



ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS

INNOVACIÓN PARA LA EXCELENCIA

UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

TEMA: EXAMEN PRÁCTICO

DESARROLLO WEB AVANZADO

NRC 2356

INTEGRANTES:

- Ronny Ibarra
- Anthony Villareal
 - Ariel Reyes
- Javier Ramos

FECHA DE ENTREGA: 04-03-2025

NOMBRE DEL PROFESOR: Dorys Chicaiza

Índice

2. Índice	2
3. Resumen.....	3
4. Introducción	3
5. Objetivos	4
6. Desarrollo o Cuerpo del Informe.....	4
7. Resultados.....	55
8. Conclusiones	60
9. Recomendaciones	60
10. Bibliografía o Referencias.....	60
11. Link del git hub	60

3. Resumen

Este proyecto consiste en el desarrollo de una tienda virtual básica que permitirá a los usuarios comprar productos en línea de forma sencilla y segura. Contará con dos tipos de usuarios: clientes y administradores. Los clientes podrán registrarse, explorar productos, agregarlos al carrito y completar sus compras, mientras que los administradores gestionarán productos, pedidos y usuarios.

Para la seguridad, se implementará un sistema de autenticación con JWT, permitiendo a los usuarios iniciar sesión, actualizar su información y cambiar contraseñas. Los administradores podrán gestionar cuentas y restringir accesos cuando sea necesario.

La tienda contará con un sistema de gestión de productos, donde los administradores podrán agregar, modificar y eliminar artículos. Los clientes podrán buscar, filtrar y ordenar productos según diferentes criterios.

El carrito de compras permitirá agregar, modificar o eliminar productos, y la información se guardará en la base de datos para que los usuarios no pierdan su selección. Al confirmar una compra, se generará un pedido con seguimiento en tiempo real, permitiendo su cancelación si aún no ha sido enviado.

Además, habrá un panel de administración donde los administradores podrán visualizar estadísticas de ventas, productos más vendidos y usuarios registrados.

El proyecto usará Spring Boot o Node.js en el backend y Angular en el frontend, con bases de datos MySQL o PostgreSQL. Su objetivo es ofrecer una experiencia de compra fluida y una gestión eficiente para el negocio.

4. Introducción

- **Contexto:** El crecimiento del comercio electrónico ha impulsado la necesidad de soluciones digitales accesibles y funcionales. Este proyecto busca desarrollar una tienda virtual básica que facilite la compra y venta de productos en línea, ofreciendo una plataforma intuitiva tanto para clientes como para administradores. La implementación de tecnologías modernas garantizará un rendimiento óptimo y una experiencia segura para los usuarios.
- **Alcance.**
 - Autenticación y gestión de usuarios con roles diferenciados (cliente y administrador).

- Gestión de productos, permitiendo la creación, edición y eliminación de artículos.
- Carrito de compras, con la posibilidad de agregar, modificar y eliminar productos.
- Desarrollo con tecnologías modernas, como Spring Boot o Node.js para backend, Angular para frontend y bases de datos relacionales.

- **Limitaciones**

- La plataforma estará optimizada para web, sin una versión móvil nativa.
- Inicialmente, la tienda solo manejará inventario básico sin control de stock avanzado.

5. Objetivos

- **General**

Desarrollar una tienda virtual básica utilizando tecnologías modernas en backend y frontend, que permita la autenticación de usuarios, gestión de productos, procesamiento de pedidos y administración de roles, garantizando una experiencia eficiente y segura para clientes y administradores.

- **Específicos**

- Implementar un sistema de autenticación con JWT para que los usuarios puedan registrarse, iniciar sesión y gestionar sus cuentas según su rol (cliente o administrador).
- Desarrollar la gestión de productos y carrito, permitiendo a los administradores administrar productos y a los clientes explorar, filtrar y agregar artículos al carrito fácilmente.
- Optimizar la compra y pedidos, facilitando a los clientes el seguimiento de sus órdenes y permitiendo a los administradores gestionarlas y actualizar su estado.

6. Desarrollo o Cuerpo del Informe.

- **Conceptos base:**

Node.js es un entorno de ejecución para JavaScript basado en el motor V8 de Google Chrome. Permite ejecutar código JavaScript en el servidor, facilitando la creación de aplicaciones escalables y de alto rendimiento mediante un modelo de operaciones asíncronas y no bloqueantes. Es ampliamente utilizado en el desarrollo de APIs,

aplicaciones web y sistemas en tiempo real debido a su eficiencia y capacidad para manejar múltiples solicitudes concurrentes. (Flanagan, 2020)

- **Herramientas de desarrollo:**

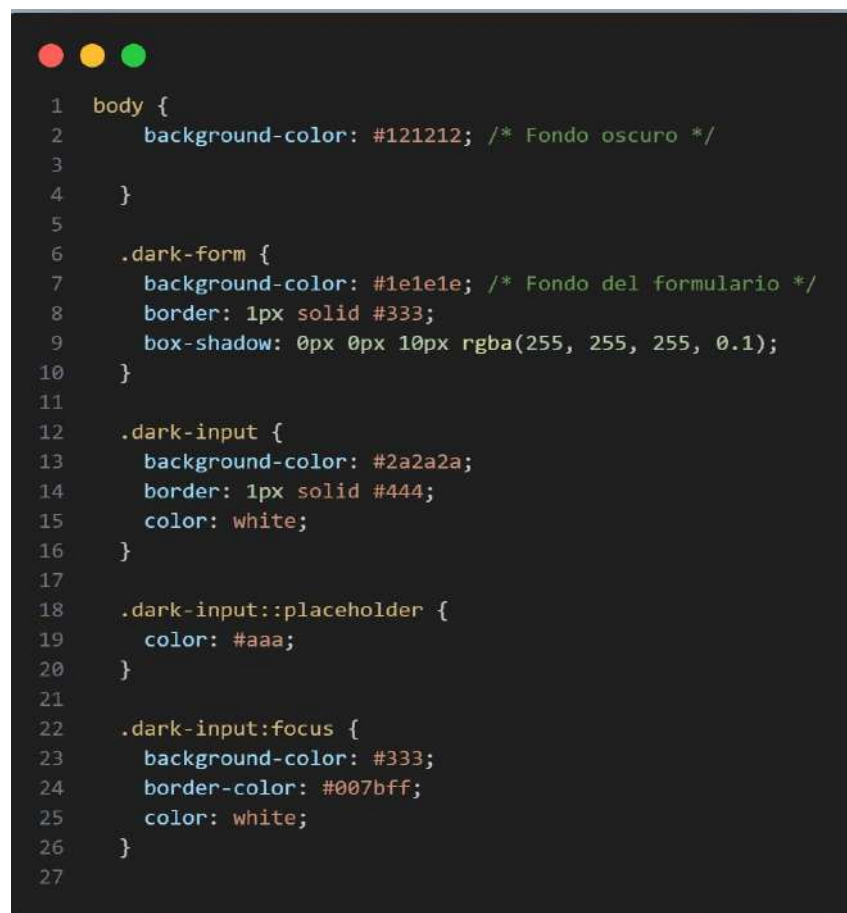
- Node.js
- MySQL
- SQL
- JavaScript
- TypeScript
- Módulos de Node.js
- Angular

- **Cuerpo o Desarrollo**

- **Tienda virtual frontend**

- **Components**

- Actualizar contraseña



```
1  body {
2      background-color: #121212; /* Fondo oscuro */
3  }
4
5
6  .dark-form {
7      background-color: #1e1e1e; /* Fondo del formulario */
8      border: 1px solid #333;
9      box-shadow: 0px 0px 10px rgba(255, 255, 255, 0.1);
10 }
11
12 .dark-input {
13     background-color: #2a2a2a;
14     border: 1px solid #444;
15     color: white;
16 }
17
18 .dark-input::placeholder {
19     color: #aaa;
20 }
21
22 .dark-input:focus {
23     background-color: #333;
24     border-color: #007bff;
25     color: white;
26 }
27
```

Fig. 1 *actualizar-contrasena.component.css*

Explicación: Este código CSS aplica un modo oscuro a una página web, estableciendo un fondo oscuro general y diseñando formularios con un estilo moderno. Define un formulario con un fondo gris oscuro, bordes sutiles y una ligera sombra para resaltar su apariencia. Los campos de entrada tienen un fondo oscuro, texto blanco y placeholders en gris claro, cambiando su estilo al enfocarse con un borde azul. Esto mejora la estética y la experiencia del usuario en interfaces con diseño oscuro.

```

1 <div class="container mt-4">
2   <div class="row justify-content-center">
3     <div class="col-md-6">
4       <h2 class="text-center mb-4 text-dark">Actualizar Contraseña</h2>
5
6       <form (ngSubmit)="actualizarContrasena()" class="dark-form p-4 rounded">
7         <!-- Contraseña actual -->
8         <div class="mb-3">
9           <label for="actual" class="form-label text-light">Contraseña actual</label>
10          <input
11            [(ngModel)]="contrasena.actual"
12            name="actual"
13            type="password"
14            id="actual"
15            class="form-control dark-input"
16            placeholder="Ingrese su contraseña actual"
17            required
18          />
19        </div>
20
21        <!-- Nueva contraseña -->
22        <div class="mb-3">
23          <label for="nueva" class="form-label text-light">Nueva contraseña</label>
24          <input
25            [(ngModel)]="contrasena.nueva"
26            name="nueva"
27            type="password"
28            id="nueva"
29            class="form-control dark-input"
30            placeholder="Ingrese su nueva contraseña"
31            required
32          />
33        </div>
34
35        <!-- Botón de actualizar -->
36        <div class="d-grid">
37          <button type="submit" class="btn btn-primary">
38            <i class="bi bi-key-fill"></i> Actualizar Contraseña
39          </button>
40        </div>
41      </form>
42
43      <!-- Botón de volver al perfil -->
44      <div class="text-center mt-3">
45        <a routerLink="/perfil" class="btn btn-secondary">
46          <i class="bi bi-arrow-left"></i> Volver al perfil
47        </a>
48      </div>
49    </div>
50  </div>
51 </div>
52

```

Fig. 2 *actualizar-contrasena.component.html*

Explicación: Este código HTML con Angular crea un formulario para actualizar la contraseña de un usuario. Incluye dos campos de entrada: uno para la contraseña actual y otro para la nueva contraseña, ambos con validación requerida y estilo oscuro. Al hacer clic en el botón "Actualizar Contraseña", se ejecuta la función `actualizarContrasena()`, que manejará la lógica de actualización. Además, hay un botón de regreso que redirige al perfil del usuario (`/perfil`). Todo está diseñado con Bootstrap para un diseño responsive y atractivo.

```

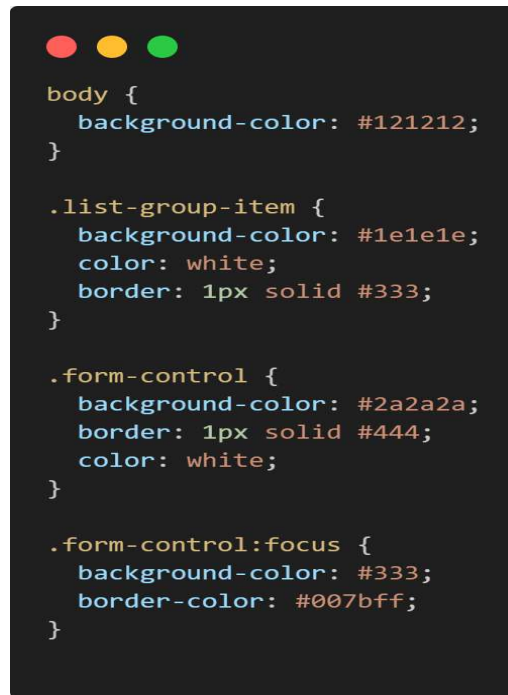
1 import { Component } from '@angular/core';
2 import { FormsModule } from '@angular/forms';
3 import { AuthService } from '../services/auth.service';
4 import { Router, RouterLink } from '@angular/router';
5
6 @Component({
7   selector: 'app-actualizar-contrasena',
8   standalone: true,
9   imports: [FormsModule, RouterLink],
10  templateUrl: './actualizar-contrasena.component.html',
11  styleUrls: ['./actualizar-contrasena.component.css'],
12 })
13 export class ActualizarContrasenaComponent {
14   contrasena = {
15     actual: '',
16     nueva: '',
17   };
18
19   constructor(private authService: AuthService, private router: Router) {}
20
21   actualizarContrasena() {
22     this.authService.actualizarContrasena(this.contrasena.nueva).subscribe({
23       next: (response) => {
24         console.log('Contraseña actualizada:', response);
25         alert('Contraseña actualizada exitosamente');
26         this.router.navigate(['/perfil']);
27       },
28       error: (error) => {
29         console.error('Error al actualizar contraseña:', error);
30         alert('Error al actualizar contraseña');
31       },
32     });
33   }
34 }
35

```

Fig. 3 *actualizar-contrasena.component.ts*

Explicación: Este código en Angular define un componente llamado ActualizarContrasenaComponent, que permite a los usuarios actualizar su contraseña. Contiene un formulario con dos campos (actual y nueva) y utiliza AuthService para enviar la nueva contraseña al backend. Si la actualización es exitosa, muestra un mensaje de confirmación y redirige al usuario a la página de perfil (/perfil). En caso de error, muestra una alerta y un mensaje en la consola.

- Carrito



```
body {  
  background-color: #121212;  
}  
  
.list-group-item {  
  background-color: #1e1e1e;  
  color: white;  
  border: 1px solid #333;  
}  
  
.form-control {  
  background-color: #2a2a2a;  
  border: 1px solid #444;  
  color: white;  
}  
  
.form-control:focus {  
  background-color: #333;  
  border-color: #007bff;  
}
```

Fig. 4 *carrito.component.css*

Explicación: Este código CSS aplica un modo oscuro a una página web, dando un fondo negro (#121212) y personalizando listas y formularios. Las listas tienen un fondo gris oscuro (#1e1e1e), texto blanco y bordes sutiles. Los formularios usan un fondo gris (#2a2a2a), con bordes oscuros y texto blanco. Al enfocar un campo, el fondo se oscurece más (#333) y el borde cambia a azul (#007bff). Esto mejora la estética y usabilidad en interfaces oscuras.


```

1 <div class="container mt-4">
2   <h2 class="text-center text-dark">Carrito de Compras</h2>
3
4   <!-- Lista de productos en el carrito -->
5   <ul class="list-group mt-3">
6     <li *ngFor="let item of carrito" class="list-group-item d-flex justify-content-between align-items-center">
7       <div>
8         <strong>{{ item.nombre }}</strong> - <span class="text-primary">Precio: ${{ item.precio }}</span>
9       </div>
10      <div class="d-flex align-items-center">
11        <input
12          type="number"
13          [(ngModel)]="item.cantidad"
14          (change)="actualizarCantidad(item.producto_id, item.cantidad)"
15          min="1"
16          class="form-control me-2"
17          style="width: 80px;"
18        />
19        <button (click)="eliminarProducto(item.producto_id)" class="btn btn-danger">
20          <i class="bi bi-trash"></i> Eliminar
21        </button>
22      </div>
23    </li>
24  </ul>
25
26  <!-- Mensaje cuando el carrito está vacío -->
27  <div *ngIf="carrito.length === 0" class="text-center mt-4">
28    <p class="alert alert-warning">El carrito está vacío.</p>
29  </div>
30
31  <!-- Botón para crear pedido -->
32  <div class="text-center mt-4" *ngIf="carrito.length > 0">
33    <button (click)="crearPedido()" class="btn btn-success btn-lg">
34      <i class="bi bi-cart-check"></i> Crear Pedido
35    </button>
36  </div>
37 </div>
38
39

```

Fig. 5 *carrito.component.html*

Explicación: Este código en Angular muestra un carrito de compras, listando productos con su nombre, precio y cantidad editable. Permite actualizar la cantidad (`actualizarCantidad()`) y eliminar productos (`eliminarProducto()`). Si el carrito está vacío, se muestra un mensaje de advertencia, y si tiene productos, aparece un botón para crear un pedido (`crearPedido()`). Usa `ngFor` para iterar los productos y `ngIf` para mostrar elementos según el estado del carrito, con diseño basado en Bootstrap.

```

1 import { Component, OnInit } from '@angular/core';
2 import { CarritoService } from '.../services/carrito.service';
3 import { PedidoService } from '.../services/pedido.service'; // Nueva importación
4 import { FormsModule } from '@angular/forms';
5 import { CommonModule } from '@angular/common';
6
7 @Component({
8   selector: 'app-carrito',
9   standalone: true,
10  imports: [FormsModule, CommonModule],
11  templateUrl: './carrito.component.html',
12  styleUrls: ['./carrito.component.css'],
13 })
14 export class CarritoComponent implements OnInit {
15   carrito: any[] = [];
16
17   constructor(private carritoService: CarritoService, private pedidoService: PedidoService) {}
18
19   ngOnInit() {
20     this.cargarCarrito();
21   }
22
23   cargarCarrito() {
24     this.carritoService.obtenerCarrito().subscribe({
25       next: (response) => {
26         this.carrito = response;
27       },
28       error: (error) => {
29         console.error('Error al cargar el carrito:', error);
30         alert('Error: ${error.error?.mensaje || 'No se pudo cargar el carrito'}');
31       },
32     });
33   }
34
35   actualizarCantidad(productoId: string, cantidad: number) {
36     if (cantidad < 1) return;
37     this.carritoService.actualizarCantidad(productoId, cantidad).subscribe({
38       next: (response) => {
39         console.log('Cantidad actualizada:', response);
40         this.cargarCarrito();
41       },
42       error: (error) => {
43         console.error('Error al actualizar cantidad:', error);
44         alert('Error: ${error.error?.mensaje || 'No se pudo actualizar la cantidad'}');
45       },
46     });
47   }
48
49   eliminarProducto(productoId: string) {
50     this.carritoService.eliminarProducto(productoId).subscribe({
51       next: (response) => {
52         console.log('Producto eliminado:', response);
53         this.cargarCarrito();
54       },
55       error: (error) => {
56         console.error('Error al eliminar producto:', error);
57         alert('Error: ${error.error?.mensaje || 'No se pudo eliminar el producto'}');
58       },
59     });
60   }
61
62   crearPedido() {
63     this.pedidoService.crearPedido().subscribe({
64       next: (response) => {
65         console.log('Pedido creado:', response);
66         alert('Pedido creado exitosamente');
67         this.cargarCarrito(); // Refrescar el carrito (debería estar vacío ahora)
68       },
69       error: (error) => {
70         console.error('Error al crear pedido:', error);
71         alert('Error: ${error.error?.mensaje || 'No se pudo crear el pedido'}');
72       },
73     });
74   }
75 }
76

```

Fig. 6 *carrito.component.ts*

Explicación: Este código define un componente en Angular llamado CarritoComponent, encargado de gestionar la funcionalidad del carrito de compras. Al inicializarse (ngOnInit), carga los productos del carrito (cargarCarrito()) a través de CarritoService. Permite actualizar la cantidad de productos (actualizarCantidad()), eliminarlos (eliminarProducto()) y crear un pedido (crearPedido()), usando servicios para comunicarse con el backend. Cada acción actualiza la vista del carrito y muestra mensajes de éxito o error según la respuesta del servidor.

- dashboard



```
1  body {  
2    background-color: #121212;  
3  }  
4  
5  .card {  
6    border: 1px solid #333;  
7  }  
8  
9  .list-group-item {  
10   background-color: #2a2a2a;  
11   border: 1px solid #444;  
12   color: white;  
13 }
```

Fig. 7 *dashboard.component.css*

Explicación: Este código CSS aplica un modo oscuro, estableciendo un fondo negro (#121212). Las tarjetas (.card) tienen un borde gris oscuro (#333), y los elementos de lista (.list-group-item) cuentan con un fondo gris (#2a2a2a), borde (#444) y texto blanco, mejorando la estética y legibilidad en interfaces oscuras.

```

1 <div class="container mt-4">
2   <h2 class="text-center text-dark">Dashboard de Administración</h2>
3
4   <!-- Mostrar estadísticas si están cargadas -->
5   <div *ngIf="estadisticas.totalVentas !== undefined; else cargando">
6     <div class="row mt-4">
7       <!-- Tarjeta: Total de Ventas -->
8       <div class="col-md-4">
9         <div class="card bg-dark text-light shadow">
10           <div class="card-body text-center">
11             <h3 class="card-title">Total de Ventas</h3>
12             <p class="fs-4 fw-bold text-success">{{ estadisticas.totalVentas | currency }}</p>
13           </div>
14         </div>
15       </div>
16
17       <!-- Tarjeta: Usuarios Registrados -->
18       <div class="col-md-4">
19         <div class="card bg-dark text-light shadow">
20           <div class="card-body text-center">
21             <h3 class="card-title">Usuarios Registrados</h3>
22             <p class="fs-4 fw-bold text-primary">{{ estadisticas.totalUsuarios }}</p>
23           </div>
24         </div>
25       </div>
26
27       <!-- Tarjeta: Productos Más Vendidos -->
28       <div class="col-md-4">
29         <div class="card bg-dark text-light shadow">
30           <div class="card-body">
31             <h3 class="card-title text-center">Productos Más Vendidos</h3>
32             <ul class="list-group">
33               <li *ngFor="let producto of estadisticas.productosMasVendidos" class="list-group-item bg-secondary text-light">
34                 <strong>{{ producto.nombre }}</strong> - {{ producto.total_vendido }} unidades
35               </li>
36             </ul>
37           </div>
38         </div>
39       </div>
40     </div>
41   </div>
42
43   <!-- Mensaje de carga -->
44   <ng-template #cargando>
45     <div class="text-center mt-4">
46       <div class="spinner-border text-primary" role="status"></div>
47       <p class="mt-2 text-light">Cargando estadísticas...</p>
48     </div>
49   </ng-template>
50 </div>
51

```

Fig. 8 *dashboard.component.html*

Explicación: Este código en Angular muestra un dashboard de administración, presentando estadísticas como total de ventas, usuarios registrados y productos más vendidos. Utiliza ngIf para mostrar los datos cuando están disponibles y un spinner de carga (ng-template) mientras se obtienen. Cada estadística se presenta en tarjetas (card) estilizadas con Bootstrap, organizadas en una cuadrícula (row). La lista de productos más vendidos usa ngFor para recorrer y mostrar cada producto con su cantidad vendida.

```

import { Component, OnInit } from '@angular/core';
import { AdminService } from '../../services/admin.service';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-dashboard',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.css'],
})
export class DashboardComponent implements OnInit {
  estadisticas: any = {};

  constructor(private adminService: AdminService) {}

  ngOnInit() {
    this.cargarEstadisticas();
  }

  cargarEstadisticas() {
    this.adminService.obtenerEstadisticas().subscribe({
      next: (response) => {
        this.estadisticas = response;
      },
      error: (error) => {
        console.error('Error al cargar estadísticas:', error);
        alert(`Error: ${error.error?.mensaje || 'No se pudo cargar las estadísticas'}`);
      }
    });
  }
}

```

Fig. 9 *dashboard.component.ts*

Explicación: Este código define un componente en Angular llamado `DashboardComponent`, encargado de cargar y mostrar estadísticas en un panel de administración. Al inicializarse (`ngOnInit`), ejecuta `cargarEstadisticas()`, que obtiene los datos desde `AdminService` mediante una suscripción (`subscribe`). Si la respuesta es exitosa, almacena las estadísticas; en caso de error, muestra un mensaje en la consola y una alerta al usuario. Usa `CommonModule` para importar funcionalidades comunes en Angular.

- `gestion-categorias`

```
1  body {
2    background-color: #121212;
3  }
4
5
6  .table-dark {
7    background-color: #2a2a2a;
8  }
9
10 .table-dark th, .table-dark td {
11   border: 1px solid #444;
12 }
13
14 .form-control {
15   background-color: #2a2a2a;
16   border: 1px solid #444;
17   color: white; /* Color del texto dentro de los inputs */
18 }
19
20 .form-control:focus {
21   background-color: #333;
22   border-color: #007bff;
23   color: white; /* Asegura que el texto sea blanco incluso cuando el input está enfocado */
24 }
25
26 textarea.form-control {
27   color: white; /* Color del texto dentro del textarea */
28 }
29
30 input::placeholder, textarea::placeholder {
31   color: #bbb; /* Color para el texto del placeholder */
32 }
33
```

Fig. 10 *gestion-categorias.component.css*

Explicación: Este CSS aplica un modo oscuro, dando un fondo negro (#121212) y estilizando tablas, inputs y formularios. Las tablas (.table-dark) tienen fondo gris oscuro (#2a2a2a) y bordes (#444). Los inputs (.form-control) usan fondo oscuro, texto blanco y borde azul (#007bff) al enfocarse. Los textarea mantienen el texto blanco, y los placeholders son gris claro (#bbb), mejorando la legibilidad en interfaces oscuras.

```

1 <div class="container mt-4">
2   <h2 class="text-center text-dark">Gestión de Categorías</h2>
3
4   <!-- Formulario para crear nueva categoría -->
5   <div class="mt-4">
6     <h3>Crear Nueva Categoría</h3>
7     <form (ngSubmit)="crearCategoría()">
8       <div class="mb-3">
9         <input
10           [(ngModel)]="nuevaCategoría.nombre"
11           name="nombre"
12           placeholder="Nombre"
13           required
14           class="form-control"
15         />
16       </div>
17       <div class="mb-3">
18         <textarea
19           [(ngModel)]="nuevaCategoría.descripcion"
20           name="descripcion"
21           placeholder="Descripción"
22           class="form-control"
23         ></textarea>
24       </div>
25       <button type="submit" class="btn btn-primary">Crear Categoría</button>
26     </form>
27   </div>
28
29   <!-- Mostrar categorías si existen -->
30   <div *ngIf="categorías.length > 0; else sinCategorías" class="mt-4">
31     <h3>Lista de Categorías</h3>
32     <table class="table table-dark table-striped">
33       <thead>
34         <tr>
35           <th>ID</th>
36           <th>Nombre</th>
37           <th>Descripción</th>
38           <th>Acciones</th>
39         </tr>
40       </thead>
41       <tbody>
42         <tr *ngFor="let categoría of categorías">
43           <td>{{ categoría.id }}</td>
44           <td>{{ categoría.nombre }}</td>
45           <td>{{ categoría.descripcion }}</td>
46           <td>
47             <button (click)="editarCategoría(categoría)" class="btn btn-warning btn-sm">
48               <i class="bi bi-pencil"></i> Editar
49             </button>
50             <button (click)="eliminarCategoría(categoría.id)" class="btn btn-danger btn-sm">
51               <i class="bi bi-trash"></i> Eliminar
52             </button>
53           </td>
54         </tr>
55       </tbody>
56     </table>
57   </div>
58
59   <!-- Formulario para editar categoría -->
60   <div *ngIf="categoríaEditada" class="edit-form mt-4">
61     <h3>Editar Categoría #{{ categoríaEditada.id }}</h3>
62     <div class="mb-3">
63       <input
64         [(ngModel)]="categoríaEditada.nombre"
65         placeholder="Nombre"
66         required
67         class="form-control"
68       />
69     </div>
70     <div class="mb-3">
71       <textarea
72         [(ngModel)]="categoríaEditada.descripcion"
73         placeholder="Descripción"
74         class="form-control"
75       ></textarea>
76     </div>
77     <button (click)="actualizarCategoría()" class="btn btn-success">Guardar</button>
78     <button (click)="categoríaEditada = null" class="btn btn-secondary">Cancelar</button>
79   </div>
80
81   <!-- Mensaje si no hay categorías -->
82   <ng-template #sinCategorías>
83     <p class="alert alert-warning mt-4">No hay categorías registradas.</p>
84   </ng-template>
85 </div>
86

```

Fig. 11 *gestion-categorias.component.html*

Explicación: Este código en Angular implementa la gestión de categorías, permitiendo crear, editar y eliminar categorías. Contiene un formulario para agregar nuevas categorías con `ngSubmit`, y una tabla que muestra la lista existente (`ngFor`). Cada categoría tiene botones para editar (`editarCategoría()`) o eliminar (`eliminarCategoría()`). También incluye un formulario dinámico para editar (`ngIf="categoríaEditada"`), permitiendo modificar y guardar cambios (`actualizarCategoría()`). Si no hay categorías, muestra un mensaje de advertencia (`ng-template`). Usa Bootstrap para mejorar la apariencia y diseño.


```

1 import { Component, OnInit } from '@angular/core';
2 import { AdminService } from '../services/admin.service';
3 import { CommonModule } from '@angular/common';
4 import { FormsModule } from '@angular/forms';
5
6 @Component({
7   selector: 'app-gestion-categorias',
8   standalone: true,
9   imports: [CommonModule, FormsModule],
10  templateUrl: './gestion-categorias.component.html',
11  styleUrls: ['./gestion-categorias.component.css'],
12 })
13 export class GestionCategoriasComponent implements OnInit {
14   categorias: any[] = [];
15   nuevaCategoria = { nombre: '', descripcion: '' };
16   categoriaEditada: any = null;
17
18   constructor(private adminService: AdminService) {}
19
20   ngOnInit() {
21     this.cargarCategorias();
22   }
23
24   cargarCategorias() {
25     this.adminService.listarCategorias().subscribe({
26       next: (response) => {
27         this.categorias = response;
28       },
29       error: (error) => {
30         console.error('Error al cargar categorías:', error);
31         alert('Error: ${error.error?.mensaje} || No se pudo cargar las categorías');
32       },
33     });
34   }
35
36   crearCategoria() {
37     this.adminService.crearCategoria(this.nuevaCategoria.nombre, this.nuevaCategoria.descripcion).subscribe({
38       next: (response) => {
39         console.log('Categoria creada:', response);
40         alert('Categoria creada exitosamente');
41         this.nuevaCategoria = { nombre: '', descripcion: '' };
42         this.cargarCategorias();
43       },
44       error: (error) => {
45         console.error('Error al crear categoria:', error);
46         alert('Error: ${error.error?.mensaje} || No se pudo crear la categoria');
47       },
48     });
49   }
50
51   editarCategoria(categoria: any) {
52     this.categoriaEditada = { ...categoria };
53   }
54
55   actualizarCategoria() {
56     if (this.categoriaEditada) {
57       this.adminService
58         .actualizarCategoria(this.categoriaEditada.id, this.categoriaEditada.nombre, this.categoriaEditada.descripcion)
59         .subscribe({
60           next: (response) => {
61             console.log('Categoria actualizada:', response);
62             alert('Categoria actualizada exitosamente');
63             this.categoriaEditada = null;
64             this.cargarCategorias();
65           },
66           error: (error) => {
67             console.error('Error al actualizar categoria:', error);
68             alert('Error: ${error.error?.mensaje} || No se pudo actualizar la categoria');
69           },
70         });
71     }
72   }
73
74   eliminarCategoria(categoriaId: string) {
75     if (confirm('¿Estás seguro de que quieres eliminar esta categoría?')) {
76       this.adminService.eliminarCategoria(categoriaId).subscribe({
77         next: (response) => {
78           console.log('Categoria eliminada:', response);
79           alert('Categoria eliminada exitosamente');
80           this.cargarCategorias();
81         },
82         error: (error) => {
83           console.error('Error al eliminar categoria:', error);
84           alert('Error: ${error.error?.mensaje} || No se pudo eliminar la categoria');
85         },
86       });
87     }
88   }
89 }

```

Fig. 12 *gestion-categorias.component.ts*

Explicación: Este código en Angular define el `GestionCategoriasComponent`, encargado de administrar categorías en una aplicación. Al inicializarse (`ngOnInit`), carga la lista de categorías (`cargarCategorias()`) desde `AdminService`. Permite crear (`crearCategoria()`), editar (`editarCategoria()`), actualizar (`actualizarCategoria()`) y eliminar (`eliminarCategoria()`) categorías, mostrando alertas y registrando mensajes en la consola según el resultado. Utiliza `CommonModule` y `FormsModule` para gestionar formularios y manejar la interacción con el usuario.

- gestion-pedidos

```
1  body {
2    background-color: #121212;
3  }
4
5  .pedido {
6    background-color: #2a2a2a;
7  }
8
9  .pedido p, .pedido h4 {
10   color: white;
11 }
12
13 .pedido button {
14   color: white;
15 }
16
17 .form-control, .form-select {
18   background-color: #333;
19   color: white;
20   border: 1px solid #444;
21 }
22
23 .form-control:focus, .form-select:focus {
24   border-color: #007bff;
25 }
26
27 .alert-warning {
28   background-color: #333;
29   color: white;
30 }
31
32 .list-group-item {
33   background-color: #2a2a2a;
34   color: white;
35 }
36
37 input::placeholder, textarea::placeholder{
38   color: #bbb; /* Color para el texto del placeholder */
39 }
```

Fig. 13 *gestion-pedidos.component.css*

Explicación: Este CSS aplica un modo oscuro, estableciendo un fondo negro (#121212) y estilizando formularios, alertas y listas. Los elementos `.pedido`, `.list-group-item` y `.alert-warning` tienen fondo oscuro y texto blanco. Los formularios (`.form-control`, `.form-select`) usan fondo gris (#333), bordes (#444) y resaltado azul (#007bff) al enfocarse. Los placeholders son gris claro (#bbb) para mejorar la legibilidad.

```

1 <div class="container mt-4">
2   <h2 class="text-center text-dark">Gestión de Pedidos</h2>
3
4   <!-- Mostrar pedidos si existen -->
5   <div *ngIf="pedidos.length > 0; else sinPedidos">
6     <div *ngFor="let pedido of pedidos" class="pedido mb-4 p-3 border rounded bg-dark">
7       <p><strong>ID del Pedido:</strong> {{ pedido.id }}</p>
8       <p><strong>Usuario:</strong> {{ pedido.usuario_nombre }}</p>
9       <p><strong>Total:</strong> {{ pedido.total | currency }}</p>
10      <p><strong>Estado:</strong> {{ pedido.estado }}</p>
11      <p><strong>Fecha de Creación:</strong> {{ pedido.creado_en | date:'medium' }}</p>
12      <p *ngIf="pedido.transportista"><strong>Transportista:</strong> {{ pedido.transportista }}</p>
13      <p *ngIf="pedido.numero_seguimiento"><strong>Número de Seguimiento:</strong> {{ pedido.numero_seguimiento }}</p>
14
15      <div>
16        <h4>Detalles:</h4>
17        <ul class="list-group">
18          <li *ngFor="let detalle of pedido.detalles" class="list-group-item bg-dark text-light">
19            {{ detalle.nombre }} - Cantidad: {{ detalle.cantidad }} - Precio Unitario: {{ detalle.precio_unitario | currency }}
20          </li>
21        </ul>
22      </div>
23
24      <button (click)="editarPedido(pedido)" class="btn btn-warning btn-sm mt-2">
25        <i class="bi bi-pencil"></i> Editar Estado
26      </button>
27    </div>
28
29    <!-- Formulario de edición -->
30    <div *ngIf="pedidoEditado" class="edit-form mt-4 p-3 border rounded bg-dark">
31      <h3>Editar Pedido #{{ pedidoEditado.id }}</h3>
32      <div class="mb-3">
33        <label for="estado" class="form-label text-light">Estado:</label>
34        <select [(ngModel)]="pedidoEditado.estado" id="estado" class="form-select">
35          <option value="Pendiente">Pendiente</option>
36          <option value="Enviado">Enviado</option>
37          <option value="Entregado">Entregado</option>
38          <option value="Cancelado">Cancelado</option>
39        </select>
40      </div>
41
42      <div class="mb-3">
43        <label for="transportista" class="form-label text-light">Transportista:</label>
44        <input [(ngModel)]="pedidoEditado.transportista" id="transportista" class="form-control" placeholder="Transportista" />
45      </div>
46
47      <div class="mb-3">
48        <label for="numero_seguimiento" class="form-label text-light">Número de Seguimiento:</label>
49        <input [(ngModel)]="pedidoEditado.numero_seguimiento" id="numero_seguimiento" class="form-control" placeholder="Número de Seguimiento" />
50      </div>
51
52      <button (click)="guardarCambios()" class="btn btn-success">
53        <i class="bi bi-save"></i> Guardar Cambios
54      </button>
55      <button (click)="cancelarEdicion()" class="btn btn-secondary ms-2">
56        <i class="bi bi-x-circle"></i> Cancelar
57      </button>
58    </div>
59  </div>
60
61  <!-- Mensaje si no hay pedidos -->
62  <ng-template #sinPedidos>
63    <p class="alert alert-warning mt-4">No hay pedidos para mostrar.</p>
64  </ng-template>
65 </div>
66

```

Fig. 14 *gestion-pedidos.component.html*

Explicación: Este código en Angular gestiona la visualización y edición de pedidos. Usa ngIf para mostrar la lista de pedidos (ngFor) con detalles como usuario, estado, fecha y transportista. Cada pedido tiene un botón para editar su estado (editarPedido()). Si se selecciona un pedido, aparece un formulario de edición donde se puede actualizar el estado, transportista y número de seguimiento, con opciones para guardar cambios (guardarCambios()) o cancelar (cancelarEdicion()). Si no hay pedidos, se muestra un mensaje de alerta. Usa Bootstrap para mejorar el diseño.

```

1 import { Component, OnInit } from '@angular/core';
2 import { PedidoService } from '../services/pedido.service';
3 import { CommonModule } from '@angular/common';
4 import { FormsModule } from '@angular/forms';
5
6 @Component({
7   selector: 'app-gestion-pedidos',
8   standalone: true,
9   imports: [CommonModule, FormsModule],
10  templateUrl: './gestion-pedidos.component.html',
11  styleUrls: ['./gestion-pedidos.component.css'],
12 })
13 export class GestionPedidosComponent implements OnInit {
14   pedidos: any[] = [];
15   pedidoEditado: any = null;
16
17   constructor(private pedidoService: PedidoService) {}
18
19   ngOnInit() {
20     this.cargarPedidos();
21   }
22
23   cargarPedidos() {
24     this.pedidoService.obtenerTodosPedidos().subscribe({
25       next: (response) => {
26         this.pedidos = response;
27       },
28       error: (error) => {
29         console.error('Error al cargar pedidos:', error);
30         alert('Error: ${error.error?.mensaje || 'No se pudo cargar los pedidos'}');
31       },
32     });
33   }
34
35   editarPedido(pedido: any) {
36     this.pedidoEditado = { ...pedido, transportista: pedido.transportista || '', numero_seguimiento: pedido.numero_seguimiento || '' };
37   }
38
39   guardarCambios() {
40     if (this.pedidoEditado) {
41       this.pedidoService
42         .actualizarEstado(
43           this.pedidoEditado.id,
44           this.pedidoEditado.estado,
45           this.pedidoEditado.transportista,
46           this.pedidoEditado.numero_seguimiento
47         )
48         .subscribe({
49           next: (response) => {
50             console.log('Estado actualizado:', response);
51             alert('Estado actualizado exitosamente');
52             this.pedidoEditado = null;
53             this.cargarPedidos();
54           },
55           error: (error) => {
56             console.error('Error al actualizar estado:', error);
57             alert('Error: ${error.error?.mensaje || 'No se pudo actualizar el estado'}');
58           },
59         });
60     }
61   }
62
63   cancelarEdicion() {
64     this.pedidoEditado = null;
65   }
66 }

```

Fig. 15 *gestion-pedidos.component.ts*

Explicación: Este código en Angular define el GestionPedidosComponent, encargado de administrar pedidos. Al inicializar (ngOnInit), carga la lista de pedidos (cargarPedidos()) desde PedidoService. Permite editar un pedido (editarPedido()), estableciendo su estado, transportista y número de seguimiento. Los cambios se guardan con guardarCambios(), enviándolos al backend y recargando la lista. También se incluye cancelarEdicion() para descartar modificaciones. Muestra alertas y maneja errores si la actualización falla.

- gestion-productos

```

1  body {
2    background-color: #f0f0f0;
3    color: black;
4  }
5
6  h2, h3 {
7    color: black;
8  }
9
10 .form-control, .form-select {
11   background-color: #333;
12   color: white;
13   border: 1px solid #ccc;
14 }
15
16 .form-control:focus, .form-select:focus {
17   border-color: #007bff;
18 }
19
20 .list-group-item {
21   background-color: #f8f9fa;
22   color: black;
23 }
24
25 .list-group-item:hover {
26   background-color: #e2e6ea;
27 }
28
29 .btn {
30   color: white;
31 }
32
33 .btn-primary {
34   background-color: #007bff;
35   border-color: #007bff;
36 }
37
38 .btn-success {
39   background-color: #28a745;
40   border-color: #28a745;
41 }
42
43 .btn-warning {
44   background-color: #ffc107;
45   border-color: #ffc107;
46 }
47
48 .btn-danger {
49   background-color: #dc3545;
50   border-color: #dc3545;
51 }
52
53 .text-dark {
54   color: black;
55 }
56
57 input::placeholder, textarea::placeholder{
58   color: #bbb; /* Color para el texto del placeholder */
59 }
60
61 /* Estilo para las opciones del select */
62 .form-select option {
63   color: #bbb; /* Color del texto de las opciones */
64   background-color: #333; /* Fondo de las opciones */
65 }

```

Fig. 16 *gestion-productos.component.css*

Explicación: Este CSS aplica un tema claro, con fondo gris (#f0f0f0) y texto negro. Los formularios (.form-control, .form-select) tienen fondo oscuro (#333), texto blanco y borde gris, resaltándose en azul (#007bff) al enfocarse. Las listas (.list-group-item) son claras y cambian de color al pasar el mouse. Los botones usan colores distintivos: azul (.btn-primary), verde (.btn-success), amarillo (.btn-warning) y rojo (.btn-danger). También se estilizan placeholders y opciones de select.

```

1 <div class="container mt-4">
2   <h2 class="text-center text-dark">Gestión de Productos</h2>
3
4   <!-- Formulario para crear producto -->
5   <div *ngIf="!editando">
6     <h3 class="text-dark">Crear Nuevo Producto</h3>
7     <form (ngSubmit)="crearProducto()">
8       <div class="mb-3">
9         <label for="nombre" class="form-label text-dark">Nombre:</label>
10        <input [(ngModel)]="nuevoProducto.nombre" name="nombre" id="nombre" class="form-control" placeholder="Nombre del producto" required />
11      </div>
12
13      <div class="mb-3">
14        <label for="descripcion" class="form-label text-dark">Descripción:</label>
15        <textarea [(ngModel)]="nuevoProducto.descripcion" name="descripcion" id="descripcion" class="form-control" placeholder="Descripción del producto"/></textarea>
16      </div>
17
18      <div class="mb-3">
19        <label for="precio" class="form-label text-dark">Precio:</label>
20        <input [(ngModel)]="nuevoProducto.precio" name="precio" id="precio" type="number" class="form-control" placeholder="Precio" required />
21      </div>
22
23      <div class="mb-3">
24        <label for="imagenes" class="form-label text-dark">URLs de Imágenes:</label>
25        <input [(ngModel)]="nuevoProducto.imagenes" name="imagenes" id="imagenes" class="form-control" placeholder="URLs de imágenes (separadas por comas)" />
26      </div>
27
28      <div class="mb-3">
29        <label for="stock" class="form-label text-dark">Stock:</label>
30        <input [(ngModel)]="nuevoProducto.stock" name="stock" id="stock" type="number" class="form-control" placeholder="Cantidad de stock" required />
31      </div>
32
33      <div class="mb-3">
34        <label for="categoria_id" class="form-label text-dark">Categoría:</label>
35        <select [(ngModel)]="nuevoProducto.categoria_id" name="categoria_id" id="categoria_id" class="form-select" required>
36          <option value="" disabled selected>Selecciona una categoría</option>
37          <option *ngFor="let categoria of categorias" [value]="categoria.id">{{ categoria.nombre }}</option>
38        </select>
39      </div>
40
41      <button type="submit" class="btn btn-primary">Crear Producto</button>
42    </form>
43  </div>
44
45  <!-- Formulario para editar producto -->
46  <div *ngIf="editando">
47    <h3 class="text-dark">Editar Producto</h3>
48    <form (ngSubmit)="actualizarProducto()">
49      <div class="mb-3">
50        <label for="nombreEdit" class="form-label text-dark">Nombre:</label>
51        <input [(ngModel)]="productoEditado.nombre" name="nombre" id="nombreEdit" class="form-control" placeholder="Nombre del producto" required />
52      </div>
53
54      <div class="mb-3">
55        <label for="descripcionEdit" class="form-label text-dark">Descripción:</label>
56        <textarea [(ngModel)]="productoEditado.descripcion" name="descripcion" id="descripcionEdit" class="form-control" placeholder="Descripción del producto"/></textarea>
57      </div>
58
59      <div class="mb-3">
60        <label for="precioEdit" class="form-label text-dark">Precio:</label>
61        <input [(ngModel)]="productoEditado.precio" name="precio" id="precioEdit" type="number" class="form-control" placeholder="Precio" required />
62      </div>
63
64      <div class="mb-3">
65        <label for="imagenesEdit" class="form-label text-dark">URLs de Imágenes:</label>
66        <input [(ngModel)]="productoEditado.imagenes" name="imagenes" id="imagenesEdit" class="form-control" placeholder="URLs de imágenes (separadas por comas)" />
67      </div>
68
69      <div class="mb-3">
70        <label for="stockEdit" class="form-label text-dark">Stock:</label>
71        <input [(ngModel)]="productoEditado.stock" name="stock" id="stockEdit" type="number" class="form-control" placeholder="Cantidad de stock" required />
72      </div>
73
74      <div class="mb-3">
75        <label for="categoria_idEdit" class="form-label text-dark">Categoría:</label>
76        <select [(ngModel)]="productoEditado.categoria_id" name="categoria_id" id="categoria_idEdit" class="form-select" required>
77          <option value="" disabled selected>Selecciona una categoría</option>
78          <option *ngFor="let categoria of categorias" [value]="categoria.id">{{ categoria.nombre }}</option>
79        </select>
80      </div>
81
82      <button type="submit" class="btn btn-success">Actualizar Producto</button>
83      <button type="button" (click)="editando = false" class="btn btn-secondary ms-2">Cancelar</button>
84    </form>
85  </div>
86
87  <!-- Lista de productos -->
88  <h3 class="text-dark">Lista de Productos</h3>
89  <ul class="list-group">
90    <li *ngFor="let producto of productos" class="list-group-item d-flex justify-content-between align-items-center bg-light text-dark mb-2">
91      <span>{{ producto.nombre }} - {{ producto.precio | currency }} - {{ producto.stock }} unidades - Categoría: {{ producto.categoria }}</span>
92    </li>
93    <li>
94      <button (click)="editarProducto(producto)" class="btn btn-warning btn-sm">
95        <i class="bi bi-pencil"></i> Editar
96      </button>
97      <button (click)="eliminarProducto(producto.id)" class="btn btn-danger btn-sm ms-2">
98        <i class="bi bi-trash"></i> Eliminar
99      </button>
100    </li>
101  </ul>
102 </div>
103

```

Fig. 17 *gestion-productos.component.html*

Explicación: Este código en Angular gestiona la administración de productos, permitiendo crear, editar y eliminar productos. Usa ngIf para mostrar formularios de creación y edición con campos como nombre, descripción, precio, imágenes, stock y categoría. Al editar, se carga la información en los campos (productoEditado). La lista de productos (ngFor) muestra cada uno con su precio, stock y categoría, junto con botones para editar (editarProducto()) y eliminar (eliminarProducto()). Usa Bootstrap para el diseño y mejora la experiencia del usuario.


```

1 import { Component, OnInit } from '@angular/core';
2 import { ProductoService } from '.../services/producto.service';
3 import { AdminService } from '.../services/admin.service'; // Nueva importación
4 import { FormsModule } from '@angular/forms';
5 import { CommonModule } from '@angular/common';
6 import { Router } from '@angular/router';
7 import { Producto } from '.../models/producto';
8
9 @Component({
10   selector: 'app-gestion-productos',
11   standalone: true,
12   imports: [FormsModule, CommonModule],
13   templateUrl: './gestion-productos.component.html',
14   styleUrls: ['./gestion-productos.component.css'],
15 })
16 export class GestionProductosComponent implements OnInit {
17   productos: Producto[] = [];
18   categorias: any[] = []; // lista de categorias
19   nuevoProducto: Producto = new Producto('', '', 0, '', 0, '', null); // Añadimos categoria_id como null inicialmente
20   editando = false;
21   productoEditado: Producto = new Producto('', '', 0, '', 0, '', null);
22
23   constructor(
24     private productoService: ProductoService,
25     private adminService: AdminService, // Nueva inyección
26     private router: Router
27   ) {}
28
29   ngOnInit() {
30     if (localStorage.getItem('token')) {
31       this.router.navigate(['/login']);
32       return;
33     }
34     this.cargarProductos();
35     this.cargarCategorias();
36   }
37
38   cargarProductos() {
39     this.productoService.listarProductos(1, 10).subscribe({
40       next: (response) => {
41         this.productos = response;
42       },
43       error: (error) => {
44         console.error('error al cargar productos:', error);
45       },
46     });
47   }
48
49   cargarCategorias() {
50     this.adminService.listarCategorias().subscribe({
51       next: (response) => {
52         this.categorias = response;
53       },
54       error: (error) => {
55         console.error('error al cargar categorias:', error);
56         alert('error al cargar categorias');
57       },
58     });
59   }
60
61   crearProducto() {
62     console.log(this.nuevoProducto);
63     this.productoService.crearProducto(this.nuevoProducto).subscribe({
64       next: (response) => {
65         console.log('Producto creado:', response);
66         this.cargarProductos();
67         this.nuevoProducto = new Producto('', '', 0, '', 0, '', null);
68       },
69       error: (error) => {
70         console.error('error al crear producto:', error);
71         alert('error al crear producto');
72       },
73     });
74   }
75
76   editarProducto(producto: Producto) {
77     this.editando = true;
78     this.productoEditado = { ...producto };
79   }
80
81   actualizarProducto() {
82     this.productoService.actualizarProducto(this.productoEditado.id, this.productoEditado).subscribe({
83       next: (response) => {
84         console.log('Producto actualizado:', response);
85         this.cargarProductos();
86         this.editando = false;
87         this.productoEditado = new Producto('', '', 0, '', 0, '', null);
88       },
89       error: (error) => {
90         console.error('error al actualizar producto:', error);
91         alert('error al actualizar producto');
92       },
93     });
94   }
95
96   eliminarProducto(id: string) {
97     this.productoService.eliminarProducto(id).subscribe({
98       next: (response) => {
99         console.log('Producto eliminado:', response);
100         this.cargarProductos();
101       },
102       error: (error) => {
103         console.error('error al eliminar producto:', error);
104         alert('error al eliminar producto');
105       },
106     });
107   }
108 }

```

Fig. 18 *gestion-productos.component.ts*

Explicación: Este código en Angular define GestionProductosComponent, un componente para administrar productos. Al inicializar (ngOnInit), verifica si hay un token de sesión y luego carga la lista de productos (cargarProductos()) y categorías (cargarCategorias()). Permite crear (crearProducto()), editar (editarProducto()), actualizar (actualizarProducto()) y eliminar (eliminarProducto()) productos, usando ProductoService para comunicarse con el backend. Usa Router para gestionar la navegación y AdminService para obtener categorías.

- gestion-usuarios

```
1 body {
2   background-color: #333;
3 }
4
5 .table th, .table td {
6   color: white;
7 }
8
9 .btn-warning, .btn-danger {
10  margin-right: 5px;
11 }
```

Fig. 19 *gestion-usuarios.component.css*

Explicación: Este código CSS aplica un tema oscuro, estableciendo un fondo gris oscuro (#333) en el body. Las tablas (.table th, .table td) tienen texto blanco para mejorar la legibilidad. Además, los botones de advertencia (.btn-warning) y peligro (.btn-danger) tienen un margen derecho de 5px para separar visualmente los elementos en la interfaz.

```
1 <h2 class="text-center text-dark">Gestión de Usuarios</h2>
2
3 <div *ngIf="usuarios.length > 0; else sinUsuarios">
4   <table class="table table-dark table-striped">
5     <thead>
6       <tr>
7         <th>ID</th>
8         <th>Nombre</th>
9         <th>Correo</th>
10        <th>Rol</th>
11        <th>Estado</th>
12        <th>Acciones</th>
13      </tr>
14    </thead>
15    <tbody>
16      <tr *ngFor="let usuario of usuarios">
17        <td>{{ usuario.id }}</td>
18        <td>{{ usuario.nombre }}</td>
19        <td>{{ usuario.correo }}</td>
20        <td>{{ usuario.rol }}</td>
21        <td>{{ usuario.bloqueado ? 'Bloqueado' : 'Activo' }}</td>
22        <td>
23          <button class="btn btn-warning" (click)="bloquearUsuario(usuario.id, !usuario.bloqueado)">
24            {{ usuario.bloqueado ? 'Desbloquear' : 'Bloquear' }}
25          </button>
26          <button class="btn btn-danger" (click)="eliminarUsuario(usuario.id)">Eliminar</button>
27        </td>
28      </tr>
29    </tbody>
30  </table>
31 </div>
32
33 <ng-template #sinUsuarios>
34   <p class="text-light">No hay usuarios registrados.</p>
35 </ng-template>
```

Fig. 20 *gestion-usuarios.component.html*

Explicación: Este código en Angular gestiona la administración de usuarios, mostrando una tabla con ID, nombre, correo, rol y estado. Usa ngIf para verificar si hay usuarios y ngFor para iterar la lista. Cada usuario tiene botones para bloquear/desbloquear (bloquearUsuario()) y eliminar (eliminarUsuario()). Si no hay usuarios registrados, muestra un mensaje de alerta (ng-template). Se usa Bootstrap para mejorar el diseño con una tabla oscura (table-dark table-striped) y botones estilizados.

```

1 import { Component, OnInit } from '@angular/core';
2 import { AdminService } from '../services/admin.service';
3 import { CommonModule } from '@angular/common';
4
5 @Component({
6   selector: 'app-gestion-usuarios',
7   standalone: true,
8   imports: [CommonModule],
9   templateUrl: './gestion-usuarios.component.html',
10  styleUrls: ['./gestion-usuarios.component.css'],
11 })
12 export class GestionUsuariosComponent implements OnInit {
13   usuarios: any[] = [];
14
15   constructor(private adminService: AdminService) {}
16
17   ngOnInit() {
18     this.cargarUsuarios();
19   }
20
21   cargarUsuarios() {
22     this.adminService.listarUsuarios().subscribe({
23       next: (response) => {
24         this.usuarios = response;
25       },
26       error: (error) => {
27         console.error('Error al cargar usuarios:', error);
28         alert('Error: ${error.error?.mensaje} || 'No se pudo cargar los usuarios');
29       },
30     });
31   }
32
33   bloquearUsuario(usuarioId: string, bloquear: boolean) {
34     if (confirm('¿Estás seguro de que quieres ${bloquear ? 'bloquear' : 'desbloquear'} este usuario?')) {
35       this.adminService.bloquearUsuario(usuarioId, bloquear).subscribe({
36         next: (response) => {
37           console.log('Usuario actualizado:', response);
38           alert(response.mensaje);
39           this.cargarUsuarios();
40         },
41         error: (error) => {
42           console.error('Error al bloquear usuario:', error);
43           alert('Error: ${error.error?.mensaje} || 'No se pudo actualizar el usuario');
44         },
45       });
46     }
47   }
48
49   eliminarUsuario(usuarioId: string) {
50     if (confirm('¿Estás seguro de que quieres eliminar este usuario?')) {
51       this.adminService.eliminarUsuario(usuarioId).subscribe({
52         next: (response) => {
53           console.log('Usuario eliminado:', response);
54           alert('Usuario eliminado exitosamente');
55           this.cargarUsuarios();
56         },
57         error: (error) => {
58           console.error('Error al eliminar usuario:', error);
59           alert('Error: ${error.error?.mensaje} || 'No se pudo eliminar el usuario');
60         },
61       });
62     }
63   }
64 }

```

Fig. 21 *gestion-usuarios.component.ts*

Explicación: Este código en Angular define el `GestionUsuariosComponent`, un componente para gestionar usuarios. Al inicializar (`ngOnInit`), carga la lista de usuarios (`cargarUsuarios()`) desde `AdminService`. Permite bloquear o desbloquear (`bloquearUsuario()`) y eliminar (`eliminarUsuario()`) usuarios, mostrando confirmaciones antes de ejecutar las acciones. Cada operación se comunica con el backend y actualiza la lista tras completarse. Si ocurre un error, se muestra un mensaje en la consola y una alerta al usuario.

- historial-pedidos

```

1  body {
2    background-color: #333;
3  }
4
5  .pedido {
6    background-color: #444;
7    border-radius: 8px;
8  }
9
10 .pedido button {
11   margin-top: 10px;
12 }
13
14 /* Carrusel de imágenes dentro de los detalles del pedido */
15 .carousel-container img {
16   object-fit: cover; /* Asegura que las imágenes no se distorsionen */
17   width: 100%;      /* Ocupa todo el espacio disponible */
18   height: 200px;    /* Fija una altura para las imágenes */
19 }
20
21 /* Asegura que el carrusel tenga un margen para evitar que quede demasiado pegado al texto */
22 .carousel-container {
23   margin-top: 10px;
24   max-width: 200px; /* Ajusta el ancho máximo */
25 }
26
27 input::placeholder, textarea::placeholder{
28   color: #bbb; /* Color para el texto del placeholder */
29 }

```

Fig. 22 *historial-pedidos.component.css*

Explicación: Este CSS aplica un diseño oscuro, con fondo gris (#333) y tarjetas de pedidos (.pedido) en tono más claro (#444) con bordes redondeados. Estiliza un carrusel de imágenes, asegurando ajuste (object-fit: cover), ancho completo y altura fija (200px). Además, define márgenes en botones y placeholders en gris claro (#bbb) para mejor legibilidad.

```

1  <h2 class="text-center text-dark">Historial de Pedidos</h2>
2
3  <div *ngIf="pedidos.length > 0; else sinPedidos">
4    <div *ngFor="let pedido of pedidos" class="pedido mb-3 p-3 bg-dark text-light rounded">
5      <p><strong>ID del Pedido:</strong> {{ pedido.id }}</p>
6      <p><strong>Total:</strong> {{ pedido.total | currency }}</p>
7      <p><strong>Estado:</strong> {{ pedido.estado }}</p>
8      <p><strong>Fecha de Creación:</strong> {{ pedido.creado_en | date:'medium' }}</p>
9      <p *ngIf="pedido.transportista"><strong>Transportista:</strong> {{ pedido.transportista }}</p>
10     <p *ngIf="pedido.numero_seguimiento"><strong>Número de Seguimiento:</strong> {{ pedido.numero_seguimiento }}</p>
11
12     <div>
13       <h4>Detalles:</h4>
14       <ul>
15         <li *ngFor="let detalle of pedido.detalles">
16           <div>
17             <!-- Muestra el nombre del producto y cantidad -->
18             {{ detalle.nombre }} - Cantidad: {{ detalle.cantidad }} - Precio Unitario: {{ detalle.precio_unitario | currency }}
19           </div>
20
21           <!-- Carrusel de imágenes de productos -->
22           <!-- Carrusel de imágenes de productos -->
23           <div *ngIf="detalle.imagen" class="text-center"> <!-- Asegura que el contenido del carrusel esté centrado -->
24             <carousel>
25               <ng-container *ngFor="let imagen of detalle.imagen.split(',')">
26                 <slide>
27                   <img [src]="imagen" style="height: 400px; width: auto; object-fit: contain;" class="d-block w-50 mx-auto" alt="Imagen del producto">
28                 </slide>
29               </ng-container>
30             </carousel>
31           </div>
32         </li>
33       </ul>
34     </div>
35
36     <button *ngIf="pedido.estado === 'Pendiente'" class="btn btn-danger" (click)="cancelarPedido(pedido.id)">Cancelar Pedido</button>
37   </div>
38 </div>
39
40 <ng-template #sinPedidos>
41   <p class="text-light">No hay pedidos en tu historial.</p>
42 </ng-template>

```

Fig. 23 *historial-pedidos.component.html*

Explicación: Este código en Angular muestra el historial de pedidos, listando cada pedido con su ID, total, estado, fecha, transportista y número de seguimiento. Usa `*ngFor` para recorrer los pedidos y sus detalles, incluyendo nombre, cantidad y precio. Si el pedido tiene imágenes, las muestra en un carrusel (`ng-container`). Si el pedido está pendiente, se muestra un botón para cancelarlo (`cancelarPedido()`). Si no hay pedidos, se muestra un mensaje (`ng-template`). Usa Bootstrap para el diseño y organización.

```

1 import { Component, OnInit } from '@angular/core';
2 import { PedidoService } from '.../services/pedido.service';
3 import { CommonModule } from '@angular/common';
4 import { CarouselModule } from 'ngx-bootstrap/carousel';
5 import { ProductoService } from '.../services/producto.service';
6 import { Producto } from '.../models/producto';
7 @Component({
8   selector: 'app-historial-pedidos',
9   standalone: true,
10  imports: [CommonModule, CarouselModule],
11  templateUrl: './historial-pedidos.component.html',
12  styleUrls: ['./historial-pedidos.component.css'],
13 })
14 export class HistorialPedidosComponent implements OnInit {
15   pedidos: any[] = [];
16
17   constructor(
18     private pedidoService: PedidoService,
19     private productoService: ProductoService
20   ) {}
21
22   ngOnInit() {
23     this.cargarHistorial();
24   }
25
26   cargarHistorial() {
27     this.pedidoService.obtenerHistorial().subscribe({
28       next: (response) => {
29         this.pedidos = response;
30         this.pedidos.forEach((pedido) => {
31           pedido.detalles.forEach((detalle: any) => {
32             this.productoService.obtenerProductoPorId(detalle.producto_id).subscribe({
33               next: (producto: Producto) => {
34                 detalle.imagen = producto.imagenes || ''; // Asigna la imagen o cadena vacía si no hay
35               },
36               error: (error) => {
37                 console.error('Error al buscar producto:', error);
38                 detalle.imagen = '';
39               }
40             });
41           });
42         });
43         this.pedidos = this.pedidosFiltrados();
44       },
45       error: (error) => {
46         console.error('Error al cargar historial:', error);
47         alert('Error: ${error.error?.mensaje || 'No se pudo cargar el historial'}');
48       },
49     });
50   }
51
52   cancelarPedido(pedidoId: string) {
53     if (confirm('¿Estás seguro de que quieres cancelar este pedido?')) {
54       this.pedidoService.cancelarPedido(pedidoId).subscribe({
55         next: (response) => {
56           console.log('Pedido cancelado:', response);
57           alert('Pedido cancelado exitosamente');
58           this.cargarHistorial(); // Refrescar el historial
59         },
60         error: (error) => {
61           console.error('Error al cancelar pedido:', error);
62           alert('Error: ${error.error?.mensaje || 'No se pudo cancelar el pedido'}');
63         },
64       });
65     }
66   }
67
68   pedidosFiltrados() {
69     return this.pedidos ? this.pedidos.filter(p => p.estado !== 'Cancelado') : [];
70   }
71 }

```

Fig. 24 *historial-pedidos.component.ts*

Explicación: Este Angular Component gestiona el historial de pedidos, cargándolos desde `PedidoService` y obteniendo imágenes desde `ProductoService`. Permite cancelar pedidos (`cancelarPedido()`), recargando la lista tras la acción. La función `pedidosFiltrados()` oculta pedidos cancelados. Usa `CarouselModule` para mostrar imágenes en carrusel y maneja errores con alertas.

- lista-productos

```
1  body {
2    background-color: #2c2f38; /* Fondo oscuro */
3  }
4
5  .form-control {
6    background-color: #444; /* Fondo oscuro para los inputs */
7    color: white; /* Texto claro dentro de los inputs */
8    border: 1px solid #555; /* Borde gris oscuro */
9  }
10
11 .form-control:focus {
12   background-color: #555; /* Fondo más oscuro cuando el input está enfocado */
13   border-color: #007bff; /* Borde azul al enfocar */
14 }
15
16 .btn-success {
17   background-color: #28a745; /* Verde para el botón */
18   border: none;
19 }
20
21 .btn-success:hover {
22   background-color: #218838; /* Verde más oscuro al pasar el mouse */
23 }
24
25 input::placeholder, textarea::placeholder{
26   color: #bbb; /* Color para el texto del placeholder */
27 }
28
29 /* Aseguramos que la lista de productos tenga un buen espaciado y bordes suaves */
30 .list-group-item {
31   border-radius: 10px;
32   border: 1px solid #444; /* Borde de color suave para cada item */
33   padding: 15px;
34 }
35
36 /* Estilo para el contenedor de las imágenes */
37 .carousel-container {
38   max-width: 200px;
39   height: 150px;
40   overflow: hidden;
41 }
42
43 /* Estilo para las imágenes dentro del carrusel */
44 .carousel-container img {
45   object-fit: cover;
46   width: 100%;
47   height: 100%;
48 }
49
50 /* Estilos para el contenedor del producto, asegurando que todo esté alineado horizontalmente */
51 .d-flex {
52   display: flex;
53   justify-content: space-between;
54   align-items: center;
55 }
```

Fig. 25 *lista-productos.component.css*

Explicación: Este código CSS aplica un tema oscuro con fondo gris (#2c2f38). Los inputs (.form-control) tienen fondo oscuro (#444), texto blanco y borde gris (#555), resaltándose en azul (#007bff) al enfocarse. Los botones (.btn-success) son verdes (#28a745), oscureciéndose al pasar el mouse. Los placeholders son gris claro (#bbb). Se estiliza la lista de productos (.list-group-item) con bordes redondeados y espaciado. El carrusel (.carousel-container) limita el tamaño de las imágenes (max-width: 200px) y mantiene su proporción (object-fit: cover). Usa flexbox (.d-flex) para alinear elementos horizontalmente.

```

1 <h2 class="text-center text-dark">Lista de Productos</h2>
2
3 <form (ngSubmit)="buscarProductos()" class="mb-4">
4   <div class="form-row align-items-center">
5     <!-- Nombre del producto -->
6     <div class="col-md-2 mb-3">
7       <label for="nombre" class="text-dark">Nombre del producto</label>
8       <input id="nombre" [(ngModel)]="filtros.nombre" name="nombre" class="form-control" placeholder="Nombre del producto" />
9     </div>
10
11     <!-- Categoría -->
12     <div class="col-md-2 mb-3">
13       <label for="categoria_id" class="text-dark">Categoría</label>
14       <select id="categoria_id" [(ngModel)]="filtros.categoria_id" name="categoria_id" class="form-control">
15         <option value="">Todas las categorías</option>
16         <option *ngFor="let categoria of categorias" [value]="categoria.id">{{ categoria.nombre }}</option>
17       </select>
18     </div>
19
20     <!-- Precio mínimo -->
21     <div class="col-md-2 mb-3">
22       <label for="precioMin" class="text-dark">Precio mínimo</label>
23       <input id="precioMin" [(ngModel)]="filtros.precioMin" name="precioMin" type="number" class="form-control" placeholder="Precio mínimo" />
24     </div>
25
26     <!-- Precio máximo -->
27     <div class="col-md-2 mb-3">
28       <label for="precioMax" class="text-dark">Precio máximo</label>
29       <input id="precioMax" [(ngModel)]="filtros.precioMax" name="precioMax" type="number" class="form-control" placeholder="Precio máximo" />
30     </div>
31
32     <!-- Ordenar por -->
33     <div class="col-md-2 mb-3">
34       <label for="ordenarPor" class="text-dark">Ordenar por</label>
35       <select id="ordenarPor" [(ngModel)]="filtros.ordenarPor" name="ordenarPor" class="form-control">
36         <option value="precio">Precio</option>
37         <option value="nombre">Nombre</option>
38         <option value="creado_en">Fecha de Creación</option>
39       </select>
40     </div>

```

Fig. 26 *lista-productos.component.html*

Explicación: Este código en Angular implementa un formulario de filtros para buscar productos. Permite ingresar el nombre del producto, seleccionar una categoría, establecer un rango de precios (precioMin y precioMax) y elegir un criterio de ordenación (precio, nombre o fecha de creación). Usa [(ngModel)] para vincular los valores de entrada con el objeto filtros. Cuando el formulario se envía (ngSubmit="buscarProductos()"), los filtros se aplican para realizar la búsqueda. Usa Bootstrap para el diseño y alineación de los elementos.

```

1 import { Component, OnInit } from '@angular/core';
2 import { ProductoService } from '../../services/producto.service';
3 import { AdminService } from '../../services/admin.service'; // Nueva importación
4 import { FormsModule } from '@angular/forms';
5 import { CommonModule } from '@angular/common';
6 import { Router } from '@angular/router';
7 import { Producto } from '../../models/producto';
8 import { CarouselModule } from 'ngx-bootstrap/carousel';
9
10 @Component({
11   selector: 'app-lista-productos',
12   standalone: true,
13   imports: [FormsModule, CommonModule, CarouselModule],
14   templateUrl: './lista-productos.component.html',
15   styleUrls: ['./lista-productos.component.css'],
16 })
17 export class ListaProductosComponent implements OnInit {
18   productos: Producto[] = [];
19   categorias: any[] = []; // Lista de categorías
20   filtros = {
21     categoria_id: '', // Cambiamos a categoria_id
22     precioMin: '',
23     precioMax: '',
24     nombre: '',
25     ordenarPor: 'precio',
26     orden: 'ASC',
27   };
28
29   constructor(
30     private productoService: ProductoService,
31     private adminService: AdminService, // Nueva inyección
32     private router: Router
33   ) {}
34
35   ngOnInit() {
36     if (!localStorage.getItem('token')) {
37       this.router.navigate(['/login']);
38       return;
39     }
40     this.cargarCategorias();
41     this.buscarProductos();
42   }

```

Fig. 27 *lista-productos.component.ts*

Explicación: Este código en Angular define `ListaProductosComponent`, encargado de mostrar productos con filtros de búsqueda. Al iniciar (`ngOnInit()`), verifica si hay un token de sesión, redirigiendo a `/login` si no existe. Luego, carga las categorías (`cargarCategorias()`) y busca productos (`buscarProductos()`). Los filtros incluyen categoría, nombre, precio mínimo/máximo y ordenación (precio, nombre o fecha). Usa `ProductoService` para obtener productos, `AdminService` para categorías y `Router` para la navegación.

- login


```

1  form {
2    max-width: 400px;
3    margin: 0 auto;
4    padding: 20px;
5    background-color: #343a40; /* Fondo oscuro */
6    border-radius: 5px;
7    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
8  }
9
10  h2 {
11    color: #000000;
12    text-align: center;
13  }
14
15  button {
16    margin-top: 10px;
17  }
18
19  a {
20    display: block;
21    text-align: center;
22    color: #ffffff;
23    margin-top: 15px;
24  }

```

Fig. 28 *login.component.css*

Explicación: Este código CSS estiliza un formulario con un ancho de 400px, fondo oscuro y bordes redondeados, centrándolo en la pantalla con `margin: 0 auto`. También define estilos para el título (`h2`) con texto negro y centrado, el botón con margen superior de 10px, y los enlaces (`a`) con texto blanco, centrado y margen inferior de 15px.

```

1  <h2>Iniciar Sesión</h2>
2
3  <form (ngSubmit)="iniciarSesion()" class="mb-4">
4    <div class="form-group">
5      <label for="correo" class="text-light">Correo</label>
6      <input id="correo" [(ngModel)]="credenciales.correo" name="correo" type="email" class="form-control" placeholder="Correo" required />
7    </div>
8
9    <div class="form-group">
10     <label for="contrasena" class="text-light">Contraseña</label>
11     <input id="contrasena" [(ngModel)]="credenciales.contrasena" name="contrasena" type="password" class="form-control" placeholder="Contraseña" required />
12   </div>
13
14   <button type="submit" class="btn btn-primary w-100">Iniciar Sesión</button>
15 </form>
16
17 <a routerLink="/registro" class="text-dark">¿No tienes una cuenta? Regístrate</a>
18
19

```

Fig. 29 *login.component.html*

Explicación: Este código crea un formulario de inicio de sesión en Angular. Captura el correo y la contraseña del usuario mediante `ngModel` y los envía al método `iniciarSesion()` cuando se envía el formulario. Además, incluye un botón de envío y un enlace para registrarse si el usuario no tiene cuenta.

```

1 import { Component } from '@angular/core';
2 import { FormsModule } from '@angular/forms';
3 import { Router } from '@angular/router';
4 import { AuthService } from '../../services/auth.service';
5 import { RouterLink } from '@angular/router';
6
7 @Component({
8   selector: 'app-login',
9   standalone: true,
10  imports: [FormsModule, RouterLink],
11  templateUrl: './login.component.html',
12  styleUrls: ['./login.component.css'],
13 })
14 export class LoginComponent {
15   credenciales = {
16     correo: '',
17     contrasena: '',
18   };
19
20   constructor(private authService: AuthService, private router: Router) {}
21
22   iniciarSesion() {
23     console.log("Enviando credenciales:", this.credenciales);
24
25     this.authService.iniciarSesion(this.credenciales).subscribe({
26       next: (response) => {
27         console.log("Inicio de sesión exitoso:", response);
28         localStorage.setItem("token", response.token);
29         // Decodificar el token para obtener el rol (usamos una librería como jwt-decode o parseamos manualmente)
30         const tokenPayload = JSON.parse(atob(response.token.split('.')[1]));
31         localStorage.setItem("rol", tokenPayload.rol); // Guardar el rol en localStorage
32         this.router.navigate(["/lista-productos"]);
33       },
34       error: (error) => {
35         console.error("Error en la solicitud:", error);
36         alert(`Error: ${error.error?.mensaje || "Error desconocido"}`);
37       },
38     });
39   }
40 }

```

Fig. 30 *login.component.ts*

Explicación: Este código define un componente de inicio de sesión en Angular. Captura las credenciales del usuario y las envía al servicio de autenticación (AuthService). Si el inicio de sesión es exitoso, almacena el token en localStorage, extrae y guarda el rol del usuario y redirige a la página de lista de productos. En caso de error, muestra un mensaje de alerta y registra el error en la consola.

- perfil

```

1  .perfil-container {
2    background-color: #343a40; /* Fondo oscuro */
3    color: #fff; /* Texto blanco */
4    padding: 20px;
5    border-radius: 10px;
6    max-width: 500px;
7    margin: 0 auto;
8    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
9  }
10
11  h2 {
12    text-align: center;
13    color: #000000;
14  }
15
16  .perfil-info p {
17    margin: 10px 0;
18  }
19
20  .perfil-actions {
21    text-align: center;
22    margin-top: 20px;
23  }
24
25  .perfil-actions a {
26    color: #007bff;
27  }
28
29  .perfil-actions a:hover {
30    text-decoration: underline;
31  }

```

Fig. 31 *perfil.component.css*

Explicación: Este código CSS estiliza un contenedor de perfil con un fondo oscuro, texto blanco y bordes redondeados. Centra el título <h2> y ajusta los márgenes de la información del usuario. También define la sección de acciones, alineando su contenido y estilizando los enlaces para que sean azules y se subrayen al pasar el mouse.

```

1  <h2>Perfil del Usuario</h2>
2
3  <div *ngIf="usuario; else cargando" class="perfil-container">
4    <div class="perfil-info">
5      <p><strong>Nombre:</strong> {{ usuario.nombre }}</p>
6      <p><strong>Correo:</strong> {{ usuario.correo }}</p>
7      <p><strong>Dirección:</strong> {{ usuario.direccion }}</p>
8      <p><strong>Rol:</strong> {{ usuario.rol }}</p>
9    </div>
10
11    <div class="perfil-actions">
12      <a routerLink="/actualizar-contrasena" class="btn btn-link">Cambiar Contraseña</a>
13    </div>
14  </div>
15
16  <ng-template #cargando>
17    <p>Cargando información del usuario...</p>
18  </ng-template>
19

```

Fig. 32 *perfil.component.html*

Explicación: Este código en Angular muestra el perfil del usuario autenticado. Usa *ngIf para verificar si usuario está disponible; si no, muestra un mensaje de carga. Se presentan el nombre, correo, dirección y rol del usuario en un contenedor. Además, incluye un enlace para cambiar la contraseña usando routerLink.

```
1 import { Component, OnInit } from '@angular/core';
2 import { AuthService } from '../../services/auth.service';
3 import { CommonModule } from '@angular/common';
4 import { RouterLink, Router } from '@angular/router';
5 import { Usuario } from '../../models/usuario';
6
7 @Component({
8   selector: 'app-perfil',
9   standalone: true,
10  imports: [CommonModule, RouterLink],
11  templateUrl: './perfil.component.html',
12  styleUrls: ['./perfil.component.css'],
13 })
14 export class PerfilComponent implements OnInit {
15   usuario: Usuario | null = null;
16
17   constructor(private authService: AuthService, private router: Router) {}
18
19   ngOnInit() {
20     if (!localStorage.getItem('token')) {
21       this.router.navigate(['/login']);
22       return;
23     }
24     this.cargarUsuario();
25   }
26
27   cargarUsuario() {
28     this.authService.obtenerUsuarioAutenticado().subscribe({
29       next: (response) => {
30         this.usuario = new Usuario(response.id, response.nombre, response.correo, response.direccion, response.rol);
31       },
32       error: (error) => {
33         console.error('Error al cargar el usuario:', error);
34         alert('Error: ${error.error?.mensaje || 'No se pudo cargar la información del usuario'}');
35         this.router.navigate(['/login']);
36       },
37     });
38   }
39 }
40
```

Fig. 33 *perfil.component.ts*

Explicación: Este código en Angular define un componente PerfilComponent que maneja la información del usuario autenticado. Al iniciar (ngOnInit), verifica si hay un token en localStorage; si no, redirige al login. Luego, cargarUsuario() obtiene los datos del usuario desde AuthService, asignándolos al objeto usuario. Si hay un error, lo muestra en consola, alerta al usuario y lo redirige al login.

- registro

```
1  .registro-form {
2      background-color: #343a40; /* Fondo oscuro */
3      color: #fff; /* Texto blanco */
4      padding: 20px;
5      border-radius: 10px;
6      max-width: 600px;
7      margin: 0 auto;
8      box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
9  }
10
11  h2 {
12      text-align: center;
13      color: #000000;
14  }
15
16  .form-row {
17      margin-bottom: 15px;
18  }
19
20  .form-row .col-md-6,
21  .form-row .col-md-12 {
22      margin-bottom: 10px;
23  }
```

Fig. 34 *registro.component.css*

Explicación: Este código CSS da estilo a un formulario de registro (.registro-form), estableciendo un fondo oscuro, texto blanco, bordes redondeados, sombra y un ancho máximo de 600px con margen centrado. El título (h2) se alinea al centro con color negro. Además, .form-row define márgenes inferiores para los elementos de formulario, ajustando la separación entre los campos (col-md-6 y col-md-12).

```

1 <h2>Registro de Usuario</h2>
2
3 <form (ngSubmit)="registrar()" class="registro-form">
4   <div class="form-row">
5     <div class="col-md-6">
6       <input [(ngModel)]="usuario.nombre" name="nombre" placeholder="Nombre"
7         required class="form-control" />
8       <small class="text-danger" *ngIf="errores.nombre">{{errores.nombre}}</small>
9     </div>
10
11     <div class="col-md-6">
12       <input [(ngModel)]="usuario.correo" type="email" name="correo"
13         placeholder="Correo" required class="form-control" />
14       <small class="text-danger" *ngIf="errores.correo">{{errores.correo}}</small>
15     </div>
16
17     <div class="col-md-6">
18       <input [(ngModel)]="usuario.contrasena" name="contrasena" type="password"
19         placeholder="Contraseña" required class="form-control" />
20       <small class="text-danger" *ngIf="errores.contrasena">{{errores.contrasena}}</small>
21     </div>
22
23     <div class="col-md-6">
24       <input [(ngModel)]="usuario.direccion" name="direccion"
25         placeholder="Dirección" required class="form-control" />
26       <small class="text-danger" *ngIf="errores.direccion">{{errores.direccion}}</small>
27     </div>
28
29     <div class="col-md-12">
30       <button type="submit" class="btn btn-primary w-100">Registrar</button>
31     </div>
32   </div>
33 </form>
34
35 <div class="form-link">
36   <a routerLink="/login">¿Ya tienes una cuenta? Inicia sesión</a>
37 </div>

```

Fig. 35 *registro.component.html*

Explicación: Este código en Angular crea un formulario de registro de usuario con validación de campos. Usa [(ngModel)] para enlazar los datos del usuario a los inputs de nombre, correo, contraseña y dirección. Si hay errores, se muestran mensajes de validación (ngIf="errores.campo"). El formulario se envía con registrar(), y un botón de envío (type="submit") ejecuta la acción. Al final, hay un enlace (routerLink="/login") para redirigir a la página de inicio de sesión si el usuario ya tiene cuenta.

```

1 import { Component } from '@angular/core';
2 import { FormsModule } from '@angular/forms';
3 import { AuthService } from '../../services/auth.service';
4 import { Router } from '@angular/router';
5
6 @Component({
7   selector: 'app-registro',
8   standalone: true,
9   imports: [FormsModule],
10  templateUrl: './registro.component.html',
11  styleUrls: ['./registro.component.css'],
12 })
13 export class RegistroComponent {
14   usuario = {
15     nombre: '',
16     correo: '',
17     contrasena: '',
18     direccion: '',
19     rol: 'cliente',
20   };
21
22   // Objeto para manejar errores
23   errores = {
24     nombre: '',
25     correo: '',
26     contrasena: '',
27     direccion: ''
28   };
29
30   constructor(private authService: AuthService, private router: Router) {}
31
32   // Función para validar email
33   validarEmail(email: string): boolean {
34     const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
35     return emailRegex.test(email);
36   }

```

Fig. 36 *registro.component.ts*

Explicación: Este código en Angular define RegistroComponent, un formulario de registro de usuarios. Contiene un objeto usuario con los campos necesarios y un objeto errores para validar datos. La función validarFormulario() revisa que nombre, correo, contraseña y dirección sean válidos antes de enviar. Usa validarEmail() para verificar correos con una expresión regular. Si la validación es exitosa, registrar() envía los datos a AuthService, mostrando una alerta y redirigiendo a /login en caso de éxito, o manejando errores si fallan.

- **models**

```

1  export class Producto {
2    constructor(
3      public id: string,
4      public nombre: string,
5      public descripcion: string,
6      public precio: number,
7      public imagenes: string,
8      public stock: number,
9      public categoria: string, // Nombre de la categoría (devuelto por el backend)
10     public categoria_id: string | null = null, // ID de la categoría para creación/actualización
11     public creado_en?: string
12   ) {}
13 }

```

Fig. 37 *producto.ts*

Explicación: Este código define una clase `Producto` en TypeScript para modelar productos en una aplicación. El constructor recibe propiedades como `id`, `nombre`, `descripcion`, `precio`, `imagenes`, `stock`, y `creado_en` para representar la información de un producto. También maneja `categoria` (nombre de la categoría) y `categoria_id` (su identificador, con valor `null` por defecto). Se usa en aplicaciones Angular para gestionar productos en un sistema de inventario o tienda en línea.

```

1  export class Usuario {
2    constructor(
3      public id: string,
4      public nombre: string,
5      public correo: string,
6      public direccion: string,
7      public rol: string
8    ) {}
9  }

```

Fig. 38 *usuario.ts*

Explicación: Este código en TypeScript define la clase `Usuario`, utilizada para modelar la información de un usuario en una aplicación. El constructor recibe cinco propiedades: `id` (identificador único), `nombre`, `correo`, `direccion` y `rol` (tipo de usuario). Se usa en Angular para manejar datos de usuarios dentro del sistema, como autenticación o gestión de perfiles.

- **services**

```

1 import { Injectable } from '@angular/core';
2 import { HttpClient, HttpHeaders } from '@angular/common/http';
3 import { Observable } from 'rxjs';
4
5 @Injectable({
6   providedIn: 'root',
7 })
8 export class AdminService {
9   private apiUrl = 'http://localhost:3000/api/admin';
10
11   constructor(private http: HttpClient) {}
12
13   private getHeaders(): HttpHeaders {
14     const token = localStorage.getItem('token');
15     return new HttpHeaders({
16       Authorization: 'Bearer ${token}',
17     });
18   }
19
20   // Obtener estadísticas del dashboard
21   obtenerEstadisticas(): Observable<any> {
22     return this.http.get(`${this.apiUrl}/estadisticas`, { headers: this.getHeaders() });
23   }
24
25   // Listar todos los usuarios
26   listarUsuarios(): Observable<any[]> {
27     return this.http.get<any[]>(`${this.apiUrl}/usuarios`, { headers: this.getHeaders() });
28   }
29
30   // Bloquear o desbloquear un usuario
31   bloquearUsuario(usuarioId: string, bloquear: boolean): Observable<any> {
32     return this.http.put(`${this.apiUrl}/usuarios/bloquear/${usuarioId}`, { bloquear }, { headers: this.getHeaders() });
33   }
34
35   // Eliminar un usuario
36   eliminarUsuario(usuarioId: string): Observable<any> {
37     return this.http.delete(`${this.apiUrl}/usuarios/eliminar/${usuarioId}`, { headers: this.getHeaders() });
38   }
39
40   // Crear una categoría
41   crearCategoria(nombre: string, descripcion: string): Observable<any> {
42     return this.http.post(`${this.apiUrl}/categorias/crear`, { nombre, descripcion }, { headers: this.getHeaders() });
43   }
44
45   // Actualizar una categoría
46   actualizarCategoria(categoriaId: string, nombre: string, descripcion: string): Observable<any> {
47     return this.http.put(`${this.apiUrl}/categorias/actualizar/${categoriaId}`, { nombre, descripcion }, { headers: this.getHeaders() });
48   }
49
50   // Eliminar una categoría
51   eliminarCategoria(categoriaId: string): Observable<any> {
52     return this.http.delete(`${this.apiUrl}/categorias/eliminar/${categoriaId}`, { headers: this.getHeaders() });
53   }
54
55   // Listar todas las categorías (público)
56   listarCategorias(): Observable<any[]> {
57     return this.http.get<any[]>(`${this.apiUrl}/categorias`);
58   }
59 }

```

Fig. 39 *admin.service.ts*

Explicación: Este código en TypeScript define un servicio de administración en Angular que interactúa con una API REST mediante HttpClient. Utiliza HttpHeaders para incluir un token de autenticación en las solicitudes. Sus principales funciones incluyen obtener estadísticas del sistema, listar y gestionar usuarios (bloquear, desbloquear y eliminar), así como administrar categorías de productos (crear, actualizar y eliminar). También permite listar todas las categorías registradas. Todas las peticiones se manejan de manera asíncrona utilizando Observable<any>.


```

1  import { Injectable } from '@angular/core';
2  import { HttpClient } from '@angular/common/http';
3  import { Observable, BehaviorSubject } from 'rxjs';
4  import { tap } from 'rxjs/operators';
5  import { Usuario } from '../models/usuario';
6
7  @Injectable({
8    providedIn: 'root',
9  })
10 export class AuthService {
11   private apiUrl = 'http://localhost:3000/api'; // URL del backend
12   private usuarioSubject = new BehaviorSubject<Usuario | null>(null);
13
14   constructor(private http: HttpClient) {}
15
16   // Obtener el usuario actual como observable
17   get usuario$(): Observable<Usuario | null> {
18     return this.usuarioSubject.asObservable();
19   }
20
21   // Obtener el usuario actual como valor
22   getUsuario(): Usuario | null {
23     return this.usuarioSubject.value;
24   }
25
26   // Verificar si el usuario es admin
27   esAdmin(): boolean {
28     const usuario = this.getUsuario();
29     return usuario?.rol === 'admin';
30   }
31
32   // Registrar un usuario
33   registrarUsuario(usuario: any): Observable<any> {
34     return this.http.post(`${this.apiUrl}/auth/registrar`, usuario);
35   }

```

Fig. 40 *auth.service.ts*

Explicación: Este código en TypeScript define un servicio de autenticación en Angular utilizando HttpClient para comunicarse con una API en `http://localhost:3000/api/`. Implementa la gestión de usuarios autenticados mediante un BehaviorSubject, lo que permite compartir la información del usuario entre componentes de la aplicación. Entre sus funcionalidades se incluyen la obtención del usuario autenticado, la verificación de si es administrador (`esAdmin()`), el registro de usuarios (`registrarUsuario()`), el inicio de sesión (`iniciarSesion()`), donde se almacena un token JWT en localStorage y se extrae su información, y el cierre de sesión (`cerrarSesion()`). También permite actualizar datos del usuario (`actualizarUsuario()`), cambiar su contraseña (`actualizarContraseña()`) y obtener información del usuario autenticado (`obtenerUsuarioAutenticado()`). Se usan tokens JWT para la autenticación y autorización en las solicitudes HTTP.

```

1 import { Injectable } from '@angular/core';
2 import { HttpClient, HttpHeaders } from '@angular/common/http';
3 import { Observable } from 'rxjs';
4
5 @Injectable({
6   providedIn: 'root',
7 })
8 export class CarritoService {
9   private apiUrl = 'http://localhost:3000/api/carrito';
10
11   constructor(private http: HttpClient) {}
12
13   private getHeaders(): HttpHeaders {
14     const token = localStorage.getItem('token');
15     return new HttpHeaders({
16       Authorization: `Bearer ${token}`,
17     });
18   }
19
20   // Actualizar la cantidad de un producto en el carrito
21   actualizarCantidad(productoId: string, cantidad: number): Observable<any> {
22     return this.http.put(
23       `${this.apiUrl}/actualizar`,
24       { productoId, cantidad },
25       { headers: this.getHeaders() }
26     );
27   }
28
29   // Eliminar un producto del carrito
30   eliminarProducto(productoId: string): Observable<any> {
31     return this.http.delete(`${this.apiUrl}/eliminar/${productoId}`, { headers: this.getHeaders() });
32   }
33
34   // Obtener el carrito del usuario
35   obtenerCarrito(): Observable<any[]> {
36     return this.http.get<any[]>(`${this.apiUrl}`, { headers: this.getHeaders() });
37   }
38 }

```

Fig. 41 *carrito.service.ts*

Explicación: Este código define un servicio CarritoService en Angular para gestionar las operaciones del carrito de compras mediante peticiones HTTP. Utiliza HttpClient para comunicarse con la API ubicada en `http://localhost:3000/api/carrito`. Se incluyen métodos para actualizar la cantidad de un producto (`actualizarCantidad`), eliminar un producto (`eliminarProducto`) y obtener el carrito (`obtenerCarrito`). Todas las peticiones incluyen un encabezado con un token de autenticación almacenado en `localStorage`. El servicio está disponible globalmente en la aplicación gracias a `@Injectable({ providedIn: 'root' })`.


```

1  import { Injectable } from '@angular/core';
2  import { HttpClient, HttpHeaders } from '@angular/common/http';
3  import { Observable } from 'rxjs';
4
5  @Injectable({
6    providedIn: 'root',
7  })
8  export class PedidoService {
9    private apiUrl = 'http://localhost:3000/api/pedidos';
10
11    constructor(private http: HttpClient) {}
12
13    private getHeaders(): HttpHeaders {
14      const token = localStorage.getItem('token');
15      return new HttpHeaders({
16        Authorization: `Bearer ${token}`,
17      });
18    }
19
20    // Crear un pedido a partir del carrito
21    crearPedido(): Observable<any> {
22      return this.http.post(`${this.apiUrl}/crear`, {}, { headers: this.getHeaders() });
23    }
24
25    // Obtener el historial de pedidos del usuario
26    obtenerHistorial(): Observable<any[]> {
27      return this.http.get<any[]>(`${this.apiUrl}/historial`, { headers: this.getHeaders() });
28    }
29
30    // Cancelar un pedido
31    cancelarPedido(pedidoId: string): Observable<any> {
32      return this.http.put(`${this.apiUrl}/cancelar/${pedidoId}`, {}, { headers: this.getHeaders() });
33    }

```

Fig. 42 *pedido.service.ts*

Explicación: Este código define el servicio PedidoService en Angular para gestionar pedidos a través de peticiones HTTP. Se comunica con la API en `http://localhost:3000/api/pedidos` y usa `HttpClient` para las solicitudes. Incluye métodos para crear un pedido (`crearPedido`), obtener el historial de pedidos (`obtenerHistorial`) y cancelar un pedido (`cancelarPedido`). Cada petición incluye un encabezado con un token de autenticación extraído del `localStorage`. El servicio es inyectable globalmente gracias a `@Injectable({ providedIn: 'root' })`.

```

1 import { Injectable } from '@angular/core';
2 import { HttpClient, HttpHeaders } from '@angular/common/http';
3 import { Observable } from 'rxjs';
4 import { Producto } from '../models/producto';
5
6 @Injectable({
7   providedIn: 'root',
8 })
9 export class ProductoService {
10   private apiUrl = 'http://localhost:3000/api/productos';
11   private carritoUrl = 'http://localhost:3000/api/carrito'; // Nueva URL para el carrito
12
13   constructor(private http: HttpClient) {}
14
15   private getHeaders(): HttpHeaders {
16     const token = localStorage.getItem('token');
17     return new HttpHeaders({
18       Authorization: `Bearer ${token}`,
19     });
20   }
21
22   crearProducto(producto: Producto): Observable<any> {
23     return this.http.post(this.apiUrl, producto, { headers: this.getHeaders() });
24   }
25
26   actualizarProducto(id: string, producto: Producto): Observable<any> {
27     return this.http.put(`${this.apiUrl}/${id}`, producto, { headers: this.getHeaders() });
28   }
29
30   eliminarProducto(id: string): Observable<any> {
31     return this.http.delete(`${this.apiUrl}/${id}`, { headers: this.getHeaders() });
32   }
33
34   listarProductos(pagina: number, limite: number): Observable<Producto[]> {
35     return this.http.get<Producto[]>(`${this.apiUrl}?pagina=${pagina}&limite=${limite}`);
36   }
37
38   buscarProductos(filtros: any): Observable<Producto[]> {
39     let params = '';
40     for (const key in filtros) {
41       if (filtros[key]) {
42         params += `${key}=${encodeURIComponent(filtros[key])}&`;
43       }
44     }
45     return this.http.get<Producto[]>(`${this.apiUrl}/buscar?${params}`);
46   }
47
48   agregarAlCarrito(productoId: string, cantidad: number): Observable<any> {
49     return this.http.post(
50       `${this.carritoUrl}/agregar`,
51       { productoId, cantidad },
52       { headers: this.getHeaders() }
53     );
54   }
55
56   obtenerProductoPorId(id: string): Observable<Producto> {
57     return this.http.get<Producto>(`${this.apiUrl}/${id}`);
58   }
59 }
60

```

Fig. 43 *producto.service.ts*

Explicación: El código define `ProductoService`, un servicio de Angular que maneja las operaciones relacionadas con productos y el carrito de compras a través de peticiones HTTP. Se comunica con las APIs en `http://localhost:3000/api/productos` y `http://localhost:3000/api/carrito`. Implementa métodos para crear (`crearProducto`), actualizar (`actualizarProducto`), eliminar (`eliminarProducto`) y listar productos (`listarProductos`). También permite buscar productos mediante filtros (`buscarProductos`), agregar productos al carrito (`agregarAlCarrito`) y obtener un producto por su ID (`obtenerProductoPorId`). Las solicitudes incluyen un token de autenticación almacenado en `localStorage`.

- Tienda virtual backend
- config

```

1  const mysql = require('mysql2');
2
3  const pool = mysql.createPool({
4    host: process.env.DB_HOST,
5    user: process.env.DB_USER || 'root', // Fallback por si no carga .env
6    password: process.env.DB_PASSWORD || '', // Fallback
7    database: process.env.DB_NAME || 'tienda_virtual',
8    waitForConnections: true,
9    connectionLimit: 10,
10   queueLimit: 0
11 });
12
13 module.exports = pool.promise();

```

Fig. 44 db.js

Explicación: El código configura una conexión MySQL en Node.js con mysql2, usando un pool de conexiones. Los parámetros (host, usuario, contraseña, BD) se obtienen de process.env, con valores por defecto (root, "", tienda_virtual). Se establecen connectionLimit: 10 y queueLimit: 0. El pool se exporta con promise() para usar promesas en las consultas.

- controladores

```

1  const db = require('../config/db'); // Importar la conexión a la base de datos
2  const Pedido = require('../modelos/Pedido');
3  const Producto = require('../modelos/Producto');
4  const Usuario = require('../modelos/Usuario');
5  const Categoria = require('../modelos/Categoria');
6
7  const obtenerEstadisticas = async (req, res) => {
8    if (req.usuario.rol !== 'admin') {
9      return res.status(403).json({ mensaje: 'No autorizado' });
10   }
11   try {
12     const [ventas] = await db.execute('SELECT SUM(total) AS total_ventas FROM pedidos WHERE estado = "Entregado"');
13     const totalVentas = ventas[0].total_ventas || 0;
14     const productosMasVendidos = await Producto.productosMasVendidos();
15     const [usuarios] = await db.execute('SELECT COUNT(*) AS total_usuarios FROM usuarios');
16     const totalUsuarios = usuarios[0].total_usuarios;
17
18     res.json({
19       totalVentas,
20       productosMasVendidos,
21       totalUsuarios,
22     });
23   } catch (error) {
24     res.status(500).json({ mensaje: 'Error al obtener estadísticas', error: error.message });
25   }
26 };
27
28 const listarUsuarios = async (req, res) => {
29   if (req.usuario.rol !== 'admin') {
30     return res.status(403).json({ mensaje: 'No autorizado' });
31   }
32   try {
33     const usuarios = await Usuario.listar();
34     res.json(usuarios);
35   } catch (error) {
36     res.status(500).json({ mensaje: 'Error al listar usuarios', error: error.message });
37   }
38 };

```

Fig. 45 adminController.js

Explicación: Este código en Node.js maneja operaciones administrativas como gestionar usuarios, pedidos y categorías, asegurando que solo los administradores puedan ejecutarlas. Se conecta a la base de datos para obtener estadísticas de ventas, listar usuarios y productos más vendidos, bloquear o eliminar usuarios, y administrar categorías (crear, actualizar, listar y eliminar). Usa async/await para manejar peticiones asíncronas y responde con mensajes adecuados en caso de éxito o error. Además, cada función verifica si el usuario tiene permisos de admin antes de proceder, garantizando seguridad en las acciones permitidas.

```

1  const bcrypt = require('bcryptjs');
2  const jwt = require('jsonwebtoken');
3  const Usuario = require('../modelos/Usuario');
4
5  const registrar = async (req, res) => {
6    const { nombre, correo, contrasena, direccion, rol } = req.body;
7    try {
8      const contrasenaHash = await bcrypt.hash(contrasena, 10);
9      const usuarioId = await Usuario.crear({ nombre, correo, contrasena: contrasenaHash, direccion, rol });
10     res.status(201).json({ mensaje: 'Usuario registrado', usuarioId });
11   } catch (error) {
12     res.status(500).json({ mensaje: 'Error al registrar', error: error.message });
13   }
14 };
15
16 const iniciarSesion = async (req, res) => {
17   const { correo, contrasena } = req.body;
18   try {
19     const usuario = await Usuario.buscarPorCorreo(correo);
20     if (!usuario) return res.status(404).json({ mensaje: 'Usuario no encontrado' });
21     if (usuario.bloqueado) return res.status(403).json({ mensaje: 'Cuenta bloqueada' });
22
23     const esContrasenaValida = await bcrypt.compare(contrasena, usuario.contrasena);
24     if (!esContrasenaValida) return res.status(401).json({ mensaje: 'Contraseña incorrecta' });
25
26     const token = jwt.sign({ id: usuario.id, rol: usuario.rol }, process.env.JWT_SECRET, { expiresIn: '1h' });
27     res.json({ token });
28   } catch (error) {
29     res.status(500).json({ mensaje: 'Error al iniciar sesión', error: error.message });
30   }
31 };
32
33 module.exports = { registrar, iniciarSesion };

```

Fig. 46 *autenticacion.Controller.js*

Explicación: Este código en Node.js define dos funciones para gestionar el registro e inicio de sesión de usuarios utilizando bcryptjs para el cifrado de contraseñas y jsonwebtoken para la autenticación con tokens JWT. La función registrar recibe los datos del usuario, cifra la contraseña y lo almacena en la base de datos, respondiendo con un mensaje de éxito o error. La función iniciarSesion busca al usuario por su correo, verifica que no esté bloqueado y compara la contraseña ingresada con la almacenada. Si es válida, genera un token JWT con el ID y rol del usuario, estableciendo una expiración de una hora. Si ocurre algún error en cualquiera de los procesos, se devuelve un mensaje con el estado HTTP correspondiente.

```

1  const Carrito = require('../modelos/Carrito');
2
3  const agregarProducto = async (req, res) => {
4      const { productId, cantidad = 1 } = req.body;
5      const usuarioId = req.usuario.id;
6
7      if (!productId || cantidad < 1) {
8          return res.status(400).json({ mensaje: 'Producto o cantidad inválida' });
9      }
10
11     try {
12         const resultado = await Carrito.agregarProducto(usuarioId, productId, cantidad);
13         res.status(200).json(resultado);
14     } catch (error) {
15         res.status(400).json({ mensaje: error.message });
16     }
17 };
18
19 const actualizarCantidad = async (req, res) => {
20     const { productId, cantidad } = req.body;
21     const usuarioId = req.usuario.id;
22
23     if (!productId || cantidad < 1) {
24         return res.status(400).json({ mensaje: 'Producto o cantidad inválida' });
25     }
26
27     try {
28         const actualizado = await Carrito.actualizarCantidad(usuarioId, productId, cantidad);
29         if (!actualizado) {
30             return res.status(404).json({ mensaje: 'Producto no encontrado en el carrito' });
31         }
32         res.json({ mensaje: 'Cantidad actualizada' });
33     } catch (error) {
34         res.status(400).json({ mensaje: error.message });
35     }
36 };
37
38 const eliminarProducto = async (req, res) => {
39     const { productId } = req.params;
40     const usuarioId = req.usuario.id;
41
42     try {
43         const eliminado = await Carrito.eliminarProducto(usuarioId, productId);
44         if (!eliminado) {
45             return res.status(404).json({ mensaje: 'Producto no encontrado en el carrito' });
46         }
47         res.json({ mensaje: 'Producto eliminado del carrito' });
48     } catch (error) {
49         res.status(500).json({ mensaje: 'Error al eliminar producto', error: error.message });
50     }
51 };
52
53 const obtenerCarrito = async (req, res) => {
54     const usuarioId = req.usuario.id;
55
56     try {
57         const carrito = await Carrito.obtenerCarrito(usuarioId);
58         res.json(carrito);
59     } catch (error) {
60         res.status(500).json({ mensaje: 'Error al obtener el carrito', error: error.message });
61     }
62 };
63
64 module.exports = { agregarProducto, actualizarCantidad, eliminarProducto, obtenerCarrito };

```

Fig. 47 *carrito.Controller.js*

Explicación: Este código en Node.js define un conjunto de funciones para gestionar un carrito de compras, permitiendo a los usuarios agregar, actualizar, eliminar productos y visualizar su carrito. La función `agregarProducto` recibe el ID del producto y la cantidad, validando que los datos sean correctos antes de almacenarlos. `actualizarCantidad` permite modificar la cantidad de un producto en el carrito y devuelve un error si el producto no está presente. `eliminarProducto` elimina un producto específico del carrito y responde con un mensaje si no se encuentra. Finalmente, `obtenerCarrito` devuelve todos los productos añadidos por el usuario. Todas las funciones capturan errores y envían respuestas HTTP adecuadas para manejar posibles fallos en la base de datos.


```

1  const Pedido = require('../modelos/Pedido');
2  const Carrito = require('../modelos/Carrito');
3
4  const crearPedido = async (req, res) => {
5      const usuarioId = req.usuario.id;
6      try {
7          const carrito = await Carrito.obtenerCarrito(usuarioId);
8          if (!carrito.length) {
9              return res.status(400).json({ mensaje: 'El carrito está vacío' });
10         }
11         const pedidoId = await Pedido.crear(usuarioId, carrito);
12         res.status(201).json({ mensaje: 'Pedido creado', pedidoId });
13     } catch (error) {
14         res.status(500).json({ mensaje: 'Error al crear pedido', error: error.message });
15     }
16 };
17
18 const obtenerHistorial = async (req, res) => {
19     const usuarioId = req.usuario.id;
20     try {
21         const historial = await Pedido.obtenerHistorial(usuarioId);
22         res.json(historial);
23     } catch (error) {
24         res.status(500).json({ mensaje: 'Error al obtener historial', error: error.message });
25     }
26 };
27
28 const cancelarPedido = async (req, res) => {
29     const { pedidoId } = req.params;
30     const usuarioId = req.usuario.id;
31     try {
32         await Pedido.cancelar(pedidoId, usuarioId);
33         res.json({ mensaje: 'Pedido cancelado' });
34     } catch (error) {
35         res.status(400).json({ mensaje: error.message });
36     }
37 };
38
39 const obtenerTodosPedidos = async (req, res) => {
40     if (req.usuario.rol !== 'admin') {
41         return res.status(403).json({ mensaje: 'No autorizado' });
42     }
43     try {
44         const pedidos = await Pedido.obtenerTodos();
45         res.json(pedidos);
46     } catch (error) {
47         res.status(500).json({ mensaje: 'Error al obtener pedidos', error: error.message });
48     }
49 };
50
51 const actualizarEstado = async (req, res) => {
52     if (req.usuario.rol !== 'admin') {
53         return res.status(403).json({ mensaje: 'No autorizado' });
54     }
55     const { pedidoId } = req.params;
56     const { estado, transportista, numeroSeguimiento } = req.body;
57     if (!['Pendiente', 'Enviado', 'Entregado', 'Cancelado'].includes(estado)) {
58         return res.status(400).json({ mensaje: 'Estado inválido' });
59     }
60     try {
61         await Pedido.actualizarEstado(pedidoId, estado, transportista, numeroSeguimiento);
62         res.json({ mensaje: 'Estado actualizado' });
63     } catch (error) {
64         res.status(500).json({ mensaje: 'Error al actualizar estado', error: error.message });
65     }
66 };
67
68 module.exports = { crearPedido, obtenerHistorial, cancelarPedido, obtenerTodosPedidos, actualizarEstado };

```

Fig. 48 *pedido.Controller.js*

Explicación: Este código en Node.js maneja pedidos en un sistema de compras. Permite a los usuarios crear pedidos si su carrito no está vacío, ver su historial y cancelar pedidos. Los administradores pueden ver todos los pedidos y actualizar su estado. Se validan permisos y se manejan errores adecuadamente.

```

1  const Producto = require('../modelos/Producto');
2
3  const crearProducto = async (req, res) => {
4    if (req.usuario.rol !== 'admin') {
5      return res.status(403).json({ mensaje: 'No autorizado' });
6    }
7    const { nombre, descripcion, precio, imagenes, stock, categoria_id } = req.body;
8    try {
9      const productoId = await Producto.crear({ nombre, descripcion, precio, imagenes, stock, categoria_id });
10     res.status(201).json({ mensaje: 'Producto creado', productoId });
11   } catch (error) {
12     res.status(500).json({ mensaje: 'Error al crear producto', error: error.message });
13   }
14 };
15
16 const actualizarProducto = async (req, res) => {
17   if (req.usuario.rol !== 'admin') {
18     return res.status(403).json({ mensaje: 'No autorizado' });
19   }
20   const { id } = req.params;
21   const { nombre, descripcion, precio, imagenes, stock, categoria } = req.body;
22   try {
23     await Producto.actualizar(id, { nombre, descripcion, precio, imagenes, stock, categoria });
24     res.json({ mensaje: 'Producto actualizado' });
25   } catch (error) {
26     res.status(500).json({ mensaje: 'Error al actualizar producto', error: error.message });
27   }
28 };
29
30 const eliminarProducto = async (req, res) => {
31   if (req.usuario.rol !== 'admin') { // Usamos "admin" como mencionaste
32     return res.status(403).json({ mensaje: 'No autorizado' });
33   }
34   const { id } = req.params;
35   try {
36     const filasAfectadas = await Producto.eliminar(id);
37     if (filasAfectadas === 0) {
38       return res.status(404).json({ mensaje: 'Producto no encontrado' });
39     }
40     res.json({ mensaje: 'Producto eliminado' });
41   } catch (error) {
42     res.status(500).json({ mensaje: 'Error al eliminar producto', error: error.message });
43   }
44 };
45
46 const listarProductos = async (req, res) => {
47   const { pagina, limite } = req.query;
48   try {
49     const productos = await Producto.listar({ pagina, limite });
50     res.json(productos);
51   } catch (error) {
52     res.status(500).json({ mensaje: 'Error al listar productos', error: error.message });
53   }
54 };
55
56 const buscarProductos = async (req, res) => {
57   const { categoria, precioMin, precioMax, nombre, ordenarPor, orden } = req.query;
58   try {
59     const productos = await Producto.buscar({ categoria, precioMin, precioMax, nombre, ordenarPor, orden });
60     res.json(productos);
61   } catch (error) {
62     res.status(500).json({ mensaje: 'Error al buscar productos', error: error.message });
63   }
64 };
65
66 const obtenerProductoPorId = async (req, res) => {
67   const { id } = req.params;
68   try {
69     const producto = await Producto.obtenerPorId(id);
70     if (!producto) {
71       return res.status(404).json({ mensaje: 'Producto no encontrado' });
72     }
73     res.json(producto);
74   } catch (error) {
75     res.status(500).json({ mensaje: 'Error al obtener producto', error: error.message });
76   }
77 };
78
79 module.exports = { crearProducto, actualizarProducto, eliminarProducto, listarProductos, buscarProductos, obtenerProductoPorId };

```

Fig. 49 *producto.Controller.js*

Explicación: Este código define funciones para gestionar productos en una API Node.js con control de acceso. Permite a los administradores crear, actualizar y eliminar productos, asegurando validaciones y manejo de errores. Además, incluye funciones para listar productos con paginación, buscarlos por filtros y obtener uno por su ID, respondiendo con mensajes adecuados según el resultado.


```

1  const Usuario = require('../modelos/Usuario');
2  const bcrypt = require('bcryptjs');
3
4  const actualizarUsuario = async (req, res) => {
5      const { id } = req.usuario;
6      const { nombre, correo, direccion } = req.body;
7      try {
8          await Usuario.actualizar(id, { nombre, correo, direccion });
9          res.json({ mensaje: 'Usuario actualizado exitosamente' });
10     } catch (error) {
11         res.status(500).json({ mensaje: 'Error al actualizar usuario', error });
12     }
13 };
14
15 const actualizarContraseña = async (req, res) => {
16     const { id } = req.usuario;
17     const { contraseña } = req.body;
18     try {
19         const contraseñaHash = await bcrypt.hash(contraseña, 10);
20         await Usuario.actualizarContraseña(id, contraseñaHash);
21         res.json({ mensaje: 'Contraseña actualizada exitosamente' });
22     } catch (error) {
23         console.error('Error al actualizar la contraseña:', error);
24         res.status(500).json({ mensaje: 'Error al actualizar contraseña', error: error.message });
25     }
26 };
27
28 // Nuevo método para obtener la información del usuario
29 const obtenerUsuario = async (req, res) => {
30     const { id } = req.usuario; // Obtenido del token JWT
31     try {
32         const usuario = await Usuario.buscarPorId(id);
33         if (!usuario) {
34             return res.status(404).json({ mensaje: 'Usuario no encontrado' });
35         }
36         // No devolvemos la contraseña por seguridad
37         const { contraseña, ...datosUsuario } = usuario;
38         res.json(datosUsuario);
39     } catch (error) {
40         res.status(500).json({ mensaje: 'Error al obtener usuario', error: error.message });
41     }
42 };
43
44 module.exports = { actualizarUsuario, actualizarContraseña, obtenerUsuario };

```

Fig. 50 *usuario.Controller.js*

Explicación: Este código maneja usuarios en una API Node.js. Permite actualizar datos personales, cambiar la contraseña con bcrypt y obtener información del usuario excluyendo la contraseña. Usa el id del request para identificar al usuario y maneja errores con respuestas adecuadas.

- middleware

```

1  const jwt = require('jsonwebtoken');
2
3  const autenticacionMiddleware = (req, res, next) => {
4    const token = req.header('Authorization')?.replace('Bearer ', '');
5    if (!token) {
6      return res.status(401).json({ mensaje: 'No se proporcionó un token' });
7    }
8    try {
9      const decodificado = jwt.verify(token, process.env.JWT_SECRET);
10     req.usuario = decodificado; // Asegúrate de que esto incluya el ID del usuario
11     next();
12   } catch (error) {
13     res.status(401).json({ mensaje: 'Token inválido' });
14   }
15 };
16
17 module.exports = autenticacionMiddleware;

```

Fig. 51 *autenticacionMiddleware.js*

Explicación: Este middleware en Node.js verifica tokens JWT de la cabecera Authorization. Si el token es válido, agrega los datos del usuario a req.usuario y permite continuar; de lo contrario, responde con error 401.

- **modelos**

```

1  const db = require('../config/db');
2  const Producto = require('../Producto');
3  class Carrito {
4    static async agregarProducto(usuarioId, productoId, cantidad) {
5      const stock = await Producto.obtenerStock(productoId);
6      if (stock === null) {
7        throw new Error('Producto no encontrado');
8      }
9      if (stock < cantidad) {
10       throw new Error('Stock insuficiente');
11     }
12
13     const [existing] = await db.execute(
14       'SELECT cantidad FROM carritos WHERE usuario_id = ? AND producto_id = ?',
15       [usuarioId, productoId]
16     );
17     if (existing.length > 0) {
18       const nuevaCantidad = existing[0].cantidad + cantidad;
19       if (stock < nuevaCantidad) {
20         throw new Error('Stock insuficiente para la cantidad solicitada');
21       }
22       await db.execute(
23         'UPDATE carritos SET cantidad = ?, actualizado_en = NOW() WHERE usuario_id = ? AND producto_id = ?',
24         [nuevaCantidad, usuarioId, productoId]
25       );
26       await Producto.actualizarStock(productoId, -cantidad); // Restar stock
27       return { mensaje: 'Cantidad actualizada', cantidad: nuevaCantidad };
28     } else {
29       await db.execute(
30         'INSERT INTO carritos (usuario_id, producto_id, cantidad) VALUES (?, ?, ?)',
31         [usuarioId, productoId, cantidad]
32       );
33       await Producto.actualizarStock(productoId, -cantidad); // Restar stock
34       return { mensaje: 'Producto agregado' };
35     }
36   }
37 }

```

Fig. 52 *Carrito.js*

Explicación: Este código en Node.js define la clase Carrito, que gestiona productos en un carrito de compras. Permite agregar, actualizar, eliminar y obtener productos, validando el stock antes de cada operación y ajustándolo según cambios. También permite vaciar el carrito completamente.

```

1  const db = require('../config/db');
2
3  class Categoria {
4      static async crear({ nombre, descripcion }) {
5          const [resultado] = await db.execute(
6              'INSERT INTO categorias (nombre, descripcion) VALUES (?, ?)',
7              [nombre, descripcion || null]
8          );
9          return resultado.insertId;
10     }
11
12     static async actualizar(id, { nombre, descripcion }) {
13         await db.execute(
14             'UPDATE categorias SET nombre = ?, descripcion = ? WHERE id = ?',
15             [nombre, descripcion || null, id]
16         );
17     }
18
19     static async eliminar(id) {
20         const [resultado] = await db.execute('DELETE FROM categorias WHERE id = ?', [id]);
21         return resultado.affectedRows > 0;
22     }
23
24     static async listar() {
25         const [filas] = await db.execute('SELECT * FROM categorias ORDER BY nombre ASC');
26         return filas;
27     }
28
29     static async buscarPorId(id) {
30         const [filas] = await db.execute('SELECT * FROM categorias WHERE id = ?', [id]);
31         return filas[0];
32     }
33 }
34
35 module.exports = Categoria;

```

Fig. 53 *Categoria.js*

Explicación: Este código define la clase *Categoria* en Node.js para gestionar categorías en MySQL. Permite crear, actualizar, eliminar, listar y buscar categorías por id, interactuando con la base de datos de forma asincrónica. La clase se exporta para su uso en otros módulos.

```

1  const db = require('../config/db');
2  const Producto = require('../Producto');
3
4  class Pedido {
5    // Crear un pedido a partir del carrito
6    static async crear(usuarioId, carrito) {
7      const total = carrito.reduce((sum, item) => sum + item.precio * item.cantidad, 0);
8      const [resultado] = await db.execute(
9        'INSERT INTO pedidos (usuario_id, total, estado) VALUES (?, ?, ?)',
10       [usuarioId, total, 'Pendiente']
11      );
12      const pedidoId = resultado.insertId;
13
14      // Insertar detalles del pedido
15      for (const item of carrito) {
16        await db.execute(
17          'INSERT INTO detalles_pedido (pedido_id, producto_id, cantidad, precio_unitario) VALUES (?, ?, ?, ?)',
18          [pedidoId, item.producto_id, item.cantidad, item.precio]
19        );
20      }
21
22      // Limpiar el carrito tras crear el pedido
23      await db.execute('DELETE FROM carritos WHERE usuario_id = ?', [usuarioId]);
24
25      return pedidoId;
26    }
27
28    // Obtener el historial de pedidos de un usuario
29    static async obtenerHistorial(usuarioId) {
30      const [pedidos] = await db.execute(
31        'SELECT p.id, p.total, p.estado, p.transportista, p.numero_seguimiento, p.creado_en, p.actualizado_en
32         FROM pedidos p
33         WHERE p.usuario_id = ?
34         ORDER BY p.creado_en DESC',
35        [usuarioId]
36      );
37    }
38  }

```

Fig. 54 *Pedido.js*

Explicación: Este código define la clase Pedido en Node.js para gestionar pedidos en una base de datos MySQL. Permite crear un pedido desde el carrito, almacenar los detalles y vaciar el carrito. También obtiene el historial de pedidos de un usuario y permite cancelarlos, devolviendo el stock. Además, los administradores pueden listar todos los pedidos y actualizar su estado, transportista y número de seguimiento. La clase se exporta para su uso en otros módulos.

```

1  const db = require('../config/db');
2
3  class Producto {
4    static async crear({ nombre, descripcion, precio, imagenes, stock, categoria_id }) {
5      const [resultado] = await db.execute(
6        'INSERT INTO productos (nombre, descripcion, precio, imagenes, stock, categoria_id) VALUES (?, ?, ?, ?, ?, ?)',
7        [nombre, descripcion, precio, imagenes, stock, categoria_id || null]
8      );
9      return resultado.insertId;
10    }
11
12    static async actualizar(id, { nombre, descripcion, precio, imagenes, stock, categoria_id }) {
13      await db.execute(
14        'UPDATE productos SET nombre = ?, descripcion = ?, precio = ?, imagenes = ?, stock = ?, categoria_id = ? WHERE id = ?',
15        [nombre, descripcion, precio, imagenes, stock, categoria_id || null, id]
16      );
17    }
18
19    static async eliminar(id) {
20      await db.execute('DELETE FROM productos WHERE id = ?', [id]);
21    }
22
23    static async listar({ pagina = 1, limite = 10 }) {
24      const offset = (pagina - 1) * limite;
25      const [filas] = await db.execute(
26        'SELECT p.*, c.nombre AS categoria FROM productos p LEFT JOIN categorias c ON p.categoria_id = c.id LIMIT ? OFFSET ?',
27        [limite, offset].map(String)
28      );
29      return filas;
30    }
31  }

```

Fig. 55 *Producto.js*

Explicación: Este código define la clase Producto en Node.js para gestionar productos en MySQL. Permite crear, actualizar, eliminar, listar con filtros, obtener stock, actualizarlo y consultar los productos más vendidos. También permite buscar productos por id, incluyendo su categoría. La clase se exporta para su uso en otros módulos.

```
1  const db = require('../config/db');
2
3  class Usuario {
4    static async crear({ nombre, correo, contrasena, direccion, rol }) {
5      const [resultado] = await db.execute(
6        'INSERT INTO usuarios (nombre, correo, contrasena, direccion, rol) VALUES (?, ?, ?, ?, ?)',
7        [nombre, correo, contrasena, direccion, rol]
8      );
9      return resultado.insertId;
10   }
11
12   static async buscarPorCorreo(correo) {
13     const [filas] = await db.execute('SELECT * FROM usuarios WHERE correo = ?', [correo]);
14     return filas[0];
15   }
16
17   static async buscarPorId(id) {
18     const [filas] = await db.execute('SELECT * FROM usuarios WHERE id = ?', [id]);
19     return filas[0];
20   }
21
22   static async actualizar(id, { nombre, correo, direccion }) {
23     await db.execute(
24       'UPDATE usuarios SET nombre = ?, correo = ?, direccion = ? WHERE id = ?',
25       [nombre, correo, direccion, id]
26     );
27   }
28
29   static async actualizarContrasena(id, contrasena) {
30     await db.execute(
31       'UPDATE usuarios SET contrasena = ? WHERE id = ?',
32       [contrasena, id]
33     );
34   }
35
36   // Nuevo: Bloquear o desbloquear usuario
37   static async bloquear(id, bloquear) {
38     await db.execute('UPDATE usuarios SET bloqueado = ? WHERE id = ?', [bloquear, id]);
39   }
40 }
```

Fig. 56 *Usuario.js*

Explicación: Este código define la clase Usuario en Node.js para gestionar usuarios en una base de datos MySQL. Permite crear usuarios con nombre, correo, contraseña, dirección y rol, además de buscarlos por correo o id. También permite actualizar datos personales, cambiar la contraseña y bloquear o desbloquear usuarios. Además, incluye funciones para eliminar usuarios y listarlos ordenados por nombre. La clase se exporta para su uso en otros módulos.

- rutas


```

1  const express = require('express');
2  const autenticacionMiddleware = require('../middleware/autenticacionMiddleware');
3  const {
4      obtenerEstadisticas,
5      listarUsuarios,
6      bloquearUsuario,
7      eliminarUsuario,
8      crearCategoria,
9      actualizarCategoria,
10     eliminarCategoria,
11     listarCategorias,
12 } = require('../controladores/adminController');
13
14 const router = express.Router();
15
16 // Rutas protegidas para administradores
17 router.get('/estadisticas', autenticacionMiddleware, obtenerEstadisticas);
18 router.get('/usuarios', autenticacionMiddleware, listarUsuarios);
19 router.put('/usuarios/bloquear/:usuarioId', autenticacionMiddleware, bloquearUsuario);
20 router.delete('/usuarios/eliminar/:usuarioId', autenticacionMiddleware, eliminarUsuario);
21 router.post('/categorias/crear', autenticacionMiddleware, crearCategoria);
22 router.put('/categorias/actualizar/:categoriaId', autenticacionMiddleware, actualizarCategoria);
23 router.delete('/categorias/eliminar/:categoriaId', autenticacionMiddleware, eliminarCategoria);
24
25 // Ruta pública
26 router.get('/categorias', listarCategorias);
27
28 module.exports = router;

```

Fig. 57 *adminRoutes.js*

Explicación: Este código en Node.js define un router con Express para manejar rutas administrativas, protegiéndolas con un middleware de autenticación. Permite a administradores ver estadísticas, listar, bloquear y eliminar usuarios, y gestionar categorías (crear, actualizar y eliminar). Solo la ruta para listar categorías es pública, mientras que las demás requieren autenticación.

```

1  const express = require('express');
2  const { registrar, iniciarSesion } = require('../controladores/autenticacionController');
3
4  const router = express.Router();
5
6  router.post('/registrar', registrar);
7  router.post('/iniciar-sesion', iniciarSesion);
8
9  module.exports = router;

```

Fig. 58 *autenticacionRoutes.js*

Explicación: Este código en Node.js define un router con Express para manejar la autenticación de usuarios. Importa dos funciones, registrar e iniciarSesion, desde un controlador y las asocia a las rutas "/registrar" y "/iniciar-sesion" mediante solicitudes POST. Finalmente, exporta el router para su uso en la aplicación.


```

1 const express = require('express');
2 const autenticacionMiddleware = require('../middleware/autenticacionMiddleware');
3 const { agregarProducto, actualizarCantidad, eliminarProducto, obtenerCarrito } = require('../controladores/carritoController');
4
5 const router = express.Router();
6
7 // Rutas protegidas por autenticación
8 router.post('/agregar', autenticacionMiddleware, agregarProducto);
9 router.put('/actualizar', autenticacionMiddleware, actualizarCantidad);
10 router.delete('/eliminar/:productoId', autenticacionMiddleware, eliminarProducto);
11 router.get('/', autenticacionMiddleware, obtenerCarrito);
12
13 module.exports = router;

```

Fig. 59 *carritoRoutes.js*

Explicación: Este código en Node.js define un router con Express para gestionar el carrito de compras. Todas las rutas están protegidas con un middleware de autenticación, asegurando que solo usuarios autenticados puedan agregar, actualizar, eliminar productos y obtener el carrito. Se exporta el router para su uso en la aplicación.

```

1 const express = require('express');
2 const autenticacionMiddleware = require('../middleware/autenticacionMiddleware');
3 const { crearPedido, obtenerHistorial, cancelarPedido, obtenerTodosPedidos, actualizarEstado } = require('../controladores/pedidoController');
4
5 const router = express.Router();
6
7 router.post('/crear', autenticacionMiddleware, crearPedido);
8 router.get('/historial', autenticacionMiddleware, obtenerHistorial);
9 router.put('/cancelar/:pedidoId', autenticacionMiddleware, cancelarPedido);
10 router.get('/todos', autenticacionMiddleware, obtenerTodosPedidos);
11 router.put('/actualizar/:pedidoId', autenticacionMiddleware, actualizarEstado);
12
13 module.exports = router;

```

Fig. 60 *pedidoRoutes.js*

Explicación: Este código en Node.js define un router con Express para gestionar pedidos en un sistema de compras. Todas las rutas están protegidas con un middleware de autenticación, asegurando que solo usuarios autenticados puedan crear pedidos, ver su historial, cancelar pedidos y actualizar su estado. También permite a los administradores obtener todos los pedidos. Se exporta el router para su uso en la aplicación.

```

1 const express = require('express');
2 const autenticacionMiddleware = require('../middleware/autenticacionMiddleware');
3 const { crearProducto, actualizarProducto, eliminarProducto, listarProductos, buscarProductos, obtenerProductoPorId } = require('../controladores/productoController');
4
5 const router = express.Router();
6
7 router.post('/', autenticacionMiddleware, crearProducto);
8 router.put('/:id', autenticacionMiddleware, actualizarProducto);
9 router.delete('/:id', autenticacionMiddleware, eliminarProducto);
10 router.get('/', listarProductos);
11 router.get('/buscar', buscarProductos);
12 router.get('/:id', obtenerProductoPorId);
13
14 module.exports = router;

```

Fig. 61 *productoRoutes.js*

Explicación: Este código en Node.js define un router con Express para gestionar productos en una aplicación. Usa un middleware de autenticación para proteger las rutas que permiten crear, actualizar y eliminar productos. También incluye rutas públicas para listar, buscar y obtener productos por ID. Finalmente, exporta el router para su uso en la aplicación.

```

1  const express = require('express');
2  const autenticacionMiddleware = require('../middleware/autenticacionMiddleware');
3  const { actualizarUsuario, actualizarContraseña, obtenerUsuario } = require('../controladores/usuarioController');
4
5  const router = express.Router();
6
7  router.put('/actualizar', autenticacionMiddleware, actualizarUsuario);
8  router.put('/actualizar-contrasena', autenticacionMiddleware, actualizarContraseña);
9  router.get('/perfil', autenticacionMiddleware, obtenerUsuario); // Nueva ruta
10
11 module.exports = router;

```

Fig. 62 *usuarioRoutes.js*

Explicación: Este código en Node.js define un router con Express para gestionar usuarios. Usa un middleware de autenticación para proteger las rutas que permiten actualizar datos del usuario, cambiar la contraseña y obtener su perfil. Todas las rutas requieren autenticación y el router se exporta para su uso en la aplicación.

Creación de la base de datos.

```

1  -- phpMyAdmin SQL Dump
2  -- version 5.2.1
3  -- https://www.phpmyadmin.net/
4  --
5  -- Servidor: 127.0.0.1
6  -- Tiempo de generación: 04-03-2025 a las 05:37:22
7  -- Versión del servidor: 10.4.32-MariaDB
8  -- Versión de PHP: 8.0.30
9
10 SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
11 START TRANSACTION;
12 SET time_zone = "+00:00";
13
14
15 /*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
16 /*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
17 /*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
18 /*!40101 SET NAMES utf8mb4 */;
19
20 --
21 -- Base de datos: `tienda_virtual`
22 --
23
24 --
25
26 --
27 -- Estructura de tabla para la tabla `carritos`
28 --
29
30 CREATE TABLE `carritos` (
31   `id` int(11) NOT NULL,
32   `usuario_id` int(11) NOT NULL,
33   `producto_id` int(11) NOT NULL,
34   `cantidad` int(11) NOT NULL DEFAULT 1,
35   `creado_en` timestamp NOT NULL DEFAULT current_timestamp(),
36   `actualizado_en` timestamp NOT NULL DEFAULT current_timestamp() ON UPDATE current_timestamp()
37 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
38
39 --
40
41 --
42 -- Estructura de tabla para la tabla `categorias`
43 --

```

```

1 CREATE TABLE `categorias` (
2   `id` int(11) NOT NULL,
3   `nombre` varchar(255) NOT NULL,
4   `descripcion` text DEFAULT NULL,
5   `creado_en` timestamp NOT NULL DEFAULT current_timestamp()
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
7
8 --
9 -- Volcado de datos para la tabla `categorias`
10 --
11
12 INSERT INTO `categorias` (`id`, `nombre`, `descripcion`, `creado_en`) VALUES
13 (4, 'Celulares', 'Dispositivos móviles', '2025-03-04 01:03:44'),
14 (5, 'Laptops', 'Computadores para todo tipo de actividades', '2025-03-04 01:04:33'),
15 (6, 'Auriculares', 'Dispositivos para escuchar música alámbricos e inalámbricos', '2025-03-04 01:05:10'),
16 (7, 'Televisores', 'Smart-TV -4k -para todos los gustos', '2025-03-04 01:05:53');
17
18 -----
19
20 --
21 -- Estructura de tabla para la tabla `detalles_pedido`
22 --
23
24 CREATE TABLE `detalles_pedido` (
25   `id` int(11) NOT NULL,
26   `pedido_id` int(11) NOT NULL,
27   `producto_id` int(11) NOT NULL,
28   `cantidad` int(11) NOT NULL,
29   `precio_unitario` decimal(10,2) NOT NULL
30 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
31

```

```

1 --
2 -- Volcado de datos para la tabla `detalles_pedido`
3 --
4
5 INSERT INTO `detalles_pedido` (`id`, `pedido_id`, `producto_id`, `cantidad`, `precio_unitario`) VALUES
6 (4, 2, 13, 2, 329.99),
7 (5, 3, 8, 5, 799.99),
8 (6, 4, 9, 1, 999.99);
9
10 -----
11
12 --
13 -- Estructura de tabla para la tabla `pedidos`
14 --
15
16 CREATE TABLE `pedidos` (
17   `id` int(11) NOT NULL,
18   `usuario_id` int(11) NOT NULL,
19   `total` decimal(10,2) NOT NULL,
20   `estado` enum('Pendiente','Enviado','Entregado','Cancelado') DEFAULT 'Pendiente',
21   `transportista` varchar(255) DEFAULT NULL,
22   `numero_seguimiento` varchar(255) DEFAULT NULL,
23   `creado_en` timestamp NOT NULL DEFAULT current_timestamp(),
24   `actualizado_en` timestamp NOT NULL DEFAULT current_timestamp() ON UPDATE current_timestamp()
25 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
26
27 --
28 -- Volcado de datos para la tabla `pedidos`
29 --
30
31 INSERT INTO `pedidos` (`id`, `usuario_id`, `total`, `estado`, `transportista`, `numero_seguimiento`, `creado_en`, `actualizado_en`) VALUES
32 (2, 9, 659.98, 'Cancelado', NULL, NULL, '2025-03-04 01:53:15', '2025-03-04 02:41:14'),
33 (3, 9, 3999.95, 'Cancelado', NULL, NULL, '2025-03-04 02:37:51', '2025-03-04 02:41:36'),
34 (4, 9, 999.99, 'Pendiente', NULL, NULL, '2025-03-04 03:10:50', '2025-03-04 03:10:50');
35
36 -----
37
38 --
39 -- Estructura de tabla para la tabla `productos`
40 --
41
42 CREATE TABLE `productos` (
43   `id` int(11) NOT NULL,
44   `nombre` varchar(255) NOT NULL,
45   `descripcion` text DEFAULT NULL,
46   `precio` decimal(10,2) NOT NULL,
47   `imagenes` text DEFAULT NULL,
48   `stock` int(11) NOT NULL,
49   `creado_en` timestamp NOT NULL DEFAULT current_timestamp(),
50   `categoria_id` int(11) DEFAULT NULL
51 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

```

```

1 --
2 -- Volcado de datos para la tabla 'productos'
3 --
4
5 INSERT INTO productos ('id', 'nombre', 'descripcion', 'precio', 'imagen', 'stock', 'estado', 'categoria_id') VALUES
6 (1, 'Samsung Galaxy S21', 'Teléfono de gama alta con cámara de 108MP, pantalla 6.2" AMOLED, 5G, 12GB RAM, 256GB almacenamiento. Incluye funda y auriculares inalámbricos.', 799.99, 'https://www.samsung.com/latinamerica/assets/images/2021/01/20/20210120_samsung_galaxy_s21_01.jpg', 10, '2025-03-03 14:37:21', 0),
7 (2, 'iPhone 13', 'Smartphone con pantalla 6.1" Super Retina XDR, cámara triple 12MP, chip A15 Bionic, 5G, 128GB almacenamiento. Incluye funda y auriculares.', 699.99, 'https://www.apple.com/latam/iphone-13/', 5, '2025-03-03 14:37:22', 0),
8 (3, 'Huawei P50 Pocket', 'Teléfono plegable con pantalla 7.6" OLED, cámara triple 50MP, chip Kirin 9000, 5G, 512GB almacenamiento. Incluye funda y auriculares.', 1299.99, 'https://www.huawei.com/latam/phones/p50-pocket', 3, '2025-03-03 14:37:23', 0),
9 (4, 'Xiaomi Mi 11 Ultra', 'Smartphone con pantalla 6.8" AMOLED, cámara triple 50MP, chip Snapdragon 8 Gen 1, 5G, 12GB RAM, 512GB almacenamiento. Incluye funda y auriculares.', 899.99, 'https://www.mi.com/latam/phones/mi-11-ultra', 8, '2025-03-03 14:37:24', 0),
10 (5, 'Google Pixel 6 Pro', 'Smartphone con pantalla 6.7" OLED, cámara triple 50MP, chip Google Tensor, 5G, 128GB almacenamiento. Incluye funda y auriculares.', 699.99, 'https://www.google.com/latam/phones/pixel-6-pro', 7, '2025-03-03 14:37:25', 0),
11 (6, 'Motorola Edge 2023', 'Smartphone con pantalla 6.7" OLED, cámara triple 50MP, chip Snapdragon 7 Gen 1, 5G, 128GB almacenamiento. Incluye funda y auriculares.', 499.99, 'https://www.motorola.com/latam/phones/edge-2023', 6, '2025-03-03 14:37:26', 0),
12 (7, 'OnePlus 10 Pro', 'Smartphone con pantalla 6.7" AMOLED, cámara triple 48MP, chip Snapdragon 8 Gen 1, 5G, 128GB almacenamiento. Incluye funda y auriculares.', 799.99, 'https://www.oneplus.com/latam/phones/10-pro', 9, '2025-03-03 14:37:27', 0),
13 (8, 'Honor Magic5 Pro', 'Smartphone con pantalla 6.8" OLED, cámara triple 50MP, chip Snapdragon 8 Gen 1, 5G, 128GB almacenamiento. Incluye funda y auriculares.', 899.99, 'https://www.honor.com/latam/phones/magic5-pro', 4, '2025-03-03 14:37:28', 0),
14 (9, 'Oppo Find N', 'Smartphone plegable con pantalla 7.1" AMOLED, cámara triple 50MP, chip Snapdragon 8 Gen 1, 5G, 512GB almacenamiento. Incluye funda y auriculares.', 1299.99, 'https://www.oppo.com/latam/phones/find-n', 2, '2025-03-03 14:37:29', 0),
15 (10, 'Nothing Phone (2)', 'Smartphone con pantalla 6.7" OLED, cámara triple 50MP, chip Snapdragon 7 Gen 1, 5G, 128GB almacenamiento. Incluye funda y auriculares.', 599.99, 'https://www.nothing.tech/latam/phones/phone-2', 1, '2025-03-03 14:37:30', 0),
16
17 --
18 --
19 --
20 --
21 --
22 --
23 --
24 --
25 --
26 --
27 --
28 --
29 --
30 --
31 --
32 --
33 --
34 --
35 --
36 --
37 --
38 --
39 --
40 --
41 --
42 --
43 --
44 --
45 --

```

```

1 --
2 -- Volcado de datos para la tabla 'usuarios'
3 --
4
5 INSERT INTO usuarios ('id', 'nombre', 'correo', 'contraseña', 'direccion', 'rol', 'creado_en', 'bloqueado') VALUES
6 (7, 'Ana Garcia', 'anagarcia@gmail.com', '$2b$10$G8e4j1J1p9wZt2z3n67eQ4Rv5t4xZvg0Cn8CPh744vNt0', 'Calle 123', 'cliente', '2025-03-03 14:37:21', 0),
7 (8, 'Nestor Pazos', 'nestor@gmail.com', '$2b$10$CkF00iG0h1eQL1d4n4y0p3m4Zjy0F7d1v1w534kQ1041z1z', 'Comite uno', 'cliente', '2025-03-03 14:53:27', 0),
8 (9, 'Anthony Villarreal', 'anvillarreal@espe.edu.ec', '$2b$10$Jdkp7YiaZ7revfC7n1d4l0uZsp6n07hmFvq2X1B:6AJH054h4u', 'Av. del maestro y Bonifaz Cumbra N60-122', 'admin', '2025-03-03 15:15:27', 0),
9 (10, 'Ariel Leonidas', 'alreyes2@espe.edu.ec', '$2b$10$B8swe1fY5QvXmD0001V0n0VERRt1uDP:Z0kydydyFuF/X14a', 'Micasa', 'cliente', '2025-03-04 01:00:21', 0),
10 (11, 'maria', 'maria@gmail.com', '$2b$10$BA.yXp839NufQh48XP4U5:3AFbkbchYYAmJ8u0IXk18zyt0DpXK', 'Sucasa', 'cliente', '2025-03-04 04:21:25', 0),
11
12 --
13 --
14 --
15 --
16 --
17 --
18 --
19 --
20 --
21 --
22 --
23 --
24 --
25 --
26 --
27 --
28 --
29 --
30 --
31 --
32 --
33 --
34 --
35 --
36 --
37 --
38 --
39 --
40 --
41 --
42 --
43 --
44 --
45 --

```



```
1  --
2  -- Indices de la tabla `productos`
3  --
4  ALTER TABLE `productos`
5      ADD PRIMARY KEY (`id`),
6      ADD KEY `categoria_id` (`categoria_id`);
7
8  --
9  -- Indices de la tabla `usuarios`
10 --
11 ALTER TABLE `usuarios`
12     ADD PRIMARY KEY (`id`),
13     ADD UNIQUE KEY `correo` (`correo`);
14
15 --
16 -- AUTO_INCREMENT de las tablas volcadas
17 --
18
19 --
20 -- AUTO_INCREMENT de la tabla `carritos`
21 --
22 ALTER TABLE `carritos`
23     MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=12;
24
25 --
26 -- AUTO_INCREMENT de la tabla `categorias`
27 --
28 ALTER TABLE `categorias`
29     MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=8;
30
31 --
32 -- AUTO_INCREMENT de la tabla `detalles_pedido`
33 --
34 ALTER TABLE `detalles_pedido`
35     MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=7;
36
```



```

1  --
2  -- AUTO_INCREMENT de la tabla 'pedidos'
3  --
4  ALTER TABLE `pedidos`
5    MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=5;
6
7  --
8  -- AUTO_INCREMENT de la tabla 'productos'
9  --
10 ALTER TABLE `productos`
11   MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=16;
12
13 --
14 -- AUTO_INCREMENT de la tabla 'usuarios'
15 --
16 ALTER TABLE `usuarios`
17   MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=17;
18
19 --
20 -- Restricciones para tablas volcadas
21 --
22 --
23 --
24 -- Filtros para la tabla 'carritos'
25 --
26 ALTER TABLE `carritos`
27   ADD CONSTRAINT `carritos_ibfk_1` FOREIGN KEY (`usuario_id`) REFERENCES `usuarios` (`id`),
28   ADD CONSTRAINT `carritos_ibfk_2` FOREIGN KEY (`producto_id`) REFERENCES `productos` (`id`);
29
30 --
31 -- Filtros para la tabla 'detalles_pedido'
32 --
33 ALTER TABLE `detalles_pedido`
34   ADD CONSTRAINT `detalles_pedido_ibfk_1` FOREIGN KEY (`pedido_id`) REFERENCES `pedidos` (`id`),
35   ADD CONSTRAINT `detalles_pedido_ibfk_2` FOREIGN KEY (`producto_id`) REFERENCES `productos` (`id`);
36
37 --
38 -- Filtros para la tabla 'pedidos'
39 --
40 ALTER TABLE `pedidos`
41   ADD CONSTRAINT `pedidos_ibfk_1` FOREIGN KEY (`usuario_id`) REFERENCES `usuarios` (`id`);
42
43 --
44 -- Filtros para la tabla 'productos'
45 --
46 ALTER TABLE `productos`
47   ADD CONSTRAINT `productos_ibfk_1` FOREIGN KEY (`categoria_id`) REFERENCES `categorias` (`id`);
48 COMMIT;
49
50 /*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
51 /*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
52 /*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
53

```

Fig. 63 creación de la base de datos

Explicación: Este es un volcado de base de datos SQL generado por phpMyAdmin para la base de datos "tienda_virtual", que almacena información de una tienda en línea. Define las estructuras de varias tablas, incluyendo usuarios, productos, categorías, pedidos y carritos de compra, además de sus relaciones. Se incluyen datos preexistentes, como productos de tecnología, usuarios con roles de "cliente" y "admin", y pedidos con estados como "Pendiente" o "Cancelado". También configura claves primarias, índices y restricciones de clave foránea para garantizar la integridad de los datos. Finalmente, establece la codificación utf8mb4 y activa el AUTO_INCREMENT en las tablas correspondientes.

7. Resultados

The image shows a user registration interface. At the top, a dark header bar contains the text 'Mi Tienda' next to a storefront icon and a red menu icon. The main content area has a light gray brick background. Centered on this background is a dark gray rounded rectangle containing the title 'Registro de Usuario' and a registration form. The form consists of four input fields: the first contains 'Javier', the second contains 'segu3103ramos@hotmail.com', the third contains four dots, and the fourth contains 'Santa Rita'. Below these fields is a blue button labeled 'Registrar'. At the bottom of the form area, there is a link that says '¿Ya tienes una cuenta? Inicia sesión'. A green arrow points from a green-bordered box labeled 'Registro de Usuario' to the title of the form.

Mi Tienda

Registro de Usuario

Javier

segu3103ramos@hotmail.com

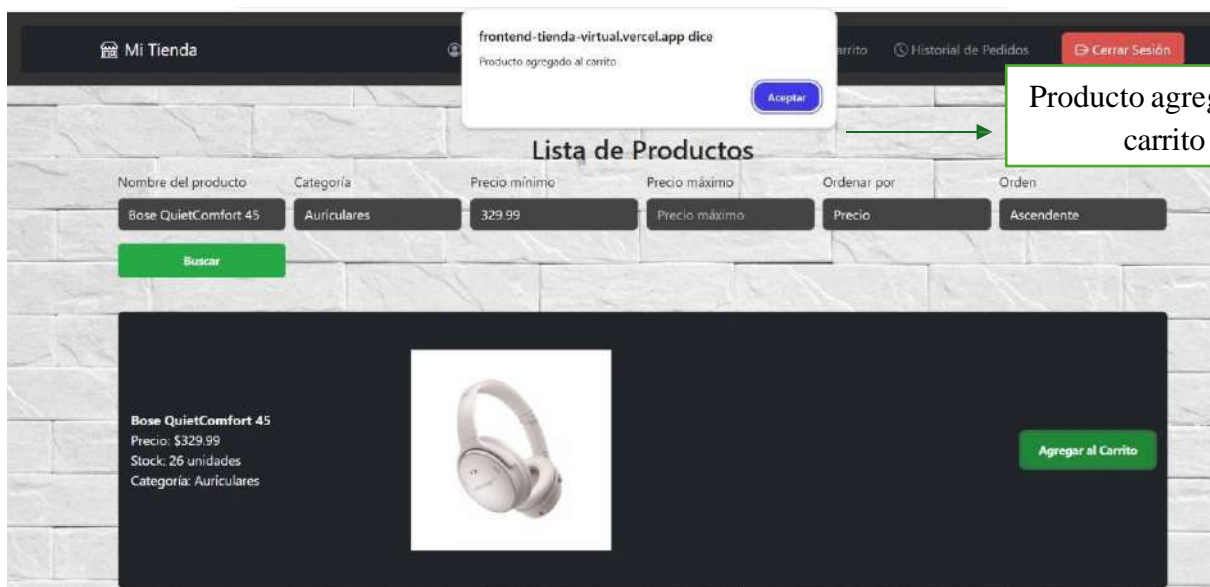
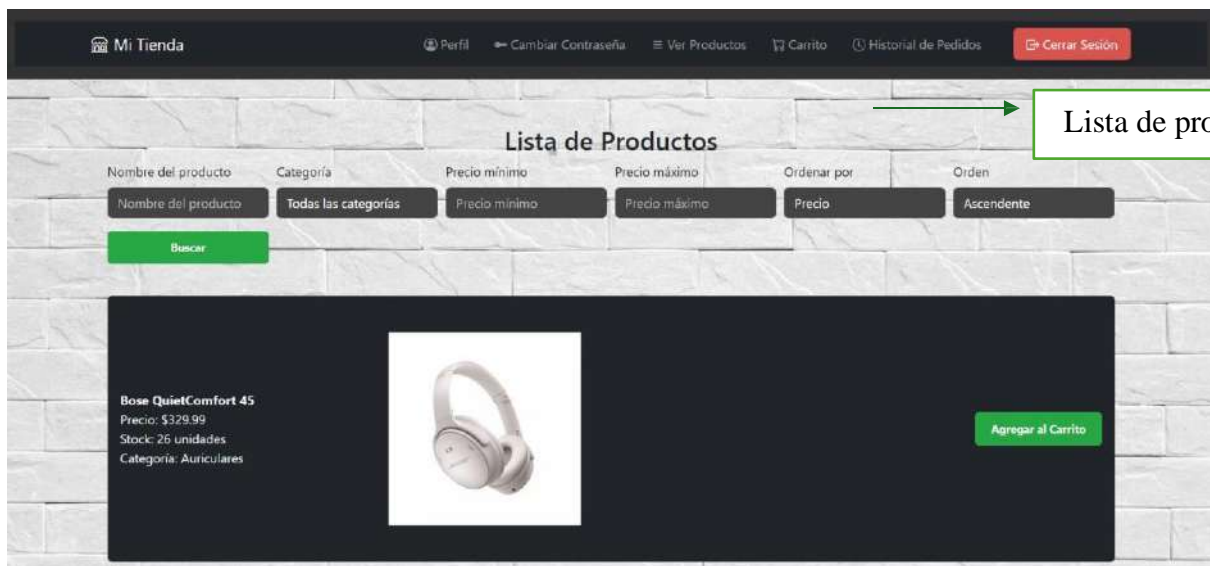
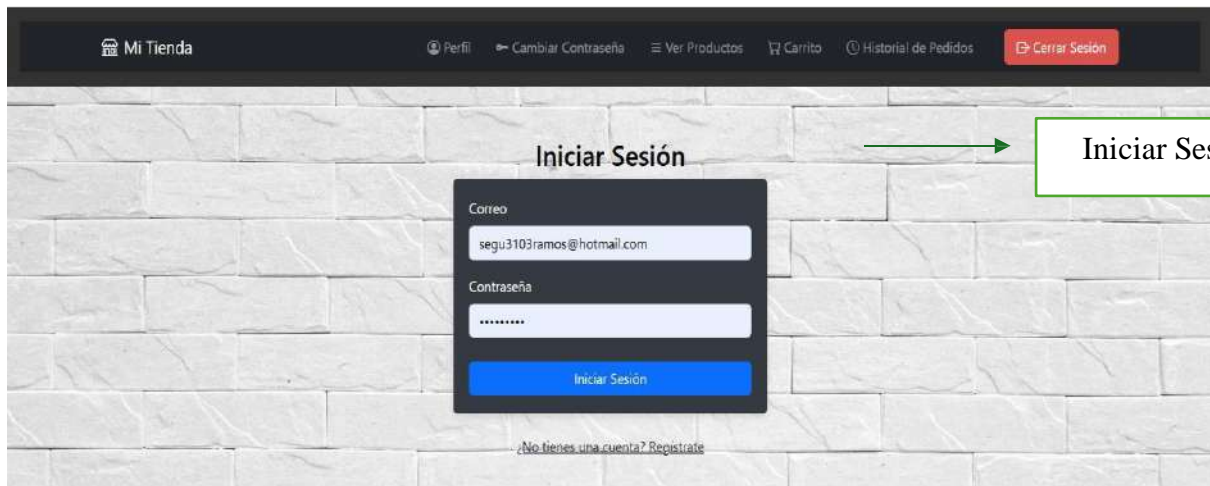
....

Santa Rita

Registrar

[¿Ya tienes una cuenta? Inicia sesión](#)

Registro de Usuario





Lista de Productos

Nombre del producto Categoría Precio mínimo Precio máximo Ordenar por Orden

Nombre del producto Todas las categorías Precio mínimo Precio máximo Precio Ascendente

Buscar

Bose QuietComfort 45
Precio: \$329.99
Stock: 25 unidades
Categoría: Auriculares

Sony WH-1000XM5
Precio: \$349.99
Stock: 39 unidades
Categoría: Auriculares

Samsung Galaxy S23
Precio: \$799.99
Stock: 29 unidades
Categoría: Celulares

Lista de productos

Gestión de Categorías

Crear Nueva Categoría

Nombre

Descripción

Crear Categoría

Gestión de Cate

ID	Nombre	Descripción	Acciones
6	Auriculares	Dispositivos para escuchar música alámbricos e inalámbricos	<div><div>Editar</div><div>Eliminar</div></div>
4	Celulares	Dispositivos móviles	<div><div>Editar</div><div>Eliminar</div></div>
5	Laptops	Computadores para todo tipo de actividades	<div><div>Editar</div><div>Eliminar</div></div>

Mi Tienda

PerfilCambiar ContraseñaVer ProductosCarritoHistorial de PedidosAdministraciónCerrar Sesión

Gestión de Productos

Crear Nuevo Producto

Nombre:

Javier

Descripción:

Samsung Galaxy S23

Precio:

799.99

URLs de Imágenes:

URLs de imágenes (separadas por comas)

Stock:

29

Categoría:

Auriculares

Crear Producto

Gestión de Productos

Lista de Productos		
Samsung Galaxy S23	\$799.99 - 30 unidades	Categoría: Celulares
iPhone 14	\$999.99 - 19 unidades	Categoría: Celulares
MacBook Pro 14	\$1,999.99 - 15 unidades	Categoría: Laptops
Dell XPS 13	\$1,499.99 - 25 unidades	Categoría: Laptops
Sony WH-1000XM5	\$349.99 - 40 unidades	Categoría: Auriculares
Bose QuietComfort 45	\$329.99 - 29 unidades	Categoría: Auriculares
LG OLED C1 55"	\$1,499.99 - 10 unidades	Categoría: Televisores
Samsung QN90A 65"	\$1,799.99 - 15 unidades	Categoría: Televisores

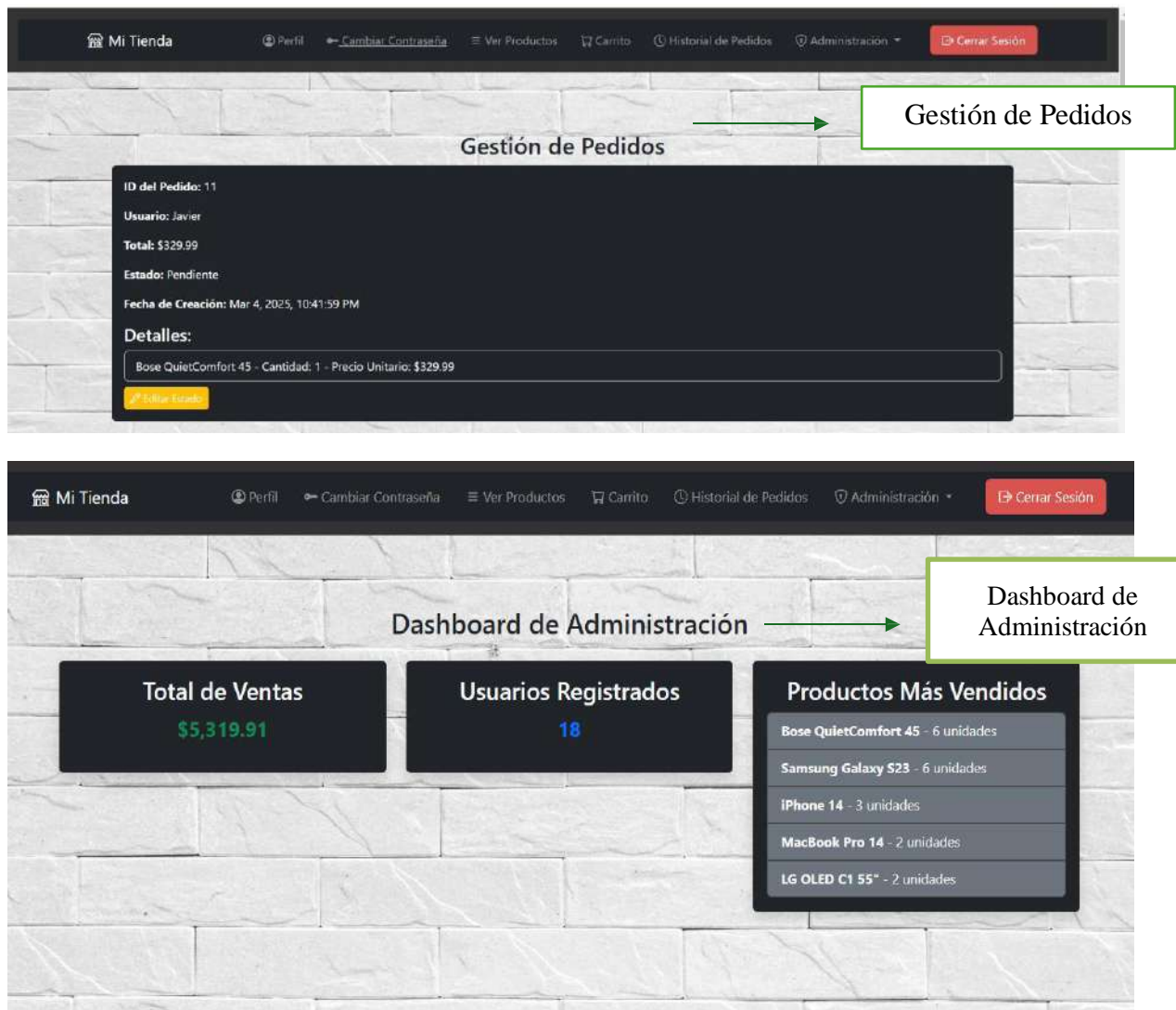
Mi Tienda

PerfilCambiar ContraseñaVer ProductosCarritoHistorial de PedidosAdministraciónCerrar Sesión

Gestión de Usuarios

ID	Nombre	Correo	Rol	Estado	Acciones	
7	Ana García	ana@example.com	cliente	Activo	Bloquear	Eliminar
22	Anthonsito	anthonyvm123@hotmail.es	cliente	Activo	Bloquear	Eliminar
9	Anthony Villarreal	anvillarreal@espe.edu.ec	admin	Activo	Bloquear	Eliminar
21	Antonio	antonio@gmail.com	cliente	Activo	Bloquear	Eliminar
11	Ariel Leonidas	alreyes2@espe.edu.ec	cliente	Activo	Bloquear	Eliminar
17	Ariel Reyes	ariel@espe.edu.ec	admin	Activo	Bloquear	Eliminar
26	Deysi	villarrealdeysi@hotmail.com	cliente	Activo	Bloquear	Eliminar
18	Fernanda	fernanda@gmail.com	cliente	Activo	Bloquear	Eliminar

Gestión de Usuarios



Se desarrolló una aplicación web utilizando Node.js para la gestión del backend con una base de datos MySQL, y Angular para la interfaz del frontend. La plataforma permite a los clientes realizar compras, visualizar los productos disponibles y gestionar los elementos en su carrito de compras. La API implementa operaciones CRUD (crear, leer, actualizar y eliminar), junto con funcionalidades como autenticación de usuarios, historial de pedidos y gestión de stock. Se empleó JavaScript con programación asíncrona para optimizar el rendimiento y garantizar una experiencia fluida. Gracias a Angular, los clientes pueden interactuar con una interfaz dinámica e intuitiva, facilitando la navegación y el proceso de compra. Este sistema ofrece una solución escalable y eficiente para la gestión de ventas en línea.

8. Conclusiones

- La implementación de esta API en Node.js facilita la gestión de usuarios en una base de datos MySQL, permitiendo operaciones CRUD de manera eficiente.
- La modularización del código permite escalabilidad y fácil mantenimiento, optimizando la integración con otros sistemas.

- La organización del código en una clase modular permite reutilización y facilita futuras ampliaciones sin afectar la estructura principal del sistema.

9. Recomendaciones

- Implementar medidas de seguridad como cifrado de contraseñas y validación de datos para proteger la información de los usuarios.
- Agregar control de acceso con autenticación y roles para restringir acciones según los permisos del usuario.
- Optimizar el manejo de errores y respuestas HTTP para mejorar la experiencia y la depuración del sistema.

10. Bibliografía o Referencias

- Flanagan, D. (2020). JavaScript: The Definitive Guide. Sebastopol, CA: O'Reilly Media.

11. Link del git hub

<https://github.com/ANTHONYNESTORVILLARREALMACIAS/Desarrollo-Web-Avanzado-AnthonyV-ArielR.git>

