

Parcial No. 2 – “Comidas Rápidas”

Andrés Felipe Cholo González

ID: 000832608

Base De Datos Masivas

Ing. William Alexander Matallana Porras

NRC: 10-60747

Escuela De Ingeniería, Ingeniería De Sistemas

Sexto Semestre, Segundo Corte

Corporación Universitaria Minuto De Dios, Sede Zipaquirá

Viernes, 25 de Abril, 2025

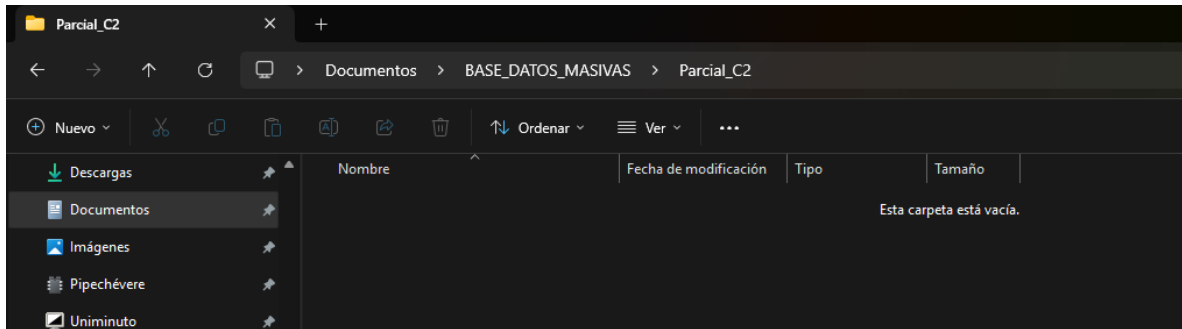
ÍNDICE DE CONTENIDO

Conexión a base de datos PostgreSQL (Supabase).....	4
Creamos una carpeta donde se alojará los archivos relacionados a la base de datos:	4
Ya en Supabase creamos el proyecto:.....	4
En la conexión tenemos en cuenta estos parámetros:	4
Ya en el editor de código, creamos los archivos que nos ayudarán con la parte funcional backend de la base de datos, conexión y funcionalidades que programaremos para que funcionen con postman.....	5
Hacemos la conexión con Supabase con el fichero desde Visual, teniendo en cuenta los parámetros de la conexión:	5
Configuramos el cuerpo del CRUD, pero primero nos fijamos en que la conexión sea exitosa:	6
Escucha del puerto:	7
Y al final, he aquí el modelo relacional:	8
Crear rutas y controladores con Express para cada entidad (CRUD).....	9
CREATE datos restaurante: >_http://localhost:3020/api/restaurantes_<	9
READ datos restaurante: >_http://localhost:3020/api/restaurantes_<.....	10
UPDATE datos restaurante: >_http://localhost:3020/api/restaurantes/##_<.....	11
DELETE datos restaurante: >_http://localhost:3020/api/restaurantes/##_<.....	12
CREATE datos empleado: >_http://localhost:3020/api/empleados_<.....	13
READ datos empleado: >_http://localhost:3020/api/empleados_<	14
UPDATE datos empleado: >_http://localhost:3020/api/empleados/##_<	15
DELETE datos empleado: >_http://localhost:3020/api/empleados/##_<	16
CREATE datos producto: >_http://localhost:3020/api/productos_<	17
READ datos producto: >_http://localhost:3020/api/productos_<	18
UPDATE datos producto: >_http://localhost:3020/api/productos/##_<	19
DELETE datos producto: >_http://localhost:3020/api/ productos/##_<	20
CREATE datos pedido: >_http://localhost:3020/api/pedidos_<	21
READ datos pedido: >_http://localhost:3020/api/ pedidos_<	22
UPDATE datos pedido: >_http://localhost:3020/api/ pedidos/##_<	23
DELETE datos pedido: >_http://localhost:3020/api/ pedidos/##_<	24
CREATE datos detalle_pedido: >_http://localhost:3020/api/detalles_<	25
READ datos detalle_pedido: >_http://localhost:3020/api/detalles_<.....	26

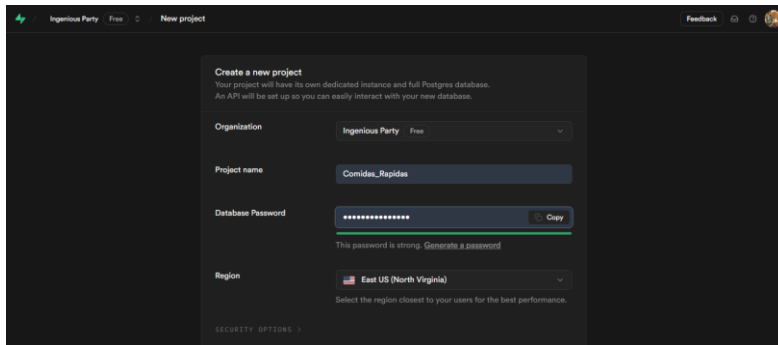
UPDATE datos detalles_pedido: >_http://localhost:3020/api/detalles/##_<	27
DELETE datos detalle_pedido: >_http://localhost:3020/api/detalles/##_<.....	28
Consultas Nativas: Con Ruta Específica, y Petición De Postman	29
Obtener todos los productos de un pedido específico:	29
Obtener los productos más vendidos (más de X unidades):	30
Obtener el total de ventas por restaurante:	31
Obtener los pedidos realizados en una fecha específica:.....	32
Obtener los empleados por rol en un restaurante:	33
Anexos	34
100 Registros Por Tabla:	34
Modelo De Base De Datos De Supabase:	34
Ficheros de la base de datos:	34
Colección de Postman:	34

Conexión a base de datos PostgreSQL (Supabase)

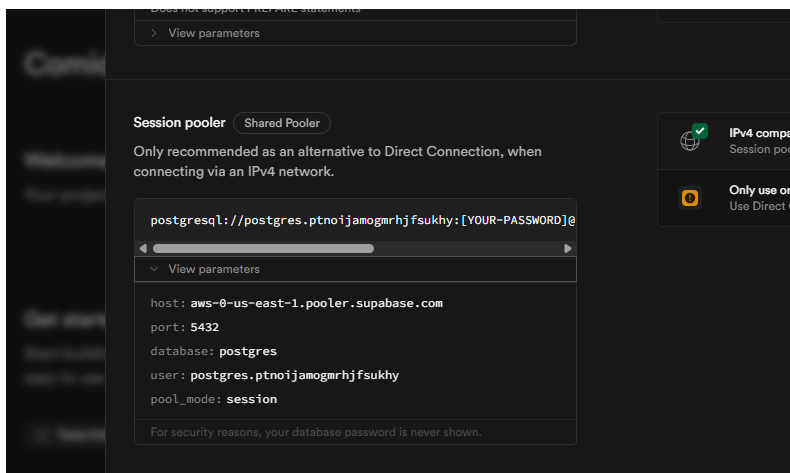
Creamos una carpeta donde se alojará los archivos relacionados a la base de datos:



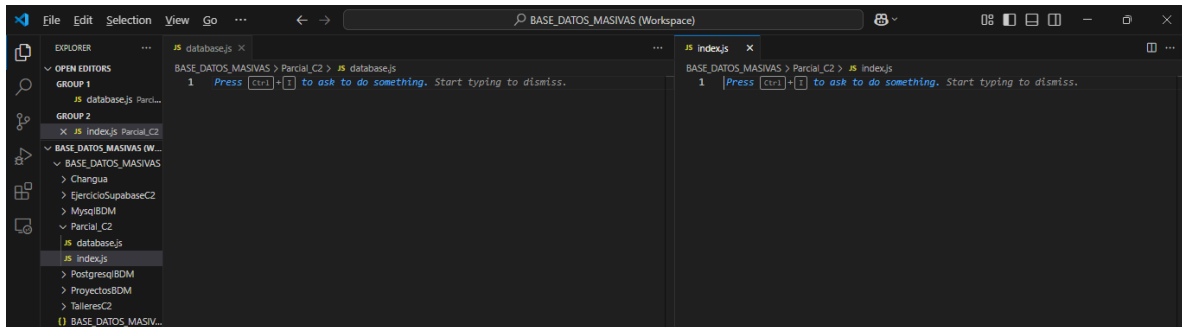
Ya en Supabase creamos el proyecto:



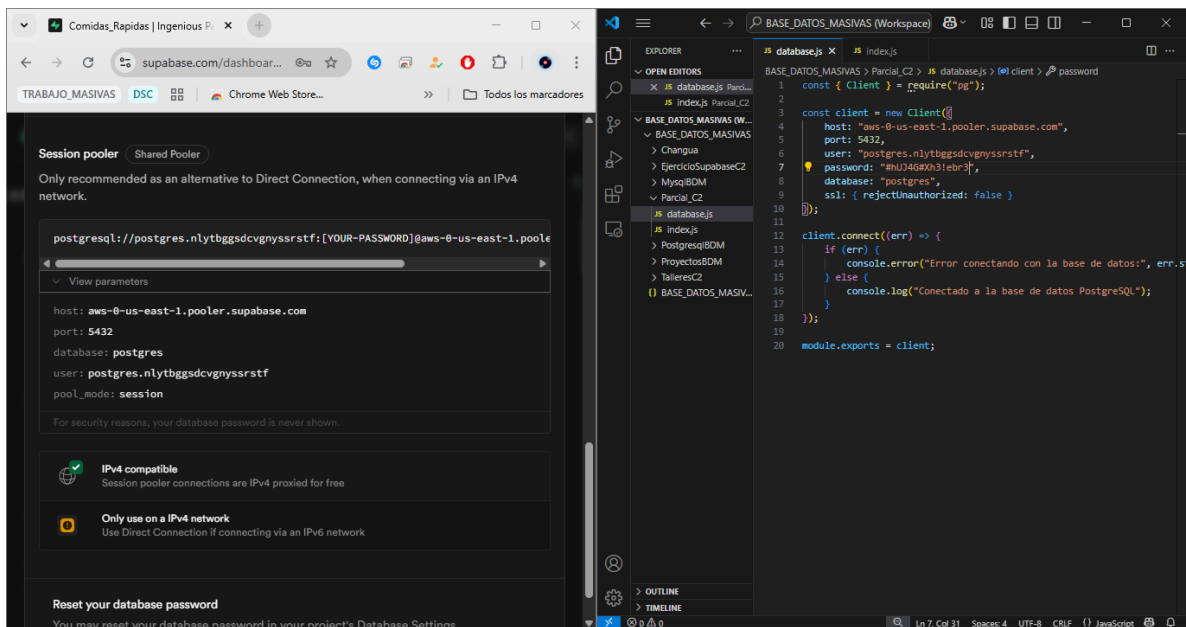
En la conexión tenemos en cuenta estos parámetros:



Ya en el editor de código, creamos los archivos que nos ayudarán con la parte funcional backend de la base de datos, conexión y funcionalidades que programaremos para que funcionen con postman



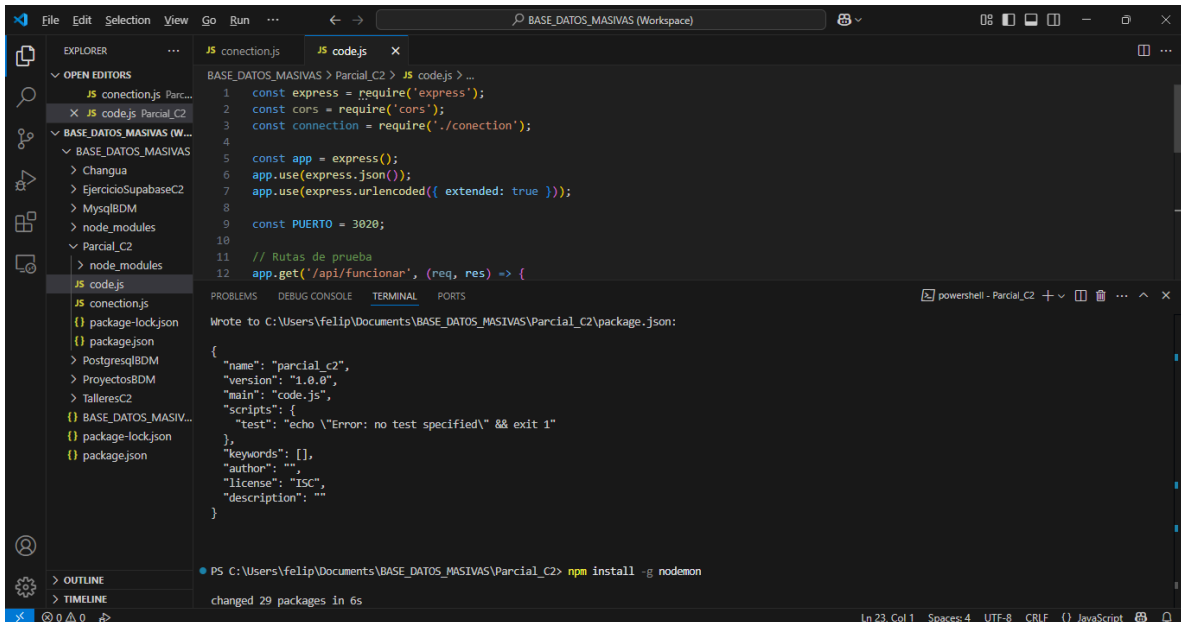
Hacemos la conexión con Supabase con el fichero desde Visual, teniendo en cuenta los parámetros de la conexión:



Configuramos el cuerpo del CRUD, pero primero nos fijamos en que la conexión sea exitosa:

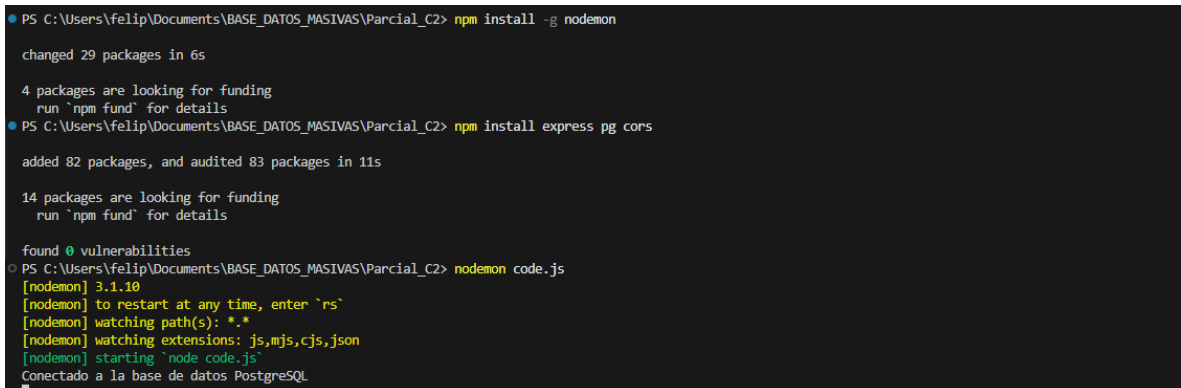


```
1  const express = require('express');
2  const cors = require('cors');
3  const connection = require('./conexion');
4
5  const app = express();
6  app.use(express.json());
7  app.use(express.urlencoded({ extended: true }));
8
9  const PUERTO = 3020;
10
11 // Rutas de prueba
12 app.get('/api/funcionar', (req, res) => {
13   res.send('API funcionando de manera correcta');
14 });
15
16 app.get('/api/pellizco', (req, res) => {
17   res.status(200).json({
18     message: 'LA API RESPONDE CORRECTAMENTE',
19     port: PUERTO,
20     status: 'success'
21   });
22 });
```



```
Wrote to C:\Users\felip\Documents\BASE_DATOS_MASIVAS\Parcial_C2\package.json:
{
  "name": "parcial_c2",
  "version": "1.0.0",
  "main": "code.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}

● PS C:\Users\felip\Documents\BASE_DATOS_MASIVAS\Parcial_C2> npm install -g nodemon
changed 29 packages in 6s
```



```
● PS C:\Users\felip\Documents\BASE_DATOS_MASIVAS\Parcial_C2> npm install -g nodemon
changed 29 packages in 6s

4 packages are looking for funding
run `npm fund` for details
● PS C:\Users\felip\Documents\BASE_DATOS_MASIVAS\Parcial_C2> npm install express pg cors

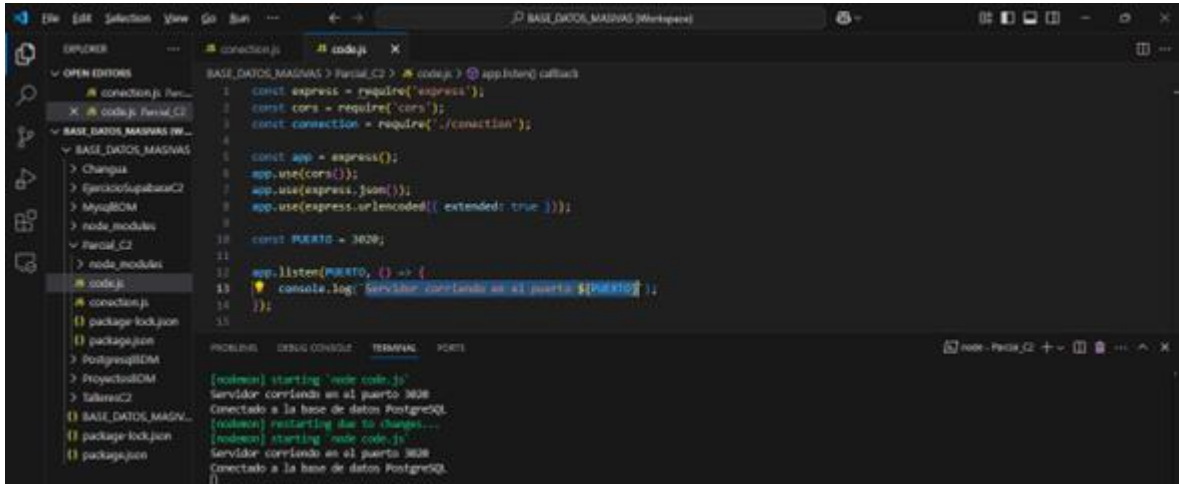
added 82 packages, and audited 83 packages in 11s

14 packages are looking for funding
run `npm fund` for details

found 0 vulnerabilities
● PS C:\Users\felip\Documents\BASE_DATOS_MASIVAS\Parcial_C2> nodemon code.js
[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node code.js`
Conectado a la base de datos PostgreSQL
```

Escucha del puerto:

Aquí importamos las dependencias express y enlazamos con el fichero de la base de datos “Conexion.js”; además le pedimos que nos responda por un puerto, que en este caso es el 3020. Ya de primera instancia encontramos un “API” que se encarga de escuchar y señalarnos por medio del terminal de visual studio code, si efectivamente está conectada a SupaBase.

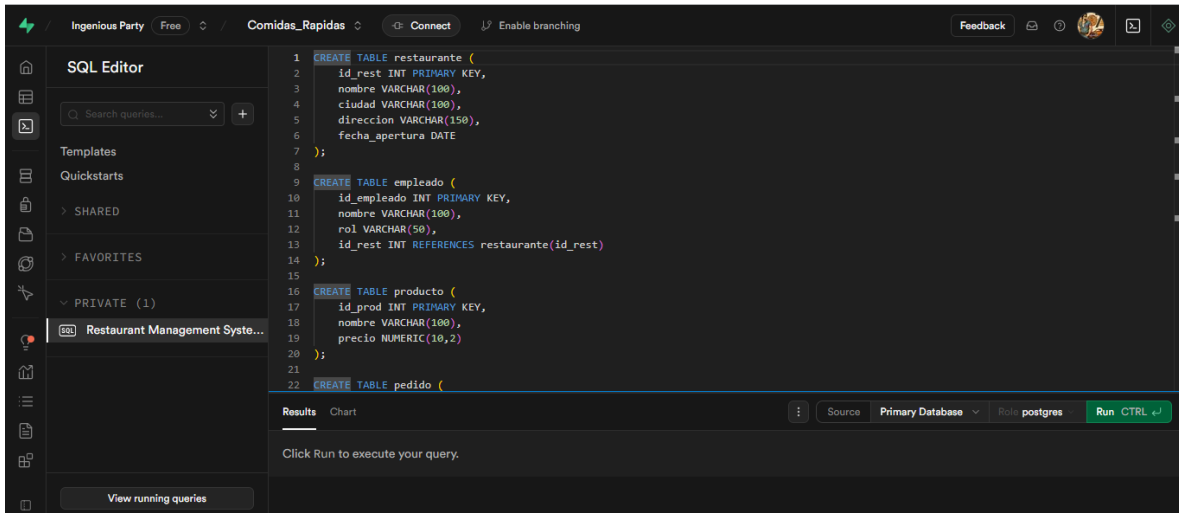


```
1 const express = require('express');
2 const cors = require('cors');
3 const connection = require('./conexion');
4
5 const app = express();
6 app.use(cors());
7 app.use(express.json());
8 app.use(express.urlencoded({ extended: true }));
9
10 const PUERTO = 3020;
11
12 app.listen(PUERTO, () => {
13   console.log(`Servidor corriendo en el puerto ${PUERTO}`);
14 });
15
```

PROBLEMS DEBUG CONSOLE TERMINAL PORTS

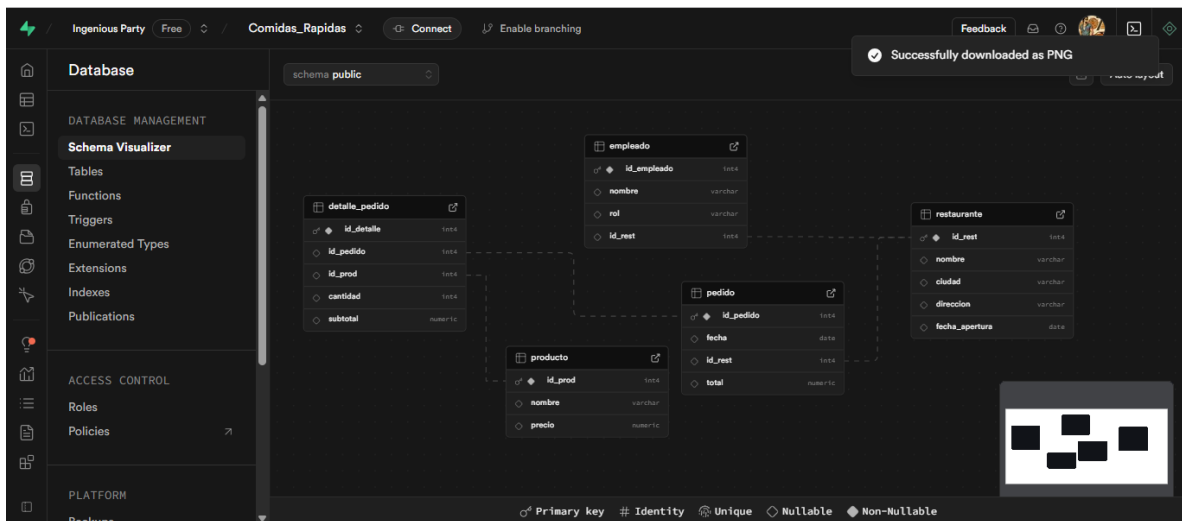
```
[nodemon] starting 'node code.js'
Servidor corriendo en el puerto 3020
Conectado a la base de datos PostgreSQL
[nodemon] restarting due to changes...
[nodemon] starting 'node code.js'
Servidor corriendo en el puerto 3020
Conectado a la base de datos PostgreSQL
```

Creamos las tablas desde Supabase:



```
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 REFERENCES restaurante(id_rest)
14 );
15
16 CREATE TABLE producto (
17  id_prod INT PRIMARY KEY,
18  nombre VARCHAR(100),
19  precio NUMERIC(10,2)
20 );
21
22 CREATE TABLE pedido (
```

Y al final, he aquí el modelo relacional:



Ingresamos 100 registros a cada tabla, por medio del SQL Editor:

The screenshot shows the SQL Editor interface in Ingenious Party. The left sidebar contains navigation options: SQL Editor, Templates, Quickstarts, SHARED, FAVORITES, and PRIVATE (1). The main area displays a list of 100 INSERT statements for each table. The statements are as follows:

```
394 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (89, '2023-11-07', 1, 233.74);
395 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (90, '2023-12-11', 1, 390.55);
396 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (91, '2025-01-02', 1, 214.25);
397 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (92, '2024-12-15', 1, 175.61);
398 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (93, '2025-03-30', 1, 328.46);
399 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (94, '2023-12-24', 1, 448.92);
400 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (95, '2023-12-26', 1, 238.81);
401 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (96, '2023-10-27', 1, 274.76);
402 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (97, '2025-02-12', 1, 124.61);
403 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (98, '2023-12-24', 1, 404.06);
404 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (99, '2023-10-19', 1, 91.78);
405 INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES (100, '2023-04-25', 1, 339.77);
406
407 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (1, 1, 43, 6, 118.74);
408 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (2, 1, 31, 10, 30.0);
409 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (3, 1, 38, 9, 128.43);
410 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (4, 1, 34, 9, 133.02);
411 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (5, 2, 64, 5, 13.6);
412 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (6, 2, 50, 3, 14.58);
413 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (7, 3, 42, 9, 85.59);
414 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (8, 3, 1, 2, 14.66);
415 INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES (9, 4, 72, 2, 24.88);
```

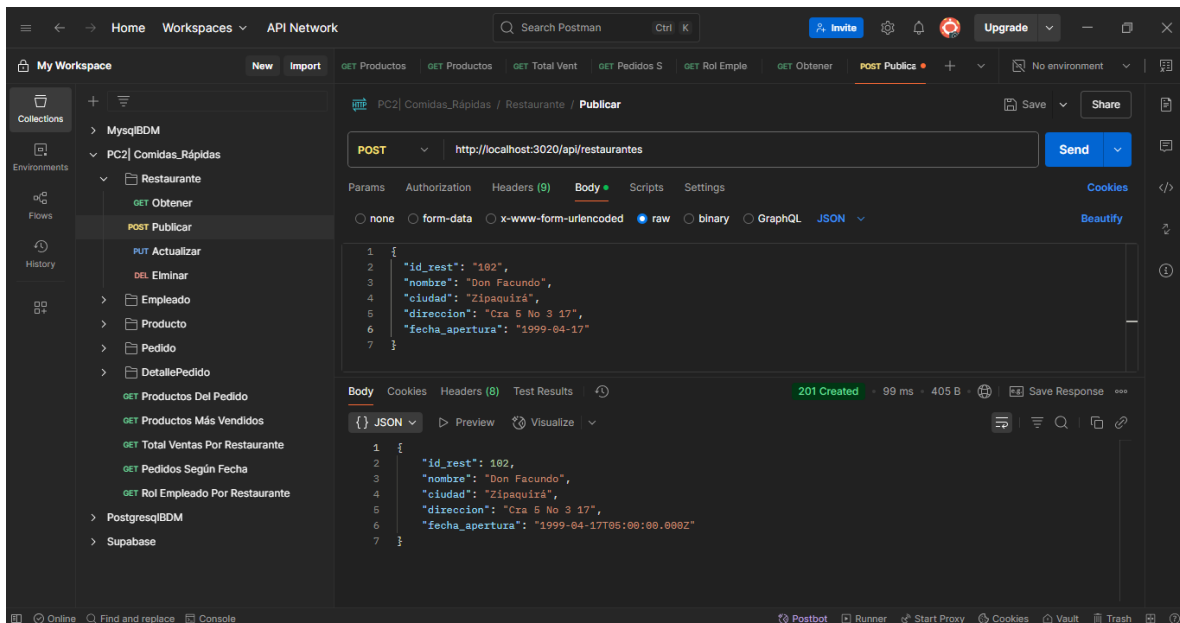
The bottom of the editor shows a "Results" tab with the message "Success. No rows returned". A "Run" button with a keyboard shortcut "CTRL ↵" is visible.

Crear rutas y controladores con Express para cada entidad (CRUD)

CREATE datos restaurante: > `_http://localhost:3020/api/restaurantes_<`

```
15 // API CREATE tabla: restaurante PUBLICAR [POST]
16 // API CREATE tabla: restaurante PUBLICAR [POST]
17 app.post('/api/restaurantes', async (req, res) => {
18   const { id_rest, nombre, ciudad, direccion, fecha_apertura } = req.body;
19   const query = 'INSERT INTO restaurante (id_rest, nombre, ciudad, direccion, fecha_apertura) VALUES ($1, $2, $3, $4, $5) RETURNING *';
20   try {
21     const result = await connection.query(query, [id_rest, nombre, ciudad, direccion, fecha_apertura]);
22     res.status(201).json(result.rows[0]);
23   } catch (err) {
24     res.status(500).json({ message: 'Error al crear el restaurante', error: err.message });
25   }
26 });
```

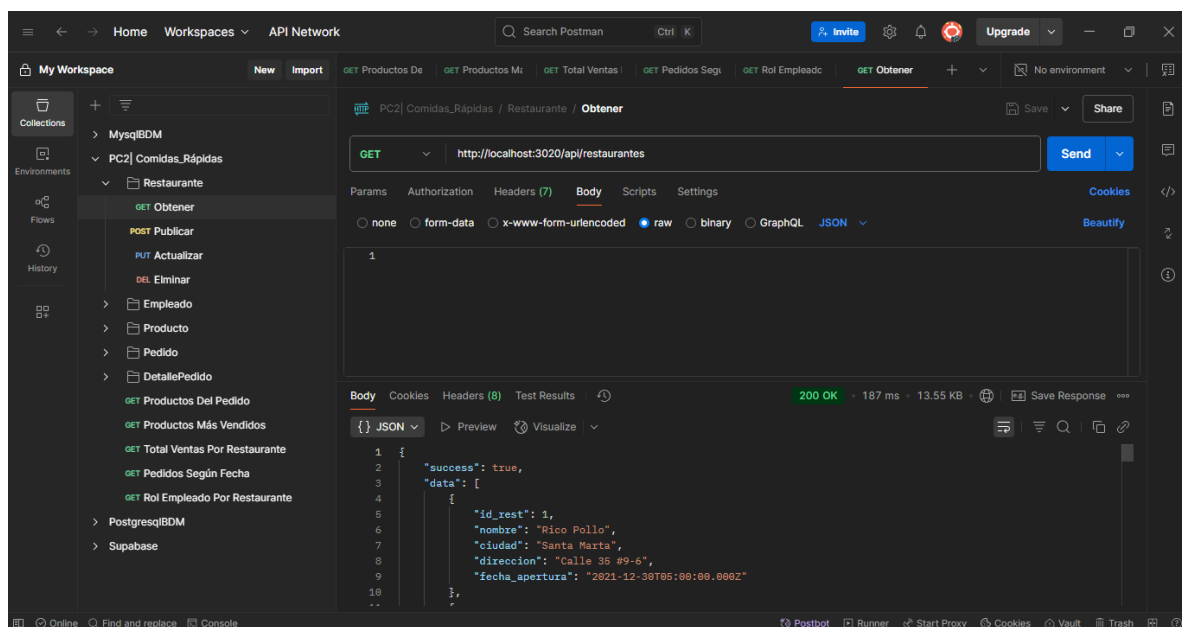
Creamos un API capaz de hacer un registro de ingreso de un **restaurante**, donde todos sus parámetros son ingresados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarlos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y como queremos ingresar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



READ datos restaurante: >_http://localhost:3020/api/restaurantes_<

```
27 // API READ tabla: restaurante OBTENER [GET]
28 app.get('/api/restaurantes', async (req, res) => {
29   try {
30     const result = await connection.query('SELECT * FROM restaurante');
31     res.status(200).json({ success: true, data: result.rows });
32   } catch (err) {
33     res.status(500).json({ success: false, message: err.message });
34   }
35 });
```

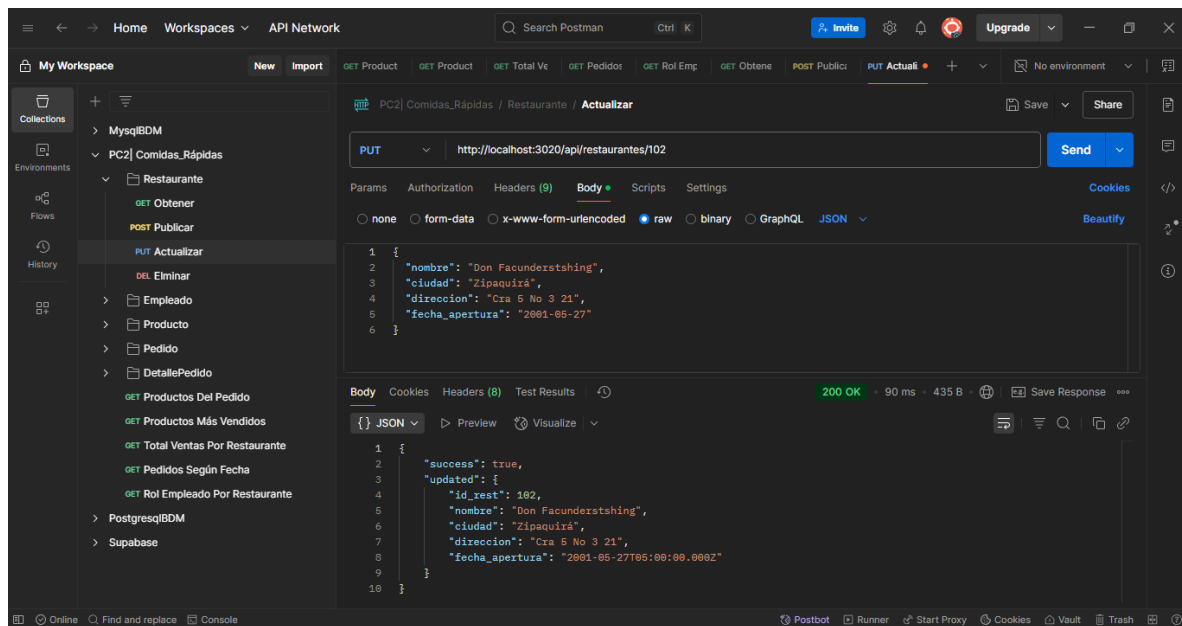
Creamos un API capaz de botarnos todos los registros existentes de todos los **restaurantes**, donde simplemente le pasamos una sentencia SQL dentro del intento de captura exitosa, y si no de lo contrario nos lanza que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y si es correcta la sintaxis, nos arroja todos los **restaurantes** registrados hasta el momento:



UPDATE datos restaurante: >_http://localhost:3020/api/restaurantes/##_<

```
36 // API UPDATE tabla: restaurante ACTUALIZAR [PUT]
37 app.put('/api/restaurantes/:id', async (req, res) => {
38   const { id } = req.params;
39   const { nombre, ciudad, direccion, fecha_apertura } = req.body;
40   const query = 'UPDATE restaurante SET nombre = $1, ciudad = $2, direccion = $3, fecha_apertura = $4 WHERE id_rest = $5 RETURNING *';
41   try {
42     const resul (property) QueryResultBase.rowCount: number | null | direccion, fecha_apertura, id]);
43     if (result.rowCount === 0) {
44       return res.status(404).json({ success: false, message: `No se encontró restaurante con ID ${id}` });
45     }
46     res.status(200).json({ success: true, updated: result.rows[0] });
47   } catch (err) {
48     res.status(500).json({ success: false, error: err.message });
49   }
50 });
```

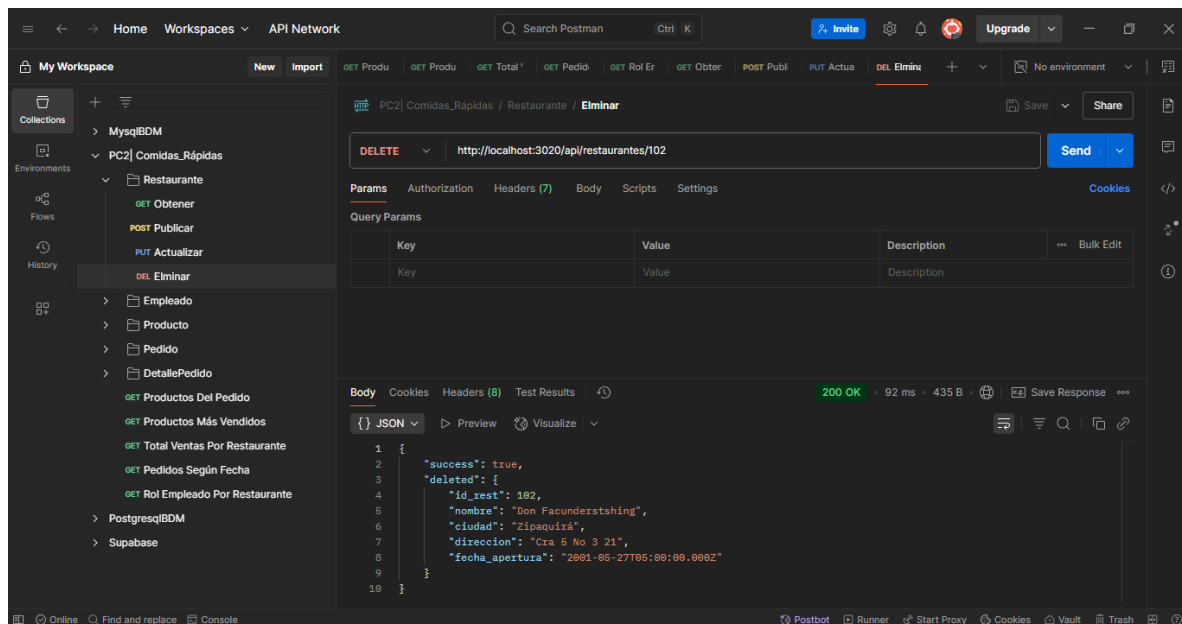
Creamos un API capaz de hacer una sobreescritura de un ingreso de un **restaurante** ya creado, donde todos sus parámetros son modificados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarnos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id a modificar, y como queremos modificar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



DELETE datos restaurante: >_http://localhost:3020/api/restaurantes/##_<

```
51 // API DELETE tabla: restaurante ELIMINAR [DELETE]
52 app.delete('/api/restaurantes/:id', async (req, res) => {
53   const { id } = req.params;
54   const query = 'DELETE FROM restaurante WHERE id_rest = $1 RETURNING *';
55   try {
56     const result = await connection.query(query, [id]);
57     if (result.rowCount === 0) {
58       return res.status(404).json({ success: false, message: `No existe restaurante con ID ${id}` });
59     }
60     res.status(200).json({ success: true, deleted: result.rows[0] });
61   } catch (err) {
62     res.status(500).json({ success: false, error: err.message });
63   }
64 });
```

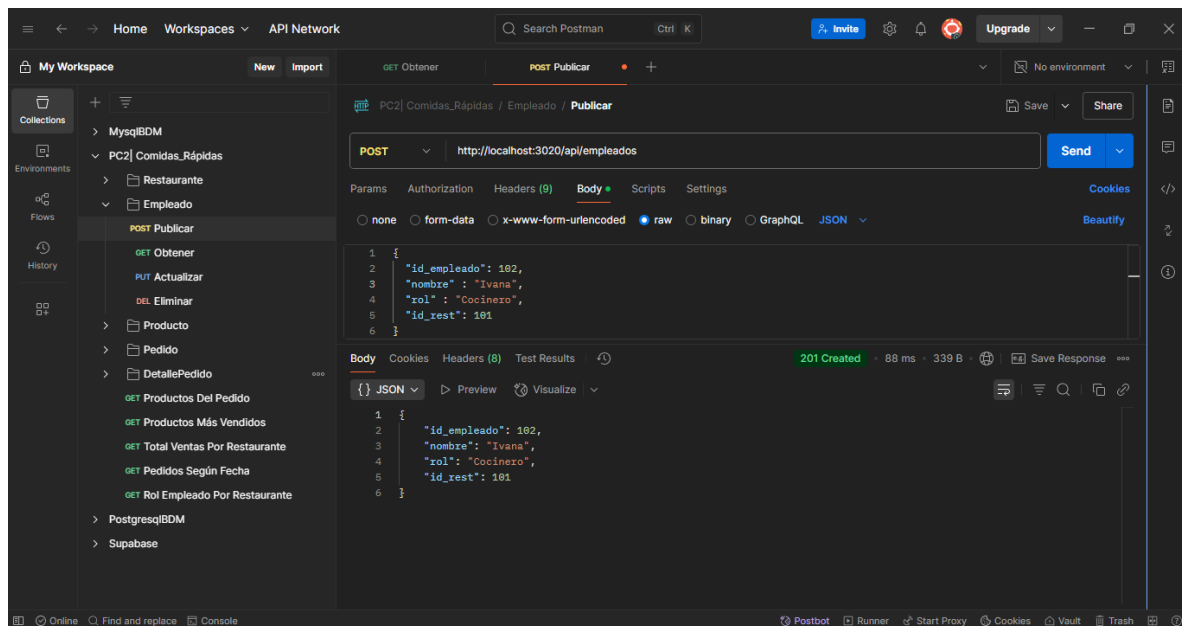
Creamos un API capaz de hacer una eliminación de un **restaurante** ya creado, donde solo por medio de la ruta ingresamos el id de este mismo; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en el intento de captura exitosa, y si no de lo contrario nos arrojará que se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id del **restaurante** a eliminar.



CREATE datos empleado: >_http://localhost:3020/api/empleados_<

```
65 ///////////////////////////////////////////////////
66 // API CREATE tabla: empleado PUBLICAR [POST]
67 app.post('/api/empleados', async (req, res) => {
68   const { id_empleado, nombre, rol, id_rest } = req.body;
69   const query = 'INSERT INTO empleado (id_empleado, nombre, rol, id_rest) VALUES ($1, $2, $3, $4) RETURNING *';
70   try {
71     const result = await connection.query(query, [id_empleado, nombre, rol, id_rest]);
72     res.status(201).json(result.rows[0]);
73   } catch (err) {
74     res.status(500).json({ message: 'Error al crear el empleado', error: err.message });
75   }
76 });
```

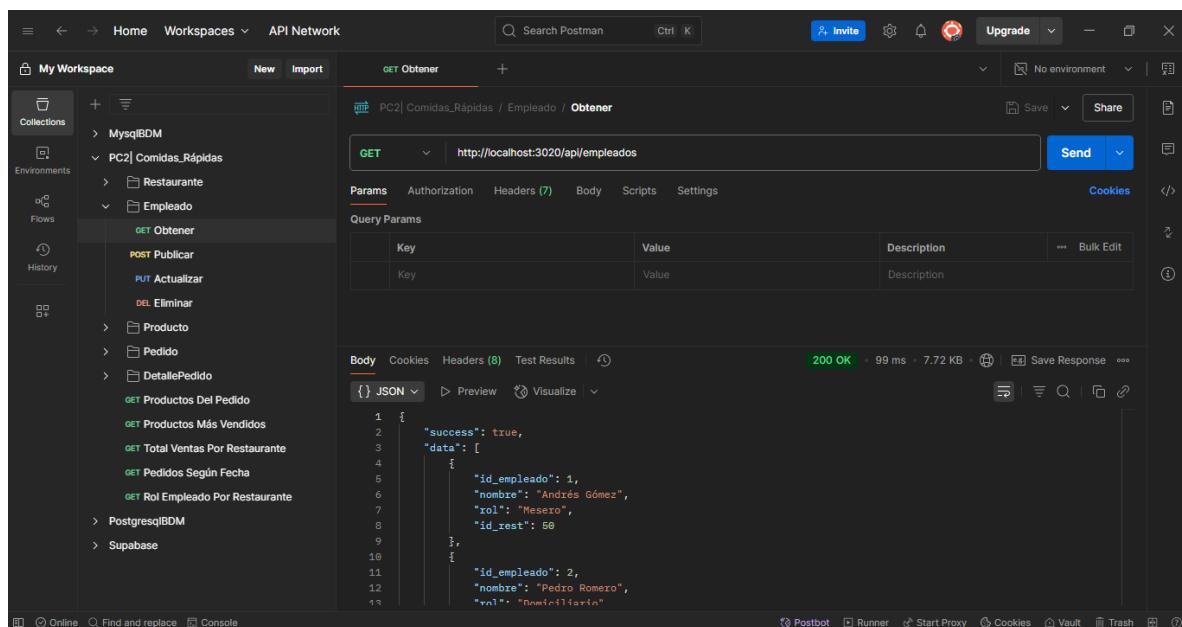
Creamos un API capaz de hacer un registro de ingreso de un **empleado**, donde todos sus parámetros son ingresados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarlos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y como queremos ingresar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



READ datos empleado: >_http://localhost:3020/api/empleados_<

```
77 // API READ tabla: empleado OBTENER [GET]
78 app.get('/api/empleados', async (req, res) => {
79   try {
80     const result = await connection.query('SELECT * FROM empleado');
81     res.status(200).json({ success: true, data: result.rows });
82   } catch (err) {
83     res.status(500).json({ success: false, message: err.message });
84   }
85 });
```

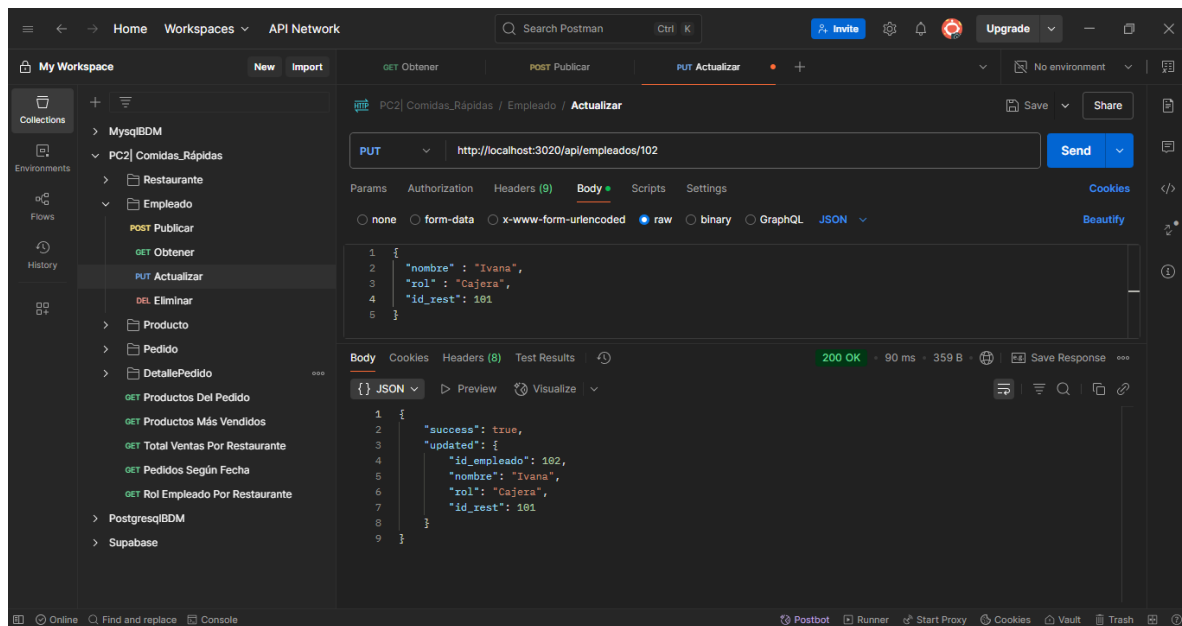
Creamos un API capaz de botarnos todos los registros existentes de todos los **empleados**, donde simplemente le pasamos una sentencia SQL dentro del intento de captura exitosa, y si no de lo contrario nos lanza que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y si es correcta la sintaxis, nos arroja todos los **empleados** registrados hasta el momento:



UPDATE datos empleado: >_http://localhost:3020/api/empleados/##_<

```
86 // API UPDATE tabla: empleado ACTUALIZAR [PUT]
87 app.put('/api/empleados/:id', async (req, res) => {
88   const { id } = req.params;
89   const { nombre, rol, id_rest } = req.body;
90   const query = 'UPDATE empleado SET nombre = $1, rol = $2, id_rest = $3 WHERE id_empleado = $4 RETURNING *';
91   try {
92     const result = await connection.query(query, [nombre, rol, id_rest, id]);
93     if (result.rowCount === 0) {
94       return res.status(404).json({ success: false, message: `No se encontró empleado con ID ${id}` });
95     }
96     res.status(200).json({ success: true, updated: result.rows[0] });
97   } catch (err) {
98     res.status(500).json({ success: false, error: err.message });
99   }
100 });
```

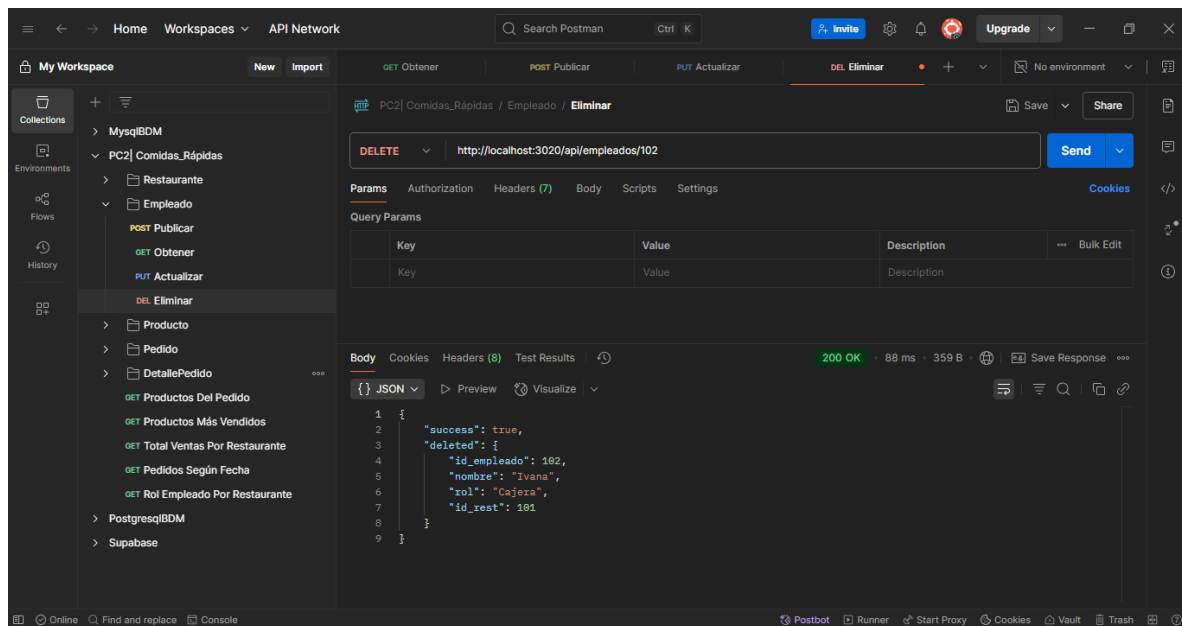
Creamos un API capaz de hacer una sobreescritura de un ingreso de un **empleado** ya creado, donde todos sus parámetros son modificados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarnos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id a modificar, y como queremos modificar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



DELETE datos empleado: >_http://localhost:3020/api/empleados/##_<

```
101 // API DELETE tabla: empleado ELIMINAR [DELETE]
102 app.delete('/api/empleados/:id', async (req, res) => {
103   const { id } = req.params;
104   const query = 'DELETE FROM empleado WHERE id_empleado = $1 RETURNING *';
105   try {
106     const result = await connection.query(query, [id]);
107     if (result.rowCount === 0) {
108       return res.status(404).json({ success: false, message: 'Empleado con ID ${id} no existe' });
109     }
110     res.status(200).json({ success: true, deleted: result.rows[0] });
111   } catch (err) {
112     res.status(500).json({ success: false, error: err.message });
113   }
114 });
```

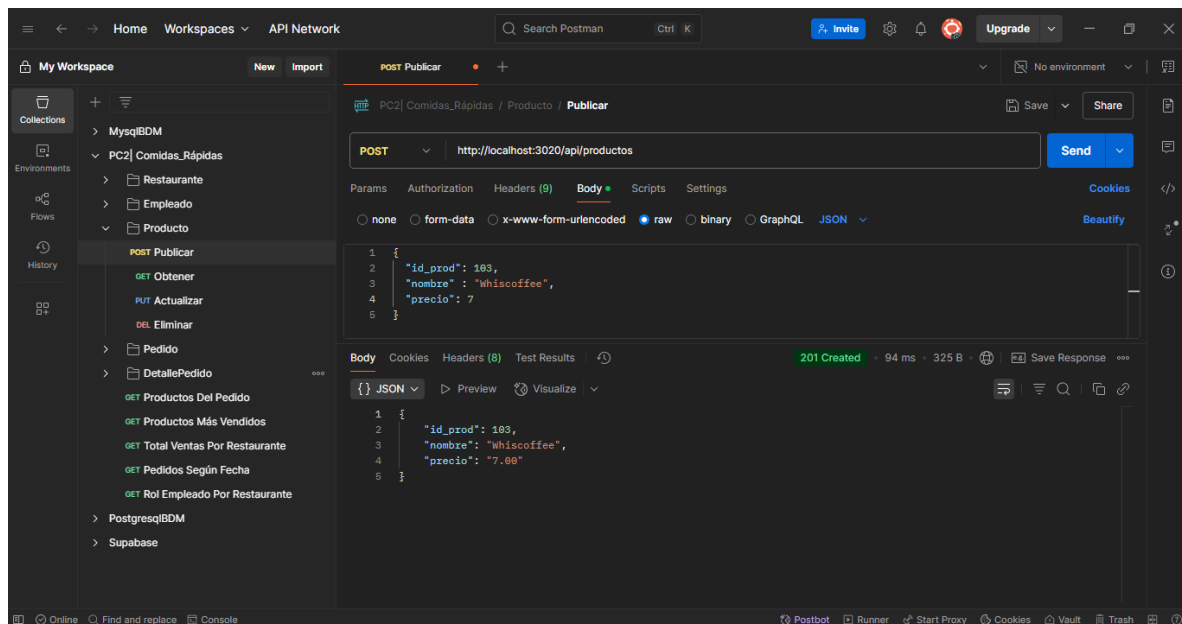
Creamos un API capaz de hacer una eliminación de un **empleado** ya creado, donde solo por medio de la ruta ingresamos el id de este mismo; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en el intento de captura exitosa, y si no de lo contrario nos arrojará que se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id del **empleado** a eliminar:



CREATE datos producto: >_http://localhost:3020/api/productos_<

```
115 ///////////////////////////////////////////////////
116 // API CREATE tabla: producto PUBLICAR [POST]
117 app.post('/api/productos', async (req, res) => {
118   const { id_prod, nombre, precio } = req.body;
119   const query = 'INSERT INTO producto (id_prod, nombre, precio) VALUES ($1, $2, $3) RETURNING *';
120   try {
121     const result = await connection.query(query, [id_prod, nombre, precio]);
122     res.status(201).json(result.rows[0]);
123   } catch (err) {
124     res.status(500).json({ message: 'Error al crear el producto', error: err.message });
125   }
126 });
```

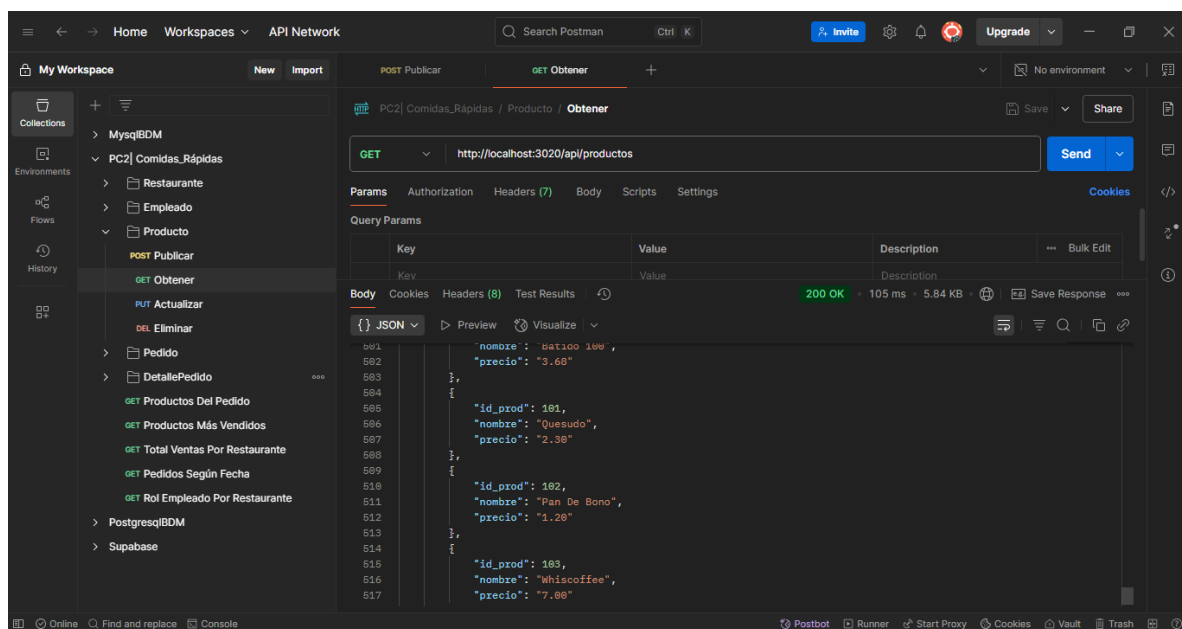
Creamos un API capaz de hacer un registro de ingreso de un **producto**, donde todos sus parámetros son ingresados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarlos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y como queremos ingresar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



READ datos producto: >_http://localhost:3020/api/productos_<

```
127 // API READ tabla: producto OBTENER [GET]
128 app.get('/api/productos', async (req, res) => {
129   try {
130     const result = await connection.query('SELECT * FROM producto');
131     res.status(200).json({ success: true, data: result.rows });
132   } catch (err) {
133     res.status(500).json({ success: false, message: err.message });
134   }
135 });
```

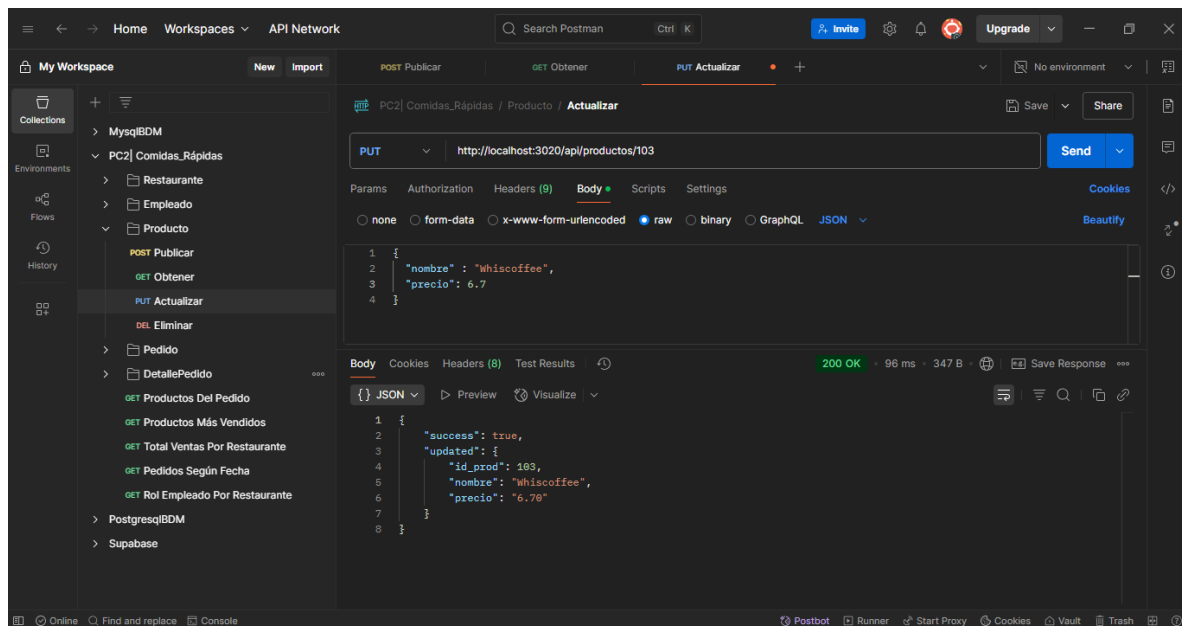
Creamos un API capaz de botarnos todos los registros existentes de todos los **productos**, donde simplemente le pasamos una sentencia SQL dentro del intento de captura exitosa, y si no de lo contrario nos lanza que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y si es correcta la sintaxis, nos arroja todos los **productos** registrados hasta el momento:



UPDATE datos producto: >_http://localhost:3020/api/productos/##_<

```
136 // API UPDATE tabla: producto ACTUALIZAR [PUT]
137 app.put('/api/productos/:id', async (req, res) => {
138   const { id } = req.params;
139   const { nombre, precio } = req.body;
140   const query = 'UPDATE producto SET nombre = $1, precio = $2 WHERE id_prod = $3 RETURNING *';
141   try {
142     const result: QueryResult<any> = await db.query(query, [nombre, precio, id]);
143     if (result.rowCount === 0) {
144       return res.status(404).json({ success: false, message: 'Producto con ID ${id} no encontrado' });
145     }
146     res.status(200).json({ success: true, updated: result.rows[0] });
147   } catch (err) {
148     res.status(500).json({ success: false, error: err.message });
149   }
150 });
```

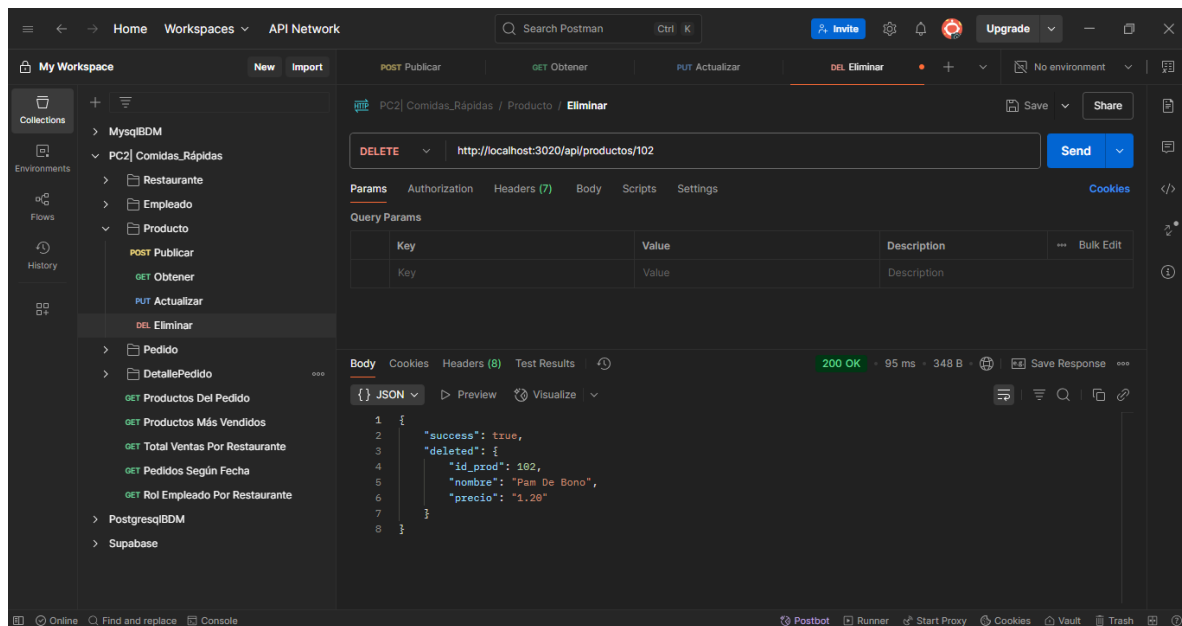
Creamos un API capaz de hacer una sobreescritura de un ingreso de un **producto** ya creado, donde todos sus parámetros son modificados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarnos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id a modificar, y como queremos modificar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



DELETE datos producto: >_http://localhost:3020/api/ productos/##_<

```
151 // API DELETE tabla: producto ELIMINAR [DELETE]
152 app.delete('/api/productos/:id', async (req, res) => {
153   const { id } = req.params;
154   const query = 'DELETE FROM producto WHERE id_prod = $1 RETURNING *';
155   try {
156     const result = await connection.query(query, [id]);
157     if (result.rowCount === 0) {
158       return res.status(404).json({ success: false, message: 'Producto con ID ${id} no existe' });
159     }
160     res.status(200).json({ success: true, deleted: result.rows[0] });
161   } catch (err) {
162     res.status(500).json({ success: false, error: err.message });
163   }
164 });
```

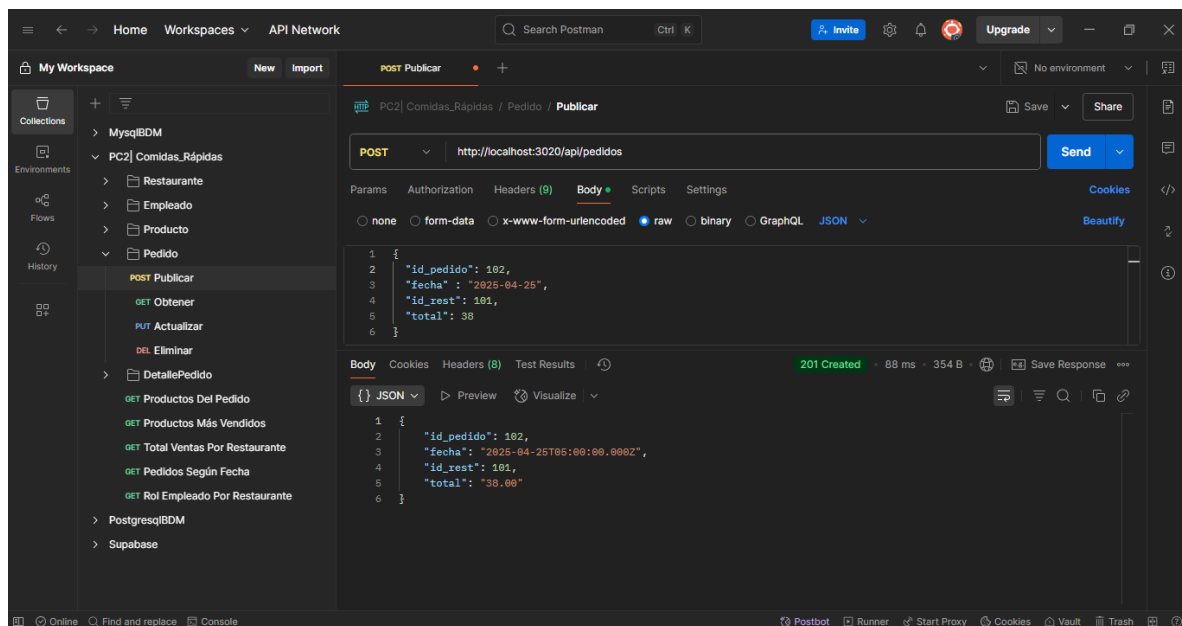
Creamos un API capaz de hacer una eliminación de un **producto** ya creado, donde solo por medio de la ruta ingresamos el id de este mismo; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en el intento de captura exitosa, y si no de lo contrario nos arrojará que se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id del **producto** a eliminar:



CREATE datos pedido: >_http://localhost:3020/api/pedidos_<

```
165 //////////////////////////////////////
166 // API CREATE tabla: pedido PUBLICAR [POST]
167 app.post('/api/pedidos', async (req, res) => {
168   const { id_pedido, fecha, id_rest, total } = req.body;
169   const query = 'INSERT INTO pedido (id_pedido, fecha, id_rest, total) VALUES ($1, $2, $3, $4) RETURNING *';
170   try {
171     const result = await connection.query(query, [id_pedido, fecha, id_rest, total]);
172     res.status(201).json(result.rows[0]);
173   } catch (err) {
174     res.status(500).json({ message: 'Error al crear el pedido', error: err.message });
175   }
176 });
```

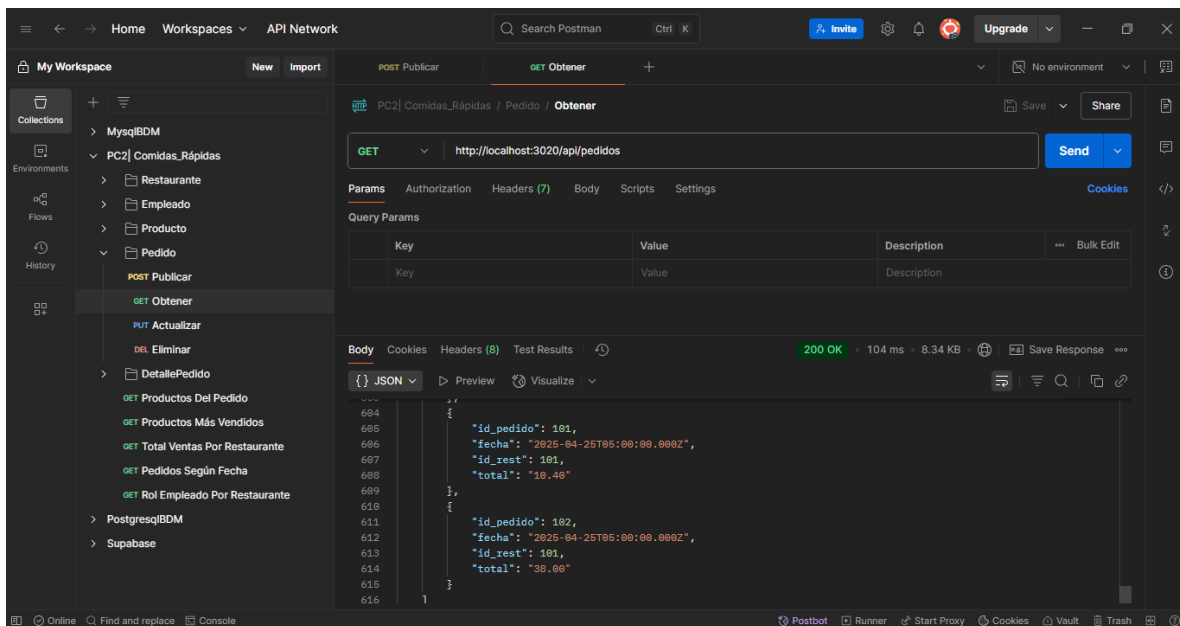
Creemos un API capaz de hacer un registro de ingreso de un **pedido**, donde todos sus parámetros son ingresados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarnos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y como queremos ingresar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



READ datos pedido: >_http://localhost:3020/api/ pedidos_<

```
177 // API READ tabla: pedido OBTENER [GET]
178 app.get('/api/pedidos', async (req, res) => {
179   try {
180     const result = await connection.query('SELECT * FROM pedido');
181     res.status(200).json({ success: true, data: result.rows });
182   } catch (err) {
183     res.status(500).json({ success: false, message: err.message });
184   }
185 });
```

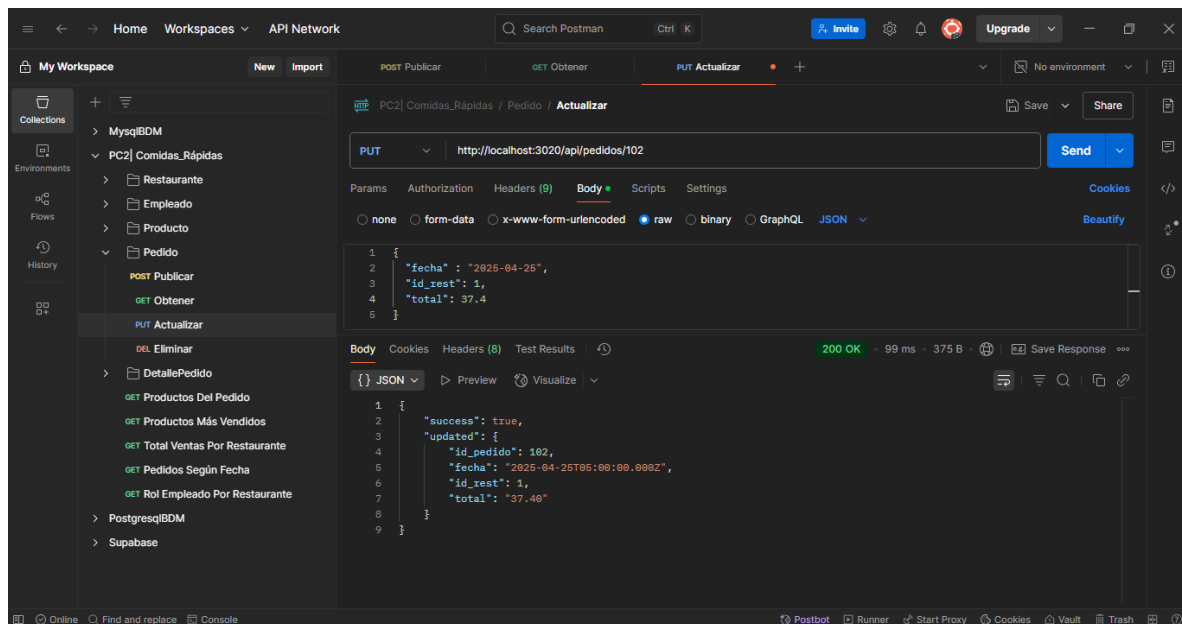
Creamos un API capaz de botarnos todos los registros existentes de todos los **pedidos**, donde simplemente le pasamos una sentencia SQL dentro del intento de captura exitosa, y si no de lo contrario nos lanza que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y si es correcta la sintaxis, nos arroja todos los **pedidos** registrados hasta el momento:



UPDATE datos pedido: >_http://localhost:3020/api/ pedidos/##_<

```
186 // API UPDATE tabla: pedido ACTUALIZAR [PUT]
187 app.put('/api/pedidos/:id', async (req, res) => {
188   const { id } = req.params;
189   const { fecha, id_rest, total } = req.body;
190   const query = 'UPDATE pedido SET fecha = $1, id_rest = $2, total = $3 WHERE id_pedido = $4 RETURNING *';
191   try {
192     const result = await connection.query(query, [fecha, id_rest, total, id]);
193     if (result.rowCount === 0) {
194       return res.status(404).json({ success: false, message: `Pedido con ID ${id} no encontrado` });
195     }
196     res.status(200).json({ success: true, updated: result.rows[0] });
197   } catch (err) {
198     res.status(500).json({ success: false, error: err.message });
199   }
200 });
```

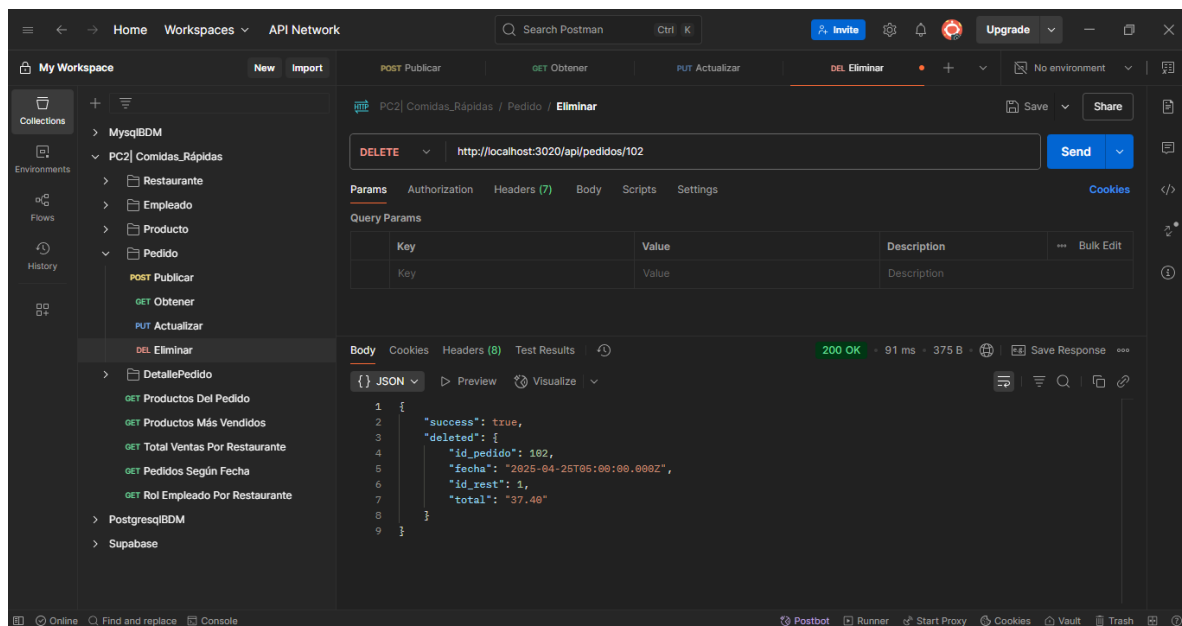
Creamos un API capaz de hacer una sobreescritura de un ingreso de un **pedido** ya creado, donde todos sus parámetros son modificados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarnos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id a modificar, y como queremos modificar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



DELETE datos pedido: >_http://localhost:3020/api/ pedidos/##_<

```
201 // API DELETE tabla: pedido ELIMINAR [DELETE]
202 app.delete('/api/pedidos/:id', async (req, res) => {
203   const { id } = req.params;
204   const query = 'DELETE FROM pedido WHERE id_pedido = $1 RETURNING *';
205   try {
206     const result = await connection.query(query, [id]);
207     if (result.rowCount === 0) {
208       return res.status(404).json({ success: false, message: `Pedido con ID ${id} no existe` });
209     }
210     res.status(200).json({ success: true, deleted: result.rows[0] });
211   } catch (err) {
212     res.status(500).json({ success: false, error: err.message });
213   }
214 });
```

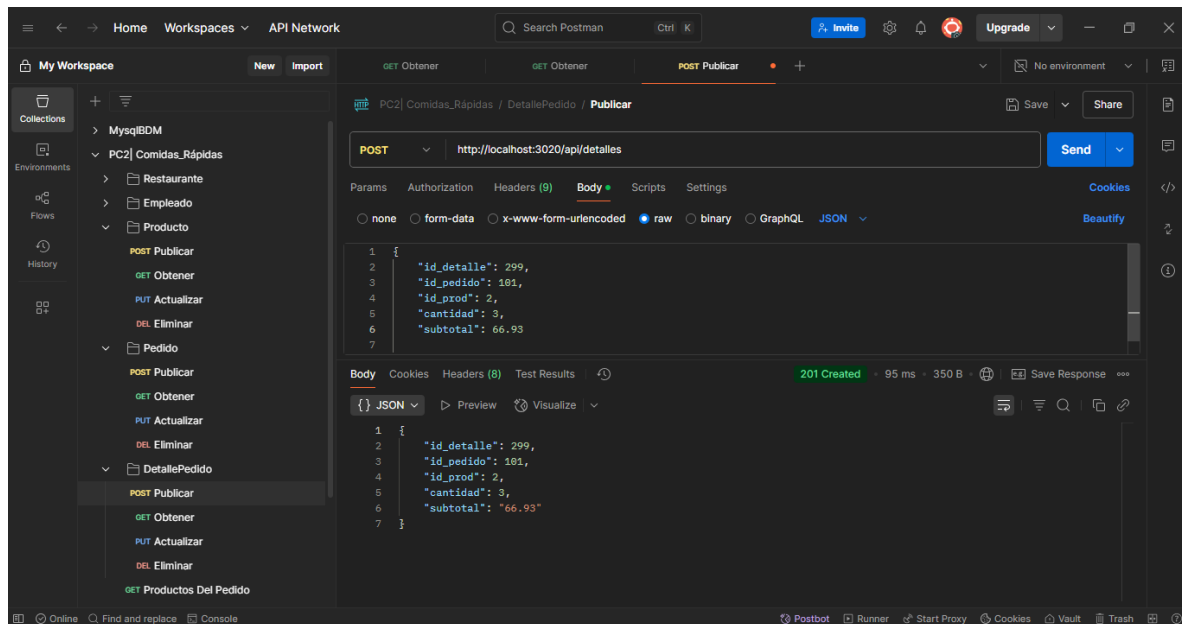
Creamos un API capaz de hacer una eliminación de un **pedido** ya creado, donde solo por medio de la ruta ingresamos el id de este mismo; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en el intento de captura exitosa, y si no de lo contrario nos arrojará que se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id del **pedido** a eliminar:



CREATE datos detalle_pedido: >_http://localhost:3020/api/detalles_<

```
215 //////////////////////////////////////////////////
216 // API CREATE tabla: detalle_pedido PUBLICAR [POST]
217 app.post('/api/detalles', async (req, res) => {
218   const { id_detalle, id_pedido, id_prod, cantidad, subtotal } = req.body;
219   const query = 'INSERT INTO detalle_pedido (id_detalle, id_pedido, id_prod, cantidad, subtotal) VALUES ($1, $2, $3, $4, $5) RETURNING *';
220   try {
221     const result = await connection.query(query, [id_detalle, id_pedido, id_prod, cantidad, subtotal]);
222     res.status(201).json(result.rows[0]);
223   } catch (err) {
224     res.status(500).json({ message: 'Error al crear el detalle', error: err.message });
225   }
226 });
```

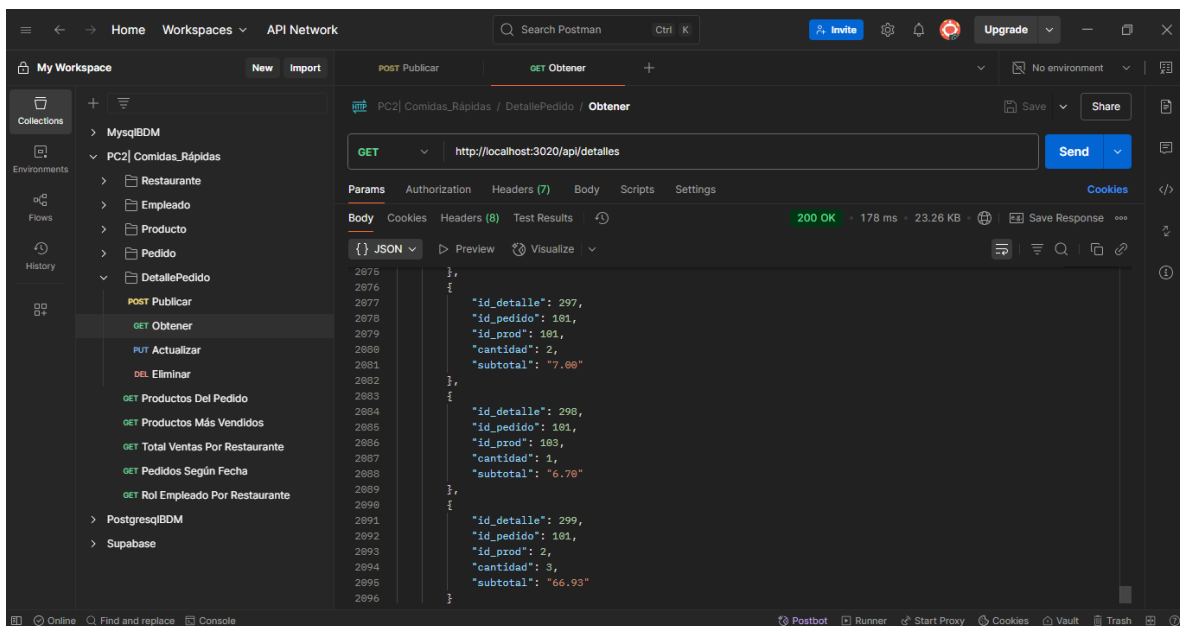
Creamos un API capaz de hacer un registro de ingreso de un **detalle de pedido**, donde todos sus parámetros son ingresados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarnos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y como queremos ingresar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



READ datos detalle_pedido: >_http://localhost:3020/api/detalles_<

```
227 // API READ tabla: detalle_pedido OBTENER [GET]
228 app.get('/api/detalles', async (req, res) => {
229   try {
230     const result = await connection.query('SELECT * FROM detalle_pedido');
231     res.status(200).json({ success: true, data: result.rows });
232   } catch (err) {
233     res.status(500).json({ success: false, message: err.message });
234   }
235 });
```

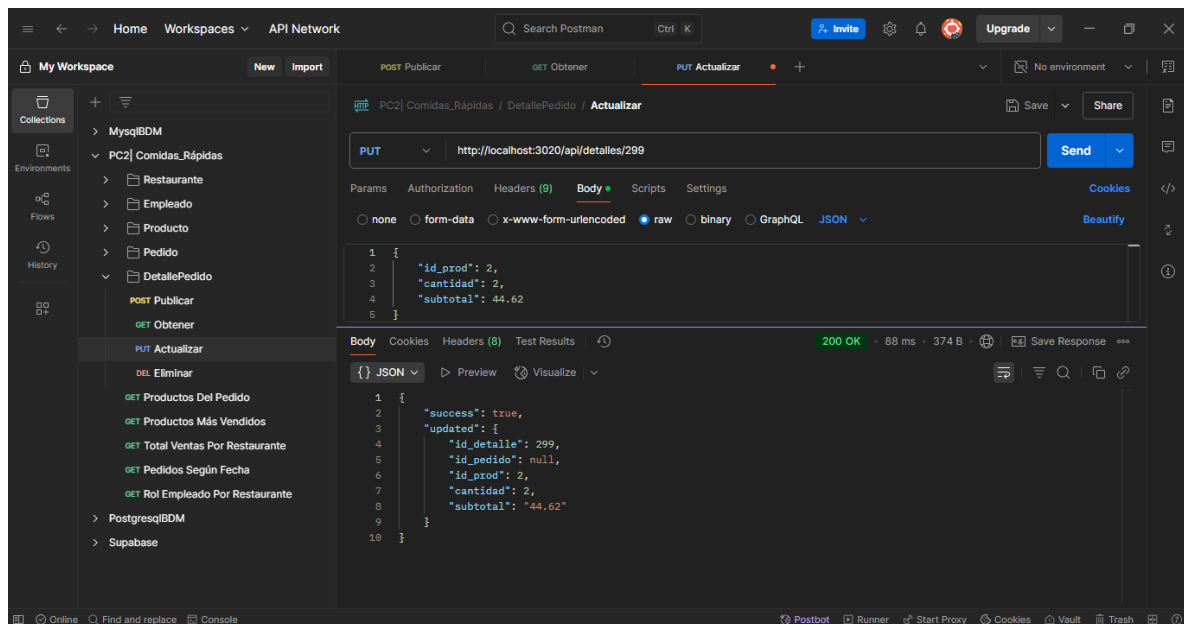
Creamos un API capaz de botarnos todos los registros existentes de todos los **detalles de pedido**, donde simplemente le pasamos una sentencia SQL dentro del intento de captura exitosa, y si no de lo contrario nos lanza que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, y si es correcta la sintaxis, nos arroja todos los **detalles de pedido** registrados hasta el momento:



UPDATE datos detalles_pedido: >_http://localhost:3020/api/detalles/##_<

```
236 // API UPDATE tabla: detalle_pedido ACTUALIZAR [PUT]
237 app.put('/api/detalles/:id', async (req, res) => {
238   const { id } = req.params;
239   const { id_pedido, id_prod, cantidad, subtotal } = req.body;
240   const query = 'UPDATE detalle_pedido SET id_pedido = $1, id_prod = $2, cantidad = $3, subtotal = $4 WHERE id_detalle = $5 RETURNING *';
241   try {
242     const result = await connection.query(query, [id_pedido, id_prod, cantidad, subtotal, id]);
243     if (result.rowCount === 0) {
244       return res.status(404).json({ success: false, message: `Detalle con ID ${id} no encontrado` });
245     }
246     res.status(200).json({ success: true, updated: result.rows[0] });
247   } catch (err) {
248     res.status(500).json({ success: false, error: err.message });
249   }
250 });
```

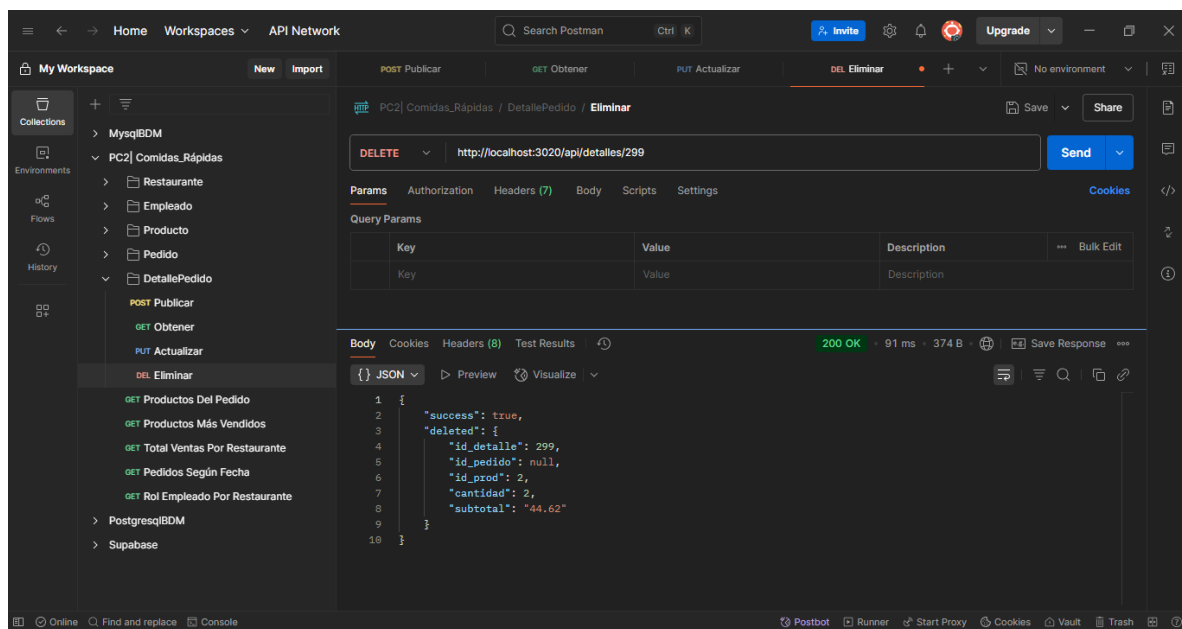
Creamos un API capaz de hacer una sobreescritura de un ingreso de un **detalle de pedido** ya creado, donde todos sus parámetros son modificados; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en una variable dependiente a la estructura de la tabla; para luego hacer un intento de captura exitosa, y si no de lo contrario arrojarlos que no se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id a modificar, y como queremos modificar hacemos uso de la estructura para indicar los valores de cada columna de la tabla:



DELETE datos detalle_pedido: >_http://localhost:3020/api/detalles/##_<

```
251 // API DELETE tabla: detalle_pedido ELIMINAR [DELETE]
252 app.delete('/api/detalles/:id', async (req, res) => {
253   const { id } = req.params;
254   const query = 'DELETE FROM detalle_pedido WHERE id_detalle = $1 RETURNING *';
255   try {
256     const result = await connection.query(query, [id]);
257     if (result.rowCount === 0) {
258       return res.status(404).json({ success: false, message: 'No existe detalle con ID ${id}' });
259     }
260     res.status(200).json({ success: true, deleted: result.rows[0] });
261   } catch (err) {
262     res.status(500).json({ success: false, error: err.message });
263   }
264 });
```

Creamos un API capaz de hacer una eliminación de un **detalle de pedido** ya creado, donde solo por medio de la ruta ingresamos el id de este mismo; y como todo funciona por medio de una consulta SQL, lo que hacemos es guardarla en el intento de captura exitosa, y si no de lo contrario nos arrojamons que se pudo lograr el registro (claro teniendo en cuenta los tipos de errores, para ver en que lo estamos configurando mal). Ya en Postman utilizamos la dirección creada al principio del app.get, junto al id del **detalle de pedido** a eliminar:



Consultas Nativas: Con Ruta Específica, y Petición De Postman

*Cabe recalcar que como son consultas SQL que obtenemos de las tablas (jalamos información) son automáticamente todas peticiones de tipo **[GET]***

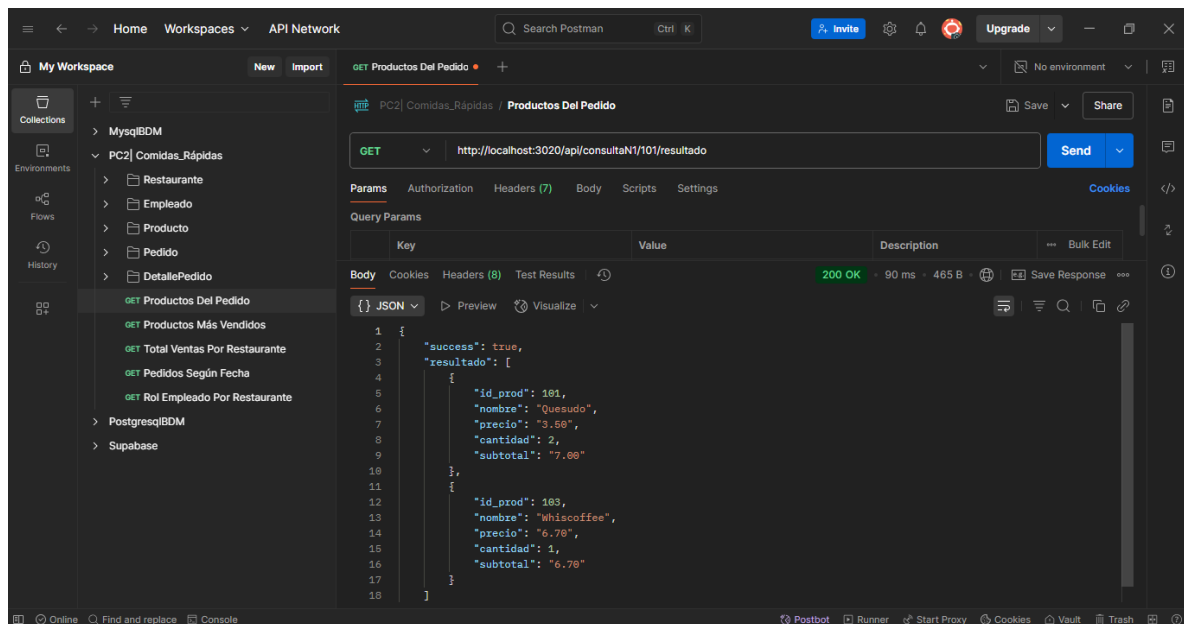
Obtener todos los productos de un pedido específico:

```
SELECT p.id_prod, p.nombre, p.precio, dp.cantidad, dp.subtotal FROM detalle_pedido dp JOIN producto p ON dp.id_prod = p.id_prod WHERE dp.id_pedido = $1
```

Se selecciona los valores de un detalle de pedido, donde este mismo sea de un pedido, al que usuario quiera obtener; se indica con el respectivo id del pedido a averiguar

```
//CONSULTA NATIVA: Obtener todos los productos de un pedido específico [GET]
app.get('/api/consultaN1/:id_pedido_especifico/resultado', async (req, res) => {
  const { id_pedido_especifico } = req.params;
  const query = 'SELECT p.id_prod, p.nombre, p.precio, dp.cantidad, dp.subtotal FROM detalle_pedido dp JOIN producto p ON dp.id_prod = p.id_prod WHERE dp.id_pedido = $1';
  try {
    const result = await connection.query(query, [id_pedido_especifico]);
    res.status(200).json({ success: true, resultado: result.rows });
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
});
```

Podemos obtener los resultados por la consulta nativa solicitada, por medio de un API que de una ruta para poder diferenciarla de todas las demás; donde esta requiere de la estructura previa de la tabla (columnas), es por ello que requiere parámetros y es guardada en una variable, que junto a los parámetros son ingresados, en un try catch, donde evalúa si efectivamente está bien definidos los parámetros. Y esto lo podemos evidenciar en Postman:



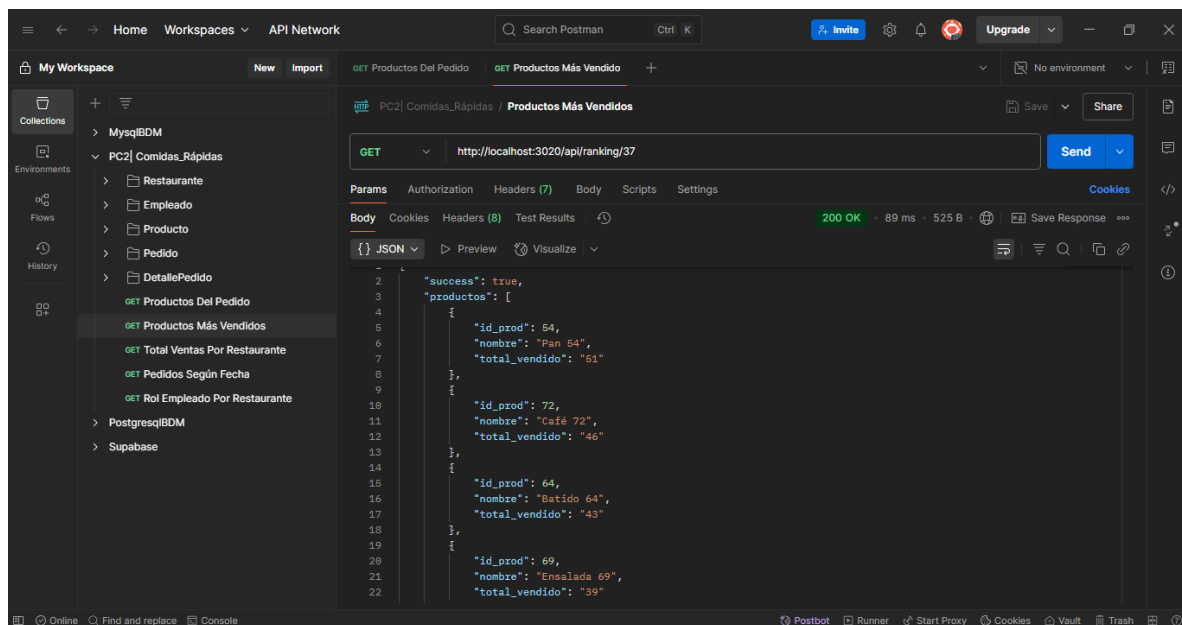
Obtener los productos más vendidos (más de X unidades):

```
SELECT p.id_prod, p.nombre, SUM(dp.cantidad) AS total_vendido FROM detalle_pedido
dp JOIN producto p ON dp.id_prod = p.id_prod GROUP BY p.id_prod, p.nombre HAVING
SUM(dp.cantidad) > $1 ORDER BY total_vendido DESC
```

Se listan los productos que más se han vendido por medio de su id y su nombre, y se van sumando todas sus cantidades en los pedidos registrados. Aun así, solo se muestran los que hayan vendido igual o más unidades al número que el usuario haya indicado por medio del parámetro de variable (\$1). El resultado se ordena del más vendido al de menor cantidad. Se van filtrando por un mínimo de ventas.

```
//CONSULTA NATIVA: Obtener los productos más vendidos (más de X unidades) [GET]
app.get('/api/ranking/neto', async (req, res) => {
  const { neto } = req.params;
  const query = `SELECT p.id_prod, p.nombre, SUM(dp.cantidad) AS total_vendido FROM detalle_pedido dp JOIN producto p ON dp.id_prod = p.id_prod GROUP BY p.id_prod, p.nombre HAVING SUM(dp.cantidad) > $1 ORDER BY total_vendido DESC`;
  try {
    const result = await connection.query(query, [neto]);
    res.status(200).json({ success: true, productos: result.rows });
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
});
```

Podemos obtener los resultados por la consulta nativa solicitada, por medio de un API que de una ruta para poder diferenciarla de todas las demás; donde esta requiere de la estructura previa de la tabla (columnas), es por ello que requiere parámetros y es guardada en una variable, que junto a los parámetros son ingresados, en un try catch, donde evalúa si efectivamente está bien definidos los parámetros. Y esto lo podemos evidenciar en Postman:



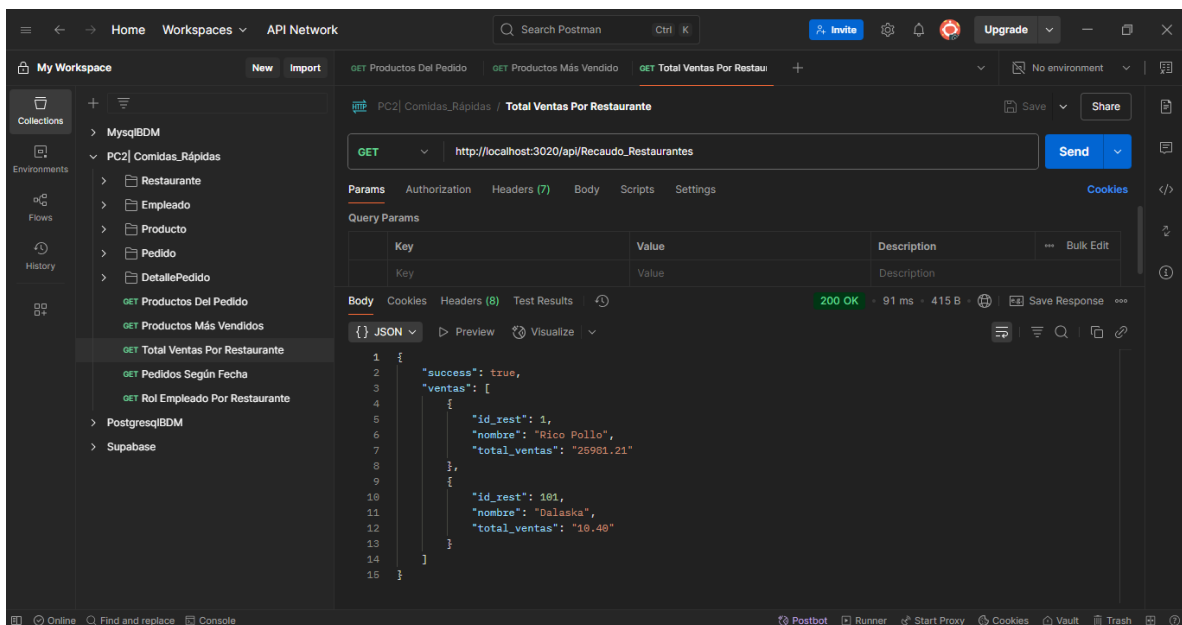
Obtener el total de ventas por restaurante:

```
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 ORDER BY total_ventas
DESC
```

Se selecciona el id del restaurante a evaluar y su respectivo nombre, y nos arroja el total de ventas de éste mismo, por medio de la suma de todos sus de pedidos hechos en cada restaurante, para al final mostrar por orden descendente.

```
//CONSULTA NATIVA: Obtener el total de ventas por restaurante [GET]
app.get('/api/Recaudo_Restaurantes', async (req, res) => {
  try {
    const result = await connection.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 ORDER BY total_ventas DESC');
    res.status(200).json({ success: true, ventas: result.rows });
  } catch (err) {
    res.status(500).json({ success: false, message: err.message });
  }
});
```

Podemos obtener los resultados por la consulta nativa solicitada, por medio de un API que de una ruta para poder diferenciarla de todas las demás; donde ésta simplemente se evalúa con un try catch, donde evalúa si efectivamente está bien definidos los parámetros. Y esto lo podemos evidenciar en Postman:



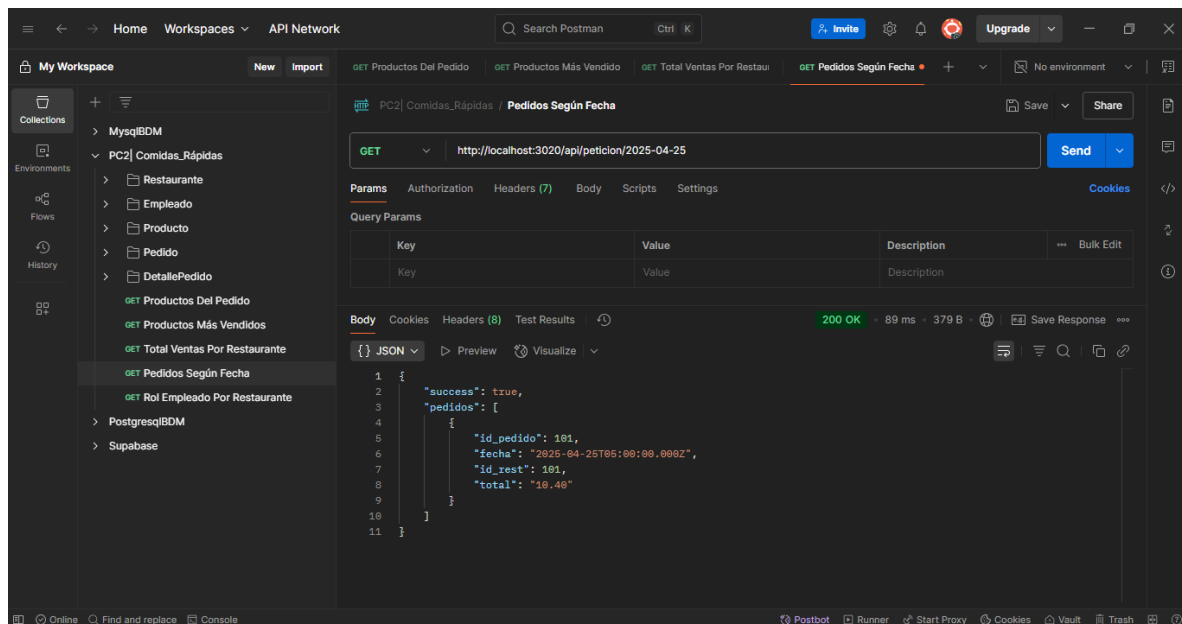
Obtener los pedidos realizados en una fecha específica:

SELECT * FROM pedido WHERE fecha = \$1 ORDER BY id_pedido

Selecciona todos los registros de la tabla pedido, devolviendo todos los pedidos hechos en una fecha específica, que el usuario indique en la ruta de la API (\$1). Y se muestra según el orden de ingreso de esa fecha indicada.

```
299 ///////////////////////////////////////////////////
300 //CONSULTA NATIVA: Obtener los pedidos realizados en una fecha específica [GET]
301 app.get('/api/peticon/:fecha', async (req, res) => {
302   const { fecha } = req.params;
303   const query = 'SELECT * FROM pedido WHERE fecha = $1 ORDER BY id_pedido';
304   try {
305     const result = await connection.query(query, [fecha]);
306     res.status(200).json({ success: true, pedidos: result.rows });
307   } catch (err) {
308     res.status(500).json({ success: false, message: err.message });
309   }
310 });
```

Podemos obtener los resultados por la consulta nativa solicitada, por medio de un API que de una ruta para poder diferenciarla de todas las demás; donde esta requiere de la estructura previa de la tabla (columnas), es por ello que requiere parámetros y es guardada en una variable, que junto a los parámetros son ingresados, en un try catch, donde evalúa si efectivamente está bien definidos los parámetros. Y esto lo podemos evidenciar en Postman:



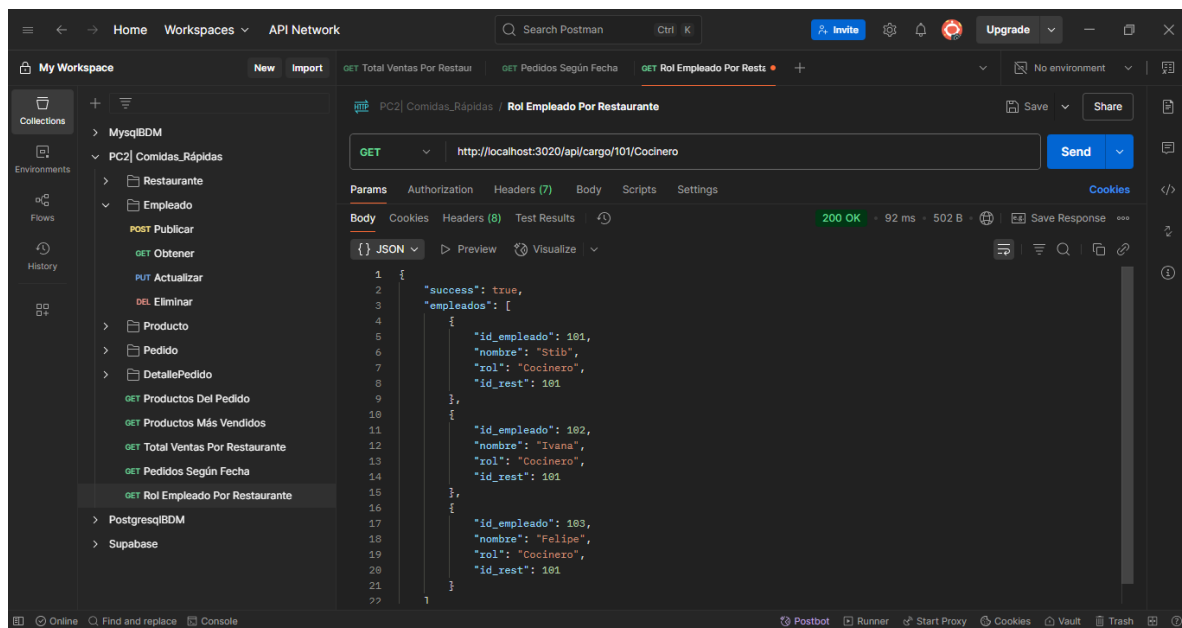
Obtener los empleados por rol en un restaurante:

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

Selecciona todos los registros de la tabla empleados, y devuelve solo aquellos que el usuario le indique el id del restaurante (\$1), y el rol que se quiera investigar (\$2), al igual que el anterior si muestra más de una coincidencia muestra por orden de registro en la tabla.

```
311 ///////////////////////////////////////////////////
312 //CONSULTA NATIVA: Obtener los empleados por rol en un restaurante [GET]
313 app.get('/api/cargo/:id_rest/:rol', async (req, res) => {
314   const { id_rest, rol } = req.params;
315   const query = 'SELECT * FROM empleado WHERE id_rest = $1 AND rol = $2';
316   try {
317     const result = await connection.query(query, [id_rest, rol]);
318     res.status(200).json({ success: true, empleados: result.rows });
319   } catch (err) {
320     res.status(500).json({ success: false, message: err.message });
321   }
322 });
323 ///////////////////////////////////////////////////
```

Podemos obtener los resultados por la consulta nativa solicitada, por medio de un API que de una ruta para poder diferenciarla de todas las demás; donde esta requiere de la estructura previa de la tabla (columnas), es por ello que requiere parámetros y es guardada en una variable, que junto a los parámetros son ingresados, en un try catch, donde evalúa si efectivamente está bien definidos los parámetros. Y esto lo podemos evidenciar en Postman:



Anexos

100 Registros Por Tabla:



restaurantes_insert.
sql



empleados_insert.s
ql



insert_productos.sq
l



pedidos_insert.sql



detalle_pedidos_ins
ert.sql

Modelo De Base De Datos De Supabase:



Ficheros de la base de datos:



conexion.js



code.js

Colección de Postman:



PC2-
Comidas_Rápidas.p