2 Corte. Parcial 2

Cristhian Andrés Burbano Mendoza

Ingeniería de sistemas, Corporación Universitaria Minuto de Dios

60747: Bases de Datos Masivas

Prof. William Alexander Matallana Porras

25 de abril de 2025

Tabla de contenido

Introducción

En la actualidad, la gestión eficiente de la información en cadenas de comida rápida es fundamental para optimizar procesos como el control de inventarios, la administración de empleados y el seguimiento de ventas. Este proyecto consiste en el desarrollo de una API REST utilizando Express.js y PostgreSQL, diseñada para manejar datos relacionados con restaurantes, empleados, productos, pedidos y ventas. La API está documentada y probada mediante Postman, garantizando su funcionalidad y robustez.

El sistema permite realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) sobre cada entidad, así como consultas específicas para análisis de ventas, productos más vendidos y gestión de empleados. Además, se implementó una conexión con Supabase como servicio de base de datos PostgreSQL en la nube, asegurando escalabilidad y disponibilidad.

Objetivo General:

Desarrollar una API REST con Express.js y PostgreSQL que permita gestionar de manera eficiente la información de una cadena de comidas rápidas, incluyendo restaurantes, empleados, productos, pedidos y ventas, garantizando operaciones seguras, documentadas y probadas.

Objetivos Específicos:

Configurar la base de datos:

- Diseñar e implementar el modelo de datos PostgresSQL (Supabase) con las tablas necesarias (Restaurante, Empleado, Producto, Pedido, DetallePedido) y sus respectivas relaciones.

Desarrollar los endpoints de la API:

- Implementar rutas y controladores en Express.js para cada entidad, permitiendo operaciones CRUD.
- Crear consultas personalizadas para obtener información específica, como productos más vendidos, ventas por restaurante y pedidos por fecha.

Documentar y probar la API:

- Documentar todas las rutas, métodos y parámetros de la API.
- Realizar pruebas exhaustivas utilizando Postman, verificando el correcto funcionamiento de cada endpoint.

Entregar el proyecto:

- Proporcionar capturas de pantalla de las solicitudes y respuestas en Postman.
- Exportar la colección de Postman en formato .json.
- Incluir el modelo de datos generado en Supabase.

Desarrollar una API REST con Express.js y PostgreSQL para gestionar la información de una cadena de comidas rápidas: restaurantes, empleados, productos, pedidos y ventas. Documentar la API y realizar pruebas con Postman.

Modelo de datos

1. ***Restaurante***

Campo	Tipo de Dato	Clave	Descripción
id_rest	INT	PK	Identificador del restaurante
nombre	VARCHAR(100)		Nombre del restaurante
ciudad	VARCHAR(100)		Ciudad donde se ubica
direccion	VARCHAR(150)		Dirección del restaurante
fecha_apertura	DATE		Fecha de apertura

2. ***Empleado***

Campo	Tipo de Dato	Clave	Descripción
id_empleado	INT	PK	Identificador del empleado
nombre	VARCHAR(100)		Nombre del empleado
rol	VARCHAR(50)		Cargo (cocinero, cajero, etc.)
id_rest	INT	FK	Restaurante al que pertenece

3. ***Producto***

Campo	Tipo de Dato	Clave	Descripción
id_prod	INT	PK	Identificador del producto
nombre	VARCHAR(100)		Nombre del producto
precio	NUMERIC(10,2)		Precio del producto

4. *Pedido*

Campo	Tipo de Dato	Clave	Descripción
id_pedido	INT	PK	Identificador del pedido
fecha	DATE		Fecha del pedido
id_rest	INT	FK	Restaurante donde se hizo
total	NUMERIC(10,2)		Monto total del pedido

5. *DetallePedido*

Campo	Tipo de Dato	Clave	Descripción
id_detalle	INT	PK	Identificador del detalle
id_pedido	INT	FK	Pedido relacionado
id_prod	INT	FK	Producto incluido en el pedido
cantidad	INT		Cantidad solicitada
subtotal	NUMERIC(10,2)		Subtotal calculado por ese producto

Relaciones

- Un restaurante tiene muchos empleados y pedidos.
- Un pedido puede tener muchos productos (a través de DetallePedido).
- Un producto puede estar en muchos pedidos.

Actividades el proyecto

1. Conexión a la base de datos PostgresSQL (Supabase)
2. Crear rutas y controladores con Express para cada entidad (CRUD)
3. Implementar las siguientes consultas nativas y exponerlas por rutas:
    - Obtener todos los productos de un pedido especifico.
    - Obtener los productos más vendidos (más de X unidades).
    - Obtener el total de ventas por restaurante.
    - Obtener los pedidos realizados con una fecha específica.
    - Obtener los empleados por rol en un restaurante.

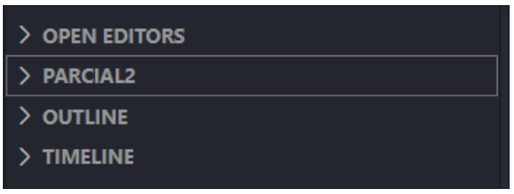
Requisitos para la entrega

- Entregar capturas de pantalla mostrando la solicitud con su método, parámetros y resultado.
- Exportar la colección de Postman en formato .json
- Entregar modelo de datos de Supabase
- Documentar las consultas nativas con las respectivas peticiones en Postman

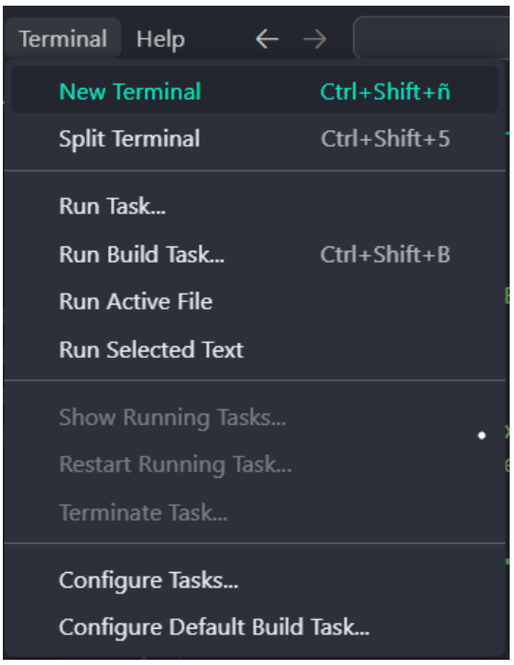
Inicio del Parcial

Creo una nueva carpeta llamada Parcial2

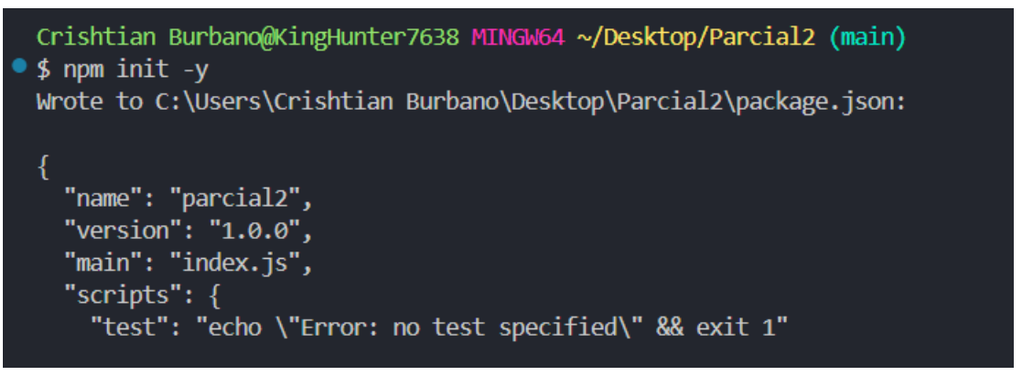
Ahora en visual Studio Code abro la carpeta creada en el escritorio que tiene como nombre Parcial2

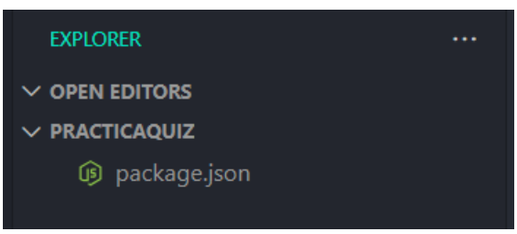


Una vez la carpeta está en visual studio code, abro una terminal nueva (New Terminal)

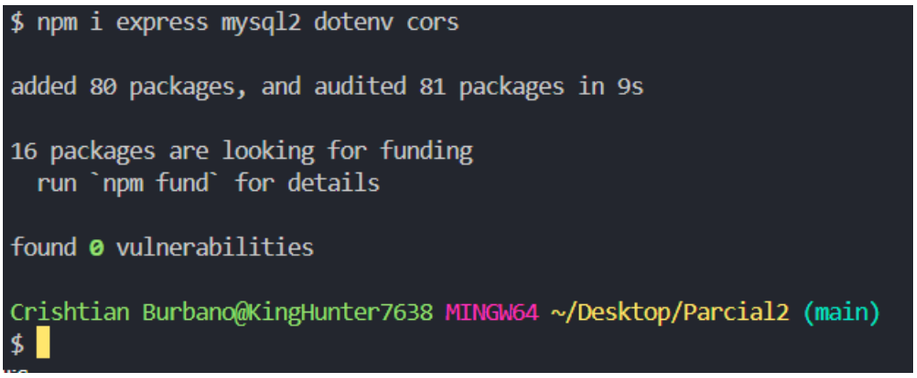


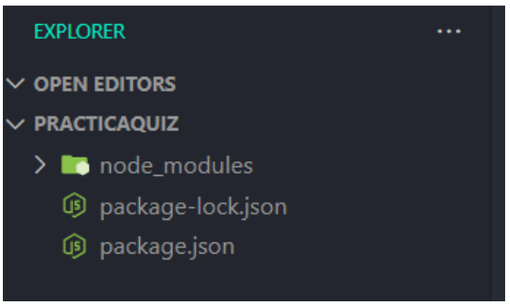
Una vez la abierta la termina tengo que descargar los paquetes .json, para eso uso el comando (npm init -y), que me creo un paquete



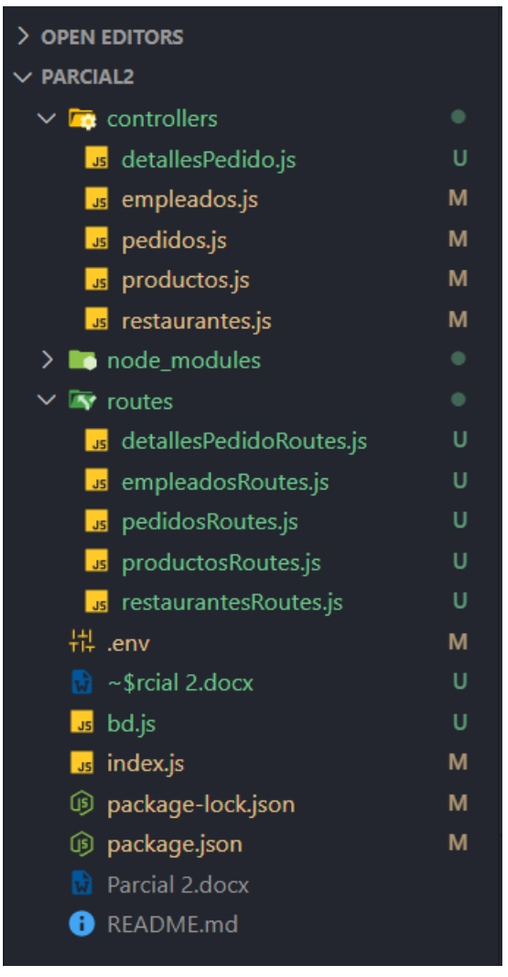


Ahora uso el comando (npm i express mysql2 dotenv cors) para descargar el siguiente paquete mysql2

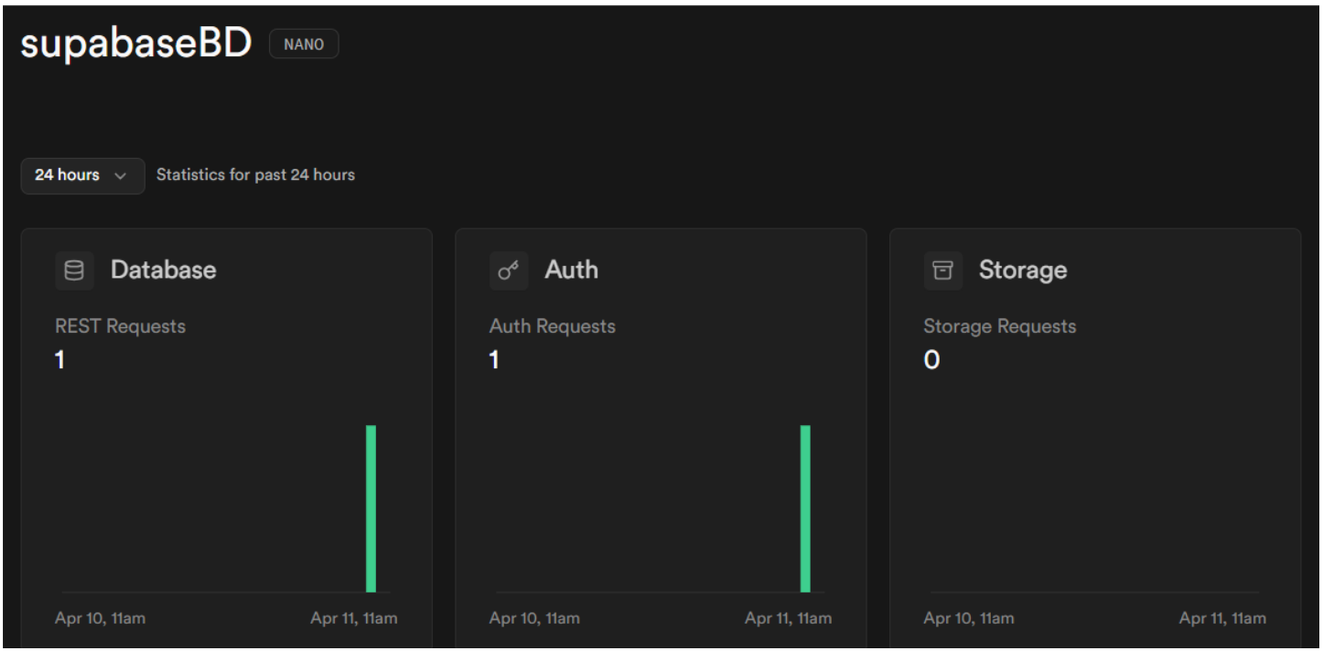




Una vez descargado los paquetes creo los Files nuevos, uno con el nombre de (bd.js) para la conexión con Supabase y los otros con nombres descriptivos para cada tabla con sus API's, además del principal que es el (index.js)



Para la conexión a la base de datos voy a usar Supabase donde tengo un proyecto ya creado llamado (supabaseBD)



Para la conexión de visual con mi proyecto de supabase uso el conecct y uso la conexión (Session pooler) postgresql://postgres.jqhnvjffprivqlfatjej:[YOUR-PASSWORD]@aws-0-us-east-1.pooler.supabase.com:5432/postgres

Conexión (bd.js)

DB_HOST=aws-0-us-east-1.pooler.supabase.com

DB_PORT=5432

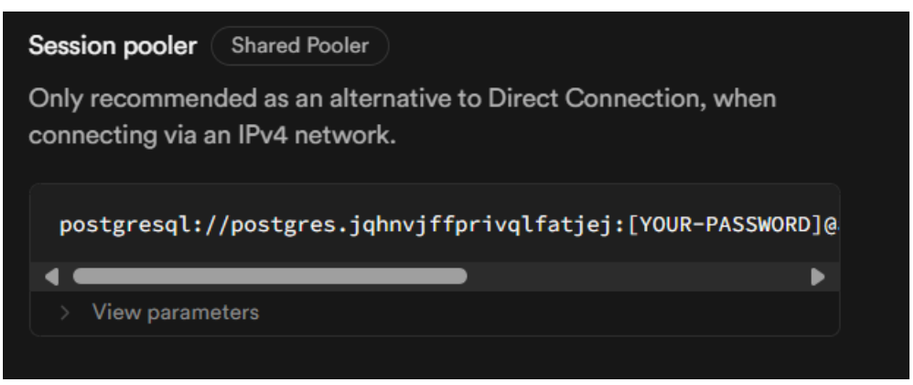
DB_USER=postgres.jqhnvjffprivqlfatjej

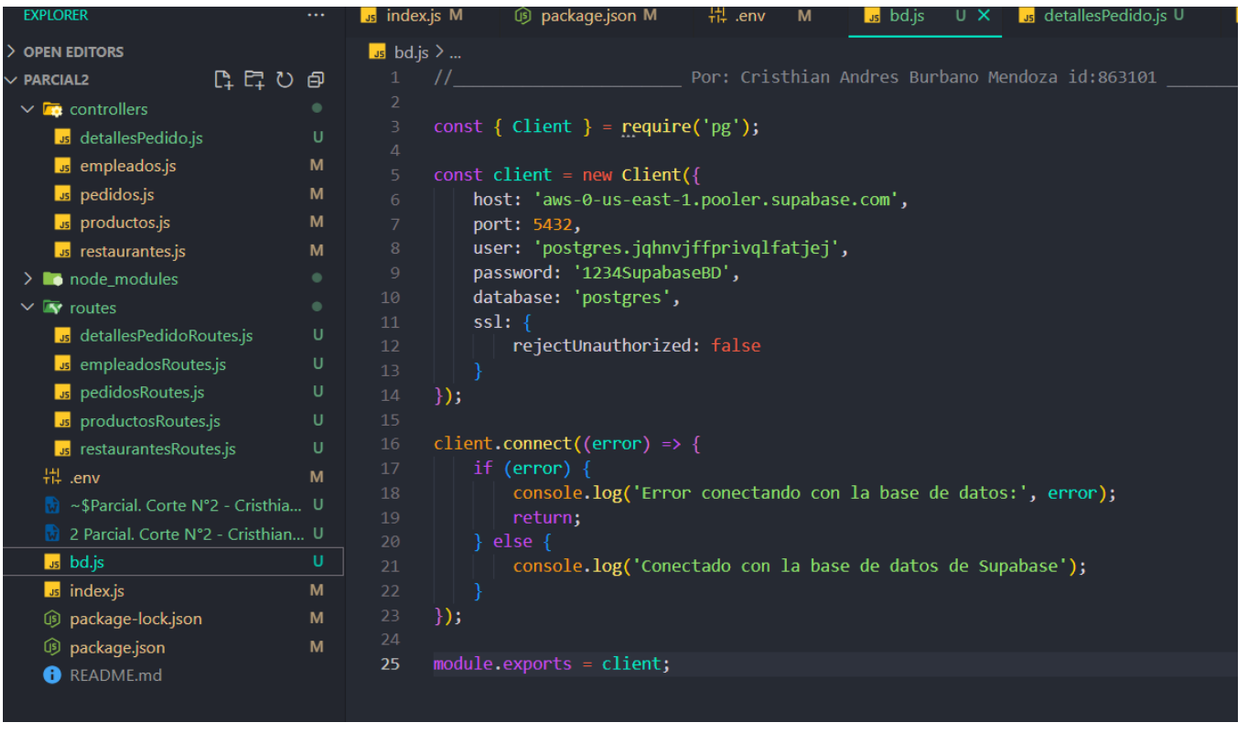
DB_PASSWORD=1234SupabaseBD

DB_DATABASE=postgres

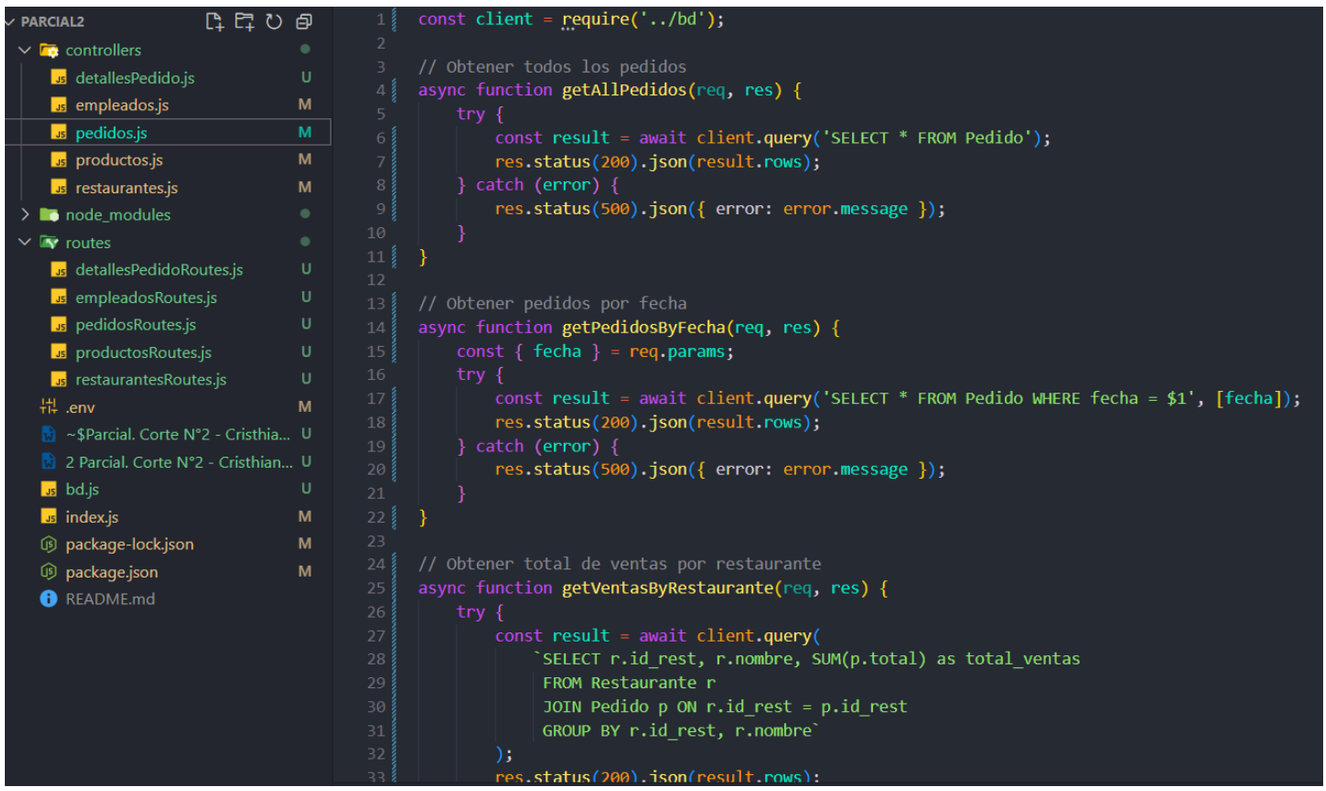
PORT=3000

Esas son las credenciales que uso para conectarme al servidor





Creación de las API's. De manera individual cada una en su propio .json para tener un mejor orden dentro de la carpeta controllers


PARCIAL2
controllers
detallesPedido.js U
empleados.js M
pedidos.js M
productos.js M
restaurantes.js M
node_modules
routes
detallesPedidoRoutes.js U
empleadosRoutes.js U
pedidosRoutes.js U
productosRoutes.js U
restaurantesRoutes.js U
.env M
~$Parcial. Corte N°2 - Cristhia... U
2 Parcial. Corte N°2 - Cristhian... U
bd.js U
index.js M
package-lock.json M
package.json M
README.md
const client = require('../bd');
// Obtener todos los pedidos
async function getAllPedidos(req, res) {
try {
const result = await client.query('SELECT * FROM Pedido');
res.status(200).json(result.rows);
} catch (error) {
res.status(500).json({ error: error.message });
}
}
// Obtener pedidos por fecha
async function getPedidosByFecha(req, res) {
const { fecha } = req.params;
try {
const result = await client.query('SELECT * FROM Pedido WHERE fecha = $1', [fecha]);
res.status(200).json(result.rows);
} catch (error) {
res.status(500).json({ error: error.message });
}
}
// Obtener total de ventas por restaurante
async function getVentasByRestaurante(req, res) {
try {
const result = await client.query(
`SELECT r.id_rest, r.nombre, SUM(p.total) as total_ventas
FROM Restaurante r
JOIN Pedido p ON r.id_rest = p.id_rest
GROUP BY r.id_rest, r.nombre`
);
res.status(200).json(result.rows);


OPEN EDITORS
PARCIAL2
controllers
detallesPedido.js U
empleados.js M
pedidos.js M
productos.js M
restaurantes.js M
node_modules
routes
detallesPedidoRoutes.js U
empleadosRoutes.js U
pedidosRoutes.js U
productosRoutes.js U
restaurantesRoutes.js U
.env M
~$Parcial. Corte N°2 - Cristhia... U
2 Parcial. Corte N°2 - Cristhian... U
bd.js U
index.js M
package-lock.json M
package.json M
README.md
controllers > empleados.js > ...
async function updateEmpleado(req, res) {
} else {
res.status(200).json(result.rows[0]);
}
} catch (error) {
res.status(500).json({ error: error.message });
}
}
// Eliminar un empleado
async function deleteEmpleado(req, res) {
const { id } = req.params;
try {
const result = await client.query('DELETE FROM Empleado WHERE id_empleado = $1 RETURNING *', [id]);
if (result.rowCount === 0) {
res.status(404).json({ message: 'Empleado no encontrado' });
} else {
res.status(200).json({ message: 'Empleado eliminado' });
}
} catch (error) {
res.status(500).json({ error: error.message });
}
}
module.exports = {
getAllEmpleados,
getEmpleadosByRestaurante,
getEmpleadosByRol,
createEmpleado,
updateEmpleado,
deleteEmpleado
};


PARCIAL2
controllers
detallesPedido.js U
empleados.js M
pedidos.js M
productos.js M
restaurantes.js M
node_modules
routes
detallesPedidoRoutes.js U
empleadosRoutes.js U
pedidosRoutes.js U
productosRoutes.js U
restaurantesRoutes.js U
.env M
~$Parcial. Corte N°2 - Cristhia... U
2 Parcial. Corte N°2 - Cristhian... U
bd.js U
index.js M
package-lock.json M
package.json M
README.md
async function createDetallePedido(req, res) {
res.status(500).json({ error: error.message });
}
}
// Actualizar un detalle de pedido
async function updateDetallePedido(req, res) {
const { id } = req.params;
const { id_pedido, id_prod, cantidad, subtotal } = req.body;
try {
const result = await client.query(
'UPDATE DetallePedido SET id_pedido = $1, id_prod = $2, cantidad = $3, subtotal = $4 WHERE id_detalle = $5 RETURNING *',
[id_pedido, id_prod, cantidad, subtotal, id]
);
if (result.rowCount === 0) {
res.status(404).json({ message: 'Detalle de pedido no encontrado' });
} else {
res.status(200).json(result.rows[0]);
}
} catch (error) {
res.status(500).json({ error: error.message });
}
}
// Eliminar un detalle de pedido
async function deleteDetallePedido(req, res) {
const { id } = req.params;
try {
const result = await client.query('DELETE FROM DetallePedido WHERE id_detalle = $1 RETURNING *', [id]);
if (result.rowCount === 0) {
res.status(404).json({ message: 'Detalle de pedido no encontrado' });
} else {

Ahora para levantar el servicio abro una terminal nueva y siempre lo hago con este comando (node --watch index.js) para que se actualice a la vez sin necesidad de volverlo a escribir o pausar el servicio

```
Crishtian Burbano@KingHunter7638 MINGW64 ~/Desktop/Parcial2 (main)
$ node --watch index.js

Crishtian Burbano@KingHunter7638 MINGW64 ~/Desktop/Parcial2 (main)
$ node --watch index.js
$ node --watch index.js
Servidor corriendo en http://localhost:3000
Conectado con la base de datos de Supabase
Conectado con la base de datos de Supabase
```

Creación de tablas:

Ahora voy a crear todas las tablas, la primera tabla se llama (Restaurante), la segunda se llama (Empleado), la tercera se llama (Producto), la cuarta se llama (Pedido) y la quinta se llama (DetallePedido) para crear todas las tablas lo voy hacer desde la terminal SQL Editor de Supabase

Tabla Restaurante

```
CREATE TABLE Restaurante (
    id_rest SERIAL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    ciudad VARCHAR(100) NOT NULL,
    direccion VARCHAR(150) NOT NULL,
    fecha_apertura DATE NOT NULL
);
```

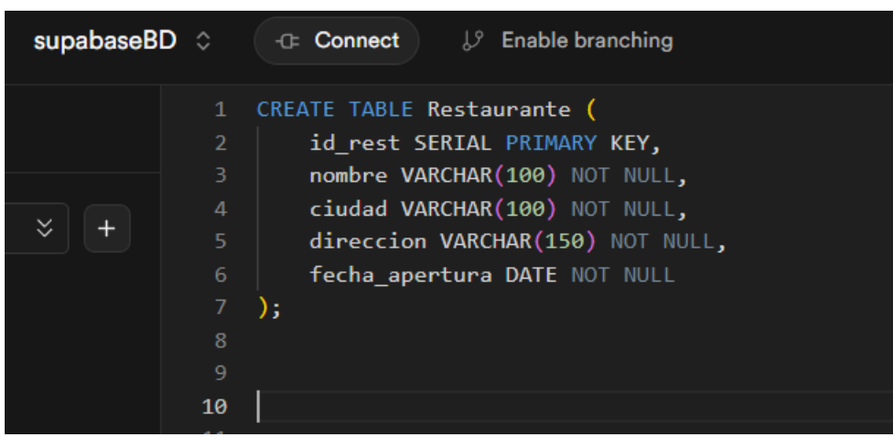


Tabla Empleado

```
CREATE TABLE Empleado (
    id_empleado SERIAL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    rol VARCHAR(50) NOT NULL,
    id_rest INT NOT NULL,
    FOREIGN KEY (id_rest) REFERENCES Restaurante(id_rest)
);
```

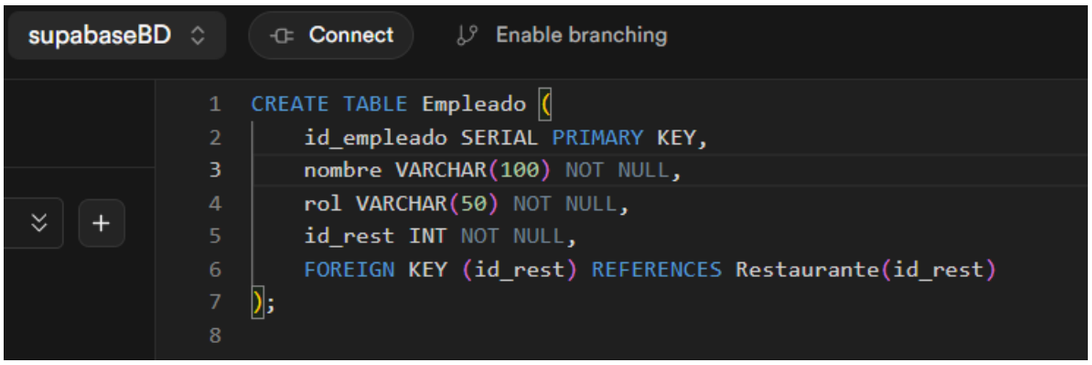


Tabla Producto

```
CREATE TABLE Producto (
    id_prod SERIAL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    precio NUMERIC(10,2) NOT NULL
);
```

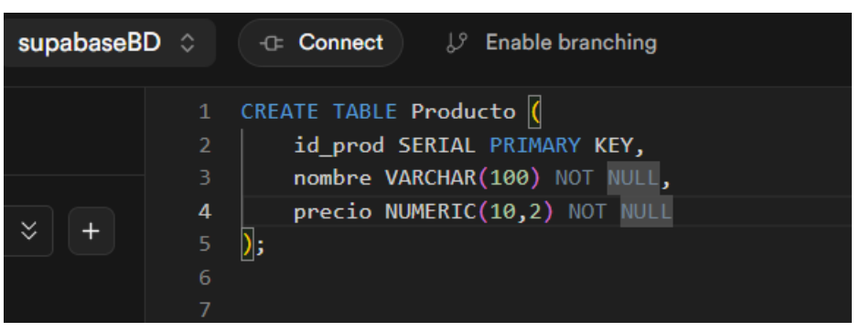


Tabla Pedido

```
CREATE TABLE Pedido (
    id_pedido SERIAL PRIMARY KEY,
    fecha DATE NOT NULL,
    id_rest INT NOT NULL,
    total NUMERIC(10,2) NOT NULL,
    FOREIGN KEY (id_rest) REFERENCES Restaurante(id_rest)
);
```

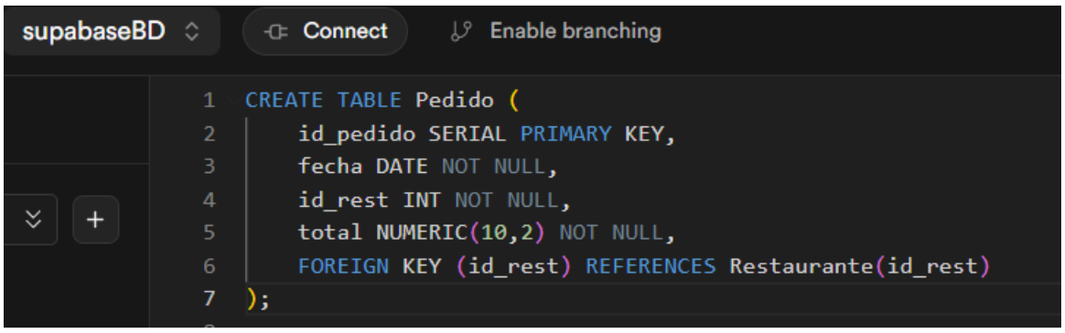
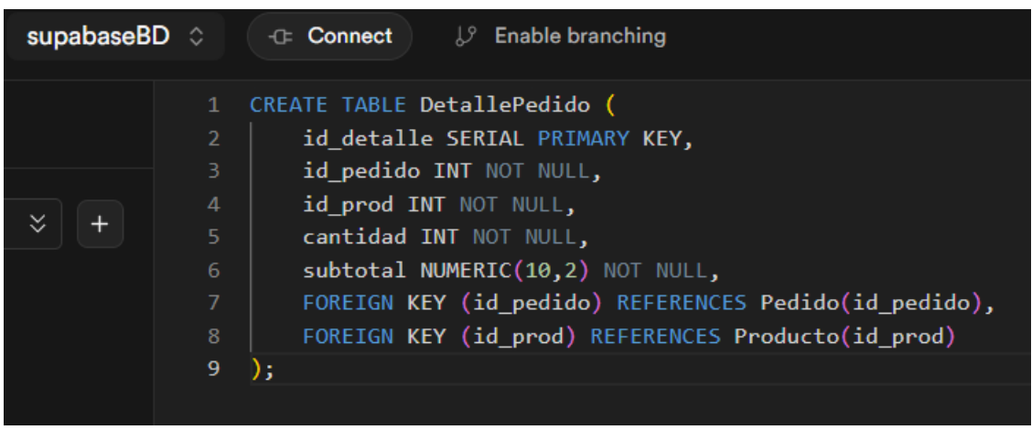
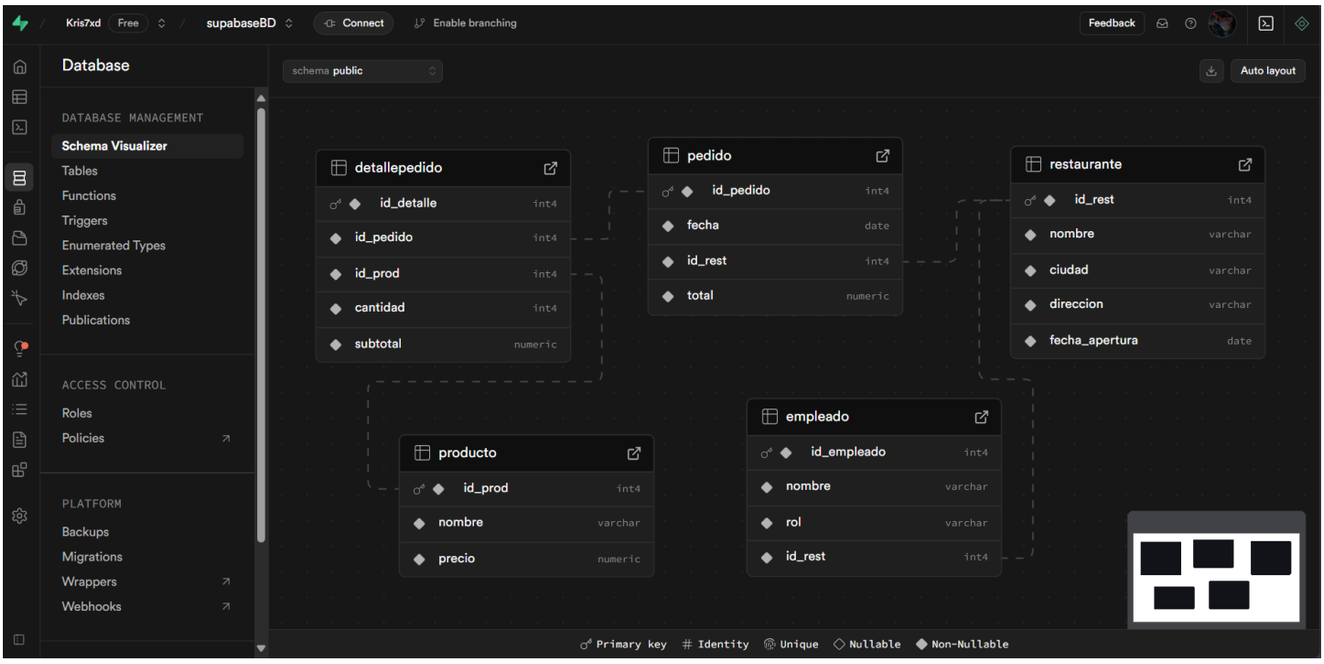


Tabla DetallePedido

```
CREATE TABLE DetallePedido (
    id_detalle SERIAL PRIMARY KEY,
    id_pedido INT NOT NULL,
    id_prod INT NOT NULL,
    cantidad INT NOT NULL,
    subtotal NUMERIC(10,2) NOT NULL,
    FOREIGN KEY (id_pedido) REFERENCES Pedido(id_pedido),
    FOREIGN KEY (id_prod) REFERENCES Producto(id_prod)
);
```

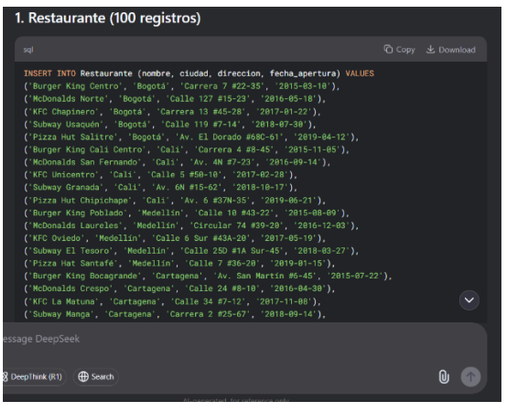


Para verificar que las tablas se crearon voy a (Database) y miro si están las tablas y como se muestra a continuación se crearon las tablas y bien relacionadas en Supabase

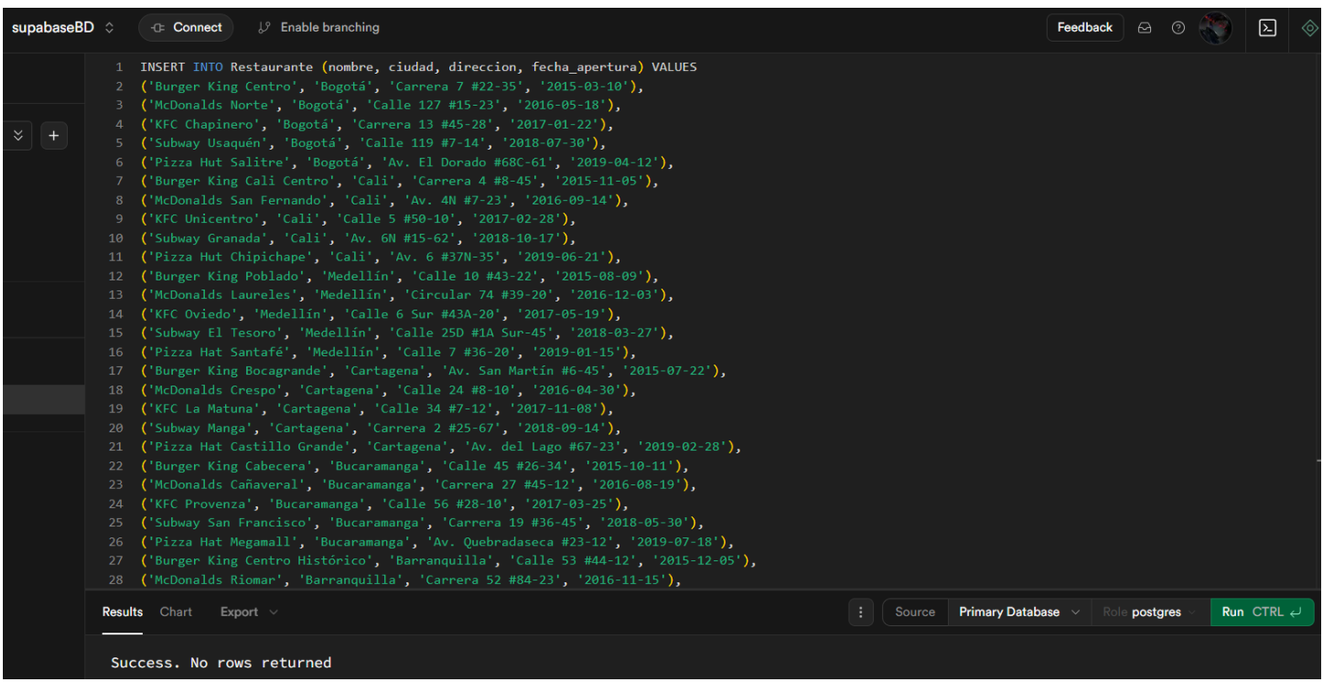


Insertar registros:

Le pedi a DeepSeek que me diera 100 registros para cada tabla y a continuacion los pongo:



Hecho con DeepSeek



2. Producto (100 registros)

```sql
INSERT INTO Producto (nombre, precio) VALUES
('Hamburguesa Clásica', 12000),
('Hamburguesa con Queso', 14000),
('Hamburguesa Doble', 16000),
('Hamburguesa Bacon', 18000),
('Hamburguesa Vegetariana', 15000),
('Hamburguesa Pollo', 13000),
('Hamburguesa BBQ', 17000),
('Hamburguesa Picante', 16000),
('Hamburguesa Mexicana', 19000),
('Hamburguesa Especial', 20000),
('Papas Pequeñas', 5000),
('Papas Medianas', 7000),
('Papas Grandes', 9000),
('Papas con Queso', 10000),
('Papas con Bacon', 11000),
('Aros de Cebolla', 8000),
('Nuggets de Pollo (6)', 10000),
('Nuggets de Pollo (12)', 18000),
```

Hecho con DeepSeek

supabaseBD — Connect — Enable branching — Feedback

```
1  INSERT INTO Producto (nombre, precio) VALUES
2  ('Hamburguesa Clásica', 12000),
3  ('Hamburguesa con Queso', 14000),
4  ('Hamburguesa Doble', 16000),
5  ('Hamburguesa Bacon', 18000),
6  ('Hamburguesa Vegetariana', 15000),
7  ('Hamburguesa Pollo', 13000),
8  ('Hamburguesa BBQ', 17000),
9  ('Hamburguesa Picante', 16000),
10 ('Hamburguesa Mexicana', 19000),
11 ('Hamburguesa Especial', 20000),
12 ('Papas Pequeñas', 5000),
13 ('Papas Medianas', 7000),
14 ('Papas Grandes', 9000),
15 ('Papas con Queso', 10000),
16 ('Papas con Bacon', 11000),
17 ('Aros de Cebolla', 8000),
18 ('Nuggets de Pollo (6)', 10000),
19 ('Nuggets de Pollo (12)', 18000),
20 ('Alitas de Pollo (6)', 15000),
21 ('Alitas de Pollo (12)', 28000),
22 ('Ensalada César', 12000),
23 ('Ensalada Griega', 13000),
24 ('Ensalada de Pollo', 14000),
25 ('Ensalada Vegetariana', 11000),
26 ('Sándwich de Pollo', 10000),
27 ('Sándwich de Pavo', 11000),
28 ('Sándwich Vegetal', 9000),
```

Results | Chart | Export — Source: Primary Database — Role: postgres — Run CTRL

Success. No rows returned

3. Empleado (100 registros)

```sql
INSERT INTO Empleado (nombre, rol, id_rest) VALUES
('Juan Pérez', 'Gerente', 1),
('María Gómez', 'Cajero', 1),
('Carlos López', 'Cocinero', 1),
('Ana Rodríguez', 'Mesero', 1),
('Pedro Martínez', 'Limpieza', 1),
('Laura Sánchez', 'Gerente', 2),
('Jorge Ramírez', 'Cajero', 2),
('Diana Torres', 'Cocinero', 2),
('Andrés Jiménez', 'Mesero', 2),
('Sofía Herrera', 'Limpieza', 2),
('Miguel González', 'Gerente', 3),
('Carmen Vargas', 'Cajero', 3),
('Ricardo Mendoza', 'Cocinero', 3),
('Patricia Castro', 'Mesero', 3),
('Fernando Rojas', 'Limpieza', 3),
('Alejandra Silva', 'Gerente', 4),
('Oscar Ortega', 'Cajero', 4),
('Lucía Medina', 'Cocinero', 4),
('Héctor Guerrero', 'Mesero', 4),
('Verónica Paredes', 'Limpieza', 4),
```

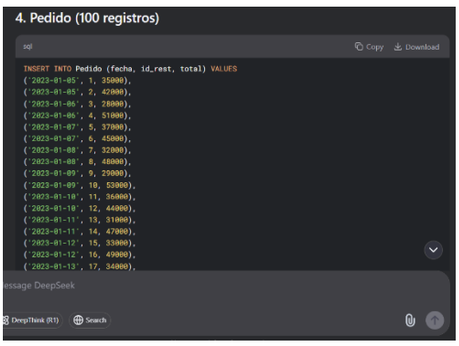
Hecho con DeepSeek

supabaseBD — Connect — Enable branching — Feedback

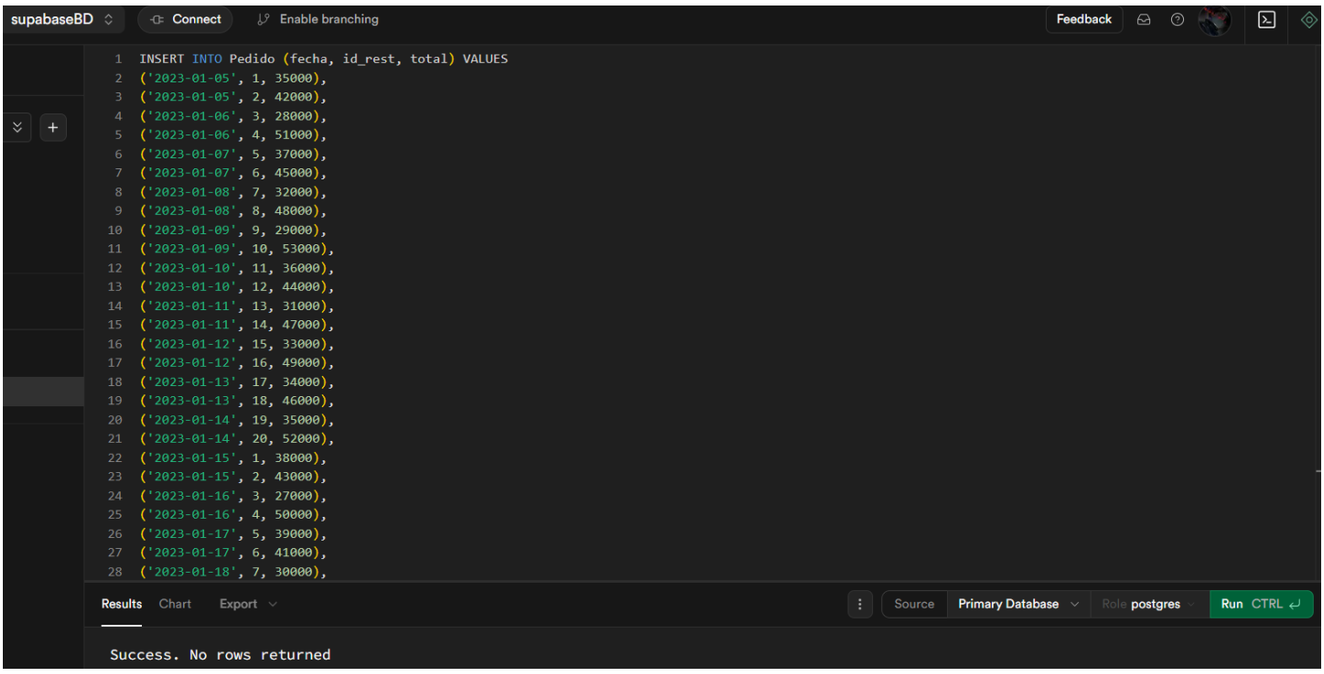
```
1  INSERT INTO Empleado (nombre, rol, id_rest) VALUES
2  ('Juan Pérez', 'Gerente', 1),
3  ('María Gómez', 'Cajero', 1),
4  ('Carlos López', 'Cocinero', 1),
5  ('Ana Rodríguez', 'Mesero', 1),
6  ('Pedro Martínez', 'Limpieza', 1),
7  ('Laura Sánchez', 'Gerente', 2),
8  ('Jorge Ramírez', 'Cajero', 2),
9  ('Diana Torres', 'Cocinero', 2),
10 ('Andrés Jiménez', 'Mesero', 2),
11 ('Sofía Herrera', 'Limpieza', 2),
12 ('Miguel González', 'Gerente', 3),
13 ('Carmen Vargas', 'Cajero', 3),
14 ('Ricardo Mendoza', 'Cocinero', 3),
15 ('Patricia Castro', 'Mesero', 3),
16 ('Fernando Rojas', 'Limpieza', 3),
17 ('Alejandra Silva', 'Gerente', 4),
18 ('Oscar Ortega', 'Cajero', 4),
19 ('Lucía Medina', 'Cocinero', 4),
20 ('Héctor Guerrero', 'Mesero', 4),
21 ('Verónica Paredes', 'Limpieza', 4),
22 ('Roberto Núñez', 'Gerente', 5),
23 ('Gabriela Espinoza', 'Cajero', 5),
24 ('Raúl Cortés', 'Cocinero', 5),
25 ('Isabel Mora', 'Mesero', 5),
26 ('Mario Salazar', 'Limpieza', 5),
27 ('Adriana Ríos', 'Gerente', 6),
28 ('Felipe Vega', 'Cajero', 6),
```

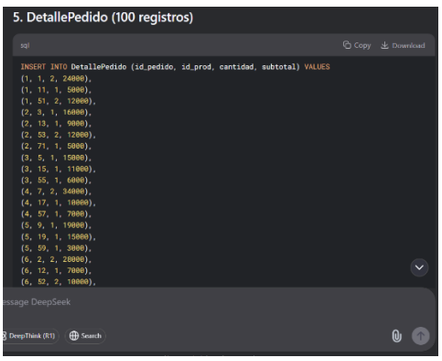
Results | Chart | Export — Source: Primary Database — Role: postgres — Run CTRL

Success. No rows returned

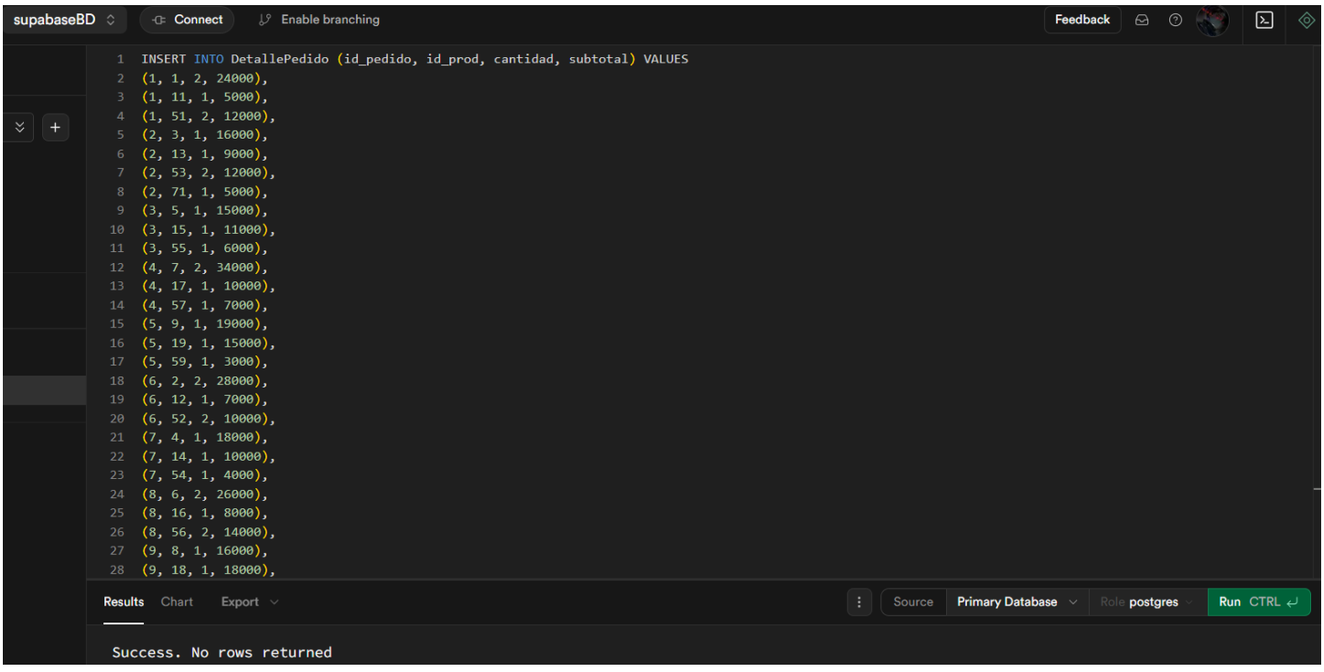

4. Pedido (100 registros)
sql
Copy
Download
INSERT INTO Pedido (fecha, id_rest, total) VALUES
('2023-01-05', 1, 35000),
('2023-01-05', 2, 42000),
('2023-01-06', 3, 28000),
('2023-01-06', 4, 51000),
('2023-01-07', 5, 37000),
('2023-01-07', 6, 45000),
('2023-01-08', 7, 32000),
('2023-01-08', 8, 48000),
('2023-01-09', 9, 29000),
('2023-01-09', 10, 53000),
('2023-01-10', 11, 36000),
('2023-01-10', 12, 44000),
('2023-01-11', 13, 31000),
('2023-01-11', 14, 47000),
('2023-01-12', 15, 33000),
('2023-01-12', 16, 49000),
('2023-01-13', 17, 34000),
Message DeepSeek
DeepThink (R1)
Search


Hecho con DeepSeek


supabaseBD
Connect
Enable branching
Feedback
INSERT INTO Pedido (fecha, id_rest, total) VALUES
('2023-01-05', 1, 35000),
('2023-01-05', 2, 42000),
('2023-01-06', 3, 28000),
('2023-01-06', 4, 51000),
('2023-01-07', 5, 37000),
('2023-01-07', 6, 45000),
('2023-01-08', 7, 32000),
('2023-01-08', 8, 48000),
('2023-01-09', 9, 29000),
('2023-01-09', 10, 53000),
('2023-01-10', 11, 36000),
('2023-01-10', 12, 44000),
('2023-01-11', 13, 31000),
('2023-01-11', 14, 47000),
('2023-01-12', 15, 33000),
('2023-01-12', 16, 49000),
('2023-01-13', 17, 34000),
('2023-01-13', 18, 46000),
('2023-01-14', 19, 35000),
('2023-01-14', 20, 52000),
('2023-01-15', 1, 38000),
('2023-01-15', 2, 43000),
('2023-01-16', 3, 27000),
('2023-01-16', 4, 50000),
('2023-01-17', 5, 39000),
('2023-01-17', 6, 41000),
('2023-01-18', 7, 30000),
Results
Chart
Export
Source
Primary Database
Role postgres
Run CTRL
Success. No rows returned

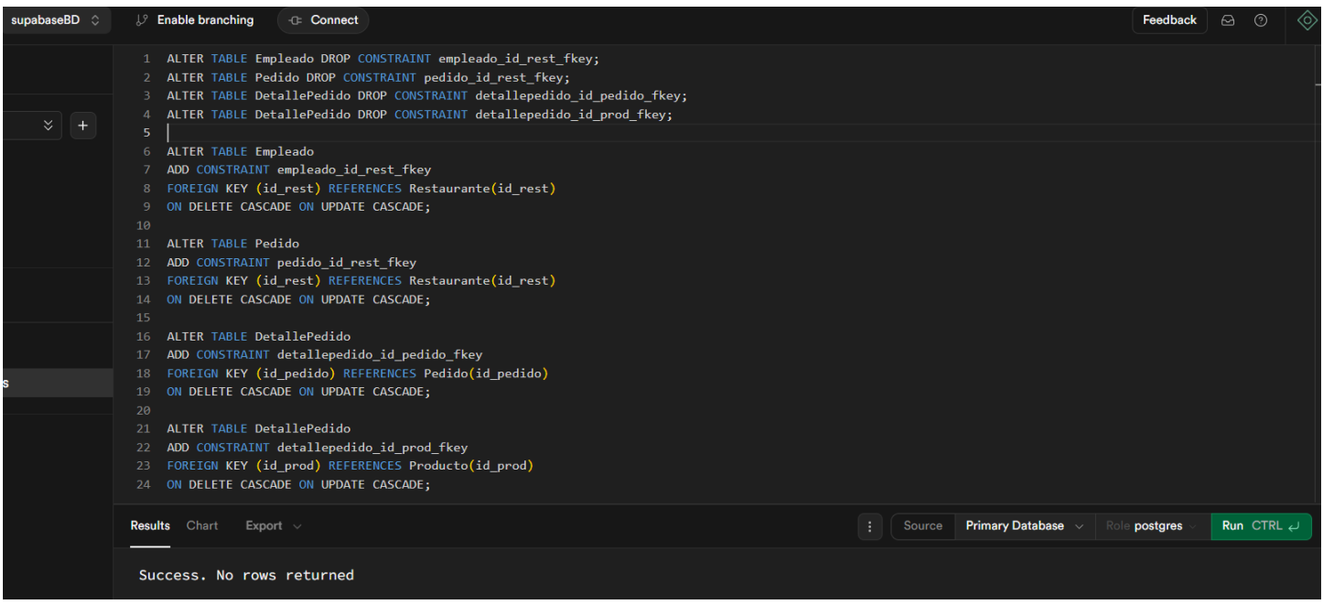

5. DetallePedido (100 registros)
sql
Copy
Download
INSERT INTO DetallePedido (id_pedido, id_prod, cantidad, subtotal) VALUES
(1, 1, 2, 24000),
(1, 11, 1, 5000),
(1, 51, 2, 12000),
(2, 3, 1, 16000),
(2, 13, 1, 9000),
(2, 53, 2, 12000),
(2, 71, 1, 5000),
(3, 5, 1, 15000),
(3, 15, 1, 11000),
(3, 55, 1, 6000),
(4, 7, 2, 34000),
(4, 17, 1, 10000),
(4, 57, 1, 7000),
(5, 9, 1, 19000),
(5, 19, 1, 15000),
(5, 59, 1, 3000),
(6, 2, 2, 28000),
(6, 12, 1, 7000),
(6, 52, 2, 10000),
Message DeepSeek
DeepThink (R1)
Search


Hecho con DeepSeek


supabaseBD
Connect
Enable branching
Feedback
INSERT INTO DetallePedido (id_pedido, id_prod, cantidad, subtotal) VALUES
(1, 1, 2, 24000),
(1, 11, 1, 5000),
(1, 51, 2, 12000),
(2, 3, 1, 16000),
(2, 13, 1, 9000),
(2, 53, 2, 12000),
(2, 71, 1, 5000),
(3, 5, 1, 15000),
(3, 15, 1, 11000),
(3, 55, 1, 6000),
(4, 7, 2, 34000),
(4, 17, 1, 10000),
(4, 57, 1, 7000),
(5, 9, 1, 19000),
(5, 19, 1, 15000),
(5, 59, 1, 3000),
(6, 2, 2, 28000),
(6, 12, 1, 7000),
(6, 52, 2, 10000),
(7, 4, 1, 18000),
(7, 14, 1, 10000),
(7, 54, 1, 4000),
(8, 6, 2, 26000),
(8, 16, 1, 8000),
(8, 56, 2, 14000),
(9, 8, 1, 16000),
(9, 18, 1, 18000),
Results
Chart
Export
Source
Primary Database
Role postgres
Run CTRL
Success. No rows returned

Aquí tuve que modificar las tablas ya que se me olvido poner el ON CASCADE y ON DELETE. Para eso uso un ALTER TABLE y modifico todas las tablas:

```
-- Primero eliminamos las restricciones de clave foránea existentes
ALTER TABLE Empleado DROP CONSTRAINT empleado_id_rest_fkey;
ALTER TABLE Pedido DROP CONSTRAINT pedido_id_rest_fkey;
ALTER TABLE DetallePedido DROP CONSTRAINT detallepedido_id_pedido_fkey;
ALTER TABLE DetallePedido DROP CONSTRAINT detallepedido_id_prod_fkey;

-- Luego las volvemos a crear con ON DELETE y ON UPDATE CASCADE
ALTER TABLE Empleado
ADD CONSTRAINT empleado_id_rest_fkey
FOREIGN KEY (id_rest) REFERENCES Restaurante(id_rest)
ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE Pedido
ADD CONSTRAINT pedido_id_rest_fkey
FOREIGN KEY (id_rest) REFERENCES Restaurante(id_rest)
ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE DetallePedido
ADD CONSTRAINT detallepedido_id_pedido_fkey
FOREIGN KEY (id_pedido) REFERENCES Pedido(id_pedido)
ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE DetallePedido
ADD CONSTRAINT detallepedido_id_prod_fkey
FOREIGN KEY (id_prod) REFERENCES Producto(id_prod)
ON DELETE CASCADE ON UPDATE CASCADE;
```



Documentación de las API's:

Restaurantes

- GET /api/restaurantes - Obtener todos los restaurantes
- GET /api/restaurantes/:id - Obtener un restaurante por ID
- POST /api/restaurantes - Crear un nuevo restaurante
- PUT /api/restaurantes/:id - Actualizar un restaurante
- DELETE /api/restaurantes/:id - Eliminar un restaurante

Empleados

- GET /api/empleados - Obtener todos los empleados
- GET /api/empleados/restaurante/:id_rest - Obtener empleados por restaurante
- GET /api/empleados/restaurante/:id_rest/rol/:rol - Obtener empleados por rol en un restaurante
- POST /api/empleados - Crear un nuevo empleado
- PUT /api/empleados/:id - Actualizar un empleado
- DELETE /api/empleados/:id - Eliminar un empleado

Productos

- GET /api/productos - Obtener todos los productos
- GET /api/productos/mas-vendidos/:unidades - Obtener productos más vendidos (más de X unidades)
- POST /api/productos - Crear un nuevo producto
- PUT /api/productos/:id - Actualizar un producto
- DELETE /api/productos/:id - Eliminar un producto

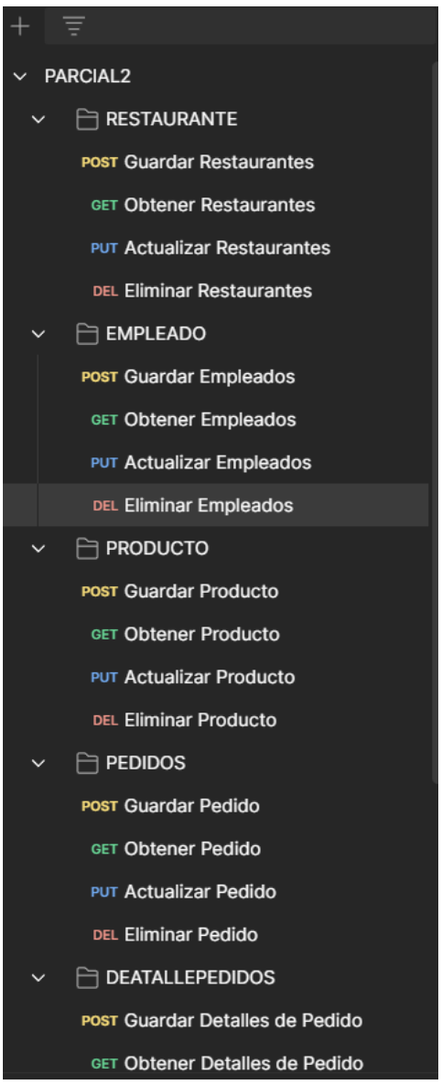
Pedidos

- GET /api/pedidos - Obtener todos los pedidos
- GET /api/pedidos/fecha/:fecha - Obtener pedidos por fecha
- GET /api/pedidos/ventas-restaurante - Obtener total de ventas por restaurante
- GET /api/pedidos/:id_pedido/productos - Obtener productos de un pedido específico
- POST /api/pedidos - Crear un nuevo pedido
- PUT /api/pedidos/:id - Actualizar un pedido
- DELETE /api/pedidos/:id - Eliminar un pedido

Detalles de Pedido

- POST /api/detalles-pedido - Crear un detalle de pedido
- PUT /api/detalles-pedido/:id - Actualizar un detalle de pedido
- DELETE /api/detalles-pedido/:id - Eliminar un detalle de pedido

Ahora uso postman para consumir las API's. Cree una nueva colección llamada "PARCIAL2" para las API's de las 5 tablas diferentes:



Aquí estoy haciendo una prueba con una solicitud POST para crear un restaurante:

Metodo: POST

URL: http://localhost:3000/api/restaurantes

Body (raw, JSON):

Asi sucesibamente para todos los demas endpoint a partir de la URL y cambiando el metodo a (GET, POST, PUT y DELETE) usando Postman, organizada por cada tabla/entidad.

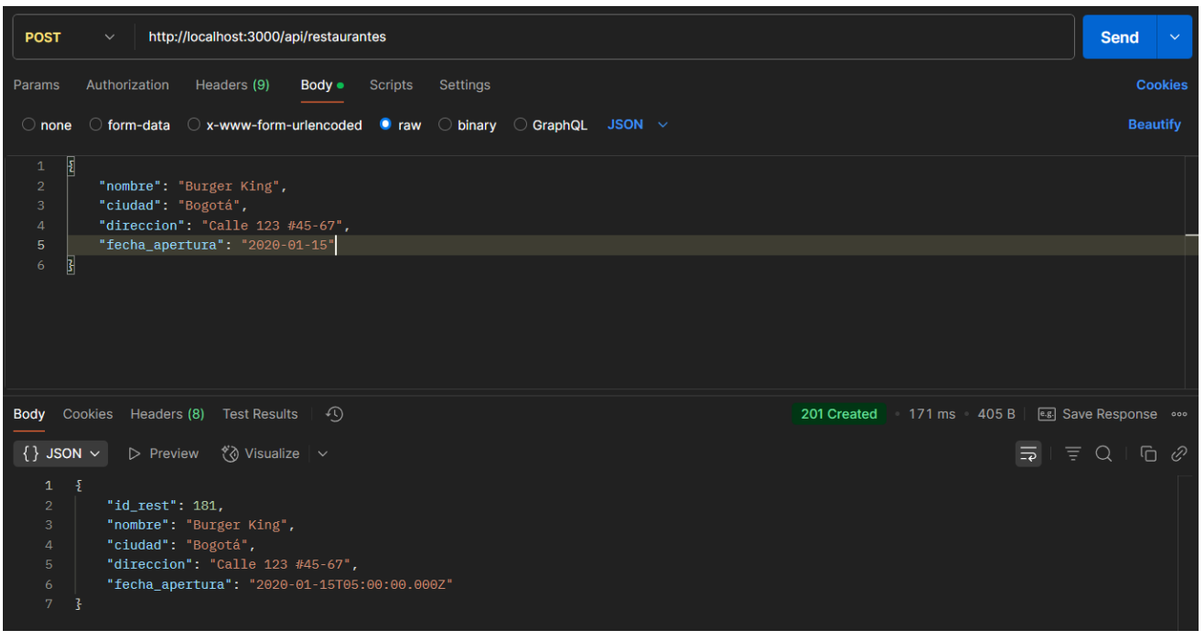


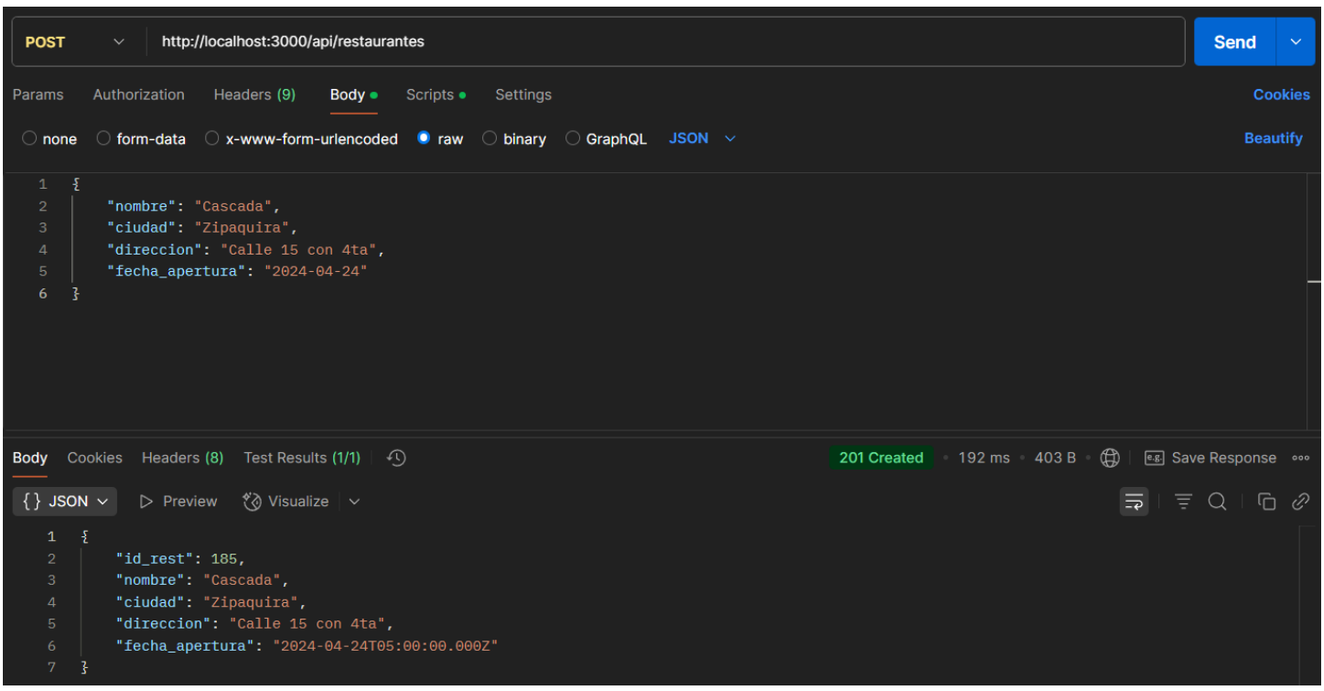
Tabla Restaurante:

POST Guardar Restaurantes:

createRestaurante: Esta API permite crear un nuevo restaurante. Recibe los datos del restaurante (nombre, ciudad, dirección y fecha de apertura) en el cuerpo de la solicitud y los inserta en la base de datos. Retorna el restaurante creado con su ID generado automáticamente y un código de estado 201 (creado).

Esta es la URL: http://localhost:3000/api/restaurantes

```
// Crear un nuevo restaurante
async function createRestaurante(req, res) {
    const { nombre, ciudad, direccion, fecha_apertura } = req.body;
    try {
        const result = await client.query(
            'INSERT INTO Restaurante (nombre, ciudad, direccion, fecha_apertura) VALUES ($1, $2, $3, $4) RETURNING *',
            [nombre, ciudad, direccion, fecha_apertura]
        );
        res.status(201).json(result.rows[0]);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

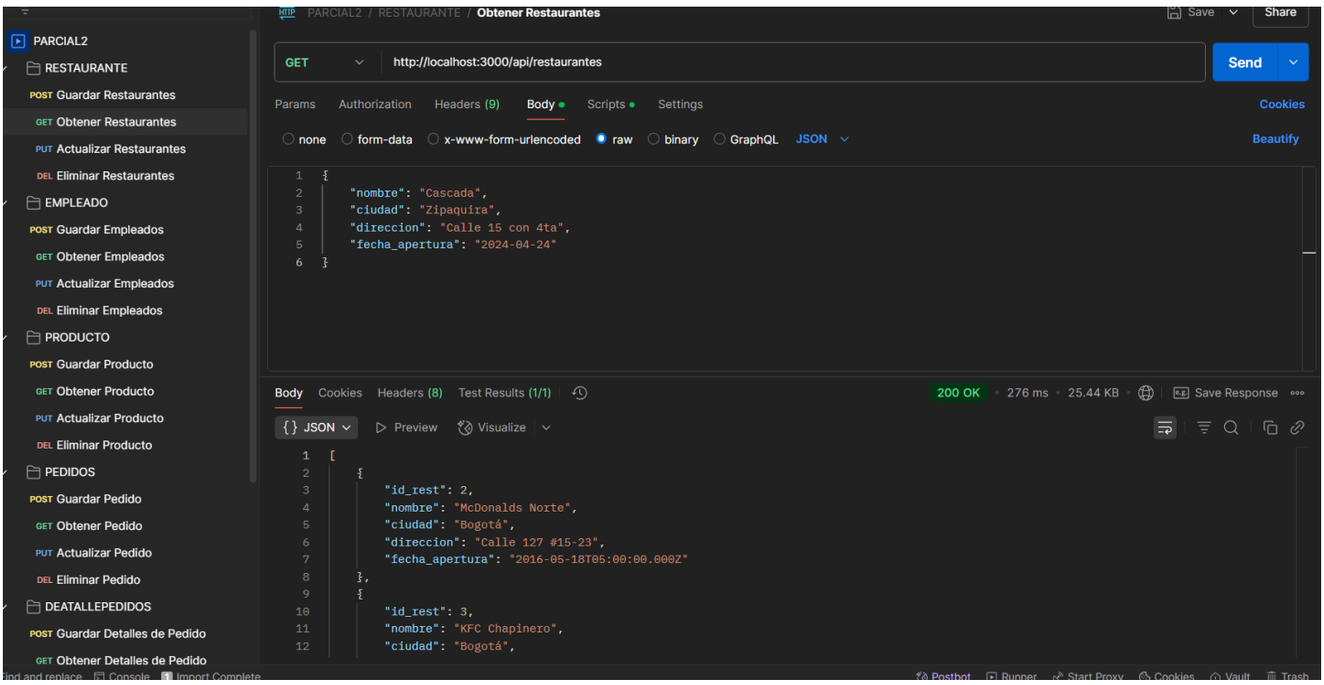


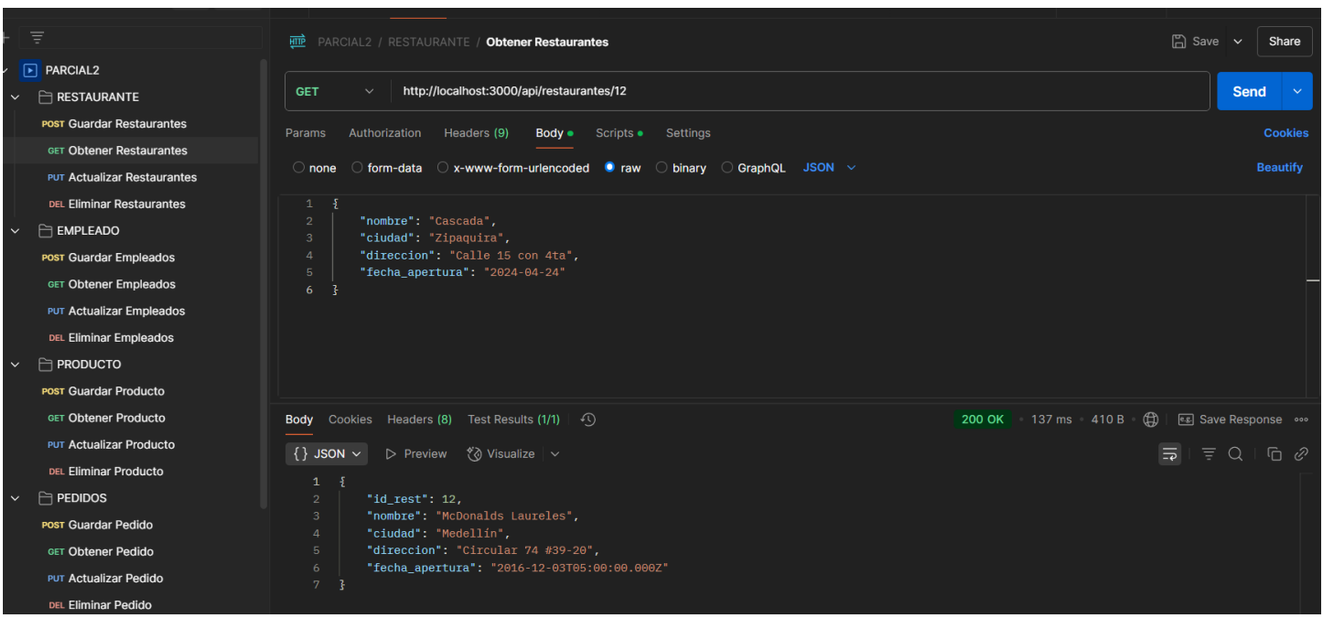
GET Obtener Restaurantes:

getAllRestaurantes: Esta API obtiene todos los restaurantes registrados en la base de datos. Realiza una consulta SQL para seleccionar todos los registros de la tabla "Restaurante" y los devuelve en formato JSON. Si ocurre un error, retorna un mensaje con el detalle del problema.

getRestauranteById: Esta API busca un restaurante específico según su ID. Recibe el ID como parámetro en la URL y realiza una consulta filtrada en la base de datos. Si no encuentra el restaurante, devuelve un mensaje de error 404; si lo encuentra, retorna los datos del restaurante en formato JSON.

```
// Obtener todos los restaurantes
async function getAllRestaurantes(req, res) {
    try {
        const result = await client.query('SELECT * FROM Restaurante');
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}

// Obtener un restaurante por ID
async function getRestauranteById(req, res) {
    const { id } = req.params;
    try {
        const result = await client.query('SELECT * FROM Restaurante WHERE id_rest = $1', [id]);
        if (result.rows.length === 0) {
            res.status(404).json({ message: 'Restaurante no encontrado' });
        } else {
            res.status(200).json(result.rows[0]);
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

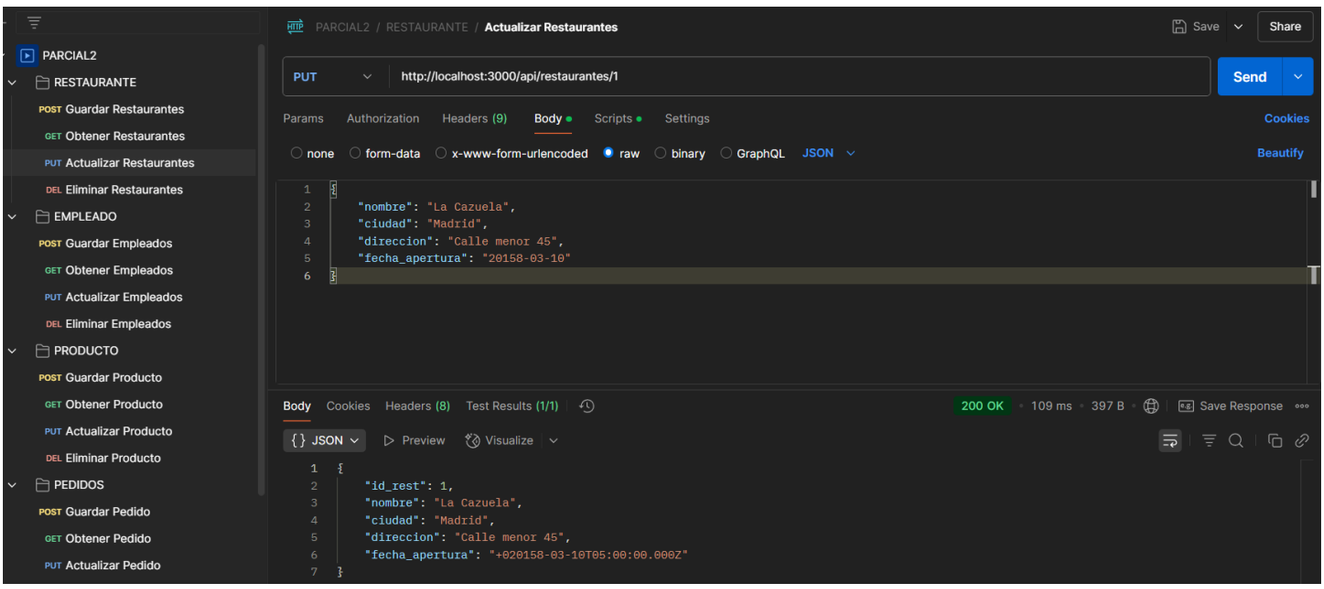




GET Actualizar Restaurantes:

updateRestaurante: Esta API actualiza los datos de un restaurante existente. Recibe el ID del restaurante en la URL y los nuevos datos en el cuerpo de la solicitud. Actualiza el registro correspondiente en la base de datos y retorna el restaurante actualizado. Si no encuentra el restaurante, devuelve un error 404.

```
// Actualizar un restaurante
async function updateRestaurante(req, res) {
    const { id } = req.params;
    const { nombre, ciudad, direccion, fecha_apertura } = req.body;
    try {
        const result = await client.query(
            'UPDATE Restaurante SET nombre = $1, ciudad = $2, direccion = $3, fecha_apertura = $4 WHERE id_rest = $5 RETURNING *',
            [nombre, ciudad, direccion, fecha_apertura, id]
        );
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Restaurante no encontrado' });
        } else {
            res.status(200).json(result.rows[0]);
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



GET Eliminar Restaurantes:

deleteRestaurante: Esta API elimina un restaurante de la base de datos según su ID. Recibe el ID como parámetro en la URL, elimina el registro correspondiente y retorna un mensaje de confirmación. Si el restaurante no existe, devuelve un error 404.

```
// Eliminar un restaurante
async function deleteRestaurante(req, res) {
    const { id } = req.params;
    try {
        const result = await client.query('DELETE FROM Restaurante WHERE id_rest = $1 RETURNING *', [id]);
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Restaurante no encontrado' });
        } else {
            res.status(200).json({ message: 'Restaurante eliminado' });
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

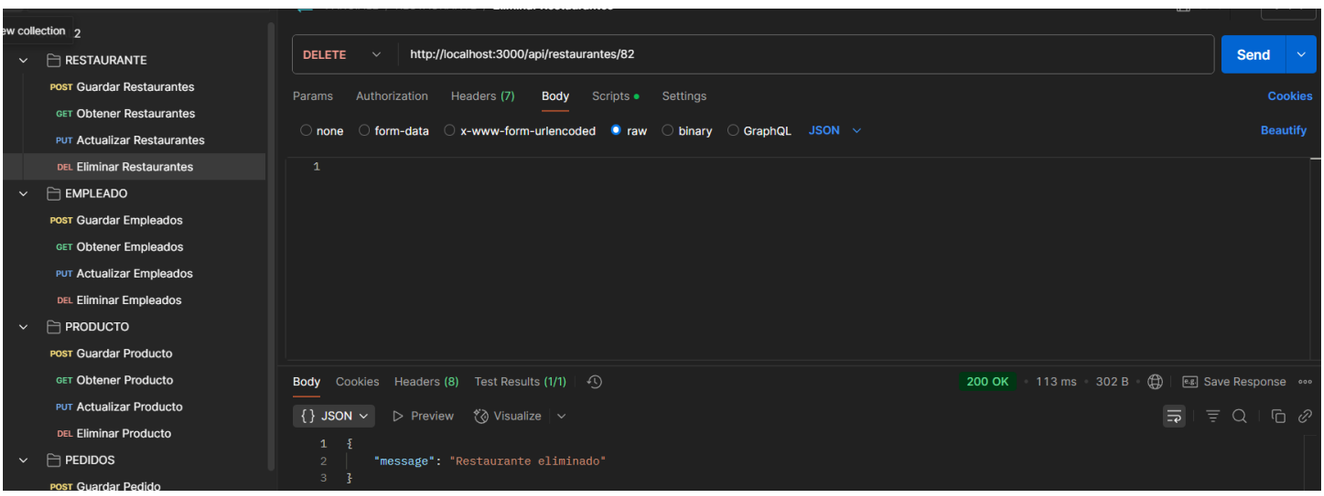
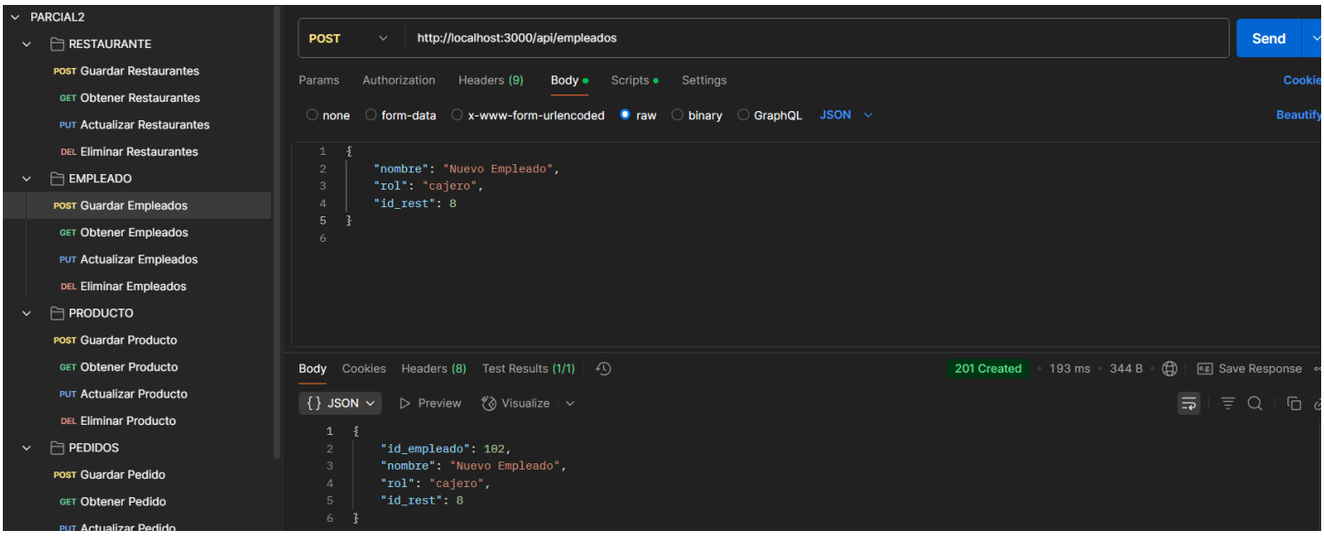


Tabla Empleado:

POST Guardar Empleado:

createEmpleado: Permite crear un nuevo registro de empleado en la base de datos. Recibe los datos del empleado (nombre, rol e ID de restaurante) en el cuerpo de la solicitud y los inserta en la tabla Empleado. Devuelve el registro creado con su ID generado o un error si la operación falla.

```
// Crear un nuevo empleado
async function createEmpleado(req, res) {
    const { nombre, rol, id_rest } = req.body;
    try {
        const result = await client.query(
            'INSERT INTO Empleado (nombre, rol, id_rest) VALUES ($1, $2, $3) RETURNING *',
            [nombre, rol, id_rest]
        );
        res.status(201).json(result.rows[0]);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



GET Obtener Empleado:

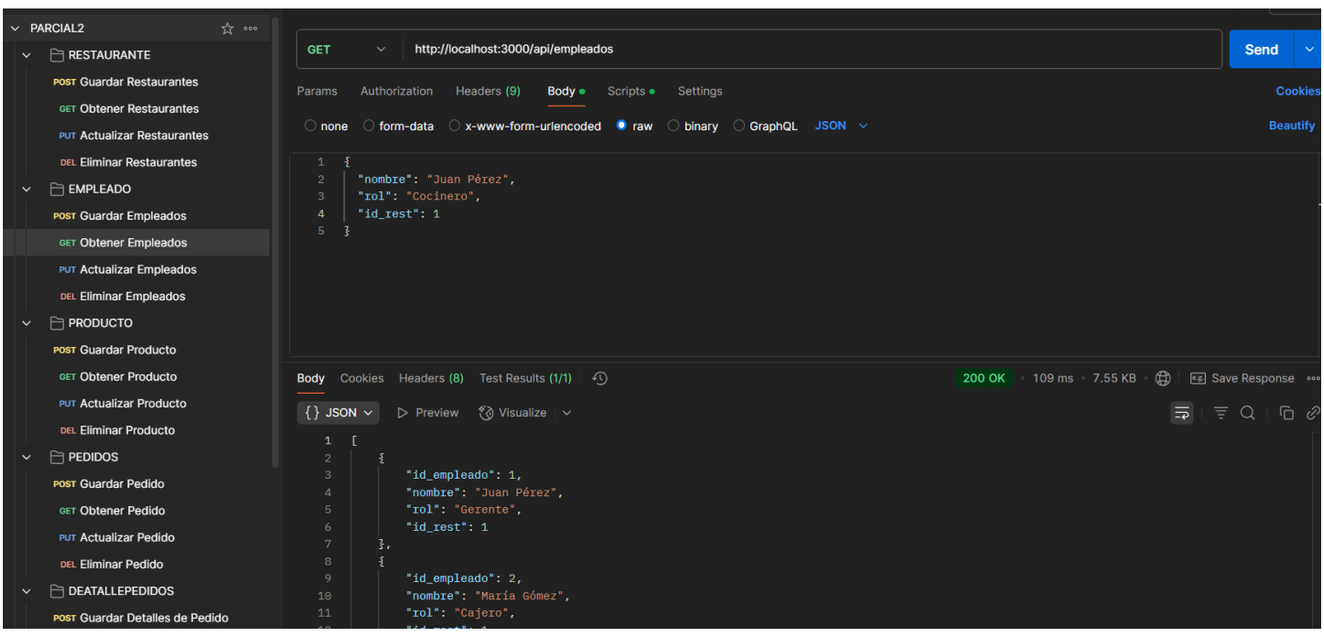
getAllEmpleados: Esta API obtiene todos los registros de empleados almacenados en la base de datos. Realiza una consulta SQL simple para seleccionar todos los campos de la tabla Empleado y devuelve los resultados en formato JSON. En caso de error, retorna un mensaje con el detalle del problema.

getEmpleadosByRestaurante: Este endpoint filtra los empleados según el restaurante al que pertenecen. Recibe como parámetro el ID del restaurante y ejecuta una consulta SQL con esta condición. Devuelve solo los empleados que coinciden con el restaurante especificado o un error si falla la consulta.

getEmpleadosByRol: Esta API combina dos criterios de búsqueda: restaurante y rol. Toma ambos parámetros de la URL y realiza una consulta SQL con estas dos condiciones simultáneas. Es útil

para obtener, por ejemplo, todos los cocineros de un restaurante específico. Retorna los empleados que cumplen con ambas condiciones.

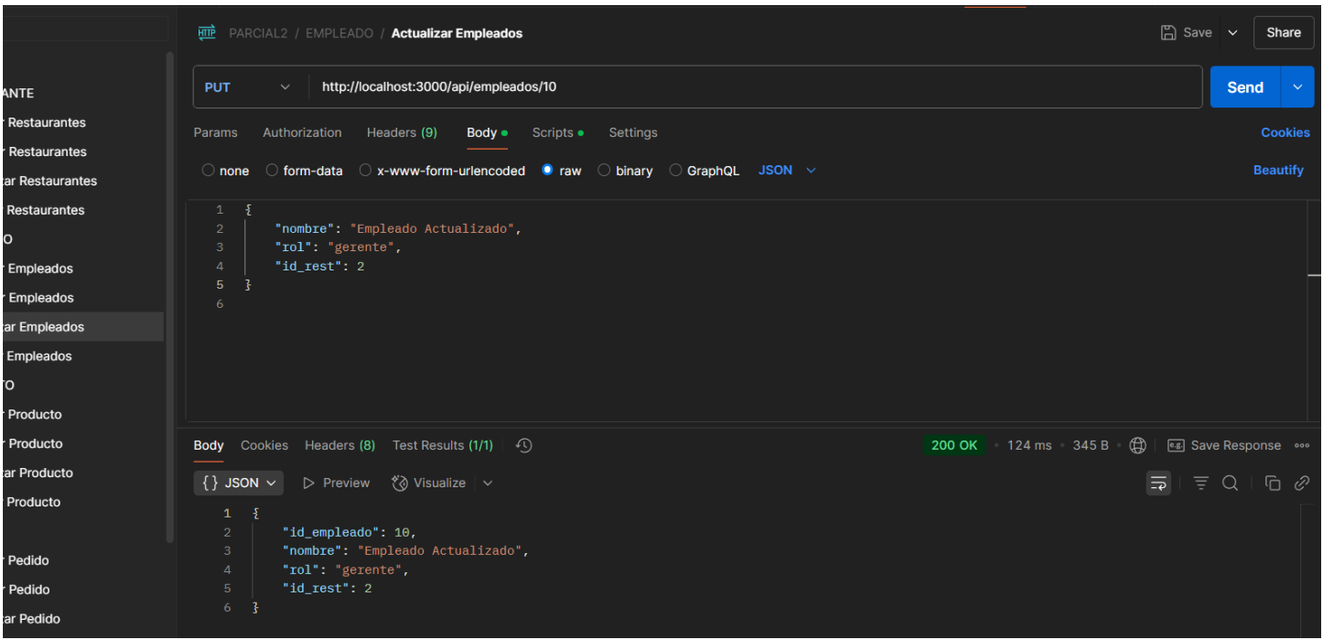
```
trollers > empleados.js > ...
const client = require('../bd');

// Obtener todos los empleados
async function getAllEmpleados(req, res) {
    try {
        const result = await client.query('SELECT * FROM Empleado');
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}

// Obtener empleados por restaurante
async function getEmpleadosByRestaurante(req, res) {
    const { id_rest } = req.params;
    try {
        const result = await client.query('SELECT * FROM Empleado WHERE id_rest = $1', [id_rest]);
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}

// Obtener empleados por rol en un restaurante
async function getEmpleadosByRol(req, res) {
    const { id_rest, rol } = req.params;
    try {
        const result = await client.query(
            'SELECT * FROM Empleado WHERE id_rest = $1 AND rol = $2',
            [id_rest, rol]
        );
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



PUT Actualizar Empleado:

updateEmpleado: Actualiza la información de un empleado existente. Recibe el ID del empleado a modificar como parámetro y los nuevos datos en el cuerpo de la solicitud. Ejecuta una sentencia SQL de actualización y retorna el registro modificado. Si no encuentra el empleado, devuelve un mensaje indicando que no existe.

```
// Actualizar un empleado
async function updateEmpleado(req, res) {
    const { id } = req.params;
    const { nombre, rol, id_rest } = req.body;
    try {
        const result = await client.query(
            'UPDATE Empleado SET nombre = $1, rol = $2, id_rest = $3 WHERE id_empleado = $4 RETURNING *',
            [nombre, rol, id_rest, id]
        );
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Empleado no encontrado' });
        } else {
            res.status(200).json(result.rows[0]);
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



DELETE Eliminar Empleado:

deleteEmpleado: Elimina un empleado de la base de datos según su ID, que recibe como parámetro en la URL. Realiza una eliminación física del registro y confirma la operación retornando un mensaje de éxito. Si el empleado no existe, devuelve un mensaje adecuado. En caso de error, retorna los detalles del problema.

```
// Eliminar un empleado
async function deleteEmpleado(req, res) {
    const { id } = req.params;
    try {
        const result = await client.query('DELETE FROM Empleado WHERE id_empleado = $1 RETURNING *', [id]);
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Empleado no encontrado' });
        } else {
            res.status(200).json({ message: 'Empleado eliminado' });
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

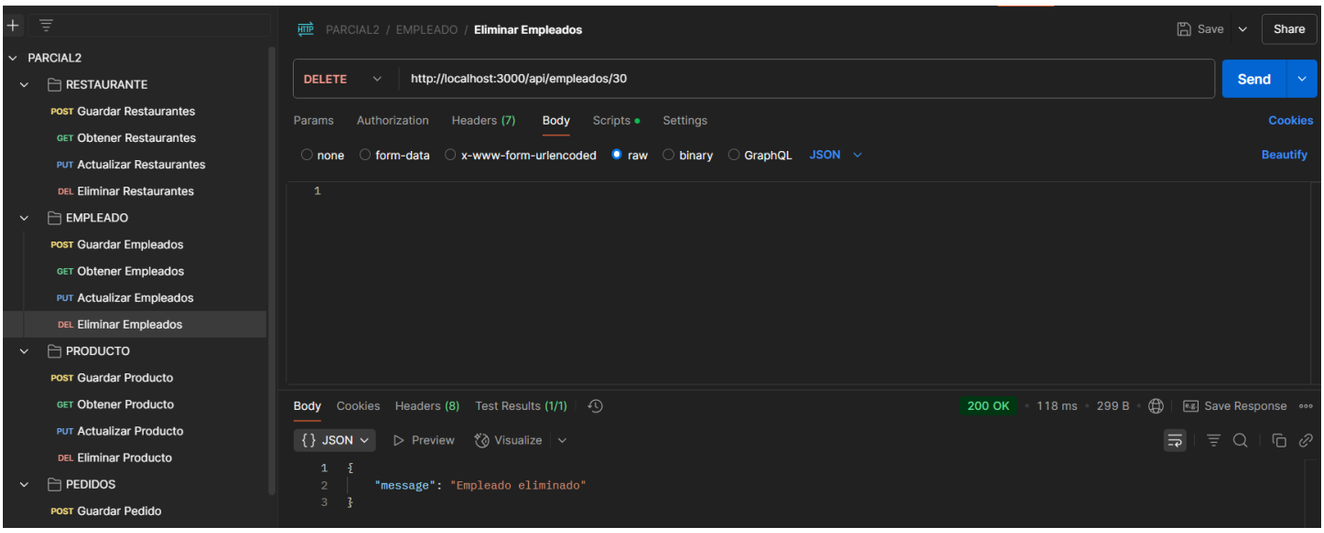
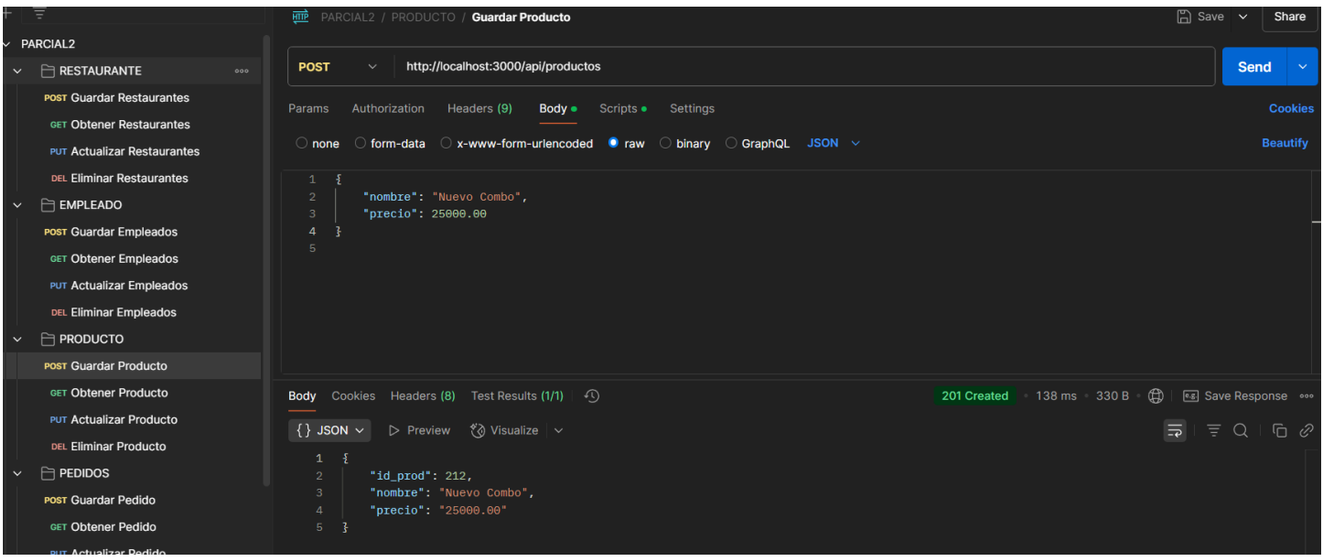


Tabla Producto:

POST Guardar Producto:

createProducto: Esta API permite crear un nuevo producto en la base de datos. Recibe en el cuerpo de la solicitud el nombre y precio del producto, los inserta en la tabla "Producto" y devuelve el registro creado. Si hay un error, retorna un mensaje descriptivo.

```
// Crear un nuevo producto
async function createProducto(req, res) {
    const { nombre, precio } = req.body;
    try {
        const result = await client.query(
            'INSERT INTO Producto (nombre, precio) VALUES ($1, $2) RETURNING *',
            [nombre, precio]
        );
        res.status(201).json(result.rows[0]);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

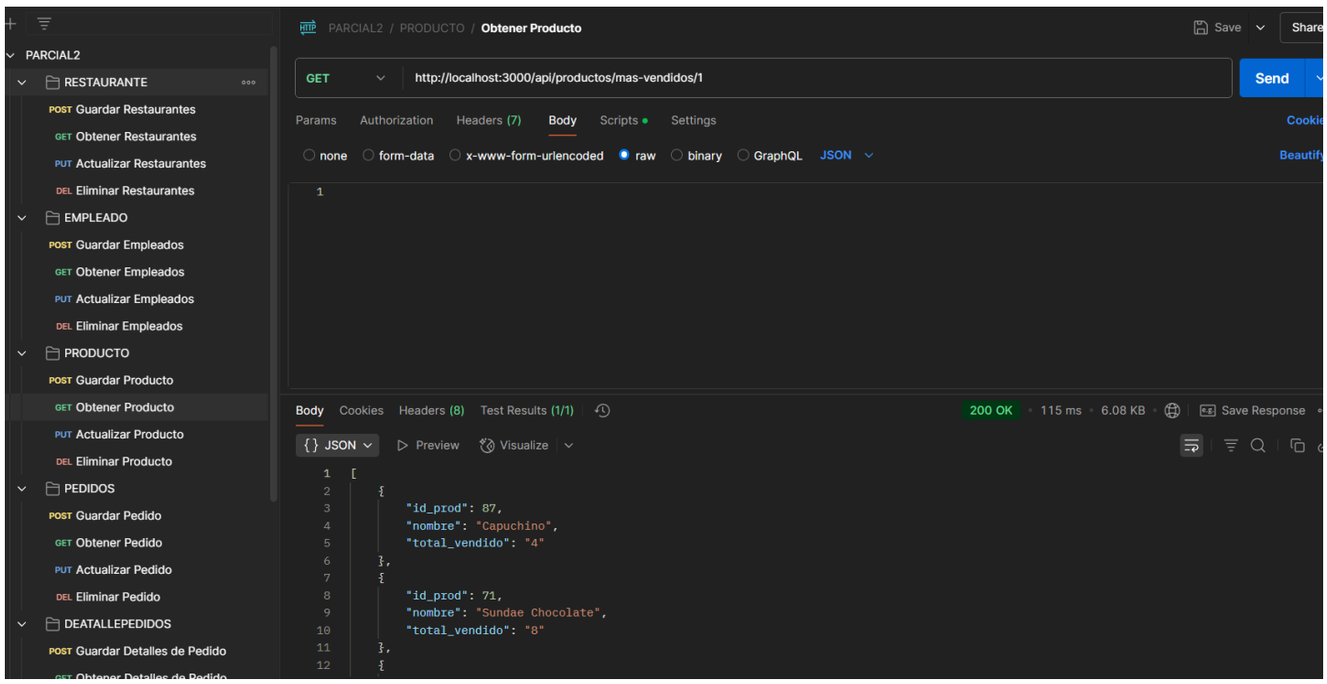


GET Obtener Producto:

getAllProductos: Esta API obtiene todos los productos registrados en la base de datos. Realiza una consulta SQL para seleccionar todos los registros de la tabla "Producto" y los devuelve en formato JSON. En caso de error, retorna un mensaje con el detalle del problema.

getProductosMasVendidos: Esta API filtra los productos más vendidos según una cantidad mínima especificada. Recibe un parámetro "unidades" y busca los productos cuya suma total de ventas (obtenida de la tabla "DetallePedido") supere ese valor. El resultado incluye el ID, nombre y total de unidades vendidas de cada producto.

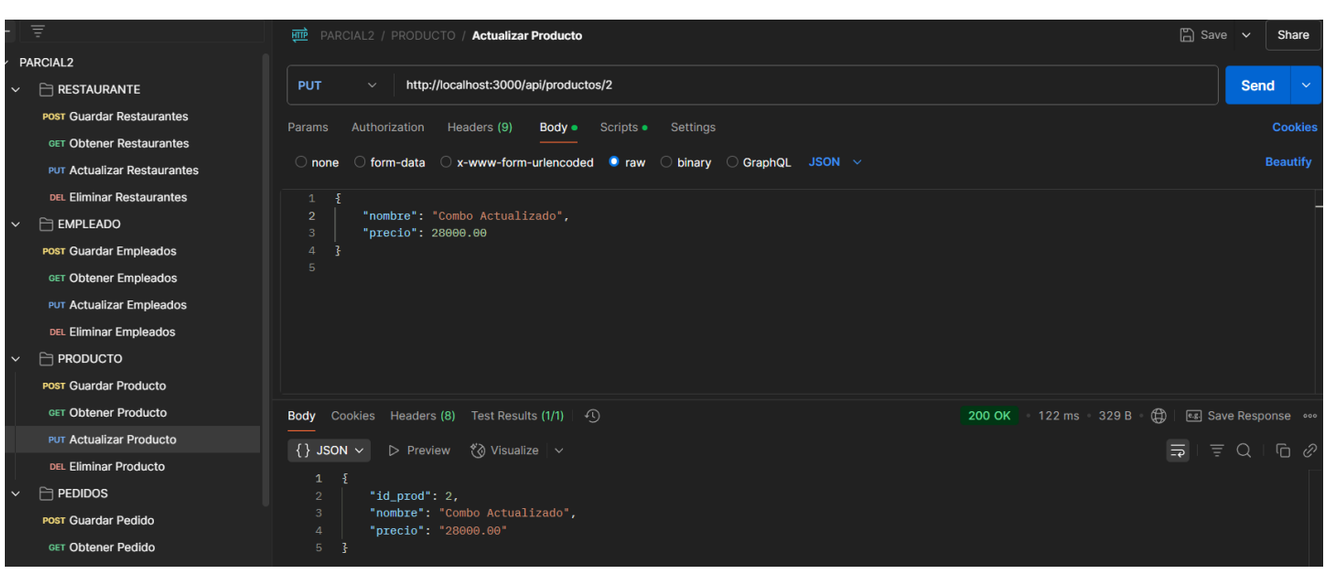
```
// Obtener todos los productos
async function getAllProductos(req, res) {
    try {
        const result = await client.query('SELECT * FROM Producto');
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}

// Obtener productos más vendidos (más de X unidades)
async function getProductosMasVendidos(req, res) {
    const { unidades } = req.params;
    try {
        const result = await client.query(
            `SELECT p.id_prod, p.nombre, SUM(dp.cantidad) as total_vendido
             FROM Producto p
             JOIN DetallePedido dp ON p.id_prod = dp.id_prod
             GROUP BY p.id_prod, p.nombre
             HAVING SUM(dp.cantidad) > $1`,
            [unidades]
        );
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



PUT Actualizar Producto:

updateProducto: Esta API actualiza la información de un producto existente. Recibe el ID del producto a modificar (como parámetro) y los nuevos valores para nombre y precio (en el cuerpo de la solicitud). Retorna el producto actualizado o un mensaje si no se encuentra el producto especificado.

```
// Actualizar un producto
async function updateProducto(req, res) {
    const { id } = req.params;
    const { nombre, precio } = req.body;
    try {
        const result = await client.query(
            'UPDATE Producto SET nombre = $1, precio = $2 WHERE id_prod = $3 RETURNING *',
            [nombre, precio, id]
        );
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Producto no encontrado' });
        } else {
            res.status(200).json(result.rows[0]);
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



DELETE Eliminar Producto:

deleteProducto: Esta API elimina un producto de la base de datos según su ID (recibido como parámetro). Si el producto existe y se elimina correctamente, retorna un mensaje de confirmación. Si no encuentra el producto o ocurre un error, devuelve el mensaje correspondiente.

```
// Eliminar un producto
async function deleteProducto(req, res) {
    const { id } = req.params;
    try {
        const result = await client.query('DELETE FROM Producto WHERE id_prod = $1 RETURNING *', [id]);
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Producto no encontrado' });
        } else {
            res.status(200).json({ message: 'Producto eliminado' });
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

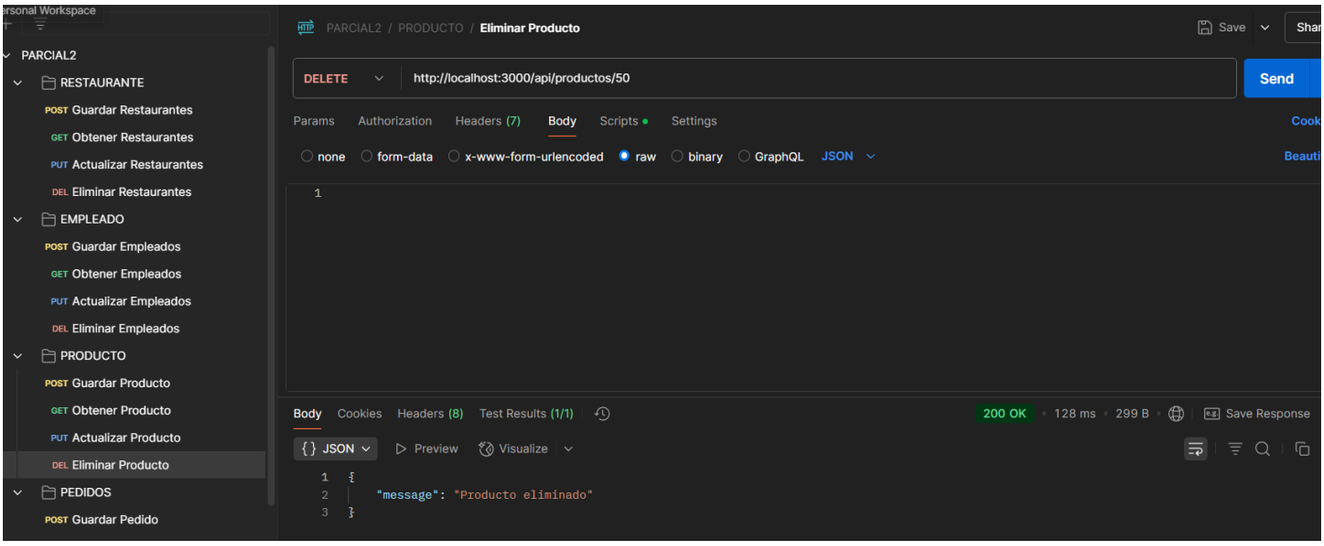
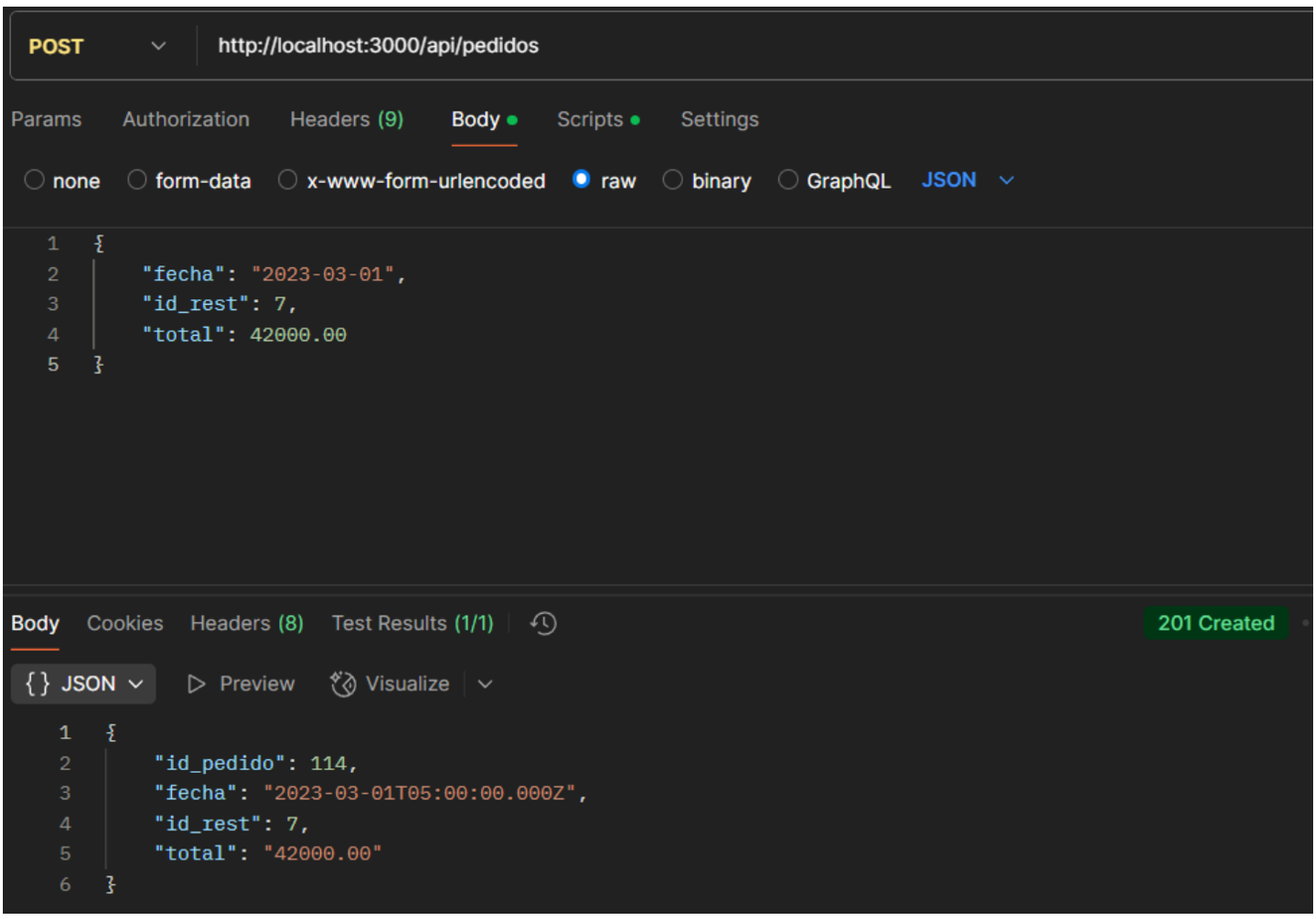


Tabla Pedido:

POST Guardar Pedido:

createPedido: Crea un nuevo registro en la tabla "Pedido" con los datos proporcionados (fecha, ID de restaurante y total). Retorna el pedido creado con su ID generado automáticamente.

```
// Crear un nuevo pedido
Changed lines
ion createPedido(req, res) {
    const { fecha, id_rest, total } = req.body;
    try {
        const result = await client.query(
            'INSERT INTO Pedido (fecha, id_rest, total) VALUES ($1, $2, $3) RETURNING *',
            [fecha, id_rest, total]
        );
        res.status(201).json(result.rows[0]);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



GET Obtener Pedido:

getAllPedidos: Esta API obtiene todos los registros de la tabla "Pedido" de la base de datos y los devuelve como respuesta en formato JSON. Maneja errores internos del servidor con un código 500.

getPedidosByFecha: Recibe una fecha como parámetro y busca en la base de datos todos los pedidos que coincidan con esa fecha específica. Retorna los resultados encontrados o un error si falla la consulta.

getVentasByRestaurante: Calcula el total de ventas agrupadas por restaurante, sumando los montos de todos los pedidos asociados a cada uno. La consulta une las tablas "Restaurante" y "Pedido" para obtener esta información consolidada.

getProductosByPedido: Obtiene los productos asociados a un pedido específico (identificado por su ID), mostrando detalles como nombre, cantidad y subtotal. Realiza un JOIN entre las tablas "Producto" y "DetallePedido" para esta consulta.

```
// Obtener pedidos por fecha
async function getPedidosByFecha(req, res) {
    const { fecha } = req.params;
    try {
        const result = await client.query('SELECT * FROM Pedido WHERE fecha = $1', [fecha]);
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}

// Obtener total de ventas por restaurante
async function getVentasByRestaurante(req, res) {
    try {
        const result = await client.query(
            `SELECT r.id_rest, r.nombre, SUM(p.total) as total_ventas
             FROM Restaurante r
             JOIN Pedido p ON r.id_rest = p.id_rest
             GROUP BY r.id_rest, r.nombre`
        );
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

```
// Obtener productos de un pedido específico
async function getProductosByPedido(req, res) {
    const { id_pedido } = req.params;
    try {
        const result = await client.query(
            `SELECT p.id_prod, p.nombre, dp.cantidad, dp.subtotal
             FROM Producto p
             JOIN DetallePedido dp ON p.id_prod = dp.id_prod
             WHERE dp.id_pedido = $1`,
            [id_pedido]
        );
        res.status(200).json(result.rows);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

GET http://localhost:3000/api/pedidos

Params Authorization Headers (7) Body Scripts Settings

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

```
1
```

Body Cookies Headers (8) Test Results (1/1) 200 OK

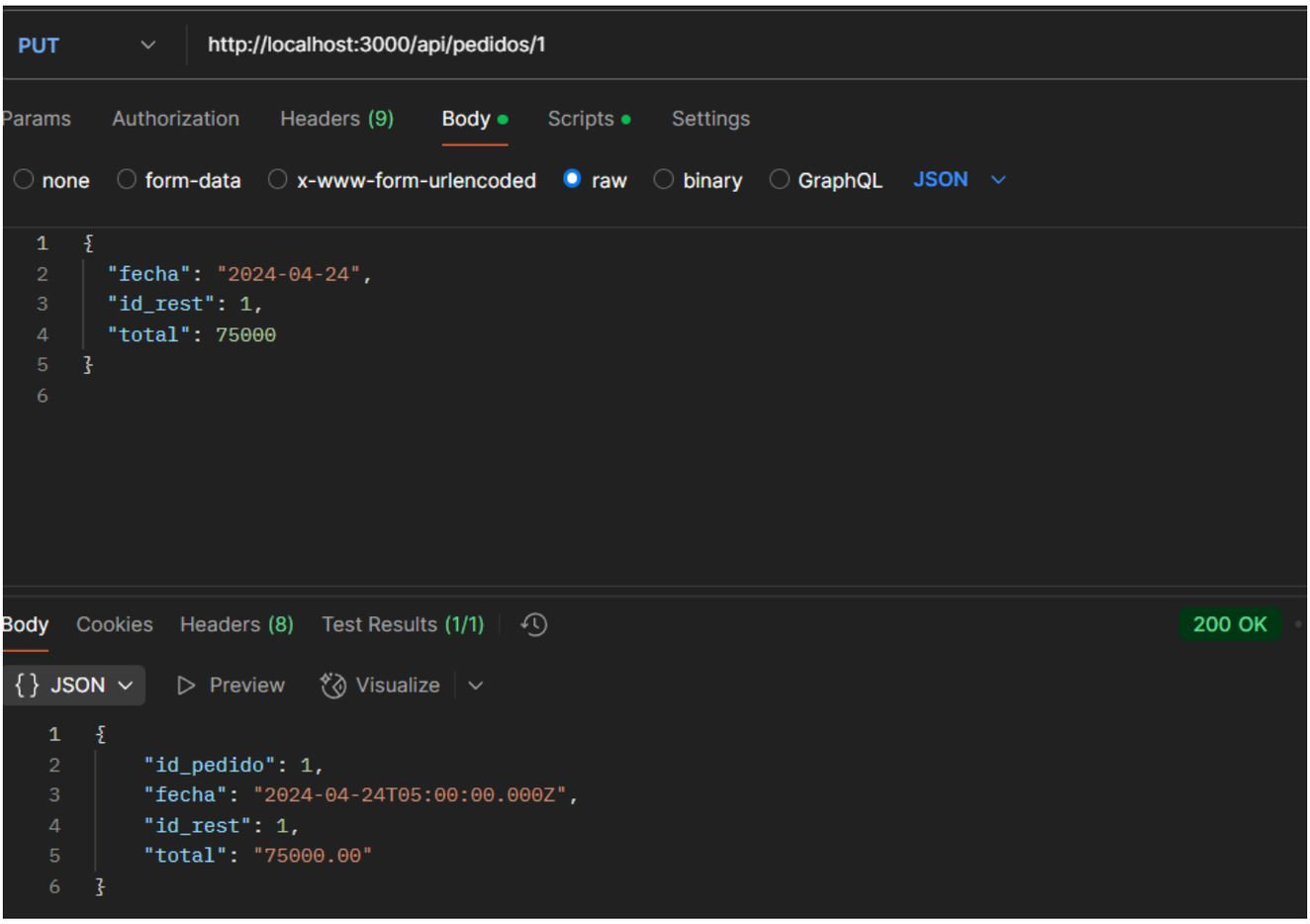
{} JSON Preview Visualize


```
[
    {
        "id_pedido": 2,
        "fecha": "2023-01-05T05:00:00.000Z",
        "id_rest": 2,
        "total": "42000.00"
    },
    {
        "id_pedido": 3,
        "fecha": "2023-01-06T05:00:00.000Z",
        "id_rest": 3,
        "total": "28000.00"
```

PUT Actualizar Pedido:

updatePedido: Actualiza la información de un pedido existente (identificado por su ID) con los nuevos datos proporcionados. Verifica si el pedido existe antes de intentar la actualización.

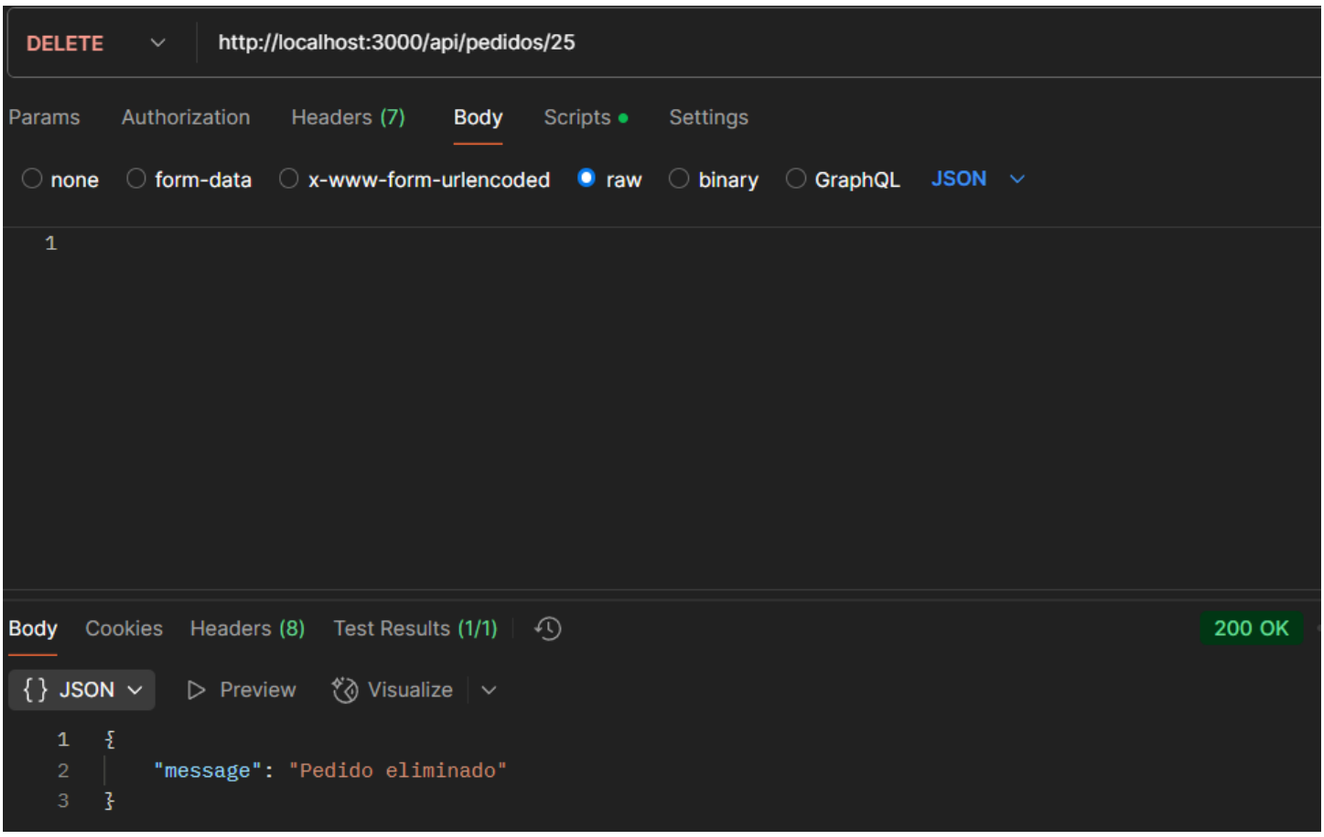
```
// Actualizar un pedido
async function updatePedido(req, res) {
    const { id } = req.params;
    const { fecha, id_rest, total } = req.body;
    try {
        const result = await client.query(
            'UPDATE Pedido SET fecha = $1, id_rest = $2, total = $3 WHERE id_pedido = $4 RETURNING *',
            [fecha, id_rest, total, id]
        );
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Pedido no encontrado' });
        } else {
            res.status(200).json(result.rows[0]);
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



DELETE Eliminar Pedido:

deletePedido: Elimina un pedido específico de la base de datos usando su ID como referencia. Confirma si el pedido existía antes de eliminarlo y retorna un mensaje apropiado.

```
// Eliminar un pedido
async function deletePedido(req, res) {
    const { id } = req.params;
    try {
        const result = await client.query('DELETE FROM Pedido WHERE id_pedido = $1 RETURNING *', [id]);
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Pedido no encontrado' });
        } else {
            res.status(200).json({ message: 'Pedido eliminado' });
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

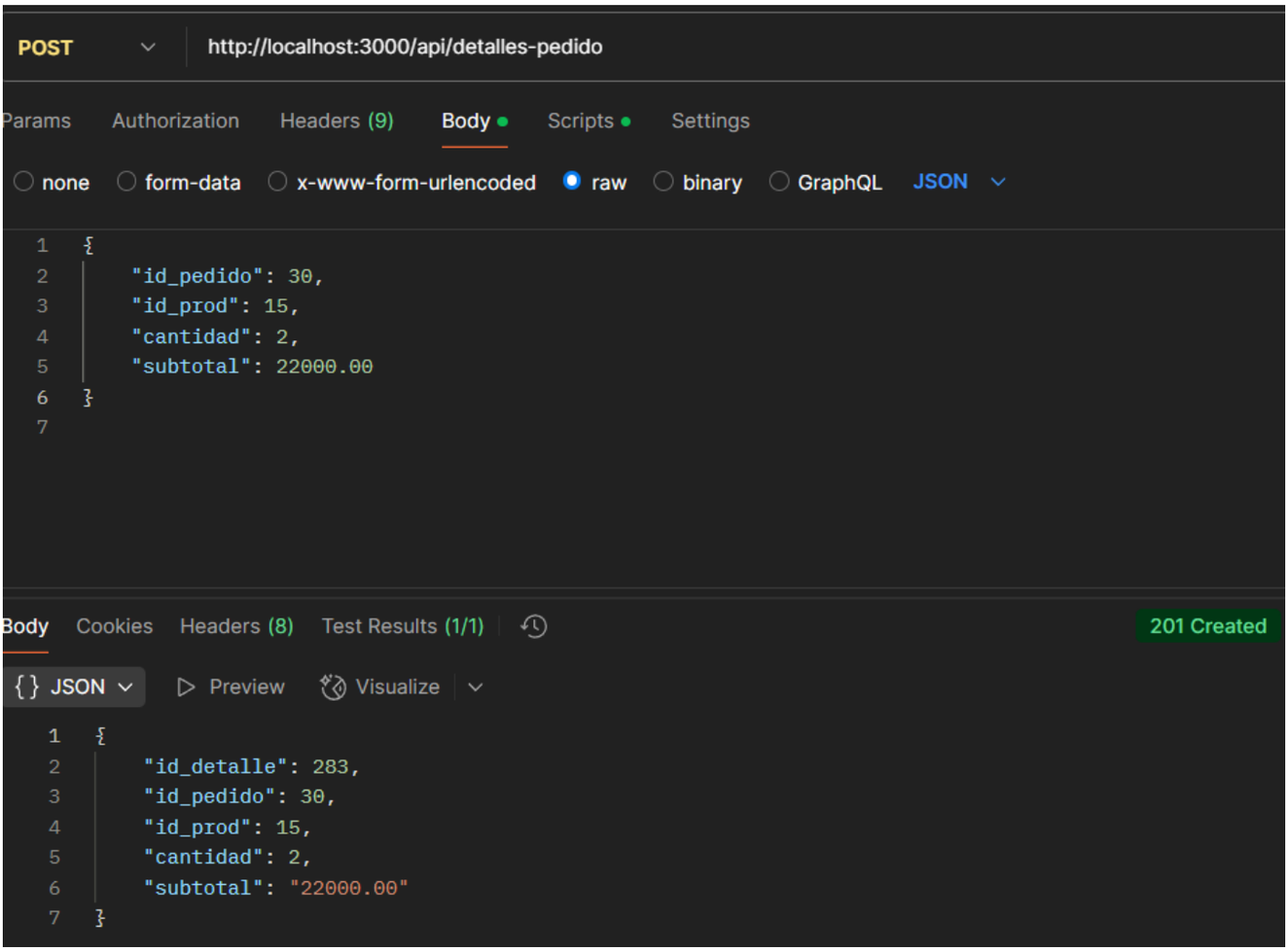


DetallePedido:

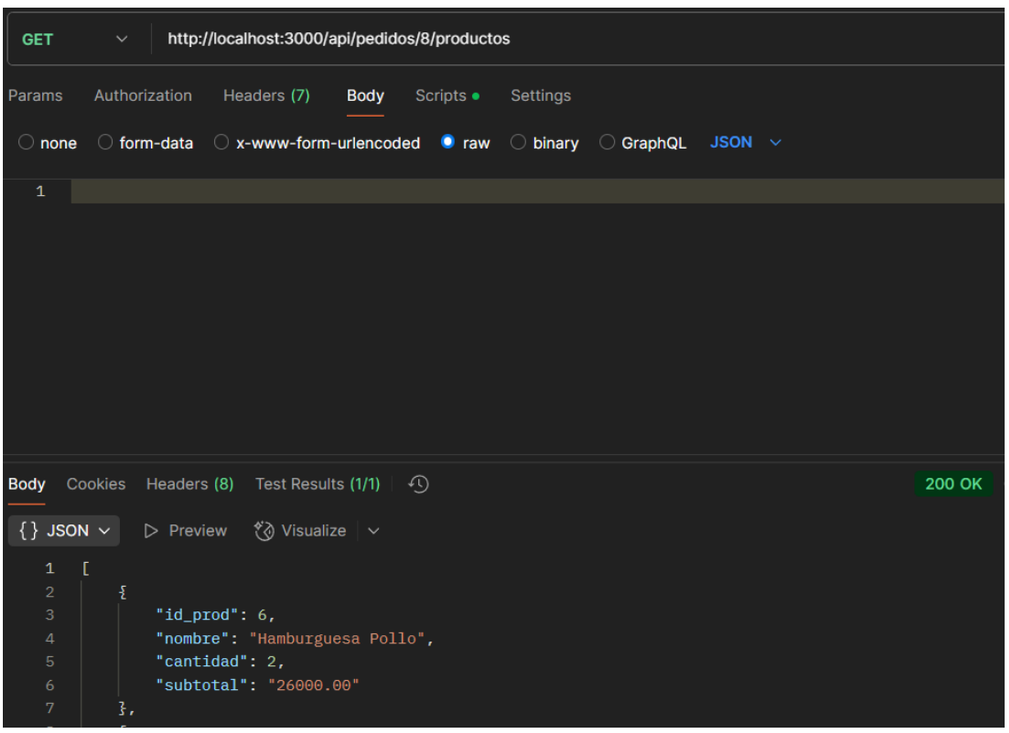
POST Guardar DetallePedido:

createDetallePedido: Esta API permite crear un nuevo detalle de pedido en la base de datos. Recibe los datos del cuerpo de la solicitud (id_pedido, id_prod, cantidad y subtotal), los inserta en la tabla DetallePedido y devuelve el registro creado con un código de estado 201. Si ocurre un error, responde con un mensaje de error y estado 500.

```
// Crear un detalle de pedido
async function createDetallePedido(req, res) {
    const { id_pedido, id_prod, cantidad, subtotal } = req.body;
    try {
        const result = await client.query(
            'INSERT INTO DetallePedido (id_pedido, id_prod, cantidad, subtotal) VALUES ($1, $2, $3, $4) RETURNING *',
            [id_pedido, id_prod, cantidad, subtotal]
        );
        res.status(201).json(result.rows[0]);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



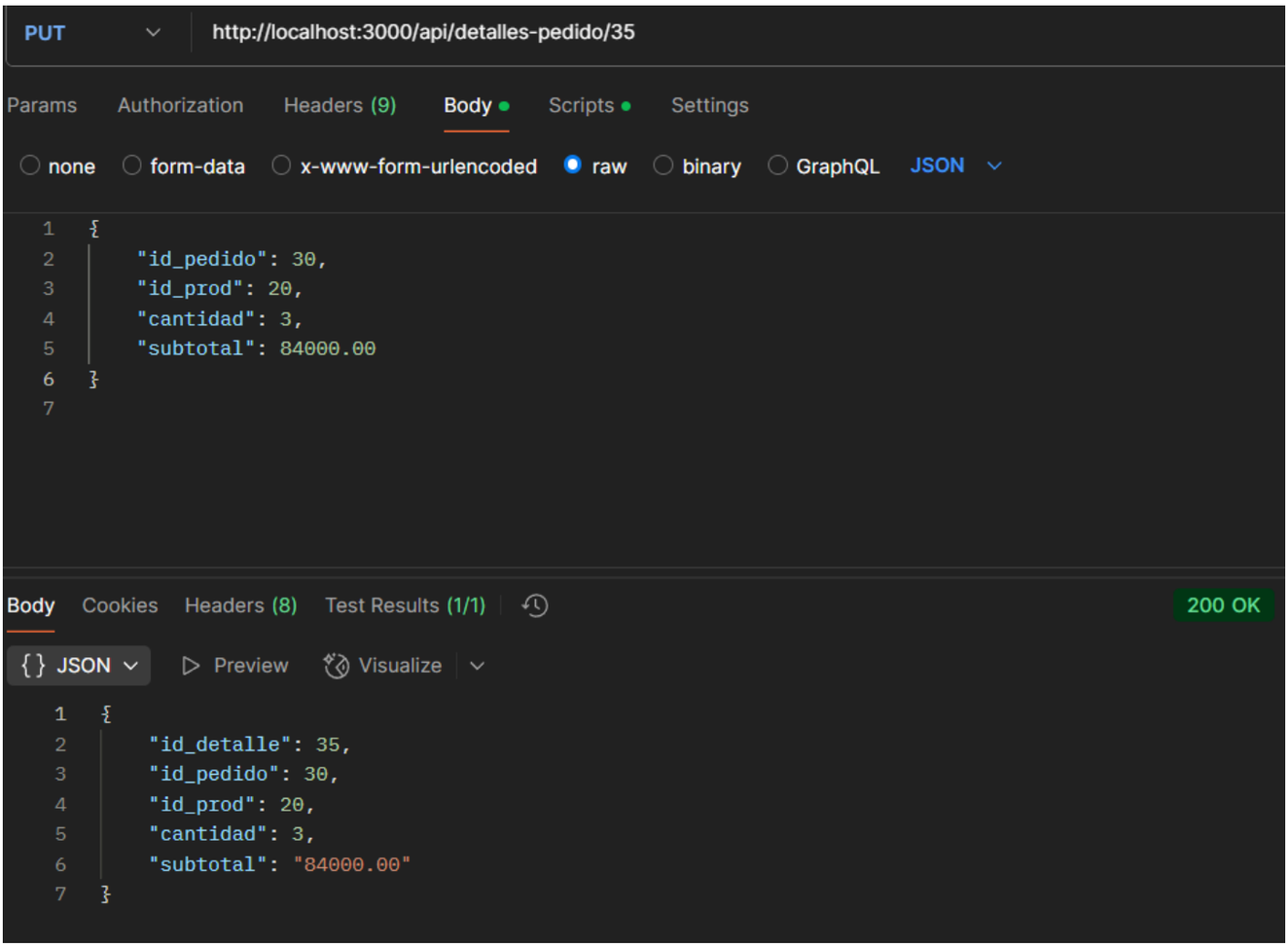
GET Obtener DetallePedido:



PUT Actualizar DetallePedido:

updateDetallePedido: Esta función actualiza un detalle de pedido existente identificado por el parámetro id. Toma los nuevos valores del cuerpo de la solicitud y actualiza el registro correspondiente en la base de datos. Si no encuentra el detalle, devuelve un error 404. En caso de éxito, retorna el detalle actualizado con estado 200, o un error 500 si falla la operación.

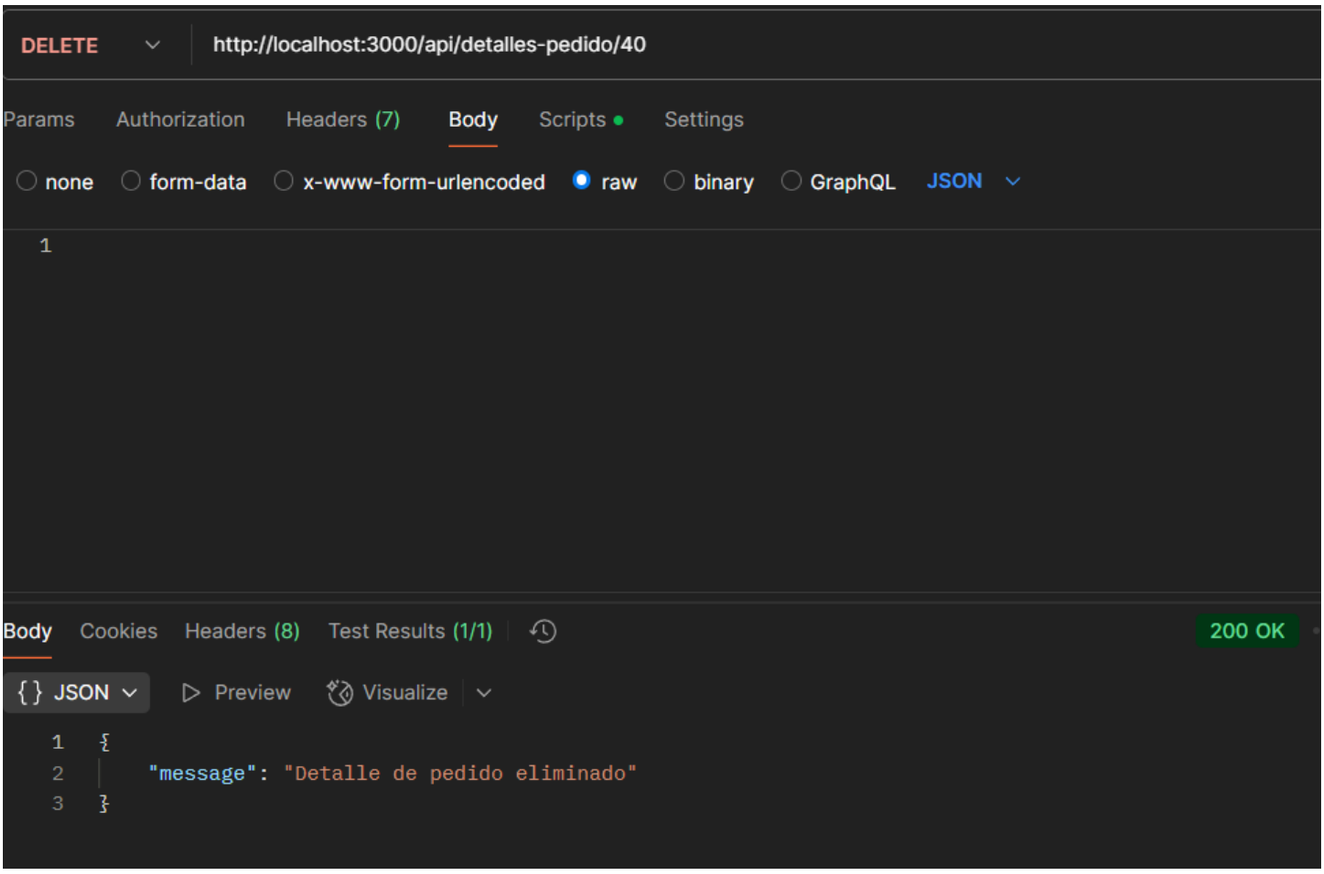
```
// Actualizar un detalle de pedido
async function updateDetallePedido(req, res) {
    const { id } = req.params;
    const { id_pedido, id_prod, cantidad, subtotal } = req.body;
    try {
        const result = await client.query(
            'UPDATE DetallePedido SET id_pedido = $1, id_prod = $2, cantidad = $3, subtotal = $4 WHERE id_detalle = $5 RETURNING *',
            [id_pedido, id_prod, cantidad, subtotal, id]
        );
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Detalle de pedido no encontrado' });
        } else {
            res.status(200).json(result.rows[0]);
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```



DELETE Eliminar DetallePedido:

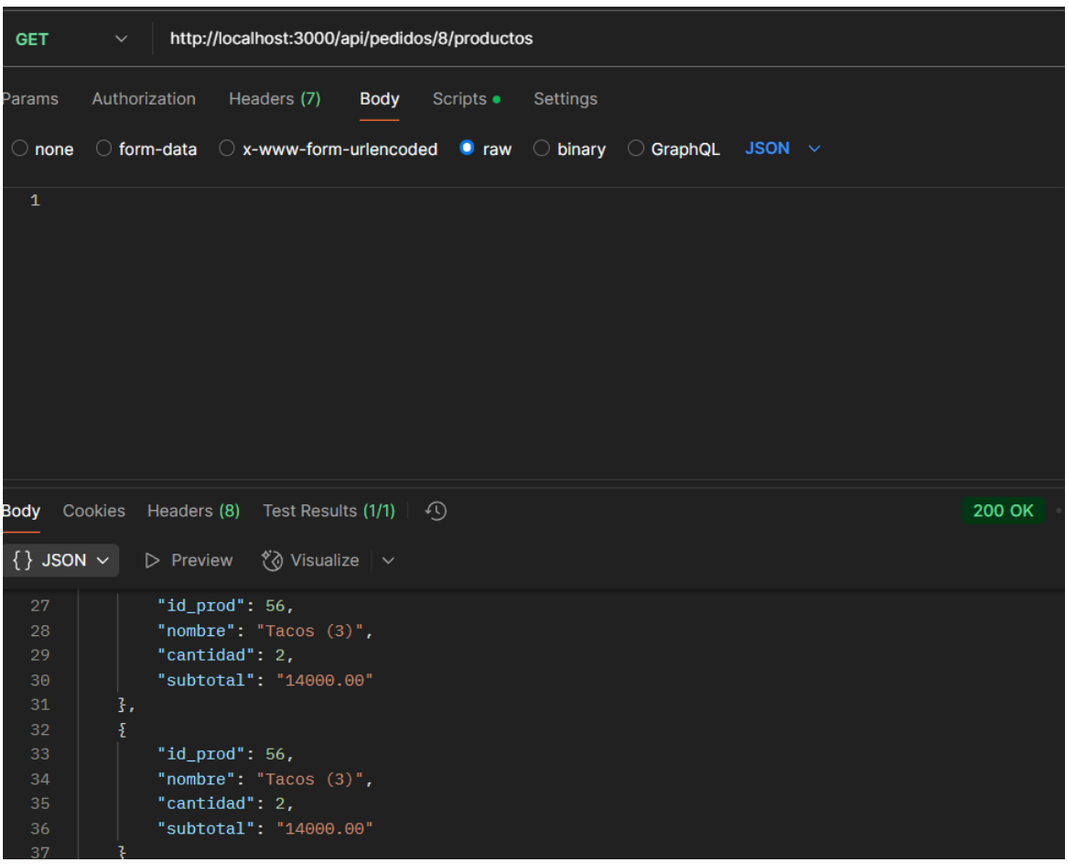
deleteDetallePedido: Este endpoint elimina un detalle de pedido específico usando el id proporcionado en los parámetros de la URL. Si el registro existe y se borra correctamente, devuelve un mensaje de confirmación con estado 200. Si no encuentra el detalle, responde con un 404, y en caso de error interno, retorna un 500 con el mensaje de error correspondiente.

```
// Eliminar un detalle de pedido
async function deleteDetallePedido(req, res) {
    const { id } = req.params;
    try {
        const result = await client.query('DELETE FROM DetallePedido WHERE id_detalle = $1 RETURNING *', [id]);
        if (result.rowCount === 0) {
            res.status(404).json({ message: 'Detalle de pedido no encontrado' });
        } else {
            res.status(200).json({ message: 'Detalle de pedido eliminado' });
        }
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
}
```

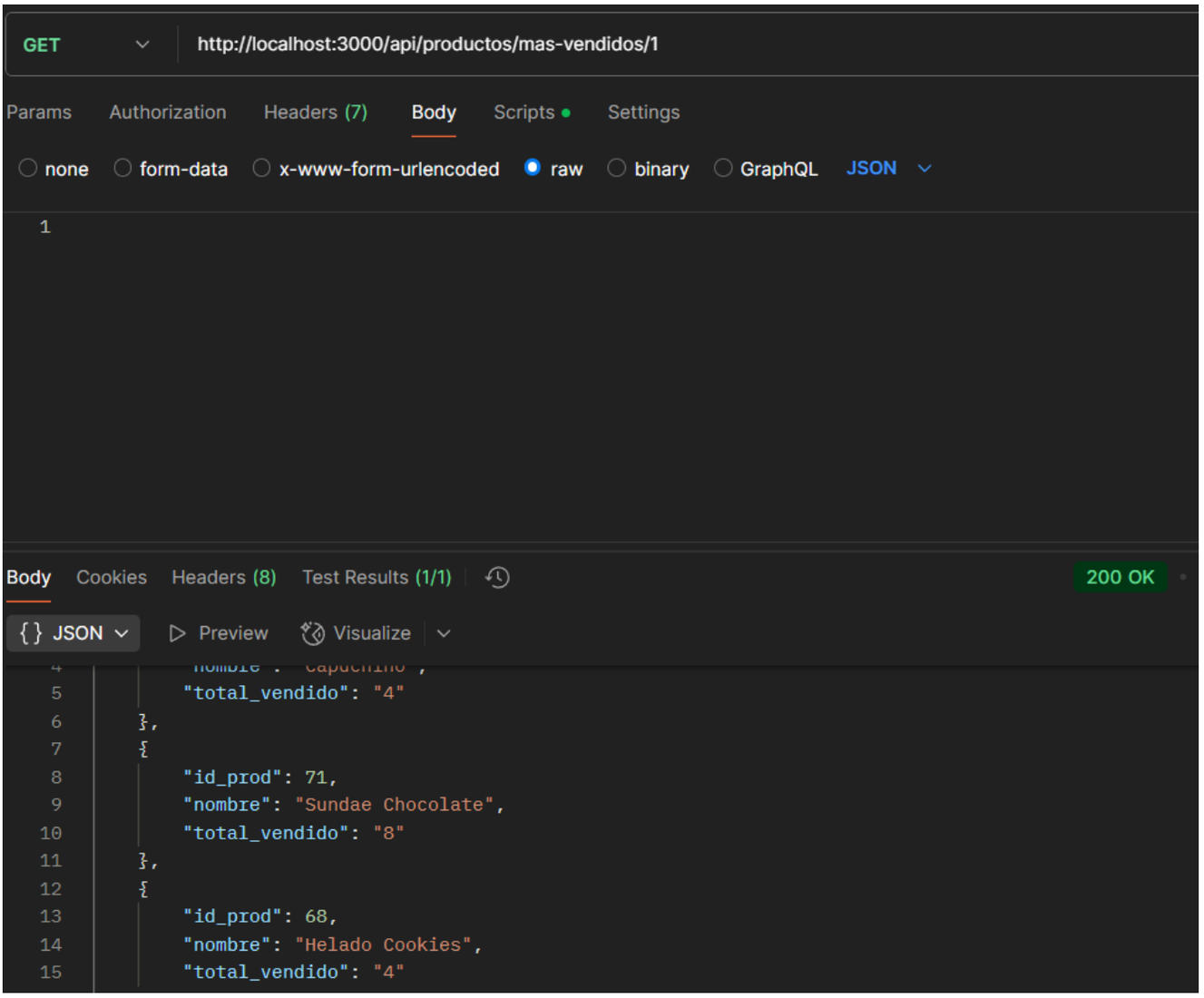


Consultas Nativas:

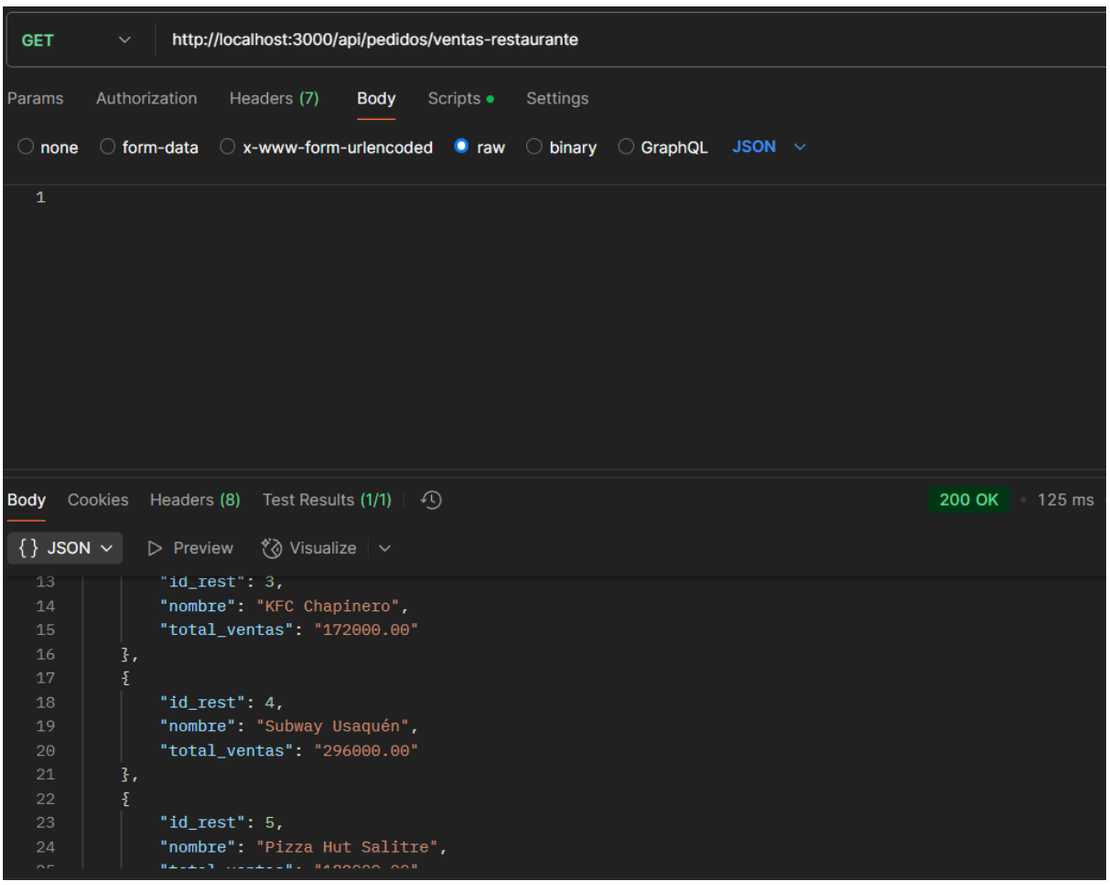
GET Productos por pedido especifico



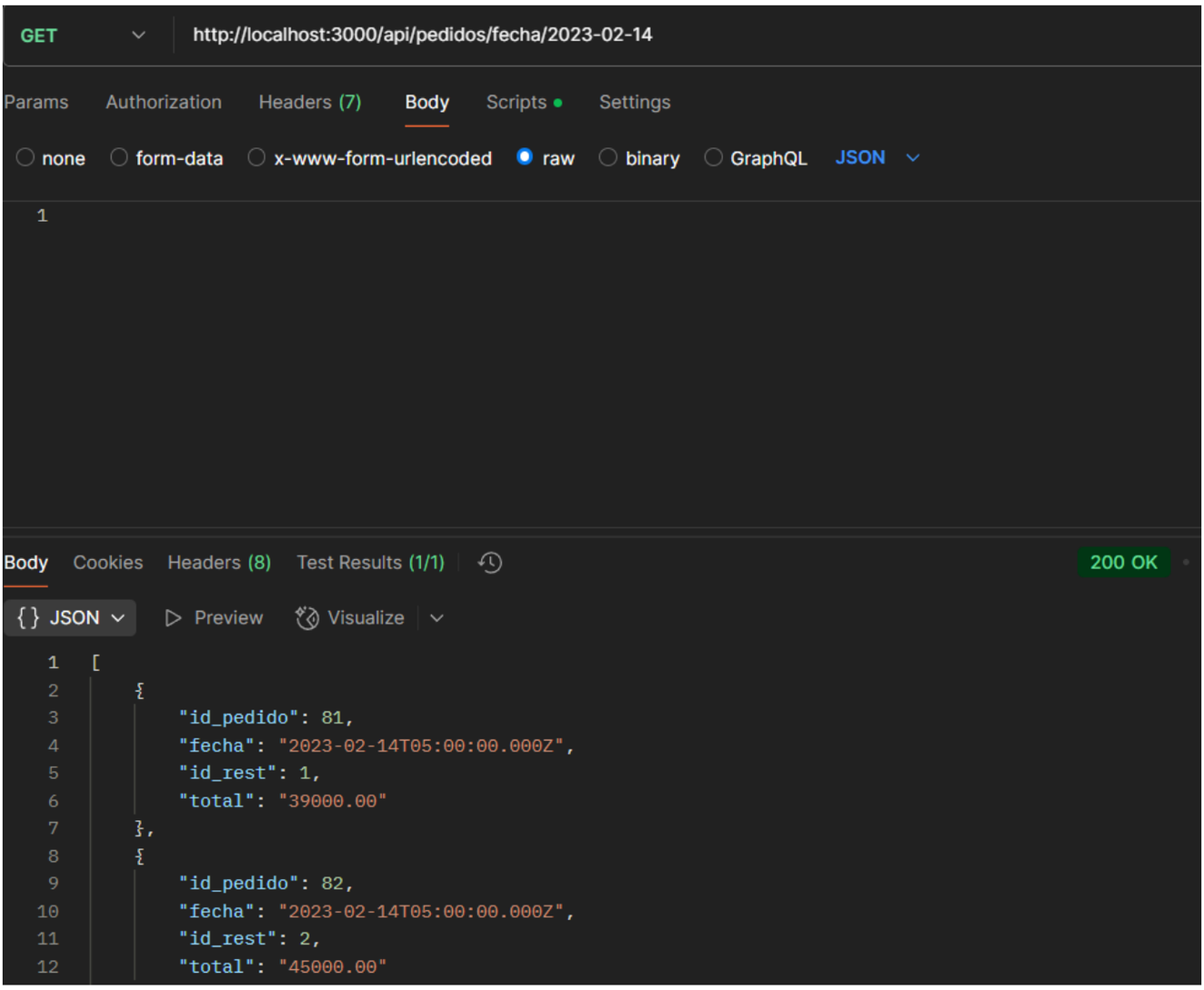
GET Producto mas Vnedido



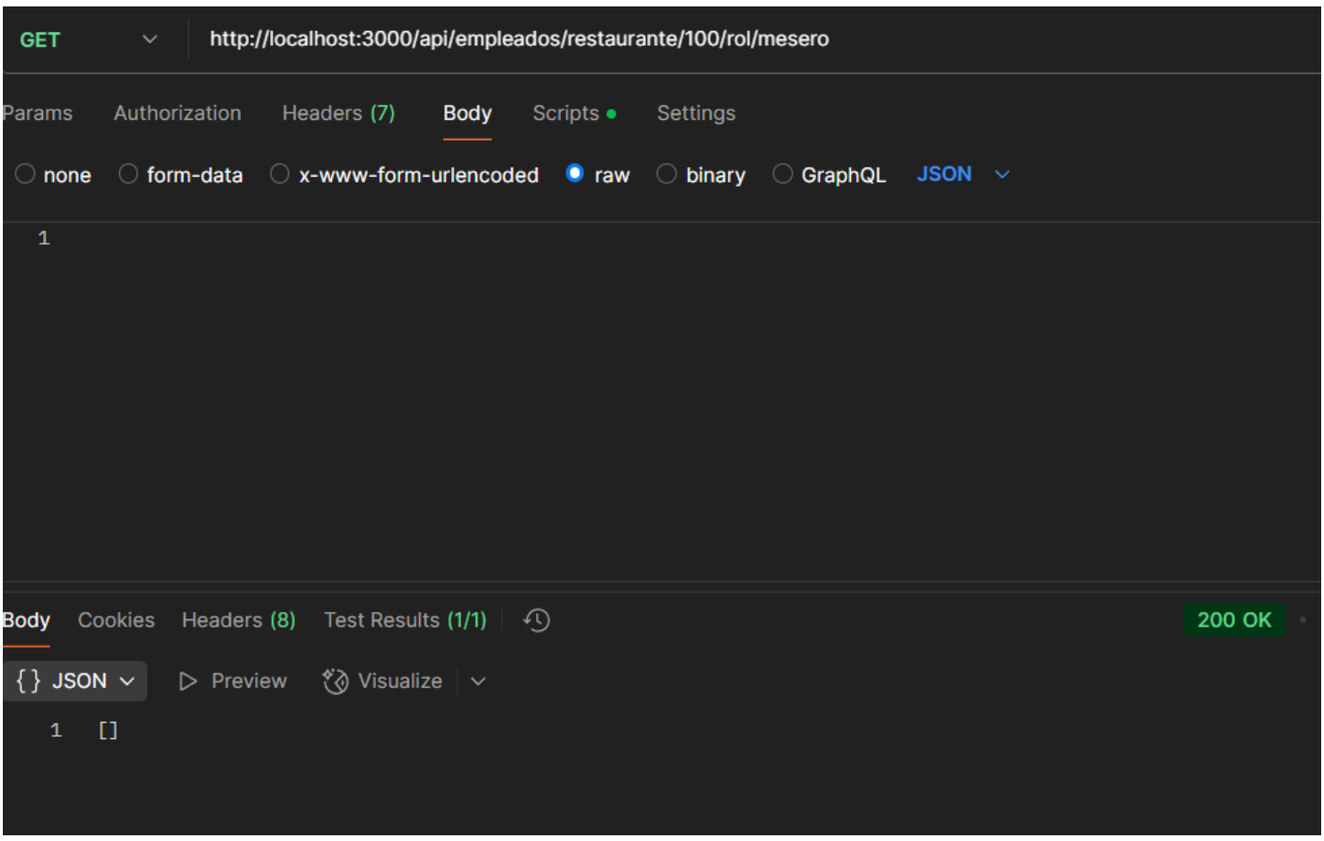
GET Total ventas por restaurante



GET pedido por fecha especifica



GET rol empleado por restaurante



Conclusión

Este proyecto logró desarrollar una API REST funcional y documentada para la gestión de una cadena de comidas rápidas, utilizando tecnologías modernas como Express.js y PostgreSQL. La implementación de operaciones CRUD y consultas específicas permite un control eficiente de la información, facilitando procesos administrativos y análisis de datos.

Mediante Postman, se verificó el correcto funcionamiento de cada endpoint, asegurando que las respuestas sean consistentes y los errores manejados adecuadamente. La conexión con Supabase garantiza un almacenamiento seguro y escalable de los datos.

Como trabajo futuro, se podría integrar un sistema de autenticación (JWT) para mayor seguridad, implementar un dashboard de análisis de datos o migrar a un entorno en la nube para mayor disponibilidad. Este proyecto sienta las bases para un sistema de gestión robusto y adaptable a las necesidades de la empresa.