

# **INGENIERIA WEB I**

**Sebastian Perez Riaño**

**Universidad Manuela Beltrán**

**Ingeniería de software**

## 1. INTRODUCCION:

En este documento se explicará la lógica detrás de los nuevos códigos implementados, así mismo se explicarán algunos términos, nuevos y conocidos, aplicados en el proyecto que fueron clave para lograr llevar a cabo este proyecto. También, se mostrarán los nuevos códigos implementados con su debida explicación, las tablas y el ejemplo usado para verificar su correcto funcionamiento.

## 2. ALGUNOS TERMINOS IMPORTANTES:

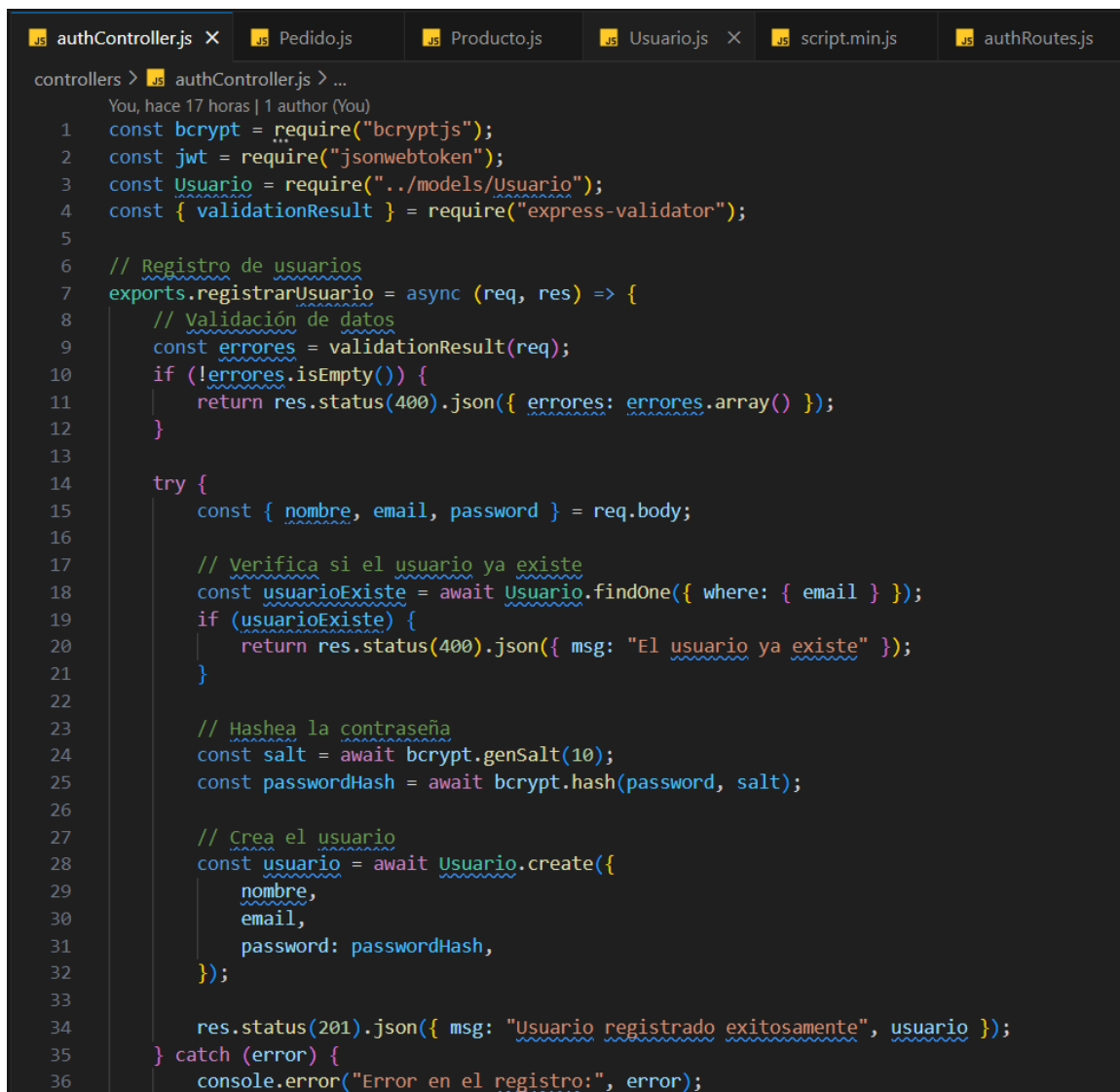
- **Node.js:** Es un entorno de ejecución de JavaScript en el servidor, este es usado para desarrollar aplicaciones Back-End.
- **Express.js:** Este es el Framework para Node.js que facilita la creación del servidor.
- **ORM (Object-Relational Mapping):** Esta es una técnica que permite la interacción con la base de datos relacional mediante objetos en el código, esto para no escribir las consultas SQL manualmente.
- **Sequelize:** Es un ORM para Node.js que facilita la gestión de bases de datos SQL en JavaScript.
- **API REST:** Interfaz que permite la comunicación del Back-end con el Front-End mediante solicitudes HTTP.
- **Endpoints:** URLs específicas de un servidor que reciben y responden solicitudes de clientes.
- **Middleware:** Esta es una función en Express.js que se ejecuta entre las solicitudes del cliente y las respuestas del servidor, esto es para la autenticación, la gestión de datos y más.

- **JSON (JavaScript Object Notation):** Este es un formato ligero de intercambio de datos utilizado para enviar información entre el Front-End y el Back-End.

### 3. EXPLICACION CODIGOS:

En esta sección se explicarán los nuevos códigos, para no hacer muy largo el PDF se eligió la opción de tomar unos pantallazos al código y poner la imagen, si se desea ver el código este estará en el repositorio GitHub.

- **authControllers.js:**



```
authController.js X  Pedido.js  Producto.js  Usuario.js X  script.min.js  authRoutes.js
controllers > authController.js > ...
You, hace 17 horas | 1 author (You)
1  const bcrypt = require("bcryptjs");
2  const jwt = require("jsonwebtoken");
3  const Usuario = require("../models/Usuario");
4  const { validationResult } = require("express-validator");
5
6  // Registro de usuarios
7  exports.registrarUsuario = async (req, res) => {
8      // Validación de datos
9      const errores = validationResult(req);
10     if (!errores.isEmpty()) {
11         return res.status(400).json({ errores: errores.array() });
12     }
13
14     try {
15         const { nombre, email, password } = req.body;
16
17         // Verifica si el usuario ya existe
18         const usuarioExiste = await Usuario.findOne({ where: { email } });
19         if (usuarioExiste) {
20             return res.status(400).json({ msg: "El usuario ya existe" });
21         }
22
23         // Hashea la contraseña
24         const salt = await bcrypt.genSalt(10);
25         const passwordHash = await bcrypt.hash(password, salt);
26
27         // Crea el usuario
28         const usuario = await Usuario.create({
29             nombre,
30             email,
31             password: passwordHash,
32         });
33
34         res.status(201).json({ msg: "Usuario registrado exitosamente", usuario });
35     } catch (error) {
36         console.error("Error en el registro:", error);
37     }
38 }
```

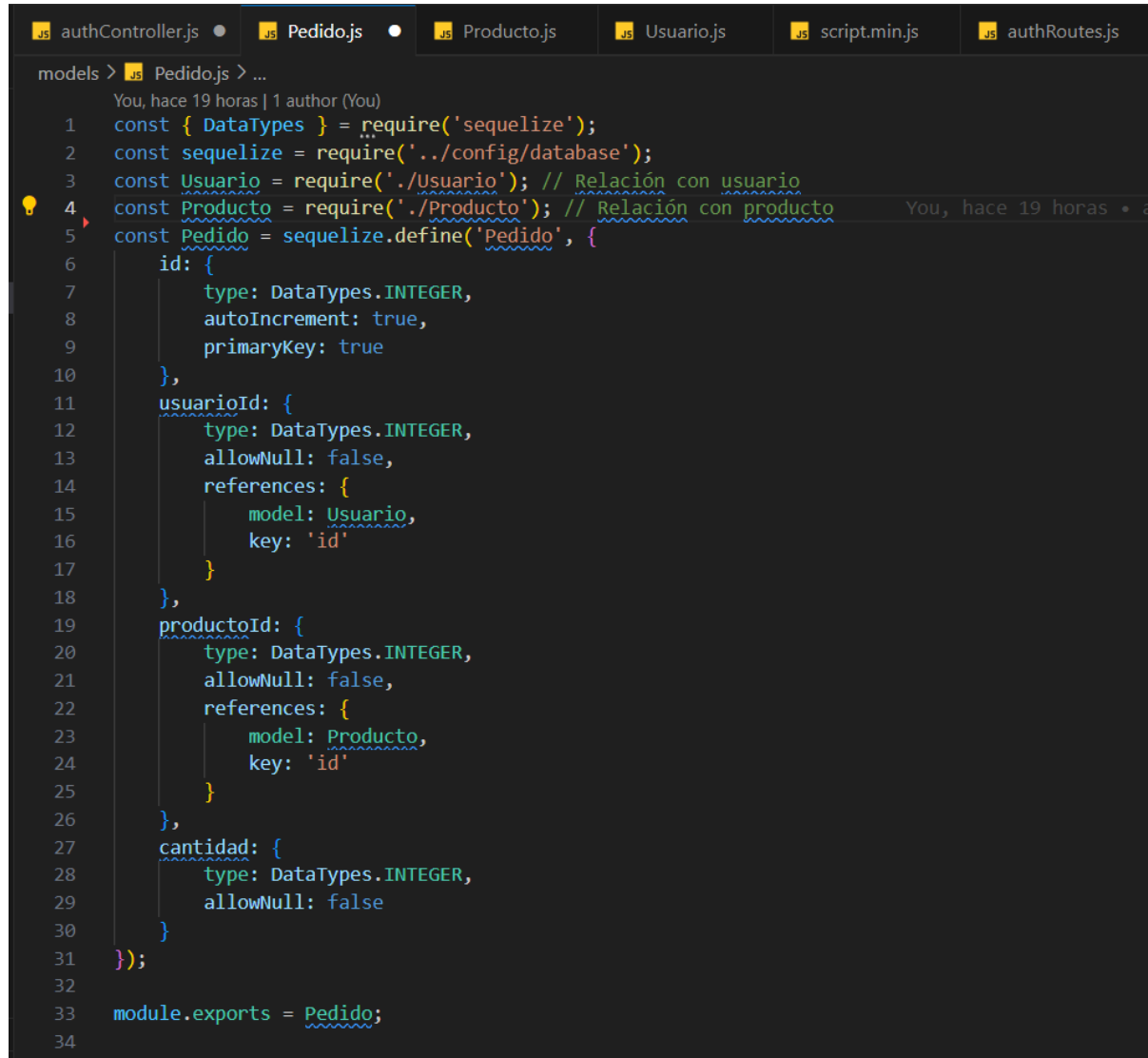
```
authController.js • Pedido.js Producto.js X Usuario.js script.min.js authRoutes.js productos.js
controllers > authController.js > registrarUsuario > registrarUsuario
36 // Inicio de sesión
37 exports.iniciarSesion = async (req, res) => {
38   try {
39     const { email, password } = req.body;
40
41     // Buscar usuario por email
42     const usuario = await Usuario.findOne({ where: { email } });
43
44     if (!usuario) {
45       return res.status(400).json({ msg: "Credenciales incorrectas" });
46     }
47
48     // Comparar contraseñas
49     const esCorrecto = await bcrypt.compare(password, usuario.password);
50     if (!esCorrecto) {
51       return res.status(400).json({ msg: "Credenciales incorrectas" });
52     }
53
54     // Verifica que la variable de entorno esté definida
55     if (!process.env.JWT_SECRET) {
56       return res.status(500).json({ msg: "Error en la configuración del servidor" });
57     }
58
59     // Generar JWT
60     const token = jwt.sign(
61       { id: usuario.id, email: usuario.email },
62       process.env.JWT_SECRET,
63       { expiresIn: "1h" }
64     );
65
66     res.json({ token, usuario: { id: usuario.id, nombre: usuario.nombre, email: usuario.email } });
67   } catch (error) {
68     console.error("Error en inicio de sesión:", error);
69     res.status(500).json({ msg: "Error en el servidor" });
70   }
71 };
```

**Explicación:** En este código se maneja lo que es la autenticación de un usuario en la aplicación,

lo primero que hace es definir las dependencias necesarias como: bcryptj para el cifrado de claves, jsonwebtoken para la generación de tokens de autenticación y express-validator para validar los datos que recibe.

Despues encontramos algunas funciones como registrarUsuario que verifica si los datos enviados por el usuario son válidos y si todo esta bien, responde con un mensaje de éxito. La función iniciarSesion recibe las credenciales del usuario, busca el correo en la base de datos y compara la contraseña ingresada con la almacenada, si sucede algún error durante el proceso este devuelve un mensaje de error adecuado.

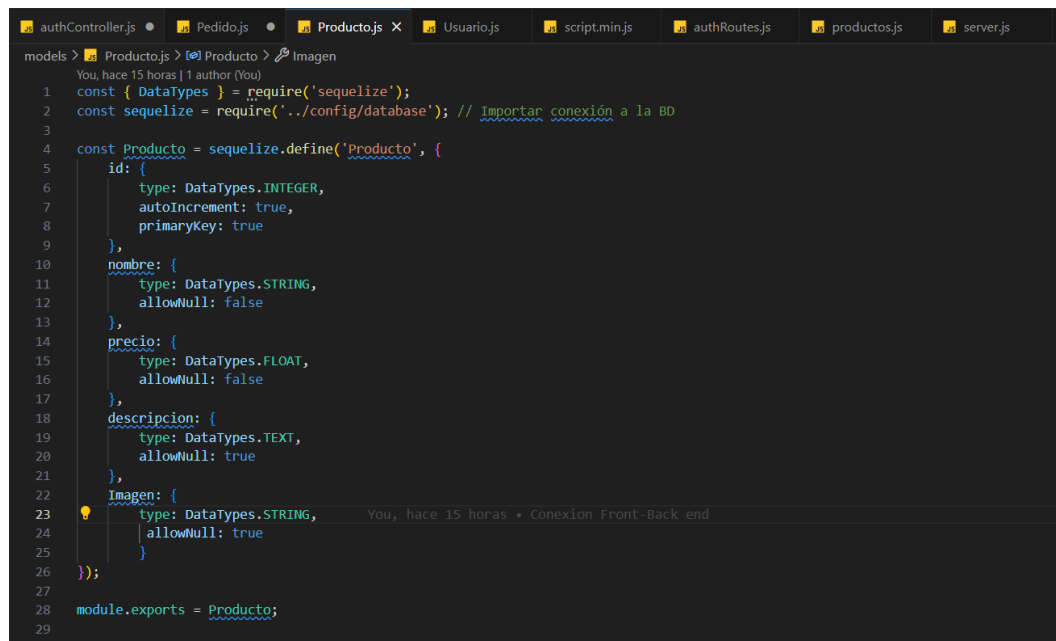
- **Pedido.js:**



```
models > JS Pedido.js > ...
You, hace 19 horas | 1 author (You)
1  const { DataTypes } = require('sequelize');
2  const sequelize = require('../config/database');
3  const Usuario = require('./Usuario'); // Relación con usuario
4  const Producto = require('./Producto'); // Relación con producto
5  const Pedido = sequelize.define('Pedido', {
6      id: {
7          type: DataTypes.INTEGER,
8          autoIncrement: true,
9          primaryKey: true
10     },
11     usuarioId: {
12         type: DataTypes.INTEGER,
13         allowNull: false,
14         references: {
15             model: Usuario,
16             key: 'id'
17         }
18     },
19     productId: {
20         type: DataTypes.INTEGER,
21         allowNull: false,
22         references: {
23             model: Producto,
24             key: 'id'
25         }
26     },
27     cantidad: {
28         type: DataTypes.INTEGER,
29         allowNull: false
30     }
31 });
32
33 module.exports = Pedido;
34
```

**Explicación:** El archivo define un modelo Sequelize llamado Pedido, que representa una tabla en la base datos. La tabla tiene cuatro campos: el id que es la clave y autoincremental, usuarioId este refiere al usuario que realiza el pedido y es la clave foránea de la tabla Usuario, productId es la referencia al producto que se pide y es la clave foránea de la tabla Producto, por ultimo cantidad es el número del producto pedido.

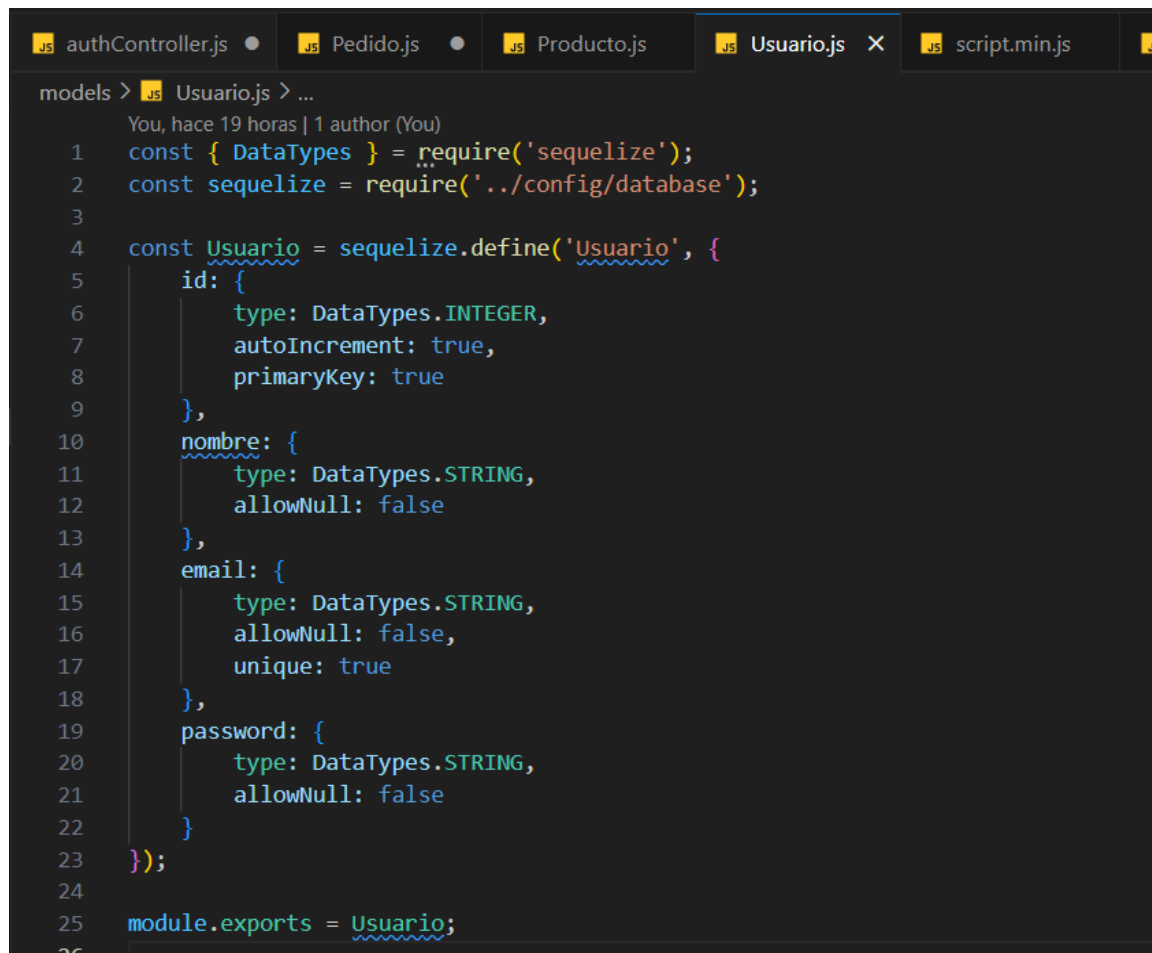
- **Producto.js:**



```
models > Producto.js > Producto > Imagen
You, hace 15 horas | 1 author (You)
1 const { DataTypes } = require('sequelize');
2 const sequelize = require('../config/database'); // Importar conexión a la BD
3
4 const Producto = sequelize.define('Producto', {
5   id: {
6     type: DataTypes.INTEGER,
7     autoIncrement: true,
8     primaryKey: true
9   },
10  nombre: {
11    type: DataTypes.STRING,
12    allowNull: false
13  },
14  precio: {
15    type: DataTypes.FLOAT,
16    allowNull: false
17  },
18  descripcion: {
19    type: DataTypes.TEXT,
20    allowNull: true
21  },
22  imagen: {
23    type: DataTypes.STRING,
24    allowNull: true
25  }
26 });
27
28 module.exports = Producto;
29
```

**Explicación:** Similar al anterior, define el modelo Producto en sequelize. Esta tabla tiene cinco columnas: id que es la clave primaria y autoincremental, nombre que es el nombre del producto y no puede ser nulo, precio que hace referencia al precio del producto y también es obligatorio, descripción es una descripción opcional del producto, por ultimo Imagen que es la ruta para la imagen del producto.

- **Usuario.js:**



```
models > .js Usuario.js > ...
You, hace 19 horas | 1 author (You)
1  const { DataTypes } = require('sequelize');
2  const sequelize = require('../config/database');
3
4  const Usuario = sequelize.define('Usuario', {
5    id: {
6      type: DataTypes.INTEGER,
7      autoIncrement: true,
8      primaryKey: true
9    },
10   nombre: {
11     type: DataTypes.STRING,
12     allowNull: false
13   },
14   email: {
15     type: DataTypes.STRING,
16     allowNull: false,
17     unique: true
18   },
19   password: {
20     type: DataTypes.STRING,
21     allowNull: false
22   }
23 });
24
25 module.exports = Usuario;
26
```

**Explicación:** Este archivo define el modelo Usuario en sequelize. La tabla tiene cuatro columnas: id que es la clave primaria y autoincrementar, nombre es el nombre del usuario y es no nulo, email es el correo electrónico único y obligatorio para cada usuario, password que es la contraseña del usuario y es obligatorio.

- **Script.min.js:**

```
authController.js • Pedido.js • Producto.js X Usuario.js script.min.js X authRoutes.js productos.js server.js
Public > scripts > script.min.js > document.addEventListener("DOMContentLoaded") callback > vaciarCarritoBtn.addEventListener("click") callback
1 document.addEventListener("DOMContentLoaded", async () => {
2   const productosContainer = document.querySelector("#productos");
3   const carritoContainer = document.querySelector("#carrito");
4   let carrito = JSON.parse(localStorage.getItem("carrito")) || [];
5
6   // Función para actualizar el carrito en el DOM
7   function actualizarCarrito() {
8     let contenido = document.querySelector("#carrito .contenido");
9     if (!contenido) {
10       contenido = document.createElement("div");
11       contenido.classList.add("contenido");
12       carritoContainer.appendChild(contenido);
13     }
14
15     contenido.innerHTML = "";
16
17     if (carrito.length === 0) {
18       contenido.innerHTML = "<p>Tu carrito está vacío</p>";
19     } else {
20       carrito.forEach((producto, index) => {
21         let div = document.createElement("div");
22         div.classList.add("producto-carrito");
23         div.innerHTML = `
24           <p>${producto.nombre} - ${producto.precio.toFixed(2)}</p>
25           <button class="eliminar" data-index=${index}>X</button>
26         `;
27         contenido.appendChild(div);
28       });
29
30       // Agregar eventos a los botones de eliminar
31       document.querySelectorAll(".eliminar").forEach(btn => {
32         btn.addEventListener("click", (e) => {
33           let index = e.target.getAttribute("data-index");
34           carrito.splice(index, 1);
35           guardarCarrito();
36         });
37     });
38   }
39 }
40
41 // Función para guardar el carrito en localStorage
42 function guardarCarrito() {
43   localStorage.setItem("carrito", JSON.stringify(carrito));
44   actualizarCarrito();
45 }
46
47 // Cargar productos desde la API
48 try {
49   const response = await fetch("http://localhost:3000/productos");
50   const productos = await response.json();
51
52   productosContainer.innerHTML = "<h2>Lista de Productos</h2>";
53
54   productos.forEach(producto => {
55     let div = document.createElement("div");
56     div.classList.add("producto");
57     div.innerHTML = `
58       <h3>${producto.nombre}</h3>
59       <p class="precio">${producto.precio}</p>
60       <button class="agregar" data-id=${producto.id}>Añadir al carrito</button>
61     `;
62     productosContainer.appendChild(div);
63   });
64
65   // Agregar eventos a los botones de "Añadir al carrito"
66   document.querySelectorAll(".agregar").forEach(btn => {
67     btn.addEventListener("click", (e) => {
68       let productoElemento = e.target.parentElement;
69       let nombre = productoElemento.querySelector("h3").textContent;
70       let precio = parseFloat(productoElemento.querySelector(".precio").textContent.replace("$", ""));
71       carrito.push([ nombre, precio ]);
72     });
73   });
74 }
```

```
authController.js • Pedido.js • Producto.js X Usuario.js script.min.js X authRoutes.js productos.js server.js
Public > scripts > script.min.js > document.addEventListener("DOMContentLoaded") callback > actualizarCarrito
1 document.addEventListener("DOMContentLoaded", async () => {
7   function actualizarCarrito() {
37     });
38   }
39 }
40
41 // Función para guardar el carrito en localStorage
42 function guardarCarrito() {
43   localStorage.setItem("carrito", JSON.stringify(carrito));
44   actualizarCarrito();
45 }
46
47 // Cargar productos desde la API
48 try {
49   const response = await fetch("http://localhost:3000/productos");
50   const productos = await response.json();
51
52   productosContainer.innerHTML = "<h2>Lista de Productos</h2>";
53
54   productos.forEach(producto => {
55     let div = document.createElement("div");
56     div.classList.add("producto");
57     div.innerHTML = `
58       <h3>${producto.nombre}</h3>
59       <p class="precio">${producto.precio}</p>
60       <button class="agregar" data-id=${producto.id}>Añadir al carrito</button>
61     `;
62     productosContainer.appendChild(div);
63   });
64
65   // Agregar eventos a los botones de "Añadir al carrito"
66   document.querySelectorAll(".agregar").forEach(btn => {
67     btn.addEventListener("click", (e) => {
68       let productoElemento = e.target.parentElement;
69       let nombre = productoElemento.querySelector("h3").textContent;
70       let precio = parseFloat(productoElemento.querySelector(".precio").textContent.replace("$", ""));
71       carrito.push([ nombre, precio ]);
72     });
73   });
74 }
```



```

authController.js • Pedidos.js • Productos.js X Usuario.js script.min.js X authRoutes.js productos.js server.js
Public > scripts > script.min.js > document.addEventListener("DOMContentLoaded") callback
1  document.addEventListener("DOMContentLoaded", async () => {
66      document.querySelectorAll(".agregar").forEach(btn => {
67          btn.addEventListener("click", (e) => {
71              carrito.push({ nombre, precio });
72              guardarCarrito();
73          });
74      });
75
76  } catch (error) {
77      console.error("❌ Error al obtener productos:", error);
78  }
79
80  // Botón para vaciar el carrito
81  const vaciarCarritoBtn = document.createElement("button");
82  vaciarCarritoBtn.textContent = "Vaciar carrito";
83  vaciarCarritoBtn.addEventListener("click", () => {
84      carrito = [];
85      localStorage.removeItem("carrito");
86      actualizarCarrito();
87  });
88  carritoContainer.appendChild(vaciarCarritoBtn);
89
90  // Ejecutar pruebas
91  function testAgregarProducto() {
92      carrito = [];
93      carrito.push({ nombre: "Mochila", precio: 20 });
94      console.assert(carrito.length === 1, "❌ Error: No se agregó el producto al carrito");
95      console.assert(carrito[0].nombre === "Mochila", "❌ Error: Nombre del producto incorrecto");
96      console.assert(carrito[0].precio === 20, "❌ Error: Precio del producto incorrecto");
97      console.log("✅ Prueba testAgregarProducto pasada");
98  }
99
100  function testEliminarProducto() {
101      carrito = [{ nombre: "Bolso", precio: 30 }];
102      carrito.splice(0, 1);
103      console.assert(carrito.length === 0, "❌ Error: No se eliminó el producto del carrito");
104      console.log("✅ Prueba testEliminarProducto pasada");
105  }
106
107  function testVaciarCarrito() {
108      carrito = [{ nombre: "Zapato", precio: 50 }, { nombre: "Reloj", precio: 100 }];
109      carrito = [];
110      console.assert(carrito.length === 0, "❌ Error: No se vació el carrito correctamente");
111      console.log("✅ Prueba testVaciarCarrito pasada");
112  }
113
114  // Cargar carrito almacenado en localStorage
115  actualizarCarrito();
116
117  // Ejecutar pruebas
118  testAgregarProducto();
119  testEliminarProducto();
120  testVaciarCarrito();
121  });
122

```

**Explicación:** En este archivo se maneja la lógica de interacción en el Front-End de la tienda permitiendo que los usuarios puedan agregar productos al carrito, ver el contenido del carrito y vaciarlo. Ahora explicare un poco el funcionamiento:

1) Inicialización:

- Se espera que el contenido del DOM se cargue y luego se inicializan las variables `productosContainer` y `carritoContainer` para manejar los elementos del DOM donde se mostraran los productos y el carrito.
- Se recupera el carrito desde `localStorage`, si es que existe y si no se inicializa como un arreglo vacío.

## 2) Actualizar el carrito:

- La función `actualizarCarrito()` muestra el contenido del carrito en el DOM y si el carrito está vacío muestra un mensaje y si no genera la lista con los productos agregados mostrando su nombre, precio y botón para eliminarlo.

## 3) Guardar el carrito:

- La función `guardarCarrito()` guarda el estado actual del carrito en `localStorage` y actualiza la vista.

## 4) Cargar productos desde la API:

- En este se hace la solicitud de `fetch` para obtener los productos desde una API (<http://localhost:3000/productos>). Los productos se muestran en el DOM con un botón de “Añadir al carrito” y al hacer clic con ese botón el producto se agrega al carrito y se guarda en `localStorage`.

## 5) Vaciar carrito:

- Se crea un botón para iniciar el carrito y al hacer clic en el botón se elimina todo el contenido del carrito y actualiza el `localStorage` y el DOM.

## 6) Pruebas:

- Hay tres funciones de prueba (`testAgregarProducto`, `testEliminarProducto`, `testVaciarCarrito`) que verifican que los productos se agreguen, eliminen y se vacíen correctamente del carrito.

Además usa `console.assert()` para asegurarse de que el comportamiento sea el esperado y reportan el resultado de la consola.

- **authRoutes.js:**

```
authController.js • Pedidos.js • Productos.js X Usuario.js script.min.js authRoutes.js X productos.js server.js
routes > authRoutes.js > router.post('/register') callback
You, hace 17 horas | 1 author (You)
1 const express = require('express');
2 const bcrypt = require('bcrypt');
3 const { Usuario } = require('../models/Usuario');
4 const jwt = require('jsonwebtoken');
5
6 const router = express.Router();
7
8 // Ruta para registrar un usuario
9 router.post('/register', async (req, res) => {
10   const { nombre, email, password } = req.body;
11
12   try {
13     // Verificar si el usuario ya existe en la BD
14     const usuarioExistente = await Usuario.findOne({ where: { email } });
15     if (usuarioExistente) {
16       return res.status(400).json({ message: "El usuario ya está registrado" });
17     }
18     // Encriptar la contraseña
19     const hashedPassword = await bcrypt.hash(password, 10);
20
21     // Guardar en la base de datos
22     const usuario = await Usuario.create({
23       nombre,
24       email,
25       password: hashedPassword
26     });
27
28     res.status(201).json({ message: "Usuario registrado con éxito", usuario });
29   } catch (error) {
30     res.status(500).json({ message: "Error en el servidor", error });
31   }
32 }
33 );
34
35 // Ruta para iniciar sesión
36 router.post('/login', async (req, res) => {
```

```
36 router.post('/login', async (req, res) => {
37   const { email, password } = req.body;
38
39   try {
40     // Verificar si el usuario existe
41     const user = await Usuario.findOne({ where: { email } });
42     if (!user) {
43       return res.status(404).json({ message: 'Usuario no encontrado' });
44     }
45
46     // Verificar la contraseña
47     const isMatch = await bcrypt.compare(password, user.password);
48     if (!isMatch) {
49       return res.status(401).json({ message: 'Contraseña incorrecta' });
50     }
51
52     // Generar token JWT
53     const token = jwt.sign({ id: user.id, email: user.email }, 'secreto', { expiresIn: '1h' });
54
55     res.json({ message: 'Inicio de sesión exitoso', token });
56   } catch (error) {
57     res.status(500).json({ message: 'Error en el servidor', error });
58   }
59 });
60
61 module.exports = router;
```

**Explicación:** Este archivo maneja la autenticación de usuarios en la API con Express y sequelize.

-Registro de usuario (/register): Recibe el nombre, email y password para verificar el usuario y ver si está registrado, después encripta la contraseña con bcrypt y guarda el usuario en la base de datos, al final responde con un mensaje de éxito o error.

-Inicio de sesión (/login): este recibe el email y password para buscar el usuario en la base de datos, si existe compara la contraseña ingresada con la almacenada, si coinciden entonces genera un token JWT con una duración de 1 hora y lo devuelve, en caso de que pase algún error responde con el mensaje correspondiente a dicho error.

- **productos.js:**

```
authController.js • Pedido.js • Producto.js • Usuario.js • script.min.js • authRoutes.js • productos.js X server.js
routes > productos.js > router.post('/') callback
You, hace 21 horas | 1 author (You)
1 const express = require('express');
2 const router = express.Router();
3 const Producto = require('../models/Producto');
4
5 // Obtener todos los productos
6 router.get('/', async (req, res) => {
7   try {
8     const productos = await Producto.findAll();
9     res.json(productos);
10  } catch (error) {
11    console.error('❌ Error al obtener productos:', error);
12    res.status(500).json({ error: 'Error al obtener productos' });
13  }
14 });
15
16 // Agregar un nuevo producto
17 router.post('/', async (req, res) => {
18   try {
19     const nuevoProducto = await Producto.create(req.body);
20     res.status(201).json(nuevoProducto);
21   } catch (error) {
22     console.error('❌ Error al agregar producto:', error);
23     res.status(500).json({ error: 'Error al agregar producto' });
24   }
25 });
26
27 // Actualizar un producto por ID
28 router.put('/:id', async (req, res) => {
29   try {
30     const { id } = req.params;
31     const [updated] = await Producto.update(req.body, { where: { id } });
32     if (updated) {
33       res.json({ message: '✅ Producto actualizado correctamente' });
34     } else {
35       res.status(404).json({ error: 'Producto no encontrado' });
36     }
37   }
38 });
```

```

36     }
37   } catch (error) {
38     console.error('❌ Error al actualizar producto:', error);
39     res.status(500).json({ error: 'Error al actualizar producto' });
40   }
41 });
42
43 // Eliminar un producto por ID
44 router.delete('/:id', async (req, res) => {
45   try {
46     const { id } = req.params;
47     const deleted = await Producto.destroy({ where: { id } });
48     if (deleted) {
49       res.json({ message: '✅ Producto eliminado correctamente' });
50     } else {
51       res.status(404).json({ error: 'Producto no encontrado' });
52     }
53   } catch (error) {
54     console.error('❌ Error al eliminar producto:', error);
55     res.status(500).json({ error: 'Error al eliminar producto' });
56   }
57 });
58
59 module.exports = router;
60
61

```

**Explicación:** Este archivo define las rutas para gestionar productos en una API con Express y sequelize.

- Obtener todos los productos (GET /): Consulta todos los productos en la base de datos y los devuelve de forma JSON

- Agregar un nuevo producto(POST/): Este recibe los datos del producto en el cuerpo solicitado y los guarda en la base de datos, después responde con el producto creado o si falla responde con un error.

- Actualizar un producto(PUT/:id): este recibe un id en la URL y los datos actualizados en el cuerpo solicitado, si existe lo actualizara y devuelve un mensaje de éxito, si no responderá con un error.

- Eliminar un producto(DELETE/: id): Este recibe un id en la URL y elimina el producto correspondiente, si se elimina de forma exitosa devuelve un mensaje de confirmación, si no, responde con un error.

- **Server.js**

```
server.js > conectarBD
You, hace 16 horas | 1 author (You)
1 const express = require('express');
2 const cors = require('cors');
3 const sequelize = require('./config/database');
4 const productosRoutes = require('./routes/productos');
5 const authRoutes = require('./routes/authRoutes');
6
7
8 const app = express();
9
10 app.use(cors());
11 app.use(express.json());
12
13 app.use('/imagenes', express.static('Public/Imagenes'));
14
15 // Rutas
16 app.use('/productos', productosRoutes);
17 app.use('/auth', authRoutes);
18
19 // Conecta a la base de datos
20 async function conectarBD() {
21   try {
22     await sequelize.authenticate();
23     console.log('✅ Conexión a la base de datos exitosa.');
```

You, hace 21 horas • agregar tablas y mas

```
24     await sequelize.sync({ alter: true }); // Evita borrar datos
25     console.log('✅ Modelos sincronizados correctamente.');
```

You, hace 21 horas • agregar tablas y mas

```
26   } catch (error) {
27     console.error('❌ Error al conectar a la base de datos:', error);
28   }
29 }
30 conectarBD();
31
32 // Iniciar el servidor
33 const PORT = 3000;
34 app.listen(PORT, () => {
35   console.log('🔥 Servidor corriendo en http://localhost:${PORT}');
```

You, hace 21 horas • agregar tablas y mas

```
36 });
```

**Explicación:** En este archivo es donde se configura y ejecuta el servidor de la aplicación usando Node.js y Express.

- 1) Importa los módulos esenciales: Al principio importa todos los modulos necesario para su funcionamiento, importa: express para manejar el servidor web, cors para permitir las solicitudes desde otros dominios, sequelize para la conexión a la base de datos.
- 2) Configuración middleware:
  - El cors() habilitara el intercambio de recursos entre diferentes orígenes
  - El express.json() permite recibir y procesar datos en formato JSON.
  - express.static('Public/Imagenes') sirve archivos estáticos, es decir imágenes.
- 3) Definir rutas: Para definirlas usamos /productos que gestiona los productos y /auth que maneja el registro e inicio de sesión de los usuarios.
- 4) Conectar la base de datos: Para esto utilizamos sequelize.authenticate() que verifica la conexión con la base de datos y sequilize.sync({ alter: true}) que sincroniza los modelos sin eliminar datos existentes.
- 5) Iniciar el servidor: Se muestra un mensaje en la consola cuando este listo (<http://localhost:3000>).