Nota: **Debido a problemas al momento de subir el proyecto a la plataforma, adjunto link a mi GitHub personal para descarga de este: GitHub Torko Motors**

Torko Motors

Luisfher David Rodríguez Olivares

10/10/2025

Bootcamp Programación Intermedio

Nivel Integrador C730

Aprendizaje Interactivo

Oswaldo Martínez Díaz

Contenido

1. Descripción

El proyecto consiste en el desarrollo de un sitio web de tienda en línea especializada en motos y repuestos (Torko Motors), que permita a los usuarios registrarse, navegar por un catálogo de productos, realizar búsquedas con filtros, y gestionar un carrito de compras.

El sitio estará disponible en dos idiomas (español e inglés), lo que permitirá ampliar su alcance y brindar una experiencia más completa a diferentes tipos de usuarios.

Este proyecto busca simular un entorno real de comercio electrónico y servir como base para futuros emprendimientos digitales en el sector automotriz.

2. Objetivo General

Desarrollar una tienda virtual especializada en motos, repuestos y accesorios, que permita a los usuarios registrarse, explorar productos, agregarlos a un carrito de compras y realizar búsquedas eficientes, con una interfaz disponible en español e inglés.

3. Objetivos específicos

- Crear un sistema de registro de usuarios.
- Diseñar un catálogo de productos visualmente atractivo y estructurado.
- Desarrollar un carrito de compras funcional.
- Ofrecer un sistema de búsqueda con filtros para categorías y precios.
- Implementar el sitio en dos idiomas (español, inglés) para alcanzar un público más amplio.
- Garantizar que el diseño sea responsivo para móviles y computadoras.

4. Análisis del problema

4.1 ¿Qué necesidad resuelve el sitio web?

La tienda en línea Torko Motors responde a la necesidad de contar con un medio digital que centralice la información y disponibilidad de productos, permitiendo a los usuarios visualizar un catálogo, realizar búsquedas con filtros, registrarse como clientes y gestionar un carrito de compras.

Este sitio soluciona la dificultad de acceder de manera rápida y organizada a diferentes tipos de productos relacionados con motocicletas, ofreciendo una experiencia práctica y sencilla para el usuario.

Al estar disponible en dos idiomas (español e inglés), también atiende la necesidad de alcanzar un público más amplio, facilitando la navegación para distintos usuarios y mejorando la comunicación entre la tienda y sus clientes.

4.2. ¿A qué usuarios va dirigido este sitio web?

El sitio web está diseñado para ser utilizado principalmente por clientes interesados en la compra de motocicletas o repuestos. Estos usuarios podrán registrarse en la plataforma, explorar el catálogo de productos, realizar búsquedas mediante filtros y gestionar un carrito de compras para organizar los artículos que desean adquirir.

Además, al soportar múltiples idiomas (español e inglés), el sitio podrá ser usado por personas de diferentes lugares, lo que amplía su alcance y facilita la navegación para una mayor cantidad de clientes.

En síntesis, los usuarios de Torko Motors serán personas que buscan una forma sencilla y digital de consultar productos relacionados con motocicletas y partes, con la posibilidad de visualizar y organizar sus compras desde un solo lugar.

4.3. ¿Qué funcionalidades tendrá el sitio web?

El sitio web de Torko Motors contará con diversas funcionalidades orientadas a ofrecer una experiencia de compra sencilla y organizada para los usuarios. Permitirá la creación de usuarios, de manera que los clientes puedan registrarse en la plataforma y guardar su información para gestionar sus compras. Además, tendrá un catálogo de productos bien estructurado, donde se mostrarán los artículos disponibles de forma visual y ordenada, facilitando la exploración y selección de productos.

El sitio incluirá un carrito de compras que permitirá a los usuarios agregar, visualizar y administrar los productos que desean adquirir antes de concretar la compra. También dispondrá de una búsqueda con filtros, lo que ayudará a localizar artículos específicos según categorías o características definidas. El sitio podrá visualizarse en dos idiomas: español e inglés, ampliando su alcance y accesibilidad para diferentes tipos de clientes. Finalmente, contará con un diseño responsive, que se adaptará automáticamente a distintos dispositivos, como computadoras, tablets y teléfonos móviles, asegurando una correcta visualización y navegación en cualquier tamaño de pantalla.

5. Diseño del sistema

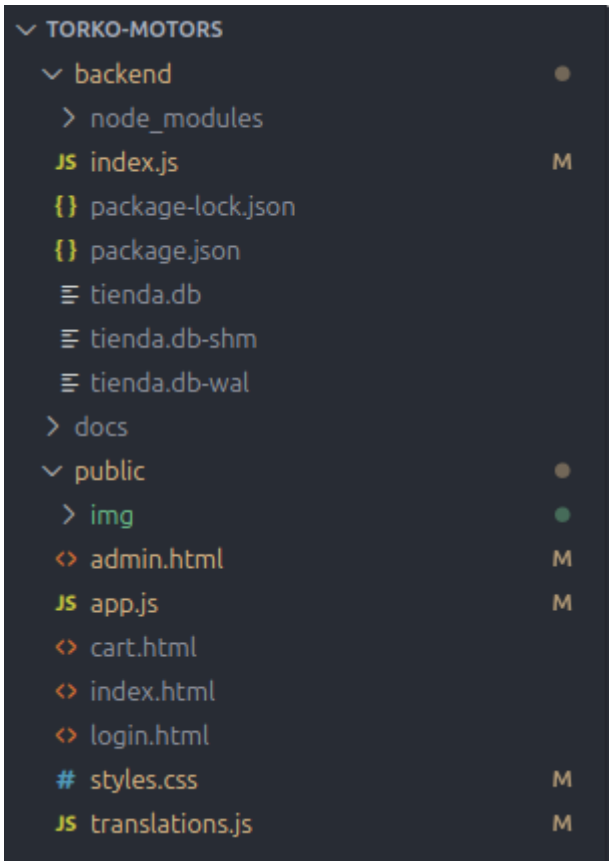
5.1. Diagrama de clases con atributos y métodos

El diagrama de clases del sistema Torko Motors muestra la estructura principal de la aplicación y cómo se relacionan las entidades que la componen. Se identifican tres clases fundamentales: Usuario, Producto y Carrito, así como las asociaciones entre ellas.

- Usuario: representa a la persona registrada en el sistema. Incluye atributos como id, nombre, email y password, que permiten identificar y autenticar al usuario.
- Producto: corresponde a los artículos disponibles en la tienda. Sus atributos principales son id, nombre, descripción, precio, stock, categoria e imagen, los cuales describen las características de cada producto.
- Carrito: funciona como una entidad intermedia que relaciona al Usuario con los Productos seleccionados. Contiene los atributos id, usuario_id, producto_id y cantidad, lo que permite saber qué productos haagregado un usuario y en qué cantidad.

De esta manera, el diagrama refleja la lógica del negocio: un usuario puede agregar varios productos a su carrito, mientras que cada producto puede estar presente en múltiples carritos de diferentes usuarios.

5.2. Diseño del sistema



Estructura de carpetas y archivos del proyecto Torko Motors

La estructura del proyecto Torko Motors se divide en dos grandes módulos: backend y frontend.

- Backend: El backend contiene la lógica principal del sistema. Incluye el archivo index.js, donde se configura el servidor con Express y las rutas de la API; la base de datos tienda.db en SQLite; y los archivos package.json y package-lock.json que gestionan las dependencias del proyecto Node.js. La carpeta node_modules/ almacena las librerías instaladas mediante npm.
- Frontend: El frontend contiene las interfaces web que permiten al usuario interactuar con la tienda virtual. Incluye páginas HTML, estilos en CSS y scripts en JavaScript que manejan la navegación, las traducciones y las peticiones al servidor.

5.3. Tecnologías utilizadas

El desarrollo del sistema Torko Motors se apoya en distintas tecnologías que permiten implementar tanto la parte lógica (backend) como la visual (frontend).

Lenguajes utilizados:

HTML5: Estructura de las páginas web.

CSS3: Estilos y diseño visual del frontend.

JavaScript: Lógica tanto en el frontend como en el backend.

Entorno de ejecución:

Node.js: Permite ejecutar JavaScript en el servidor y manejar las peticiones de los clientes.

Frameworks y librerías:

Express.js: Framework para Node.js que facilita la creación de una API REST.

CORS: Librería que habilita el acceso entre frontend y backend de manera segura.

dotenv: Librería para gestionar variables de entorno.

nodemon: Herramienta que reinicia automáticamente el servidor durante el desarrollo.

Base de datos:

SQLite3: Base de datos ligera, embebida y de fácil uso. Permite almacenar usuarios, productos y carritos sin necesidad de instalar un servidor de base de datos adicional.

Herramientas de desarrollo:

Visual Studio Code (VSCode): Editor de código utilizado para la programación del proyecto.

Postman: Plataforma para probar y validar las rutas del backend.

DB Browser for SQLite: Herramienta gráfica para inspeccionar y modificar la base de datos SQLite.

6. Implementación

6.1. Explicación por clase/componente

6.1.1. Usuario

Atributos (columnas en usuarios)

- id (INTEGER, PK, AUTOINCREMENT)
- nombre (TEXT, NOT NULL)
- email (TEXT, UNIQUE, NOT NULL)
- password (TEXT, NOT NULL)
- rol (TEXT, DEFAULT 'cliente')

Responsabilidades

- Representar a la persona que usa la aplicación (cliente o admin).
- Permitir el registro, inicio de sesión y recuperación de su información pública.
- Controlar permisos básicos (rol: cliente / admin) que determinan acceso a ciertas operaciones.

Endpoints / Métodos que operan sobre Usuario

- POST /usuarios — registrar nuevo usuario.
- POST /login — verificar credenciales e iniciar sesión.
- GET /usuarios/:id — obtener datos públicos de un usuario (sin password).
- GET /usuarios — obtener todos los usuarios (acceso restringido a admin; en la implementación actual se valida rol vía consulta).

Validaciones / restricciones

- nombre, email y password son obligatorios al registrar.
- email debe ser único en la tabla.
- rol por defecto es cliente; ciertos endpoints requieren rol admin.

Interacciones

- Relación con carrito: un Usuario puede tener 0..* entradas en carrito (carritos por usuario).
- Cuando un usuario realiza operaciones administrativas (crear/editar/eliminar productos) se verifica su rol.

6.1.2. Producto

Atributos (columnas en productos)

- id (INTEGER, PK, AUTOINCREMENT)
- nombre (TEXT, NOT NULL)
- precio (REAL, NOT NULL)
- descripcion (TEXT, opcional)
- stock (INTEGER, DEFAULT 0)
- categoria (TEXT, opcional)
- imagen (TEXT, valor por defecto 'default.png' si no se envía)

Responsabilidades

- Mantener el catálogo de items disponibles en la tienda (motos, repuestos, accesorios).
- Proveer información mostrable en la ficha de producto (nombre, precio, descripción, imagen, categoría).
- Administrar stock disponible al añadirse/eliminarse del carrito.

Endpoints / Métodos que operan sobre Producto

- POST /productos — agregar producto (solo admin).
- PUT /productos/:id — actualizar producto (solo admin).
- DELETE /productos/:id — eliminar producto (solo admin).
- GET /productos — listar todos los productos.
- GET /productos/:id — obtener producto por id.

Validaciones / restricciones

- nombre y precio son obligatorios en la creación.
- stock manejado como entero; en la implementación actual permite 0 o mayor.
- Operaciones de creación/actualización/eliminación requieren rol admin (la implementación actual valida vía consulta al usuario).

Interacciones

- Producto se referencia desde carrito.producto_id.
- Al añadir un producto al carrito se reduce inmediatamente el stock (comportamiento implementado). Al eliminar del carrito o vaciarlo, el stock se devuelve.

6.1.3. Carrito

Atributos (columnas en carrito)

- id (INTEGER, PK, AUTOINCREMENT)
- usuario_id (INTEGER, NULLABLE, FK → usuarios.id)

- producto_id (INTEGER, FK → productos.id)
- cantidad (INTEGER, DEFAULT 1)

Responsabilidades

- Representar los ítems que un usuario (o sesión anónima) desea comprar.
- Gestionar operaciones de añadir, actualizar cantidad, eliminar ítem y vaciar carrito.
- Mantener consistencia con el inventario (stock) al agregar/eliminar ítems.

Endpoints / Métodos que operan sobre Carrito

- POST /carrito — agregar producto al carrito (reduce stock).
- GET /carrito/:usuario_id — obtener carrito de un usuario (devuelve subtotales y total).
- PUT /carrito/:id — actualizar cantidad de un item en el carrito (ajusta stock según diferencia).
- DELETE /carrito/:id — eliminar un item (devuelve stock).
- DELETE /carrito/usuario/:usuario_id — vaciar carrito completo (devuelve stock de todos los items).
- GET /carritos — ver todos los carritos (acceso admin).

Validaciones / restricciones

- Antes de insertar en carrito, se verifica que el producto exista y que stock >= cantidadSolicitada.
- usuario_id puede ser NULL según la implementación (soporta carritos anónimos).
- Actualizar cantidad comprueba disponibilidad adicional (diferencia positiva requiere stock suficiente).

Interacciones

- Opera directamente con productos para ajustar stock.
- Referencia usuarios para asociar el carrito a un usuario cuando corresponde.
- Las operaciones de vaciado o eliminación devuelven stock a productos.

6.1.4. Frontend (app.js)

El archivo app.js contiene la lógica de la interfaz web que interactúa con el backend: carga y muestra productos, aplica filtros, gestiona el estado de autenticación del usuario y permite añadir productos al carrito. Usa fetch para consumir la API en http://localhost:4000.

Funciones / módulos principales

- initAuth() — Inicializa el estado de autenticación en la UI leyendo localStorage y mostrando/ocultando elementos (login, logout, enlace admin).
- logout() — Elimina la información del usuario en localStorage y redirige a la página principal.
- loadProducts() — Consulta GET /productos y carga allProducts / filteredProducts, luego llama a displayProducts().
- displayProducts(products) — Renderiza las cards de productos en el DOM (imagen, nombre, descripción, precio, stock y botón). Maneja onerror de la imagen y estado disabled cuando stock === 0.
- addToCart(productId) — Envía POST /carrito con { usuario_id, producto_id, cantidad }. Si el usuario no está autenticado redirige a login.html. Tras añadir con éxito recarga loadProducts() para actualizar stock.
- setupFilters() — Enlaza los inputs de búsqueda, categoría y botones de aplicar/limpiar a los controladores de filtro.
- filterProducts() — Aplica búsqueda por nombre/descripcion, filtro por categoria (si existe en el producto) y rango de precios (minPrice, maxPrice), actualiza filteredProducts y llama a displayProducts().
- clearFilters() — Resetea inputs de filtro y restaura filteredProducts = allProducts.

Endpoints consumidos

- GET /productos — para listar todos los productos (usado en loadProducts).
- POST /carrito — para agregar un producto al carrito (usado en addToCart).
  (Otros endpoints del backend no son consumidos desde app.js en el código enviado, pero existen en la API.)

Estado y almacenamiento

- allProducts, filteredProducts — arreglos en memoria que contienen los productos cargados.
- localStorage.user — objeto JSON con datos del usuario (id, nombre, rol, etc.) usado para determinar auth y enviar usuario_id en peticiones al carrito.

Validaciones y comportamiento

- Verifica presencia de user en localStorage antes de permitir añadir al carrito; si no existe redirige a login.html.
- Deshabilita el botón de añadir si product.stock === 0 y cambia el texto usando traducciones (t('out_of_stock')).

- En loadProducts() y addToCart() maneja errores de red con mensajes visibles/alerts (t('connection_error'), t('add_error')).
- filterProducts() contempla que product.categoria pueda estar ausente; el filtro compara también con nombre y descripcion si hace falta.

Interacciones con backend y otras partes del sistema

- Al agregar un producto (POST /carrito) el frontend espera que el backend reduzca el stock (por eso llama a loadProducts() luego de un add).
- El control de visibilidad del enlace admin depende del user.rol guardado en localStorage.
- Las imágenes se cargan desde public/img/<imagen>; el src por defecto es default.png en caso de error.

6.2. Fragmentos clave de código

En esta sección se presentan los bloques de código más relevantes de la implementación del proyecto Torko Motors.
Cada fragmento pertenece al archivo correspondiente dentro del backend o frontend, acompañado de una breve descripción de su función dentro del sistema.

6.2.1. Conexión a la base de datos (SQLite)

Archivo**:** backend/index.js

Este bloque establece la conexión con la base de datos tienda.db utilizando sqlite3. Además, habilita el modo WAL (Write-Ahead Logging) para mejorar el rendimiento y permitir concurrencia entre lecturas y escrituras.

```
// Conexión a la base de datos SQLite
const DB_PATH = path.join(__dirname, "tienda.db");
const db = new sqlite3.Database(DB_PATH, (err) => {
    if (err) {
        console.error("Error al conectar a la base de datos:", err.message);
    } else {
        console.log("Conectado a la base de datos SQLite.");

        // Habilitacion WAL (Write-Ahead Logging)
        db.run("PRAGMA journal_mode = WAL;", (err) => {
            if (err) {
                console.error("No se pudo habilitar WAL:", err.message);
            } else {
                console.log("Modo WAL activado para mejorar concurrencia y evitar bloqueos.");
            }
        });
    }
});
```

6.2.2. Creación de tablas principales

Archivo: backend/index.js

Este fragmento crea las tablas usuarios, productos y carrito en la base de datos si no existen, definiendo las claves primarias y foráneas que establecen las relaciones del sistema.

```
db.serialize(() => {
    db.run(`
     CREATE TABLE IF NOT EXISTS usuarios (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        nombre TEXT NOT NULL,
        email TEXT UNIQUE NOT NULL,
        password TEXT NOT NULL,
        rol TEXT DEFAULT 'cliente'
     );
    `);
    db.run(`
     CREATE TABLE IF NOT EXISTS productos (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        nombre TEXT NOT NULL,
        precio REAL NOT NULL,
        descripcion TEXT,
        stock INTEGER DEFAULT 0
     );
    `);
    db.run(`
     CREATE TABLE IF NOT EXISTS carrito (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        usuario_id INTEGER NULL,
        producto_id INTEGER,
        cantidad INTEGER DEFAULT 1,
        FOREIGN KEY (usuario_id) REFERENCES usuarios(id),
        FOREIGN KEY (producto_id) REFERENCES productos(id)
     );
    `);
});
```

Nota: En la tabla productos, las columnas categoria e imagen, se insertaron directamente en la base de datos usando la opción Execute SQL de DB Browser for SQLite, para probar distintas formas de agregar elementos a la base de datos.

6.2.3. Registro de nuevo usuario

Archivo: backend/index.js

Maneja la creación de nuevos usuarios a través del endpoint POST /usuarios. Valida campos obligatorios e inserta el registro en la tabla usuarios, asignando por defecto el rol cliente.

```
// Registrar nuevo usuario (cliente por defecto)
app.post("/usuarios", (req, res) => {
    const { nombre, email, password } = req.body;

    if (!nombre || !email || !password) {
        return res.status(400).json({ error: "Todos los campos son obligatorios" });
    }

    const query = `
        INSERT INTO usuarios (nombre, email, password, rol)
        VALUES (?, ?, ?, 'cliente')
    `;

    db.run(query, [nombre, email, password], function (err) {
        if (err) {
            console.error("Error al registrar usuario:", err.message);
            return res.status(500).json({ error: "No se pudo registrar el usuario (email duplicado o error interno)" });
        }

        res.status(201).json({
            mensaje: "Usuario registrado exitosamente",
            id: this.lastID
        });
    });
});
```

6.2.4. Consulta de productos disponibles

Archivo: backend/index.js

Endpoint que devuelve todos los productos registrados en la base de datos. Es utilizado por el frontend para mostrar el catálogo general de la tienda.

```
// obtener todos los productos de la DB
app.get("/productos", (req, res) => {
    db.all("SELECT * FROM productos", [], (err, rows) => {
        if (err) {
            console.error("Error al obtener productos:", err.message);
            return res.status(500).json({ error: "No se pudieron obtener los productos" });
        }
        res.json(rows);
    });
});
```

6.2.5. Agregar producto al carrito

Archivo: backend/index.js

Permite añadir un producto al carrito, verificando primero que exista stock suficiente. En caso de éxito, reduce el stock del producto en la tabla productos.

```
// Agregar producto al carrito
app.post("/carrito", (req, res) => {
    const { usuario_id, producto_id, cantidad } = req.body;
    const cantidadSolicitada = cantidad || 1;

    // Verificar existencia del producto y stock disponible
    db.get("SELECT * FROM productos WHERE id = ?", [producto_id], (err, producto) => {
        if (err) {
            console.error("Error al verificar producto:", err.message);
            return res.status(500).json({ error: "Error interno del servidor" });
        }
        if (!producto) {
            return res.status(404).json({ error: "Producto no encontrado" });
        }

        // Validación de stock
        if (producto.stock < cantidadSolicitada) {
            return res.status(400).json({
                error: `Stock insuficiente. Disponible: ${producto.stock}`
            });
        }

        // Agregar producto al carrito
        const query = `
            INSERT INTO carrito (usuario_id, producto_id, cantidad)
            VALUES (?, ?, ?)
        `;
        db.run(query, [usuario_id || null, producto_id, cantidadSolicitada], function (err) {
            if (err) {
                console.error("Error al agregar al carrito:", err.message);
                return res.status(500).json({ error: "No se pudo agregar al carrito" });
            }

            // Reducir stock del producto en la tabla productos
            db.run(
                "UPDATE productos SET stock = stock - ? WHERE id = ?",
                [cantidadSolicitada, producto_id],
                (err2) => {
                    if (err2) {
                        console.error("Error al actualizar stock:", err2.message);
                    }
                }
            );

            res.status(201).json({
                mensaje: "Producto agregado al carrito",
                producto_id: producto_id,
                cantidad: cantidadSolicitada
            });
        });
    });
});
```

6.2.6. Carga y visualización de productos en el frontend

Archivo: public/app.js

El frontend obtiene los productos del backend mediante fetch y los muestra dinámicamente en la interfaz, generando tarjetas con nombre, imagen, precio y stock.

```
// Cargar todos los productos
async function loadProducts() {
    try {
        const response = await fetch(`${API_URL}/productos`);
        const data = await response.json();

        if (response.ok) {
            allProducts = data;
            filteredProducts = data;
            displayProducts(filteredProducts);
        } else {
            document.getElementById('products-container').innerHTML =
                `<p class="error">${t('no_products')}</p>`;
        }
    } catch (error) {
        document.getElementById('products-container').innerHTML =
            `<p class="error">${t('connection_error')}</p>`;
    }
}
```

6.2.7. Filtrado de productos en la interfaz

Archivo: public/app.js

Permite aplicar filtros por nombre, descripción, categoría o rango de precios, actualizando la vista de productos mostrada al usuario.

```
// Filtrar productos
function filterProducts() {
    const searchTerm = document.getElementById('search-input').value.toLowerCase();
    const categorySelect = document.getElementById('category-filter');
    const category = categorySelect ? categorySelect.value.toLowerCase() : '';
    const minPrice = parseFloat(document.getElementById('min-price').value) || 0;
    const maxPrice = parseFloat(document.getElementById('max-price').value) || Infinity;

    filteredProducts = allProducts.filter(product => {
        const matchesSearch = product.nombre.toLowerCase().includes(searchTerm) ||
                              (product.descripcion && product.descripcion.toLowerCase().includes(searchTerm));

        // Filtro por categoría - verifica si el producto tiene el campo categoria
        let matchesCategory = true;
        if (category) {
            matchesCategory = (product.categoria && product.categoria.toLowerCase() === category) ||
                              product.nombre.toLowerCase().includes(category) ||
                              (product.descripcion && product.descripcion.toLowerCase().includes(category));
        }

        const matchesPrice = product.precio >= minPrice && product.precio <= maxPrice;

        return matchesSearch && matchesCategory && matchesPrice;
    });

    displayProducts(filteredProducts);
}
```

6.2.8. Interfaz y lógica del carrito

Archivo: public/cart.html

Este archivo implementa la interfaz del carrito de compras. Permite mostrar los productos añadidos por el usuario, actualizar cantidades, eliminar ítems y vaciar el carrito completo.
Los datos se obtienen del backend mediante peticiones fetch hacia los endpoints /carrito.

```
<main class="container">
    <h2 data-i18n="cart_title">Mi Carrito</h2>

    <div id="cart-container">
        <p data-i18n="loading">Cargando carrito...</p>
    </div>

    <div id="cart-summary" style="display:none;">
        <h3 data-i18n="cart_total">Total:</h3>
        <p id="total-amount">$0.00</p>
        <button id="clear-cart-btn" data-i18n="cart_clear">Vaciar Carrito</button>
    </div>

    <div id="cart-message" class="message"></div>
</main>
```

6.2.9. Cargar y mostrar el carrito

Archivo: public/cart.html

Consulta el carrito del usuario autenticado y lo muestra en pantalla con los subtotales y el total general.

```
async function loadCart() {
    const user = JSON.parse(localStorage.getItem('user'));

    if (!user) {
        document.getElementById('cart-container').innerHTML =
            `<p data-i18n="cart_login_required">${t('cart_login_required')}</p>`;
        return;
    }

    try {
        const response = await fetch(`http://localhost:4000/carrito/${user.id}`);
        const data = await response.json();

        if (response.ok) {
            displayCart(data);
        } else {
            showCartMessage(data.error || t('cart_load_error'), 'error');
        }
    } catch (error) {
        showCartMessage(t('connection_error'), 'error');
    }
}

function displayCart(data) {
    const container = document.getElementById('cart-container');

    if (!data.productos || data.productos.length === 0) {
        container.innerHTML = `<p data-i18n="cart_empty">${t('cart_empty')}</p>`;
        document.getElementById('cart-summary').style.display = 'none';
        return;
    }

    container.innerHTML = data.productos.map(item => `
        <div class="cart-item" data-cart-id="${item.carrito_id}">
            <div class="cart-item-info">
                <h3>${item.nombre}</h3>
                <p>${t('price')}: $${item.precio.toFixed(2)}</p>
            </div>
            <div class="cart-item-controls">
                <input type="number"
                       value="${item.cantidad}"
                       min="1"
                       class="cart-quantity"
                       data-cart-id="${item.carrito_id}">
                <button onclick="updateCartItem(${item.carrito_id})" data-i18n="update">${t('update')}</button>
                <button onclick="removeCartItem(${item.carrito_id})" data-i18n="remove">${t('remove')}</button>
            </div>
            <div class="cart-item-subtotal">
                <strong>${t('subtotal')}: $${item.subtotal.toFixed(2)}</strong>
            </div>
        </div>
    `).join('');

    document.getElementById('total-amount').textContent = `$${data.total.toFixed(2)}`;
    document.getElementById('cart-summary').style.display = 'block';
}
```

6.2.10. Actualizar y eliminar ítems del carrito

Archivo: public/cart.html

Permite modificar la cantidad de un producto o eliminarlo completamente mediante peticiones PUT y DELETE.

```
async function updateCartItem(cartId) {
        showCartMessage(t('invalid_quantity'), 'error');
        return;
    }

    try {
        const response = await fetch(`http://localhost:4000/carrito/${cartId}`, {
            method: 'PUT',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ cantidad })
        });

        const data = await response.json();

        if (response.ok) {
            showCartMessage(t('cart_updated'), 'success');
            loadCart();
        } else {
            showCartMessage(data.error || t('update_error'), 'error');
            loadCart();
        }
    } catch (error) {
        showCartMessage(t('connection_error'), 'error');
    }
}

async function removeCartItem(cartId) {
    if (!confirm(t('confirm_remove'))) return;

    try {
        const response = await fetch(`http://localhost:4000/carrito/${cartId}`, {
            method: 'DELETE'
        });

        const data = await response.json();

        if (response.ok) {
            showCartMessage(t('item_removed'), 'success');
            loadCart();
        } else {
            showCartMessage(data.error || t('remove_error'), 'error');
        }
    } catch (error) {
        showCartMessage(t('connection_error'), 'error');
    }
}
```

6.2.11. Vaciar carrito

Archivo: public/cart.html

Elimina todos los ítems del carrito y actualiza la vista.

```
async function clearCart() {
    const user = JSON.parse(localStorage.getItem('user'));
    if (!user) return;

    if (!confirm(t('confirm_clear_cart'))) return;

    try {
        const response = await fetch(`http://localhost:4000/carrito/usuario/${user.id}`, {
            method: 'DELETE'
        });

        const data = await response.json();

        if (response.ok) {
            showCartMessage(t('cart_cleared'), 'success');
            loadCart();
        } else {
            showCartMessage(data.error || t('clear_error'), 'error');
        }
    } catch (error) {
        showCartMessage(t('connection_error'), 'error');
    }
}
```

6.2.12. Autenticación

Archivo: public/login.html

Formularios de login y registro con manejo de UI (mostrar/ocultar formularios) y llamadas fetch a los endpoints POST /login y POST /usuarios.

```html
<main class="container">
    <div class="auth-container">
        <!-- Login Form -->
        <div class="auth-form" id="login-form">
            <h2 data-i18n="login_title">Iniciar Sesión</h2>
            <form id="form-login">
                <input type="email" id="login-email" placeholder="Email" required>
                <input type="password" id="login-password" placeholder="Contraseña" required>
                <button type="submit">Entrar</button>
            </form>
            <p>¿No tienes cuenta? <a href="#" id="show-register">Regístrate</a></p>
        </div>

        <!-- Register Form -->
        <div class="auth-form" id="register-form" style="display:none;">
            <h2 data-i18n="register_title">Crear Cuenta</h2>
            <form id="form-register">
                <input type="text" id="register-nombre" placeholder="Nombre completo" required>
                <input type="email" id="register-email" placeholder="Email" required>
                <input type="password" id="register-password" placeholder="Contraseña" required>
                <button type="submit">Registrarse</button>
            </form>
            <p>¿Ya tienes cuenta? <a href="#" id="show-login">Inicia sesión</a></p>
        </div>

        <div id="auth-message" class="message"></div>
    </div>
</main>
```

6.2.13. Handler: Login (POST /login)

Archivo: public/login.html

```javascript
document.getElementById('form-login').addEventListener('submit', async (e) => {
    e.preventDefault();
    const email = document.getElementById('login-email').value;
    const password = document.getElementById('login-password').value;

    try {
        const response = await fetch(`${API_URL}/login`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ email, password })
        });

        const data = await response.json();

        if (response.ok) {
            localStorage.setItem('user', JSON.stringify(data.usuario));
            showMessage('Inicio de sesión exitoso', 'success');
            setTimeout(() => window.location.href = 'index.html', 1000);
        } else {
            showMessage(data.error || 'Error al iniciar sesión', 'error');
        }
    } catch (error) {
        console.error('Error:', error);
        showMessage('Error de conexión con el servidor', 'error');
    }
});
```

6.2.14. Handler: Registro (POST /usuarios)

Archivo: public/login.html

```
document.getElementById('form-register').addEventListener('submit', async (e) => {
    e.preventDefault();
    const nombre = document.getElementById('register-nombre').value;
    const email = document.getElementById('register-email').value;
    const password = document.getElementById('register-password').value;

    try {
        const response = await fetch(`${API_URL}/usuarios`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ nombre, email, password })
        });

        const data = await response.json();

        if (response.ok) {
            showMessage('Registro exitoso. Ahora puedes iniciar sesión', 'success');
            setTimeout(() => {
                document.getElementById('show-login').click();
            }, 1500);
        } else {
            showMessage(data.error || 'Error al registrarse', 'error');
        }
    } catch (error) {
        console.error('Error:', error);
        showMessage('Error de conexión con el servidor', 'error');
    }
});
```

6.2.15. Sistema de traducciones

Archivo: public/translations.js

Gestión de textos en es/en, función t(key) para obtener traducciones, aplicación automática en el DOM y soporte básico para mostrar el nombre de usuario en textos dinámicos.

```
const translations = {
  es: {
    site_title: "Torko Motors",
    nav_catalog: "Catálogo",
    nav_cart: "Carrito",
    add_to_cart: "Agregar al Carrito",
    cart_title: "Mi Carrito",
    connection_error: "Error de conexión con el servidor",
    welcome: "Bienvenido",
    welcome_guest: "Bienvenido"
  },
  en: {
    site_title: "Torko Motors",
    nav_catalog: "Catalog",
    nav_cart: "Cart",
    add_to_cart: "Add to Cart",
    cart_title: "My Cart",
    connection_error: "Server connection error",
    welcome: "Welcome",
    welcome_guest: "Welcome"
  }
};
```

6.2.16. Función para obtener traducción

Archivo: public/translations.js

```
// Obtener traduccion
function t(key) {
    return translations[currentLang][key] || key;
}
```

6.2.17. Aplicar traducciones en el DOM (incluye soporte dinámico con nombre de usuario)

```
// Aplicar traducciones al DOM
function applyTranslations() {
    // Translate text content
    document.querySelectorAll('[data-i18n]').forEach(el => {
        const key = el.getAttribute('data-i18n');
        el.textContent = t(key);
    });

    // Traducir placeholders
    document.querySelectorAll('[data-i18n-placeholder]').forEach(el => {
        const key = el.getAttribute('data-i18n-placeholder');
        el.placeholder = t(key);
    });

    // Traducir contenido dinamico como "Bienvenido, [Nombre]"
    document.querySelectorAll('[data-i18n-dynamic]').forEach(el => {
        const key = el.getAttribute('data-i18n-dynamic');
        const userName = el.getAttribute('data-user-name');
        if (userName) {
            el.textContent = `${t(key)}, ${userName}`;
        }
    });
}
```

6.2.18. Inicializar selector de idioma y aplicar traducciones

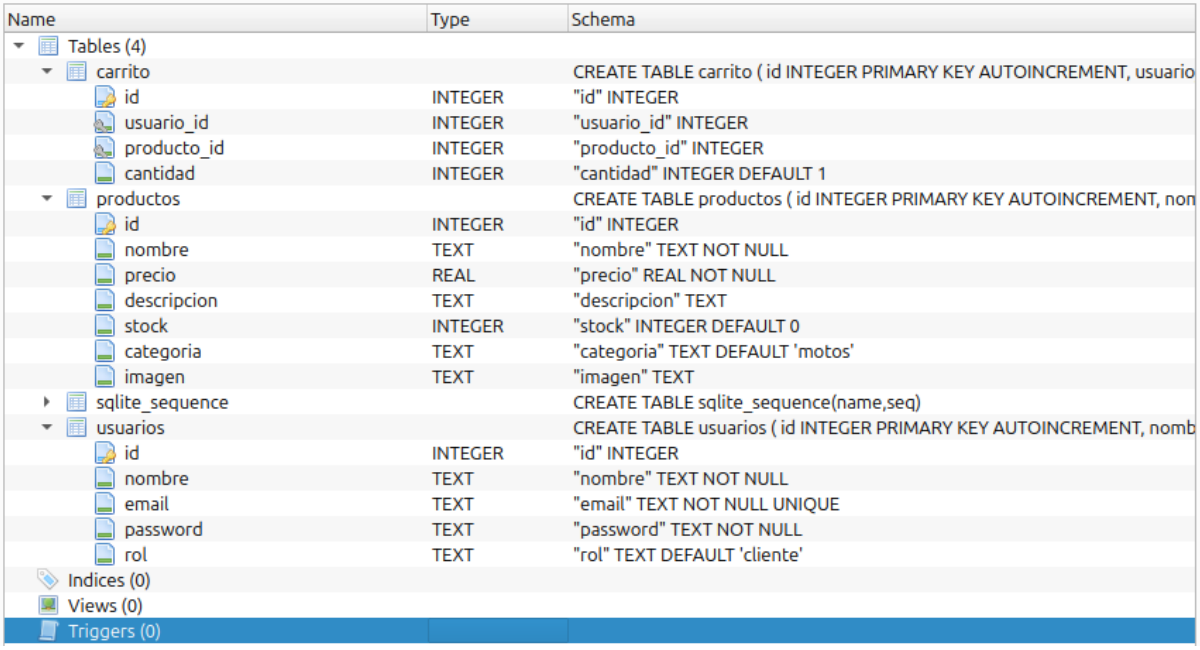
```
// Inicializar selector de idioma
function initLanguage() {
    const savedLang = localStorage.getItem('language') || 'es';
    currentLang = savedLang;

    const selector = document.getElementById('lang-selector');
    if (selector) {
        selector.value = currentLang;
        selector.addEventListener('change', (e) => {
            currentLang = e.target.value;
            localStorage.setItem('language', currentLang);
            applyTranslations();
        });
    }

    applyTranslations();
}
```

6.3 Pruebas y validación

6.3.1. Pruebas del backend

En las siguientes capturas se evidencian las pruebas de los endpoints principales del backend. Las respuestas devueltas por el servidor confirman que las operaciones CRUD de usuarios, productos y carrito funcionan correctamente.

En esta captura se observa la ejecución del comando node index.js dentro del entorno de Visual Studio Code.
El mensaje de salida confirma que el servidor se encuentra corriendo correctamente en la dirección http://localhost:4000, y que se ha establecido conexión con la base de datos SQLite. Además, se indica que el modo WAL (Write-Ahead Logging) está activado, lo que mejora la concurrencia y reduce la posibilidad de bloqueos en la base de datos.

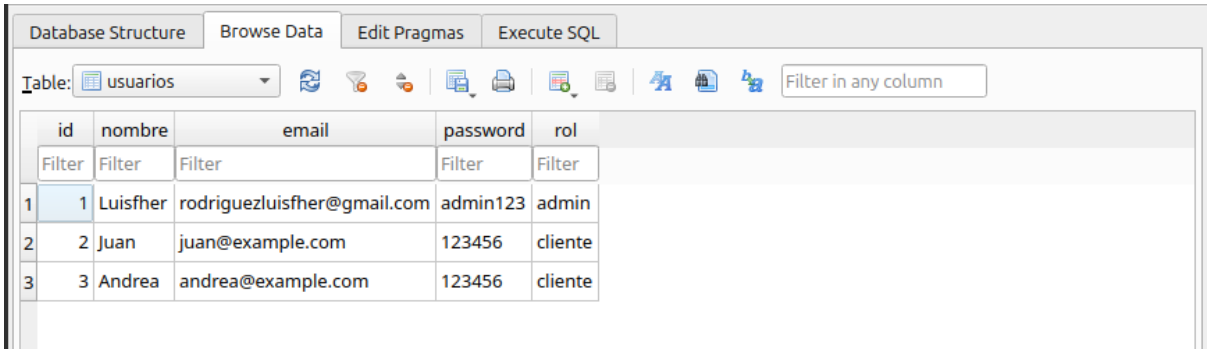


Name	Type	Schema
Tables (4)		
carrito		CREATE TABLE carrito (id INTEGER PRIMARY KEY AUTOINCREMENT, usuario
id	INTEGER	"id" INTEGER
usuario_id	INTEGER	"usuario_id" INTEGER
producto_id	INTEGER	"producto_id" INTEGER
cantidad	INTEGER	"cantidad" INTEGER DEFAULT 1
productos		CREATE TABLE productos (id INTEGER PRIMARY KEY AUTOINCREMENT, non
id	INTEGER	"id" INTEGER
nombre	TEXT	"nombre" TEXT NOT NULL
precio	REAL	"precio" REAL NOT NULL
descripcion	TEXT	"descripcion" TEXT
stock	INTEGER	"stock" INTEGER DEFAULT 0
categoria	TEXT	"categoria" TEXT DEFAULT 'motos'
imagen	TEXT	"imagen" TEXT
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)
usuarios		CREATE TABLE usuarios (id INTEGER PRIMARY KEY AUTOINCREMENT, nomb
id	INTEGER	"id" INTEGER
nombre	TEXT	"nombre" TEXT NOT NULL
email	TEXT	"email" TEXT NOT NULL UNIQUE
password	TEXT	"password" TEXT NOT NULL
rol	TEXT	"rol" TEXT DEFAULT 'cliente'
Indices (0)		
Views (0)		
Triggers (0)		

Esta imagen muestra la estructura de las tablas definidas dentro de la base de datos SQLite utilizada por el backend.
Se observan cuatro tablas principales:

- usuarios: almacena la información de los usuarios del sistema, incluyendo su nombre, correo electrónico, contraseña y rol (cliente o administrador).
- productos: contiene los datos de los productos disponibles, con campos como nombre, precio, descripción, stock, categoría e imagen.
- carrito: registra los productos añadidos al carrito de compras por cada usuario, incluyendo las relaciones usuario_id y producto_id.

- sqlite_sequence: tabla interna de SQLite que mantiene el control del valor autoincremental de los identificadores.



Database Structure | Browse Data | Edit Pragmas | Execute SQL

Table: usuarios — Filter in any column

	id	nombre	email	password	rol
	Filter	Filter	Filter	Filter	Filter
1	1	Luisfher	rodriguezluisfher@gmail.com	admin123	admin
2	2	Juan	juan@example.com	123456	cliente
3	3	Andrea	andrea@example.com	123456	cliente

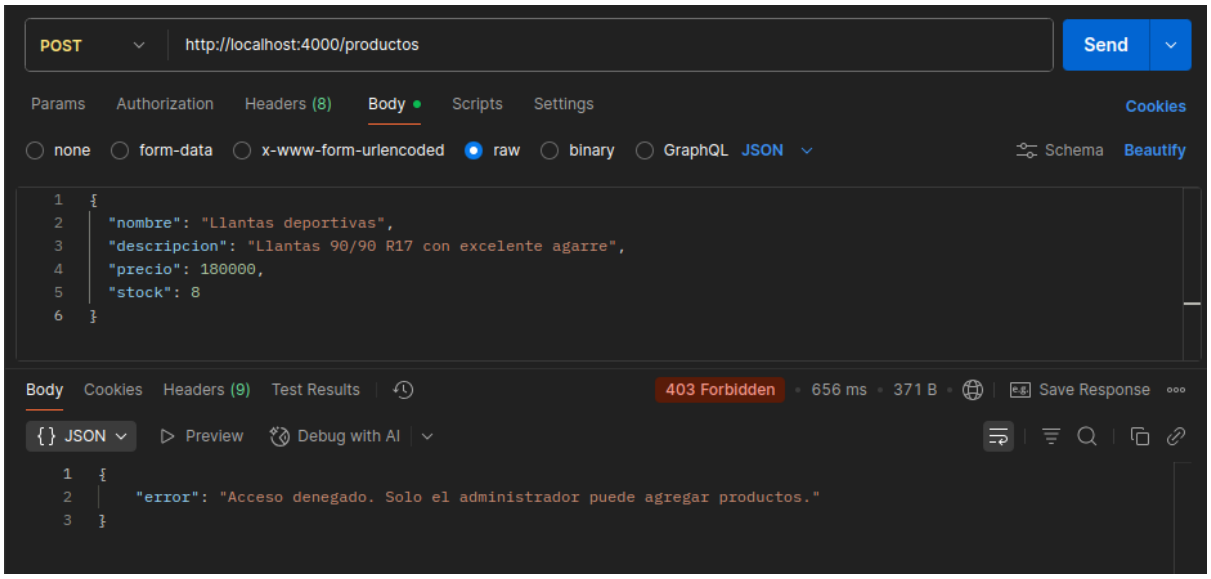
En esta vista se muestran los registros almacenados en la tabla usuarios.
Actualmente existen tres usuarios:

1. Luisfher, con rol *admin* y correo rodriguezluisfher@gmail.com.
2. Juan, con rol *cliente*.
3. Andrea, también con rol *cliente*.
   Estos registros permiten diferenciar los privilegios de acceso, ya que solo los usuarios con rol *admin* pueden realizar ciertas operaciones, como la creación o modificación de productos.

Table: productos — Filter in any column

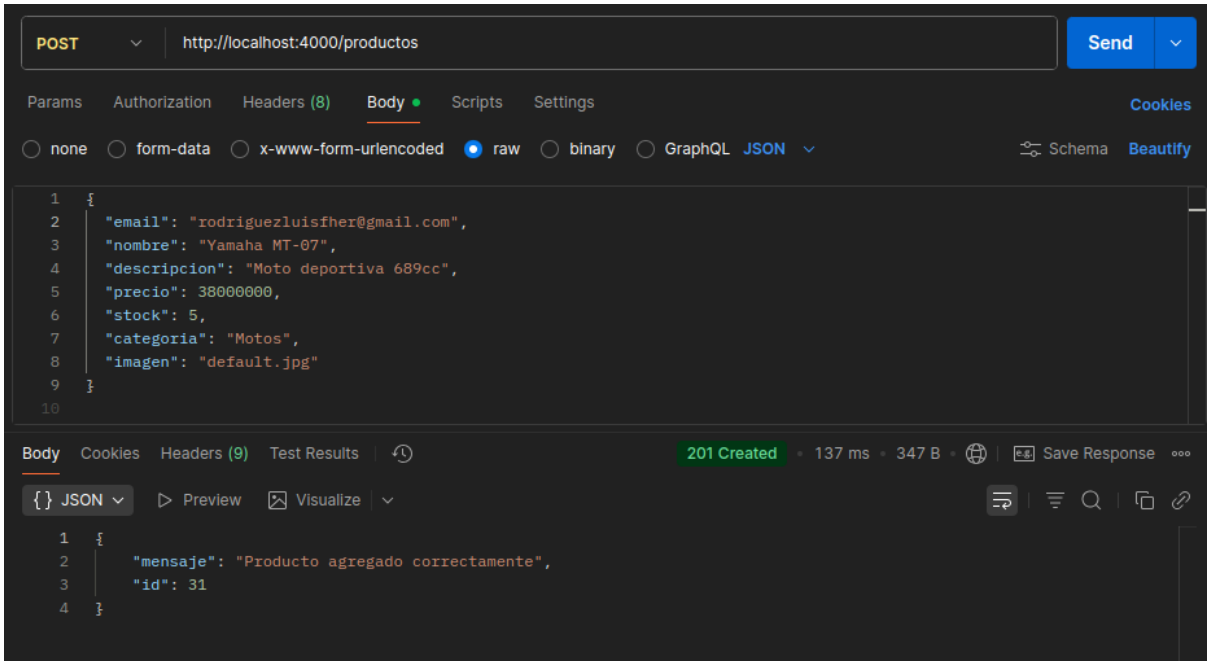
	id	nombre	precio	descripcion	stock	categoria	ima
	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Yamaha MT-09	58000000.0	Una naked con alma deportiva, motor ...	2	motos	yamal
2	2	Honda CB650R	52000000.0	Estilo neo-retro con motor de 4 cilindros en ...	4	motos	honda
3	3	Kawasaki Z900	60000000.0	Potencia brutal, diseño agresivo y una ...	3	motos	kawas
4	4	Suzuki GSX-S750	55000000.0	Inspirada en la superbike GSX-R, ofrece ...	3	motos	suzuk
5	5	BMW F 900 R	65000000.0	Naked deportiva con motor bicilíndrico y ...	2	motos	defau
6	6	KTM Duke 890	59000000.0	Ligera y poderosa, ofrece un manejo ágil con ...	3	motos	defau
7	7	Yamaha R7	63000000.0	Superdeportiva con diseño agresivo y motor ...	2	motos	defau
8	8	Honda CBR650R	57000000.0	Deportiva de media cilindrada con un equilibri...	4	motos	defau
9	9	Kawasaki Ninja 650	48000000.0	Diseño aerodinámico y motor bicilíndrico para...	5	motos	defau
10	10	Suzuki V-Strom 650	52000000.0	Versátil y confiable, ideal para viajes largos y ...	3	motos	defau
11	11	Casco LS2 FF327 Challenger	480000.0	Casco integral con visor solar interno y ...	12	accesorios	defau
12	12	Guantes Alpinestars SP-8	180000.0	Guantes deportivos de cuero con refuerzos en...	15	accesorios	defau
13	13	Chaqueta para moto Xtreme Rider	350000.0	Chaqueta con protecciones desmontables y ...	10	accesorios	defau
14	14	Botas Fox Comp	420000.0	Botas de motocross con refuerzos y cierre de ...	8	accesorios	defau
15	15	Maletero Givi E300	270000.0	Baúl trasero de 30L con sistema de cierre ...	5	accesorios	defau
16	16	Rodilleras Leatt Dual Axis	200000.0	Protección articulada para rodillas con ...	14	accesorios	defau

Aquí se presenta la información contenida en la tabla productos.
Los registros incluyen motocicletas y accesorios con sus respectivos nombres, precios, descripciones, cantidad en stock, categorías y rutas de imagen.
Esta tabla es utilizada para listar los productos disponibles en el catálogo del sistema y gestionar su inventario.



Esta captura muestra una prueba realizada desde Postman para agregar un nuevo producto mediante una solicitud POST al endpoint http://localhost:4000/productos.
En este primer intento, la petición no incluye la información del usuario administrador.
Como resultado, el servidor responde con el código de estado **403 Forbidden** y el mensaje:

“Acceso denegado. Solo el administrador puede agregar productos.”
Esto evidencia que el sistema cuenta con un mecanismo de control de roles que restringe la creación de productos únicamente a los usuarios con privilegios de administrador.

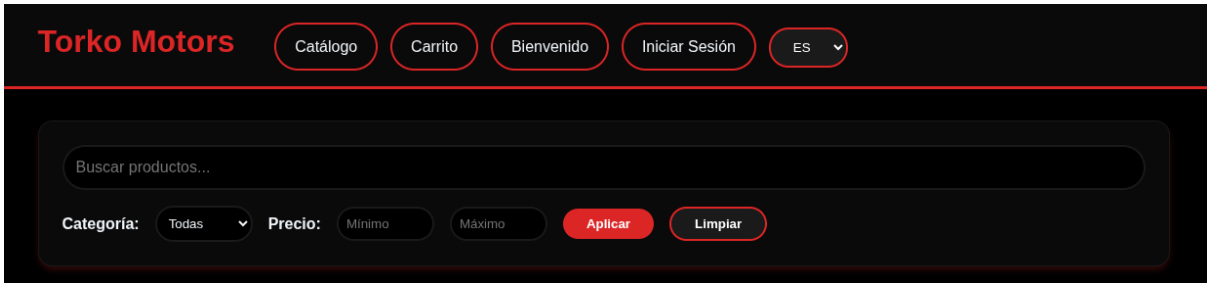


En esta segunda prueba, también realizada desde Postman, se envía nuevamente una solicitud POST al mismo endpoint (/productos), pero incluyendo el campo "email": "rodriguezluisfher@gmail.com", correspondiente al usuario con rol de administrador.
El servidor procesa correctamente la solicitud y responde con el código **201 Created**, junto con el mensaje:

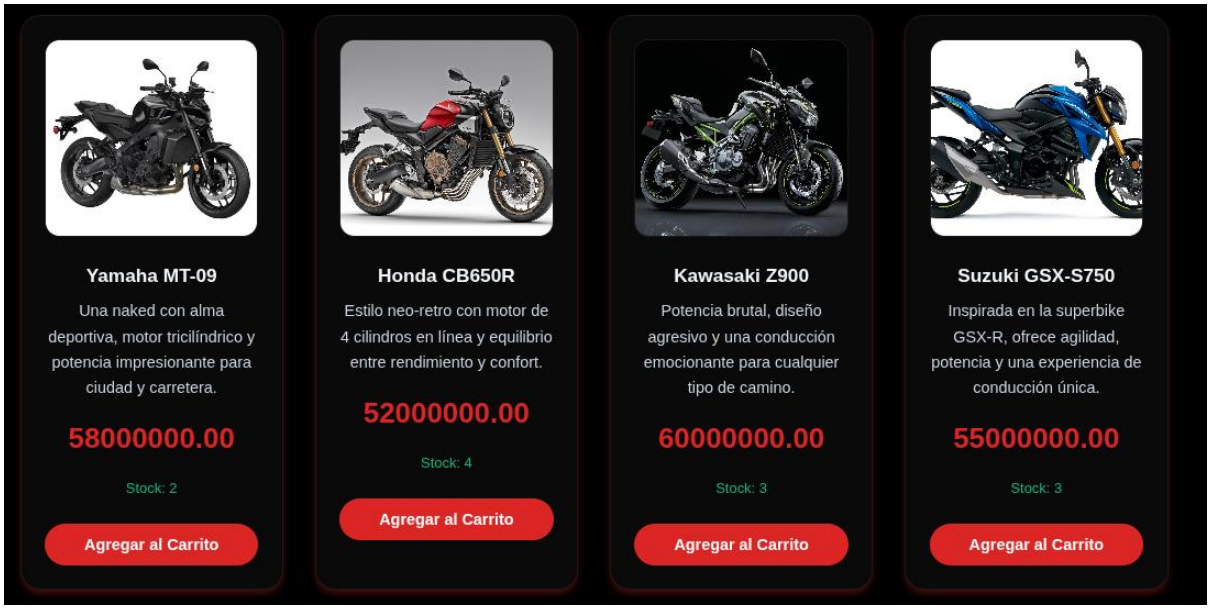
“Producto agregado correctamente.”
y el ID del nuevo registro insertado en la base de datos.
Esta respuesta confirma que la autenticación y el control de permisos funcionan correctamente, permitiendo únicamente a los administradores realizar operaciones de alta sobre la tabla de productos.

6.3.2. Pruebas del frontend

El frontend refleja correctamente los datos obtenidos del backend y responde a las acciones del usuario. Se verificó que la información del carrito se actualiza en tiempo real, y que la interfaz se adapta al idioma seleccionado gracias al sistema de traducciones implementado.

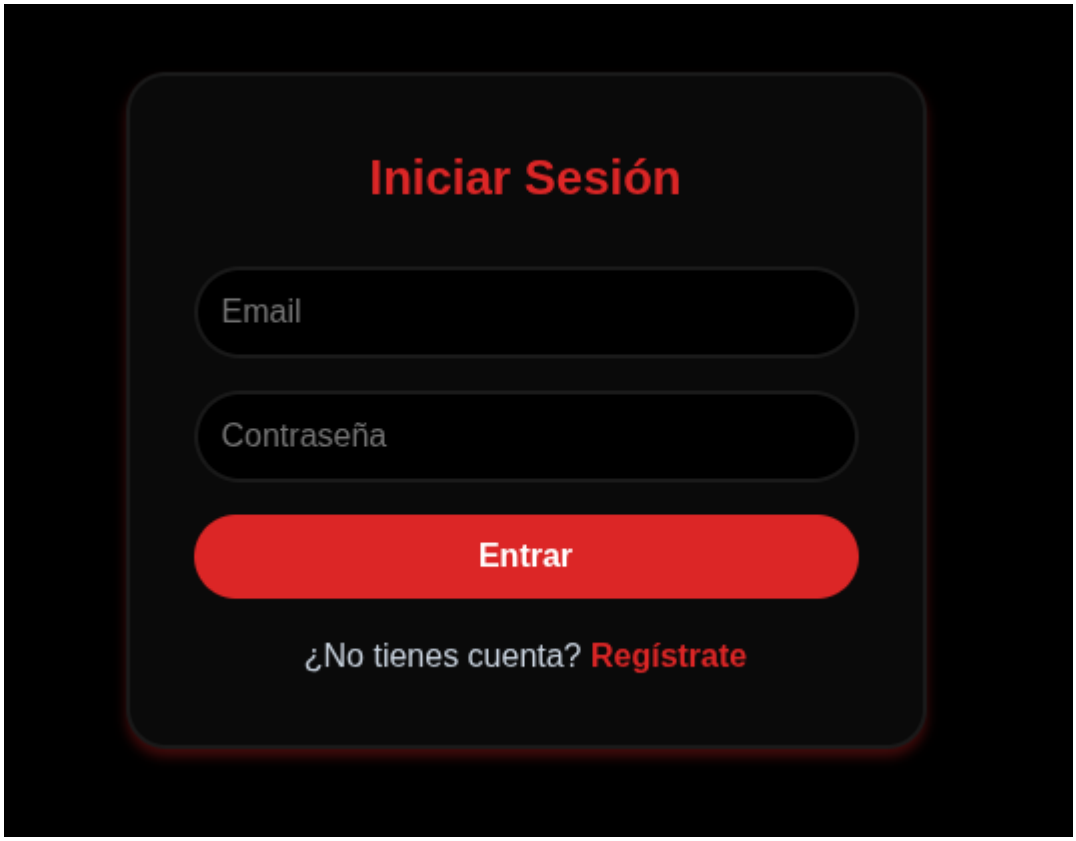


La interfaz muestra la barra de navegación principal de la aplicación web *Torko Motors*.
En la parte superior se incluyen los botones de navegación hacia las secciones Catálogo, Carrito, Bienvenido, e Iniciar Sesión, además de un selector de idioma (en este caso configurado en “ES”).
Debajo de la barra se encuentra un buscador de productos que permite filtrar resultados por categoría y rango de precios, con botones para aplicar o limpiar los filtros.
El diseño mantiene una estética oscura con detalles en rojo, reforzando la identidad visual de la marca.



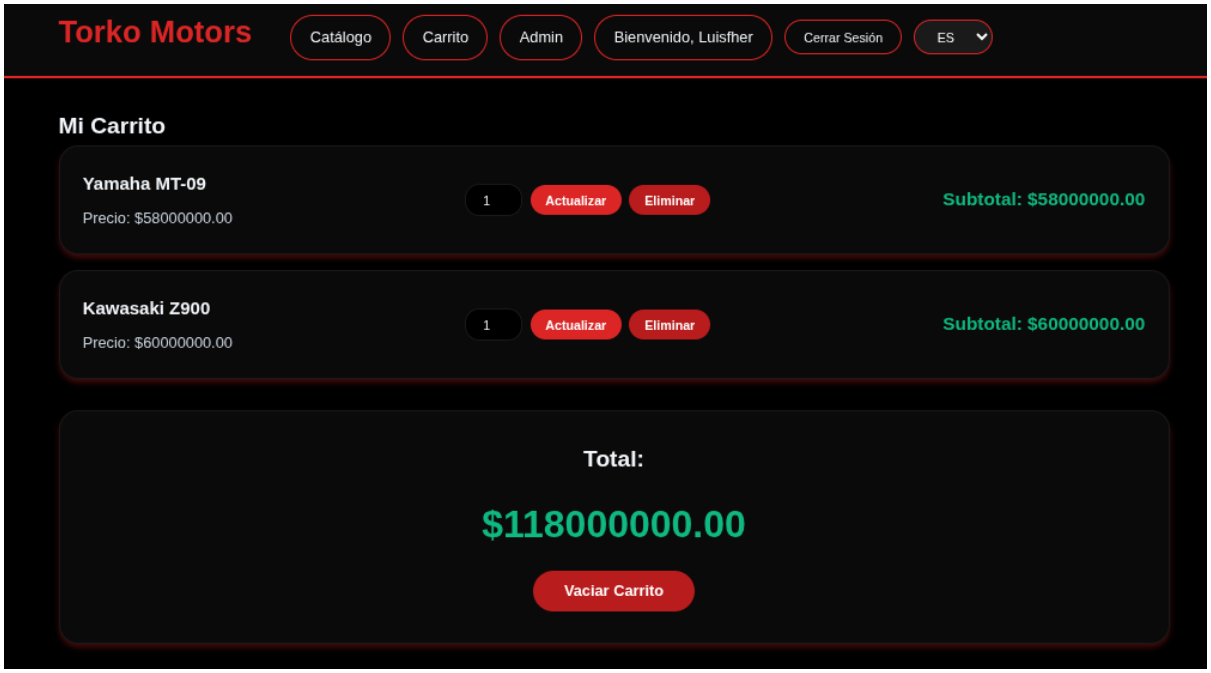
Esta imagen presenta el catálogo principal de productos, donde se muestran distintas motocicletas disponibles para la venta.
Cada producto se representa dentro de una tarjeta que incluye:

- Imagen del artículo.
- Nombre y breve descripción.
- Precio formateado.
- Cantidad disponible en stock.
- Botón “Agregar al Carrito”.
  El diseño prioriza la claridad y la presentación visual de los productos, facilitando al usuario la selección e interacción directa con el carrito de compras.

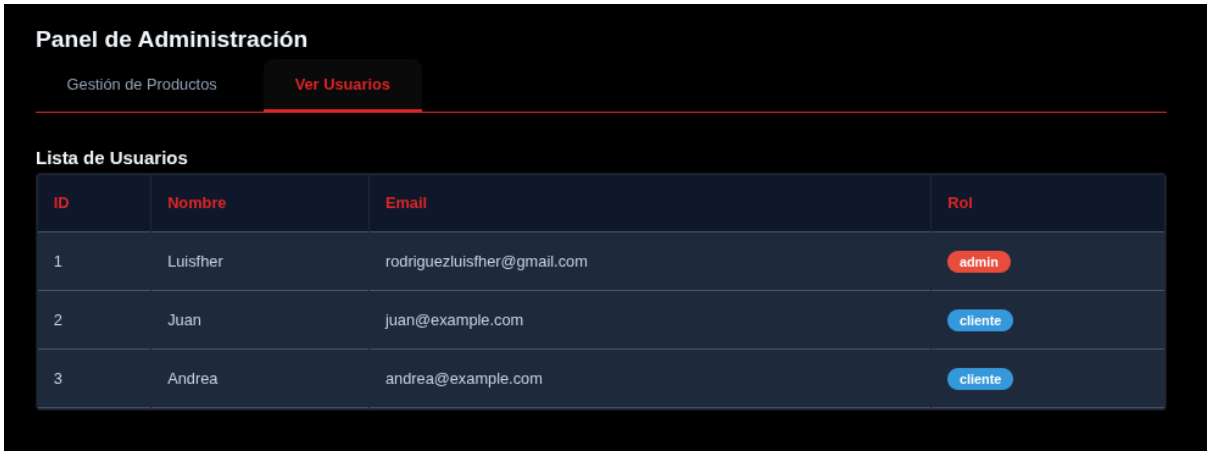


En esta pantalla se muestra el formulario de autenticación del sistema.
El usuario debe ingresar su correo electrónico y contraseña para acceder a su cuenta.
El botón “Entrar” valida las credenciales contra la base de datos y determina el tipo de usuario (cliente o administrador).
Además, se incluye un enlace que permite registrarse en caso de no tener cuenta:

“¿No tienes cuenta? Regístrate”.
El diseño mantiene la coherencia visual con la interfaz general del sistema, destacando el botón de acción en color rojo.



En esta sección se muestra el contenido actual del carrito de compras del usuario.
Cada producto agregado aparece con su nombre, precio unitario, cantidad seleccionada y subtotal.
El usuario puede actualizar la cantidad o eliminar un producto directamente desde esta vista.
En la parte inferior se muestra el total general del carrito y un botón “Vaciar Carrito” para eliminar todos los elementos.
En la barra de navegación superior se observan nuevas opciones visibles tras iniciar sesión como administrador: el botón “Admin” y el saludo personalizado “Bienvenido, Luisfher”, junto con la opción para cerrar sesión.



La sección Panel de Administración es accesible únicamente para los usuarios con rol de administrador a través del botón “Admin” ubicado en la barra de navegación.
En este panel se agrupan las principales herramientas de gestión del sistema:

- Gestión de Productos: permite al administrador agregar, editar o eliminar productos del catálogo.

- Ver Usuarios: muestra una lista completa de los usuarios registrados en la base de datos.

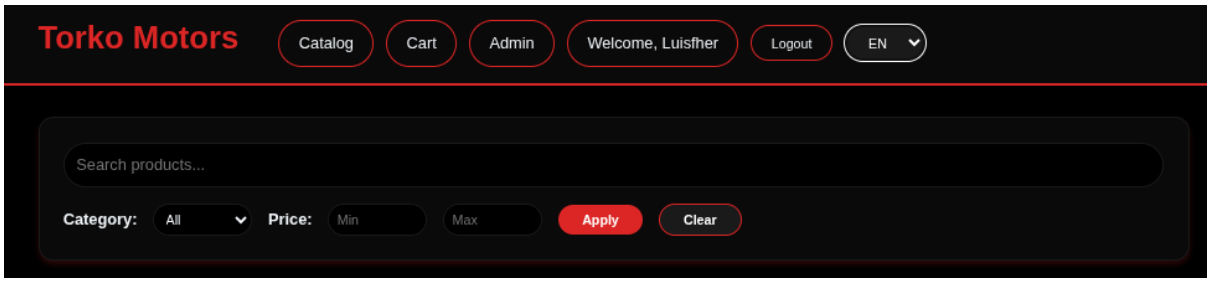
En la tabla de usuarios se presenta la siguiente información:

- ID: identificador único del usuario.
- Nombre: nombre del usuario registrado.
- Email: correo electrónico asociado a la cuenta.
- Rol: tipo de usuario dentro del sistema (por ejemplo, *admin* o *cliente*).

El panel mostrado en el ejemplo contiene tres registros:

1. Luisfher, con rol *admin*.
2. Juan, con rol *cliente*.
3. Andrea, también con rol *cliente*.

Esta vista centraliza las tareas administrativas, facilitando la supervisión de usuarios y el control de los productos disponibles en la plataforma.



Esta versión muestra la misma interfaz que la primera imagen, pero con el idioma cambiado a inglés mediante el selector desplegable.
Los textos de los botones y etiquetas del sistema se actualizan dinámicamente al idioma seleccionado, confirmando la implementación de la funcionalidad multilenguaje en la aplicación.

6.3.3. Conclusión

Las pruebas realizadas confirman que el sistema *Torko Motors* cumple con los requerimientos establecidos: registro e inicio de sesión de usuarios, gestión de productos, operaciones del carrito de compras y traducción dinámica de la interfaz. La aplicación funciona de manera estable tanto a nivel de servidor como en la experiencia del usuario final.

7. Pruebas

7.1 ¿Qué pruebas se hicieron?

Se realizaron pruebas funcionales y de integración tanto en el backend como en el frontend del sistema *Torko Motors*.
Las principales pruebas realizadas fueron:

- Conexión y creación de base de datos: verificación de la conexión a SQLite y generación automática de las tablas usuarios, productos y carrito.
- Pruebas de endpoints (API): mediante Postman se validaron las operaciones CRUD para usuarios, productos y carrito.
- Autenticación: registro e inicio de sesión de usuarios, comprobando la restricción de acceso de rutas para administradores.
- Gestión de productos: validación del alta, actualización y eliminación de productos, solo permitidos para el rol “admin”.
- Carrito de compras: agregar, modificar y eliminar productos del carrito, asegurando que el stock se actualiza correctamente.
- Interfaz de usuario: pruebas visuales en el navegador para confirmar la correcta carga de productos, actualización dinámica del carrito y funcionamiento del cambio de idioma (ES/EN).

7.2 Resultados esperados vs. resultados obtenidos

Los resultados obtenidos durante las pruebas fueron satisfactorios y coincidieron en su mayoría con los resultados esperados.
La base de datos se conectó correctamente al servidor, y las tablas fueron creadas de forma automática al iniciar la aplicación. Las pruebas de registro e inicio de sesión confirmaron que el sistema valida los datos de los usuarios y restringe el acceso a las funciones administrativas solo a aquellos con el rol correspondiente.

En las pruebas de productos, el sistema permitió al administrador agregar, actualizar y eliminar registros correctamente, mientras que los usuarios comunes recibieron el mensaje de restricción esperado (“Acceso denegado. Solo el administrador puede agregar productos”).

Las operaciones del carrito también respondieron según lo previsto: los productos se añadieron y eliminaron correctamente, el stock se actualizó de forma automática, y los subtotales y totales se calcularon sin errores. La interfaz del frontend reflejó adecuadamente los cambios realizados en el backend, mostrando mensajes claros cuando el stock era insuficiente o cuando se vaciaba el carrito.

Finalmente, el sistema de traducciones funcionó correctamente, cambiando todos los textos visibles al idioma seleccionado desde el menú desplegable.

7.3 ¿Cómo se corrigieron errores encontrados?

Durante las pruebas iniciales se identificaron algunos errores menores que fueron corregidos antes de la validación final.
Uno de los primeros problemas detectados fue el error **403 (Forbidden)** al intentar agregar productos; esto se debía a que el usuario utilizado no tenía el rol de administrador. Se solucionó actualizando manualmente el campo rol a “admin” en la base de datos.

También se encontró que, al eliminar productos del carrito, el stock del inventario no se actualizaba correctamente. Esto se corrigió agregando una sentencia SQL adicional dentro de la función que elimina los productos, para devolver la cantidad eliminada al stock original.

Otro error frecuente fue la falta de conexión entre el frontend y el servidor, causado por una dirección incorrecta en la constante API_URL. Al corregirla con http://localhost:4000, el frontend logró comunicarse sin problemas con la API.

Por último, se detectaron traducciones incompletas en el archivo translations.js, por lo que se añadieron nuevas claves de texto, y se ajustó el código para que los scripts se ejecutaran después de cargar el DOM, evitando errores de visualización en el navegador.

Después de estas correcciones, todas las pruebas se ejecutaron exitosamente y el sistema funcionó de forma estable y coherente.

8. Conclusiones y recomendaciones

8.1 Aprendizajes del proyecto

A lo largo del desarrollo del proyecto Torko Motors, se logró comprender de forma práctica cómo se integran los distintos componentes de una aplicación web moderna.
Se aprendió a estructurar y conectar un backend con Node.js y Express a una base de datos SQLite, implementando rutas seguras, control de roles y validaciones de usuario. En paralelo, se desarrolló un frontend funcional con HTML, CSS y JavaScript, capaz de comunicarse con el servidor mediante peticiones fetch y reflejar los datos en tiempo real.

Además, el proceso permitió afianzar conocimientos sobre el manejo de APIs REST, pruebas con Postman, gestión de dependencias, y organización del código en proyectos escalables. También se fortalecieron habilidades relacionadas con la depuración de errores, el uso de herramientas como DB Browser y VSCode, y la importancia de mantener una buena comunicación entre las capas del sistema.

8.2 ¿Qué mejoraría?

Aunque el sistema Torko Motors cumple con todas las funcionalidades planteadas, existen varios aspectos que podrían mejorarse para optimizar la seguridad, la experiencia del usuario y la escalabilidad del proyecto.

En primer lugar, sería conveniente implementar un sistema de encriptación de contraseñas, garantizando la protección de los datos de los usuarios. Además, la integración de una autenticación basada en tokens JWT permitiría manejar sesiones de forma más segura y confiable.

También se considera importante migrar a una base de datos más robusta, como MySQL o PostgreSQL, lo que facilitaría la gestión de mayores volúmenes de información y usuarios.

Otro punto por mejorar es el sistema de alertas y notificaciones del frontend. Actualmente, algunas alertas se muestran con mensajes simples del navegador, como *“Product added to cart”*, lo cual podría reemplazarse por notificaciones personalizadas dentro de la página, con un diseño más intuitivo y visualmente integrado.

Finalmente, una de las mejoras más relevantes sería la implementación de un sistema de pagos en línea, que permitiría completar el proceso de compra

directamente desde la aplicación, brindando una experiencia más completa y profesional al usuario final.

8.3 Recomendaciones para otros estudiantes

A los estudiantes que desarrollen proyectos similares, se les recomienda planificar la estructura del sistema desde el inicio, definiendo claramente las rutas, las tablas y las responsabilidades de cada módulo. Esto evita confusiones al momento de integrar las partes del proyecto.

Es fundamental probar constantemente cada componente (backend, frontend y base de datos) de manera independiente antes de unificarlos, utilizando herramientas como Postman para las APIs y DB Browser para verificar los datos.

También se recomienda mantener una organización clara de carpetas y archivos, utilizar nombres descriptivos y comentar el código para facilitar la comprensión y futuras modificaciones. Finalmente, es importante hacer respaldos frecuentes del proyecto en GitHub, tanto para evitar pérdidas de información como para llevar un control de versiones y poder colaborar fácilmente en equipo.

9. Referencias

- ChatGPT (OpenAI, 2025). Orientación en la estructuración del backend del proyecto *Torko Motors*, incluyendo redacción de documentación y generación de ejemplos de código.
  Disponible en: https://chat.openai.com
- Claude (Anthropic, 2025). Asistencia técnica complementaria en la estructuración del frontend. Además, asesoría en depuración de código y comprensión de rutas del backend en Node.js.
  Disponible en: https://claude.ai
- W3Schools. Tutoriales y referencias sobre HTML, CSS, JavaScript y manejo del DOM.
  Disponible en: https://www.w3schools.com
- MDN Web Docs (Mozilla Developer Network). Documentación oficial sobre estándares web, JavaScript, Fetch API y buenas prácticas en desarrollo frontend.
  Disponible en: https://developer.mozilla.org
- Node.js Official Documentation. Guía oficial sobre instalación, uso de módulos, Express.js y manejo de peticiones HTTP.
  Disponible en: https://nodejs.org/en/docs
- Express.js Documentation. Referencia para la creación de rutas, middlewares y conexión con bases de datos en aplicaciones web.
  Disponible en: https://expressjs.com
- SQLite Documentation. Manual y referencias de uso de comandos SQL, estructura de tablas y consultas.
  Disponible en: https://www.sqlite.org/docs.html
- Postman Learning Center. Guía sobre la creación y prueba de endpoints para verificar la correcta comunicación entre backend y frontend.
  Disponible en: https://learning.postman.com
- DB Browser for SQLite. Herramienta utilizada para visualizar y manipular la base de datos local del proyecto.
  Disponible en: https://sqlitebrowser.org
- GitHub Docs. Documentación sobre control de versiones, sincronización de repositorios y despliegue de proyectos.
  Disponible en: https://docs.github.com