

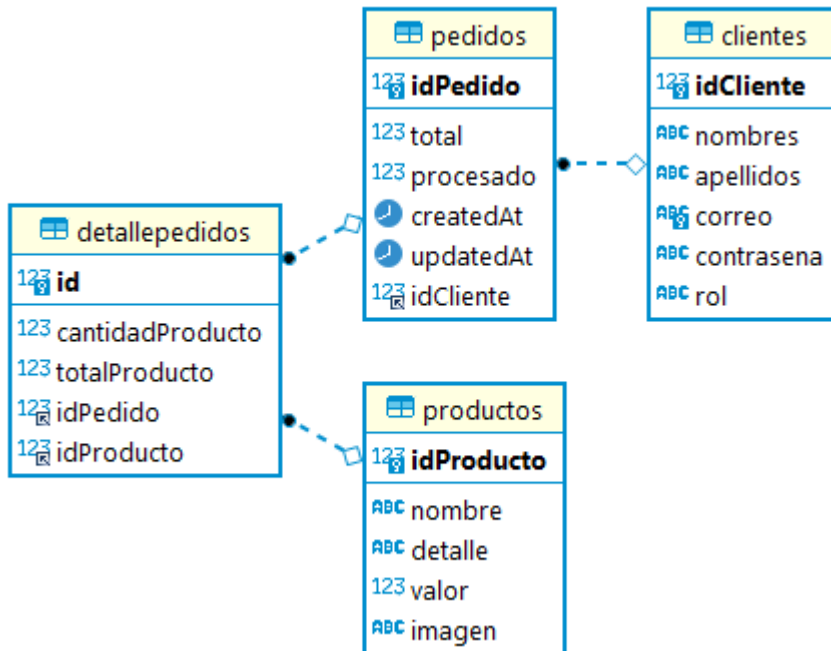
TALLER UNIDAD 2 BACKEND

BRAYAN DANIEL CERON PORTILLA

UNIVERSIDAD DE NARIÑO

## 1. Base de datos:

Para la representación del modelo de datos del negocios, se utilizó el siguiente esquema, generado automáticamente por la codificación de los modelos en Sequelize



El motor de base de datos que se utilizó fue MySQL, y la conexión se realizó utilizando el ORM sequelize. La conexión se realizó en un ambiente local hacia una base de datos previamente creada, la cual fue llamada "fruver"

```
import Sequelize from "sequelize";
import dotenv from 'dotenv';
dotenv.config()

const sequelize = new Sequelize(
  process.env.DB_DATABASE,
  process.env.DB_USER,
  process.env.DB_PASS, {
    host: process.env.DB_HOST,
    dialect: "mysql",
  });

export { sequelize }
```

Como se muestra en el esquema de la base de datos, se codificaron 4 modelos: clientes, productos, pedidos y detallePedidos

## Modelo Cliente

```
import { DataTypes } from "sequelize";
import { sequelize } from "../Database/database.js";

const Cliente = sequelize.define(
  "clientes",
  {
    // Definicion de Atributos
    idCliente: {
      type: DataTypes.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true
    },
    nombres: {
      type: DataTypes.STRING,
      allowNull: false
    },
    apellidos: {
      type: DataTypes.STRING,
      allowNull: false
    },
    correo: {
      type: DataTypes.STRING,
      allowNull: false,
      unique: true
    },
    contrasena: {
      type: DataTypes.STRING,
      allowNull: false
    },
    rol: {
      type: DataTypes.STRING,
      allowNull: false,
      defaultValue: "user"
    },
  },
  {
    timestamps: false,
  },
);
```

## Modelo Producto

```
import { DataTypes } from "sequelize";
import { sequelize } from "../Database/database.js";

const Producto = sequelize.define(
  "productos",
  {
    // Definicion de Atributos
    idProducto: {
      type: DataTypes.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true,
    },
    nombre: {
      type: DataTypes.STRING,
      allowNull: false
    },
    detalle: {
      type: DataTypes.STRING,
      allowNull: false
    },
    valor: {
      type: DataTypes.INTEGER,
      allowNull: false
    },
    imagen: {
      type: DataTypes.STRING,
      allowNull: false,
    }
  },
  {
    timestamps: false,
  },
);
```

## Modelo Pedido

```
import { DataTypes } from "sequelize";
import { sequelize } from "../Database/database.js";

const Pedido = sequelize.define(
  "pedidos",
  {
    // Definicion de Atributos
    idPedido: {
      type: DataTypes.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true
    },
    total: {
      type: DataTypes.INTEGER,
      allowNull: false
    },
    procesado: {
      type: DataTypes.BOOLEAN,
      allowNull: false,
      defaultValue: false
    }
  },
  {
    timestamps: true,
  }
);
```

## Modelo Detalle Pedido

```
import { DataTypes } from "sequelize";
import { sequelize } from "../Database/database.js";
import { Pedido } from "../pedidos.js";
import { Producto } from "../productos.js";
import { Cliente } from "../cliente.js";

const detallePedido = sequelize.define(
  "detallePedido",
  {
    // Definicion de Atributos
    cantidadProducto: {
      type: DataTypes.INTEGER,
      allowNull: false
    },
    totalProducto: {
      type: DataTypes.INTEGER,
      allowNull: false
    }
  },
  {
    timestamps: false,
  }
);
```

Las relaciones entre los modelos se establecieron de la siguiente forma:

```
//Relaciones
Cliente.hasMany(Pedido,{foreignKey:"idCliente",sourceKey:"idCliente",onDelete:'restrict'});
Pedido.belongsTo(Cliente,{foreignKey:"idCliente",foreign:"idCliente",onDelete:'restrict'});

Pedido.hasMany(detallePedido,{foreignKey:"idPedido",sourceKey:"idPedido",onDelete:'cascade'});
detallePedido.belongsTo(Pedido,{foreignKey:"idPedido", foreign:"idPedido",onDelete:'cascade'});

Producto.hasMany(detallePedido,{foreignKey:"idProducto",sourceKey:"idProducto", onDelete:'restrict'});
detallePedido.belongsTo(Producto,{foreignKey:"idProducto",foreign:"idProducto", onDelete:'restrict'});

export { detallePedido };
```

## 2. Rutas

Las rutas que expone la api, representan principalmente las operaciones CRUD que se puede realizar sobre los modelos, pero para este caso también se adicionaron unas rutas extra que permiten manejar todo el tema de autenticación y el envío de correos.

```
import { Router } from 'express';
import { postClientes, getClientes, getCliente, deleteClientes, putClientes, login, verificarTokenSession, getRol } from '../Controllers/cliente.controller.js';
import { getProducto, getProductos, postProductos, putProductos, deleteProductos } from '../Controllers/producto.controller.js';
import { getPedido, getPedidos, postPedidos, putPedidos, deletePedido, procesarPedido, getPedidosCliente } from '../Controllers/pedido.controller.js';
import { verificarToken } from '../Middleware/autenticacion.js';
import { upload } from '../Middleware/multer.js';
const router = Router();

// Rutas
//-----
router.get("/productos", verificarToken, getProductos);
router.get("/producto/:idProducto", verificarToken, getProducto);
router.post("/productos", verificarToken, upload.single('file'), postProductos);
router.put("/productos/:idProducto", verificarToken, upload.single('file'), putProductos);
router.delete("/productos/:idProducto", verificarToken, deleteProductos);
//-----
router.get("/pedidos", verificarToken, getPedidos);
router.get("/pedido/:idPedido", verificarToken, getPedido);
router.get("/pedidos/cliente/:idCliente", verificarToken, getPedidosCliente);
router.post("/pedidos", verificarToken, postPedidos);
router.put("/pedidos/procesar/:idPedido", verificarToken, procesarPedido);
router.put("/pedidos/:idPedido", verificarToken, putPedidos);
router.delete("/pedidos/:idPedido", verificarToken, deletePedido);
//-----
router.get("/clientes", verificarToken, getClientes);
router.get("/cliente/:idCliente", verificarToken, getCliente);
router.post("/clientes", postClientes);
router.post("/login", login);
router.get("/verificarToken", verificarTokenSession);
router.get("/getRol", verificarToken, getRol);
router.put("/clientes/:idCliente", verificarToken, putClientes);
router.delete("/clientes/:idCliente", verificarToken, deleteClientes);

export default router;
```

Se puede evidenciar, que antes de llamar al respectivo controlador, la mayoría de las rutas son precedidas por un middleware llamado “verificarToken”, este middleware es el encargado de gestionar todo lo relacionado a la autenticación. También es el caso del middleware “upload”, que se encarga de gestionar lo relacionado al tema de la carga de archivos.

### 3. Controladores

#### Controlador del modelo producto:

Para obtener tanto todos los productos como uno solo, se debe utilizar el verbo GET, estableciendo como parámetro el id del producto en caso de querer obtener un único producto y ningún parámetro en caso de querer obtener todos los productos

```
const getProducto = async (req, res) => {
  const { idProducto } = req.params;
  try {
    const producto = await Producto.findByPk(idProducto);
    res.status(200).json([producto]);
  } catch (error) {
    res.status(400).json({ mensaje: err });
  }
};

const getProductos = async (req, res) => {
  try {
    const productos = await Producto.findAll();
    res.status(200).json(productos);
  } catch (error) {
    res.status(400).json({ mensaje: error });
  }
};
```

Para poder registrar un producto en la base de datos se debe utilizar el verbo POST, estableciendo como parámetros en este caso el nombre del producto, una descripción del producto y su valor unitario

```
const postProductos = async (req, res) => {
  const { nombre, detalle, valor } = req.body;
  console.log(req.body);
  console.log(req.file);
  console.log(req.file.filename);

  try {
    const newProducto = await Producto.create({
      nombre,
      detalle,
      valor,
      imagen: req.file.filename
    });
    res.status(200).json(newProducto);
  } catch (error) {
    res.status(400).json({ mensaje: err });
  }
};
```

Al igual que para registrar un producto, para actualizar un producto es necesario recibir todos los atributos del modelo (con la nueva información) más el id del producto que se va actualizar. Para este caso se debe utilizar el verbo PUT

```
const putProductos = async (req, res) => {
  const { idProducto } = req.params;
  const { nombre, detalle, valor } = req.body;
  try {
    const oldProducto = await Producto.findByPk(idProducto);
    oldProducto.nombre = nombre;
    oldProducto.detalle = detalle;
    oldProducto.valor = valor;
    oldProducto.imagen = req.file.filename;
    const modProducto = await oldProducto.save();
    res.status(200).json(modProducto);
  } catch (error) {
    res.status(400).json({ mensaje: error });
  }
};
```

Para poder eliminar un producto se debe utilizar el verbo DELETE, estableciendo como parámetro el id del producto que se va a eliminar.

```
const deleteProductos = async (req, res) => {
  const { idProducto } = req.params;
  try {
    const respuesta = await Producto.destroy({
      where: {
        idProducto,
      },
    });
    res.status(200).json({ mensaje: "Registro Eliminado" });
  } catch (error) {
    if (error.name == "SequelizeForeignKeyConstraintError") {
      res.status(500).json({ mensaje: "Este producto no se puede eliminar porque aparece registrado en pedido(s)" });
      return
    }
    res.status(400).json({ mensaje: "Registro No Eliminado" + error });
  }
};
```

## Controlador del modelo pedido y detalle pedido:

Para obtener tanto todos los pedidos como uno solo, se debe utilizar el verbo GET, estableciendo como parámetro el id del pedido en caso de querer obtener un único pedido y ningún parámetro en caso de querer obtener todos los pedidos. Para obtener la información completa, este controlador también utiliza el modelo detalle pedido, ya que es en este modelo donde se guarda parte de esta información

```
const getPedido = async (req, res) => {
  const { idPedido } = req.params;

  try {
    const pedidos = await Pedido.findByPk(idPedido, {
      include: [{
        model: detallePedido,
        include: { model: Producto }
      }]
    });
    res.status(200).json(pedidos);
  } catch (error) {
    res.status(400).json({ mensaje: error.message });
  }
};

const getPedidos = async (req, res) => {
  try {
    const pedidos = await Pedido.findAll({
      include: [{
        model: detallePedido,
        include: { model: Producto, /*include: { model: Cliente */ }
      },
      {
        model: Cliente
      }
    ],
    order: [["procesado", "ASC"], ["createdAt", "DESC"]]
  });
  res.status(200).json(pedidos);
} catch (error) {
  res.status(400).json({ mensaje: error.message });
}
};
```

Dado que es muy recurrente consultar por todos los pedidos de un cliente, se implementó una ruta para obtener dichos productos, en esta ruta, lo único que se debe establecer es el id del cliente

```
const getPedidosCliente = async (req, res) => {
  const { idCliente } = req.params;

  try {
    const pedidos = await Pedido.findAll({
      where: {
        idCliente: idCliente
      },
      include: [{
        model: detallePedido,
        include: { model: Producto }
      },
      {
        model: Cliente
      }
    ],
    order: [["procesado", "ASC"], ["createdAt", "DESC"]]
  });
  res.status(200).json(pedidos);
} catch (error) {
  res.status(400).json({ mensaje: error.message });
}
};
```



Para poder registrar un pedido en la base de datos se debe utilizar el verbo POST, estableciendo como parámetros en este caso el id del cliente que ordena el pedido y un vector de productos, en el cual, cada elemento de este vector debe especificar el id del producto y la cantidad que se solicita de este producto. Los cálculos del total por producto y total del pedido se calculan directamente en el controlador, por lo que no es necesario enviar más información. La información generada en este controlador también se almacena en el modelo detalle pedido, por lo que no fue necesario implementar un controlador específico para ese modelo

```
const postPedidos = async (req, res) => {
  const { total, idCliente, productos } = req.body;

  try {
    let totalCompra = 0;

    const newPedido = await Pedido.create({
      total: 0,
      idCliente: idCliente,
    });
    let idPedido = newPedido.idPedido;

    let itemsProcesado = 0;
    productos.forEach(async producto => {
      const productoEncontrado = await Producto.findByPk(producto.idProducto);
      let cantidadProducto = producto.cantidadProducto;
      let totalProducto = productoEncontrado.valor * cantidadProducto;
      totalCompra = totalCompra + totalProducto;

      const newProducto = await detallePedido.create({
        idProducto: producto.idProducto,
        idPedido: idPedido,
        cantidadProducto: cantidadProducto,
        totalProducto: totalProducto
      });
      itemsProcesado++;
      if(itemsProcesado === productos.length) {callback();}
    });

    async function callback () {
      const newPedido2 = await Pedido.findByPk(idPedido);
      newPedido2.total = totalCompra;
      const modPedido = await newPedido2.save();
      res.status(200).json(modPedido);
    }
  } catch (error) {
    res.status(400).json({ err: error, mensaje: error.message });
  }
};
```

Algo similar que para registrar un pedido, para la actualización de un pedido se debe establecer como parámetros en este caso el id del pedido a actualizar y un vector de productos, en el cual, cada elemento de este vector debe especificar el id del producto y la cantidad que se solicita de este producto. De igual forma se vuelven a realizar los cálculos y se actualiza el modelo detalle pedido con los nuevos cambios. Para este caso se debe utilizar el verbo PUT

```

const putPedidos = async (req, res) => {
  const { idPedido } = req.params;
  const { productos, procesado } = req.body;

  try {
    const newPedido = await Pedido.findByPk(idPedido);
    if(newPedido.procesado==true){return res.status(409).json({ mensaje: "El pedido no puede ser modificado porque ya fue procesado" }});}

    let totalCompra = 0;
    await detallePedido.destroy({
      where: {idPedido},
    });

    let itemsProcesado = 0;
    productos.forEach(async (producto, index) => {
      const productoEncontrado = await Producto.findByPk(producto.idProducto);
      let cantidadProducto = producto.cantidadProducto;
      let totalProducto = productoEncontrado.valor * cantidadProducto;
      totalCompra = totalCompra + totalProducto;

      const newProducto = await detallePedido.create({
        idProducto: producto.idProducto,
        idPedido: idPedido,
        cantidadProducto: cantidadProducto,
        totalProducto: totalProducto
      });
      itemsProcesado++;
      if(itemsProcesado === productos.length) {callback();}
    });

    async function callback () {
      newPedido.total= totalCompra;
      newPedido.procesado=procesado;
      const modPedido = await newPedido.save();
      res.status(200).json(modPedido);
    }
  } catch (error) {
    res.status(400).json({ err: error, mensaje: error.message });
  }
};

```

Dado que también es muy recurrente cambiar el estado de un producto de pendiente a procesado, se implementó una ruta para realizar esta acción, en esta ruta, lo único que se debe establecer es el id del pedido. Para este caso se también debe utilizar el verbo PUT

```

const procesarPedido = async (req, res) => {
  const { idPedido } = req.params;

  try {
    const pedido = await Pedido.findByPk(idPedido, {
      include: [{
        model: detallePedido,
        include: { model: Producto, }
      },
      {
        model: Cliente
      }
    ]
  });
  if(pedido.procesado==true){return res.status(409).json({ mensaje: "El pedido no puede ser modificado porque ya fue procesado" }});}
  pedido.procesado = true;
  const modPedido = await pedido.save();
  //console.log("Si");
  let correoCliente = pedido.cliente.correo;
  let asunto = "Pedido Fruver";

  enviarCorreoElectronico(correoCliente, asunto, generarReporte(pedido));

  res.status(200).json(modPedido);
} catch (error) {
  res.status(400).json({ mensaje: error.message });
}
};

```

Para poder eliminar/cancelar un pedido se debe utilizar el verbo DELETE, estableciendo como parámetro el id del pedido que se va a eliminar.

```
const deletePedido = async (req, res) => {
  const { idPedido } = req.params;
  try {
    const respuesta = await Pedido.destroy({
      where: {
        idPedido,
      },
    });
    res.status(200).json({ mensaje: "Registro Eliminado" });
  } catch (error) {
    res.status(400).json({ mensaje: "Registro No Eliminado" + error });
  }
};

const procesarPedido = async (req, res) => { ...
};
```

## Controlador del modelo cliente:

Para obtener tanto todos los clientes como uno solo, se debe utilizar el verbo GET, estableciendo como parámetro el id del cliente en caso de querer obtener un único cliente y ningún parámetro en caso de querer obtener todos los clientes

```
const getCliente = async (req, res) => {
  const { idCliente } = req.params;
  try {
    const cliente = await Cliente.findByPk(idCliente);
    res.status(200).json([cliente]);
  } catch (error) {
    res.status(400).json({ mensaje: error });
  }
};

const getClientes = async (req, res) => {
  try {
    const clientes = await Cliente.findAll();
    res.status(200).json(clientes);
  } catch (error) {
    res.status(400).json({ mensaje: error.message });
  }
};
```

Para poder registrar un cliente en la base de datos se debe utilizar el verbo POST, estableciendo como parámetros en este caso los nombres y apellidos del cliente, su correo electrónico y una contraseña de acceso

```
const postClientes = async (req, res) => {
  req.body.contrasena = bcrypt.hashSync(req.body.contrasena, 8);
  const { nombres, apellidos, correo, contrasena } = req.body;
  try {
    const newCliente = await Cliente.create({
      nombres,
      apellidos,
      correo,
      contrasena
    });
    res.status(200).json(newCliente);
  } catch (error) {
    if (error.name == "SequelizeUniqueConstraintError") {
      res.status(400).json({ mensaje: "Ya existe una cuenta registrada con esos datos" });
      return;
    }
    res.status(400).json({ mensaje: error });
  }
};
```

Para actualizar un cliente, solamente es necesario establecer los nombres, apellidos( ya que solo es posible actualizar esos datos) y el id del cliente que se va actualizar. Para este caso se debe utilizar el verbo PUT

```
const putClientes = async (req, res) => {
  const { idCliente } = req.params;
  const { nombres, apellidos } = req.body;
  try {
    const oldCliente = await Cliente.findByPk(idCliente);
    oldCliente.nombres = nombres;
    oldCliente.apellidos = apellidos;
    const modCliente = await oldCliente.save();
    res.status(200).json(modCliente);
  } catch (error) {
    res.status(400).json({ mensaje: error });
  }
};
```

Para poder eliminar un cliente se debe utilizar el verbo DELETE, estableciendo como parámetro el id del cliente que se va a eliminar.

```
const deleteClientes = async (req, res) => {
  const { idCliente } = req.params;
  try {
    const respuesta = await Cliente.destroy({
      where: {
        idCliente,
      },
    });
    res.status(200).json({ mensaje: "Registro Eliminado" });
  } catch (error) {
    if (error.name == "SequelizeForeignKeyConstraintError") {
      res.status(500).json({ mensaje: "Este usuario no se puede eliminar porque tiene pedidos registrados" });
      return;
    }
    res.status(400).json({ mensaje: "Registro No Eliminado" + error });
  }
};
```

Dado que para los clientes también es necesario programar lo relacionado a la autenticación, se añadieron rutas extras que permitan gestionar dichas funcionalidades.

La función login, permite iniciar sesión, validar si el usuario existe y validar si las credenciales son correctas. Esta ruta funciona mediante el verbo POST

```
const login = async (req, res) => {
  try {
    const { correo, contrasena } = req.body;

    const user = await Cliente.findOne({
      where: { correo: correo }
    });

    if (!user) {
      return res.status(404).json({ msg: "Usuario no encontrado" });
    }

    if (!bcrypt.compareSync(contrasena, user.contrasena)) {
      return res.status(401).json({ msg: "Credenciales incorrectas" });
    }

    res.status(200).json({ token: crearToken(user), msg: "Login exitoso" });
  } catch (error) {
    res.status(400).json({ mensaje: error.message });
  }
}
```

Esta función a su vez, llama a una segunda función, la cual permite generar el token con el cual el usuario obtendrá permisos para visitar las demás rutas. Este token solo tiene validez durante 20 minutos, posterior a esto el usuario tiene que generar un nuevo token.

```
function crearToken(user) {  
  const payload = {  
    nombres: user.nombres,  
    apellidos: user.apellidos,  
    correo: user.correo,  
    //rol: user.rol,  
    exp: (Date.now() + (20 * 60 * 1000))  
  };  
  return jwt.sign(payload, SECRET)  
}
```

La función verificarTokenSesion, permite validar si el token del usuario es válido y no está expirado o alterado. Esta ruta funciona mediante el verbo GET.

```
const verificarTokenSesion = (req, res) => {  
  const token = req.headers['authorization'];  
  if (!token) {  
    return res.status(401).json({ msg: "No se encontro el token", status:"401" });  
  }  
  else {  
    try {  
      let payload = jwt.verify(token, SECRET);  
      //comprobacion de si el token expiro o no  
      if (Date.now() > payload.exp) {  
        return res.status(401).json({ msg: "Token expirado", status:"401" });  
      }  
      return res.status(200).json({ msg: "Token valido", status:"200" });  
    } catch (error) {  
      return res.status(401).json({ msg: error.message, status:"401" });  
    }  
  }  
}
```

También se implementó un controlador para obtener información básica del usuario, como el rol y el id del usuario. Este controlador se pensó para facilitar la programación en el lado Front.

```
const getRol = async (req, res) => {  
  const token = req.headers['authorization'];  
  
  try {  
    let payload = jwt.verify(token, SECRET);  
    let correo = payload.correo;  
    const user = await Cliente.findOne({ where: { correo: correo } });  
    return res.status(200).json({ rol: user.rol, idCliente: user.idCliente });  
  } catch (error) {  
    return res.status(400).json({ msg: error.message });  
  }  
}
```

## 4. Middleware

Para cargar las imágenes de los productos, se implementó un middleware mediante la librería multer. La función única de este middleware es almacenar y asignar un nombre único a las imágenes que se cargan desde el frontend

```
import multer from "multer";
import * as mimeTypes from "mime-types"

const storage = multer.diskStorage({
  destination: "public/productos/",
  filename: function (req, file, cb) {
    cb("", Date.now() + "." + mimeTypes.extension(file.mimetype));
  }
})
const upload = multer({ storage: storage })

export { upload }
```

Para proteger las rutas y hacer necesario que el usuario se autentique, se implementó un middleware que se ejecuta antes que la mayoría de las rutas. Este middleware valida que la petición traiga el token y que además este no esté expirado.

```
import jwt from "jsonwebtoken";

let SECRET = process.env.SECRET

function verificarToken (req, res, next){
  const token= req.headers['authorization'];

  if(!token){
    return res.status(401).json({msg:"No se encontro el token"});
  }
  else{
    try{
      let payload=jwt.verify(token,SECRET);
      //comprobacion de si el token expiro
      if(Date.now()>payload.exp){
        return res.status(401).json({msg:"Token expirado"});
      }
    }catch(error){
      return res.status(401).json({msg:"Error 500 "+error.message});
    }
  }
  next();
}

export {verificarToken };
```

Para el envío de correos electrónicos, se implementó la librería nodemailer para construir la funcionalidad que permite enviar correos electrónicos a los clientes cuando sus pedidos han sido procesados

```

import nodemailer from "nodemailer"

let EMAIL= process.env.EMAIL_USER;
let PASS= process.env.EMAIL_PASS

const transporter= nodemailer.createTransport({
  host:"smtp.office365.com",
  port: 587,
  secure: false,
  auth:{
    user:EMAIL,
    pass:PASS
  }
});

const enviarCorreoElectronico =async (destinatario,asunto,cuerpo)=>{
  try{
    const result= await transporter.sendMail({
      from:`Fruver ${EMAIL}`,
      to:destinatario,
      subject:asunto,
      text:cuerpo
    });
    console.log(result);
    return true;
  }
  catch(error){
    console.log(error);
    return false;
  }
}

```

Y para finalizar, también se implementó una función para generar el reporte de la compra, la cual se envía como cuerpo del correo electrónico de confirmación

```

const generarReporte=(pedido)=>{
  let nombresCliente=pedido.cliente.nombres;

  let cuerpoCorreo="Hola "+nombresCliente+", hemos procesado y enviado tu pedido \n\n";
  cuerpoCorreo+="Descripción del pedido:\n\n";

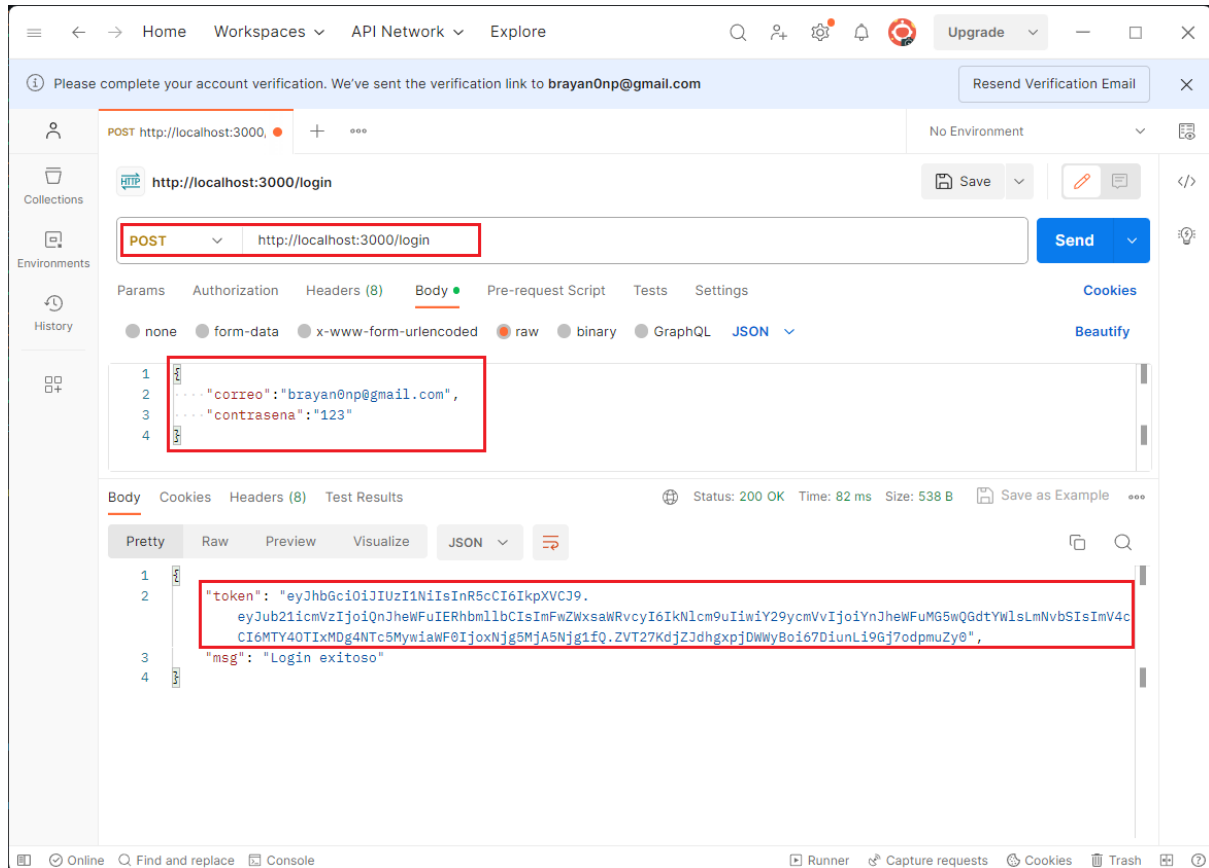
  pedido.detallePedidos.forEach(element => {
    //cuerpoCorreo+=" "+element.producto.nombre+"\t\t"+(element.cantidadProducto+" X "+element.producto.valor+"$").padEnd(15)+"\t\t= "+element.totalProducto+"$\n";
    cuerpoCorreo+=" "+element.producto.nombre+" = "+element.cantidadProducto+" X "+element.producto.valor+"$"+" = "+element.totalProducto+"$\n";
  });
  cuerpoCorreo+="-----\n";
  cuerpoCorreo+="Total: "+pedido.total+"$\n\n";
  cuerpoCorreo+="Gracias por tu compra!";
  console.log(cuerpoCorreo);
  return cuerpoCorreo;
}

```



## Validaciones con Postman

Debido a que la mayoría de las rutas requieren de autenticación para poderlas usar, es necesario en primer lugar generar dicho token, para ello debemos autenticarnos desde postman enviando un correo electrónico y la contraseña de un usuario ya registrado.



El token que se genera, se debe utilizar de aquí en adelante para poder autenticarse, dicho token debe enviarse en la cabecera Authorization de la petición.

## Rutas para productos:

### GET /productos

The screenshot shows a Postman interface with a GET request to `http://localhost:3000/productos`. The request has an Authorization header with a Bearer token. The response status is 200 OK, and the body is a JSON array of three products.

Key	Value	Description
Authorization	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJub21icmVzIjoiaWJheWVudlERhbmllbC...	

```
[{"idProducto": 1, "nombre": "Manzana", "detalle": "Manzana roja", "valor": 1000, "imagen": "1689131252220.jpeg"}, {"idProducto": 2, "nombre": "Tomate", "detalle": "Tomate de carne", "valor": 500, "imagen": "1689131261287.jpeg"}, {"idProducto": 3, "nombre": "Banana", "detalle": "Banana del pacifico", "valor": 1500, "imagen": "1689131269300.jpeg"}]
```

### GET /producto/:idProducto

The screenshot shows a Postman interface with a GET request to `http://localhost:3000/producto/1`. The response status is 200 OK, and the body is a JSON object representing the first product.

```
{ "idProducto": 1, "nombre": "Manzana", "detalle": "Manzana roja", "valor": 1000, "imagen": "1689131252220.jpeg" }
```

## POST /productos:

Debido a que los productos deben tener una imagen, la información se debe enviar como un form data

POST http://localhost:3000/productos

Body: form-data

Key	Value	Description
nombre	Pera	
detalle	Pera verde	
valor	700	
file	perasjpg.jpg	

Status: 200 OK Time: 118 ms Size: 366 B

```
1 {
2   "idProducto": 5,
3   "nombre": "Pera",
4   "detalle": "Pera verde",
5   "valor": "700",
6   "imagen": "1689211131971.jpeg"
7 }
```

## PUT /productos/:idProducto

PUT http://localhost:3000/productos/5

Body: form-data

Key	Value	Description
nombre	Peras	
detalle	Peras verdes de Colombia	
valor	750	
file	peras2.jpg	

Status: 200 OK Time: 90 ms Size: 382 B

```
1 {
2   "idProducto": 5,
3   "nombre": "Peras",
4   "detalle": "Peras verdes de Colombia",
5   "valor": "750",
6   "imagen": "1689211514137.jpeg"
7 }
```

## DELETE /productos/:idProducto

The screenshot displays the Postman interface for a DELETE request. The URL bar shows `http://localhost:3000/productos/5` with the method `DELETE` selected. The response status is `200 OK`, and the response body is a JSON object: `{"mensaje": "Registro Eliminado"}`.

**Request Details:**

- Method: `DELETE`
- URL: `http://localhost:3000/productos/5`
- Body: This request does not have a body.

**Response Details:**

- Status: `200 OK`
- Time: 32 ms
- Size: 299 B
- Body (JSON):

```
{  "mensaje": "Registro Eliminado"}
```

## Rutas para pedidos:

GET /pedidos

The screenshot shows the Postman interface with a GET request to `http://localhost:3000/pedidos` successfully executed. The response status is 200 OK, and the body contains a JSON object representing an order.

```
1 {
2   "idPedido": 6,
3   "total": 2050,
4   "procesado": true,
5   "createdAt": "2023-07-12T17:38:29.000Z",
6   "updatedAt": "2023-07-12T17:39:06.000Z",
7   "idCliente": 3,
8   "detallePedidos": [
9     {
10      "id": 11,
11      "cantidadProducto": 3,
12      "totalProducto": 1050,
13      "idPedido": 6,
14      "idProducto": 4,
15      "producto": {
16        "idProducto": 4,
17        "nombre": "Limon",
18        "detalle": "Limon Tahiti",
19        "valor": 350,
20        "imagen": "1689174659074.jpeg"
21      }
22    },
23    {
24      "id": 12,
25      "cantidadProducto": 2,
26      "totalProducto": 1000,
27      "idPedido": 6,
28      "idProducto": 2,
29      "producto": {
30        "idProducto": 2,
31        "nombre": "Tomate",
32        "detalle": "Tomate de carne",
33        "valor": 500,
34        "imagen": "1689131261287.jpeg"
35      }
36    }
37  ],
38   "cliente": {
39     "idCliente": 3,
40     "nombres": "Brayan Daniel",
41     "apellidos": "Ceron",
42     "correo": "brayan0np@gmail.com",
43     "contrasena": "$2b$08$wQW.NcAtC6FvNbigtPSf7evwKBKgGXKh2F3DDqFBonmFXc6A/zhy",
44     "rol": "user"
45   }
46 }
47 { ...
48 }
```

## GET /pedido/:idPedido

The screenshot shows the Postman interface with a GET request to `http://localhost:3000/pedido/1` successfully executed. The response status is 200 OK, and the body contains a JSON object representing a purchase order.

**Request:**

- Method: GET
- URL: `http://localhost:3000/pedido/1`

**Response:**

```
1 {
2   "idPedido": 1,
3   "total": 15000,
4   "procesado": true,
5   "createdAt": "2023-07-12T01:54:38.000Z",
6   "updatedAt": "2023-07-12T02:42:51.000Z",
7   "idCliente": 2,
8   "detallePedidos": [
9     {
10      "id": 1,
11      "cantidadProducto": 8,
12      "totalProducto": 12000,
13      "idPedido": 1,
14      "idProducto": 3,
15      "producto": {
16        "idProducto": 3,
17        "nombre": "Banana",
18        "detalle": "Banana del pacifico",
19        "valor": 1500,
20        "imagen": "1689131269300.jpeg"
21      }
22    },
23    {
24      "id": 2,
25      "cantidadProducto": 6,
26      "totalProducto": 3000,
27      "idPedido": 1,
28      "idProducto": 2,
29      "producto": {
30        "idProducto": 2,
31        "nombre": "Tomate",
32        "detalle": "Tomate de carne",
33        "valor": 500,
34        "imagen": "1689131261287.jpeg"
35      }
36    }
37  ]
38 }
```

GET /pedidos/cliente/idCliente

Esta ruta es para obtener los pedidos de un cliente

The screenshot shows the Postman API client interface. The request is a GET to `http://localhost:3000/pedidos/cliente/2`. The response is a JSON array of two order objects. The first object represents an order for 'Limon' (Lemon) with a total value of 900. The second object represents an order for 'Banana' with a total value of 15000. Both orders are associated with a customer named 'Brayan Dan'.

```
1 {
2   {
3     "idPedido": 5,
4     "total": 900,
5     "procesado": true,
6     "createdAt": "2023-07-12T15:05:40.000Z",
7     "updatedAt": "2023-07-12T15:07:51.000Z",
8     "idCliente": 2,
9     "detallePedidos": [
10      {
11        "id": 10,
12        "cantidadProducto": 3,
13        "totalProducto": 900,
14        "idPedido": 5,
15        "idProducto": 4,
16        "producto": {
17          "idProducto": 4,
18          "nombre": "Limon",
19          "detalle": "Limon Tahiti",
20          "valor": 300,
21          "imagen": "1689174659074.jpeg"
22        }
23      }
24    ],
25    "cliente": {
26      "idCliente": 2,
27      "nombres": "Brayan Dan",
28      "apellidos": "Ceron",
29      "correo": "brayanCeron@email.com",
30      "contrasena": "$2b$08$I5fS1Eydh8CyZELs710DGTePIZyh6vP/okn.7zhdBZkLmz0g/xSQUg",
31      "rol": "user"
32    }
33  },
34  {
35    "idPedido": 1,
36    "total": 15000,
37    "procesado": true,
38    "createdAt": "2023-07-12T01:54:38.000Z",
39    "updatedAt": "2023-07-12T02:42:51.000Z",
40    "idCliente": 2,
41    "detallePedidos": [
42      {
43        "id": 1,
44        "cantidadProducto": 8,
45        "totalProducto": 12000,
46        "idPedido": 1,
47        "idProducto": 3,
48        "producto": {
49          "idProducto": 3,
50          "nombre": "Banana",
51          "detalle": "Banana del pacifico",
52          "valor": 1500,
53          "imagen": "1689131269300.jpeg"
54        }
55      },
56      {
57        "id": 2,
58        "cantidadProducto": 6,
59        "totalProducto": 3000,
60        "idPedido": 1,
61        "idProducto": 2,
62        "producto": {
63          "idProducto": 2,
64          "nombre": "Tomate",
65          "detalle": "Tomate de carne",
66          "valor": 500,
67          "imagen": "1689131261287.jpeg"
68        }
69      }
70    ],
71    "cliente": {
72      "idCliente": 2,
73      "nombres": "Brayan Dan",
74      "apellidos": "Ceron",
75      "correo": "brayanCeron@email.com",
76      "contrasena": "$2b$08$I5fS1Eydh8CyZELs710DGTePIZyh6vP/okn.7zhdBZkLmz0g/xSQUg",
77      "rol": "user"
78    }
79  }
80 }
```

## POST /pedidos

Postman interface showing a POST request to `http://localhost:3000/pedidos`. The request body is a JSON object:

```
1 {
2   "idCliente": 3,
3   "productos": [
4     {
5       "idProducto": 1,
6       "cantidadProducto": 2
7     },
8     {
9       "idProducto": 2,
10      "cantidadProducto": 3
11     }
12   ]
13 }
```

The response status is 200 OK. The response body is a JSON object:

```
1 {
2   "idPedido": 7,
3   "total": 3500,
4   "procesado": false,
5   "createdAt": "2023-07-13T01:53:47.000Z",
6   "updatedAt": "2023-07-13T01:53:47.594Z",
7   "idCliente": 3
8 }
```

## PUT /pedidos/:idPedido

Postman interface showing a PUT request to `http://localhost:3000/pedidos/10`. The request body is a JSON object:

```
1 {
2   "productos": [
3     {
4       "idProducto": 2, "cantidadProducto": 4
5     },
6     {
7       "idProducto": 3, "cantidadProducto": 6
8     }
9   ],
10  "procesado": false
11 }
```

The response status is 200 OK. The response body is a JSON object:

```
1 {
2   "idPedido": 10,
3   "total": 7000,
4   "procesado": false,
5   "createdAt": "2023-07-18T21:38:23.000Z",
6   "updatedAt": "2023-07-26T15:23:13.856Z",
7   "idCliente": 2
8 }
```



## PUT /pedidos/procesar/:idPedido

Esta ruta marca como procesado un pedido

The screenshot shows a Postman interface with a PUT request to `http://localhost:3000/pedidos/procesar/7`. The request is successful, returning a `200 OK` status. The response body is a JSON object with the following structure:

```
1 {
2   "idPedido": 7,
3   "total": 3500,
4   "procesado": true,
5   "createdAt": "2023-07-13T01:53:47.000Z",
6   "updatedAt": "2023-07-13T01:56:48.578Z",
7   "idCliente": 3,
8   "detallePedidos": [
9     {
10      "id": 13,
11      "cantidadProducto": 2,
12      "totalProducto": 2000,
13      "idPedido": 7,
14      "idProducto": 1,
15      "producto": {
16        "idProducto": 1,
17        "nombre": "Manzana",
18        "detalle": "Manzana roja",
19        "valor": 1000,
20        "imagen": "1689131252220.jpeg"
21      }
22    },
23    { ... }
24  ],
25   "cliente": { ... }
26 }
```

## DELETE /pedidos/:idPedido

The screenshot shows a Postman interface with a DELETE request to `http://localhost:3000/pedidos/7`. The request is successful, returning a `200 OK` status. The response body is a JSON object with the following structure:

```
1 {
2   "mensaje": "Registro Eliminado"
3 }
```

## Rutas para clientes:

### GET /clientes

POST http://localhost:3000 GET http://localhost:3000/clientes

http://localhost:3000/clientes

GET http://localhost:3000/clientes

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Status: 200 OK Time: 22 ms Size: 988 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {
2   {
3     "idCliente": 1,
4     "nombres": "admin",
5     "apellidos": "",
6     "correo": "admin",
7     "contrasena": "$2b$08$H2cbI9KqpG0eH4HJGx2nzeIbtc/weoi51uj7Iy3K0rtjuji0zxGva",
8     "rol": "admin"
9   },
10  { ... },
17  { ... },
18  { ... },
25  { ... },
26  {
27    "idCliente": 4,
28    "nombres": "Maria Isabel",
29    "apellidos": "Ceron Portilla",
30    "correo": "ceronportillamaria@gmail.com",
31    "contrasena": "$2b$08$P92vMtjvA0Sm5Hmp4P88Au2Wui.ZK.iwwyU3ltYGcPP/3ZMgvQkFq",
32    "rol": "user"
33  }
34 }
```

### GET /cliente/:idCliente

POST http://localhost:3000 GET http://localhost:3000/cliente/3

http://localhost:3000/cliente/3

GET http://localhost:3000/cliente/3

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Status: 200 OK Time: 30 ms Size: 451 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {
2   {
3     "idCliente": 3,
4     "nombres": "Brayan Daniel",
5     "apellidos": "Ceron",
6     "correo": "brayan0np@gmail.com",
7     "contrasena": "$2b$08$WQW.NcAtC6FvNbgtPSf7evwKBKgGXKh2F3DDqFBonmFxc6A/zhhhy",
8     "rol": "user"
9   }
10 }
```

## POST /clientes

The screenshot shows a Postman interface with a POST request to `http://localhost:3000/clientes`. The request body is a JSON object with the following fields: `nombres`, `apellidos`, `correo`, and `contrasena`. The response status is `200 OK` with a time of 127 ms and a size of 446 B. The response body is a JSON object with the following fields: `rol`, `idCliente`, `nombres`, `apellidos`, `correo`, and `contrasena`.

```
1 {
2   "nombres": "Juanito",
3   "apellidos": "Perez",
4   "correo": "juanitoperez@gmail.com",
5   "contrasena": "123"
6 }
```

```
1 {
2   "rol": "user",
3   "idCliente": 5,
4   "nombres": "Juanito",
5   "apellidos": "Perez",
6   "correo": "juanitoperez@gmail.com",
7   "contrasena": "$2b$08$Mf13JfXKzJWzAGCLjAT3xeU9I7aDZnqT4S2N2qhyvN6Ic5y0xggm"
8 }
```

## PUT /clientes/:idCliente

The screenshot shows a Postman interface with a PUT request to `http://localhost:3000/clientes/5`. The request body is a JSON object with the following fields: `nombres` and `apellidos`. The response status is `200 OK` with a time of 100 ms and a size of 454 B. The response body is a JSON object with the following fields: `idCliente`, `nombres`, `apellidos`, `correo`, `contrasena`, and `rol`.

```
1 {
2   "nombres": "Juanito Pepito",
3   "apellidos": "Perez"
4 }
```

```
1 {
2   "idCliente": 5,
3   "nombres": "Juanito Pepito",
4   "apellidos": "Perez ",
5   "correo": "juanitoperez@gmail.com",
6   "contrasena": "$2b$08$Mf13JfXKzJWzAGCLjAT3xeU9I7aDZnqT4S2N2qhyvN6Ic5y0xggm",
7   "rol": "user"
8 }
```

## DELETE /clientes/:idCliente

The screenshot shows a Postman interface with a DELETE request to `http://localhost:3000/clientes/5`. The response status is `200 OK` with a time of 77 ms and a size of 299 B. The response body is a JSON object with the following field: `mensaje`.

```
1 {
2   "mensaje": "Registro Eliminado"
3 }
```