Tutorial de apirest repo: https://github.com/arlemorales23/ApirestJs.git

1. Node.js:

Definición: Es un entorno de ejecución para JavaScript del lado del servidor, que permite ejecutar código JavaScript fuera del navegador.

Analogía: Piensa en Node.js como conductor de carro que permite que JavaScript, que tradicionalmente solo "conducía" dentro del navegador (como un carro en la ciudad), pueda también circular por otras partes, como una carretera fuera de la ciudad (el servidor).

2. Express:

Definición: Es un marco de trabajo minimalista para crear aplicaciones web con Node.js. Ayuda a gestionar rutas, peticiones HTTP, respuestas y middleware de manera más sencilla.

Analogía: Express sería el plano para construir una autopista eficiente. Organiza los carriles (rutas) y los semáforos (peticiones y respuestas) para que el tráfico de información fluya correctamente.

3. HTTP (Hypertext Transfer Protocol)

Definicion: Es el protocolo que permite la comunicación entre los navegadores web y los servidores. Es el conjunto de reglas que define cómo se deben formatear, transmitir y responder las solicitudes y respuestas que se intercambian a través de la web.

¿Cómo funciona HTTP?

HTTP es un protocolo sin estado (**stateless**), lo que significa que cada petición entre el cliente (navegador) y el servidor es independiente, y el servidor no recuerda las interacciones previas. Aunque hoy en día, tecnologías como las **cookies** y **sesiones** permiten mantener cierto estado entre varias peticiones.

Tipos de peticiones HTTP

Cada vez que un cliente (como un navegador) quiere interactuar con un servidor, envía una **petición HTTP**. Estas peticiones pueden ser de diferentes tipos, dependiendo de la acción que el cliente desea realizar:

1. **GET**:
    - **Descripción**: Solicita datos del servidor. Se usa para obtener información sin modificar nada en el servidor.
    - **Ejemplo**: Cuando visitas una página web, el navegador realiza una petición GET para obtener el contenido de esa página.
    - **Analogía**: Es como leer un periódico en línea. No estás modificando nada, solo accediendo a la información.

2. **POST**:
   - **Descripción**: Envía datos al servidor, generalmente para crear o procesar información. Se utiliza en formularios o al subir archivos.
   - **Ejemplo**: Cuando llenas un formulario de contacto o de registro, el navegador envía los datos con una petición POST.
   - **Analogía**: Es como enviar una carta por correo; envías información que el destinatario (servidor) debe procesar.
3. **PUT**:
   - **Descripción**: Actualiza o reemplaza un recurso existente en el servidor con los datos proporcionados.
   - **Ejemplo**: Cuando actualizas tu perfil en una red social, se podría usar una petición PUT para reemplazar tu información antigua con la nueva.
   - **Analogía**: Es como reemplazar el contenido de un archivo que ya existe en tu computadora con nueva información.
4. **PATCH**:
   - **Descripción**: Similar a PUT, pero en lugar de reemplazar todo el recurso, solo modifica partes específicas de él.
   - **Ejemplo**: Actualizar solo el campo "nombre" en tu perfil de usuario, sin tocar otros datos.
   - **Analogía**: Es como corregir una página de un libro con un marcador, en lugar de reescribir todo el capítulo.
5. **DELETE**:
   - **Descripción**: Elimina un recurso existente en el servidor.
   - **Ejemplo**: Cuando eliminas una publicación o archivo de un servicio web.
   - **Analogía**: Es como tirar un archivo a la papelera de reciclaje.
6. **HEAD**:
   - **Descripción**: Solicita el encabezado de una página sin obtener el cuerpo del contenido. Es útil para verificar si un recurso está disponible o para obtener metadatos.
   - **Ejemplo**: Un buscador web podría usar una petición HEAD para comprobar si una página ha sido actualizada sin descargar todo el contenido.
   - **Analogía**: Es como mirar la portada de un libro para ver si es nuevo, sin abrirlo para leerlo.
7. **OPTIONS**:
   - **Descripción**: Solicita información sobre las opciones de comunicación permitidas para un recurso, como los métodos HTTP soportados por el servidor.
   - **Ejemplo**: Un servidor podría responder a una solicitud OPTIONS indicando que soporta GET, POST y DELETE, pero no PUT o PATCH.
   - **Analogía**: Es como preguntar a un restaurante qué platos tienen disponibles antes de hacer un pedido.
8. **CONNECT**:
   - **Descripción**: Se utiliza para crear un túnel de comunicación entre el cliente y el servidor. Se suele usar con HTTPS para establecer conexiones seguras.
   - **Ejemplo**: Un navegador puede usar CONNECT al conectarse a un servidor web seguro mediante SSL.
   - **Analogía**: Es como crear una línea directa de comunicación segura entre dos personas.
9. **TRACE**:
   - **Descripción**: Sirve para probar y diagnosticar el camino que sigue una solicitud hasta el servidor, reflejando el contenido de la solicitud en la respuesta.
   - **Ejemplo**: Es poco común y se usa principalmente para depuración.
   - **Analogía**: Es como un eco que devuelve la misma información que enviaste, solo para comprobar que llegó correctamente.

4. Middleware

Es una función intermedia que se ejecuta durante el ciclo de vida de una solicitud HTTP, antes de que la respuesta final llegue al cliente. Su propósito es procesar la solicitud de alguna manera antes de pasarla a la siguiente capa, que puede ser otro middleware o el manejador final que genera la respuesta.

Analogía

Imagina que tienes una fábrica de coches. El coche (solicitud) pasa por diferentes estaciones (middlewares), donde se le realizan ajustes o inspecciones antes de ser entregado al cliente (respuesta). Cada estación puede:

- Modificar el coche (agregar o quitar datos a la solicitud).
- Realizar una verificación (autenticación, validaciones).
- Decidir si el coche sigue adelante o no (permitir o bloquear la solicitud).

Ejemplo en Express

En Express, puedes tener middleware para diferentes propósitos, como:

- Autenticación: Verificar si el usuario tiene permiso para acceder.
- Procesar datos: Analizar el cuerpo de una solicitud (ej. JSON, formularios).
- Registro de logs: Guardar información sobre la solicitud (quién, cuándo, qué).

5. Gestor de dependencias:

Definición: Es una herramienta que facilita la instalación, actualización y manejo de las bibliotecas externas (dependencias) que tu proyecto necesita.

Analogía: Imagina que tienes una cocina (tu proyecto) y necesitas diferentes ingredientes (dependencias) para preparar un platillo. El gestor de dependencias sería el supermercado donde vas a buscar esos ingredientes, siempre con versiones actualizadas y listas para usar.

Tipos de gestores de dependencias en Node.js:
Los más comunes son:

- **npm (Node Package Manager)**: El gestor oficial de Node.js, es como un supermercado central.
- **Yarn**: Una alternativa más rápida en algunos casos, que te ayuda a instalar dependencias como en un supermercado express, optimizando el tiempo de compra.
- **pnpm**: Un gestor que optimiza el uso de espacio al compartir paquetes comunes entre proyectos, como si estuvieras usando una despensa compartida.

6. ORM (Object-Relational Mapping):

Definición: Es una técnica para interactuar con bases de datos relacionales desde el código, mapeando las tablas a objetos en el lenguaje de programación.

Analogía: Imagina que tu base de datos es una gran hoja de cálculo (tablas), y un ORM es una herramienta que convierte esa información en objetos que puedes manipular fácilmente en tu código, como si transformaras celdas en pequeños autos con los que puedes interactuar en una ciudad (tu aplicación).

Tipos de ORM en JavaScript:

- **Sequelize**: Un ORM que trabaja con varias bases de datos como PostgreSQL, MySQL, SQLite, etc. Es como un traductor que te permite comunicarte en varios lenguajes (bases de datos) usando el mismo idioma (JavaScript).
- **TypeORM**: Similar a Sequelize, pero está más enfocado en el uso de TypeScript, con características avanzadas para aplicaciones complejas. Es como un traductor que también revisa la gramática mientras traduce.
- **Mongoose**: Un ORM específicamente diseñado para bases de datos no relacionales como MongoDB. Es como un traductor especializado en un solo idioma, pero que funciona muy bien en su campo.

7. Creacion de Apirest

 Configuración inicial:

- Crea un nuevo directorio para tu proyecto y navega hacia él.
- Inicializa un nuevo proyecto Node.js con npm init -y.
- Instala las dependencias necesarias:

```
npm install express mysql2 dotenv joi helmet express-async-errors
npm install --save-dev nodemon
```

 Estructura del proyecto: Organiza tu proyecto según la estructura de directorios mostrada en el comentario al inicio del código.
 Configuración de la base de datos (src/config/database.js):

- Crea un archivo para la configuración de la conexión a MySQL.
- Utiliza un pool de conexiones para una mejor gestión de recursos.

 Modelo de Producto (src/models/producto.js):

- Implementa una clase Producto con métodos estáticos para interactuar con la base de datos.
- Incluye métodos para CRUD (Crear, Leer, Actualizar, Eliminar).

 Controlador de Producto (src/controllers/productoController.js):

- Crea funciones para manejar las operaciones CRUD.
- Utiliza el modelo Producto para interactuar con la base de datos.

 Middleware de validación (src/middleware/validate.js):

- Usa Joi para validar los datos de entrada.
- Crea un esquema para validar los productos.

 Middleware de manejo de errores (src/middleware/errorHandler.js):

- Implementa un manejador de errores centralizado.

 Rutas de Producto (src/routes/productoRoutes.js):

- Define las rutas para las operaciones CRUD.
- Asocia las rutas con los controladores correspondientes.

 Aplicación principal (src/app.js):

- Configura la aplicación Express.
- Incluye middleware para parsing JSON.
- Agrega las rutas de producto.
- Implementa el manejador de errores.

 Servidor (server.js):

- Configura el archivo principal para iniciar el servidor.
- Usa dotenv para cargar variables de entorno.

 Variables de entorno (.env):

    - Crea un archivo .env para almacenar configuraciones sensibles.

Para ejecutar la API:

1. Asegúrate de tener MySQL instalado y corriendo.
2. Crea la base de datos y la tabla de productos:

```
CREATE DATABASE ejemploproducto;
USE ejemploproducto;
CREATE TABLE productos (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(255) NOT NULL,
  precio DECIMAL(10, 2) NOT NULL,
  descripcion TEXT
);
```

3. Configura las variables de entorno en el archivo .env.
4. Ejecuta npm run dev para iniciar el servidor en modo de desarrollo.

Estructura del proyecto:

```
// Estructura del proyecto
/*
producto-api/
  ├── src/
  │   ├── config/
  │   │   └── database.js
  │   ├── controllers/
  │   │   └── productoController.js
  │   ├── middleware/
  │   │   ├── errorHandler.js
  │   │   └── validate.js
  │   ├── models/
  │   │   └── producto.js
  │   ├── routes/
  │   │   └── productoRoutes.js
  │   └── app.js
  ├── .env
  ├── package.json
  └── server.js
*/
```

```
// Actualización del package.json
{
  "name": "producto-api",
  "version": "1.0.0",
  "description": "API REST de Producto con seguridad mejorada",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mysql2": "^3.6.5",
    "dotenv": "^16.3.1",
    "joi": "^17.11.0",
    "express-async-errors": "^3.1.1",
    "helmet": "^7.1.0"
  },
```

```
  "devDependencies": {
    "nodemon": "^3.0.2"
  }
}

// src/app.js
const express = require('express');
require('express-async-errors');
const helmet = require('helmet');
const productoRoutes = require('./routes/productoRoutes');
const errorHandler = require('./middleware/errorHandler');

const app = express();

// Aplicar Helmet para mejorar la seguridad
app.use(helmet());

// Configuraciones específicas de Helmet (opcional)
app.use(helmet.contentSecurityPolicy({
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'", "'unsafe-inline'"],
    styleSrc: ["'self'", "'unsafe-inline'"],
    imgSrc: ["'self'", "data:", "https:"],
  },
}));

app.use(helmet.referrerPolicy({ policy: 'strict-origin-when-cross-
origin' }));

app.use(express.json());

app.use('/api/productos', productoRoutes);

app.use(errorHandler);

module.exports = app;
```

```
// .env
PORT=3000
DB_HOST=localhost
DB_USER=tu_usuario
DB_PASSWORD=tu_contraseña
DB_NAME=tu_base_de_datos

// src/config/database.js
const mysql = require('mysql2/promise');
require('dotenv').config();

const pool = mysql.createPool({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});

module.exports = pool;

// src/models/producto.js
const pool = require('../config/database');

class Producto {
  static async getAll() {
    const [rows] = await pool.query('SELECT * FROM productos');
    return rows;
  }

  static async getById(id) {
    const [rows] = await pool.query('SELECT * FROM productos WHERE id
= ?', [id]);
    return rows[0];
  }
```

```
  static async create(producto) {
    const { nombre, precio, descripcion } = producto;
    const [result] = await pool.query(
      'INSERT INTO productos (nombre, precio, descripcion) VALUES (?,
?, ?)',
      [nombre, precio, descripcion]
    );
    return result.insertId;
  }

  static async update(id, producto) {
    const { nombre, precio, descripcion } = producto;
    const [result] = await pool.query(
      'UPDATE productos SET nombre = ?, precio = ?, descripcion = ?
WHERE id = ?',
      [nombre, precio, descripcion, id]
    );
    return result.affectedRows;
  }

  static async delete(id) {
    const [result] = await pool.query('DELETE FROM productos WHERE id
= ?', [id]);
    return result.affectedRows;
  }
}

module.exports = Producto;

// src/controllers/productoController.js
const Producto = require('../models/producto');

exports.getAllProductos = async (req, res) => {
  const productos = await Producto.getAll();
  res.json(productos);
};

exports.getProductoById = async (req, res) => {
  const producto = await Producto.getById(req.params.id);
```

```
  if (producto) {
    res.json(producto);
  } else {
    res.status(404).json({ message: 'Producto no encontrado' });
  }
};

exports.createProducto = async (req, res) => {
  const id = await Producto.create(req.body);
  res.status(201).json({ id, ...req.body });
};

exports.updateProducto = async (req, res) => {
  const affectedRows = await Producto.update(req.params.id,
req.body);
  if (affectedRows) {
    res.json({ id: req.params.id, ...req.body });
  } else {
    res.status(404).json({ message: 'Producto no encontrado' });
  }
};

exports.deleteProducto = async (req, res) => {
  const affectedRows = await Producto.delete(req.params.id);
  if (affectedRows) {
    res.status(204).end();
  } else {
    res.status(404).json({ message: 'Producto no encontrado' });
  }
};

// src/middleware/validate.js
const Joi = require('joi');

const productoSchema = Joi.object({
  nombre: Joi.string().required(),
  precio: Joi.number().positive().required(),
  descripcion: Joi.string().required()
});
```

```
exports.validateProducto = (req, res, next) => {
  const { error } = productoSchema.validate(req.body);
  if (error) {
    return res.status(400).json({ message: error.details[0].message
});
  }
  next();
};

// src/middleware/errorHandler.js
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Algo salió mal!' });
};

module.exports = errorHandler;

// src/routes/productoRoutes.js
const express = require('express');
const router = express.Router();
const productoController =
require('../controllers/productoController');
const { validateProducto } = require('../middleware/validate');

router.get('/', productoController.getAllProductos);
router.get('/:id', productoController.getProductoById);
router.post('/', validateProducto,
productoController.createProducto);
router.put('/:id', validateProducto,
productoController.updateProducto);
router.delete('/:id', productoController.deleteProducto);

module.exports = router;

// server.js
require('dotenv').config();
const app = require('./src/app');
```

```
const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Servidor corriendo en el puerto ${PORT}`);
});
```

8. Probar apiRest postman o extensión Thunder client

Paso 1: Iniciar Postman

- Abre la aplicación Postman en tu computadora.

Paso 2: Crear una nueva solicitud

- Haz clic en el botón "+" para crear una nueva pestaña de solicitud.

Paso 3: Configurar la solicitud para obtener todos los productos (GET)

1. Selecciona el método HTTP "GET" en el menú desplegable.
2. Ingresa la URL: http://localhost:3000/api/productos (ajusta el puerto si es diferente).
3. Haz clic en "Send" para enviar la solicitud.
4. Deberías ver la lista de productos en el panel de respuesta.

Paso 4: Obtener un producto específico (GET)

1. Crea una nueva pestaña de solicitud.
2. Selecciona el método HTTP "GET".
3. Ingresa la URL: http://localhost:3000/api/productos/1 (reemplaza "1" con un ID válido).
4. Envía la solicitud y verifica la respuesta.

Paso 5: Crear un nuevo producto (POST)

1. Nueva pestaña, selecciona método "POST".
2. URL: http://localhost:3000/api/productos
3. Ve a la pestaña "Body".
4. Selecciona "raw" y luego "JSON" en el menú desplegable.
5. Ingresa los datos del producto en formato JSON:

```
{
  "nombre": "Nuevo Producto",
  "precio": 29.99,
  "descripcion": "Descripción del nuevo producto"
}
```

6. Envía la solicitud y verifica la respuesta.

Paso 6: Actualizar un producto (PUT)

1. Nueva pestaña, método "PUT".
2. URL: http://localhost:3000/api/productos/1 (usa un ID existente).
3. En la pestaña "Body", selecciona "raw" y "JSON".
4. Ingresa los datos actualizados:

```
{
  "nombre": "Producto Actualizado",
  "precio": 39.99,
  "descripcion": "Descripción actualizada"
}
```

5. Envía la solicitud y verifica la respuesta.

Paso 7: Eliminar un producto (DELETE)

1. Nueva pestaña, método "DELETE".
2. URL: http://localhost:3000/api/productos/1 (usa un ID existente).
3. Envía la solicitud y verifica la respuesta.

Consejos adicionales:

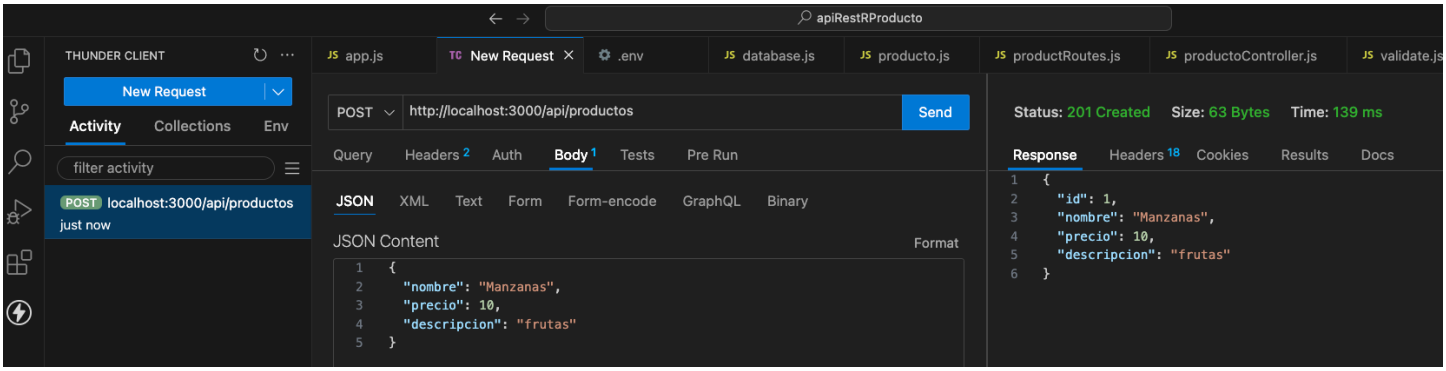
1. Verifica los encabezados de respuesta para información adicional.
2. Usa la pestaña "Headers" para agregar encabezados personalizados si es necesario (por ejemplo, tokens de autenticación).
3. Guarda tus solicitudes en una colección de Postman para futuras pruebas.
4. Utiliza variables de entorno en Postman para manejar diferentes URLs (desarrollo, producción, etc.).

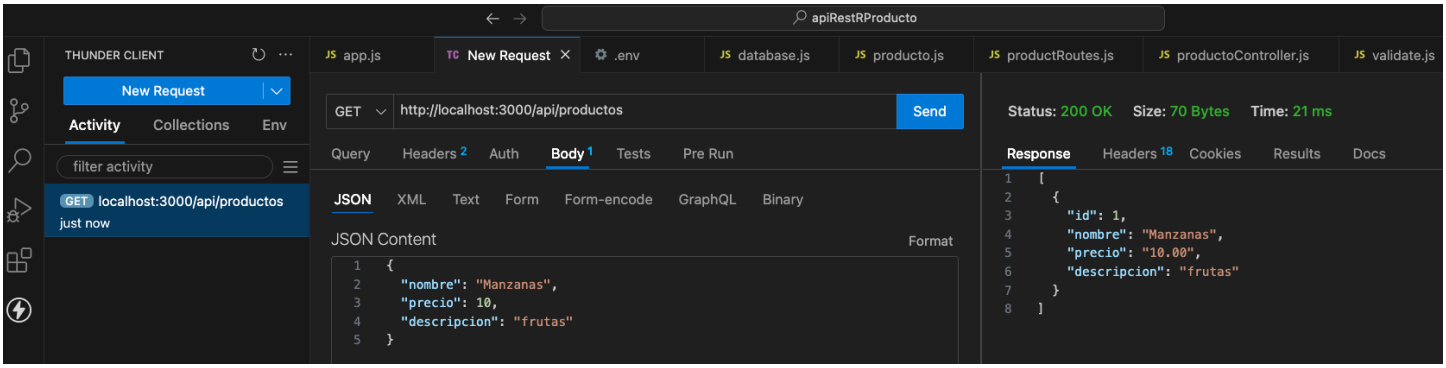
Prueba de error:

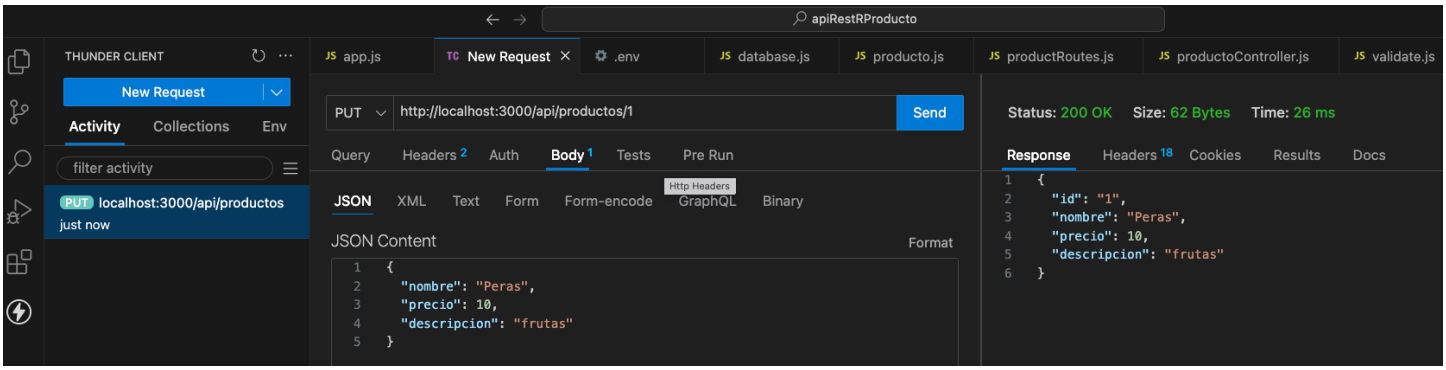
1. Intenta crear un producto con datos inválidos (por ejemplo, omite un campo requerido).
2. Verifica que obtienes una respuesta de error apropiada.

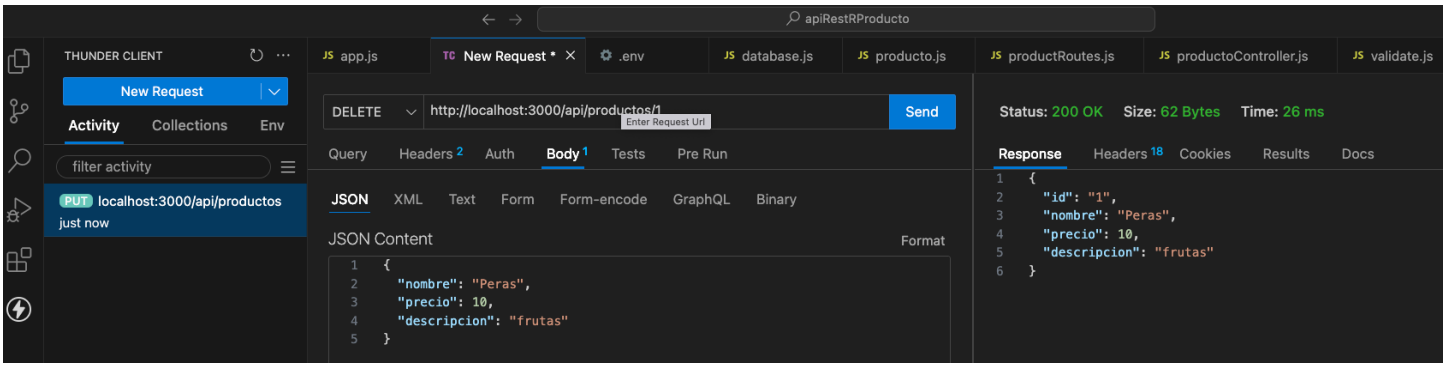
Recuerda ajustar las URLs y los puertos según tu configuración específica. Si encuentras algún problema o error durante las pruebas, revisa los logs de tu servidor Node.js para obtener más información sobre lo que podría estar fallando.

Ver:




apiRestRProducto
THUNDER CLIENT
New Request
Activity
Collections
Env
filter activity
GET localhost:3000/api/productos
just now
app.js
New Request
.env
database.js
producto.js
productRoutes.js
productoController.js
validate.js
GET
http://localhost:3000/api/productos
Send
Status: 200 OK
Size: 70 Bytes
Time: 21 ms
Query
Headers 2
Auth
Body 1
Tests
Pre Run
Response
Headers 18
Cookies
Results
Docs
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
JSON Content
Format
{
"nombre": "Manzanas",
"precio": 10,
"descripcion": "frutas"
}
[
{
"id": 1,
"nombre": "Manzanas",
"precio": "10.00",
"descripcion": "frutas"
}
]


apiRestRProducto
THUNDER CLIENT
New Request
Activity
Collections
Env
filter activity
PUT localhost:3000/api/productos
just now
app.js
New Request
.env
database.js
producto.js
productRoutes.js
productoController.js
validate.js
PUT
http://localhost:3000/api/productos/1
Send
Status: 200 OK
Size: 62 Bytes
Time: 26 ms
Query
Headers 2
Auth
Body 1
Tests
Pre Run
Http Headers
Response
Headers 18
Cookies
Results
Docs
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
JSON Content
Format
{
"nombre": "Peras",
"precio": 10,
"descripcion": "frutas"
}


apiRestRProducto
THUNDER CLIENT
New Request
Activity
Collections
Env
filter activity
PUT localhost:3000/api/productos
just now
app.js
New Request *
.env
database.js
producto.js
productRoutes.js
productoController.js
validate.js
DELETE
http://localhost:3000/api/productos/1
Enter Request Url
Send
Status: 200 OK
Size: 62 Bytes
Time: 26 ms
Query
Headers 2
Auth
Body 1
Tests
Pre Run
Response
Headers 18
Cookies
Results
Docs
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
JSON Content
Format
{
"nombre": "Peras",
"precio": 10,
"descripcion": "frutas"
}