

# Documentación programación Web

**VerdeVital**

## **Integrantes:**

Marcelo Contreras

Kameron Rivera

Benjamín Alcayaga

**Fecha:** 10/06/2025

**Profesor:** Carlos Zepeda

# Documentación creación de archivos e implementación



## Backend: Estructura general

▼ backend	#Controladores con la lógica del Backend
▼ controllers	
JS loginController.js	
JS productoController.js	#Configuración de la base de datos SQLite
▼ db	
JS database.js	
≡ verdevital.sqbpro	
verdevital.sqlite	#Middleware para autenticación y permisos
▼ middleware	
JS authMiddleware.js	
▼ models	#Modelo para manejar datos
JS compraModel.js	
JS productoModel.js	
JS usuarioModel.js	#Rutas API (productos, login)
> node_modules	
▼ routes	
JS loginRoutes.js	
JS productoRoutes.js	#Funcionalidades auxiliares (Telegram)
▼ utils	
JS telegram.js	
{ } package-lock.json	
{ } package.json	
JS server.js	#Punto de entrada del servidor

## Base de datos:

verdevital.sqlite

- Lo primero fue implementar la base de datos por medio de SQLite, se implementaron las tablas necesarias para la implementación de requerimientos, inicialmente se generaron INSERT INTO durante la creación a la tabla productos, con lo que se creo un usuario admin y normal.

	id	nombre	correo	contrasena	rol
1	1	Admin	admin@gmail.com	admin123	admin
2	2	Goku	goku@gmail.com	12345	cliente
+	3				

## controllers/loginController.js → Manejo de autenticación

- Tras la creación de los usuarios, creamos login controller para poder validar las credenciales insertada por medio del inicio de sesión, validamos los warnings por medio de comparaciones y llamamos a los datos de usuario desde la base de datos para verificar que los datos sean correctos, por ultimo implementamos "bcrypt" para que al registrar usuarios, la contraseña se guarde encriptada.

```
const bcrypt = require('bcrypt');
const db = require('../db/database');
const jwt = require('jsonwebtoken');

const login = (req, res) => {
  const { correo, contraseña } = req.body;

  if (!correo || !contraseña) {
    return res.status(400).json({ mensaje: 'Correo y contraseña requeridos' });
  }

  const sql = 'SELECT * FROM usuarios WHERE correo = ?';
  db.get(sql, [correo], async (err, usuario) => {
    if (err) return res.status(500).json({ mensaje: 'Error en la base de datos' });

    if (!usuario || !(await bcrypt.compare(contraseña, usuario.contrasena))) {
      return res.status(401).json({ mensaje: 'Credenciales incorrectas' });
    }

    // Crear token JWT
    const token = jwt.sign({ id: usuario.id, rol: usuario.rol }, 'secreto', { expiresIn: '2h' });

    res.json({
      mensaje: 'Login exitoso',
      usuario: { id: usuario.id, nombre: usuario.nombre, correo: usuario.correo, rol: usuario.rol },
      token
    });
  });
};

module.exports = { login };
```

### Propósito:

### Implementación:

- Recibe el correo y contraseña desde el frontend.
- Consulta la tabla usuarios en SQLite.
- Si es válido, genera un token JWT con jsonwebtoken.
- Retorna los datos del usuario y el token.

### Desafío:

- Manejar errores de autenticación correctamente. Se resolvió con respuestas 400 y 401 según el caso.

## controllers/productoController.js → Gestión de productos y compras

- Con este controlador, nos encargamos de gestionar el inventario de los productos, y conectar con compraModel para gestionar los envíos de mensajes de compra hacia Telegram, estos pasos fueron importantísimos para la gestión del carrito y la gestión de inventario por medio del usuario administrador.

```
const {
  obtenerProductos,
  agregarProducto,
  actualizarProducto,
  eliminarProducto
} = require('../models/productoModel');
const { notificarPedidoTelegram } = require('../utils/telegram'); // ✅ Importar función para enviar mensajes a Telegram
const { procesarCompra } = require('../models/compraModel'); // ✅ Asegurar que el backend procese compras

// Obtener todos los productos
const getProductos = async (req, res) => {
  try {
    console.log('GET /api/productos llamado');
    const productos = await obtenerProductos();
    res.json(productos);
  } catch (error) {
    console.error('Error en getProductos:', error);
    res.status(500).json({ error: 'Error al obtener productos' });
  }
};

// Agregar un nuevo producto
const postProducto = async (req, res) => {
  try {
    const { nombre, precio, stock, descripcion, categoria } = req.body;

    if (!nombre || isNaN(precio) || isNaN(stock) || !descripcion || !categoria) {
      return res.status(400).json({ mensaje: 'Todos los campos son obligatorios' });
    }

    const nuevoProducto = await agregarProducto({ nombre, precio, stock, descripcion, categoria });
    res.status(201).json(nuevoProducto);
  } catch (error) {
    console.error('Error en postProducto:', error);
    res.status(500).json({ error: 'Error al agregar producto' });
  }
};

// Actualizar un producto
const putProducto = async (req, res) => {
  try {
    const { id } = req.params;
    const { nombre, precio, stock, descripcion, categoria } = req.body;

    if (!nombre || isNaN(precio) || isNaN(stock) || !descripcion || !categoria) {
      return res.status(400).json({ mensaje: 'Todos los campos son obligatorios' });
    }

    const productoActualizado = await actualizarProducto(id, { nombre, precio, stock, descripcion, categoria });
    res.json(productoActualizado);
  } catch (error) {
    console.error('Error en putProducto:', error);
    res.status(500).json({ error: 'Error al actualizar producto' });
  }
};
```

### Propósito:

- Para mejorar las funciones CRUD de productos y el proceso de compra.

### Implementación:

- getProductos() → Obtiene todos los productos.

- postProducto() → Agrega nuevos productos (solo admin).
- putProducto() → Actualiza productos existentes.
- deleteProducto() → Elimina productos existentes.
- postCompra() → Registra una compra y notifica por Telegram/token personal).

### Desafío:

- Validar los datos antes de almacenarlos. Se solucionó con verificaciones isNaN() y control de errores.

### db/database.js → Configuración de SQLite Propósito:

- Establece la conexión con SQLite, nuestra base de datos "verdevital.sqlite".

```

and > db > JS database.js > ...
const sqlite3 = require('sqlite3').verbose();
const db = new sqlite3.Database('./db/verdevital.sqlite');

module.exports = db;

```

### Implementación:

- Usa sqlite3.verbose() para manejar la base de datos verdevital.sqlite.
- Centraliza la conexión para ser utilizada en **models** y **controllers**.

### Desafío:

- Quisimos que la base de datos esté accesible en todas las partes del backend. Se solucionó importando database.js en todos los archivos que lo necesitan.

## middleware/authMiddleware.js → Autenticación de administradores

- Se encarga de mejorar la seguridad y control de accesos no autorizados, centralizando la lógica de seguridad y evitando la repetición de código.

```
const jwt = require('jsonwebtoken');

const verificarAdmin = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];

  if (!token) {
    return res.status(403).json({ mensaje: 'Acceso denegado' });
  }

  try {
    const decoded = jwt.verify(token, 'secreto');
    if (decoded.rol !== 'admin') {
      return res.status(403).json({ mensaje: 'Acceso solo para administradores' });
    }
    next();
  } catch (error) {
    res.status(403).json({ mensaje: 'Token inválido' });
  }
};

module.exports = { verificarAdmin };
```

### Propósito:

- Restringe ciertas acciones solo a administradores verificando su JWT.

### Implementación:

- Extrae el token de Authorization en la cabecera.
- Lo valida con jwt.verify().
- Permite el acceso solo si rol === 'admin'.

### Desafío:

- Evitar acceso no autorizado. Se manejó con respuestas 403 y validaciones de token.

## models/compraModel.js → Manejo de compras

- Para poder gestionar el proceso de compra en el backend, integramos este modelo para así asegurarnos que los productos adquiridos sean registrados en la base de datos, se encarga también de reducir el stock de cada producto que es comprado.

```
back&gt; models > .js compraModel.js > @ procesarCompra > <function> > db.run() callback > then() callback
3 const procesarCompra = async (productos, usuario_id) => {
4   return new Promise((resolve, reject) => {
5     return reject({ exito: false, mensaje: 'El carrito esta vacio' });
6   })
7 }
8
9 const sqlPedido = 'INSERT INTO pedidos (usuario_id, fecha, estado, total) VALUES (?, ?, ?, ?)';
10 const fecha = new Date().toISOString();
11 const estado = 'pendiente';
12 const total = productos.reduce((acc, producto) => acc + (producto.precio * producto.cantidad), 0);
13
14 db.run(sqlPedido, [usuario_id, fecha, estado, total], function (err) {
15   if (err) {
16     console.error('Error al registrar pedido:', err);
17     return reject({ exito: false, mensaje: 'Error al registrar pedido' });
18   }
19 }
20
21 const pedido_id = this.lastID;
22
23 const registrarItemsPromises = productos.map(producto => {
24   return new Promise((res, rej) => {
25     const sqlItem = 'INSERT INTO pedido_items (pedido_id, producto_id, cantidad, precio_unitario) VALUES (?, ?, ?, ?)';
26     db.run(sqlItem, [pedido_id, producto.id, producto.cantidad, producto.precio], (err) => {
27       if (err) {
28         console.error('Error al registrar item del pedido:', err);
29         return rej('Error al registrar item del pedido');
30       }
31       res();
32     });
33   });
34 });
35
36 Promise.all(registrarItemsPromises)
37   .then(() => {
38     // ♦ Reducir stock después de registrar el pedido
39     const actualizarStockPromises = productos.map(producto => {
40       return new Promise((res, rej) => {
41         db.run('UPDATE productos SET stock = stock - ? WHERE id = ?', [producto.cantidad, producto.id], (err) => {
42           if (err) {
43             console.error('Error al actualizar stock:', err);
44             return rej('Error al actualizar stock');
45           }
46           res();
47         });
48       });
49     });
50     return Promise.all(actualizarStockPromises);
51   })
52   .then(() => {
53     console.log('✅ Pedido y stock actualizados con éxito.');
```

### Propósito:

Guarda las compras en la base de datos y actualiza el stock.

### Implementación:

- Reduce el stock al comprar.
- Registra el pedido en compras.
- Convierte los datos a JSON antes de almacenarlos.

### Desafío:



- Evitar problemas de concurrencia en la actualización de stock. Se implementó un manejo de errores con reject().

## models/productoModel.js → Gestión de productos

- así, continuamos con la implementación de este modelo, que se conecta a la base de datos para realizar operaciones CRUD sobre la tabla productos

```
const db = require('../db/database');

// Obtener todos los productos
const obtenerProductos = () => {
  return new Promise((resolve, reject) => {
    db.all('SELECT id, nombre, precio, stock, descripcion, categoria FROM productos', (err, rows) => {
      if (err) {
        console.error('Error al consultar productos:', err);
        reject(err);
      } else {
        console.log('Consulta de productos exitosa. Total:', rows.length);
        resolve(rows);
      }
    });
  });
};

// Agregar nuevo producto
const agregarProducto = ({ nombre, precio, stock, descripcion, categoria }) => {
  return new Promise((resolve, reject) => {
    const sql = 'INSERT INTO productos (nombre, precio, stock, descripcion, categoria) VALUES (?, ?, ?, ?, ?)';
    db.run(sql, [nombre, precio, stock, descripcion, categoria], function (err) {
      if (err) reject(err);
      else resolve({ id: this.lastID, nombre, precio, stock, descripcion, categoria });
    });
  });
};

// Actualizar producto
const actualizarProducto = (id, { nombre, precio, stock, descripcion, categoria }) => {
  return new Promise((resolve, reject) => {
    const sql = 'UPDATE productos SET nombre = ?, precio = ?, stock = ?, descripcion = ?, categoria = ? WHERE id = ?';
    db.run(sql, [nombre, precio, stock, descripcion, categoria, id], function (err) {
      if (err) reject(err);
      else resolve({ id, nombre, precio, stock, descripcion, categoria });
    });
  });
};

// Eliminar producto
const eliminarProducto = (id) => {
  return new Promise((resolve, reject) => {
    const sql = 'DELETE FROM productos WHERE id = ?';
    db.run(sql, [id], function (err) {
      if (err) reject(err);
      else resolve({ eliminado: true });
    });
  });
};

module.exports = {
  obtenerProductos,
  agregarProducto,
  actualizarProducto,
  eliminarProducto
};
```

## Propósito:

- Maneja la lógica de productos en la base de datos.

## Implementación:

- obtenerProductos() → Obtiene productos.
- agregarProducto() → Inserta un nuevo producto.
- actualizarProducto() → Modifica un producto existente.
- eliminarProducto() → Borra un producto.

## Desafío:

- Mejorar la estructura de consultas para evitar errores SQL. Se utilizó Promise para mayor control.

## models/usuarioModel.js → Manejo de usuarios

- Para gestionar los usuarios en la base de datos, creamos este controlador el cual se encarga de validar las credenciales para iniciar sesión.

```
const db = require('../db/database');

const buscarPorCredenciales = (correo, contrasena, callback) => {
  const sql = 'SELECT * FROM usuarios WHERE correo = ? AND contrasena = ?';
  db.get(sql, [correo, contrasena], callback);
};

module.exports = { buscarPorCredenciales };
```

## Propósito:

- Maneja la autenticación de usuarios en la base de datos. Implementación:
- buscarPorCredenciales() → Busca usuarios con correo y contraseña.

## Desafío:

- Optimizar la consulta para mayor seguridad. Se mejoró con parámetros preparados.

## routes/loginRoutes.js → Rutas de autenticación

- Creamos este controlador para gestionar las rutas relacionadas con la autenticación de usuarios, específicamente el inicio de sesión

```
const express = require('express');
const router = express.Router();
const { login } = require('../controllers/loginController');

// Ruta POST para login
router.post('/login', login);

module.exports = router;
```

## Propósito:

- Define la ruta para login (POST /login).

## Implementación:

- Recibe las credenciales y llama a loginController.js.

## Desafío:

- Hacer la ruta accesible desde el frontend. Se solucionó con cors().

## routes/productoRoutes.js → Rutas de productos

- Lo siguiente también fue crear este controlador el cual se encarga de definir las rutas relacionadas con la gestión de productos, asegurando que solo los Administradores puedan modificar, agregar o eliminar productos en la base de datos.

```
1  const express = require('express');
2  const router = express.Router();
3  const { verificarAdmin } = require('../middleware/authMiddleware');
4  const {
5    getProductos,
6    postProducto,
7    putProducto,
8    deleteProducto,
9    postCompra // ♦ Ahora representa pedidos
10 } = require('../controllers/productoController');
11
12 router.get('/productos', getProductos); // ♦ Obtener productos
13 router.post('/productos', verificarAdmin, postProducto); // ♦ Agregar productos (solo admin)
14 router.put('/productos/:id', verificarAdmin, putProducto); // ♦ Modificar productos (solo admin)
15 router.delete('/productos/:id', verificarAdmin, deleteProducto); // ♦ Eliminar productos (solo admin)
16
17 // ♦ Agregar la ruta para procesar pedidos en lugar de compras
18 router.post('/pedidos', postCompra);
19
20 module.exports = router;
21 |
```

## Propósito:

- Define las rutas de gestión de productos (GET, POST, PUT, DELETE).

## Implementación:

- Solo admin pueden agregar, modificar y eliminar productos.

## Desafío:

- Validar permisos correctamente. Se resolvió con authMiddleware.js.

## utils/telegram.js → Envío de mensajes a Telegram

- La implementación de este Telegram.js (El cual se encarga de definir el envío de mensajes a Telegram) fue principalmente para poder implementar el envío de mensajes al momento de generar una compra/pedido dentro del carrito, los mensajes constan con la información del producto comprado.

```
const fetch = require('node-fetch');
const TELEGRAM_TOKEN = ''; // ♦ Reemplaza con tu token de bot
const CHAT_ID = ''; // ♦ Reemplaza con tu chat ID

exports.notificarPedidoTelegram = async (productos, usuario_id) => {
  let mensaje = `🛒 *Nuevo pedido de usuario ${usuario_id || 'anónimo'}:*` + '\n';
  productos.forEach(p => {
    mensaje += `♦ *${p.nombre}* x${p.cantidad} (${p.precio * p.cantidad})` + '\n';
  });

  try {
    const res = await fetch(`https://api.telegram.org/bot${TELEGRAM_TOKEN}/sendMessage`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ chat_id: CHAT_ID, text: mensaje, parse_mode: 'Markdown' }) // ✅ Mensaje con formato Markdown
    });

    const data = await res.json();
    if (!res.ok) {
      throw new Error(`Error en Telegram: ${data.description}`);
    }
    console.log('✅ Pedido enviado a Telegram');
  } catch (error) {
    console.error('❌ Error al enviar a Telegram:', error);
  }
};
```

### Propósito:

- Notifica compras a Telegram.

### Implementación:

- fetch() envía un mensaje al bot.

### Desafío:

- Hacer que el mensaje sea claro y estructurado. Se solucionó con Markdown.

## server.js → Punto de entrada del backend

- y bueno, este es el punto de entrada del backend, server.js sirve para gestionar las rutas, se maneja la comunicación con Telegram al realizar una compra/pedido.

```
const express = require('express');
const cors = require('cors');
const app = express();
const telegram = require('./utils/telegram');

// Rutas
const loginRoutes = require('./routes/loginRoutes');
const productoRoutes = require('./routes/productoRoutes');

app.use(cors());
app.use(express.json());

app.use('/api', loginRoutes);
app.use('/api', productoRoutes);

// Ruta para comprar productos y notificar por Telegram
app.post('/api/pedidos', async (req, res) => {
  const { productos, usuario } = req.body;

  if (!productos || !Array.isArray(productos) || productos.length === 0) {
    return res.status(400).json({ mensaje: 'No hay productos en el carrito' });
  }

  try {
    // ✅ Registrar el pedido en la base de datos y reducir el stock
    const resultadoPedido = await procesarCompra(productos, usuario);
    if (!resultadoPedido.exito) {
      return res.status(500).json({ mensaje: resultadoPedido.mensaje });
    }

    console.log('✅ Pedido registrado correctamente.');
```

## Propósito:

- Configura el servidor Express y registra las rutas.

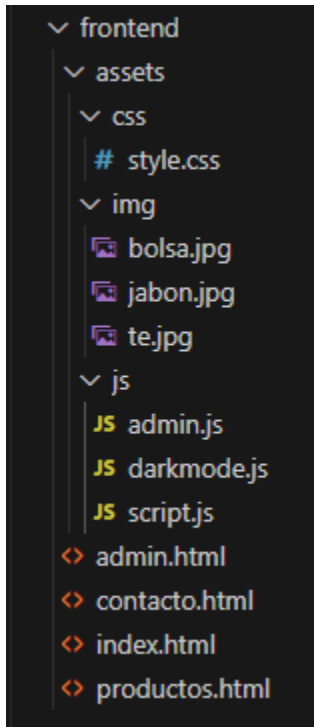
## Implementación:

- Usa cors() y express.json().
- Importa loginRoutes y productoRoutes.
- Maneja la compra y notificación a Telegram.

**Desafío:**

- Hacer que el backend sea accesible desde el frontend. Se resolvió con cors().

## Frontend: Estructura general



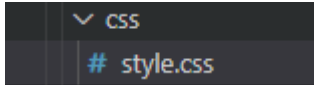
**#Estilos visuales para la pagina**

**#Imagenes para los productos**

**#Logica e interactividad de la pagina**

**#Paginas principales del sitio**

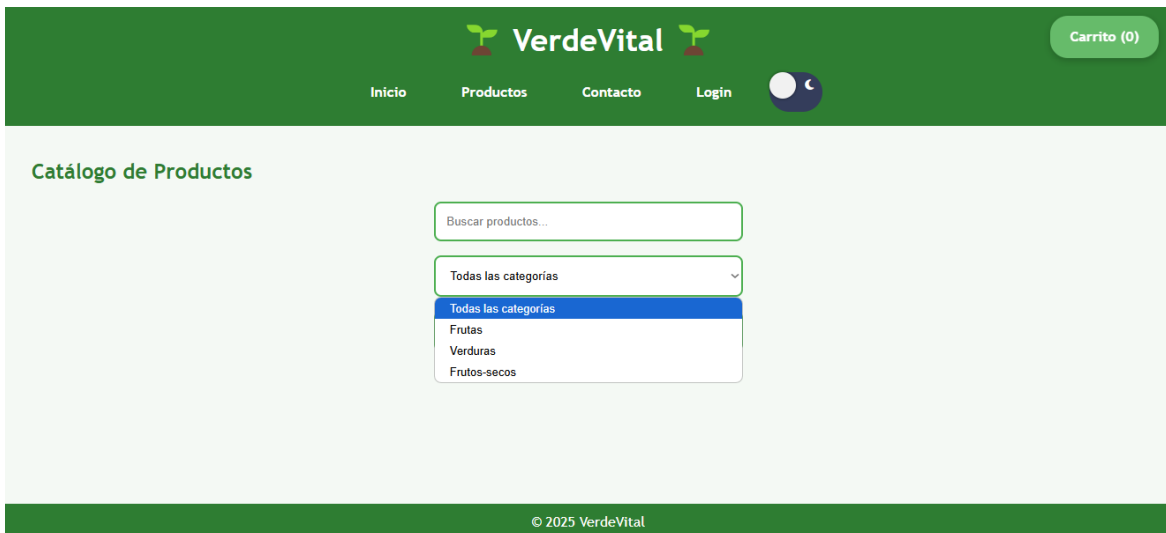
## Assets/css/style.css:



## Propósito:

- Define la apariencia visual de la pagina para que sea atractiva y responsiva. Como por ejemplo en la siguiente regla css:
- En la que se aplican estilos a los elementos del filtro de productos como el campo de búsqueda, los selectores de categoría y precio:

```
# style.css  X
frontend > assets > css > # style.css > ...
222
223  /* ----- Filtros de Productos ----- */
224  input#buscarProducto, select#filtrarCategoria, select#filtrarPrecio {
225      width: 100%;
226      max-width: 350px;
227      padding: 0.8rem;
228      margin: 1rem auto;
229      display: block;
230      border-radius: 8px;
231      border: 2px solid #4caf50;
232  }
```





## Assets/js/admin.js:

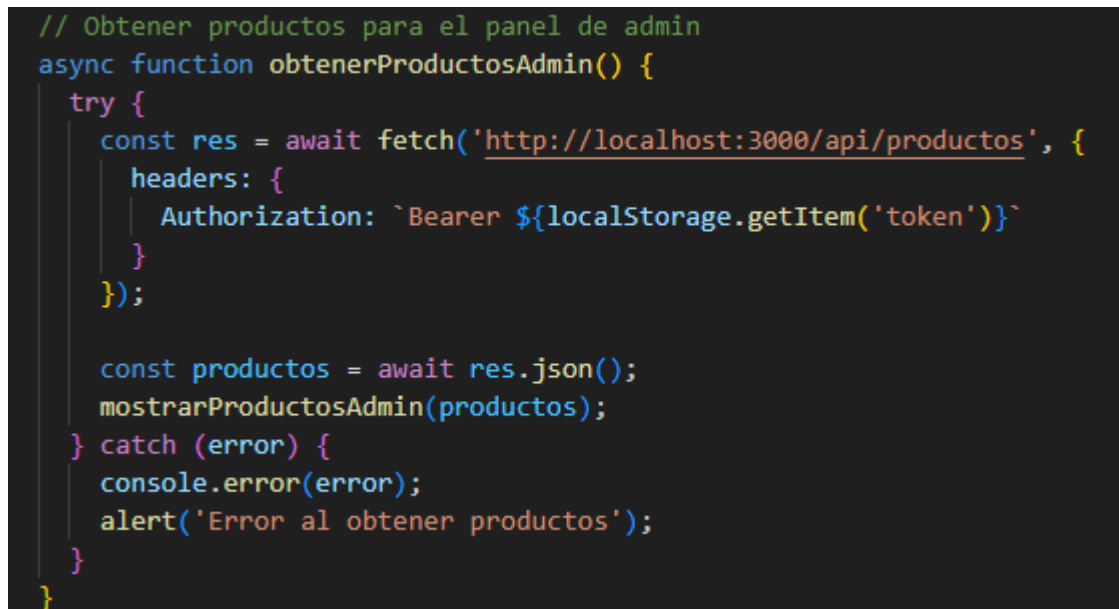
```
JS admin.js x
frontend > assets > js > JS admin.js > ...
1  const usuarioActual = JSON.parse(localStorage.getItem('usuario'));
2
3  if (!usuarioActual || usuarioActual.rol !== 'admin') {
4    alert('Acceso denegado. Esta página es solo para administradores.');
```



```
5    window.location.href = 'index.html';
6  }
7
8  window.onload = () => {
9    obtenerProductosAdmin();
10    document.getElementById('formAgregarProducto').addEventListener('submit', agregarProducto);
11  };
12
```

- Este método verifica si el usuario actual es administrador. Si no lo es, se muestra una alerta y se redirige a la página principal (index.html).
- En window.onload al cargar la página se llama a obtenerProductosAdmin() para mostrar los productos en el panel y se agrega un listener al formulario de agregar producto para manejar el evento de envío.

```
// Obtener productos para el panel de admin
async function obtenerProductosAdmin() {
  try {
    const res = await fetch('http://localhost:3000/api/productos', {
      headers: {
        Authorization: `Bearer ${localStorage.getItem('token')}`
      }
    });
    const productos = await res.json();
    mostrarProductosAdmin(productos);
  } catch (error) {
    console.error(error);
    alert('Error al obtener productos');
  }
}
```



- Este método hace una petición GET a la API para obtener la lista de productos. Si tiene éxito, llama a [mostrarProductosAdmin\(productos\)](#) para mostrar los productos en la tabla. Si hay error, muestra una alerta "Error al obtener productos".

```

30 // Mostrar productos en tabla con inputs editables
31 function mostrarProductosAdmin(productos) {
32     const tbody = document.querySelector('#tablaAdmin tbody');
33     tbody.innerHTML = '';
34
35     productos.forEach(p => {
36         const fila = document.createElement('tr');
37         fila.innerHTML = `
38             <td><input value="${p.nombre}" data-id="${p.id}" data-campo="nombre"/></td>
39             <td><input type="number" value="${p.precio}" data-id="${p.id}" data-campo="precio"/></td>
40             <td><input type="number" value="${p.stock}" data-id="${p.id}" data-campo="stock"/></td>
41             <td><input value="${p.categoria}" data-id="${p.id}" data-campo="categoria"/></td>
42             <td><input value="${p.descripcion}" data-id="${p.id}" data-campo="descripcion"/></td>
43             <td>
44                 <button onclick="actualizarProducto(${p.id})">Actualizar</button>
45                 <button onclick="eliminarProducto(${p.id})">Eliminar</button>
46             </td>
47         `;
48         tbody.appendChild(fila);
49     });
50 }

```

- Este método recibe un array de productos y limpia el contenido de la tabla. Por cada producto crea una fila con inputs editables y botones para actualizar o eliminar.

```

53 async function agregarProducto(e) {
54     e.preventDefault();
55     const nombre = document.getElementById('nombre').value.trim();
56     const precio = parseFloat(document.getElementById('precio').value);
57     const stock = parseInt(document.getElementById('stock').value);
58     const descripcion = document.getElementById('descripcion').value.trim();
59     const categoria = document.getElementById('categoria').value;
60
61     if (!nombre || isNaN(precio) || precio < 0 || isNaN(stock) || stock < 0 || !descripcion || !categoria) {
62         return alert('Por favor completa los campos correctamente');
63     }
64     try {
65         const res = await fetch('http://localhost:3000/api/productos', {
66             method: 'POST',
67             headers: {
68                 'Content-Type': 'application/json',
69                 Authorization: `Bearer ${localStorage.getItem('token')}`
70             },
71             body: JSON.stringify({ nombre, precio, stock, descripcion, categoria })
72         });
73
74         const data = await res.json();
75         if (res.ok) {
76             alert('Producto agregado correctamente');
77             e.target.reset();
78             obtenerProductosAdmin();
79         } else {
80             alert(data.mensaje || 'Error al agregar el producto');
81         }
82     } catch (error) {
83         console.error(error);
84         alert('Error de red al agregar producto');
85     }
86 }

```

- agregarProducto(e) maneja el envío del formulario para agregar un nuevo producto:

- Valida los campos del formulario.
- Envía una petición POST a la API con los datos del nuevo producto.
- Si tiene éxito, muestra un mensaje, limpia el formulario y actualiza la tabla. Si hay error, muestra una alerta.

```

JS admin.js
frontend > assets > js > JS admin.js > agregarProducto

89  async function actualizarProducto(id) {
90    const inputs = document.querySelectorAll(`input[data-id="${id}"]`);
91    const datos = {};
92
93    inputs.forEach(input => {
94      const valor = input.value.trim();
95      datos[input.dataset.campo] = input.type === 'number' ? parseFloat(valor) : valor;
96    });
97
98    if (!datos.nombre || isNaN(datos.precio) || isNaN(datos.stock)) {
99      return alert('Datos inválidos al actualizar');
100    }
101
102    try {
103      const res = await fetch(`http://localhost:3000/api/productos/${id}`, {
104        method: 'PUT',
105        headers: {
106          'Content-Type': 'application/json',
107          Authorization: `Bearer ${localStorage.getItem('token')}`
108        },
109        body: JSON.stringify(datos)
110      });
111
112      const data = await res.json();
113      if (res.ok) {
114        alert('Producto actualizado correctamente');
115        obtenerProductosAdmin();
116      } else {
117        alert(data.mensaje || 'Error al actualizar producto');
118      }
119    } catch (error) {

```

- actualizarProducto(id) permite actualizar un producto existente:
  - Obtiene los valores de los inputs de la fila correspondiente.
  - Valida los datos.
  - Envía una petición PUT a la API con los datos actualizados.
  - Si tiene éxito, muestra un mensaje y actualiza la tabla.

```

125 // Eliminar producto
126 async function eliminarProducto(id) {
127   if (!confirm('¿Estás seguro de eliminar este producto?')) return;
128
129   try {
130     const res = await fetch(`http://localhost:3000/api/productos/${id}`, {
131       method: 'DELETE',
132       headers: {
133         Authorization: `Bearer ${localStorage.getItem('token')}`
134       }
135     });
136
137     const data = await res.json();
138     if (res.ok) {
139       alert('Producto eliminado correctamente');
140       obtenerProductosAdmin();
141     } else {
142       alert(data.mensaje || 'Error al eliminar producto');
143     }
144   } catch (error) {
145     console.error(error);
146     alert('Error de red al eliminar producto');
147   }
148 }

```

- eliminarProducto(id) permite eliminar un producto:
  - Pide confirmación al usuario.
  - Si acepta, envía una petición DELETE a la API.
  - Si tiene éxito, muestra un mensaje y actualiza la tabla. Si no, un mensaje de error.

## Assets/js/darkmode.js:

```
JS darkmode.js X
frontend > assets > js > JS darkmode.js > document.addEventListener('DOMContentLoaded')
1 document.addEventListener('DOMContentLoaded', () => {
2   const btnSwitch = document.querySelector('#switch');
3   const iconoModo = document.querySelector('#icono-modo');
4 }
```

- Al cargar la página (DOMContentLoaded), obtiene los elementos del botón de cambio de modo (#switch) y el icono (#icono-modo).

```
JS darkmode.js X
frontend > assets > js > JS darkmode.js > document.addEventListener('DOMContentLoaded') callback
1 document.addEventListener('DOMContentLoaded', () => {
5   // Restaurar estado previo
6   const modoOscuro = localStorage.getItem('modo') === 'oscuro';
7
8   if (modoOscuro) {
9     document.body.classList.add('dark');
10    iconoModo.classList.remove('fa-sun');
11    iconoModo.classList.add('fa-moon');
12    btnSwitch.classList.add('active');
13  } else {
14    iconoModo.classList.remove('fa-moon');
15    iconoModo.classList.add('fa-sun');
16    btnSwitch.classList.remove('active');
17  }
18 }
```

- Lee del [localStorage](#) si el usuario tenía activado el modo oscuro anteriormente.
  - Si estaba activado, agrega la clase dark al <body>, cambia el icono a luna y activa el botón. Si no muestra el icono de sol y desactiva el botón.



```

18
19 btnSwitch.addEventListener('click', () => {
20   document.body.classList.toggle('dark');
21   const esOscuro = document.body.classList.contains('dark');
22
23   if (esOscuro) {
24     iconoModo.classList.remove('fa-sun');
25     iconoModo.classList.add('fa-moon');
26     localStorage.setItem('modo', 'oscuro');
27     btnSwitch.classList.add('active');
28   } else {
29     iconoModo.classList.remove('fa-moon');
30     iconoModo.classList.add('fa-sun');
31     localStorage.setItem('modo', 'claro');
32     btnSwitch.classList.remove('active');
33   }
34 });
35 });
36

```

- Al hacer clic en el botón de modo:
  - Alterna la clase dark en el <body>.
  - Cambia el icono entre sol y luna.
  - Guarda la preferencia actual (oscuro o claro) en el [localStorage](#).
  - Cambia el estado visual del botón.

## Assets/js/script.js:

```

JS script.js  X
frontend > assets > js > JS script.js > document.addEventListener('DOMContentLoaded') callback
1  document.addEventListener('DOMContentLoaded', async () => {
2    const loginModal = document.getElementById('modalLogin');
3    const btnLogin = document.getElementById('btnLogin');
4    const cerrarLogin = document.getElementById('cerrarLogin');
5    const formLogin = document.getElementById('formLogin');
6    const mensajeError = document.getElementById('mensajeError');
7    const nav = document.querySelector('nav');
8    let usuario = JSON.parse(localStorage.getItem('usuario')) || null;
9

```

- Esta sección obtiene referencias de varios elementos de la interfaz relacionados con el inicio de sesión: el modal de login, el botón para abrirlo, el botón para cerrarlo, el formulario de login, el área de mensajes de error y la barra de navegación. Además, intenta recuperar la información del usuario almacenada en el navegador para saber si ya hay una sesión activa.

```

10 // Si es admin, mostrar "Inventario"
11 if (usuario?.rol === 'admin') {
12   const inventarioLink = document.createElement('a');
13   inventarioLink.href = 'admin.html';
14   inventarioLink.textContent = 'Inventario';
15   nav.appendChild(inventarioLink);
16 }
17
18 // Mostrar el botón de cerrar sesión si el usuario está logueado
19 const btnLogout = document.getElementById('btnLogout');
20 if (usuario && btnLogout) {
21   btnLogout.style.display = 'inline-block';
22 }
23
24 // Cerrar sesión
25 btnLogout?.addEventListener('click', () => {
26   localStorage.removeItem('usuario');
27   localStorage.removeItem('token');
28   window.location.href = 'index.html';
29 });

```

- Verifica si hay un usuario logueado y si es administrador, muestra el enlace al panel de inventario. Permite cerrar sesión, eliminando los datos del usuario y el token del localStorage.

```

31 // Mostrar modal de login al hacer clic en el botón
32 btnLogin?.addEventListener('click', () => {
33   loginModal.classList.add('activo');
34 });
35
36 // Cerrar modal de login
37 cerrarLogin?.addEventListener('click', () => {
38   loginModal.classList.remove('activo');
39 });
40
41 // Inicio de sesión
42 if (formLogin) {
43   formLogin.addEventListener('submit', async (e) => {
44     e.preventDefault();
45     const correo = e.target.loginEmail.value;
46     const contrasena = e.target.loginPassword.value;
47     if (mensajeError) mensajeError.textContent = '';
48
49     try {
50       const res = await fetch('http://localhost:3000/api/login', {
51         method: 'POST',
52         headers: { 'Content-Type': 'application/json' },
53         body: JSON.stringify({ correo, contrasena })
54       });
55
56       loginModal.classList.remove('activo');
57       usuario.rol === 'admin' ? window.location.href = 'admin.html' : alert('Bienvenido');
58     } else {
59       mensajeError.textContent = data.mensaje || 'Error al iniciar sesión';
60     }
61   } catch (err) {
62     mensajeError.textContent = 'Error en el servidor';
63     console.error('Error en fetch login:', err);
64   }
65 });
66
67
68
69
70
71
72

```

- Esta sección muestra y oculta el modal de inicio de sesión
- Envía los datos del formulario de login a la API y, si son correctos, guarda el usuario y token en [localStorage](#).
- Si el usuario es admin, lo redirige al panel de administración.

```

74 // --- Carrito ---
75 const productosDiv = document.getElementById('productos');
76 const carritoIcono = document.getElementById('abrirCarrito');
77 const modalCarrito = document.getElementById('modalCarrito');
78 const cerrarCarrito = document.getElementById('cerrarCarrito');
79 const contenidoCarrito = document.getElementById('contenidoCarrito');
80 const cantidadCarrito = document.getElementById('cantidadCarrito');
81 const comprarCarrito = document.getElementById('comprarCarrito');
82 const vaciarCarrito = document.getElementById('vaciarCarrito');
83 let carrito = JSON.parse(localStorage.getItem('carrito')) || [];
84
85 console.log('🛒 Carrito cargado desde localStorage:', localStorage.getItem('carrito'));
86 console.log('Objeto carrito:', carrito);
87

```

- Esta parte del código carga el carrito desde [localStorage](#) y lo mantiene actualizado.
- Permite agregar productos al carrito, aumentando la cantidad si ya existe y respetando el stock.
- Permite eliminar productos del carrito.
- Muestra el contenido del carrito en un modal.
- Permite comprar (envía los productos y usuario a la API) y vaciar el carrito.



```

88 // Mostrar productos
89 if (productosDiv) {
90   fetch('http://localhost:3000/api/productos')
91     .then(res => {
92       if (!res.ok) throw new Error('No se pudo obtener productos');
93       return res.json();
94     })
95     .then(productos => {
96       productosDiv.innerHTML = '';
97       if (!productos.length) {
98         productosDiv.innerHTML = '<p>No hay productos disponibles.</p>';
99         return;
100       }
101       productos.forEach(p => {
102         const div = document.createElement('div');
103         div.className = 'producto';
104         div.innerHTML = `
105           <h3>${p.nombre}</h3>
106           <p>${p.precio} | Stock: ${p.stock}</p>
107           <button data-id="${p.id}">Agregar al carrito</button>
108         `;
109         div.querySelector('button').onclick = () => agregarAlCarrito(p);
110         productosDiv.appendChild(div);
111       });
112     })
113     .catch(err => {
114       productosDiv.innerHTML = `<p style="color:red;">Error: ${err.message}</p>`;
115       console.error(err);
116     });
117

```

- Este código obtiene la lista de productos desde la API y los muestra en la página. Si el contenedor de productos existe, realiza una petición a la API; si la respuesta es exitosa, limpia el contenedor y agrega un bloque por cada producto con su nombre, precio, stock y un botón para agregarlo al carrito. Si no hay productos, muestra un mensaje indicándolo. Si ocurre un error, muestra un mensaje de error en pantalla.

```

119 function agregarAlCarrito(producto) {
120   console.log(' * Agregando producto al carrito:', producto);
121
122   const idx = carrito.findIndex(item => item.id === producto.id);
123   if (idx !== -1) {
124     if (carrito[idx].cantidad < producto.stock) {
125       carrito[idx].cantidad++;
126     } else {
127       alert('No hay más stock disponible');
128       return;
129     }
130   } else {
131     carrito.push({ id: producto.id, nombre: producto.nombre, precio: producto.precio, cantidad: 1 });
132   }
133   guardarCarrito();
134   actualizarCantidadCarrito();
135   console.log('✅ Carrito actualizado:', carrito);
136 }
137
138 function guardarCarrito() {
139   localStorage.setItem('carrito', JSON.stringify(carrito));
140 }
141
142 function actualizarCantidadCarrito() {
143   if (cantidadCarrito) cantidadCarrito.textContent = carrito.reduce((acc, item) => acc + item.cantidad, 0)
144 }

```

- La función [agregarAlCarrito](#) recibe un objeto [producto](#) y primero muestra en consola el producto que se intenta agregar. Luego busca si ese producto ya existe en el carrito usando su [id](#). Si ya está y la cantidad actual es menor al stock disponible, incrementa la cantidad. Si se intenta superar el stock, muestra una alerta y no permite agregar más unidades. Si el producto no está en el carrito, lo agrega con cantidad uno.
- Después de modificar el carrito, se llama a [guardarCarrito](#), que guarda el estado actual del carrito en el [localStorage](#) usando JSON, asegurando que los datos persistan aunque el usuario recargue la página. Luego, se llama a [actualizarCantidadCarrito](#), que suma la cantidad total de productos en el carrito y actualiza el contador visual en la interfaz (si existe el elemento correspondiente). Finalmente, se muestra en consola el estado actualizado del carrito para facilitar la depuración.

```

146 function mostrarCarrito() {
147   if (!contenidoCarrito) return;
148   if (carrito.length === 0) {
149     contenidoCarrito.innerHTML = '<p>El carrito está vacío.</p>';
150     return;
151   }
152   contenidoCarrito.innerHTML = carrito.map(item =>
153     `<div>
154       ${item.nombre} x${item.cantidad} - ${item.precio * item.cantidad}
155       <button onclick="eliminarDelCarrito(${item.id})">Eliminar</button>
156     </div>`
157   ).join('');
158 }

```

- La función `mostrarCarrito` se encarga de renderizar el contenido del carrito en el área correspondiente (`contenidoCarrito`). Si el carrito está vacío, muestra un mensaje indicándolo. Si hay productos, genera un bloque HTML para cada uno, mostrando su nombre, cantidad, precio total y un botón para eliminarlo.

```

190   setTimeout(() => { location.reload(); }, 500);
191 } else {
192   alert(data.mensaje || 'Error al procesar la compra');
193 }
194 } catch (e) {
195   alert('Error de red al comprar');
196 }
197 });
198
199 vaciarCarrito?.addEventListener('click', () => {
200   carrito = [];
201   localStorage.removeItem('carrito');
202   mostrarCarrito();
203   actualizarCantidadCarrito();
204 });
205
206 document.getElementById('cerrarSuccess').addEventListener('click', () => {
207   document.getElementById('modalSuccess').classList.remove('activo');
208 });
209
210 actualizarCantidadCarrito();

```

- La función global `eliminarDelCarrito` permite eliminar un producto específico del carrito usando su `id`. Después de filtrar el producto se actualiza el almacenamiento local, se vuelve a mostrar el carrito actualizado y se actualiza el contador de productos.

- Se agregan manejadores de eventos para la interfaz del carrito: al hacer clic en el icono del carrito, se muestra el modal con los productos actuales; al hacer clic en el botón de cerrar el modal se oculta.
- El botón de compra envía el contenido del carrito y la información del usuario al backend mediante una petición POST. Si la compra es exitosa, se limpia el carrito del almacenamiento local y se recarga la página tras un breve retraso. Si ocurre un error, se muestra un mensaje al usuario.
- El botón para vaciar el carrito elimina todos los productos del carrito, borra el almacenamiento local y actualiza la interfaz y el contador.
- Finalmente, hay un evento para cerrar el modal de éxito tras una compra, y se llama a [actualizarCantidadCarrito](#) para mantener el contador sincronizado con el estado real del carrito.

```

212 // --- Filtros de productos ---
213 const buscarInput = document.getElementById('buscarProducto');
214 const filtrarCategoria = document.getElementById('filtrarCategoria');
215 const filtrarPrecio = document.getElementById('filtrarPrecio');
216 let productos = [];
217
218 async function cargarProductos() {
219   try {
220     const res = await fetch('http://localhost:3000/api/productos');
221     if (!res.ok) throw new Error('Error al obtener productos');
222     productos = await res.json();
223     mostrarProductos(productos);
224   } catch (err) {
225     productosDiv.innerHTML = '<p>Error al cargar productos.</p>';
226   }
227 }
228
229 function mostrarProductos(lista) {
230   productosDiv.innerHTML = '';
231   lista.forEach(prod => {
232     const div = document.createElement('div');
233     div.className = 'producto';
234     div.innerHTML = `
235     <h3>${prod.nombre}</h3>
236     <h5>${prod.descripcion}</h5>
237     <p>Desde: $${prod.precio}</p>
238     <button onclick="agregarAlCarrito(${prod.id})">Agregar al carrito</button>
239     `;
240     const btnAgregar = div.querySelector('button');
241     btnAgregar.addEventListener('click', () => agregarAlCarrito(prod));

```

```

242     productosDiv.appendChild(div);
243   });
244 }
245
246 function filtrarProductos() {
247   let filtroNombre = buscarInput.value.toLowerCase();
248   let filtroCategoria = filtrarCategoria.value;
249   let filtroPrecio = filtrarPrecio.value;
250
251   let productosFiltrados = productos.filter(prod => {
252     let coincideNombre = prod.nombre.toLowerCase().includes(filtroNombre);
253     let coincideCategoria = filtroCategoria === "" || prod.categoria === filtroCategoria;
254     let coincidePrecio = filtroPrecio === "" || (filtroPrecio === "10-50" ? prod.precio >= 10
255
256     return coincideNombre && coincideCategoria && coincidePrecio;
257   });
258
259   mostrarProductos(productosFiltrados);
260 }
261
262 buscarInput?.addEventListener('input', filtrarProductos);
263 filtrarCategoria?.addEventListener('change', filtrarProductos);
264 filtrarPrecio?.addEventListener('change', filtrarProductos);
265
266 await cargarProductos();
267 });
268

```

- Este parte del código implementa el filtrado de productos en la tienda. Obtiene los productos desde la API y los muestra en pantalla. Los usuarios pueden buscar productos por nombre, filtrar por categoría o por rango de precio. Cada vez que el usuario cambia un filtro o escribe en la búsqueda, se actualiza la lista mostrada aplicando los filtros seleccionados. Además, cada producto mostrado incluye un botón para

[Es importante destacar que algunas secciones corresponden a una versión inicial del proyecto. Aunque su funcionamiento es adecuado, no fueron elaboradas con extremo detalle, ya que priorizamos el control de versiones y el progreso registrado. Esto se debe a que no logramos crear los repositorios en GitHub; esperamos que esto no afecte nuestra evaluación.

De igual forma, queremos aclarar que la implementación de funcionalidades del proyecto se realizó hasta el último momento. Por ello, es posible que algunas partes presenten ligeras diferencias respecto a la versión final.]