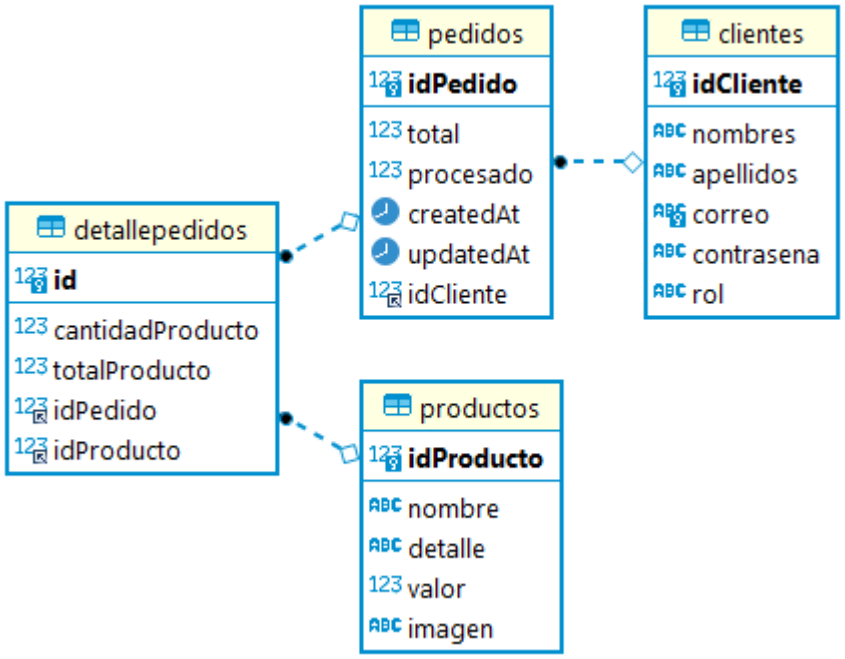
TALLER UNIDAD 2 BACKEND

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, verificarTokenSesion, 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", verificarTokenSesion);
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 toquen 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

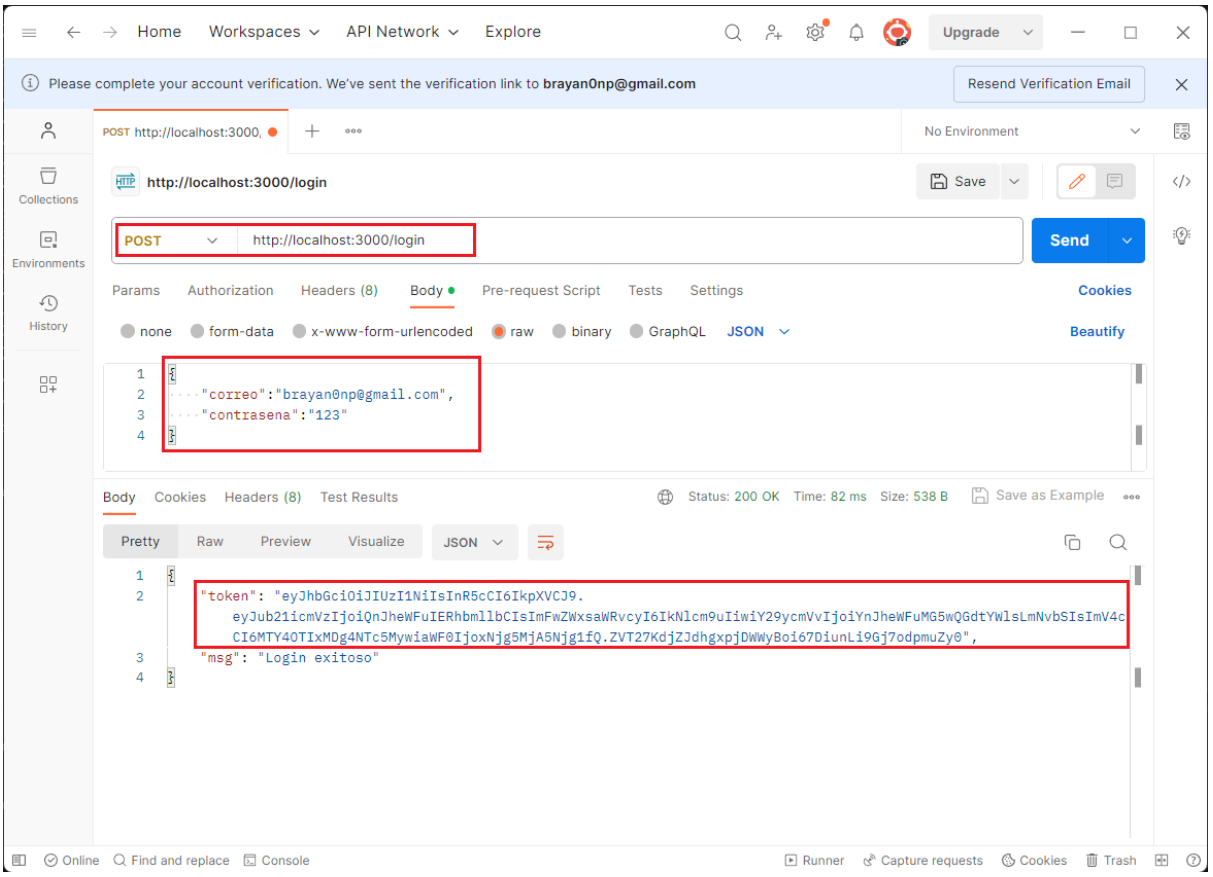
```
import nodemailer from "nodemailer"

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.

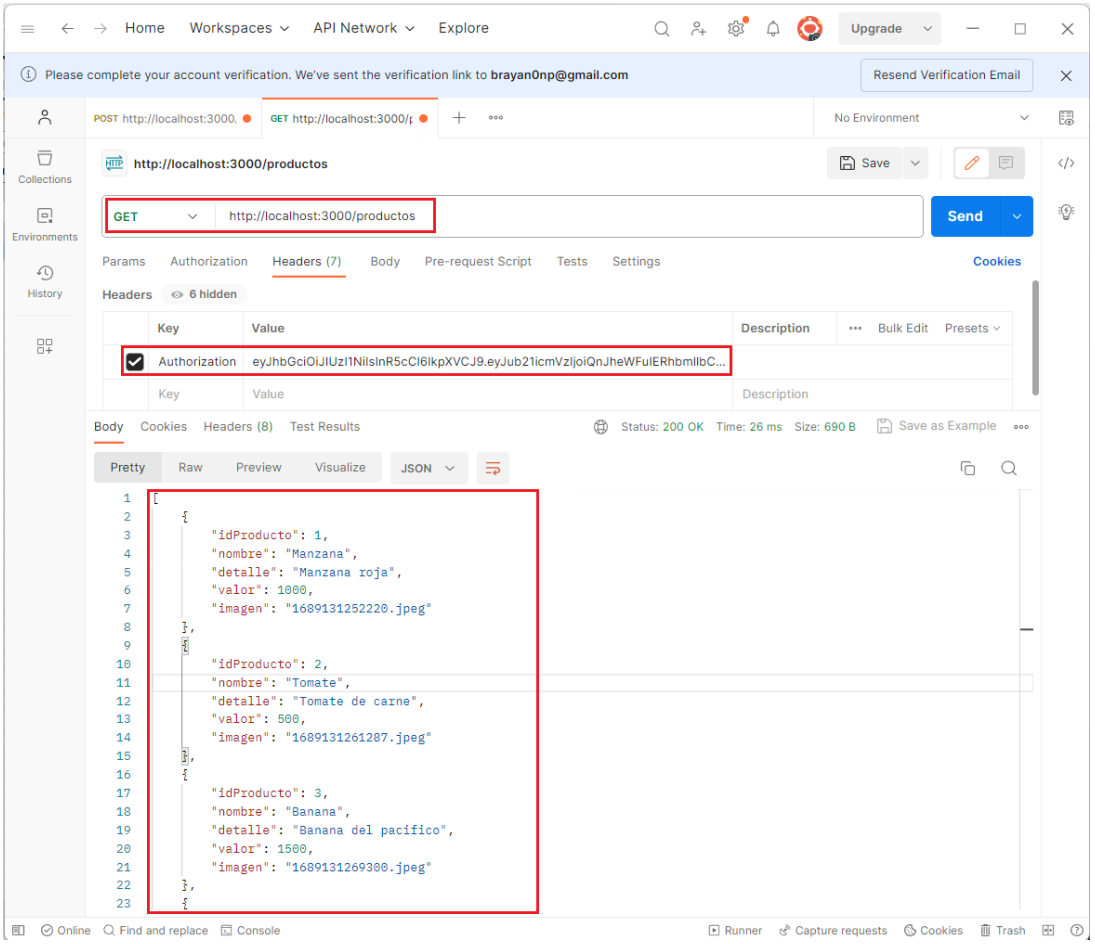


.

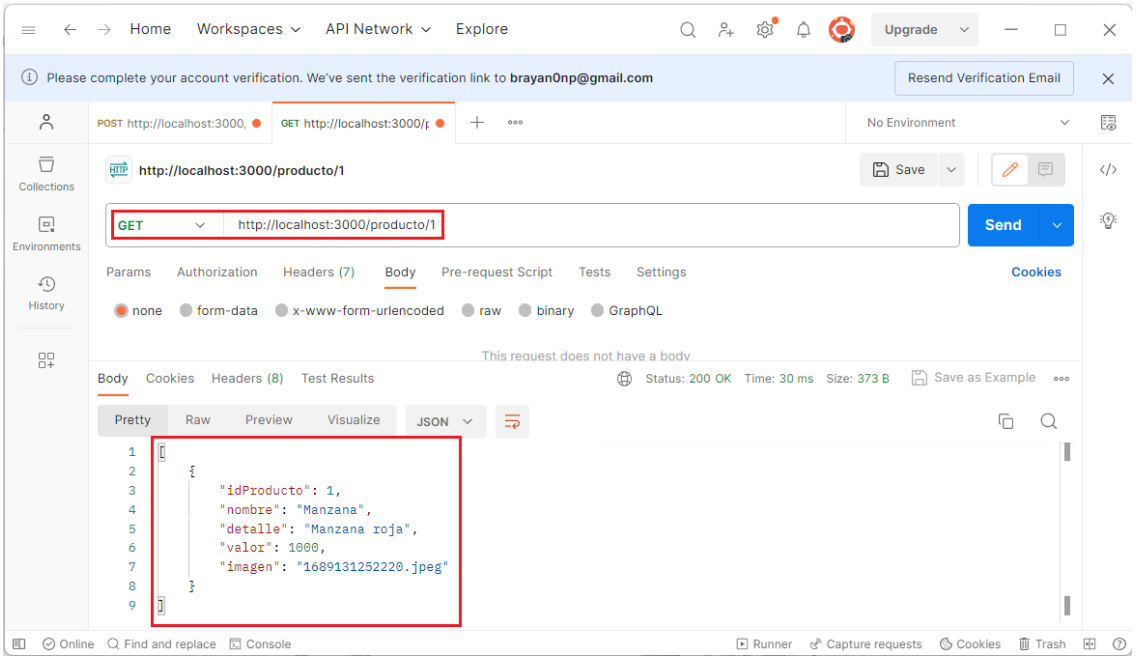
El token que se genera, se debe utilizar de aquí en adelante para poder autenticarse, dicho token debe enviarse en la cabecera Authorization de la petición.

Rutas para productos:

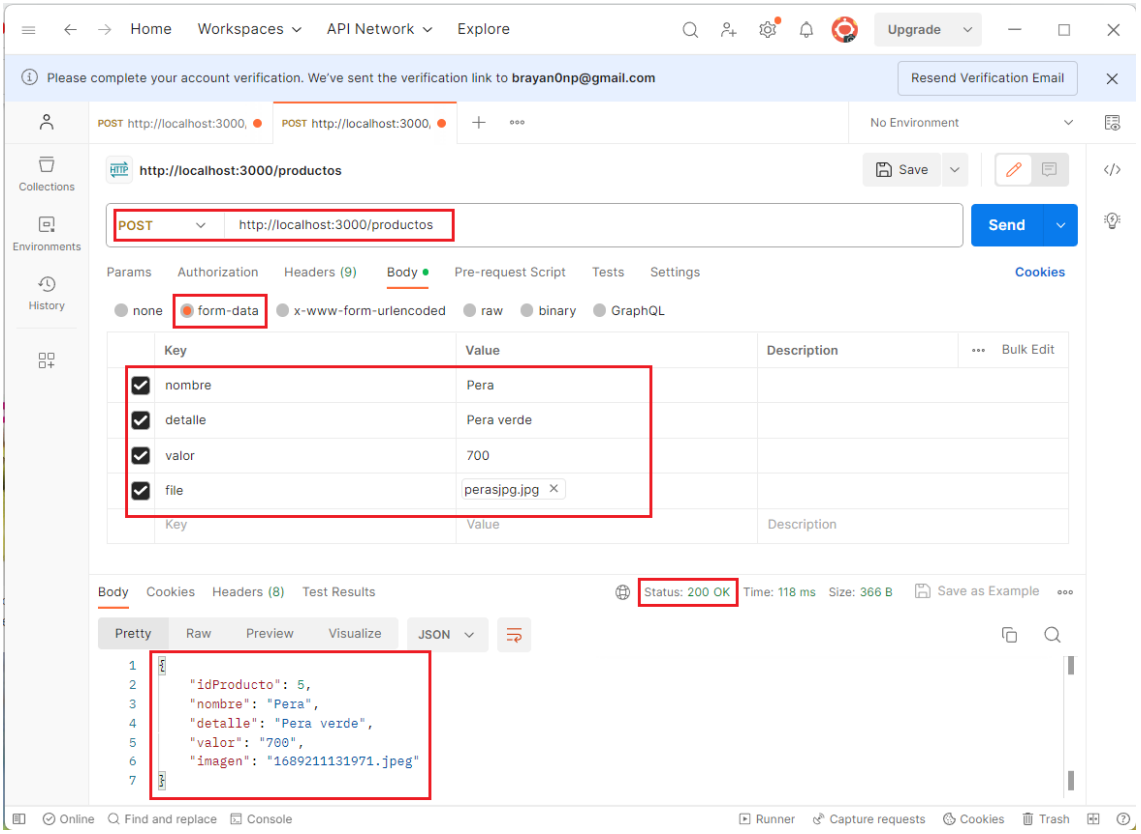
GET /productos



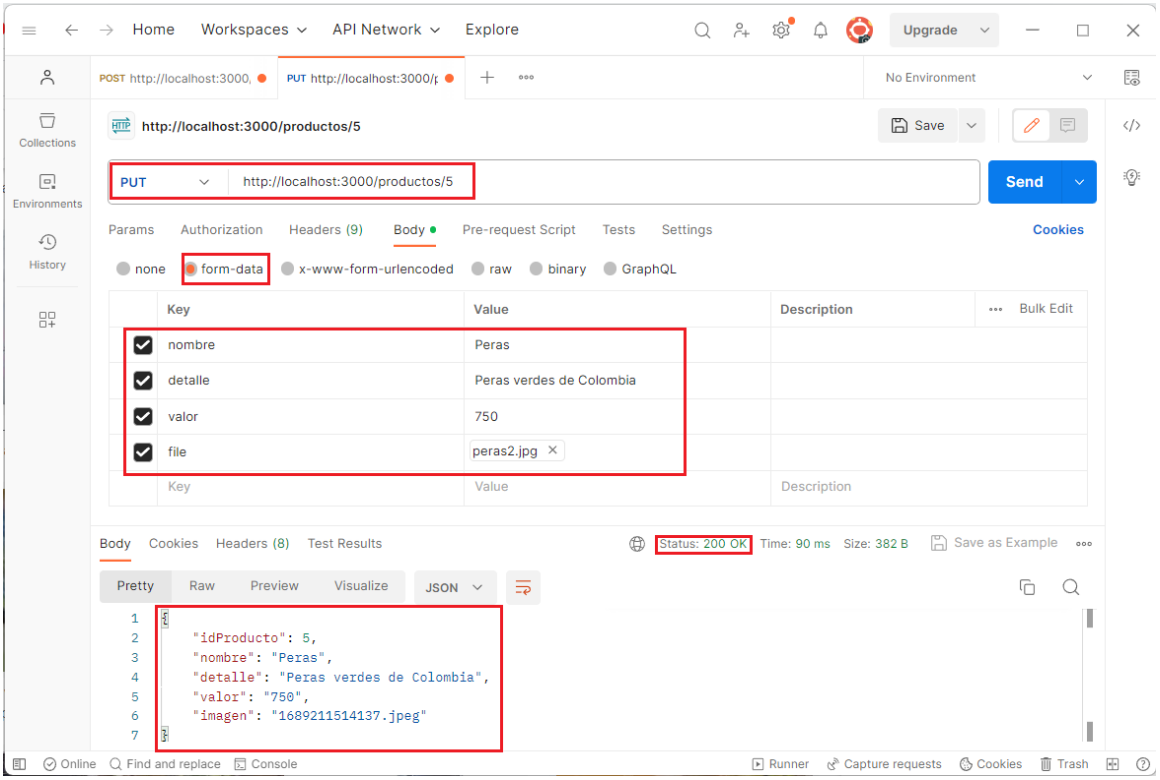
GET /producto/:idProducto



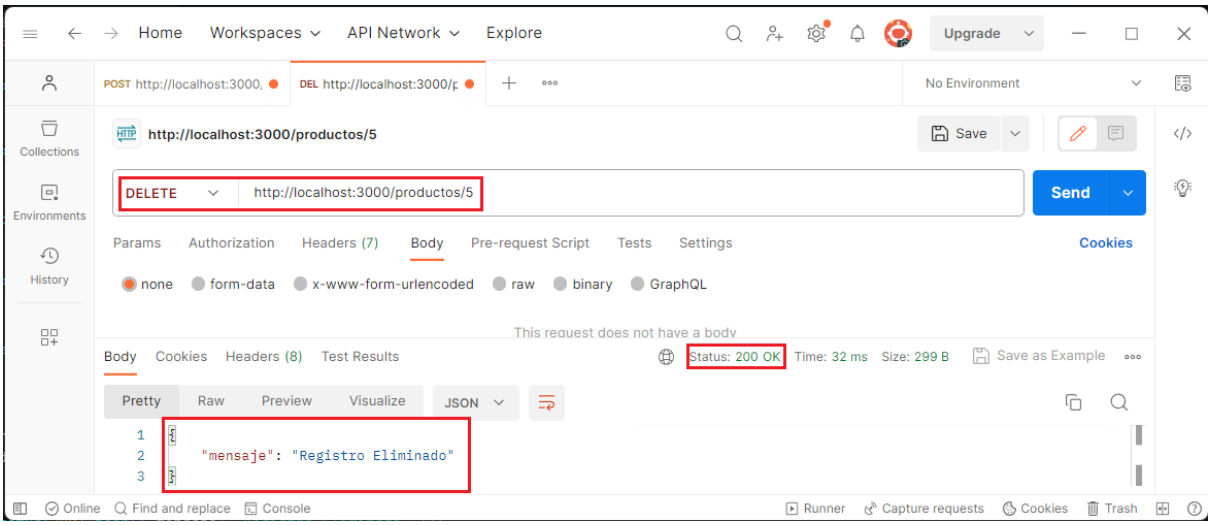
POST /productos:
Debido a que los productos deben tener una imagen, la información se debe enviar como un form data



PUT /productos/:idProducto

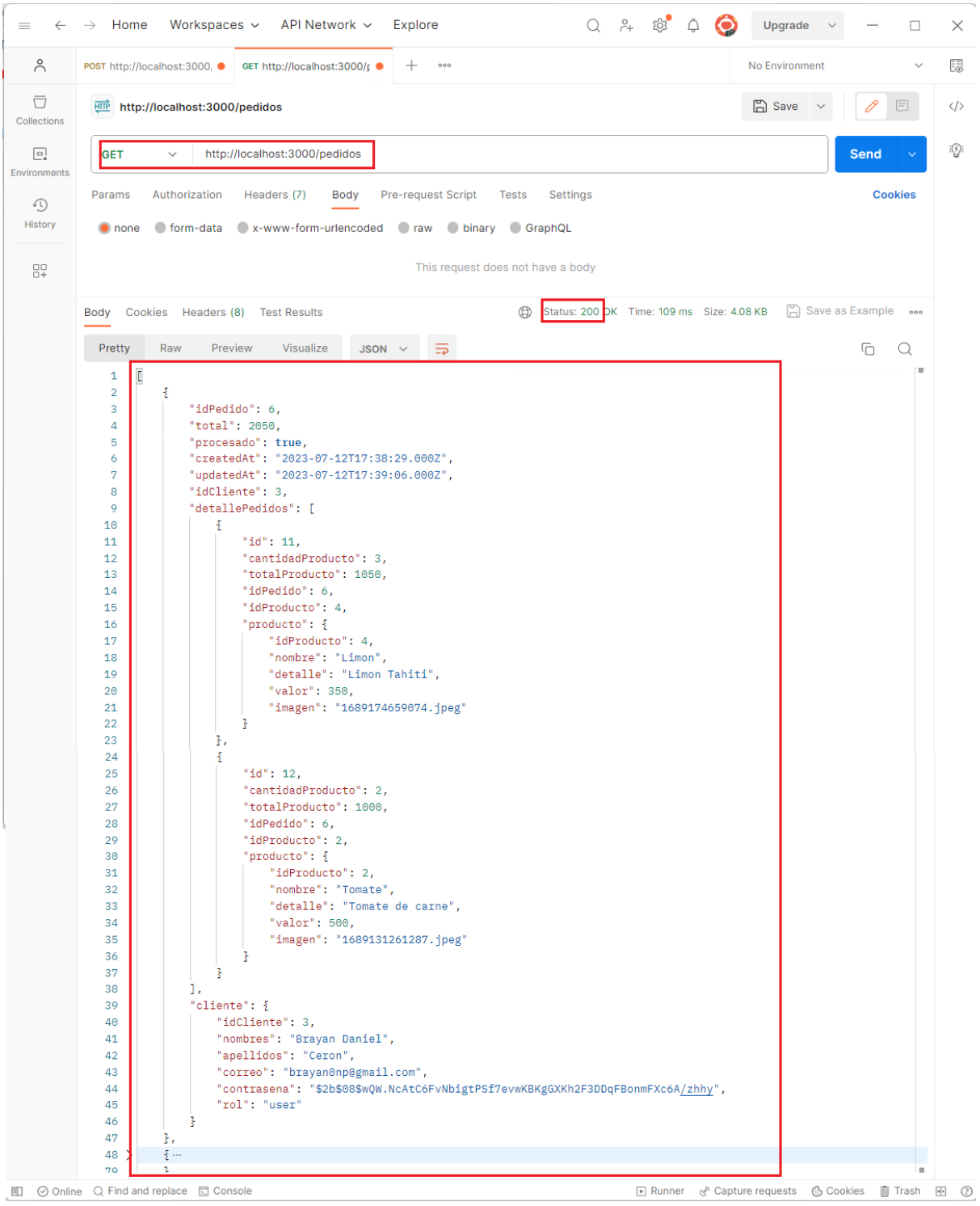


DELETE /productos/:idProducto

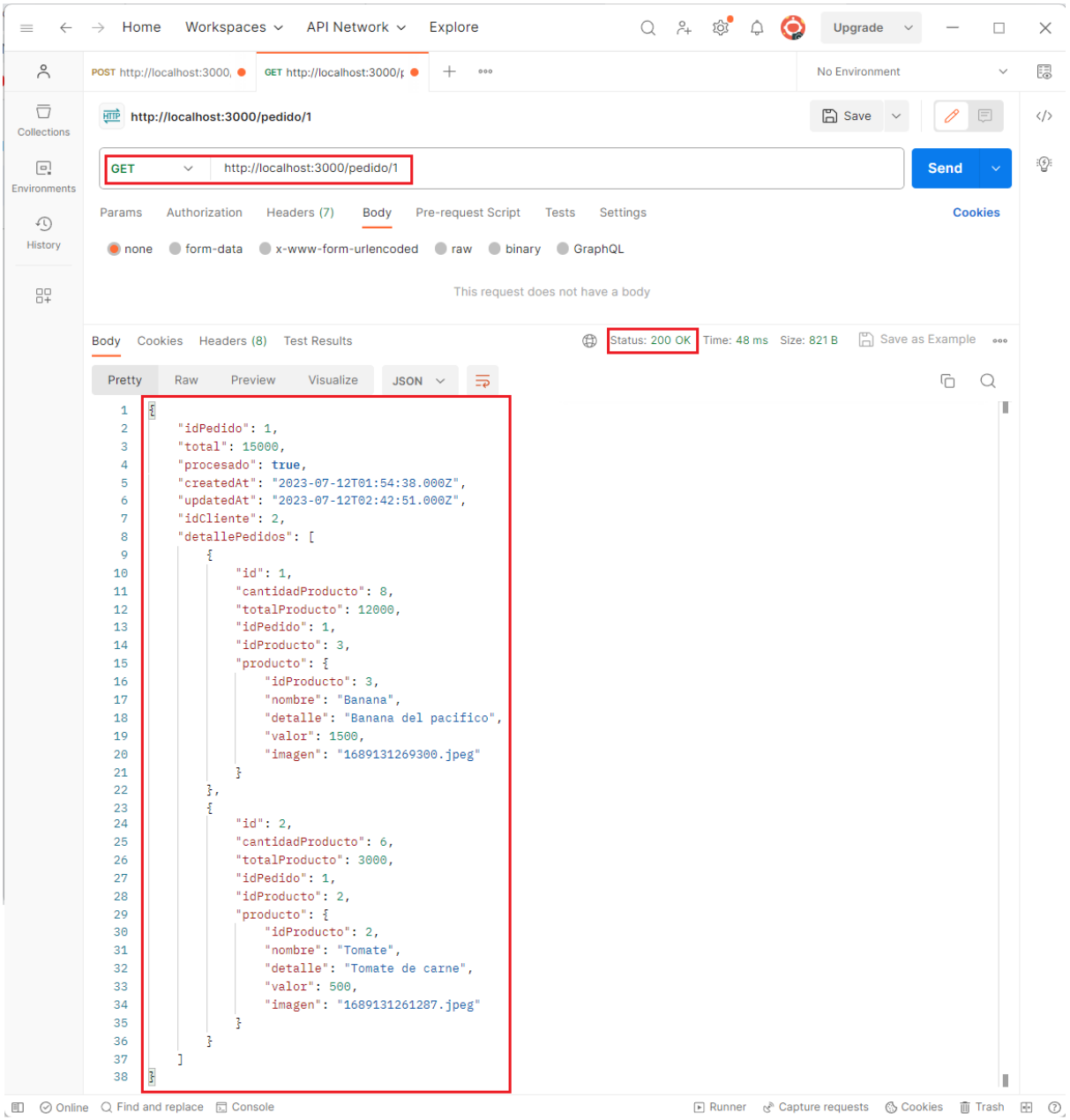


Rutas para pedidos:

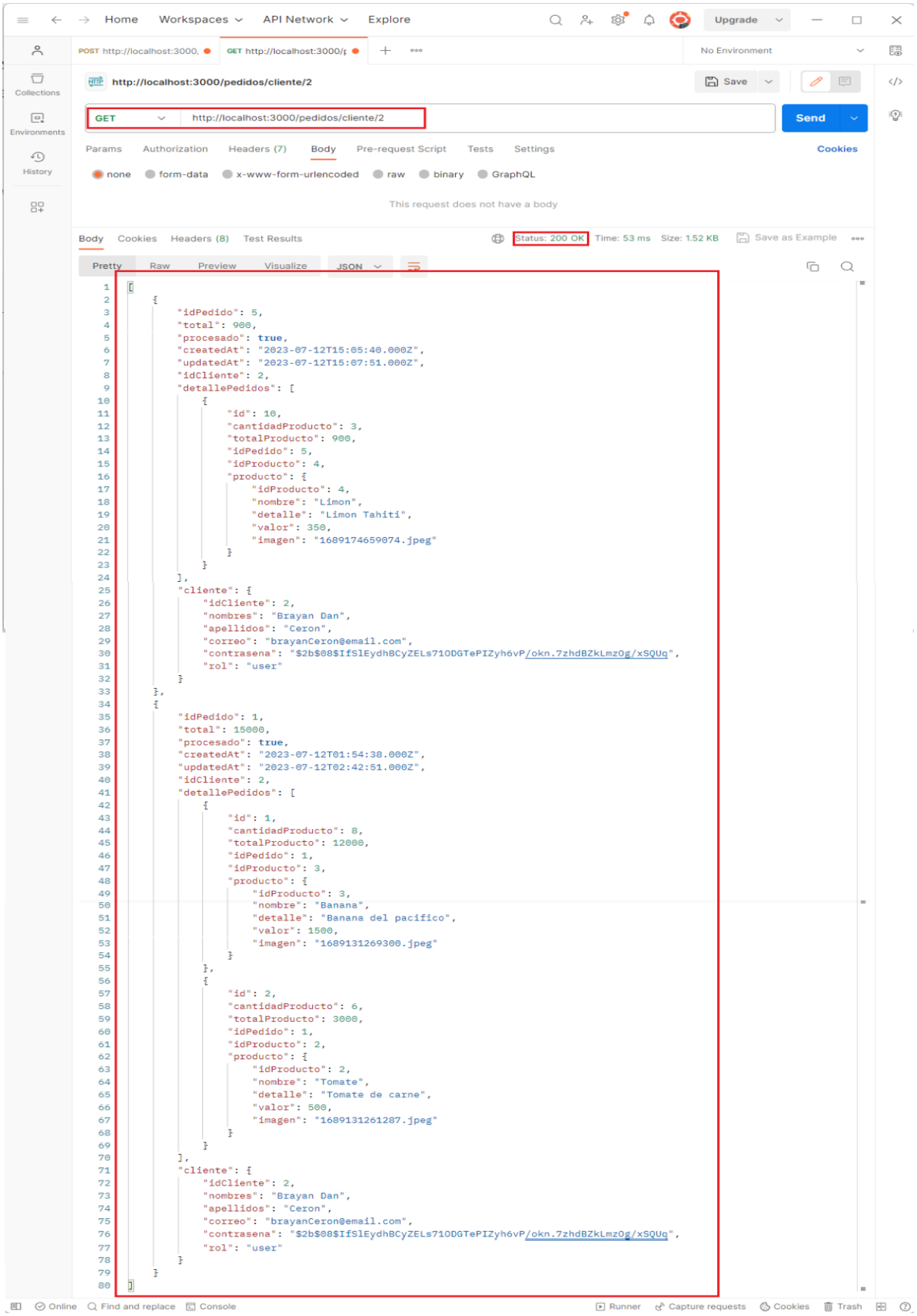
GET /pedidos



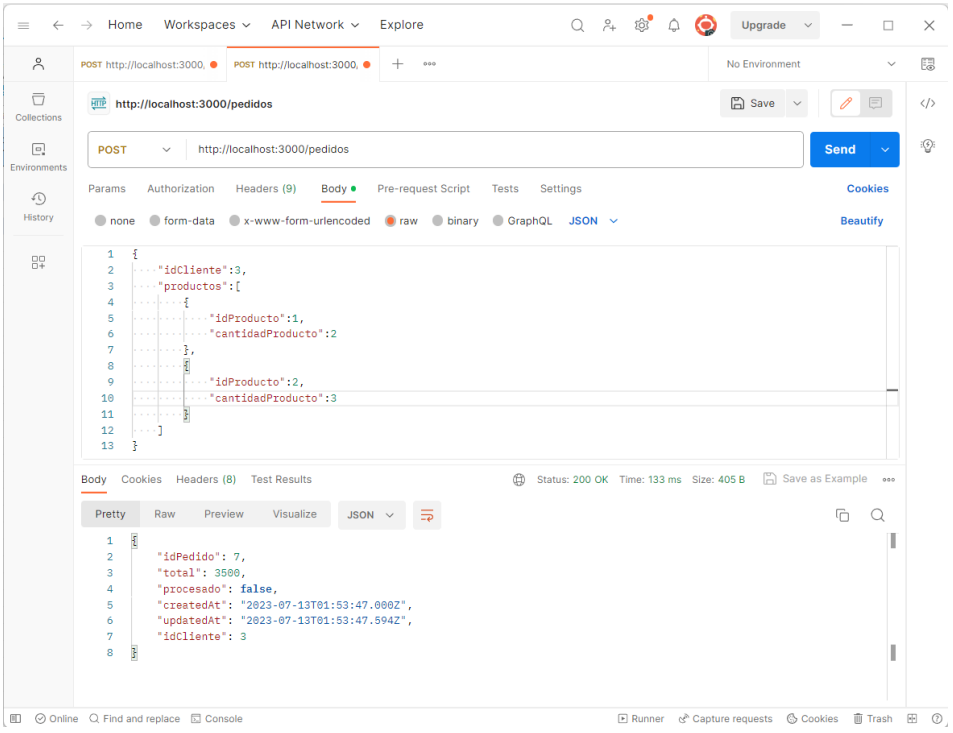
GET /pedido/:idPedido



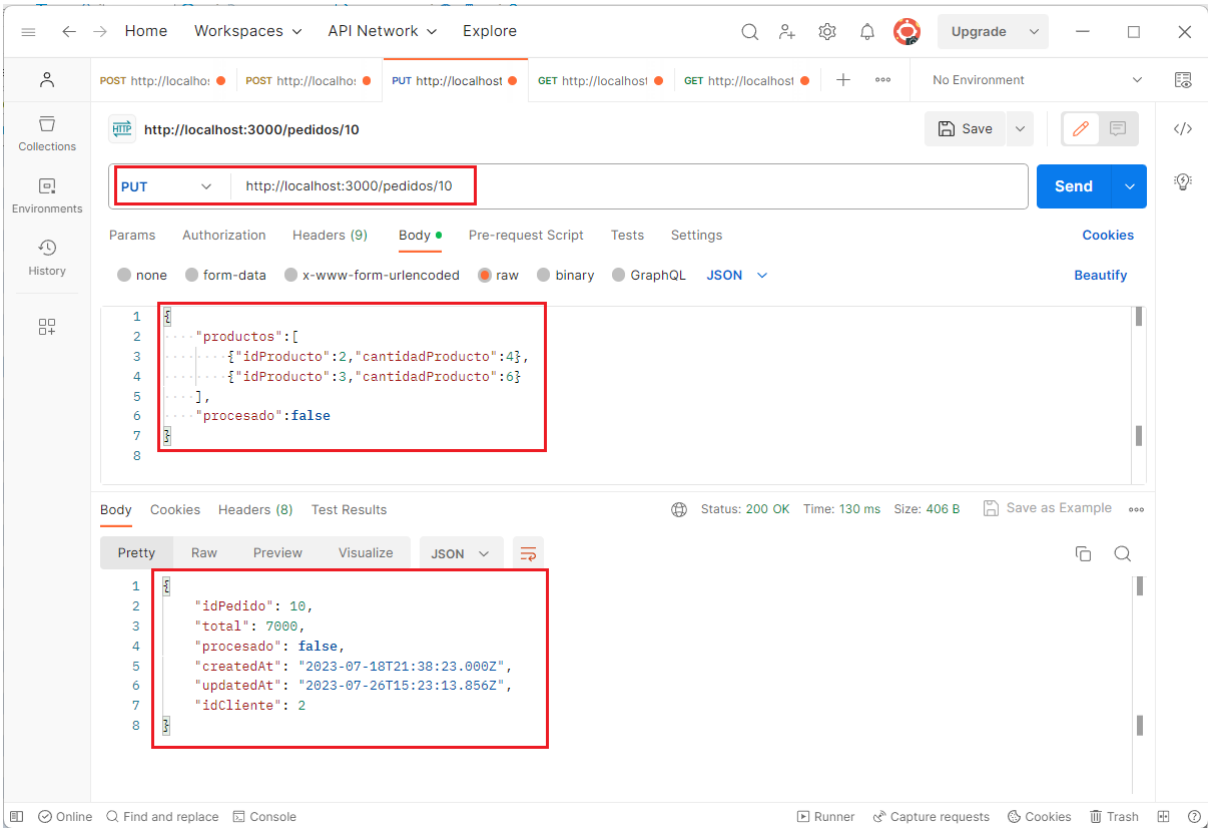
GET /pedidos/cliente/idCliente
Esta ruta es para obtener los pedidos de un cliente



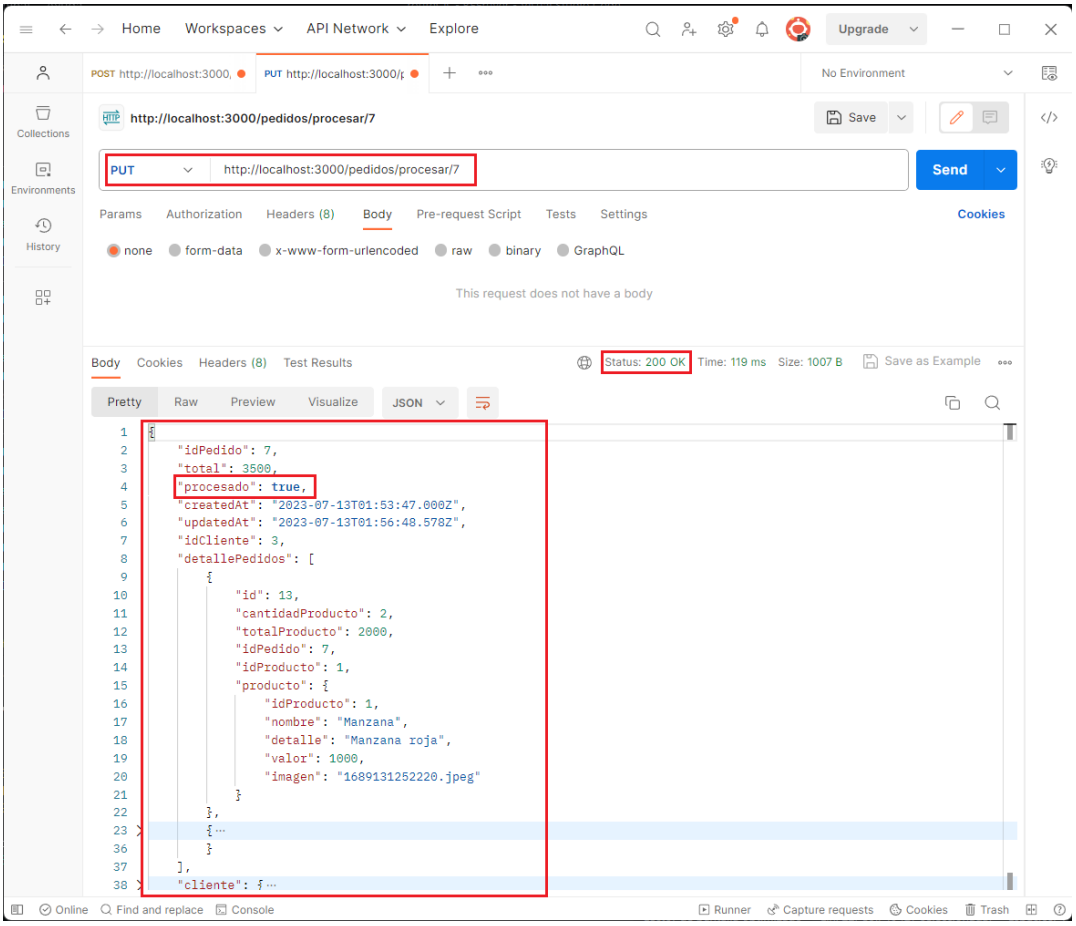
POST /pedidos



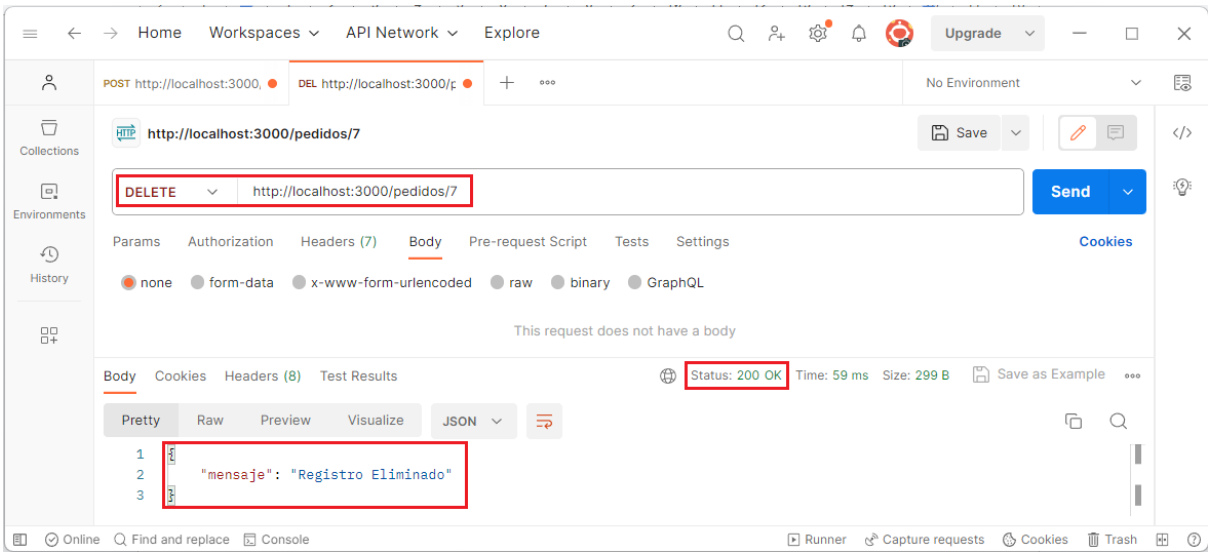
PUT /pedidos/:idPedido



PUT /pedidos/procesar/:idPedido
Esta ruta marca como procesado un pedido

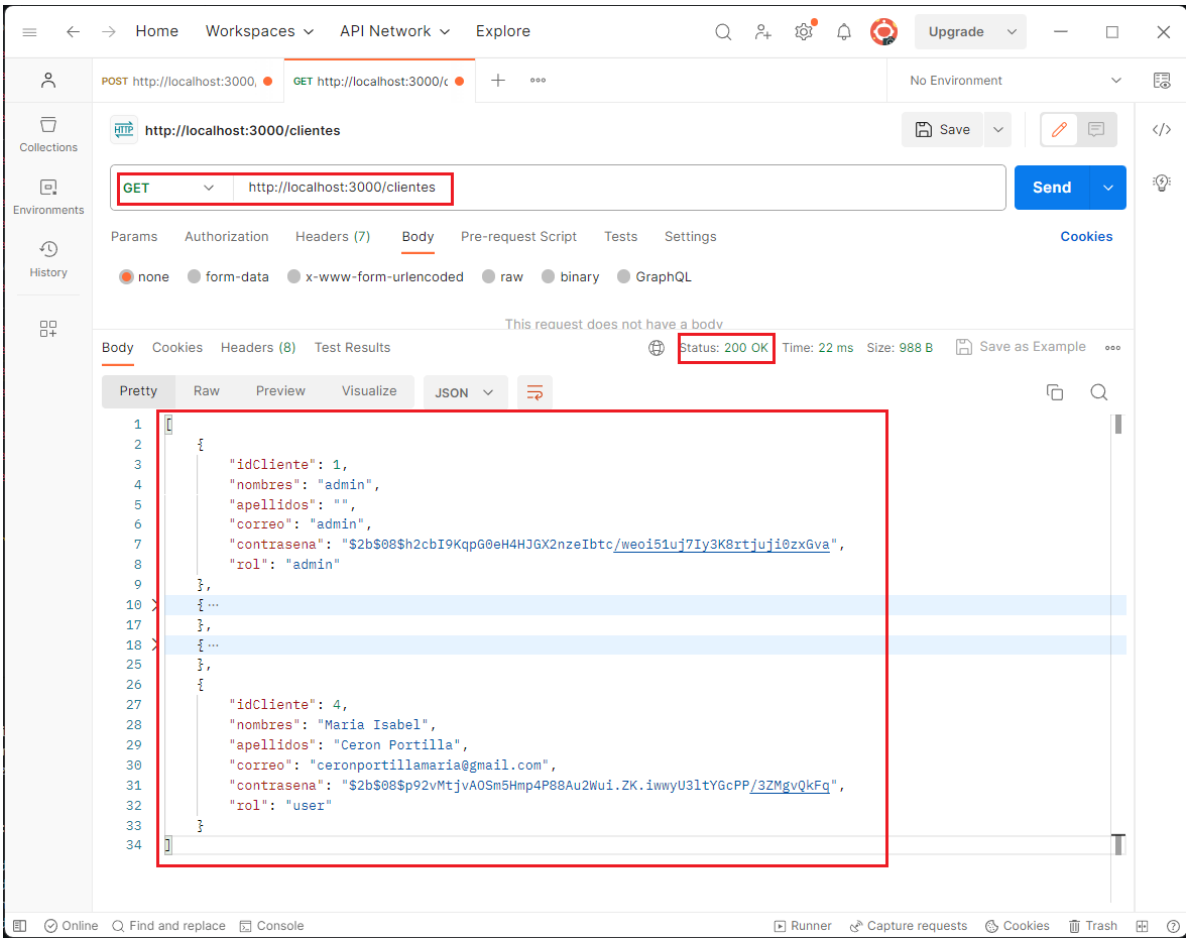


DELETE /pedidos/:idPedido

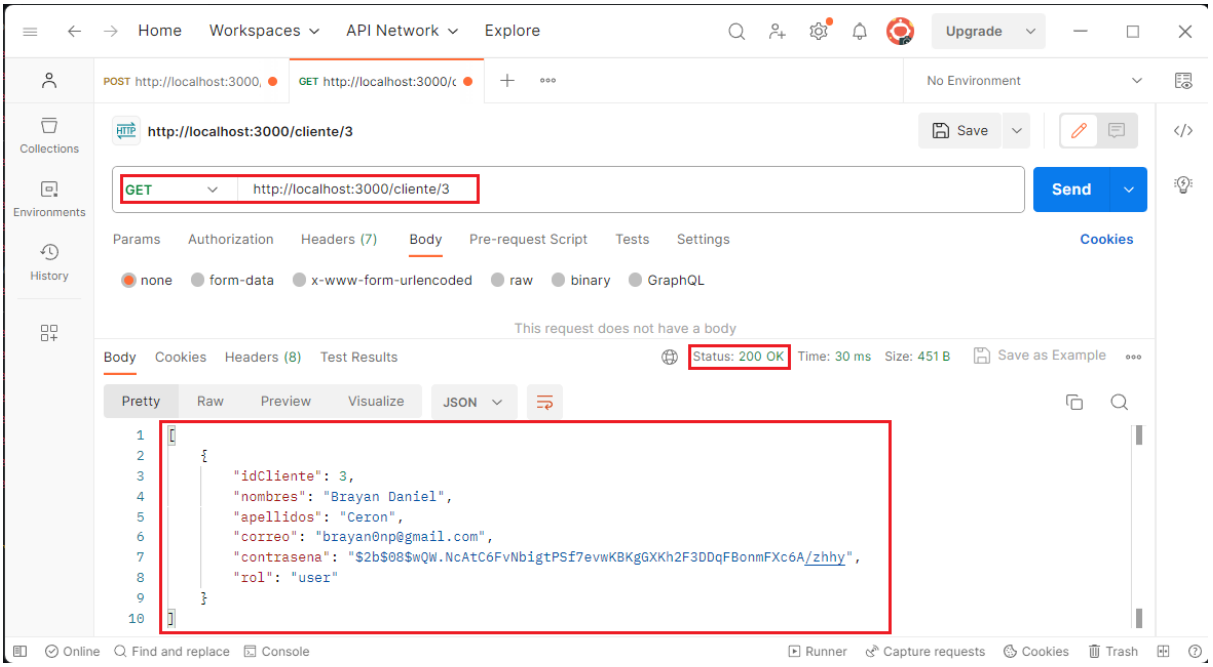


Rutas para clientes:

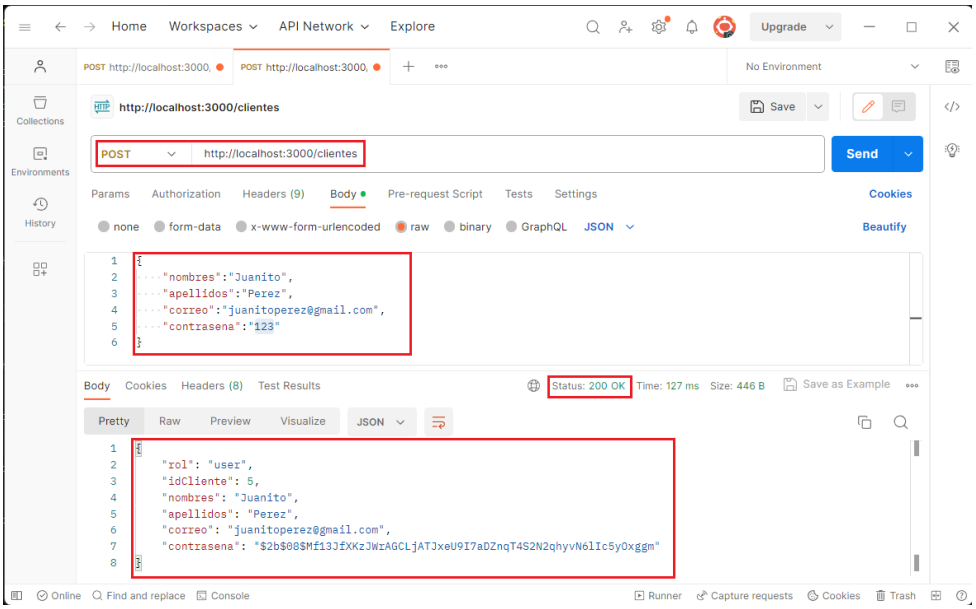
GET /clientes



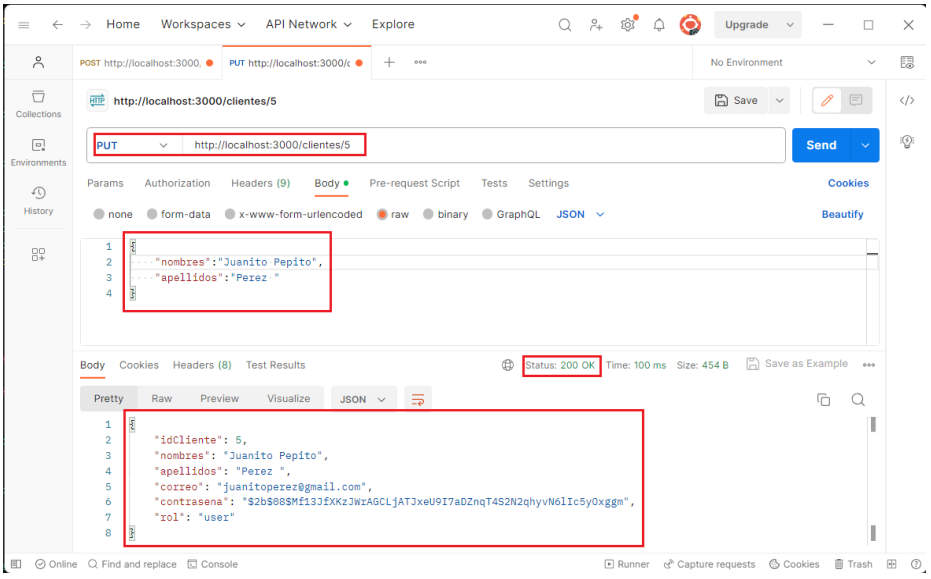
GET /cliente/:idCliente



POST /clientes



PUT /clientes/:idCliente



DELETE /clientes/:idCliente