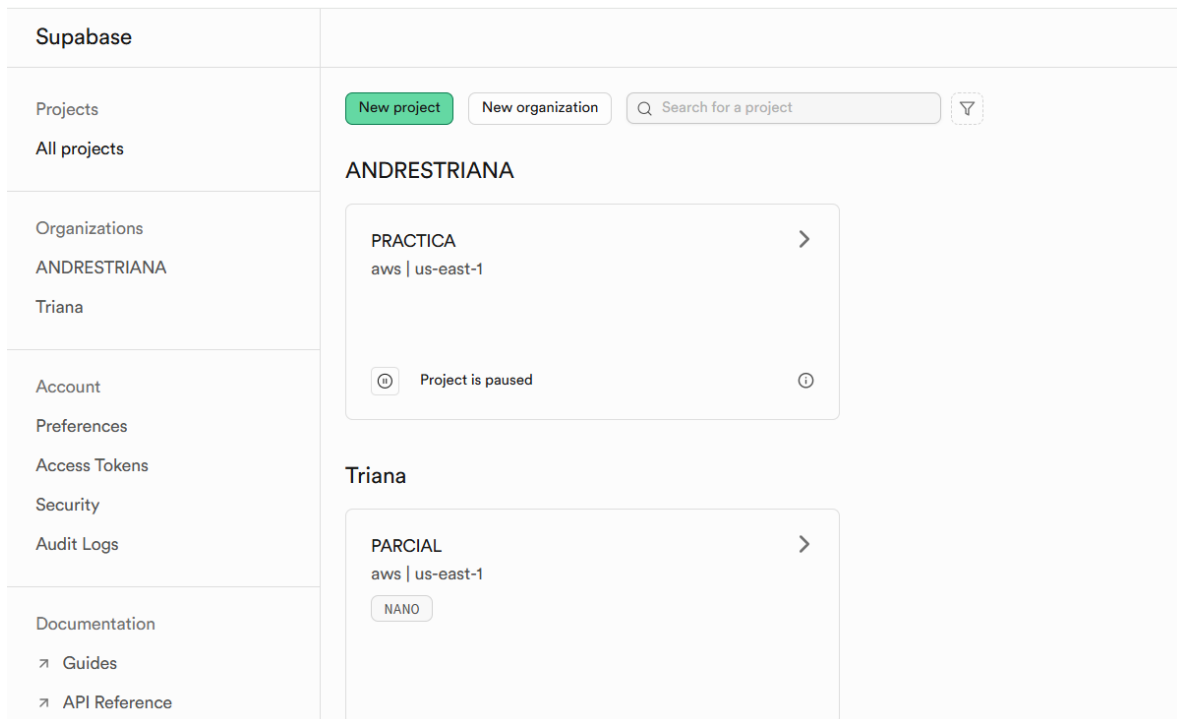


PARCIAL 2 CORTE

ANDRES FELIPE TRIANA TORRES

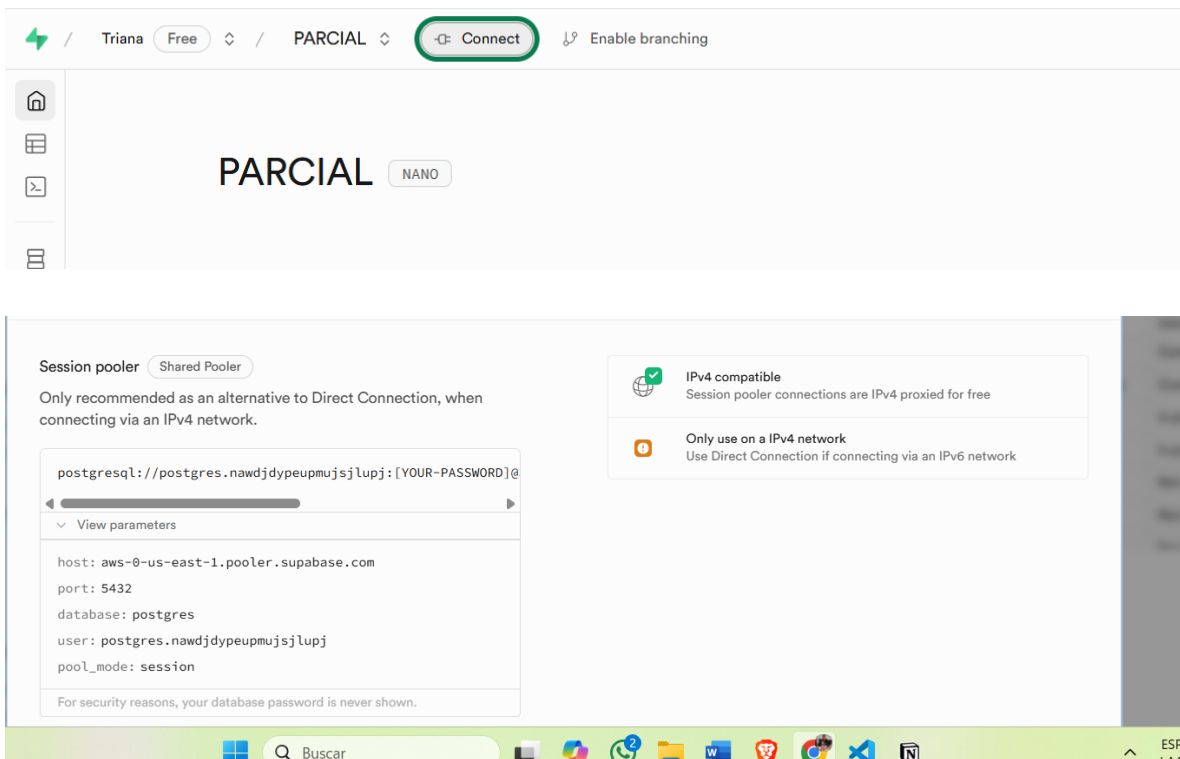
BASES DE DATOS MASIVAS

1. Ingresamos a la plataforma de Supabase y procedemos a crear un nuevo proyecto desde el panel principal.



2. Una vez creado el proyecto en Supabase, hacemos clic en la opción **“Connect”** para obtener los datos de conexión a la base de datos. Supabase proporciona una URL tipo **Session Pooler**, que se utiliza para conectarse desde clientes externos como **pgAdmin**.

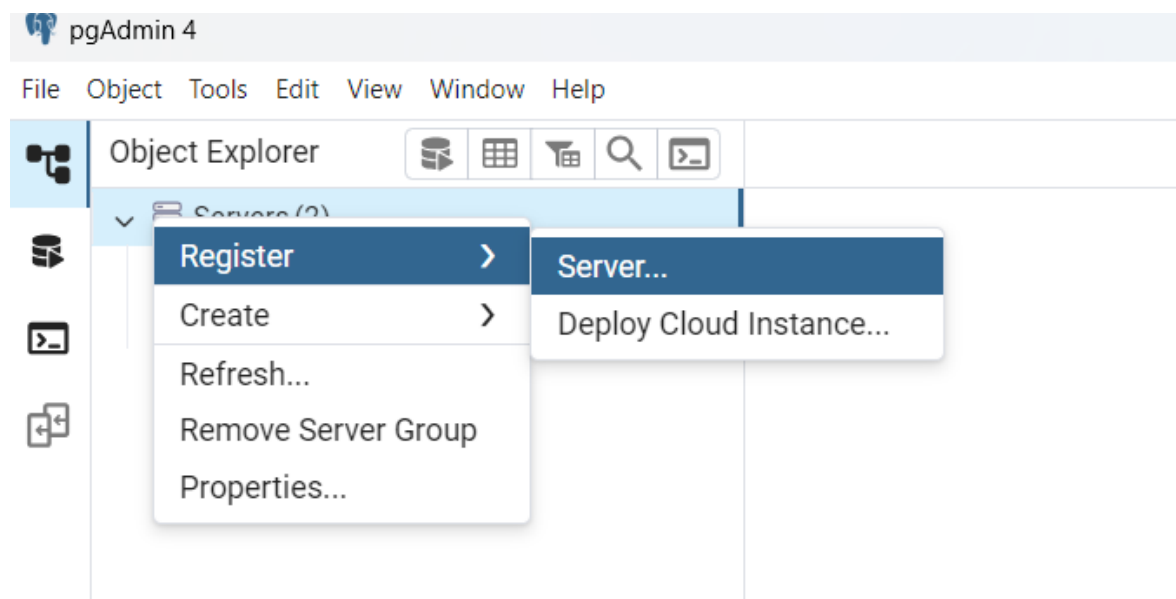
si deseas ver más detalles o copiar los parámetros de conexión por separado, puedes hacer clic en **“View Parameters”**, donde también se muestran enlaces útiles para conectarse.



3.Después de obtener los datos de conexión en Supabase, abrimos **pgAdmin** para registrar un nuevo servidor. Para ello, hacemos clic derecho sobre la opción "**Servers**", seleccionamos "**Register**" y luego "**Server...**". Esto abrirá una ventana emergente donde debemos configurar los parámetros de conexión.

En la pestaña **General**, asignamos un nombre descriptivo al servidor, como por ejemplo "PARCIAL". Luego, en la pestaña **Connection**, llenamos los campos con la información obtenida del *Session Pooler* de Supabase: en **Host name/address** colocamos el host, en **Port** dejamos el valor por defecto 5432, en **Username** ingresamos el usuario proporcionado y en **Password** escribimos la contraseña que configuramos al crear el proyecto. Finalmente, en **Maintenance database** colocamos postgres.

Una vez completados todos los campos, hacemos clic en **Save**. Si los datos son correctos, se establecerá la conexión y podremos acceder a nuestra base de datos de Supabase directamente desde pgAdmin.



Register - Server

×

General

Connection

Parameters

SSH Tunnel

Advanced

Tags

Name

PARCIAL

Server group

Servers

▼

Background

×

Foreground

×

Connect now?

☒

Comments

i

?

×

Close

↺

Reset

Save

Register - Server

General

Connection

Parameters

SSH Tunnel

Advanced

Tags

Host name/address

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

Port

5432

Maintenance database

postgres

Username

postgres.nawdjdypeupmuisilupi

Kerberos authentication?

☐

Password

....

Save password?

☐

Role

Service

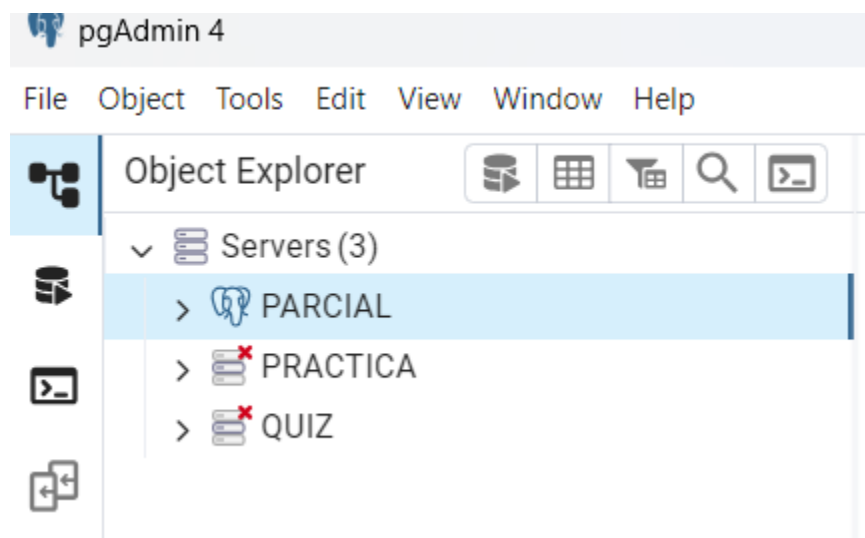
i

?

Close

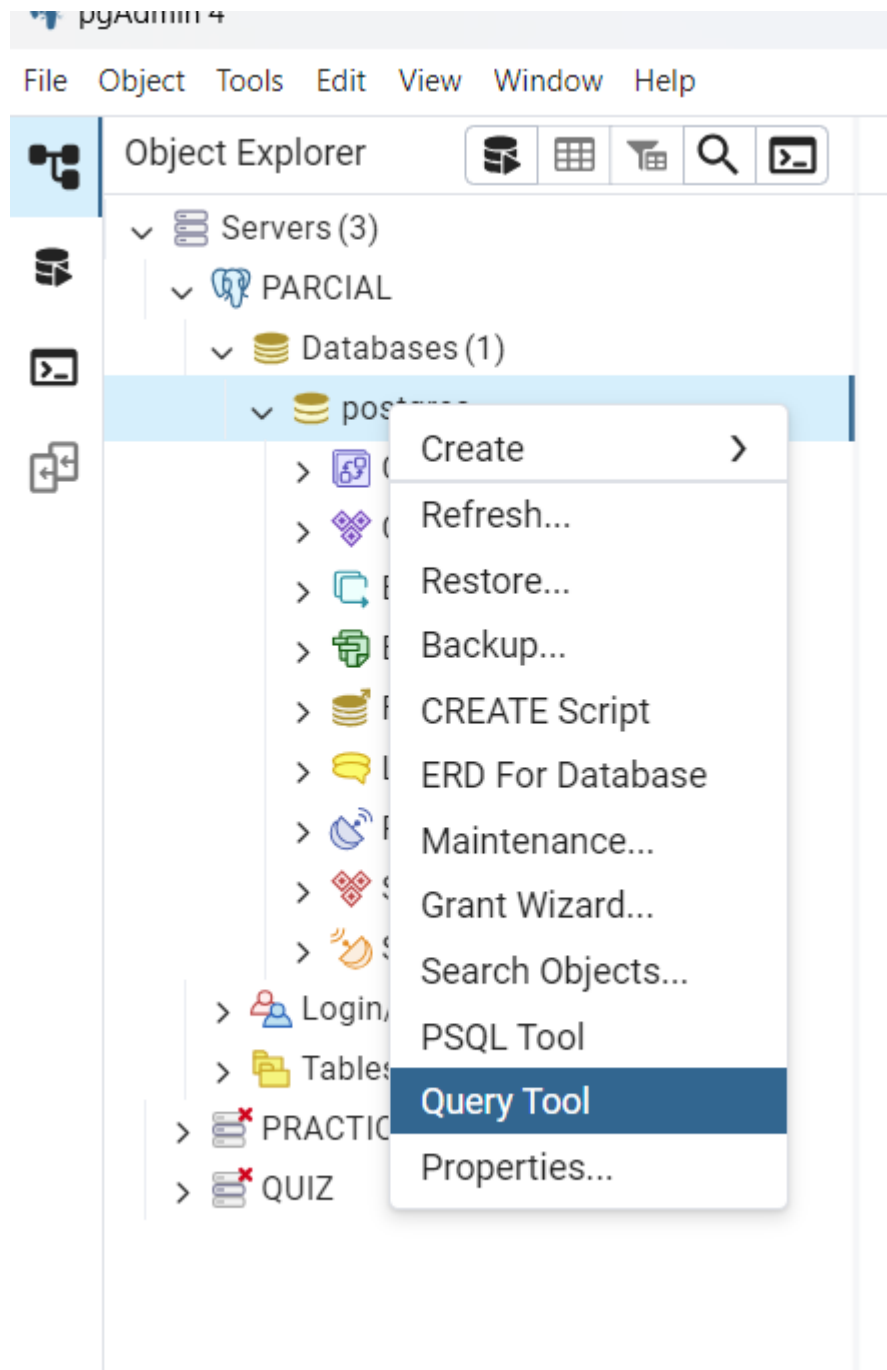
Reset

Save



4. Para crear las tablas en la base de datos desde **pgAdmin**, primero accedemos a la base de datos correspondiente (en este caso, se llama parcial). Hacemos clic sobre su nombre para seleccionarla. Luego, en el panel superior o haciendo clic derecho sobre la base de datos, seleccionamos la opción **"Query Tool"** (herramienta de consultas).

Esta acción abrirá una ventana emergente en la que podremos escribir comandos SQL. Desde allí es posible crear tablas, insertar datos o realizar cualquier tipo de consulta.



5. Creamos las tablas en la base de datos y revisamos que la estructura (o el modelo UML) esté bien en Supabase.

Query Query History

```
1  CREATE TABLE restaurante (  
2      id_rest INT PRIMARY KEY,  
3      nombre VARCHAR(100) NOT NULL,  
4      ciudad VARCHAR(100) NOT NULL,  
5      direccion VARCHAR(150) NOT NULL,  
6      fecha_apertura DATE NOT NULL  
7  );
```

```
CREATE TABLE empleado (  
    id_empleado INT 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)  
);
```

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

```

✓ CREATE TABLE pedido (
    id_pedido INT 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)
);

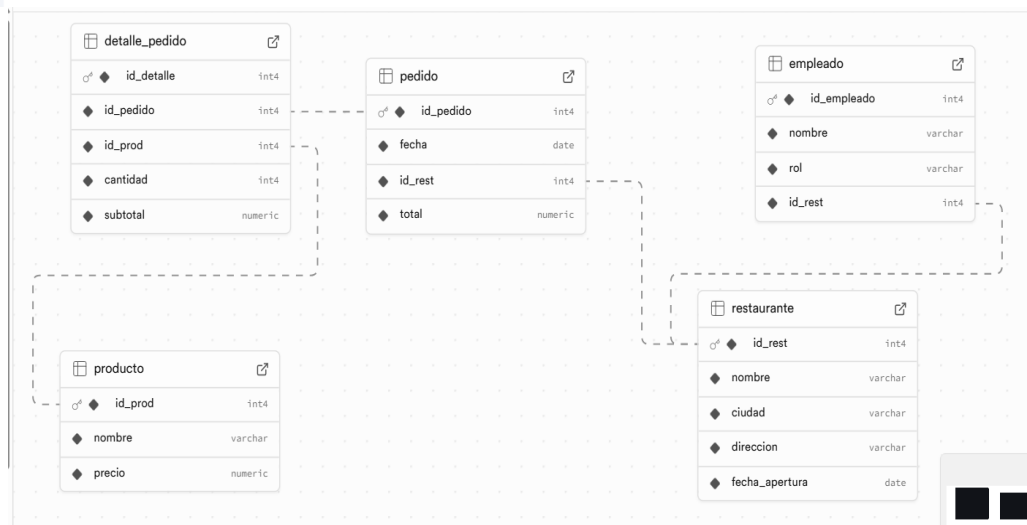
```

Query Query History

```

1 ✓ CREATE TABLE detalle_pedido (
2     id_detalle INT PRIMARY KEY,
3     id_pedido INT NOT NULL,
4     id_prod INT NOT NULL,
5     cantidad INT NOT NULL,
6     subtotal NUMERIC(10,2) NOT NULL,
7     FOREIGN KEY (id_pedido) REFERENCES pedido(id_pedido),
8     FOREIGN KEY (id_prod) REFERENCES producto(id_prod)
9 );

```



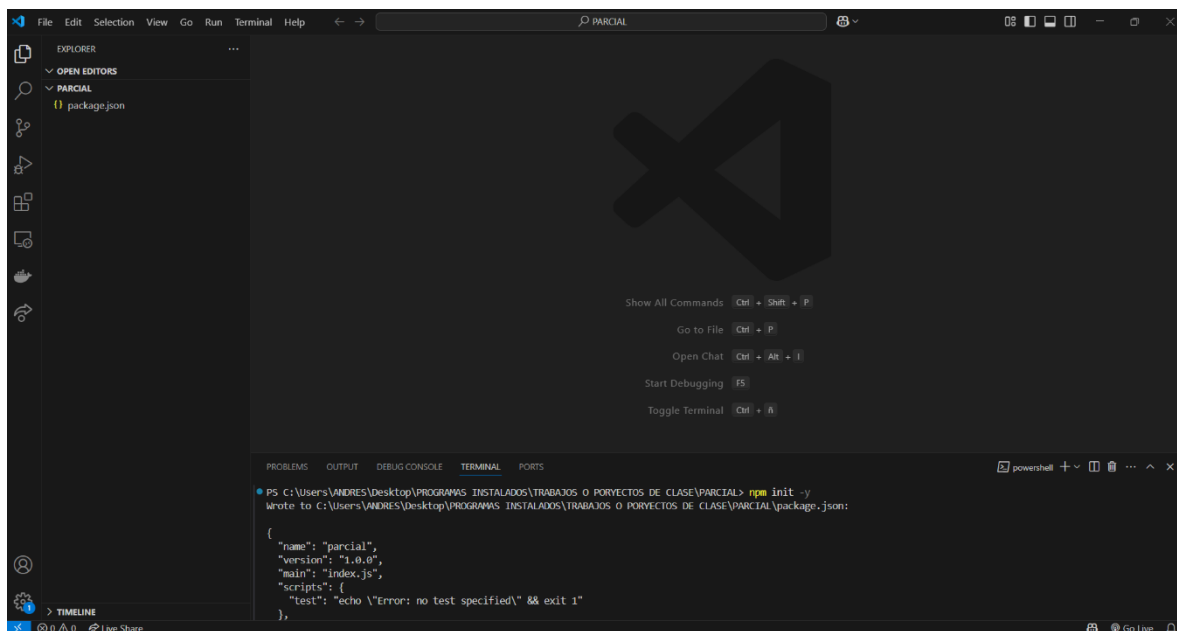
6. Creamos una carpeta nueva y la abrimos con Visual Studio Code para empezar a trabajar en el ejercicio.

Nombre	Fecha de modificación	Tipo	Tamaño
BASESDEDATOS	9/04/2025 3:04 p. m.	Carpeta de archivos	
PARCIAL	23/04/2025 10:47 a. m.	Carpeta de archivos	
PRACTICA	14/04/2025 9:26 p. m.	Carpeta de archivos	
QUIZ_BASES_DE_DATOS	11/04/2025 12:48 p. m.	Carpeta de archivos	

7. Abrimos la terminal en Visual Studio Code y escribimos el siguiente comando:

```
npm init -y
```

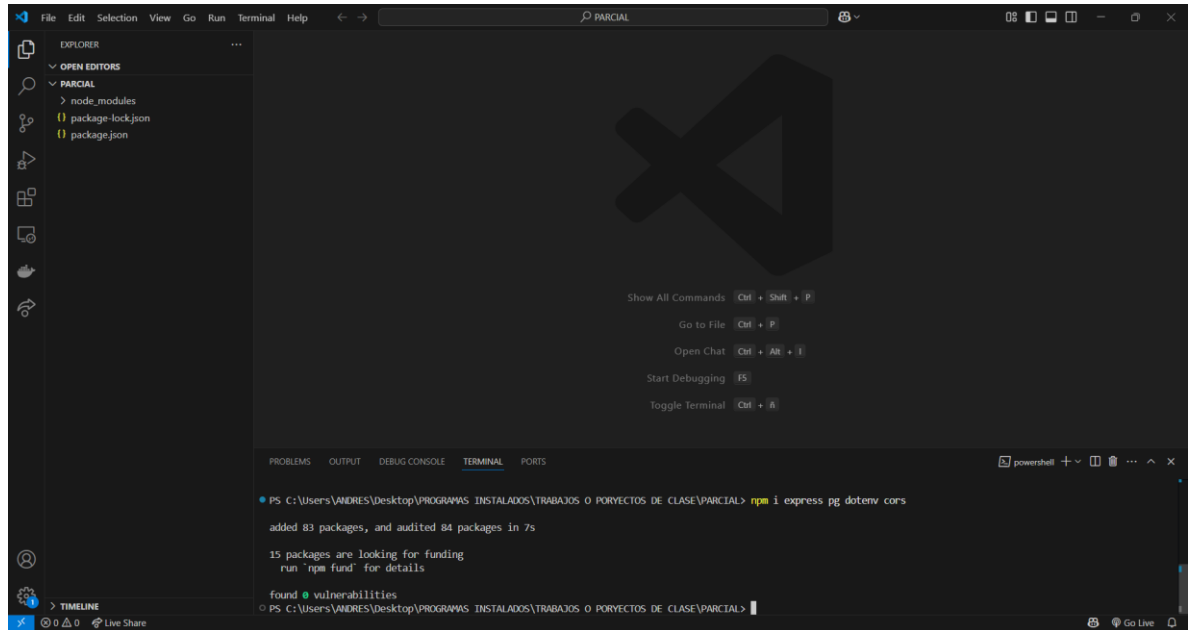
Este comando sirve para iniciar un proyecto en Node.js de forma rápida. Crea automáticamente el archivo package.json con la configuración por defecto.



8. Después, escribimos este comando:

```
npm i express pg dotenv cors
```

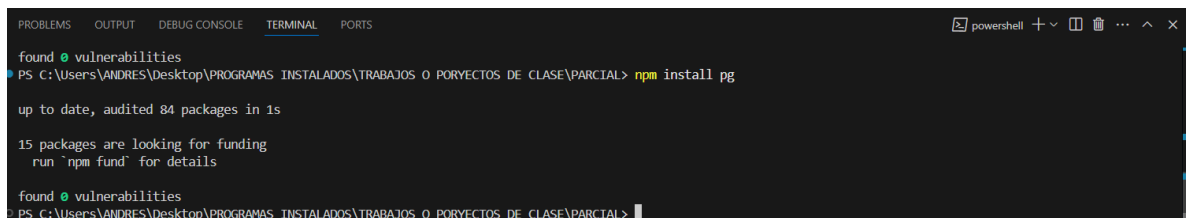
Este comando instala los paquetes que vamos a necesitar para crear el servidor en Node.js y conectarlo a una base de datos PostgreSQL.



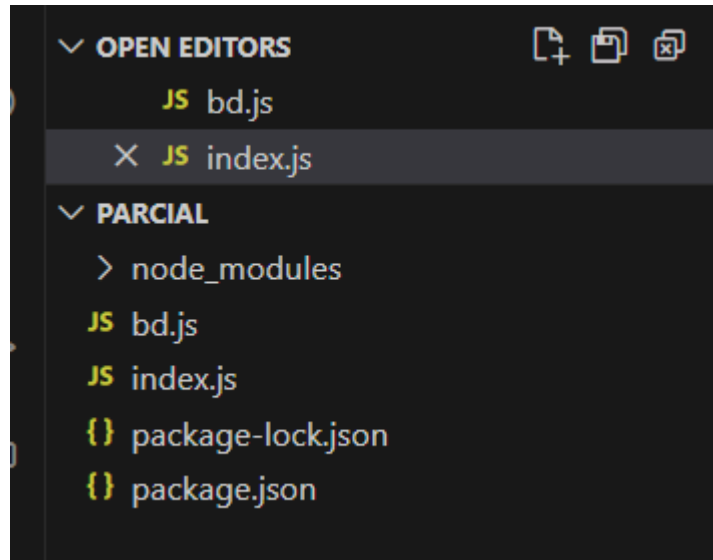
9. Luego, ejecutamos este comando:

```
npm install pg
```

Este paquete nos permite conectar nuestra app de Node.js con una base de datos PostgreSQL y hacer consultas directamente desde el código.



10. Luego, abrimos dos archivos nuevos donde vamos a escribir el código que necesitamos para el proyecto.

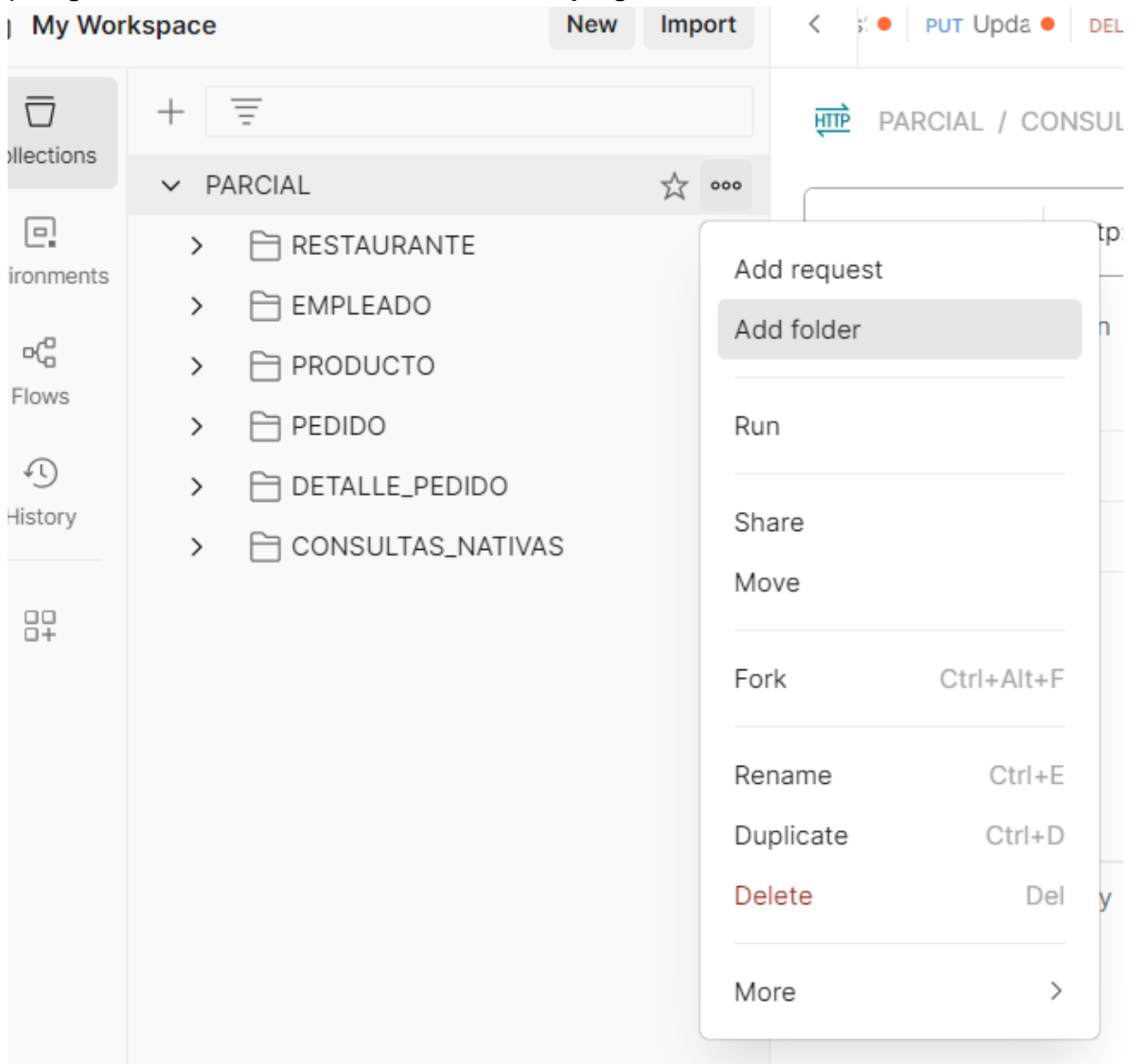


11. El siguiente código permite configurar y verificar la conexión a una base de datos PostgreSQL alojada en Supabase.

```
1  const { Pool } = require('pg');
2
3
4  const pool = new Pool({
5    host: 'aws-0-us-east-1.pooler.supabase.com',
6    port: 5432,
7    user: 'postgres.nawdjdypeupmujsjlupj',
8    password: '1234',
9    database: 'postgres',
10   ssl: { rejectUnauthorized: false }
11 });
12
13
14 pool.connect((err, client, release) => {
15   if (err) {
16     console.error('Error conectando con la base de datos', err.stack);
17   } else {
18     console.log('Conectado a la base de datos');
19     release();
20   }
21 });
22
23 module.exports = pool;
```

12. Como siguiente paso, abrimos Postman y hacemos clic en el botón "+" para crear una nueva pestaña de solicitud. Luego, seleccionamos la opción "REST API (Basic)", lo cual nos permite acceder a los cuatro métodos principales: **GET**, **POST**, **PUT** y **DELETE**. Estos métodos serán utilizados para interactuar con nuestra API.

A continuación, creamos cinco carpetas, cada una correspondiente a una de nuestras tablas. Dentro de cada carpeta, agregamos las solicitudes **GET**, **POST**, **PUT** y **DELETE** necesarias para gestionar los datos de forma estructurada y organizada.



13. Después de crear nuestras carpetas, continuamos en el archivo index.js. Como primer paso, importamos las dependencias necesarias (express y db.js para la conexión a la base de datos). Luego, creamos una instancia de Express para iniciar nuestro servidor, configuramos los middlewares para procesar datos en formato JSON y definimos el puerto 3000 para su ejecución.

Esto lo hacemos mediante el siguiente código:

```
1  const express = require('express');
2  const connection = require('./db');
3
4  const app = express();
5
6  app.use(express.json());
7  app.use(express.urlencoded({ extended: true }));
8
9  const PORT = 3000;
```

14. Como siguiente paso, creamos los archivos CRUD dentro de las carpetas correspondientes, los cuales contendrán las rutas de la API necesarias para realizar las consultas y operaciones sobre cada tabla.

a. OBTENER: Este código crea un endpoint GET que consulta todos los registros de la tabla. Si la operación es exitosa, devuelve los datos en formato JSON; en caso de error, responde con un mensaje y el código de estado 500. En resumen, permite obtener todos los datos de la tabla usando una solicitud GET.

```
// ----- CRUD RESTAURANTE -----
// 1. Obtener restaurantes (GET)
app.get('/api/restaurante/obtener', (req, res) => {
  const query = 'SELECT * FROM restaurante';

  connection.query(query, (error, result) => {
    if (error) {
      res.status(500).json({ message: 'Error al recuperar restaurantes', error: error.message });
    } else {
      res.status(200).json({ restaurantes: result.rows });
    }
  });
});
```

b. CREAR: Este código crea un endpoint POST que permite registrar un nuevo en la base de datos. Recibe los datos del restaurante en el cuerpo de la solicitud y los inserta en la tabla correspondiente. Si la operación es exitosa, devuelve un código 201; en caso de error, devuelve un código 500.

```
// 2. Crear restaurante (POST)
app.post('/api/restaurante/guardar', (req, res) => {
  const { id_rest, nombre, ciudad, direccion, fecha_apertura } = req.body;

  const query = 'INSERT INTO restaurante (id_rest, nombre, ciudad, direccion, fecha_apertura) VALUES ($1, $2, $3, $4, $5)';

  connection.query(query, [id_rest, nombre, ciudad, direccion, fecha_apertura], (error, result) => {
    if (error) {
      res.status(500).json({
        message: 'Error creando el restaurante',
        error: error.message
      });
    } else {
      res.status(201).json({ id_rest, nombre, ciudad, direccion, fecha_apertura });
    }
  });
});
```

c. ACTUALIZAR: Este código crea un endpoint PUT que actualiza los datos de una tabla en la base de datos utilizando el ID proporcionado en la URL. Recibe los nuevos datos en el cuerpo de la solicitud y los actualiza en la tabla. Si la actualización es exitosa, devuelve un código 200; si ocurre un error, devuelve un código 500.

```
// 3. Actualizar restaurante (PUT)
app.put('/api/restaurante/actualizar/:id', (req, res) => {
  const { id } = req.params;
  const { nombre, ciudad, direccion, fecha_apertura } = req.body;

  const query = `
    UPDATE restaurante
    SET nombre = $1, ciudad = $2, direccion = $3, fecha_apertura = $4
    WHERE id_rest = $5
  `;

  connection.query(query, [nombre, ciudad, direccion, fecha_apertura, id], (error, result) => {
    if (error) {
      res.status(500).json({ message: 'Error al actualizar restaurante', error: error.message });
    } else if (result.rowCount === 0) {
      res.status(404).json({ message: `No se encontró el restaurante con ID ${id}` });
    } else {
      res.status(200).json({ message: 'Restaurante actualizado correctamente' });
    }
  });
});
```

d. ELIMINAR: Este código crea un endpoint DELETE que elimina un registro de la base de datos utilizando el ID proporcionado en la URL. Si la eliminación es exitosa, devuelve un mensaje con código 200; si ocurre un error, devuelve un código 500.

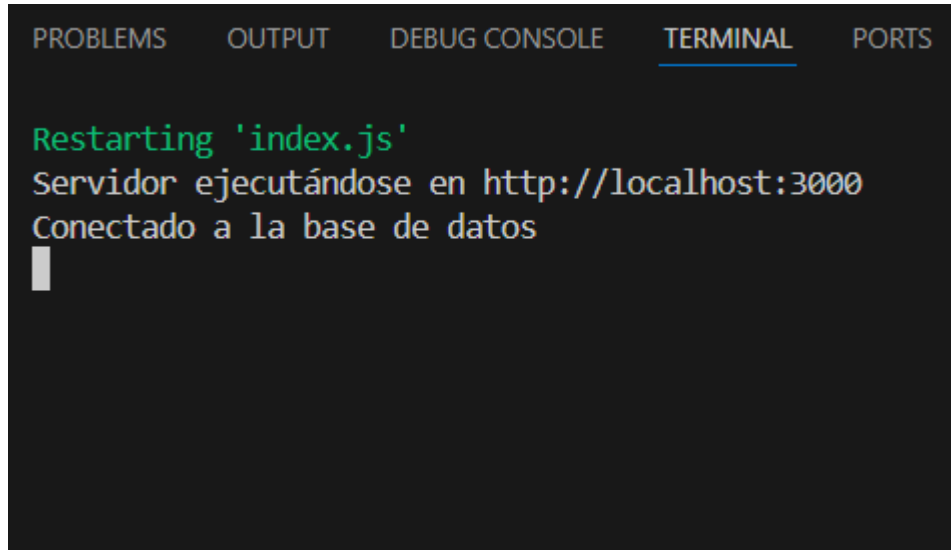
```
// 4. Eliminar restaurante (DELETE)
app.delete('/api/restaurante/eliminar/:id', (req, res) => {
  const { id } = req.params;
  const query = 'DELETE FROM restaurante WHERE id_rest = $1';

  connection.query(query, [id], (error, result) => {
    if (error) {
      res.status(500).json({ message: 'Error al eliminar restaurante', error: error.message });
    } else if (result.rowCount === 0) {
      res.status(404).json({ message: `No se encontró el restaurante con ID ${id}` });
    } else {
      res.status(200).json({ message: 'Restaurante eliminado correctamente' });
    }
  });
});
```

Realizamos lo mismo con los CRUD de empleado, producto, pedido y detalle_pedido, creando los endpoints para cada operación (POST, GET, PUT, DELETE) y conectándolos con las tablas correspondientes en la base de datos.

15. Como siguiente paso, verificamos que todo esté correctamente conectado. Para hacerlo, usamos los siguientes comandos:

1. **node index.js:** Inicia la aplicación en el entorno de Node.js.
 2. **node --watch index.js:** Inicia la aplicación y la mantiene en observación, recargándola automáticamente cuando se realicen cambios en el archivo.
- Si todo está correcto, veremos el siguiente mensaje:



The screenshot shows a terminal window with the following text:

```
Restarting 'index.js'
Servidor ejecutándose en http://localhost:3000
Conectado a la base de datos
```

The terminal window has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS.

16. Para verificar que en postman nuestras consultas están bien entramos a postman y verificamos

a.GET:

The screenshot shows the Postman interface with a GET request to `http://localhost:3000/api/restaurantes/obtener`. The response is a 200 OK status with a JSON body containing an array of restaurant data.

```
1 {
2   "restaurantes": [
3     {
4       "id_rest": 3,
5       "nombre": "Restaurante 3",
6       "ciudad": "Ciudad 4",
7       "direccion": "Calle 66 # 35",
8       "fecha_apertura": "2018-04-30T05:00:00.000Z"
9     },
10    {
11      "id_rest": 4,
12      "nombre": "Restaurante 4",
13      "ciudad": "Ciudad 5",
14      "direccion": "Calle 21 # 45",
15    }
16  ]
17 }
```

b.POST:

The screenshot shows the Postman interface with a POST request to `http://localhost:3000/api/restaurantes/guardar`. The response is a 201 Created status with a JSON body containing the created restaurant data.

```
1 {
2   "id_rest": 52,
3   "nombre": "El Buen Sabor",
4   "ciudad": "Bogotá",
5   "direccion": "Calle 45 #12-34",
6   "fecha_apertura": "2023-05-15"
7 }
```


c.PUT

The screenshot shows the Postman interface with a workspace named "My Workspace". On the left sidebar, under "Collections", the "PARCIAL" collection is expanded, showing a sub-collection "RESTAURANTE" with several endpoints. The "PUT Update data" endpoint is selected. The main panel displays the details of this PUT request. The URL is "http://localhost:3000/api/restaurantes/actualizar/2". The request body is a JSON object:

```
{  "nombre": "La Morciditya",  "ciudad": "Chia",  "direccion": "Calle 163",  "fecha_apertura": "2023-07-20"}
```

. The response is a 200 OK status with a JSON body:

```
{  "message": "Restaurante actualizado correctamente"}
```

. The status bar at the bottom indicates the response time is 673 ms and the body size is 286 B.

d.DELETE

The screenshot shows the Postman interface with the same workspace. The "DELETE data" endpoint under the "RESTAURANTE" collection is selected. The URL is "http://localhost:3000/api/restaurantes/eliminar/52". The request body is empty. The response is a 200 OK status with a JSON body:

```
{  "message": "Restaurante eliminado correctamente"}
```

. The status bar at the bottom indicates the response time is 804 ms and the body size is 284 B.

17. Para hacer consultas nativas se siguen estos pasos:

Obtener todos los productos de un pedido específico

Este comando sirve para traer todos los productos que están relacionados con un pedido, usando su ID. Los pasos que seguimos son:

1. Primero se crea una ruta tipo GET que reciba el id del pedido por la URL (por ejemplo: /api/pedido/5/productos).
2. Luego se extrae ese id desde la URL y se guarda en una variable.
3. Después se hace la consulta en SQL, juntando las tablas detalle_pedido y producto para obtener la información de cada producto (nombre, precio, cantidad, etc.).
4. Por último, se ejecuta la consulta:

-Si hay un error, se responde con un código 500 y un mensaje de error.

-Si todo sale bien, se responde con un 200 y se envía la lista de productos del pedido.

```
//Obtener todos los productos de un pedido específico
app.get('/api/pedido/:id/productos', (req, res) => {
  const { id } = 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
  `;

  connection.query(query, [id], (error, result) => {
    if (error) {
      res.status(500).json({ message: 'Error al obtener productos del pedido', error: error.message });
    } else {
      res.status(200).json({ productos: result.rows });
    }
  });
});
```

Obtener los productos más vendidos (más de X unidades)

Este comando trae una lista de productos que se han vendido más veces que la cantidad que le pasamos por la URL.

Pasos:

1. Se crea una ruta GET que reciba un número (cantidad) por la URL.
2. Se guarda ese número en una variable.
3. Se hace la consulta en SQL:
 - Se hace un JOIN entre detalle_pedido y producto.
 - Se suman las unidades vendidas de cada producto.
 - Se filtran los que superan la cantidad indicada.
 - Se ordenan de mayor a menor.
4. Si hay error, devuelve un 500. Si todo sale bien, devuelve un 200 con la lista de productos.

```
//Obtener los productos más vendidos (más de X unidades)
app.get('/api/productos/mas-vendidos/:cantidad', (req, res) => {
  const { cantidad } = 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
  `;

  connection.query(query, [cantidad], (error, result) => {
    if (error) {
      res.status(500).json({ message: 'Error al obtener productos más vendidos', error: error.message });
    } else {
      res.status(200).json({ productos: result.rows });
    }
  });
});
```

Obtener el total de ventas por restaurante

Este comando obtiene el total de ventas por restaurante, sumando todas las ventas de los pedidos de cada restaurante.

1. Usamos una ruta GET para obtener el total de ventas por restaurante.
2. La consulta SQL:
 - Hace un JOIN entre las tablas pedido y restaurante.
 - Suma las ventas de cada restaurante con SUM(p.total).
 - Agrupa los resultados por id_rest y nombre del restaurante.
 - Ordena los restaurantes de mayor a menor según el total de ventas.
3. Ejecutamos la consulta:
 - Si hay un error, responde con un código 500.
 - Si todo está bien, responde con un 200 y los resultados.

```
3 //Obtener el total de ventas por restaurante
4 app.get('/api/ventas/por-restaurante', (req, res) => {
5   const query = `
6     SELECT r.id_rest, r.nombre AS restaurante, SUM(p.total) AS total_ventas
7     FROM pedido p
8     JOIN restaurante r ON p.id_rest = r.id_rest
9     GROUP BY r.id_rest, r.nombre
10    ORDER BY total_ventas DESC
11  `;
12
13   connection.query(query, (error, result) => {
14     if (error) {
15       res.status(500).json({ message: 'Error al obtener total de ventas', error: error.message });
16     } else {
17       res.status(200).json({ ventas: result.rows });
18     }
19   });
20 });
```

Obtener los pedidos realizados en una fecha específica

Este comando obtiene los pedidos realizados en una fecha específica.

1. Se define una ruta GET que espera el parámetro fecha en la URL.
2. Extraemos la fecha desde la URL y la guardamos en la variable fecha.
3. Definimos la consulta SQL para obtener los pedidos que coincidan con la fecha proporcionada.
4. Ejecutamos la consulta:
 - Si ocurre un error, responde con un código 500.
 - Si todo está bien, responde con un código 200 y los pedidos obtenidos.

```
//Obtener los pedidos realizados en una fecha específica
app.get('/api/pedidos/fecha/:fecha', (req, res) => {
  const { fecha } = req.params;
  const query = `
    SELECT * FROM pedido WHERE fecha = $1
  `;

  connection.query(query, [fecha], (error, result) => {
    if (error) {
      res.status(500).json({ message: 'Error al obtener pedidos por fecha', error: error.message });
    } else {
      res.status(200).json({ pedidos: result.rows });
    }
  });
});
```

Obtener los empleados por rol en un restaurante

Este comando obtiene los empleados de un restaurante con un rol específico.

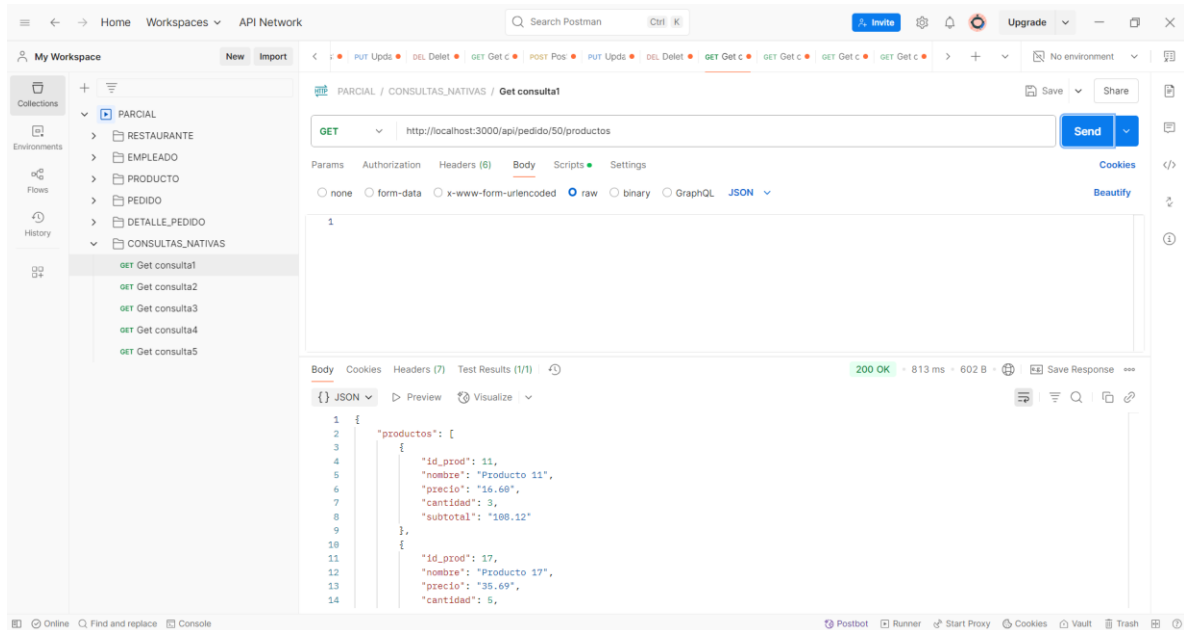
1. Definimos la ruta GET, que espera los parámetros `id_rest` y `rol` en la URL.
2. Extraemos los parámetros `id_rest` y `rol` de la URL y los guardamos en las variables correspondientes.
3. Definimos la consulta SQL para obtener los empleados con esos parámetros.
4. Ejecutamos la consulta:
 - Si hay un error, se responde con un código 500.
 - Si todo está bien, se responde con un código 200 y la lista de empleados.

```
//Obtener los empleados por rol en un restaurante
app.get('/api/empleados/por-rol/:id_rest/:rol', (req, res) => {
  const { id_rest, rol } = req.params;
  const query = `
    SELECT * FROM empleado
    WHERE id_rest = $1 AND rol = $2
  `;

  connection.query(query, [id_rest, rol], (error, result) => {
    if (error) {
      res.status(500).json({ message: 'Error al obtener empleados por rol', error: error.message });
    } else {
      res.status(200).json({ empleados: result.rows });
    }
  });
});
```

18. verificamos que todas las consultas estén bien

Consulta 1



PARCIAL / CONSULTAS_NATIVAS / Get consulta1

GET http://localhost:3000/api/pedido/50/productos

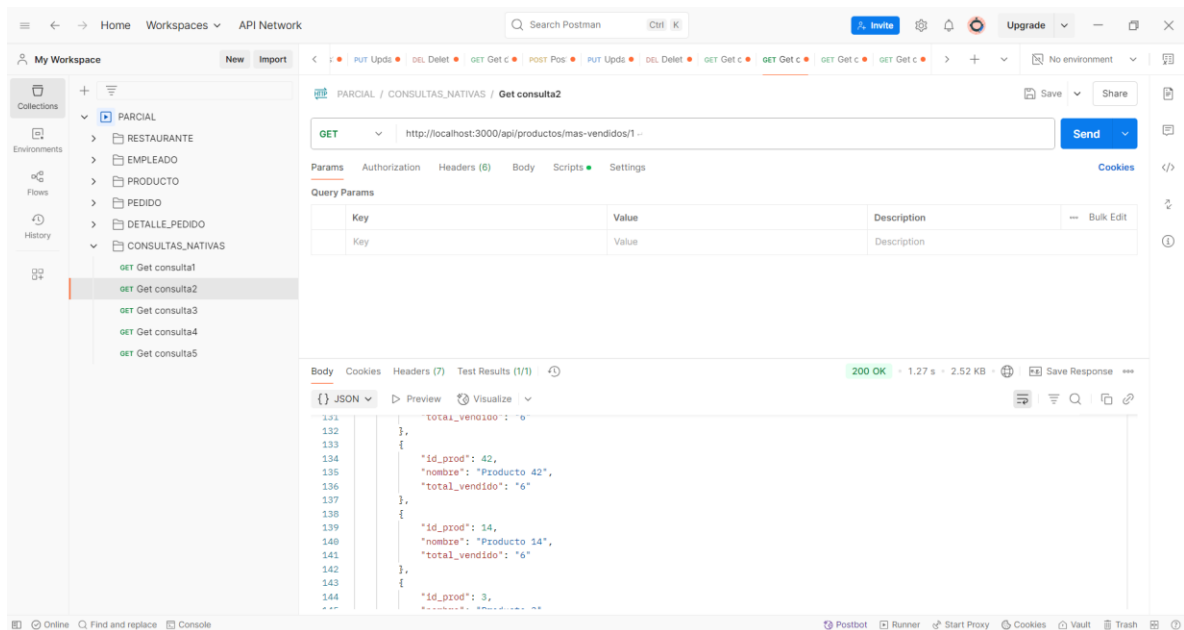
Params Authorization Headers (6) Body Scripts Settings

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

200 OK - 813 ms - 602 B

```
1 {
2   "productos": [
3     {
4       "id_prod": 11,
5       "nombre": "Producto 11",
6       "precio": "16.68",
7       "cantidad": 3,
8       "subtotal": "188.12"
9     },
10    {
11      "id_prod": 17,
12      "nombre": "Producto 17",
13      "precio": "35.69",
14      "cantidad": 5,
```

Consulta2



PARCIAL / CONSULTAS_NATIVAS / Get consulta2

GET http://localhost:3000/api/productos/mas-vendidos/1-

Params Authorization Headers (6) Body Scripts Settings

Query Params

Key	Value	Description
Key	Value	Description

200 OK - 1.27 s - 2.52 KB

```
131 {
132   "total_vendido": "0",
133 },
134 {
135   "id_prod": 42,
136   "nombre": "Producto 42",
137   "total_vendido": "6"
138 },
139 {
140   "id_prod": 14,
141   "nombre": "Producto 14",
142   "total_vendido": "6"
143 },
144 {
145   "id_prod": 3,
```

Consulta 3

The screenshot shows the Postman interface with a workspace named 'My Workspace'. The left sidebar shows a collection named 'PARCIAL' with several endpoints. The selected endpoint is 'Get consulta3' with a URL of 'http://localhost:3000/api/ventas/por-restaurante'. The request is a GET method. The response is a 200 OK status with a response time of 1.38s and a body size of 2.42 KB. The response body is a JSON array of sales data grouped by restaurant.

Query Params

Key	Value	Description
Key	Value	Description

Body

```
1 {
2   "ventas": [
3     {
4       "id_rest": 31,
5       "restaurant": "Restaurante 31",
6       "total_ventas": "774.74"
7     },
8     {
9       "id_rest": 13,
10      "restaurant": "Restaurante 13",
11      "total_ventas": "676.18"
12    },
13    {
14      "id_rest": 15,
```

Consulta 4

The screenshot shows the Postman interface with a workspace named 'My Workspace'. The left sidebar shows a collection named 'PARCIAL' with several endpoints. The selected endpoint is 'Get consulta4' with a URL of 'http://localhost:3000/api/pedidos/fecha/2024-03-31'. The request is a GET method. The response is a 200 OK status with a response time of 1.17s and a body size of 330 B. The response body is a JSON object containing a list of orders for the specified date.

Body

```
1 {
2   "pedidos": [
3     {
4       "id_pedido": 36,
5       "fecha": "2024-03-31T05:00:00.000Z",
6       "id_rest": 39,
7       "total": "257.84"
8     }
9   ]
10 }
```


Consulta 5

Postman interface showing a workspace named "My Workspace" with a collection named "CONSULTAS_NATIVAS". The selected item is "Get consulta5". The URL is "http://localhost:3000/api/empleados/por-rol/2/Mesero". The method is "GET".

Params: Authorization, Headers, Body, Scripts, Settings

Query Params:

Key	Value	Description
Key	Value	Description

Response: History

Click Send to get a response

System tray: 16°C, Lluvia suave, Buscar, Postbot, Runner, Start Proxy, Cookies, Vault, Trash, 11:41 a.m., 25/04/2025