Universidad Mariano Gálvez

Centro Universitario El Naranjo

Ingeniería en Sistemas

Desarrollo Web

Ciclo Vlll

Octavo Semestre

Manual Técnico

Kevin Enrique Lopez Morataya - 9490-20327

Israel Alexander silva Morales – 949020156

Introducción

Descripción general del proyecto: El proyecto consiste en el desarrollo de una plataforma de comercio electrónico mediante el uso de APIs REST en Node.js. Se aplicarán conceptos de autenticación, seguridad, y gestión de datos para crear un sistema completo de compras en línea.

Objetivos de aprendizaje:

- Comprender y aplicar conceptos de APIs REST, autenticación y seguridad en aplicaciones web.
- Utilizar Node.js y Express para crear una base sólida de servidor y rutas para las APIs.
- Implementar la autenticación de usuarios utilizando JWT para garantizar la seguridad de las peticiones.
- Diseñar y desarrollar módulos de registro de usuario, login y gestión de perfiles con actualizaciones y eliminaciones.
- Exponer una API que permita acceder al catálogo de productos disponibles y realizar búsquedas por nombre y categoría.

Contenido

1. **Arquitectura del Sistema**

Esquema Conceptual: El esquema conceptual representa una vista general de cómo se estructura el proyecto. Incluye componentes como usuarios, productos, carritos de compra, ventas, y administración.

Esquema Lógico: El esquema lógico se adentra en los detalles de cómo funcionan los componentes del sistema. Describe los flujos de datos y procesos clave que ocurren en el proyecto, incluyendo la autenticación, la gestión de productos y las compras.

Esquema Físico: El esquema físico describe la infraestructura utilizada en el proyecto, como servidores, bases de datos y servicios externos. Detalla cómo se conectan estos elementos para que la aplicación funcione.

2. **Base de Datos MongoDB**

La base de datos MongoDB se utiliza para almacenar datos críticos del proyecto, incluyendo usuarios, productos y ventas. Las colecciones principales incluyen usuarios, productos, carritos de compra, ventas y bitácoras de compras. Cada colección tiene campos específicos que se describen en detalle en la documentación.

3. **API REST en Node.js**

La API se construye en Node.js utilizando Express.js como framework. Las rutas de la API se definen en archivos separados para una mejor organización. Se utilizan controladores para manejar las solicitudes HTTP y las respuestas correspondientes.

4. Autenticación mediante JWT

La autenticación de usuarios se logra mediante el uso de JSON Web Tokens (JWT).

Cada solicitud a la API debe incluir un token válido en el encabezado para autenticarse.

Se detalla cómo se genera y valida un token JWT en el código.

5. Módulos del Proyecto

Módulo de Registro de Usuarios: Este módulo permite a los usuarios registrarse en la plataforma proporcionando información personal.Se realizan diversas validaciones, como la verificación de la dirección de correo electrónico y la fuerza de la contraseña.

Módulo de Login: Permite a los usuarios autenticarse en la plataforma proporcionando sus credenciales. Si la autenticación es exitosa, se genera un token JWT para su uso en solicitudes posteriores.

Módulo de Gestión de Perfil: Permite a los usuarios ver y actualizar su perfil de usuario, así como eliminar su cuenta.

Módulo Catálogo de Productos: Proporciona acceso al catálogo de productos disponibles en la plataforma. Los usuarios pueden buscar productos por nombre y categoría.

Módulo de Gestión de Producto: Permite a los administradores realizar operaciones CRUD en productos. Se implementa una lógica para calcular el precio con descuento y gestionar las categorías de productos.

Módulo de Carrito de Compra: Los usuarios pueden agregar productos al carrito de compra y gestionar las cantidades. El carrito de compra se asocia con un usuario y se valida la disponibilidad de productos.

Módulo de Compra: Permite a los usuarios realizar compras, descontando productos del inventario y registrando la venta en la bitácora.

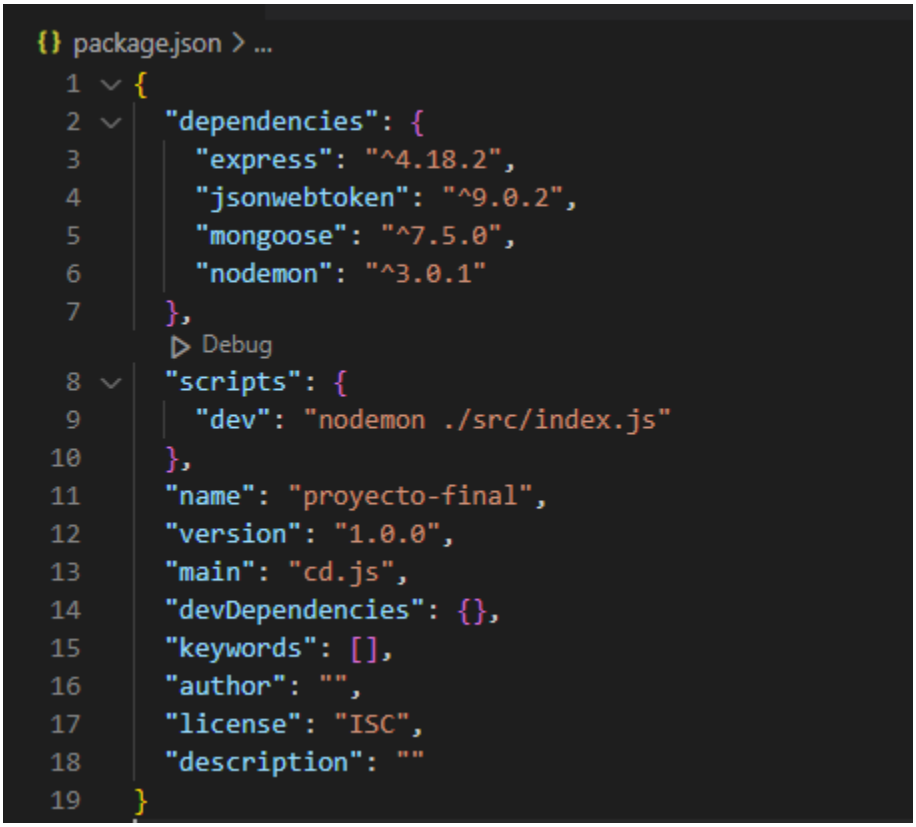
6. **Seguridad**

Se aplican medidas de seguridad, como la validación de tokens, para proteger la API. Se definen roles de usuario para limitar el acceso a ciertas funcionalidades.

7. **Comunicación con la Base de Datos**

La API se comunica con la base de datos MongoDB para almacenar y recuperar datos. Se explican las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) que se realizan en la base de datos.

Package.json



dependencies: Esta sección enumera las dependencias del proyecto, es decir, las librerías y módulos de Node.js que son necesarios para que tu aplicación funcione correctamente. Las dependencias especificadas en esta sección se instalarán automáticamente cuando ejecutes el comando npm install en la raíz del proyecto.

scripts:En esta sección, defines scripts personalizados que pueden ser ejecutados mediante el comando npm run <script-name>. En tu caso, tienes un script llamado "dev" que utiliza nodemon para ejecutar el archivo ./src/index.js. Esto es útil durante el desarrollo, ya que nodemon reiniciará automáticamente la aplicación cuando detecte cambios en el código, lo que facilita la depuración.

name:El nombre del proyecto. En este caso, el proyecto se llama "proyecto-final".

main: Este campo se especifica el archivo principal del proyecto. En el package.json, se establece como "cd.js".

devDependencies:Esta sección se utiliza para enumerar las dependencias que son necesarias solo durante el desarrollo..

verificarToken.js

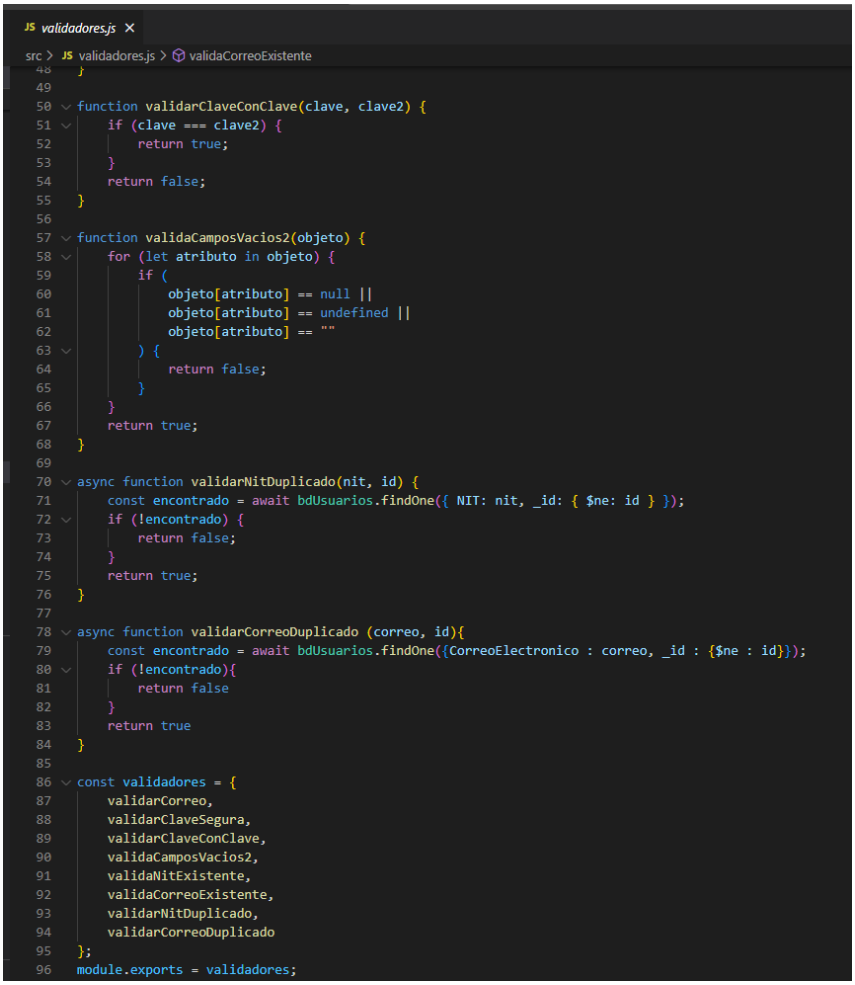
```
JS verificaToken.js > ...
const jwt = require('jsonwebtoken');

function verificaToken(req, res, next){
    const token = req.headers["x-access-token"];
    let tokenDesifrado
    try{
        tokenDesifrado = jwt.verify(token, 'textosupersecreto');
    }catch{
        return res.json({Error: 'Token no valido'})
    }
    req.tokenD = tokenDesifrado.id;
    next();
}

module.exports = verificaToken;
```

- Extrae el token del encabezado "x-access-token" de la solicitud.
- Intenta verificar y decodificar el token utilizando una clave secreta ("textosupersecreto" en este caso) utilizando la función jwt.verify(). Si la verificación es exitosa, el token decodificado se almacena en la variable tokenDesifrado.
- Si la verificación falla, se captura un error y se responde con un mensaje de error JSON indicando que el token no es válido.

- Si la verificación es exitosa, el ID (o cualquier otro dato relevante) contenido en el token se almacena en req.tokenD para que los controladores o middleware posteriores puedan acceder a esta información.
- Finalmente, se llama a next() para permitir que la solicitud continúe su flujo y se maneje por el controlador o middleware siguiente.

Validadores.js

```
src > validadores.js > validaCorreoExistente
1   const bdUsuarios = require("./models/usuario");
2
3   async function validaNitExistente(NIT) {
4       const data = await bdUsuarios.findOne({ NIT });
5       if (!data) {
6           return false;
7       }
8       return true;
9   }
10
11  async function validaCorreoExistente(CorreoElectronico) {
12      const data = await bdUsuarios.findOne({ CorreoElectronico });
13      if (!data) {
14          return false;
15      }
16      return true;
17  }
18
19  async function validaCorreoExistente(CorreoElectronico) {
20      const usuarioExiste = await bdUsuarios.findOne({
21          CorreoElectronico: CorreoElectronico,
22      });
23      if (!usuarioExiste) {
24          return false;
25      }
26      return true;
27  }
28
29  function validarCorreo(correo) {
30      let regex = /^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$/;
31      return regex.test(correo);
32  }
33
34  function validarClaveSegura(clave) {
35      let tieneLongitudValida = clave.length >= 8;
36      let tieneMayusculas = /[A-Z]/.test(clave);
37      let tieneMinusculas = /[a-z]/.test(clave);
38      let tieneNumeros = /\d/.test(clave);
39      let tieneCaracteresEspeciales = /[\W_]/.test(clave);
40
41      return (
42          tieneLongitudValida &&
43          tieneMayusculas &&
44          tieneMinusculas &&
45          tieneNumeros &&
46          tieneCaracteresEspeciales
47      );
48  }
49
```



Contiene un conjunto de funciones que se utilizan para validar diferentes aspectos en una aplicación. Estas funciones se pueden utilizar para verificar la existencia de NIT o correos electrónicos en la base de datos, validar la estructura de correos electrónicos, verificar la seguridad de contraseñas, verificar si los campos de un objeto están vacíos y más

- validaNitExistente(NIT):Esta función verifica si un número de identificación tributaria (NIT) ya existe en la base de datos de usuarios. Devuelve true si el NIT existe y false si no existe.

- validaCorreoExistente(CorreoElectronico):Esta función verifica si una dirección de correo electrónico ya existe en la base de datos de usuarios. Devuelve true si el correo electrónico existe y false si no existe.
- validarCorreo(correo):Esta función utiliza una expresión regular para verificar si una cadena dada es una dirección de correo electrónico válida. Devuelve true si la cadena es una dirección de correo electrónico válida y false si no lo es.
- validarClaveSegura(clave):Esta función verifica si una contraseña cumple con ciertos criterios de seguridad, como longitud mínima, uso de mayúsculas, minúsculas, números y caracteres especiales. Devuelve true si la contraseña es segura y false si no lo es.
- validarClaveConClave(clave, clave2):Esta función compara dos contraseñas para verificar si son iguales. Devuelve true si las contraseñas son iguales y false si son diferentes.
- validaCamposVacios2(objeto):Esta función verifica si alguno de los valores de un objeto (atributos) es nulo, indefinido o una cadena vacía. Devuelve true si no hay campos vacíos y false si al menos uno de los campos está vacío.
- validarNitDuplicado(nit, id):Esta función verifica si un número de identificación tributaria (NIT) ya existe en la base de datos, excluyendo un usuario específico por su ID. Devuelve true si el NIT está duplicado y false si no lo está.
- validarCorreoDuplicado(correo, id):Esta función verifica si una dirección de correo electrónico ya existe en la base de datos, excluyendo un usuario específico por su ID. Devuelve true si el correo electrónico está duplicado y false si no lo está.

Index.js

```
src > JS index.js > ...
1   const express = require ('express')
2   const rutasUsuario = require('./controllers/rutasUsuario')
3   const rutasProducto = require('./controllers/rutasProducto')
4   const rutasCarrito = require ('./controllers/rutasCarrito')
5   const rutasCompra = require('./controllers/rutasCompra')
6   const DB_UMG = require("./baseDeDatos");
7
8   const app = express();
9
10  app.use(express.json());
11
12  app.use(rutasUsuario);
13  app.use(rutasProducto);
14  app.use(rutasCarrito);
15  app.use(rutasCompra);
16
17  app.listen(3000, ()=>{
18      console.log('Servidor iniciado en puerto 3000');
19  })
```

Es el punto de entrada principal de tu aplicación Node.js. Aquí se configuran las rutas y se inicia el servidor Express.

- Se importan los módulos necesarios para la aplicación, incluyendo express y varios archivos de rutas, como rutasUsuario, rutasProducto, rutasCarrito, rutasCompra, y DB_UMG (que parece ser el módulo de base de datos).
- Creación de una instancia de Express:Se crea una instancia de la aplicación Express llamada app.
- Uso de middleware express.json(): Se utiliza el middleware express.json() para permitir que la aplicación pueda analizar el cuerpo de las solicitudes HTTP que contienen datos en formato JSON. Esto es útil para procesar las solicitudes POST y PUT que envían datos en formato JSON.

- Asignación de rutas a la aplicación: Se utilizan los métodos app.use() para asignar las rutas a la aplicación. Cada app.use() corresponde a un conjunto de rutas manejadas por un controlador específico. Las rutas están definidas en los archivos rutasUsuario, rutasProducto, rutasCarrito, y rutasCompra.
- Inicio del servidor:Se inicia el servidor Express llamando al método listen() en el puerto 3000.

Basededatos.js

```
src > JS baseDeDatos.js > ...
1   const mongoose = require("mongoose");
2
3   mongoose
4     .connect(
5       "mongodb+srv://lk3v1nn:dE658bz9Pr88GPgS@cluster0.i5qgmef.mongodb.net/PryectoUMG"
6     )
7     .catch((err) => {
8       console.log(err);
9     });
10
11  const connection = mongoose.connection;
12
13  //Evento de cuando se conecta a la BD
14  connection.on("open", () => {
15    console.log("Conectado a la Base de datos");
16  });
17
18  connection.on("err", (err) => {
19    console.log("Error DB: ", err);
20  });
21
```

contiene la configuración y conexión a la base de datos MongoDB utilizando la biblioteca mongoose

- Se importa la biblioteca mongoose, que es una biblioteca de Node.js que se utiliza para interactuar con bases de datos MongoDB de manera sencilla y eficiente.
- Conexión a la base de datos: Se utiliza el método mongoose.connect() para establecer la conexión a la base de datos MongoDB. La URL de conexión a la base de datos se proporciona como argumento de este método.

Carpeta Models

Usuarios.js

src > models > JS usuario.js > ...

```
1   const { Schema, model } = require("mongoose");
2
3   const usuarioSchema = new Schema ({
4       'Nombres' : String,
5       'Apellidos' : String,
6       'FechaNacimiento' : String,
7       'Clave': String,
8       'ValidacionClave': String,
9       'DireccionEntrega': String,
10      'NIT': Number,
11      'NumeroTelefonico': Number,
12      'CorreoElectronico': String
13  });
14
15  const usuario = model('usuarios', usuarioSchema);
16
17  module.exports = usuario;
```

- Definición del esquema del usuario: Se define el esquema del usuario utilizando new Schema({ ... }). Cada campo del esquema representa un atributo de un usuario, como "Nombres," "Apellidos," "FechaNacimiento," "Clave," "ValidacionClave," "DireccionEntrega," "NIT," "NumeroTelefonico," y "CorreoElectronico."
- Creación del modelo de usuario: Se utiliza model('usuarios', usuarioSchema) para crear un modelo de datos llamado "usuarios" utilizando el esquema definido anteriormente. El nombre "usuarios" se utiliza para referirse a la colección de MongoDB donde se almacenarán los documentos de usuarios.

- Exportación del modelo:Se exporta el modelo de usuario creado para que pueda ser utilizado en otras partes de la aplicación.

Producto.js

```
src > models > JS producto.js > ...
1   const {Schema, model} = require ('mongoose');
2
3   const productoSchema = new Schema ({
4       Identificador : String,
5       Nombre : String,
6       Marca : String,
7       Disponibilidad : Number,
8       Descuento : Number,
9       PrecioDescuento : Number,
10      Imagen : String,
11      Descripcion : String,
12      Categorias : Array
13  });
14
15  const producto = model('productos', productoSchema);
16
17  module.exports = producto;
```

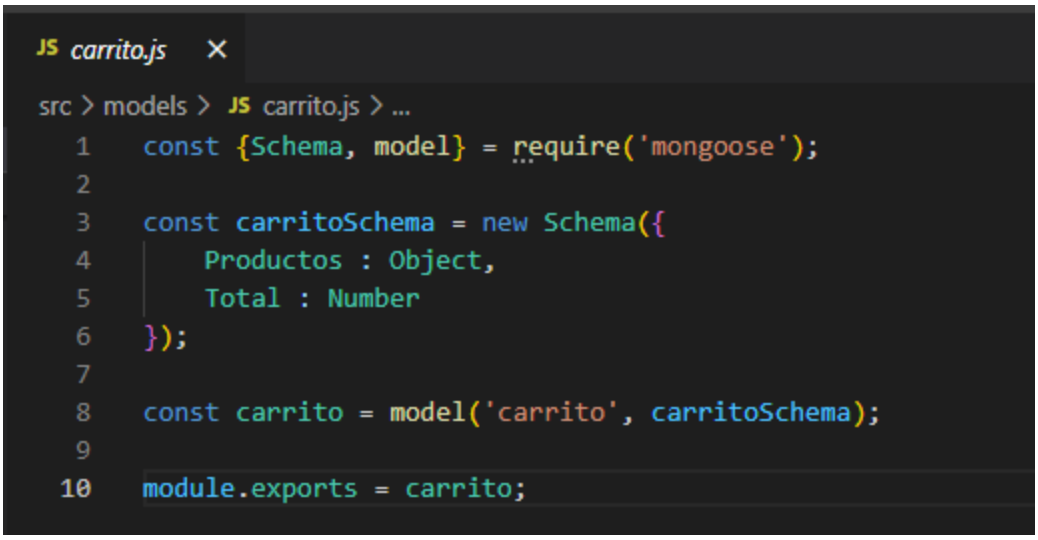
- Definición del esquema del producto: Se define el esquema del producto utilizando new Schema({ ... }). Cada campo del esquema representa un atributo de un producto, como "Identificador," "Nombre," "Marca," "Disponibilidad," "Descuento," "PrecioDescuento," "Imagen," "Descripcion," y "Categorias." Los tipos de datos de estos campos se definen como String, Number, o Array, según corresponda.
- Creación del modelo de producto: Se utiliza model('productos', productoSchema) para crear un modelo de datos llamado "productos" utilizando el esquema definido anteriormente. El nombre "productos" se utiliza para referirse a la colección de MongoDB donde se almacenarán los documentos de productos.
- Exportación del modelo: Se exporta el modelo de producto creado para que pueda ser utilizado en otras partes de la aplicación.

Compra.js

- Definición del esquema de compra: Se define el esquema de compra utilizando new Schema({ ... }). El esquema tiene dos campos: "id_user" y "Productos". "id_user" es una cadena que almacena el identificador del usuario que realizó la compra, y "Productos" es un objeto que almacena información sobre los productos comprados.

- Creación del modelo de compra:Se utiliza model('compras', compraSchema) para crear un modelo de datos llamado "compras" utilizando el esquema definido anteriormente. El nombre "compras" se utiliza para referirse a la colección de MongoDB donde se almacenarán los documentos de compras.
- Exportación del modelo: Se exporta el modelo de compra creado para que pueda ser utilizado en otras partes de la aplicación.

Carrito.js

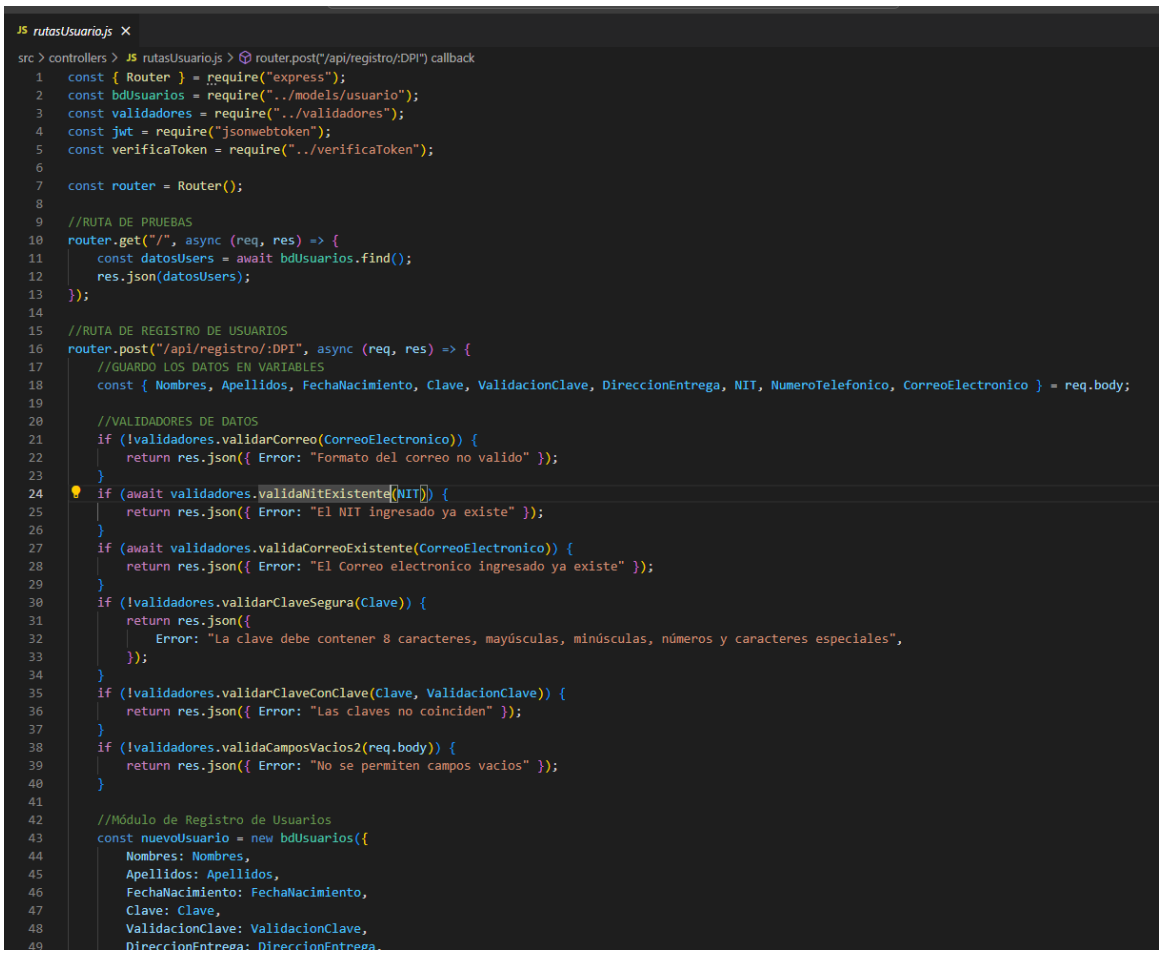


- Definición del esquema del carrito de compra: Se define el esquema del carrito de compra utilizando new Schema({ ... }). El esquema tiene dos campos: "Productos" y "Total". "Productos" es un objeto que almacena información sobre los productos en el carrito de compra, y "Total" es un número que representa el total a pagar por los productos en el carrito.

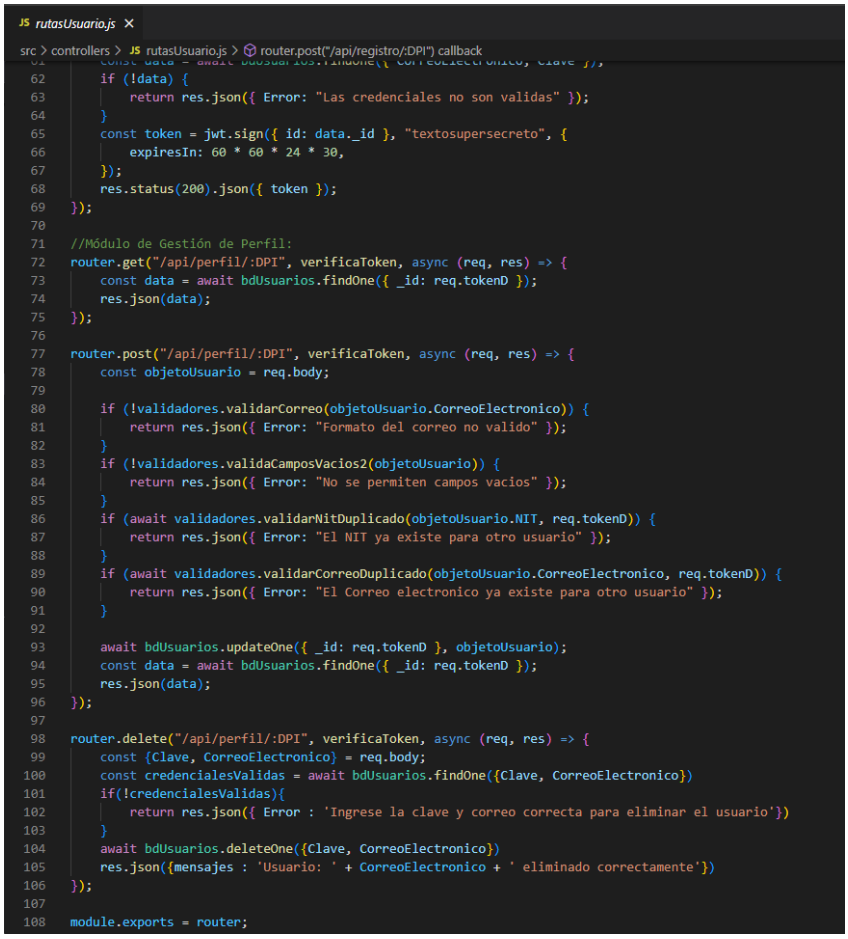
- Creación del modelo de carrito de compra: Se utiliza model('carrito', carritoSchema) para crear un modelo de datos llamado "carrito" utilizando el esquema definido anteriormente. El nombre "carrito" se utiliza para referirse a la colección de MongoDB donde se almacenarán los documentos de carritos de compra.
- Exportación del modelo: Se exporta el modelo de carrito de compra creado para que pueda ser utilizado en otras partes de la aplicación.

Carpeta controllers

Rutasusuario.js



```
JS rutasUsuario.js ×
src > controllers > JS rutasUsuario.js > router.post("/api/registro/:DPI") callback
1   const { Router } = require("express");
2   const bdUsuarios = require("../models/usuario");
3   const validadores = require("../validadores");
4   const jwt = require("jsonwebtoken");
5   const verificaToken = require("../verificaToken");
6
7   const router = Router();
8
9   //RUTA DE PRUEBAS
10  router.get("/", async (req, res) => {
11      const datosUsers = await bdUsuarios.find();
12      res.json(datosUsers);
13  });
14
15  //RUTA DE REGISTRO DE USUARIOS
16  router.post("/api/registro/:DPI", async (req, res) => {
17      //GUARDO LOS DATOS EN VARIABLES
18      const { Nombres, Apellidos, FechaNacimiento, Clave, ValidacionClave, DireccionEntrega, NIT, NumeroTelefonico, CorreoElectronico } = req.body;
19
20      //VALIDADORES DE DATOS
21      if (!validadores.validarCorreo(CorreoElectronico)) {
22          return res.json({ Error: "Formato del correo no valido" });
23      }
24      if (await validadores.validaNitExistente(NIT)) {
25          return res.json({ Error: "El NIT ingresado ya existe" });
26      }
27      if (await validadores.validaCorreoExistente(CorreoElectronico)) {
28          return res.json({ Error: "El Correo electronico ingresado ya existe" });
29      }
30      if (!validadores.validarClaveSegura(Clave)) {
31          return res.json({
32              Error: "La clave debe contener 8 caracteres, mayúsculas, minúsculas, números y caracteres especiales",
33          });
34      }
35      if (!validadores.validarClaveConClave(Clave, ValidacionClave)) {
36          return res.json({ Error: "Las claves no coinciden" });
37      }
38      if (!validadores.validaCamposVacios2(req.body)) {
39          return res.json({ Error: "No se permiten campos vacios" });
40      }
41
42      //Módulo de Registro de Usuarios
43      const nuevoUsuario = new bdUsuarios({
44          Nombres: Nombres,
45          Apellidos: Apellidos,
46          FechaNacimiento: FechaNacimiento,
47          Clave: Clave,
48          ValidacionClave: ValidacionClave,
49          DireccionEntrega: DireccionEntrega,
```

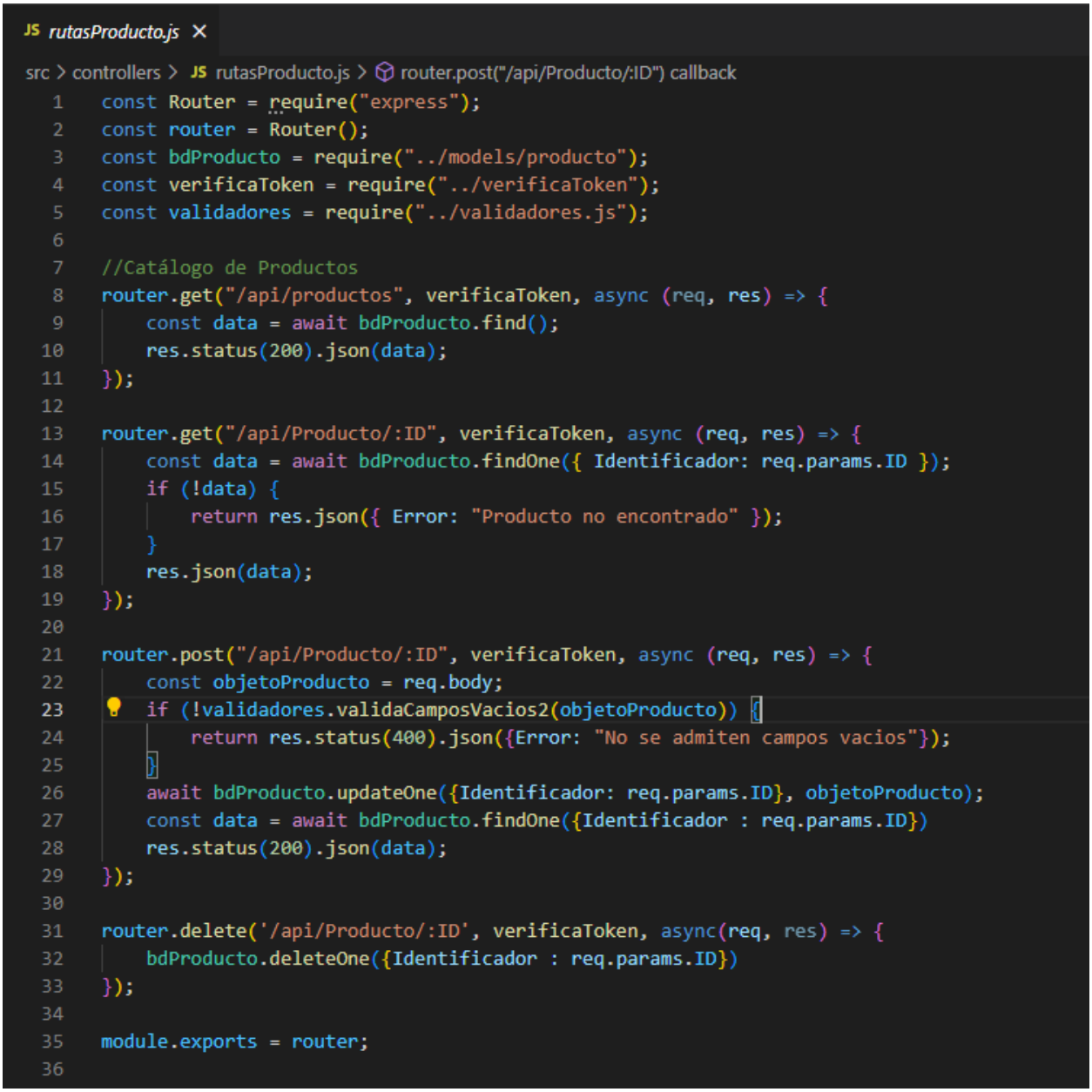


Contiene las rutas y controladores relacionados con la gestión de usuarios en tu aplicación.

- Importación de módulos y dependencias:Se importan los módulos y dependencias necesarios, como el objeto Router de Express, el modelo de usuario bdUsuarios, validadores personalizados (validadores), la biblioteca jsonwebtoken para manejar tokens JWT y el middleware verificaToken para la autenticación de usuarios.
- Ruta de pruebas:Esta ruta es para propósitos de prueba y devuelve todos los datos de usuarios almacenados en la base de datos. Se utiliza el método GET para esta ruta.

- Ruta de registro de usuarios:Esta ruta permite a los usuarios registrarse en la aplicación. Se utiliza el método POST y se validan los datos de registro, como el formato del correo electrónico, la existencia del NIT y la duplicación del correo electrónico. Si los datos son válidos, se crea un nuevo usuario y se almacena en la base de datos.
- Ruta de inicio de sesión (login):Los usuarios pueden iniciar sesión en la aplicación proporcionando su correo electrónico y contraseña. Se utiliza el método POST y se verifica si las credenciales son válidas. Si lo son, se genera un token JWT y se devuelve al cliente.
- Ruta de gestión de perfil:Esta ruta permite a los usuarios acceder y actualizar su perfil. Se utiliza el método GET para obtener la información del perfil y el método POST para actualizarla. Se validan los datos actualizados, como el formato del correo electrónico y la duplicación del NIT o correo electrónico. También se proporciona una ruta DELETE para eliminar el perfil del usuario.

RutasProducto.js



Contiene las rutas y controladores relacionados con la gestión de productos en tu aplicación.

- Importación de módulos y dependencias: Se importan los módulos y dependencias necesarios, como el objeto Router de Express, el modelo de producto bdProducto, el middleware verificaToken para la autenticación de usuarios y los validadores personalizados (validadores).
- Ruta de catálogo de productos:Esta ruta permite a los usuarios obtener una lista de todos los productos disponibles en la aplicación. Se utiliza el método GET y se verifica el token de autenticación.
- Ruta de obtención de un producto específico:Los usuarios pueden obtener información detallada sobre un producto específico utilizando esta ruta. Se utiliza el método GET y se proporciona el identificador del producto en la URL. Si el producto existe, se devuelve su información; de lo contrario, se envía un mensaje de error.
- Ruta de actualización de un producto:Los usuarios pueden actualizar la información de un producto utilizando esta ruta. Se utiliza el método POST y se proporciona el identificador del producto en la URL. Se validan los datos actualizados y se actualiza el producto en la base de datos.
- Ruta de eliminación de un producto:Los administradores pueden eliminar un producto de la aplicación utilizando esta ruta. Se utiliza el método DELETE y se proporciona el identificador del producto en la URL. El producto correspondiente se elimina de la base de datos.

Rutascompras.js

```
src > controllers > JS rutasCompra.js > ...
1   const Router = require("express");
2   const router = Router();
3   const bdCarrito = require("../models/carrito");
4   const bdProducto = require("../models/producto");
5   const bdCompra = require("../models/compra");
6
7   const verificaToken = require("../verificaToken");
8
9   router.post("/api/compra", verificaToken, async (req, res) => {
10      const dataCarrito = await bdCarrito.findOne({ id_user: req.tokenD });
11      if(!dataCarrito){return res.status(400).json({Error : "El usuario no tiene articulos en el carrito"})}
12
13      const cantidad = dataCarrito?.Productos.Cantidad;
14      const idProducto = dataCarrito?.Productos.Identificador;
15      const dataProducto = await bdProducto.findOne({ Identificador: idProducto });
16
17      dataProducto.Disponibilidad = dataProducto.Disponibilidad - cantidad;
18      await bdProducto.updateOne({ Identificador: idProducto }, dataProducto);
19
20      const nuevaCompra = new bdCompra({ id_user: req.tokenD, Productos: dataCarrito?.Productos });
21      nuevaCompra.save();
22
23      res.json({ "Mensaje": "Inventario actualizado", "Mensaje2": "Bitácora creada" });
24  });
25
26  module.exports = router;
27
```

Contiene las rutas y controladores relacionados con la gestión de compras en tu aplicación

- Importación de módulos y dependencias:Se importan los módulos y dependencias necesarios, como el objeto Router de Express, los modelos bdCarrito, bdProducto, y bdCompra, y el middleware verificaToken para la autenticación de usuarios.
- Ruta de realización de compra:Esta ruta permite a los usuarios realizar una compra de los productos que tienen en su carrito de compras. Se utiliza el método POST y se verifica el token de autenticación.
- Validación del carrito:Se busca el carrito de compras del usuario identificado por el token. Si el usuario no tiene artículos en el carrito, se devuelve un mensaje de error.

- Actualización del inventario:Se obtiene la cantidad y el identificador del producto en el carrito y se busca el producto correspondiente en la base de datos. Se actualiza la disponibilidad del producto restando la cantidad comprada.
- Registro de la compra:Se crea un registro de compra en la base de datos, que incluye el ID del usuario y los productos comprados. Este registro se almacena en la colección de compras (bdCompra).
- Respuesta al cliente:Se devuelve una respuesta JSON que indica que el inventario se ha actualizado correctamente y que se ha creado una bitácora de la compra.

Rutascarrito.js

Contiene las rutas y controladores relacionados con la gestión del carrito de compras en tu aplicación.

- Importación de módulos y dependencias:Se importan los módulos y dependencias necesarios, como el objeto Router de Express, el middleware verificaToken para la autenticación de usuarios y el modelo bdCarrito para el carrito de compras.
- Ruta de obtención del carrito de compras:Esta ruta permite a los usuarios obtener el contenido de su carrito de compras. Se utiliza el método GET y se verifica el token de autenticación.
- Ruta de actualización del carrito de compras:Esta ruta permite a los usuarios actualizar la cantidad de productos en su carrito de compras. Se utiliza el método POST y se verifica el token de autenticación.
- Validación de la cantidad:Se verifica que se haya proporcionado una cantidad válida en el cuerpo de la solicitud. Si no se proporciona, se devuelve un mensaje de error.
- Búsqueda y actualización del carrito:Se busca el carrito de compras del usuario identificado por el token. Si no se encuentra un carrito para ese usuario, se devuelve un mensaje de error. Luego, se actualiza la cantidad de productos en el carrito y se guarda.
- Respuesta al cliente:Se devuelve una respuesta JSON que contiene el contenido actualizado del carrito de compras después de la actualización.
- Ruta de eliminación del carrito de compras:Esta ruta permite a los usuarios eliminar todo el contenido de su carrito de compras. Se utiliza el método DELETE y se verifica el token de autenticación.
- Eliminación del carrito:Se busca y elimina el carrito de compras del usuario identificado por el token.
- Respuesta al cliente:Se devuelve una respuesta JSON que indica que el carrito de compras ha sido eliminado.