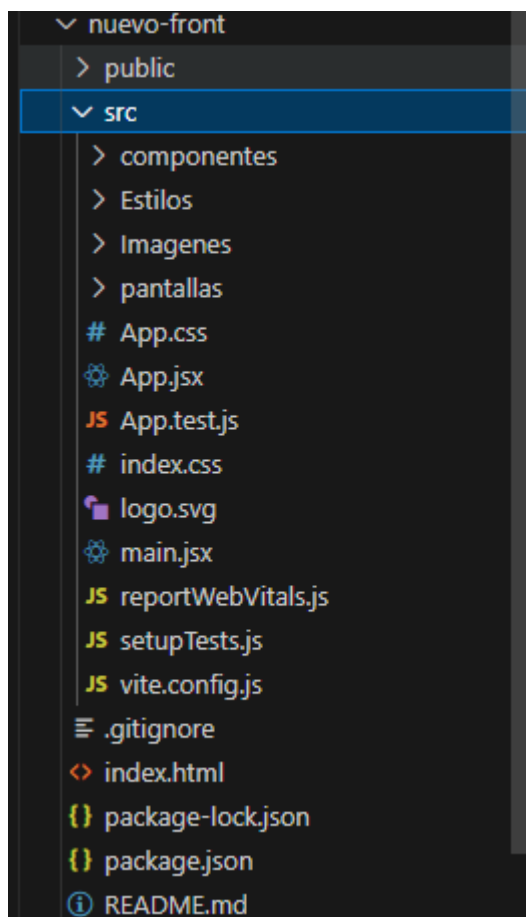
Archivos principales

1. **App.jsx:**
 - Es el componente principal de la aplicación.
 - Maneja la lógica y estructura básica del frontend.
2. **App.css e index.css:**
 - Contienen estilos para la aplicación.

- `App.css` se aplica principalmente al componente `App`, mientras que `index.css` suele contener estilos globales.
3. **`main.jsx`:**
 - Archivo de entrada del proyecto frontend.
 - Renderiza el componente raíz en el DOM.
 4. **`App.test.js` y `setupTests.js`:**
 - Archivos relacionados con pruebas unitarias usando herramientas como Jest o Testing Library.
 5. **`reportWebVitals.js`:**
 - Herramienta opcional para medir métricas de rendimiento en la aplicación.
 6. **`vite.config.js`:**
 - Archivo de configuración para Vite, el empaquetador utilizado en este proyecto.
 7. **`logo.svg`:**
 - Archivo de imagen para un logo (probablemente usado en `App.jsx`).

Carpetas principales

1. **`componentes/`:**
 - Contiene componentes reutilizables del frontend.
 - Ejemplo: botones, formularios, modales, etc.
2. **`pantallas/`:**
 - Agrupa los componentes relacionados con cada página o vista principal del frontend.
 - Ejemplo: vistas como "Home", "Product", "Cart", etc.
3. **`Estilos/`:**
 - Puede contener archivos CSS o SCSS organizados para estilos específicos.
4. **`Imagenes/`:**
 - Contiene recursos de imágenes utilizados en la aplicación.



frontend/ (Proyecto React)	
├─ public/	(Archivos públicos, HTML principal)
├─ src/	
│ ├─ components/	(Componentes de React)
│ ├─ services/	(Funciones para llamar a APIs)
│ ├─ pages/	(Páginas que agrupan componentes)
│ └─ App.js	(Componente raíz de React)
└─ index.js	(Punto de entrada del frontend)
├─ package.json	(Dependencias de React)
└─ .env	(Configuración para la API, si aplicable)

3.1.7 README.md

- Archivo Markdown que explica el propósito del proyecto y proporciona instrucciones.

3.1.8 tarjeta_de_prueba.txt

- Archivo que probablemente contiene datos de prueba, como una tarjeta de crédito ficticia para pruebas.

3.1.9 webcomic_dump.sql

- Archivo SQL con un volcado de la base de datos.

```
DROP TABLE IF EXISTS `comic`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `comic` (
  `precio` double NOT NULL,
  `stock` int NOT NULL,
  `id` bigint NOT NULL AUTO_INCREMENT,
  `autor` varchar(255) DEFAULT NULL,
  `genero` varchar(255) DEFAULT NULL,
  `imagen_url` varchar(255) DEFAULT NULL,
  `titulo` varchar(255) DEFAULT NULL,
  `descripcion` varchar(1000) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=52 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `comic`
--

LOCK TABLES `comic` WRITE;
/*!40000 ALTER TABLE `comic` DISABLE KEYS */;
INSERT INTO `comic` VALUES (1000,10,8,'Hajime Isayama','Manga, Acción, fantasía oscura, drama post-
Titan Vol.1','La humanidad vive rodeada de enormes murallas para protegerse de los titanes, gigantes
Mikasa y su amigo Armin son testigos de la devastadora invasión de los titanes. Lo que los lleva a
```


4. ANÁLISIS DE REQUISITOS

El proyecto tiene como motivo la creación de una página web de compra y venta de comics para eso desarrollamos las siguientes páginas:

- Una página para poder registrar a los usuarios y otra página para que los usuarios
- Una página main o home donde poder ver el catálogo total
- Un navbar que nos permitirá hacer búsqueda de comics, y navegar por las distintas páginas
- Una pagina para profundizar más en los detalles de cada comic y poder agregarlos en el carrito
- Una página para revisar los datos del usuario y los pedidos realizados
- Una página para poder comprar puntos que la forma de pago que queremos agregar para ser mas diferencial
- Un usuario admin que tenga privilegios para crear, modificar y eliminar comic en la página main o home

4.1 Login

La página de Inicio de Sesión permite a los usuarios acceder a sus cuentas en la plataforma. Proporciona un formulario intuitivo donde los usuarios ingresan su nombre y contraseña. Al iniciar sesión, se notifica al usuario sobre el estado de su acceso y se le redirige a la página correspondiente según su rol. Además, incluye una opción para navegar a la página de registro en caso de que aún no tengan una cuenta.



Inicio Sesion

Nombre

Contraseña*


Iniciar Sesion

O bien

Registrarse

4.2 Registro

La página de Registro de Usuario permite a los nuevos usuarios crear una cuenta en la plataforma. Proporciona un formulario donde se recopilan datos como nombre, correo electrónico, contraseña, confirmación de contraseña y dirección (calle, ciudad, y código postal). Además, valida que las contraseñas coinciden antes de registrar al usuario. También ofrece un botón para volver a la página de inicio de sesión en caso de ser necesario.



Crear cuenta

Iniciar sesión con Google

O bien

Nombre

E-mail*

Contraseña*

Repetir contraseña*

Calle

Ciudad

Código Postal

Registrarse

Volver

4.3 Navbar

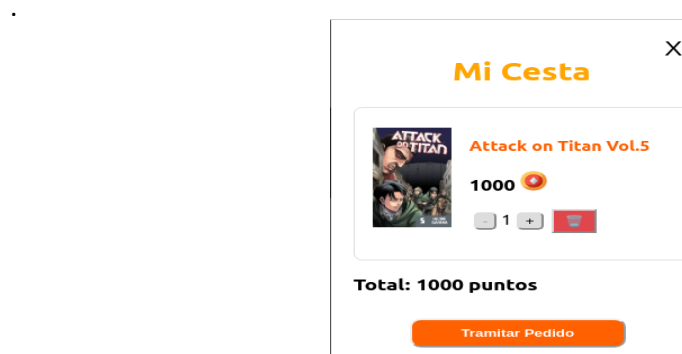
La barra de navegación (Navbar) sirve como elemento principal de navegación para la tienda online de cómics. Ofrece funcionalidades clave para mejorar la experiencia del usuario:

Búsqueda de productos:

Permite buscar cómics o mangas mediante un campo de texto y un filtro por categoría, mejorando la accesibilidad a los productos deseados.

Gestión del carrito:

Muestra los productos añadidos al carrito, permitiendo a los usuarios ajustar cantidades, eliminar productos y ver el total acumulado en puntos. También incluye una opción para proceder al pago



Acceso a la cuenta de usuario:

Contiene enlaces directos para ver y modificar el perfil del usuario.

Visualización de puntos:

Los puntos acumulados por el usuario se muestran en tiempo real, incentivando su uso en la plataforma.

Categorías de productos:

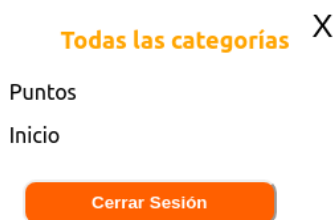
Mediante un menú lateral, los usuarios pueden explorar diferentes categorías de productos o acceder a páginas específicas como campañas de ofertas o novedades.

Opciones adicionales:

Cerrar sesión de manera rápida.

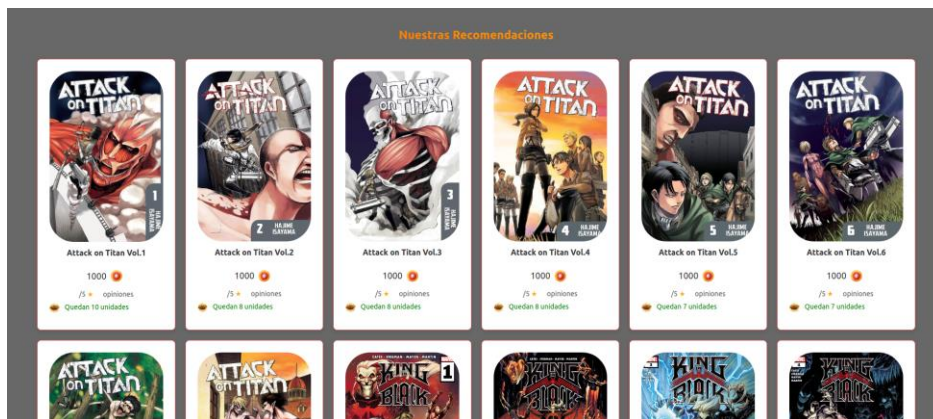
Redirigir a páginas importantes, como el inicio o la sección de puntos.

El diseño incluye dos sidebars (menús laterales), uno para categorías y otro para el carrito, que se despliegan de forma interactiva para mantener un diseño limpio y ordenado.



4.4 MainPage

La página principal muestra un catálogo de cómics disponibles en la plataforma. Los usuarios pueden buscar cómics por título, filtrar por género y visualizar información como el título, imagen, precio, calificación y reseñas. Los administradores tienen acceso adicional para gestionar los cómics, pudiendo agregar nuevos, editar la información existente o eliminar cómics del catálogo.



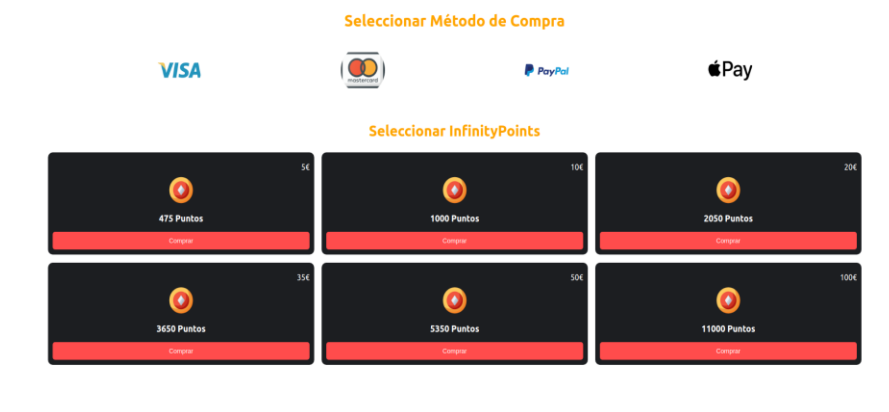
4.5 ProductDetailPage

La página de detalles del producto muestra información específica sobre un cómic seleccionado, accesible a través de su ID en la URL. Está diseñada para proporcionar a los usuarios una vista detallada del producto y permitirles añadirlo a su carrito de compras.



4.6 PointsPage

La página de compra de puntos permite a los usuarios seleccionar un método de compra y gestionar la adquisición de productos o servicios utilizando puntos de la plataforma. Combina elementos de navegación, selección de métodos de pago y visualización de información relacionada con los puntos.



4.7 UserProfile

La página Perfil de usuario o Mi cuenta permite a los usuarios gestionar su información personal y visualizar un historial de sus pedidos confirmados. Combina funciones de edición de perfil y consulta de compras realizadas, proporcionando una experiencia intuitiva para los usuarios.

Perfil de Usuario

Nombre:

javi

Email:

javi@gmail.com

Contraseña:

52u610SPNQQwlltp5St0P8uy8Lw4K5U8A8UEout27gP8n0ZuAn0wE8y8u86

Calle:

calle

Ciudad:

madrid

Código Postal:

28017


Actualizar Datos Personales

Actualizar Datos

Mis pedidos


Pedido ID: 12

Precio Total: 4000 puntos

Venom: King in Black


Cantidad: 1

Subtotal: 1100 puntos

Attack on Titan Vol.5


Cantidad: 1

Subtotal: 1000 puntos

Attack on Titan Vol.3

Cantidad: 1

Subtotal: 1000 puntos

Tokyo Ghoul Vol.8

Cantidad: 1

Subtotal: 900 puntos

Pedido ID: 11

Precio Total: 0 puntos

4.8 TramitarPedido

La página **TramitarPedido** permite a los usuarios confirmar su pedido, proporcionando información de envío y revisando los productos en su carrito antes de completar la compra. Combina funciones de validación de datos, cálculo de totales y confirmación de pedidos.

Confirmar Pedido

Datos de Envío

Nombre

Correo Electrónico

Ciudad

Calle

Código postal

Total: 1000 puntos

Confirmar Pedido

Volver al menú

Tu Pedido

Attack on Titan Vol.5

Cantidad: 1

Precio: 1000 puntos

5. PRUEBAS

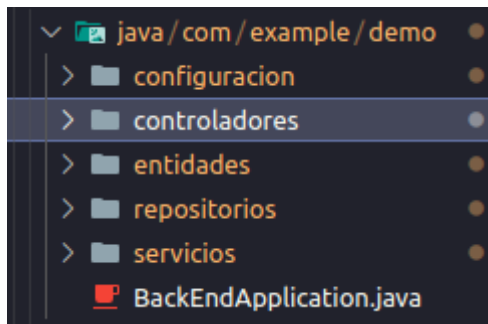
Escenario	Resultado esperado	Resultado obtenido	Solución
Agregar un cómic como administrador	Crear un cómic con todos sus datos y que se muestre en la mainPage.	La imagen del cómic no se muestra de primeras	Actualmente sigue persistiendo el error.
Formulario para modificar comics	Que al completar el formulario, los datos de los comics se modificaran correctamente y se mostraran en la mainPage	El formulario se descolocaba, los botones no funcionaban y no cambiaba nada	Decidimos usar sweetAlert para los formularios
Conexión front back	Inicialmente queríamos que al registrar un usuario con el formulario del front se guarda en la bbdd	El usuario no se guardaba en la bbdd	Configuramos correctamente el cors del backend
Visualización de los productos en el carrito	Al añadir un producto se muestra correctamente en el sidebar	No se muestran los productos en el carrito	Corrección del código en el front, agregar la variable carrito en el navbar en vez de hacerlo global.
Compra de puntos	Poder comprar los puntos con dinero real	No se usa dinero real	Hemos hecho una simulación con un formulario para poner una tarjeta, con los campos validados
No poder tramitar un pedido sin ningun producto en el carrito	Si no hay ningún producto en el carrito, no te deja seguir con la tramitación del pedido	Aun no habiendo nada en el carrito, te dejaba ir al formulario de tramitar pedido.	Aplicamos una validación a la hora de tramitar el el pedido en el carrito
No poder confirmar el pedido, si no se han rellenado los campos del formulario de datos de envío	Si no has completado algunos de los campos, saltan mensajes de error y no te deja continuar	Sin rellenar ningún campo continuaba con la compra	Aplicamos validaciones a cada campo del formulario con los mensajes de error

6. Desarrollo del proyecto

El backend está desarrollado utilizando Spring Boot, este backend se encarga de la lógica del servidor, la gestión de bases de datos, así como del manejo de las APIs REST que nuestro frontend consume. Por otro lado, la parte del frontend está construida con React, una biblioteca de JavaScript enfocada en la creación de interfaces de usuario interactivas y dinámicas.

Carpetas del BackEnd

Primero de todo hemos organizado una serie de carpetas que son las siguientes



Conexión front-back

```

@Configuration
@EnableWebSecurity
public class confWeb {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf.disable()) // Desactivar CSRF para facilitar pruebas en desarrollo
            .authorizeHttpRequests(auth -> auth
                .requestMatchers(...patterns: "/api/usuarios/registro", "/api/usuarios/login", "/api/comics/todos",
                    "/api/comics/{id}", "/api/comics/modificar/{id}", "/api/comics/eliminar/{id}", "/imagenes/**",
                    "/api/pedidos/usuario/{usuarioId}/carrito", "/api/pedidos/usuario/{usuarioId}/agregarProducto/{productoId}", "/api/usuarios/{id}"
                    "/api/pedidos/usuario/{usuarioId}/actualizarProducto/{comicId}", "/api/pedidos/{pedidoId}/confirmar", "/api/pedidos/usuario/{usuar
                )
            .permitAll()
            .anyRequest().authenticated() // Proteger todas las demás rutas
        )
        .cors(corsCustomizer -> corsCustomizer.configurationSource(corsConfigurationSource())); // Configurar CORS
    }
    return http.build();
}

```

Por una parte, aquí tenemos todas las api que se usarán. Todas tienen permitido conectar con el front, cualquier otra api que no este aqui, no le dejara actuar.

```

@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(List.of("http://localhost:5173")); // Permitir solo este origen
    configuration.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
    configuration.setAllowedHeaders(List.of("*"));
    configuration.setAllowCredentials(true);

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration(pattern: "/*", configuration);
    return source;
}

```

Aquí tenemos el puerto 5173, que es donde actúan nuestras apis, cualquier tipo de petición está permitida.

Creación admin-user

```

@Bean
public CommandLineRunner initAdminUser() {
    return args -> {
        // Verificar si el usuario administrador ya existe
        String adminUsername = "admin";
        String adminEmail = "admin@example.com";
        if (!usuarioRepository.existsByUsername(adminUsername)) {
            // Crear el usuario administrador con el rol adecuado
            Usuario admin = new Usuario();
            admin.setUsername(adminUsername);
            admin.setEmail(adminEmail);
            admin.setPassword(passwordEncoder.encode(rawPassword:"admin1234"));
            admin.setRoles(usuarioService.ROLE_ADMIN); // Asegúrate de que tu en

            usuarioRepository.save(admin);
            System.out.println("Usuario administrador creado.");
        } else {
            System.out.println("Usuario administrador ya existe.");
        }
    };
}

```

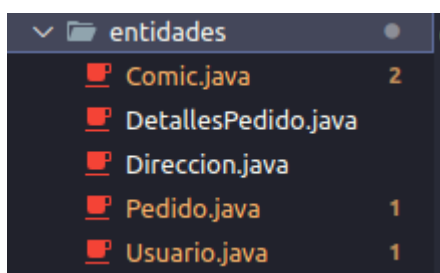
En esta clase se crea el usuario admin. La primera vez que arrancas el proyecto se crea, las siguientes veces te saltará el mensaje de que el admin ya existe.

Entidades

En esta carpeta tendremos todas las entidades que compondrán nuestra

aplicación. Las entidades que tenemos son Comic, Pedido, DetallesPedido, Direccion y Usuario. Cada clase tiene sus respectivos atributos, getters, setters,

puestos como entidad.



Ejemplo de la clase pedido:

```

@Entity

public class Pedido {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "usuario_id")
    private Usuario usuario;

    @OneToMany(mappedBy = "pedido", cascade = CascadeType.ALL, orphanRemoval = true)
    @JsonManagedReference
    private List<DetallesPedido> items = new ArrayList<>();

    private double total;

    private String estado; // "CARRITO" o "CONFIRMADO"

    public Pedido() {}

    public Pedido(Usuario usuario, String estado) {
        this.usuario = usuario;
        this.estado = estado;
        this.items = new ArrayList<>();
        this.total = 0.0;
    }

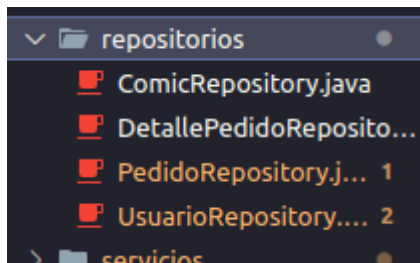
    // Métodos para agregar, actualizar y eliminar productos
    public void agregarProducto(Comic comic, int cantidad) {
        DetallesPedido detalle = items.stream()
            .filter(item -> item.getComic().equals(comic))
            .findFirst()
            .orElse(null);

        if (detalle == null) {
            detalle = new DetallesPedido(this, comic, cantidad, comic.getPrecio());
            items.add(detalle);
        } else {
            detalle.setCantidad(detalle.getCantidad() + cantidad);
        }
    }
}

```

Repositorios

En esta carpeta estarán todos los repositorios de las entidades que conformen una tabla para la base de datos y sus respectivas columnas. Para que conecte con la base de datos hacemos que extienda a JpaRepository.



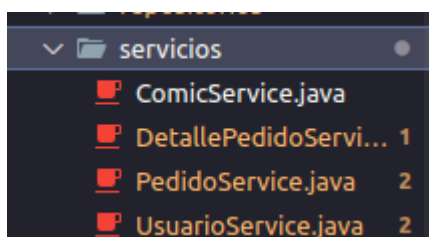
```
You, el mes pasado | 1 author (You)
public interface UsuarioRepository extends JpaRepository<Usuario, Long>{

    public Usuario findByUsername(String username);

    boolean existsByUsername(String username);
    boolean existsByEmail(String email);
}
```

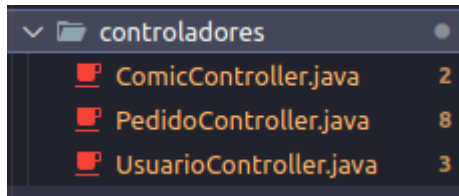
Repositorios

En esta carpeta estarán todos los repositorios de las entidades que conformen una tabla para la base de datos y sus respectivas columnas. Para que conecte con la base de datos hacemos que extienda a JpaRepository.



Controlador

En esta carpeta estarán las Apis. Cada clase tendra de atributo un clase Servicio. De esta forma, a traves de postmappings indicamos las rutas que tendran. Dentro estas solicitudes solo tendríamos que usar los metodos de los



Funcionalidades:

Servicio de usuario:

El servicio **UsuarioService** gestiona las operaciones de negocio relacionadas con los usuarios en la aplicación. Utiliza un repositorio **UsuarioRepository** para interactuar con la base de datos. Los métodos disponibles son:

- **RegistrarUsuario(Usuario usuario):** Guarda un nuevo usuario en la base de datos. Antes de registrarlo, verifica que el nombre de usuario y el correo electrónico no estén en uso para evitar duplicados. Luego, codifica la contraseña usando un PasswordEncoder para mayor seguridad y asigna el rol por defecto User si no se especifica otro. Finalmente, guarda el usuario en la base de datos.
- **CargarUsuario(String nombre):** Busca un usuario en la base de datos por su nombre de usuario. Si el usuario no existe, lanza una excepción con el mensaje "Usuario no encontrado". Si lo encuentra, devuelve el usuario correspondiente.
- **CargarUsuarioPorId(Long id):** Busca un usuario por su ID en la base de datos. Si no se encuentra, lanza una excepción con un mensaje que indica que el usuario no fue hallado para ese ID. Si lo encuentra, devuelve el usuario correspondiente.
- **ActualizarPuntos(Long id, int puntos):** Permite modificar los puntos acumulados de un usuario. Busca el usuario por su ID, suma los puntos proporcionados a los actuales, actualiza el registro del usuario y lo guarda en la base de datos. Si no encuentra el usuario, lanza una excepción indicando que el usuario no existe.

Captura de la clase UsuarioService:

```
You, hace 3 semanas | 1 author (You)
@Service
public class UsuarioService {
    public static final String ROLE_USER = "USER";
    public static final String ROLE_ADMIN = "ADMIN";
    private UsuarioRepository usuarioRepository;
    private PasswordEncoder passwordEncoder;

    public UsuarioService(UsuarioRepository usuarioRepository, PasswordEncoder passwordEncoder) {
        this.usuarioRepository = usuarioRepository;
        this.passwordEncoder = passwordEncoder;
    }

    public Usuario registrarUsuario(Usuario usuario) {
        if (usuarioRepository.existsByUsername(usuario.getUsername())) {
            throw new RuntimeException("El nombre ya esta en uso");
        }

        if (usuarioRepository.existsByEmail(usuario.getEmail())) {
            throw new RuntimeException("El correo ya esta en uso");
        }

        usuario.setPassword(passwordEncoder.encode(usuario.getPassword()));

        if (usuario.getRoles() == null) {
            usuario.setRoles(ROLE_USER); // Rol por defecto
        }
    }
}
```

APIS DE USUARIO

El controlador **UsuarioController** gestiona las solicitudes HTTP relacionadas con los usuarios, como registro, inicio de sesión, compra de puntos y consulta de datos de usuario. Utiliza el servicio **UsuarioService** para realizar las operaciones necesarias.

- **RegistroUsuario:**

- **Ruta:** POST /api/usuarios/registro
- Permite registrar un nuevo usuario en la aplicación. Los datos del usuario (nombre, contraseña, email, etc.) se reciben en el cuerpo de la solicitud. Si el nombre de usuario o el correo ya existen, devuelve un error.

- **LoginUsuario:**

- **Ruta:** POST /api/usuarios/login
- Gestiona el inicio de sesión de los usuarios. Busca al usuario por su nombre de usuario y verifica la contraseña utilizando **PasswordEncoder**. Si las credenciales son correctas, devuelve el rol y el ID del usuario. Si son incorrectas, devuelve un error.

- **ComprarPuntos:**

- **Ruta:** POST /api/usuarios/{id}/comprar-puntos
- Permite a un usuario comprar puntos enviando los datos de su tarjeta de crédito (número de tarjeta, fecha de expiración, CVV) y la cantidad de puntos deseados. Valida los datos de la tarjeta y, si son correctos, actualiza el saldo de puntos del usuario en la base de datos.

- **ObtenerUsuario:**

- **Ruta:** GET /api/usuarios/{id}
- Recupera los datos de un usuario específico buscando por su ID. Si el usuario no existe, lanza un error indicando que no se encontró.

Captura de la clase UsuarioController:

```
@RestController
@RequestMapping("/api/usuarios")

public class UsuarioController {

    private UsuarioService usuarioService;
    private PasswordEncoder passwordEncoder;

    public UsuarioController(UsuarioService usuarioService, PasswordEncoder passwordEncoder) {

        this.usuarioService = usuarioService;
        this.passwordEncoder = passwordEncoder;
    }

    @PostMapping("/registro")
    public ResponseEntity<String> registroUsuario(@RequestBody Usuario usuario) {
        usuarioService.registrarUsuario(usuario);

        return new ResponseEntity<>(body:"Usuario registrado exitosamente", HttpStatus.CREATED);
    }

    @PostMapping("/login")
    public ResponseEntity<Map<String, String>> loginUsuario(@RequestBody Usuario usuario) {

        try {
            Usuario usuarioCargado = usuarioService.cargarUsuario(usuario.getUsername());
            System.out.print(usuarioCargado.getUsername() + ", " + usuarioCargado.getPassword());

            // Verificar la contraseña
            if (passwordEncoder.matches(usuario.getPassword(), usuarioCargado.getPassword())) {
                System.out.println("Contraseña correcta");

                // Crear la respuesta con el rol y el userId
                Map<String, String> response = new HashMap<>();
                response.put("role", usuarioCargado.getRoles());
                response.put("userId", usuarioCargado.getId().toString());

                return ResponseEntity.ok(response);
            }
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Usuario no encontrado");
        }
    }
}
```

Servicio de Comic:

El servicio **ComicService** gestiona las operaciones de negocio relacionadas con los cómics en la aplicación. Utiliza un repositorio **ComicRepository** para interactuar con la base de datos. Los métodos disponibles son:

- **AregarComic(Comic comic)**: Guarda un nuevo cómic en la base de datos.
- **ModificarComic(Long id, Comic comic)**: Modifica un cómic existente. Busca el cómic por su id y, si no se encuentra, lanza una excepción. Luego, actualiza sus atributos con los nuevos valores proporcionados.
- **EliminarComic(Long id)**: Elimina un cómic de la base de datos. Si el cómic no existe, lanza una excepción.
- **ObtenerTodosLosComics()**: Devuelve una lista con todos los cómics almacenados en la base de datos.
- **ObtenerComicPorId(Long id)**: Busca un cómic por su id. Si no se encuentra, lanza una excepción.

Captura de la clase ComicService:

```

You, hace 3 semanas | 1 author (You)
@Service
public class ComicService {
    private ComicRepository comicRepository;

    public ComicService(ComicRepository comicRepository) {
        this.comicRepository = comicRepository;
    }

    public Comic agregarComic(Comic comic) {
        return comicRepository.save(comic);
    }

    public Comic modificarComic(Long id, Comic comic) {
        Comic comicExistente = comicRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException(message:"Comic no encontrado"));

        if (comic.getTitulo() != null && !comic.getTitulo().isEmpty()) {
            comicExistente.setTitulo(comic.getTitulo());
        }
        if (comic.getAutor() != null && !comic.getAutor().isEmpty()) {
            comicExistente.setAutor(comic.getAutor());
        }
        if (comic.getGenero() != null && !comic.getGenero().isEmpty()) {
            comicExistente.setGenero(comic.getGenero());
        }
    }
}

```

Apis de Comic:

El controlador **ComicController** maneja las solicitudes HTTP relacionadas con la gestión de cómics en la aplicación. Utiliza el servicio **ComicService** para realizar las operaciones de negocio necesarias. Las APIs disponibles son:

- **agregarComic(titulo, autor, genero, precio, stock, imagen, descripcion):**
 - **Ruta:** POST /api/comics/agregar
 - Guarda un nuevo cómic en la base de datos junto con una imagen. La imagen se almacena en la carpeta static/imagenes y su ruta se registra en la base de datos.
- **modificarComic(Long id, Comic comic):**
 - **Ruta:** PUT /api/comics/modificar/{id}
 - Modifica un cómic existente. Busca el cómic por su id, actualiza los atributos proporcionados en el cuerpo de la solicitud y devuelve el cómic actualizado.
- **eliminarComic(Long id):**
 - **Ruta:** DELETE /api/comics/eliminar/{id}
 - Elimina un cómic de la base de datos identificado por su id. Si el cómic no existe, lanza una excepción.
- **obtenerListaComics():**

- **Ruta:** GET /api/comics/todos
- Devuelve una lista con todos los cómics almacenados en la base de datos.
- **obtenerComicId(Long id):**
 - **Ruta:** GET /api/comics/{id}
 - Busca un cómic por su id y devuelve los detalles. Si no se encuentra, lanza una excepción.

Captura de la clase ComicController:

```

6 @RestController
7 @RequestMapping("/api/comics")
8 public class ComicController {
9
10     private ComicService comicService;
11
12     public ComicController(ComicService comicService) {
13         this.comicService = comicService;
14     }
15
16     @PostMapping("/agregar")
17     public ResponseEntity<String> agregarComic(
18         @RequestParam("titulo") String titulo,
19         @RequestParam("autor") String autor,
20         @RequestParam("genero") String genero,
21         @RequestParam("precio") Double precio,
22         @RequestParam("stock") Integer stock,
23         @RequestParam("imagen") MultipartFile imagen,
24         @RequestParam("descripcion") String descripcion){
25
26         try {
27             String nombreArchivo = System.currentTimeMillis() + "_" + imagen.getOriginalFilename();
28             Path rutaArchivo = Paths.get("src/main/resources/static/imagenes/" + nombreArchivo);
29             Files.copy(imagen.getInputStream(), rutaArchivo);
30
31             // Crear y guardar el objeto Comic
32             Comic nuevoComic = new Comic();
33             nuevoComic.setTitulo(titulo);
34             nuevoComic.setAutor(autor);
35             nuevoComic.setGenero(genero);
36             nuevoComic.setPrecio(precio);
37             nuevoComic.setStock(stock);
38             // Guardar la URL relativa de la imagen en la base de datos
39
40             nuevoComic.setImagenUrl("/imagenes/" + nombreArchivo);
41             nuevoComic.setDescripcion(descripcion);
42             comicService.agregarComic(nuevoComic);
43
44             return new ResponseEntity<>("Cómico agregado con imagen", HttpStatus.CREATED);
45         }
46     }
47 }

```

Servicio de Pedido

El servicio **PedidoService** gestiona las operaciones de negocio relacionadas con los pedidos en la aplicación. Utiliza los repositorios **PedidoRepository**, **ComicRepository** y **UsuarioRepository** para interactuar con la base de datos. Los métodos disponibles son:

- **ObtenerCarrito(Long usuarioid):** Recupera el carrito del usuario identificado por su ID. Si no existe, crea un nuevo carrito con estado "CARRITO" y lo guarda en la base de datos.

- **AgregarProducto(Long usuarioid, Long comicid, int cantidad):** Agrega un producto al carrito del usuario. Si el cómic no tiene suficiente stock para la cantidad solicitada, lanza una excepción.
- **ActualizarCantidad(Long usuarioid, Long comicid, int cantidad):** Actualiza la cantidad de un cómic en el carrito del usuario. Guarda los cambios en la base de datos.
- **ConfirmarPedido(Long pedidoid):** Confirma un pedido. Verifica si el usuario tiene suficientes puntos para completar la compra. Luego, resta los puntos y actualiza el stock de los cómics. Cambia el estado del pedido a "CONFIRMADO" y guarda los cambios.
- **ObtenerPedidosConfirmados(Long usuarioid):** Devuelve una lista de todos los pedidos confirmados de un usuario, ordenados de más reciente a más antiguo.

Captura de la clase Pedido Service:

```

@Service
public class PedidoService {

    @Autowired
    private PedidoRepository pedidoRepository;

    @Autowired
    private ComicRepository comicRepository;

    @Autowired
    private UsuarioRepository usuarioRepository;

    public Pedido obtenerCarrito(Long usuarioid) {
        return pedidoRepository.findByUsuarioIdAndEstado(usuarioid, estado:"CARRITO")
            .orElseGet(() -> {
                Usuario usuario = usuarioRepository.findById(usuarioid)
                    .orElseThrow(() -> new EntityNotFoundException(message:"Usuario no encontrado"));
                Pedido nuevoCarrito = new Pedido(usuario, estado:"CARRITO");
                return pedidoRepository.save(nuevoCarrito);
            });
    }

    public Pedido agregarProducto(Long usuarioid, Long comicid, int cantidad) {
        Pedido carrito = obtenerCarrito(usuarioid);
        Comic comic = comicRepository.findById(comicid)
            .orElseThrow(() -> new EntityNotFoundException(message:"Comic no encontrado"));

        if (comic.getStock() < cantidad) {
            throw new IllegalArgumentException("No hay suficiente stock disponible");
        }

        carrito.agregarProducto(comic, cantidad);
        return pedidoRepository.save(carrito);
    }

    public Pedido actualizarCantidad(Long usuarioid, Long comicid, int cantidad) {
        Pedido carrito = obtenerCarrito(usuarioid);
        Comic comic = comicRepository.findById(comicid)
            .orElseThrow(() -> new EntityNotFoundException(message:"Comic no encontrado"));
    }
}

```

Apis de pedidos:

El controlador **PedidoController** gestiona las solicitudes HTTP relacionadas con los pedidos en la aplicación, como la creación de carritos, la adición de productos y la confirmación de pedidos. Utiliza el servicio **PedidoService** para realizar las operaciones necesarias.

- **ObtenerCarrito:**

- Ruta: GET /api/pedidos/usuario/{usuarioId}/carrito
- Recupera el carrito del usuario identificado por usuarioId. Si no existe, crea un nuevo carrito para el usuario.

- **AgregarProducto:**

- Ruta: POST
/api/pedidos/usuario/{usuarioId}/agregarProducto/{comicId}
- Permite agregar un cómic al carrito del usuario identificado por usuarioId. La cantidad de productos se pasa como parámetro. Si el cómic no existe o no hay suficiente stock, devuelve un error.

- **ActualizarCantidad:**

- Ruta: PUT
/api/pedidos/usuario/{usuarioId}/actualizarProducto/{comicId}
- Actualiza la cantidad de un cómic en el carrito del usuario identificado por usuarioId. Si el cómic no está en el carrito, lanza una excepción.

- **ConfirmarPedido:**

- Ruta: POST /api/pedidos/{pedidoId}/confirmar
- Confirma el pedido identificado por pedidoId. Verifica que el usuario tenga suficientes puntos para completar el pedido. Si los puntos son insuficientes, lanza un error. Además, actualiza el stock de los cómics.

- **ObtenerPedidosConfirmados:**

- Ruta: GET /api/pedidos/usuario/{usuarioId}/confirmados
- Recupera todos los pedidos confirmados de un usuario identificado por usuarioId.

Captura de la clase PedidoController:

```
You, hace 2 semanas | 1 author (you)
@RestController
@RequestMapping("/api/pedidos")
public class PedidoController {

    @Autowired
    private PedidoService pedidoService;

    @GetMapping("/usuario/{usuarioId}/carrito")
    public ResponseEntity<Pedido> obtenerCarrito(@PathVariable Long usuarioId) {
        Pedido carrito = pedidoService.obtenerCarrito(usuarioId);
        return ResponseEntity.ok(carrito);
    }

    @PostMapping("/usuario/{usuarioId}/agregarProducto/{comicId}")
    public ResponseEntity<Pedido> agregarProducto(
        @PathVariable Long usuarioId,
        @PathVariable Long comicId,
        @RequestParam int cantidad) {
        Pedido carrito = pedidoService.agregarProducto(usuarioId, comicId, cantidad);
        return ResponseEntity.ok(carrito);
    }

    @PutMapping("/usuario/{usuarioId}/actualizarProducto/{comicId}")
    public ResponseEntity<Pedido> actualizarCantidad(
        @PathVariable Long usuarioId,
        @PathVariable Long comicId,
        @RequestParam int cantidad) {
        Pedido carrito = pedidoService.actualizarCantidad(usuarioId, comicId, cantidad);
        return ResponseEntity.ok(carrito);
    }
}
```

6.2 Frontend

6.2.1 Navbar

El componente Navbar es una barra de navegación clave para la interfaz de usuario del proyecto, proporcionando funcionalidad de búsqueda, navegación, gestión de carrito de compras y redirecciones a diferentes secciones de la aplicación. A continuación se resumen las funciones principales:

Funciones principales

1. Gestión del menú y carrito:

- **toggleMenu**: Alterna la visibilidad del menú lateral izquierdo (categorías).
- **toggleCart**: Alterna la visibilidad del carrito de compras en el sidebar derecho.

2. Redirecciones:

- **irPagPuntos:** Redirige a la página de puntos.
- **irInicio:** Redirige a la página de inicio.
- **irUserProfile:** Redirige al perfil del usuario.
- **logOut:** Elimina las credenciales del usuario del almacenamiento local y redirige a la página principal.

3. Gestión de puntos:

- **obtenerPuntos:** Consulta el backend para obtener los puntos acumulados del usuario y los actualiza en el estado.

4. Gestión del carrito de compras:

- **fetchCarrito:** Obtiene los productos en el carrito del usuario desde el backend y actualiza el estado local.
- **calculateTotal:** Calcula el total de puntos necesarios para los productos del carrito.
- **updateCantidad:** Actualiza la cantidad de un producto en el carrito y sincroniza los cambios con el backend.
- **removeFromCart:** Elimina un producto del carrito actualizando su cantidad a 0.
- **handleCheckout:** Valida el carrito y redirige a la página de tramitación de pedidos si hay productos en el carrito.

6.2.1 PurchaseOption

El componente PurchaseOption permite al usuario adquirir puntos mediante un pago simulado con una tarjeta de crédito. Está diseñado para integrarse con el backend para registrar la compra y actualizar los puntos del usuario.

Funciones principales

1. Interfaz de compra:

- **handlePurchase:** Muestra un formulario de pago usando SweetAlert2 para capturar los datos de la tarjeta de crédito.
- Valida la información introducida por el usuario mediante el método preConfirm de SweetAlert2.

2. Interacción con el backend:

- Envía una solicitud POST al endpoint del backend (/api/usuarios/{userId}/comprar-puntos) para procesar la compra.
- Utiliza axios para realizar la solicitud y pasa los datos capturados del formulario como parámetros en la URL.

3. Notificaciones al usuario:

- Usa SweetAlert2 para:
- Confirmar la compra exitosa con un mensaje de éxito.
- Mostrar errores en caso de fallos, indicando el motivo o un mensaje genérico.

6.2.3 InicioSesion

El componente InicioSesion permite a los usuarios iniciar sesión en el sistema. Está diseñado para enviar credenciales al backend y gestionar tanto los roles de los usuarios como el almacenamiento de datos importantes en el navegador.

Funciones principales

1. Inicio de sesión:

- manejarInicioSesion:
 - Envía las credenciales ingresadas (nombre y password) al backend usando axios.
 - Maneja las respuestas del servidor para determinar si el usuario es un administrador o un usuario estándar.
 - Almacena el rol y el ID del usuario en localStorage para su uso posterior.

2. Gestión de formularios:

- manejarCambio:
 - Actualiza los datos del formulario a medida que el usuario escribe en los campos.

3. Navegación:

- Utiliza useNavigate para redirigir al usuario:
 - A la página principal después de un inicio de sesión exitoso.
 - A la página de registro al hacer clic en el botón "Registrarse".

4. Notificaciones:

- Utiliza SweetAlert2 para mostrar mensajes personalizados de inicio de sesión según el rol.

6.2.4 MainPage

El componente MainPage actúa como la página principal de la aplicación. Muestra una lista de cómics disponibles, permite a los administradores añadir, editar y eliminar cómics, y ofrece funciones de búsqueda y filtrado para los usuarios. Este componente interactúa con el backend para gestionar los datos de los cómics y utiliza el componente ProductCard para renderizar cada cómic en formato de tarjeta.

Estado

1. **selectedGenre:** Género seleccionado para el filtrado (por defecto "all").
2. **comic:** Lista de cómics cargados desde el backend.
3. **mostrarFormComic:** Booleano que controla la visibilidad del formulario de adición de cómics.
4. **nuevoComic:** Objeto que almacena los datos del nuevo cómic a añadir.
5. **rol:** Rol del usuario actual (ADMIN o USER), obtenido desde localStorage.

6. searchTerm: Término de búsqueda para filtrar los cómics por título.

Efectos

- **useEffect:** Llama al método `listaComics` al cargar el componente para obtener la lista de cómics desde el backend.

Funciones Principales

1. **manejoAddComic:**

- Agrega un nuevo cómic al backend.
- Utiliza un objeto `FormData` para enviar datos y archivos al servidor.
- Actualiza la lista de cómics tras una adición exitosa.

2. **manejoBorrarComic:**

- Elimina un cómic usando su ID.
- Actualiza el estado eliminando el cómic localmente tras una eliminación exitosa.

3. **manejoEditarComic:**

- Muestra un formulario emergente (`SweetAlert`) para editar un cómic.
- Envía los datos modificados al backend y actualiza la lista de cómics localmente.

4. **handleImageChange:**

- Captura y almacena la imagen seleccionada para un nuevo cómic en el estado `nuevoComic`.

5. **filteredComics:**

- Filtra los cómics basándose en el género seleccionado y el término de búsqueda.

6.2.5 ProductDetailPage

Descripción: Componente que muestra los detalles de un producto seleccionado, específicamente un cómic. El usuario puede ver la imagen, descripción, precio y una opción para agregar el producto al carrito.

Funcionalidad principal:

- **Obtención del producto:**
El componente utiliza useParams para obtener el productId desde la URL, que luego se usa para hacer una solicitud a la API y obtener los detalles del producto específico. Los datos del producto (como título, descripción, precio, etc.) se almacenan en el estado local usando useState.
- **Redirección:**
El useNavigate permite redirigir al usuario a la página principal cuando hace clic en "Home" en la barra de navegación.
- **Añadir al carrito:**
Cuando el usuario hace clic en el botón "Añadir al carrito", se envía una solicitud POST para agregar el producto al carrito del usuario en el backend. Si la solicitud es exitosa, se muestra una alerta de confirmación usando SweetAlert2 y la página se recarga automáticamente para reflejar el cambio.
- **Carga de datos:**
Utiliza useEffect para realizar una solicitud GET al backend cuando el componente se monta. Si la solicitud es exitosa, los detalles del producto se cargan en el estado product. Si no se ha cargado el producto, se muestra un mensaje de "Cargando producto...".

6.2.6 RegistroUsuario

Descripción: Componente que permite a un usuario crear una cuenta en la plataforma. El formulario incluye campos para ingresar datos personales como nombre, email, contraseña, dirección, etc. Al enviar el formulario, se realiza una solicitud al backend para registrar al usuario.

Funcionalidad principal:

- **Estado de los datos del formulario:**
Se utiliza useState para manejar el estado de los datos del formulario (nombre, email, password, confirmPassword, calle, codigoPostal, ciudad) y para manejar el mensaje de error si ocurre algún problema durante el registro.
- **Manejo de cambios en el formulario:**
El evento onChange en cada campo de entrada llama a la función manejarCambio, que actualiza el estado con los valores introducidos por el usuario.
- **Redirección al inicio de sesión:**
La función InicioSesion redirige al usuario a la página de inicio (/) si decide no registrarse o quiere volver al inicio de sesión.
- **Manejo del registro:**
La función manejarRegistro se ejecuta cuando el formulario es enviado. Esta función valida que las contraseñas coincidan y, si todo está correcto, envía los datos del formulario al backend usando axios. Si la solicitud es exitosa, se muestra una alerta y se redirige al usuario a la página de inicio. Si hay errores (como nombre o correo ya en uso), se muestra un mensaje de error.

6.2.7 TramitarPedido

Descripción: Componente encargado de gestionar el proceso de confirmación de un pedido. Permite a los usuarios verificar los productos en su carrito, ingresar sus datos de envío y confirmar el pedido. Además, calcula el total del pedido y valida los campos del formulario antes de enviarlo.

Funcionalidad principal:

- **Obtención del carrito y ID de pedido:**
Se hace una solicitud al backend para obtener el carrito del usuario y el ID del pedido asociado, utilizando el ID de usuario almacenado en el localStorage. La función obtenerIdPedido obtiene el ID del pedido y fetchCarrito obtiene los productos del carrito.
- **Cálculo del total:**
La función calculateTotal calcula el total de los productos en el carrito multiplicando el precio de cada producto por la cantidad y sumando los resultados.
- **Manejo del formulario:**
El formulario contiene campos para ingresar el nombre, email, ciudad, calle y código postal del usuario. La función handleFormChange actualiza el estado del formulario cada vez que el usuario cambia un campo.
- **Validación del formulario:**
La función validarForm verifica que todos los campos del formulario estén completos y que el email tenga un formato válido. Si algún campo es inválido, muestra un mensaje de error correspondiente.
- **Confirmación del pedido:**
La función confirmarPedido valida el formulario y, si no hay errores, envía una solicitud al backend para confirmar el pedido. Si la solicitud es exitosa, se muestra una alerta con SweetAlert2 indicando que el pedido fue confirmado. Luego, redirige al usuario al menú principal.

6.2.8 UserProfile

Este es el componente UserProfile, que muestra y permite editar la información de usuario, además de listar los pedidos confirmados del usuario. Aquí tienes una explicación detallada del código:

Funciones:

- **useEffect:** Al montar el componente, se realizan dos peticiones:
 - **fetchUserData:** Obtiene los datos del usuario desde el backend, basándose en el `userId` almacenado en `localStorage`. Luego, actualiza el estado `userData` con la respuesta.
 - **fetchListaComics:** Obtiene los pedidos confirmados por el usuario y actualiza el estado `comics`.
- **handleInputChange:** Esta función se ejecuta cuando el usuario cambia el valor de algún input en el formulario de edición. Actualiza el estado `userData` con los nuevos valores.
- **handleSubmit:** Esta función se ejecuta cuando el formulario de edición se envía. Envía los datos actualizados al backend mediante una solicitud PUT. Si la actualización es exitosa, cambia el estado `isEditing` a `false` y muestra una alerta indicando que los datos fueron actualizados.

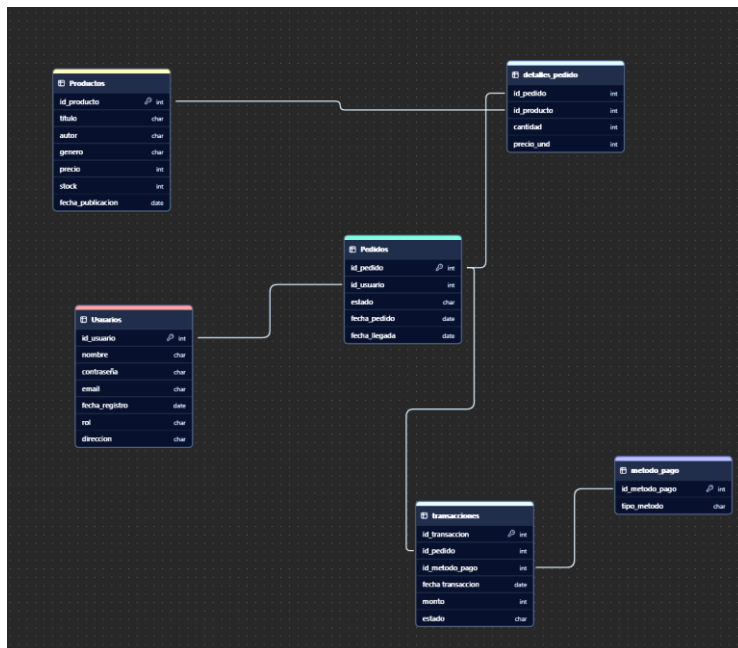
7. Desarrollo de Base de datos

En este apartado se documentará todo lo relacionado con la base de datos de Infinity Comics

Aquí podemos ver el esquema original que se tenía pensado para el proyecto pero por ciertas limitaciones en la realización de transacciones reales, decidimos omitir la tabla de transacciones y metodo_pago, así que nuestro proyecto solo cuenta con 4 tablas.

```
+-----+
| Tables_in_webcomic |
+-----+
| comic                |
| detalles_pedido      |
| pedido               |
| usuario              |
+-----+
4 rows in set (0,00 sec)
```

7.1 Esquema Entidad-Relación



Usuarios → Pedidos:

- Relación 1:N
Un usuario puede realizar uno o varios pedidos, pero un pedido solo puede pertenecer a un usuario.
Claves: **Usuarios.id_usuario** → **Pedidos.id_usuario**.

Pedidos → Detalles Pedido:

- Relación 1:N
Un pedido puede contener uno o varios productos (detallados en **detalles_pedido**), pero cada entrada en **detalles_pedido** pertenece a un único pedido.
Claves: **Pedidos.id_pedido** → **detalles_pedido.id_pedido**.

Productos → Detalles Pedido:

- Relación 1:N
Un producto puede estar asociado a uno o varios detalles de pedido (es decir, puede aparecer en múltiples pedidos), pero cada entrada en **detalles_pedido** está asociada a un solo producto.
Claves: **Productos.id_producto** → **detalles_pedido.id_producto**.

7.2 Scripts

Creación Tabla Usuarios:

```
CREATE TABLE IF NOT EXISTS Usuarios (  
  id_usuario int NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  nombre char,  
  contraseña char,  
  email char,  
  fecha_registro date DEFAULT NULL,  
  rol char,  
  direccion char  
);  
  
CREATE UNIQUE INDEX index_1 ON Usuarios (id_usuario);
```

Creación Tabla Productos/Comic:

```
CREATE TABLE IF NOT EXISTS Productos (  
  id_producto int NOT NULL PRIMARY KEY,  
  titulo char,  
  autor char,  
  genero char,  
  precio int,  
  stock int,  
  fecha_publicacion date DEFAULT NULL  
);  
  
CREATE INDEX index_1 ON Productos (id_producto);
```

Creación Tabla Pedidos:

```
CREATE TABLE IF NOT EXISTS Pedidos (  
  id_pedido int NOT NULL PRIMARY KEY,  
  id_usuario int,  
  estado char,  
  fecha_pedido date DEFAULT NULL,  
  fecha_llegada date DEFAULT NULL  
);
```

Creación Tabla detalles_pedido:

```
CREATE TABLE IF NOT EXISTS detalles_pedido (  
  id_pedido int,  
  id_producto int,  
  cantidad int,  
  precio_und int,  
  PRIMARY KEY (id_pedido, id_producto)  
);
```

7.3 Tablas

7.3.1 Comic

Productos:

- **id_producto (int):** Identificador único del producto.
- **título (char):** Nombre del producto o cómic.
- **autor (char):** Autor del producto.
- **género (char):** Género o categoría del producto.
- **precio (int):** Precio del producto.
- **stock (int):** Cantidad disponible en inventario.
- **fecha_publicación (date):** Fecha en que se publicó el producto.

```
mysql> describe comic;
```

Field	Type	Null	Key	Default	Extra
precio	double	NO		NULL	
stock	int	NO		NULL	
id	bigint	NO	PRI	NULL	auto_increment
autor	varchar(255)	YES		NULL	
genero	varchar(255)	YES		NULL	
imagen_url	varchar(255)	YES		NULL	
titulo	varchar(255)	YES		NULL	
descripcion	varchar(1000)	YES		NULL	

```
8 rows in set (0,00 sec)
```

Usuarios:

- **id_usuario (int):** Identificador único del usuario.
- **nombre (char):** Nombre del usuario.
- **contraseña (char):** Contraseña del usuario.
- **email (char):** Dirección de correo electrónico del usuario.
- **fecha_registro (date):** Fecha en que el usuario se registró.
- **rol (char):** Rol del usuario (por ejemplo, administrador o cliente).
- **dirección (char):** Dirección del usuario.

```
mysql> describe usuario;
```

Field	Type	Null	Key	Default	Extra
fecha_registro	date	YES		NULL	
id	bigint	NO	PRI	NULL	auto_increment
calle	varchar(255)	YES		NULL	
ciudad	varchar(255)	YES		NULL	
codigo_postal	varchar(255)	YES		NULL	
email	varchar(255)	YES		NULL	
password	varchar(255)	YES		NULL	
roles	varchar(255)	YES		NULL	
username	varchar(255)	YES		NULL	
puntos	double	NO		NULL	

```
10 rows in set (0,01 sec)
```

Pedidos:

- **id_pedido (int):** Identificador único del pedido.
- **id_usuario (int):** Relación con el usuario que realizó el pedido.
- **estado (char):** Estado del pedido (ej. "pendiente", "completado").
- **fecha_pedido (date):** Fecha en que se realizó el pedido.
- **fecha_llegada (date):** Fecha estimada o real de entrega.

```
mysql> describe pedido;
```

Field	Type	Null	Key	Default	Extra
id	bigint	NO	PRI	NULL	auto_increment
estado	varchar(255)	YES		NULL	
fecha_creacion	datetime(6)	YES		NULL	
usuario_id	bigint	YES	MUL	NULL	
total	double	NO		NULL	

```
5 rows in set (0,00 sec)
```


Detalles_pedido

- **id_pedido (int)**: Relación con el pedido correspondiente.
- **id_producto (int)**: Relación con el producto pedido.
- **cantidad (int)**: Número de unidades solicitadas.
- **precio_und (int)**: Precio unitario del producto en el pedido.

```
mysql> describe detalles_pedido;
+-----+-----+-----+-----+-----+-----+
| Field      | Type   | Null | Key | Default | Extra           |
+-----+-----+-----+-----+-----+-----+
| id         | bigint | NO   | PRI | NULL    | auto_increment |
| cantidad  | int    | NO   |     | NULL    |                 |
| precio     | double | NO   |     | NULL    |                 |
| comic_id   | bigint | YES  | MUL | NULL    |                 |
| pedido_id  | bigint | YES  | MUL | NULL    |                 |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

8. CONCLUSIONES

Este apartado será dividido en cuatro partes: Dificultades, Alcance y limitaciones, Posibles mejoras a futuro y conclusión.

En este apartado se hablara desde un punto más subjetivo y por lo tanto los temas que se aborden serán tratados de esa manera

8.1 Dificultades

En este proyecto nos hemos encontrado con varias dificultades o limitaciones, ya que al principio del proyecto tuvimos una “crisis” creativa al no saber con exactitud qué tipo de proyecto mercería la pena realizar y cómo sería posible abarcarlo, esto sumado a las nuevas tecnologías que queríamos implementar dado a que parecía un momento ideal para aprender cosas que no dio tiempo a tratar en el curso o que nos quedamos con ganas de aprender durante este.

8.2 Alcance y limitaciones

Al tratarse de una página web de compra y venta de cómics el alcance potencial del proyecto ha ido en aumento correlativamente se avanzaba en su desarrollo pues a medida que conseguimos cubrir alguna de las funciones básicas que teníamos planeadas se nos ocurrían posibles mejoras podrían darle un nivel superior a este proyecto y que podríamos haber abordado más profundamente si no hubiese limitaciones que retrasaran o descartaran esas mejoras.

A grandes rasgos podríamos decir que las funciones básicas que tendría que tener una página de compra y venta de comics están completas cerrando el círculo de venta y compra.

Las limitaciones que hemos tenido más presentes podríamos decir que están ligadas al tiempo, ya que al estar compaginado prácticas y proyecto a la misma vez nos encontrábamos un cierto apurados en ciertos momentos en los que debimos hacer un mayor esfuerzo por entregar progresos que consideramos notables en las entregas llevando esto a una mayor fatiga, esto también lo podemos unir a la multitud de fallos que hemos encontrado al probar o afianzar las funciones de nuestros entornos de desarrollo y al experimentar o afianzar nuevas fronteras en lenguajes como Spring y React.

Todo esto nos ha llevado a tener que ser más realistas acerca de las implementaciones que se podían agregar y mejorar acorde al tiempo y esto ha causado que funcionalidades como la capacidad de hacer reseñas sobre los productos o descuentos se descartaron

Otra de las limitaciones que nos encontramos es a la hora de hacer compras reales dentro del proyecto ya que al inicio del proyecto pensamos que sería una de esas mejoras y detalles que le darían un nivel superior al proyecto, pero esto se vio descartado debido a la cantidad de trámites que había que hacer para que esa opción fuese una posibilidad real, ya que entre ellos se nos pedía crear una empresa y darnos de alta como autónomo y decidimos que esto no sería conveniente.

8.3 Posibles mejoras a futuro

Entre las posibles mejoras a futuro podemos destacar las siguientes:

- La creación de ofertas y packs de productos con ofertas especiales
- La habilidad de crear reseñas sobre productos
- La creación de un foro de preguntas sobre todo lo que rodea el proyecto
- La creación de un enlace para preguntar dudas sobre los productos o las condiciones de envío
- La creación de un chat con IA para resolver dudas
- La habilidad de realizar transacciones reales si el proyecto se continúa desarrollando

8.4 Conclusión

En conclusión podríamos decir que con este proyecto hemos podido aprender un poco mejor las partes que corresponden al Backend y al Frontend, nuevas tecnologías como React y extender nuestros conocimientos sobre Spring Boot, pero nos ha sido una experiencia dura ya que todos los días se nos hacían iguales y repetitivos sin tiempo para nosotros o para otra cosa y eso creemos que ha hecho mella en nuestra productividad y en nuestros estados de ánimo también, dejando eso a un lado con la experiencia ya en el pasado, estamos bastante orgullosos del proyecto que hemos entregado.

9. BIBLIOGRAFÍA

Páginas o fuentes de las cual nos hemos servido o ayudado

- Visual Studio Code: <https://code.visualstudio.com/>
- Página oficial React: <https://es.reactjs.org/>
- Página oficial SpringBoot: <https://spring.io/projects/spring-boot>
- Página oficial Mysql: <https://www.mysql.com/>
- W3schools: <https://www.w3schools.com/>
- Node tutoriales: <https://www.tutorialspoint.com/nodejs/index.htm>
- Node W3Schools: <https://www.w3schools.com/nodejs/default.asp>
- Node Descarga y Documentación: <https://nodejs.org/es/>
- canal de Youtube Midudev: <https://www.youtube.com/midudev>
- ChatGPT: <https://chatgpt.com/>
- Looka: <https://looka.com/logo-maker/>

Páginas o fuentes en las cuales fue inspirado el diseño del proyecto

- PcComponentes: <https://www.pccomponentes.com/>
- Pagina de Puntos del video juego Valorant: <https://playvalorant.com/en-gb/>