

Parcial Api Con ExpressJS

Cristhian Andrey Poveda Gaviria

ID: 843183

Corporación Universitaria Minuto de Dios

Base de Datos Masivas

Ingeniería De Sistemas

2025

Contenido

Introducción.....	3
Objetivos.....	3
Desarrollo	3
Configurar entornos de desarrollo	4
Creación de archivos y base de datos	6
Pruebas desde Postman.....	11
Creación y pruebas de consultas nativas	18
Conclusiones.....	23
Referencias	23

Taller Api Con ExpressJS

Introducción

El presente taller presenta el desarrollo de una API REST utilizando el framework Express.js de Node.js. Esta API tiene como objetivo gestionar la información de una cadena de comidas rápidas, incluyendo datos sobre restaurantes, empleados, productos y demás.

Para lograr esto, utilizaremos varias herramientas como PostgreSQL para la gestión de la base de datos, Supabase como plataforma que nos facilita el uso de PostgreSQL, y Postman para realizar pruebas de la API, verificando la correcta implementación de las rutas y la integridad de los datos. Finalmente, utilizaremos GitHub para el posible control de versiones y la colaboración facilitando la entrega del proyecto.

Objetivos

- Desarrollar una API REST funcional que permita realizar operaciones sobre las tablas de la base de datos.
- Implementar la conexión a la base de datos: Establecer una conexión exitosa y funcional con una base de datos PostgreSQL, utilizando Supabase.
- Ejecutar consultas de PostgreSQL, exponiéndolas a través de la API las consultas en el taller, demostrando la capacidad de realizar operaciones directamente en la base de datos.
- Utilizar Postman para realizar pruebas, asegurando su correcto funcionamiento.
- Documentar la API y las consultas: Generar una documentación clara y completa de la API, incluyendo la descripción de las rutas, los parámetros de entrada/salida y el funcionamiento de las consultas nativas.

Desarrollo

Express js es un framework para Node.js que permite desarrollar aplicaciones web y APIs de manera más optimizada y flexible, ofreciendo muchas funcionalidades facilitando la gestión de rutas, solicitudes y otros aspectos de las aplicaciones web.

Por otro lado, una API (Interfaz de Programación de Aplicaciones), es un conjunto de reglas y protocolos que permite que las aplicaciones se comuniquen entre ellas. Definen cómo las solicitudes y respuestas deben formatearse y qué operaciones están disponibles, como puede ser acceder a datos o funcionalidades de otro sistema.

Supabase es un gestor de proyectos de base de datos con la diferencia de que la base de datos PostgreSQL estará alojada en la nube, permitiendo una autenticación, un almacenamiento de archivos, una API autogenerada, y una interfaz web con muchas de las funciones de un gestor de base de datos.

Conectaremos Supabase con apis generadas usando Express para hacer peticiones y operaciones CRUD, para conectarlos, simplemente necesitamos las

credenciales de conexión de la base de datos en Supabase (host, puerto, nombre de la base de datos, usuario y contraseña), que encontramos en el panel de configuración de Supabase.

Configurar entornos de desarrollo

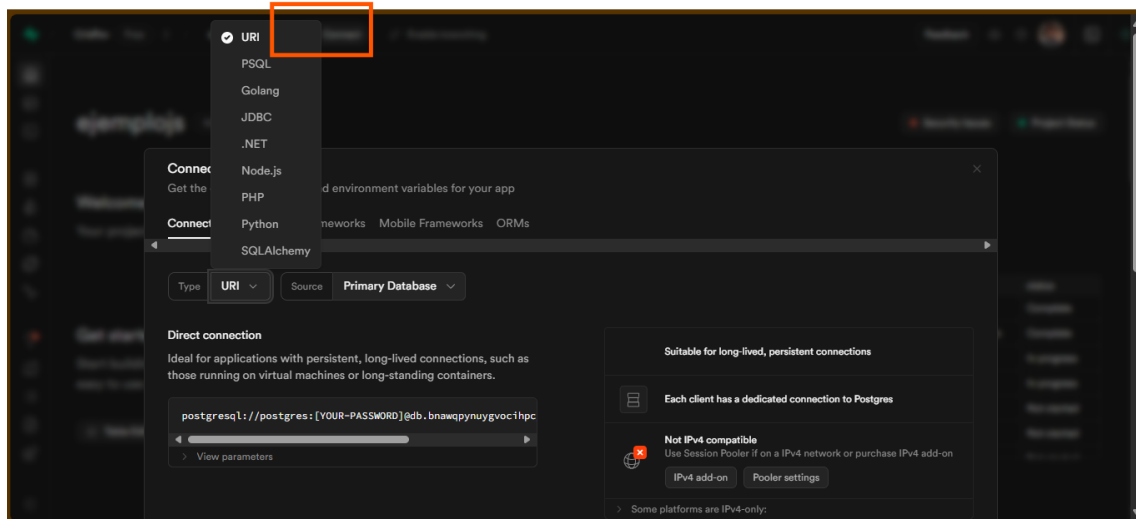
Ingresaremos a *supabase.com* para registrarnos, usando nuestra cuenta de GitHub, luego configuramos nuestra “organización”, escogiendo el plan gratuito, y creamos un proyecto en supabase.

The image displays two screenshots of the Supabase web interface. The top screenshot shows the 'Create a new organization' form. It includes a header 'Create an organization > Create a new project'. The form has a title 'Create a new organization' and a subtitle 'This is your organization within Supabase. For example, you can use the name of your company or department.' The form fields are: 'Name' (CrisPov), 'Type of organization' (Personal), and 'Plan' (Free - \$0/month). There are 'Cancel' and 'Create organization' buttons. The bottom screenshot shows the 'Create a new project' form. It includes a header 'Create a new project' and a subtitle 'Your project will have its own dedicated instance and full Postgres database. An API will be set up so you can easily interact with your new database.' The form fields are: 'Organization' (CrisPov), 'Project name' (First_Project), 'Database Password' (masked), 'Region' (East US (North Virginia)), and a 'Generate a password' link. There are 'Cancel' and 'Create new project' buttons.

❖ *BaseDatosMasivas*

Creamos una posible organización, luego asignamos un nombre al proyecto, y una contraseña para la base de datos.

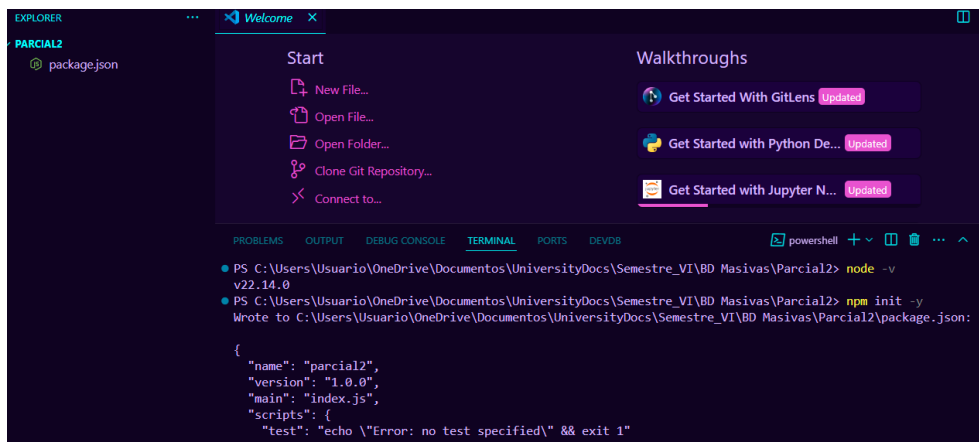
En pantalla, Vamos al apartado donde dice “**connect**”, para ver las direcciones y parámetros necesarias para conectar con esta base de datos en la nube.



- `postgres://postgres:[YOUR-PASSWORD]@db.bnawqpynygvocihpcjx.supabase.co:5432/postgres`

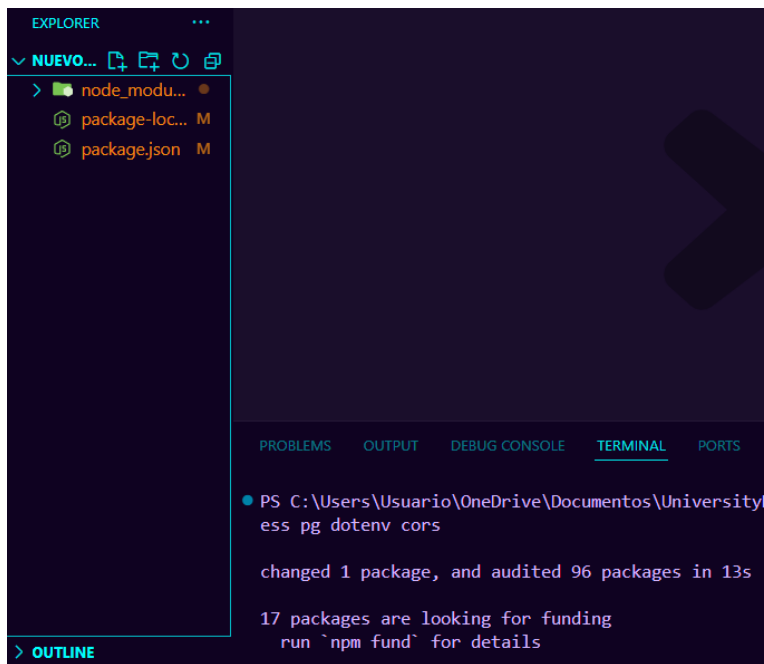
Pero antes, necesitamos tener un entorno configurado para crear las APIs y la conexión. Usaremos Visual Studio.

Abrimos un folder vacío, Inicializamos el proyecto con node (`npm init -y`), y descargaremos las dependencias a usar, a la vez que verificamos el tener instalado el Node.js:



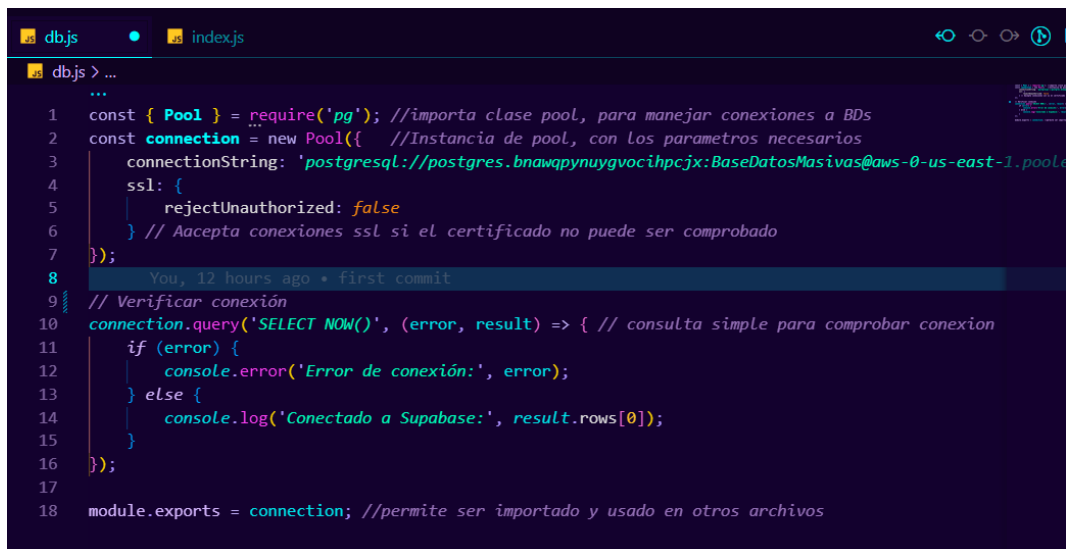
- **Express:** Framework para Node.js que nos facilita la creación de aplicaciones web y APIs.
- **PG:** Cliente para conectarse a bases de datos PostgreSQL desde Node.js.
- **Dotenv:** Permite gestionar las variables de entorno desde un archivo `.env`. Es útil para mantener datos sensibles fuera del código fuente principal.
- **CORS (Cross-Origin Resource Sharing):** Es un *middleware* que nos habilita el intercambio de recursos entre diferentes dominios, lo necesitamos para que el servidor responda a solicitudes de otros sitios web.

Se nos habrán descargado y se nos mostrarán:



Creación de archivos y base de datos

Crearemos dos archivos, el primero, que nos asegurará la conexión:



En este utilizaremos la URI que nos dio supabase, solo reemplazando el apartado de contraseña.

El siguiente archivo será donde crearemos las apis.

```
db.js  index.js M X
index.js > ...
You, 20 seconds ago | 1 author (You)
1  const express = require('express'); //importamos framework express
2  const cors = require('cors');
3  const connection = require('./db'); //importamos la conexion del otro archivo
4  const path = require('path'); //Importamos el modulo que nos permite trabajar con rutas y di
5
6  const app = express(); //instanciamos
7
8  app.use(express.json());
9  app.use(express.urlencoded({ extended: true })); //permite procesar peticiones codificados en URL
10
11  const PORT = 3000; //Puerto opr el que escucha el servidor
12  You, 12 hours ago + first commit
```

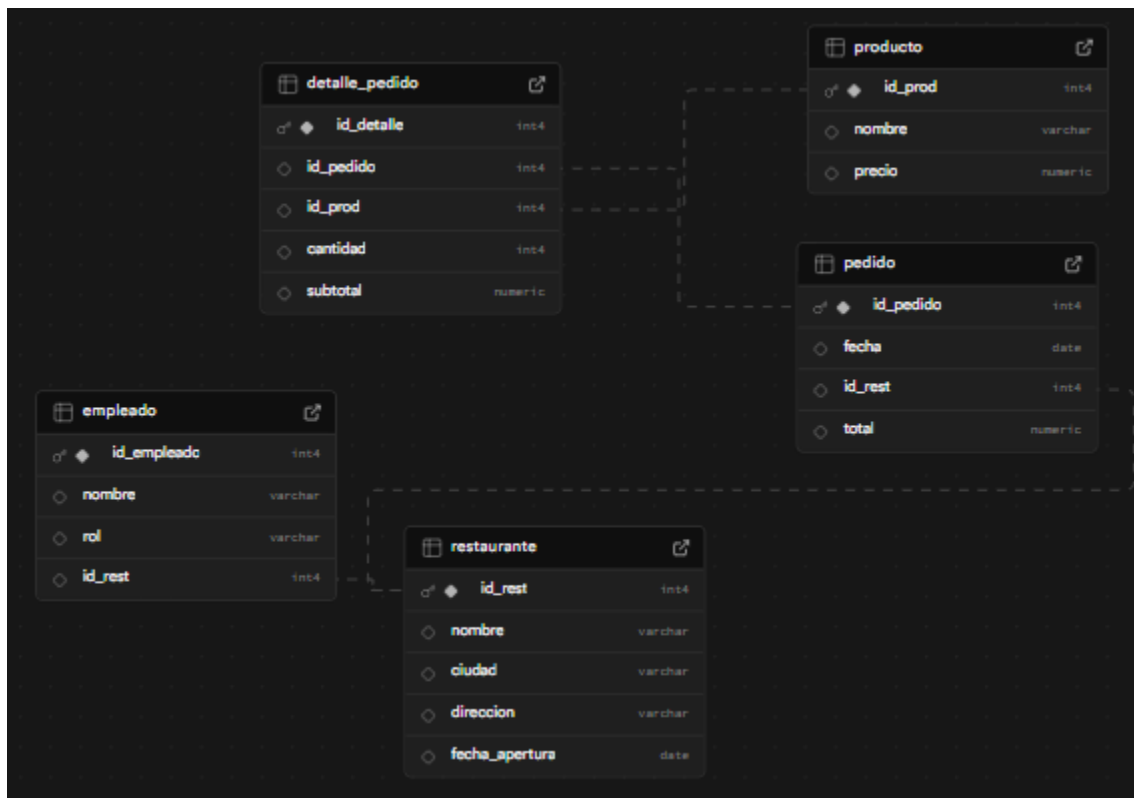
Definimos e importamos las constantes que vamos a utilizar.

Antes de crear las apis, crearemos las tablas en la base de datos en la nube que tenemos en Supabase:

En el apartado SQL editor, en Supabase podemos escribir el código para crearlas, aunque también está el apartado para crearlas visualmente.

```
Connect  Enable branching
1  CREATE TABLE Restaurante (
2    id_rest INT PRIMARY KEY,
3    nombre VARCHAR(100),
4    ciudad VARCHAR(100),
5    direccion VARCHAR(150),
6    fecha_apertura DATE
7  );
8
9  CREATE TABLE Empleado (
10   id_empleado INT PRIMARY KEY,
11   nombre VARCHAR(100),
12   rol VARCHAR(50),
13   id_rest INT,
14   FOREIGN KEY (id_rest) REFERENCES Restaurante(id_rest)
15 );
16
17 CREATE TABLE Producto (
18   id_prod INT PRIMARY KEY,
19   nombre VARCHAR(100),
20   precio NUMERIC(10,2)
21 );
22
Results  Chart  Export
Success. No rows returned

22
23 CREATE TABLE Pedido (
24   id_pedido INT PRIMARY KEY,
25   fecha DATE,
26   id_rest INT,
27   total NUMERIC(10,2),
28   FOREIGN KEY (id_rest) REFERENCES Restaurante(id_rest)
29 );
30
31 CREATE TABLE Detalle_Pedido (
32   id_detalle INT PRIMARY KEY,
33   id_pedido INT,
34   id_prod INT,
35   cantidad INT,
36   subtotal NUMERIC(10,2),
37   FOREIGN KEY (id_pedido) REFERENCES Pedido(id_pedido),
38   FOREIGN KEY (id_prod) REFERENCES Producto(id_prod)
39 );
```



En la visualización nos queda tal que así.

Añadimos los datos:

```

1  -- Insertar 50 productos
2  INSERT INTO producto (id_prod, nombre, precio)
3  \SELECT
4      n,
5      'Producto ' || n,
6      ROUND((RANDOM() * 1000)::numeric, 2)
7  FROM generate_series(1, 50) AS n;
8
9  -- Insertar 52 restaurantes
10 INSERT INTO restaurante (id_rest, nombre, ciudad, direccion, fecha_apertura)
11 \SELECT
12     n,
13     'Restaurante ' || n,
14     (ARRAY['Madrid', 'Barcelona', 'Valencia', 'Sevilla', 'Bilbao'])[1 + mod(n, 5)],
15     'Calle Principal ' || n,
16     CURRENT_DATE - (RANDOM() * 1000)::integer
17 FROM generate_series(1, 52) AS n;
18
19 -- Insertar 54 empleados
20 INSERT INTO empleado (id_empleado, nombre, rol, id_rest)
21 \SELECT
22     n,

```

Results | Chart | Export

Success. No rows returned

0 row

-- Insertar 50 productos

INSERT INTO producto (id_prod, nombre, precio)

SELECT


```

n,
'Producto ' || n,
ROUND((RANDOM() * 1000)::numeric, 2)
FROM generate_series(1, 50) AS n;

```

-- Insertar 52 restaurantes

```

INSERT INTO restaurante (id_rest, nombre, ciudad, direccion, fecha_apertura)

```

```

SELECT

```

```

n,
'Restaurante ' || n,
(ARRAY['Madrid', 'Barcelona', 'Valencia', 'Sevilla', 'Bilbao'])[1 + mod(n, 5)],
'Calle Principal ' || n,
CURRENT_DATE - (RANDOM() * 1000)::integer
FROM generate_series(1, 52) AS n;

```

-- Insertar 54 empleados

```

INSERT INTO empleado (id_empleado, nombre, rol, id_rest)

```

```

SELECT

```

```

n,
'Empleado ' || n,
(ARRAY['Camarero', 'Cocinero', 'Gerente', 'Recepcionista', 'Limpieza'])[1 + mod(n, 5)],
1 + mod(n, 52) -- Para asegurarnos que id_rest existe
FROM generate_series(1, 54) AS n;

```

-- Insertar 56 pedidos

```

INSERT INTO pedido (id_pedido, fecha, id_rest, total)

```

```

SELECT

```

```

n,
CURRENT_DATE - (RANDOM() * 365)::integer,
1 + mod(n, 52), -- Para asegurarnos que id_rest existe

```

```

0 -- El total será actualizado después
FROM generate_series(1, 56) AS n;

-- Insertar 58 detalles de pedido
INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal)
SELECT
    n,
    1 + mod(n, 56), -- Para asegurarnos que id_pedido existe
    1 + mod(n, 50), -- Para asegurarnos que id_prod existe
    1 + (RANDOM() * 5)::integer,
    0 -- El subtotal será calculado después
FROM generate_series(1, 58) AS n;

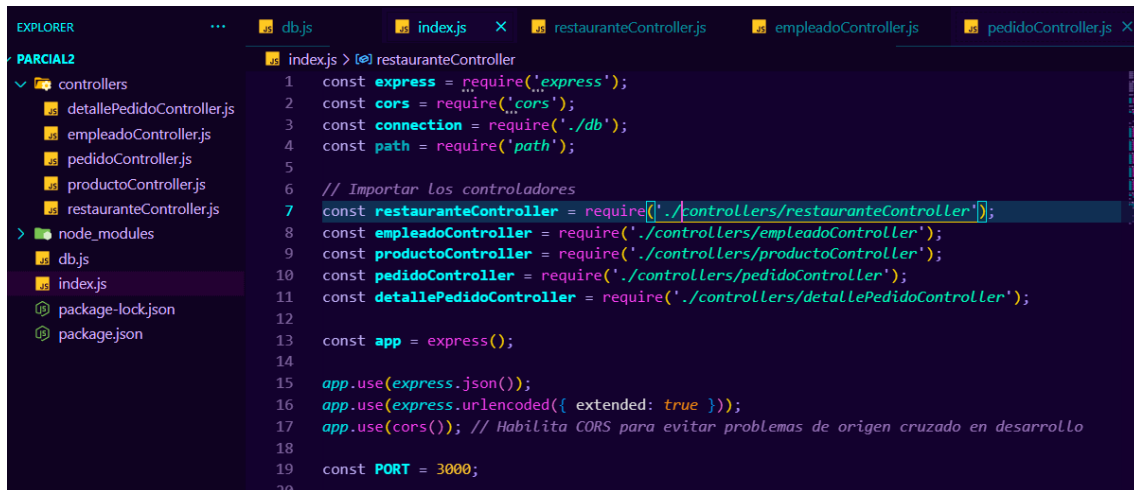
-- Actualizar los subtotales en detalle_pedido
UPDATE detalle_pedido
SET subtotal = dp.cantidad * p.precio
FROM detalle_pedido dp
JOIN producto p ON dp.id_prod = p.id_prod;

-- Actualizar los totales en pedido
UPDATE pedido p
SET total = (
    SELECT SUM(subtotal)
    FROM detalle_pedido dp
    WHERE dp.id_pedido = p.id_pedido
);

```

- El anterior código para agregar los datos fue proporcionado por Claude.ai <https://claude.ai/chat>

Organizamos las carpetas del proyecto de la siguiente manera:



The screenshot shows the VS Code interface. On the left, the Explorer sidebar displays the project structure under 'PARCIAL2':

- controllers
 - detallePedidoController.js
 - empleadoController.js
 - pedidoController.js
 - productoController.js
 - restauranteController.js
- node_modules
- db.js
- index.js
- package-lock.json
- package.json

The main editor shows the content of `index.js`:

```
1 const express = require('express');
2 const cors = require('cors');
3 const connection = require('./db');
4 const path = require('path');
5
6 // Importar Los controladores
7 const restauranteController = require('./controllers/restauranteController');
8 const empleadoController = require('./controllers/empleadoController');
9 const productoController = require('./controllers/productoController');
10 const pedidoController = require('./controllers/pedidoController');
11 const detallePedidoController = require('./controllers/detallePedidoController');
12
13 const app = express();
14
15 app.use(express.json());
16 app.use(express.urlencoded({ extended: true }));
17 app.use(cors()); // Habilita CORS para evitar problemas de origen cruzado en desarrollo
18
19 const PORT = 3000;
```

Definiendo un archivo de controlador a cada tabla.

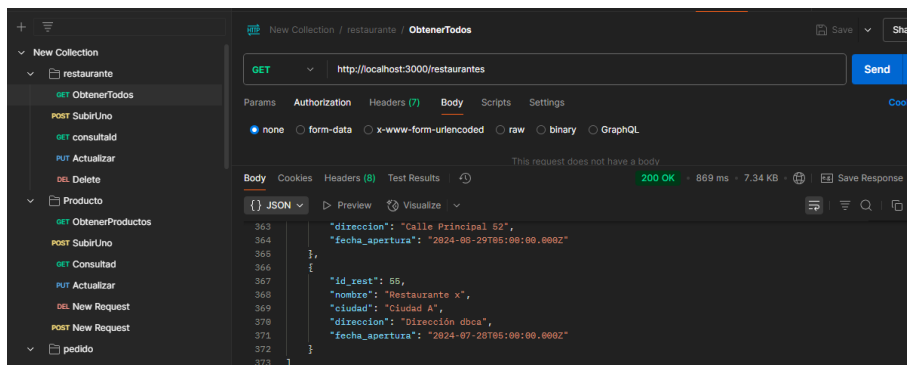
- Cada archivo contiene las operaciones CRUD de su respectiva tabla.
- En el archivo `index.js` importamos los controladores, y luego definimos las rutas para cada uno de sus métodos (CRUD) de cada controlador (tabla).

Pruebas desde Postman

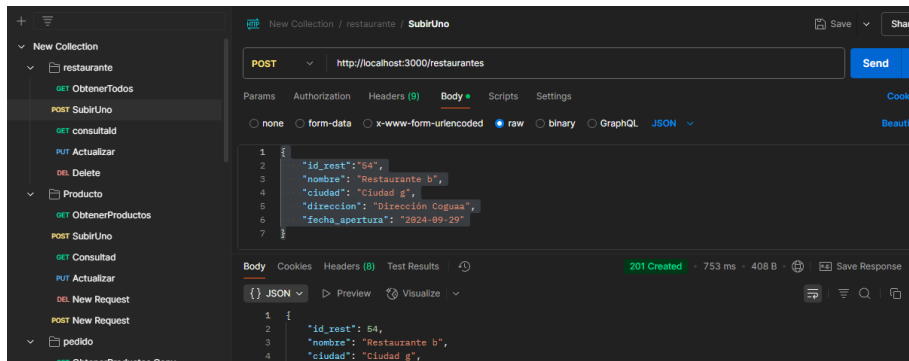
Ahora, en Postman Crearemos una colección de pruebas de cada una de las apis.

En restaurantes:

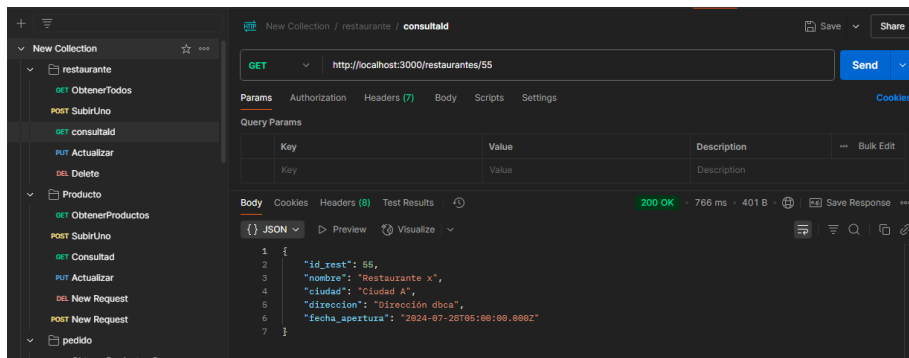
1. Para obtener todos los datos



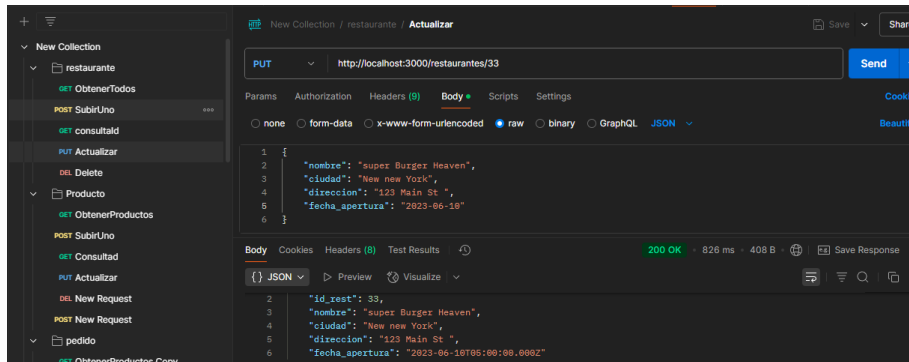
2. Para Subir dato



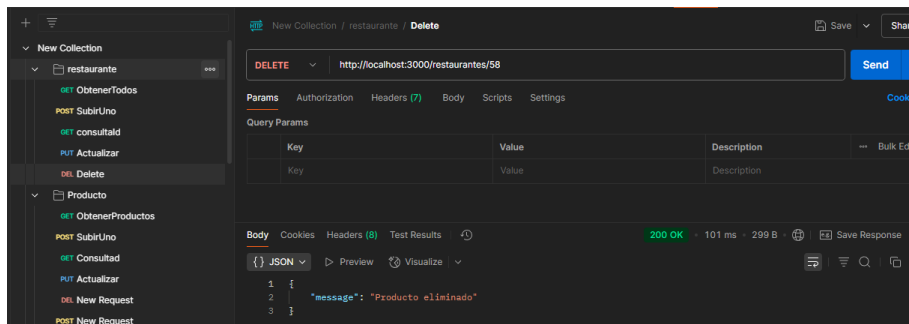
3. Para obtener por ID



4. Para actualizar por ID

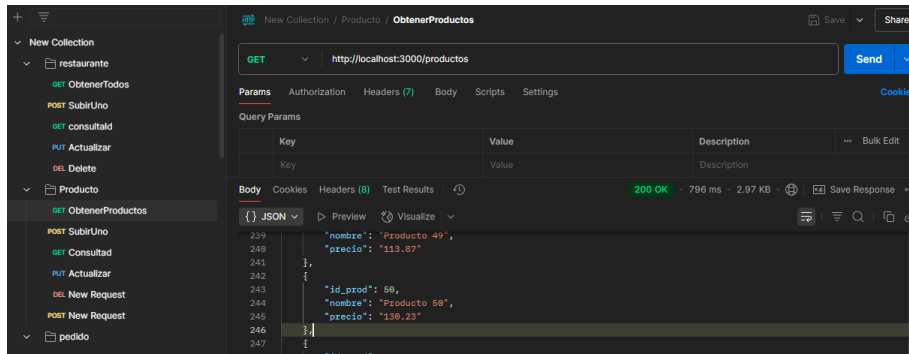


5. Para Eliminar por ID

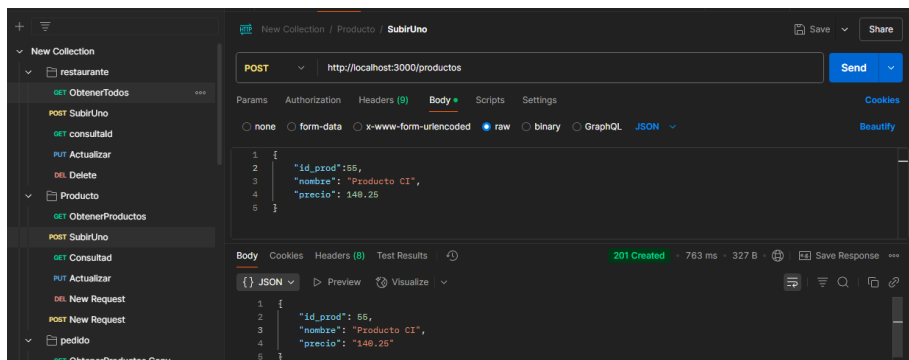


En Productos:

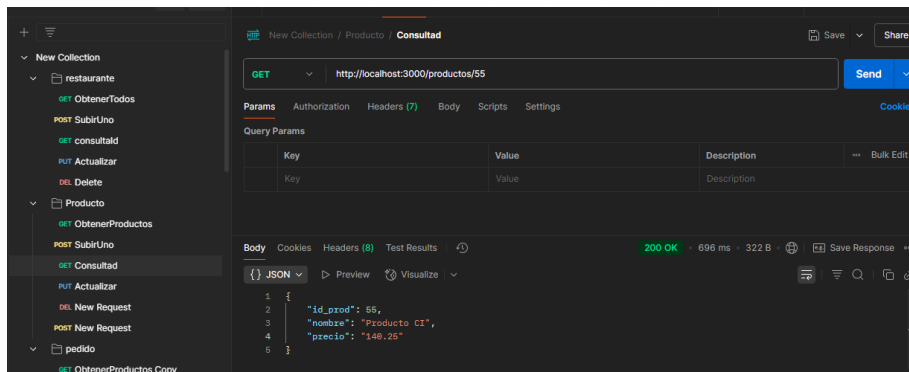
1. Para obtener todos los datos



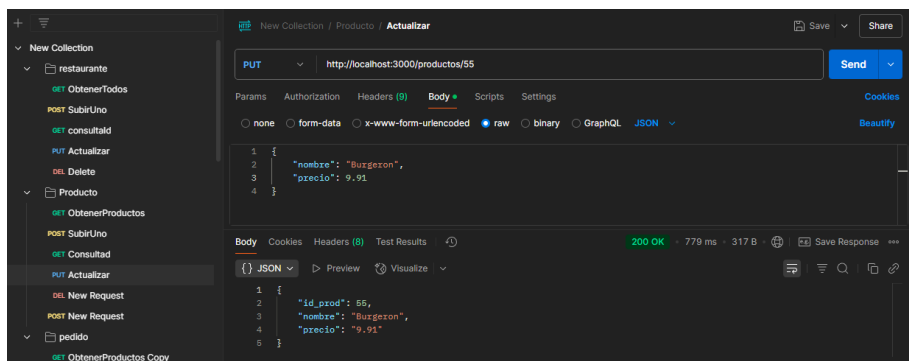
2. Para Subir dato



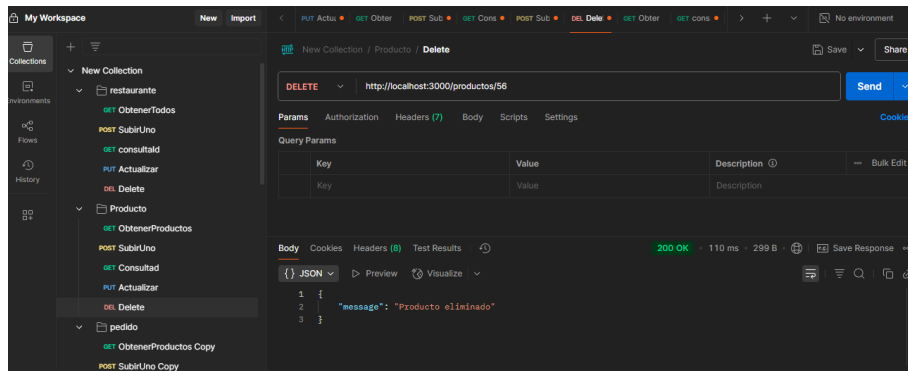
3. Para obtener por ID



4. Para actualizar por ID

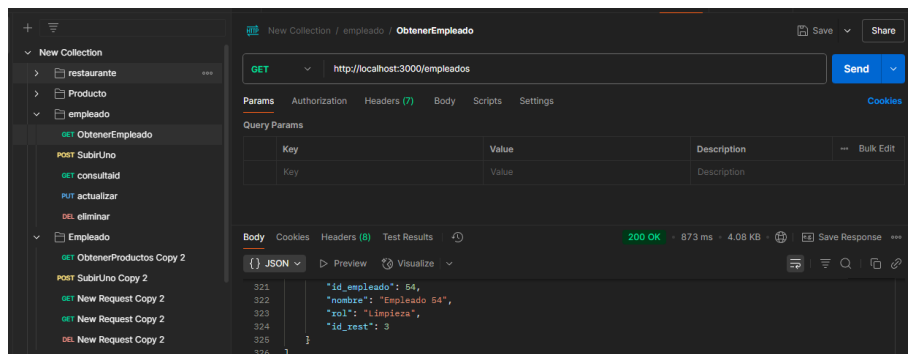


5. Para Eliminar por ID

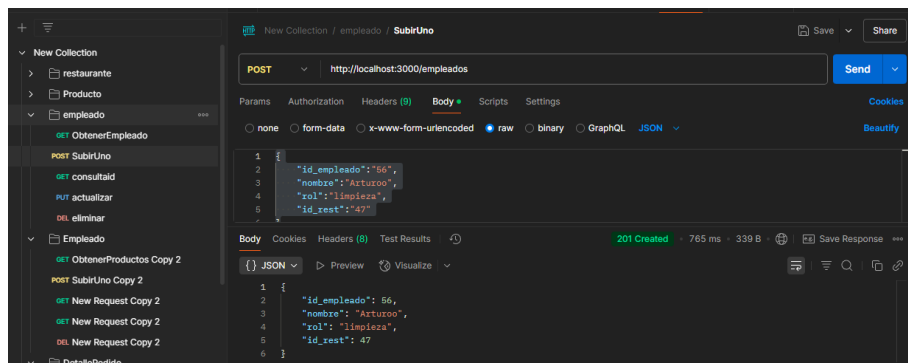


En Empleados:

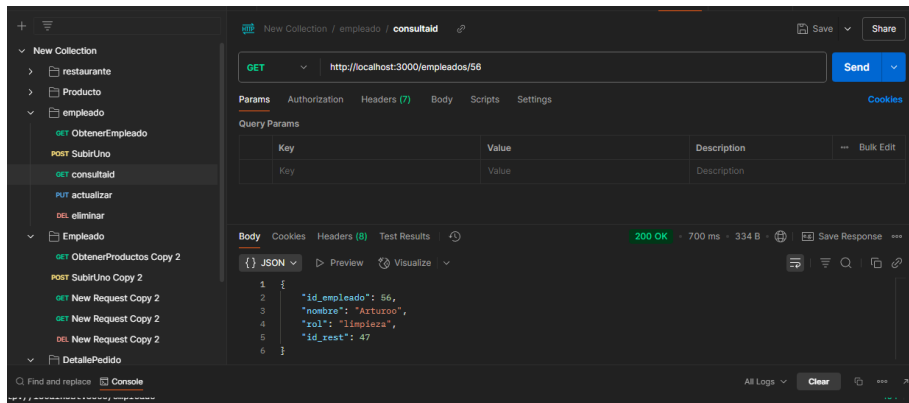
1. Para obtener todos los datos



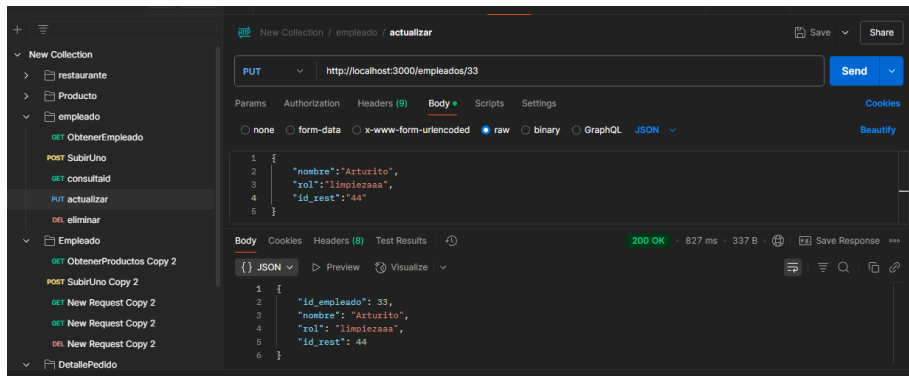
2. Para Subir dato



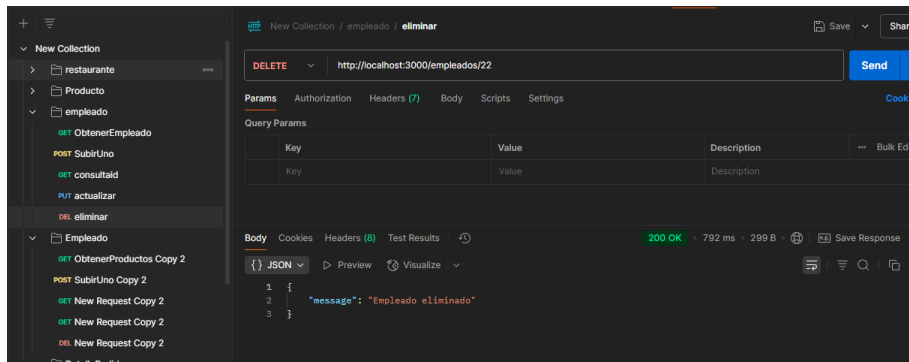
3. Para obtener por ID



4. Para actualizar por ID

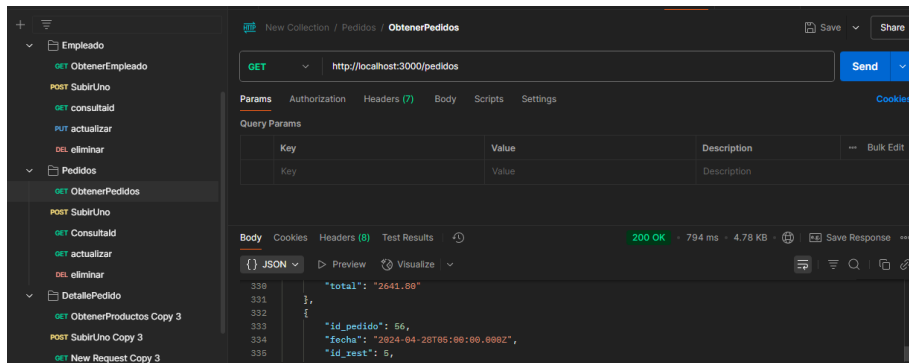


5. Para Eliminar por ID

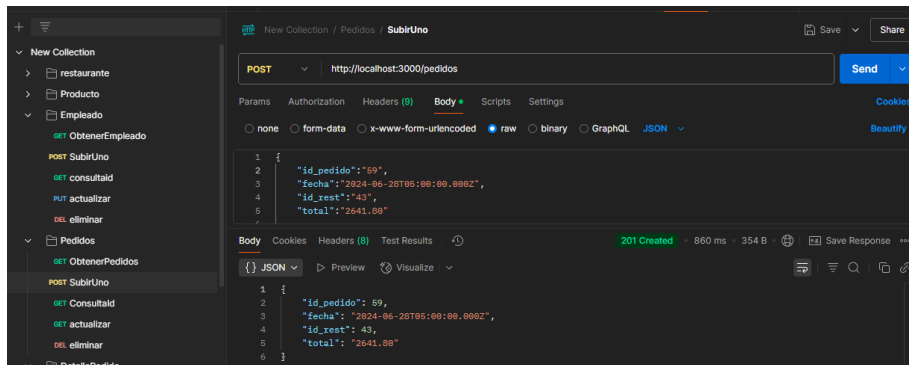


En Pedidos:

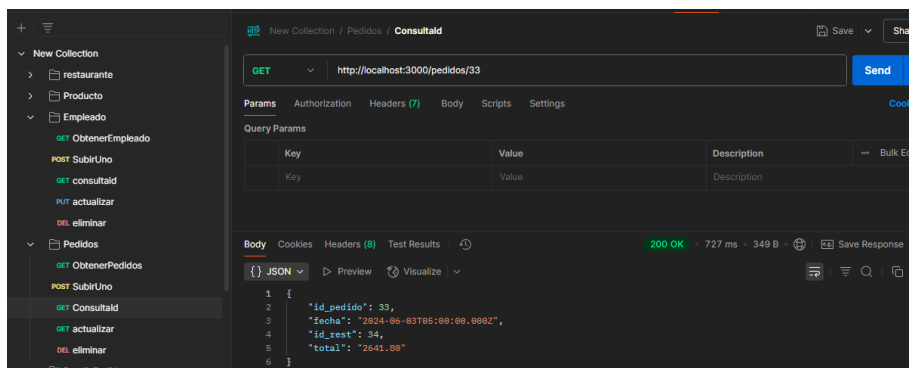
1. Para obtener todos los datos



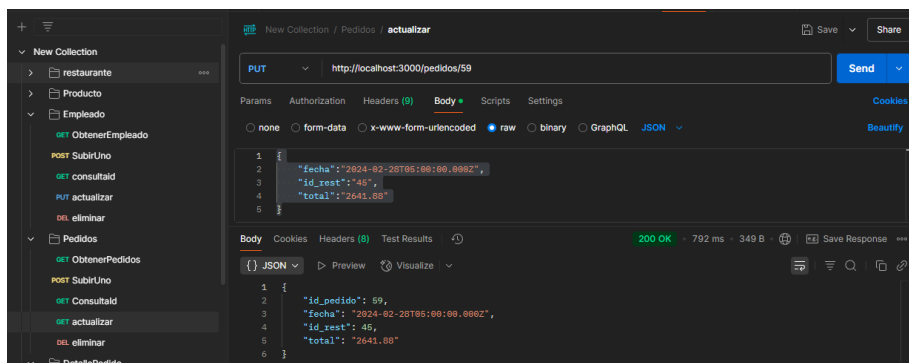
2. Para Subir dato



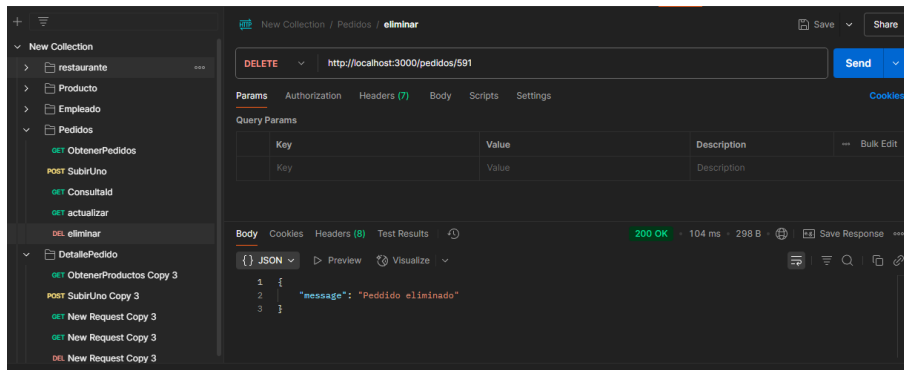
3. Para obtener por ID



4. Para actualizar por ID

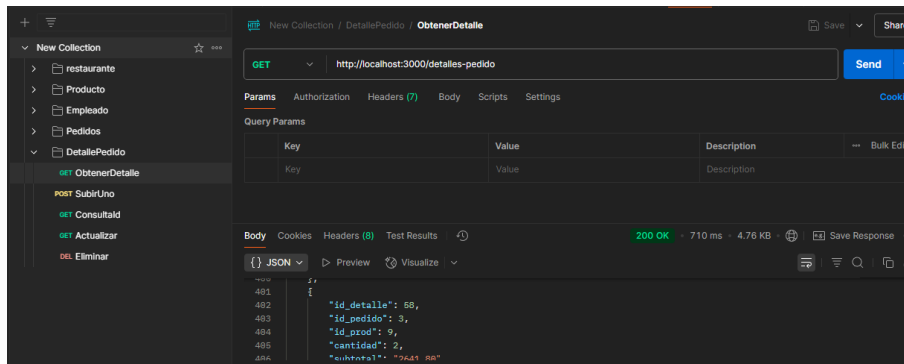


5. Para Eliminar por ID

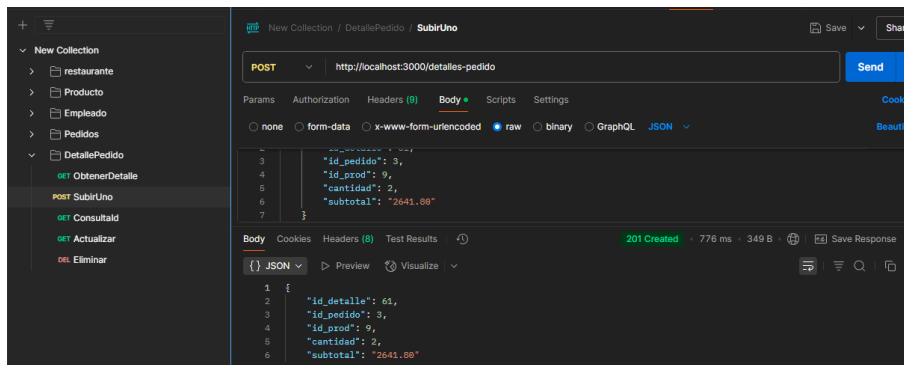


En Detalles pedido:

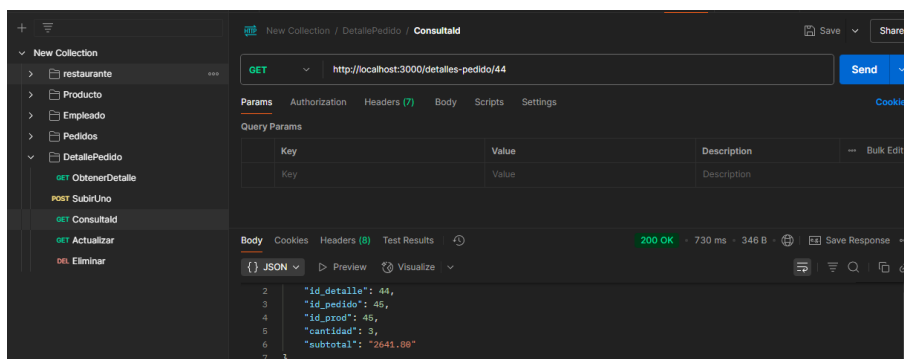
1. Para obtener todos los datos



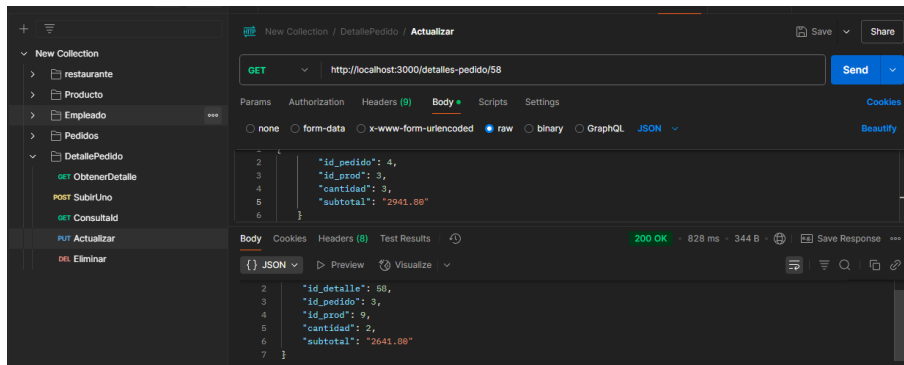
2. Para Subir dato



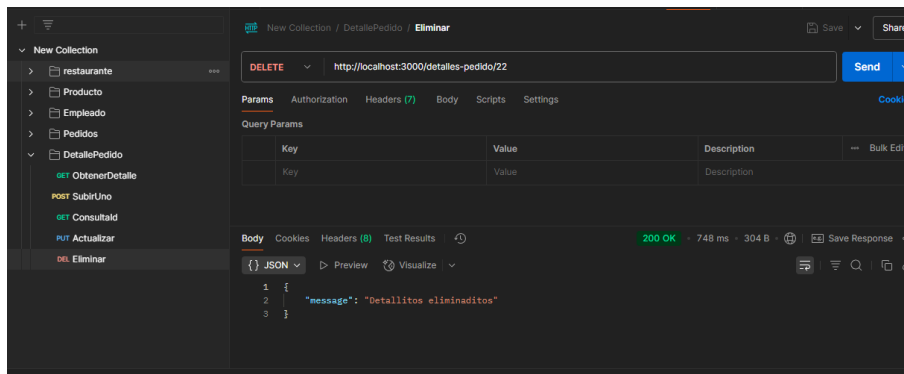
3. Para obtener por ID



4. Para actualizar por ID



5. Para Eliminar por ID



Creación y pruebas de consultas nativas

Las consultas nativas irán dentro de nuestro archivo `Index.js`, ya que no lo extenderán tanto.

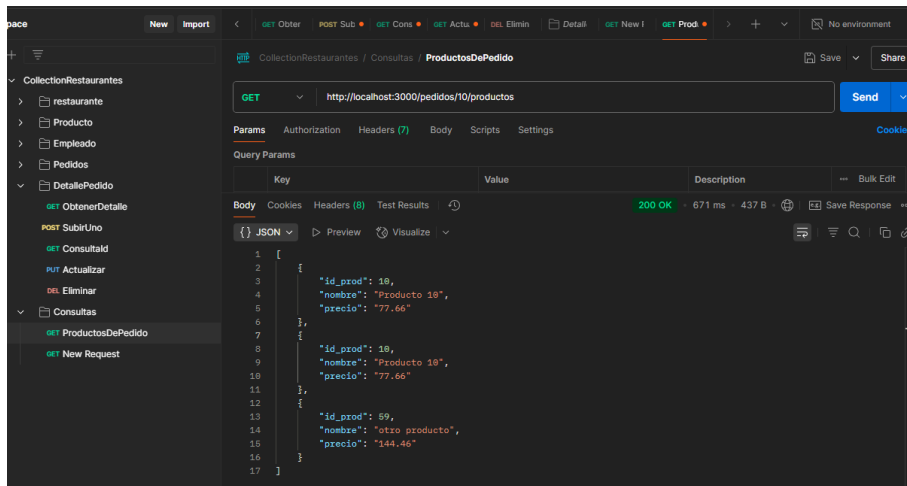
1. Obtener todos los productos de un pedido específico

Creamos la estructura de la api desde el archivo `Index`, con su respectivo Try Catch, dentro irá la consulta que nos devolverá los productos relacionados a un pedido específico (`id_pedido`).

“SELECT p. FROM Producto p JOIN Detalle_Pedido dp ON p.id_prod = dp.id_prod WHERE dp.id_pedido = \$1”*

```
70 // 1. Obtener todos los productos de un pedido específico
71 app.get('/pedidos/:id_pedido/productos', async (req, res) => {
72   const idPedido = req.params.id_pedido; //recibimos el id en la URL
73   try {
74     const result = await connection.query(`//creamos, ejecutamos y guardamos la query y su resultado
75     `SELECT p.*
76     FROM Producto p
77     JOIN Detalle_Pedido dp ON p.id_prod = dp.id_prod
78     WHERE dp.id_pedido = $1`,
79     [idPedido]
80     ); //Seleccionamos los productos que se relacionan a cierto pedido
81     res.json(result.rows); //respondemos con el resultado
82   } catch (error) {
83     console.error(error); //mostramos el error en consola y el mensaje en la petición
84     res.status(500).json({ error: 'Error al obtener los productos del pedido' });
85   }
86 });
```

Prueba en Postman:



2. Obtener los productos más vendidos (más de X unidades)

Igualmente creamos la estructura de la Api desde el archivo Index, con su respectivo Try Catch. Dentro irá la consulta que nos devolverá los productos con más de X unidades vendidas, el cual es un parámetro que recibe en la petición.

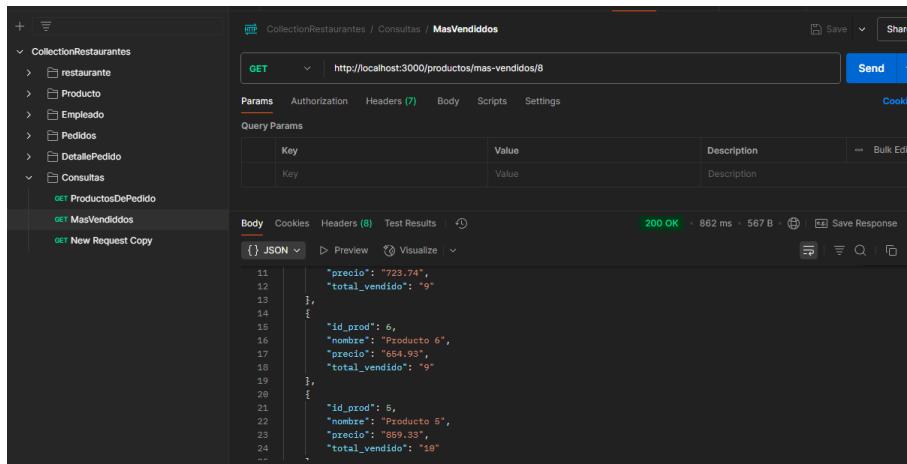
SELECT p., SUM(dp.cantidad) as total_vendido FROM Producto p JOIN Detalle_Pedido dp ON p.id_prod = dp.id_prod GROUP BY p.id_prod HAVING SUM(dp.cantidad) > \$1;*

```

89 // 2. Obtener los productos más vendidos (más de X unidades)
90 app.get('/productos/mas-vendidos/:cantidad_minima', async (req, res) => { //en la petición se especifica la cantidad
91   const cantidadMinima = parseInt(req.params.cantidad_minima); //convertimos el dato recibido a entero
92   try {
93     const result = await connection.query( //creamos, ejecutamos y guardamos la query y su resultado
94       `SELECT p.*, SUM(dp.cantidad) as total_vendido
95        FROM Producto p
96        JOIN Detalle_Pedido dp ON p.id_prod = dp.id_prod
97        GROUP BY p.id_prod
98        HAVING SUM(dp.cantidad) > $1;`,
99       [cantidadMinima]
100     ); //seleccionamos los productos con mas cantidad vendida, mayor a x
101     res.json(result.rows); //respondemos con el resultado
102   } catch (error) {
103     console.error(error); //mostramos el error en consola y el mensaje en la petición
104     res.status(500).json({ error: 'Error al obtener los productos más vendidos' });
105   }

```

Prueba en Postman:



- Nos devuelve los productos con más de 8 (X) ventas, en este ejemplo.

3. Obtener el total de ventas por restaurante.

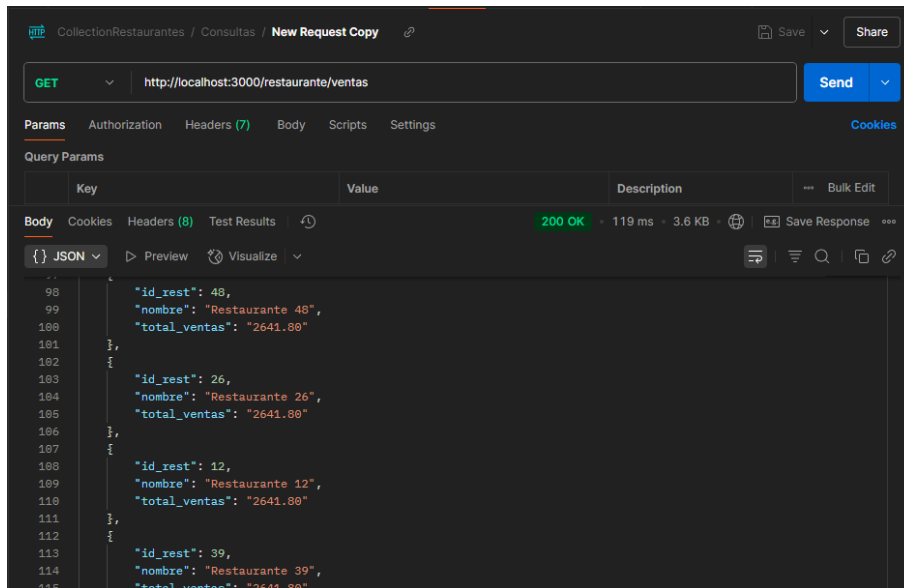
Creamos la estructura de la api desde el archivo Index, con su respectivo Try Catch. Dentro irá la consulta que nos devolverá todos los restaurantes, su id, y sus ventas acumuladas.

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;

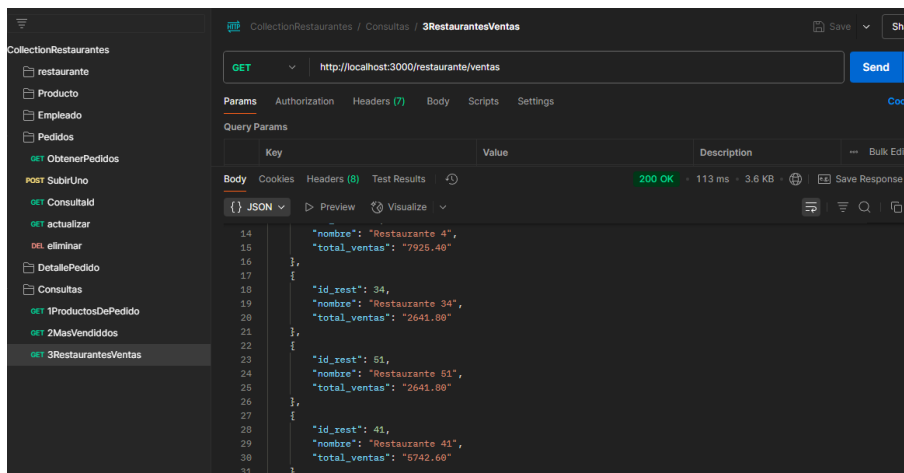
```

108 // 3. Obtener el total de ventas por restaurante
109 app.get('/restaurante/ventas', async (req, res) => { //definimos la ruta
110   try {
111     const result = await connection.query( //creamos, ejecutamos y guardamos la query y su resultado
112       `SELECT r.id_rest, r.nombre, SUM(p.total) as total_ventas
113        FROM Restaurante r
114        JOIN Pedido p ON r.id_rest = p.id_rest
115        GROUP BY r.id_rest, r.nombre;`
116     ); //consultamos los restaurantes con sus ventas, sumando los valores de sus pedidos
117     res.json(result.rows); //respondemos con el resultado
118   } catch (error) {
119     console.error(error); //mostramos el error en consola y el mensaje en la petición
120     res.status(500).json({ error: 'Error al obtener el total de ventas por restaurante' });
121   }
122 });
  
```

Prueba en Postman:



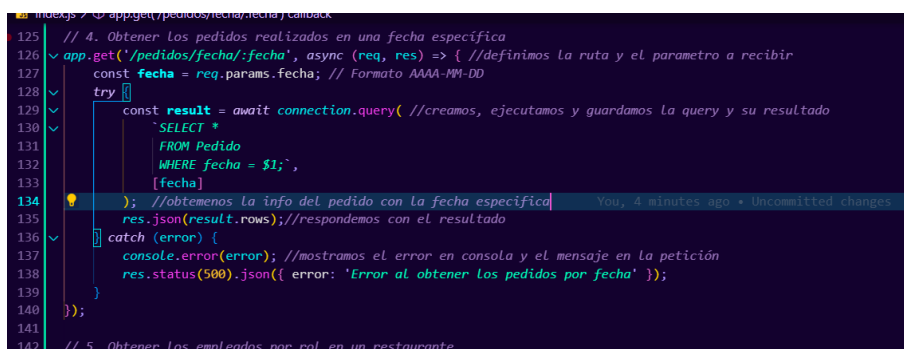
- Obtenemos el listado de los restaurantes con sus ventas.



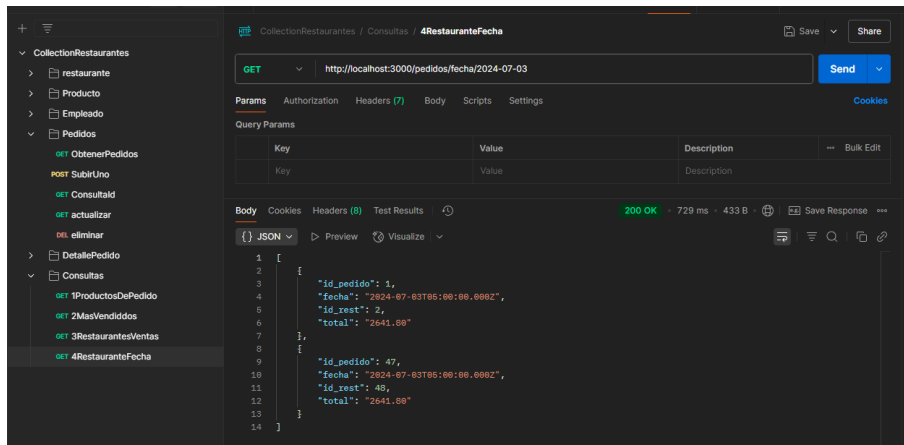
4. Obtener los pedidos realizados en una fecha específica

Creamos la estructura de la api desde el archivo Index, con su respectivo Try Catch. Dentro irá la consulta que nos devolverá los pedidos realizados en la fecha especificada en la petición.

*SELECT * FROM Pedido WHERE fecha = \$1;*



Prueba en Postman:



- Obtenemos los pedidos con la fecha especificada (2024-07-03).

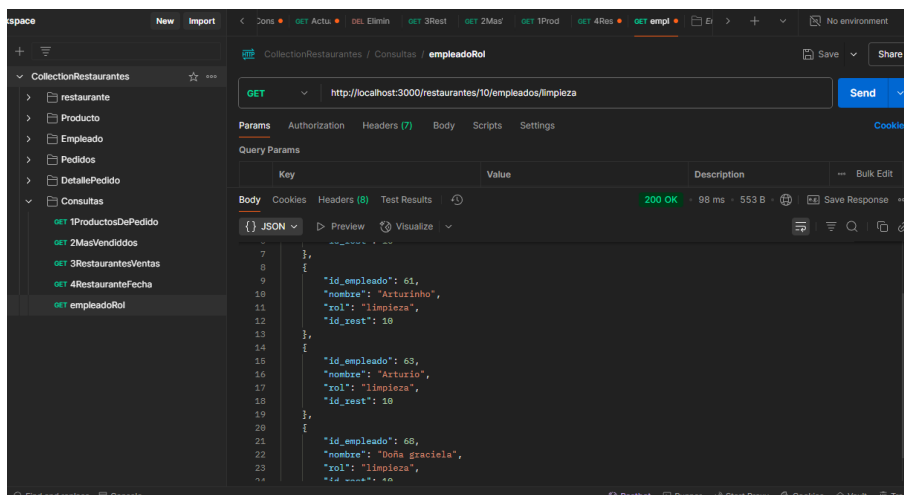
5. Obtener los empleados por rol en un restaurante

Creamos la estructura de la Api desde el archivo Index, con su respectivo Try Catch. Dentro irá la consulta que nos devolverá los datos de los empleados del restaurante y del rol especificados en la petición.

*SELECT * FROM Empleado WHERE id_rest = \$1 AND rol = \$2;*

```
142 // 5. Obtener Los empleados por rol en un restaurante
143 app.get('/restaurantes/:id_rest/empleados/:rol', async (req, res) => { //recibidos tanto restaurante como rol
144   const idRest = req.params.id_rest;
145   const rol = req.params.rol; //guardamos los parametros recibidos
146   try {
147     const result = await connection.query( //creamos, ejecutamos y guardamos la query y su resultado
148       `SELECT *
149        FROM Empleado
150        WHERE id_rest = $1 AND rol = $2;`,
151       [idRest, rol]
152     ); //obtenemos los datos de los empleados de cierto restaurante y con cierto Rol
153     res.json(result.rows); //respondemos con el resultado
154   } catch (error) {
155     console.error(error); //mostramos el error en consola y el mensaje en la petición
156     res.status(500).json({ error: 'Error al obtener los empleados por rol' });
157   }
158 }
```

Prueba en Postman



- Obtenemos los empleados del restaurante “10” del área “Limpieza”

Conclusiones

En este taller, se desarrolló una API REST utilizando Express.js para gestionar la información de una cadena restaurantes, desde la configuración del entorno con Supabase y PostgreSQL hasta la implementación de rutas CRUD y consultas nativas. El proceso permitió mostrar y exponer los conocimientos sobre el desarrollo de APIs, la interacción con bases de datos y el uso de herramientas como Postman para las pruebas.

Referencias

Aplyca. (2023, febrero 28). *Supabase: una alternativa ágil de código abierto*.

Recuperado de <https://www.aplyca.com/blog/blog-supabase-una-alternativa-agil-de-codigo-abierto>

Generación de datos con: <https://claude.ai/chat/>