

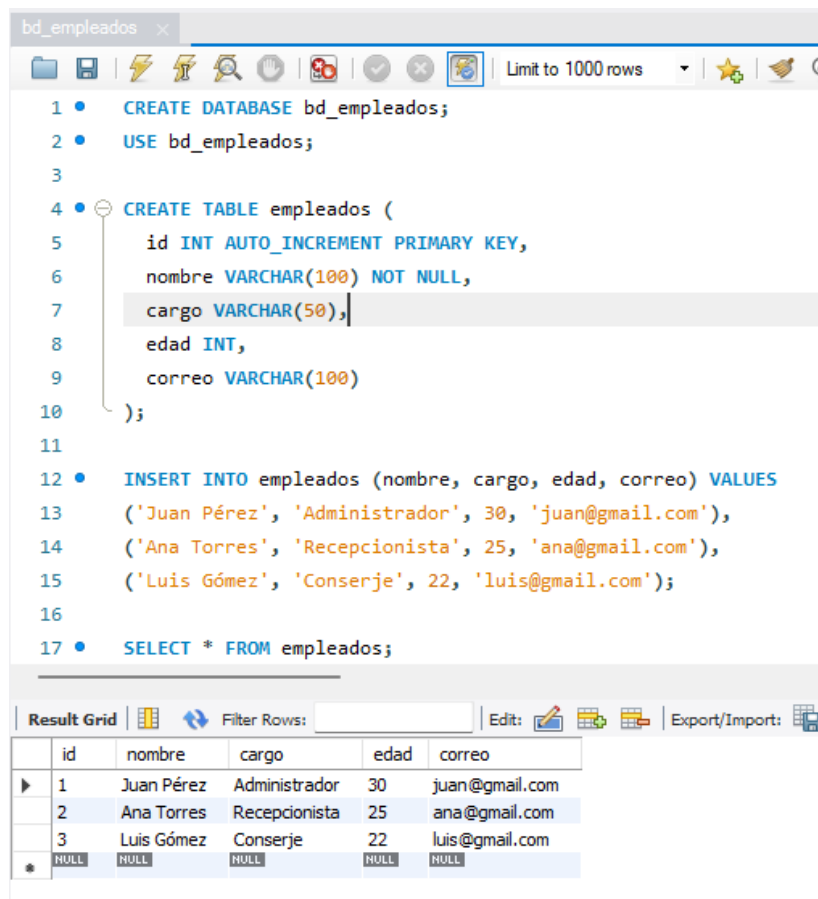
# TAREA 08

Implementación del CRUD de Empleados en Node.js (Backend) y mysql

PROCEDIMIENTOS:

La base de datos se llama **bd\_empleados** y contiene una tabla principal llamada **empleados**, que almacena la información básica de cada trabajador.

Esta base de datos se conecta al backend desarrollado



The screenshot shows a database management interface with a tab labeled 'bd\_empleados'. The SQL editor contains the following queries:

```
1 • CREATE DATABASE bd_empleados;
2 • USE bd_empleados;
3
4 • CREATE TABLE empleados (
5     id INT AUTO_INCREMENT PRIMARY KEY,
6     nombre VARCHAR(100) NOT NULL,
7     cargo VARCHAR(50),
8     edad INT,
9     correo VARCHAR(100)
10 );
11
12 • INSERT INTO empleados (nombre, cargo, edad, correo) VALUES
13     ('Juan Pérez', 'Administrador', 30, 'juan@gmail.com'),
14     ('Ana Torres', 'Recepcionista', 25, 'ana@gmail.com'),
15     ('Luis Gómez', 'Conserje', 22, 'luis@gmail.com');
16
17 • SELECT * FROM empleados;
```

Below the editor, the 'Result Grid' shows the data returned by the SELECT query:

	id	nombre	cargo	edad	correo
▶	1	Juan Pérez	Administrador	30	juan@gmail.com
	2	Ana Torres	Recepcionista	25	ana@gmail.com
	3	Luis Gómez	Conserje	22	luis@gmail.com
*	NULL	NULL	NULL	NULL	NULL

## BACKEND:

### Config/db.js:


Esto define la conexión a la base de datos sin esto el controlador no podrá ejecutar consultas.

```
config > JS db.js > ...
1 //Acceder al archivo .env
2 require('dotenv').config()
3
4 const mysql = require('mysql2/promise')
5
6 //Pool de conexiones = acceso
7 const pool = mysql.createPool({
8   host: process.env.DB_HOST,
9   user: process.env.DB_USER,
10  password: process.env.DB_PASSWORD,
11  database: process.env.DB_DATABASE,
12  port: process.env.DB_PORT
13 })
14
15
16 module.exports = pool
```


## Controllers/empleadoController:

Usa la conexión de db.js para hacer las operaciones (crear, leer, actualizar, eliminar).

### (Listar, crear)

```
controllers >  empleadoController.js > ...
1  const db = require('../config/db')
2
3  // Listar
4  const obtenerTodos = async (req, res) => {
5    try {
6      const [rows] = await db.query('SELECT * FROM empleados')
7      res.json(rows)
8    } catch (e) {
9      res.status(500).json({ error: 'Error interno en el servidor' })
10   }
11 }
12
13 // Crear
14 const crear = async (req, res) => {
15   try {
16     const { nombre, cargo, edad, correo } = req.body
17     if (!nombre || !cargo || !edad || !correo)
18       return res.status(400).json({ error: 'Todos los campos son obligatorios' })
19
20     const [result] = await db.query(
21       'INSERT INTO empleados (nombre, cargo, edad, correo) VALUES (?, ?, ?, ?)',
22       [nombre, cargo, edad, correo]
23     )
24     res.status(201).json({ id: result.insertId, message: 'Empleado creado correctamente' })
25   } catch (e) {
26     res.status(500).json({ error: 'Error al crear empleado' })
27   }
28 }
```

### (Actualizar, Eliminar)

```
controllers >  empleadoController.js > ...
30 // Actualizar
31 const actualizar = async (req, res) => {
32   try {
33     const { id } = req.params
34     const { nombre, cargo, edad, correo } = req.body
35     const [result] = await db.query(
36       'UPDATE empleados SET nombre=?, cargo=?, edad=?, correo=? WHERE id=?',
37       [nombre, cargo, edad, correo, id]
38     )
39     if (result.affectedRows === 0) return res.status(404).json({ error: 'Empleado no encontrado' })
40     res.json({ message: 'Empleado actualizado correctamente' })
41   } catch (e) {
42     res.status(500).json({ error: 'Error al actualizar empleado' })
43   }
44 }
45
46 // Eliminar
47 const eliminar = async (req, res) => {
48   try {
49     const { id } = req.params
50     const [result] = await db.query('DELETE FROM empleados WHERE id=?', [id])
51     if (result.affectedRows === 0) return res.status(404).json({ error: 'Empleado no encontrado' })
52     res.json({ message: 'Empleado eliminado correctamente' })
53   } catch (e) {
54     res.status(500).json({ error: 'Error al eliminar empleado' })
55   }
56 }
57
58 module.exports = { obtenerTodos, crear, actualizar, eliminar }
```

## -routes/empleadoRoutes.js

Las rutas de la API llaman a las funciones del controlador.

```
server.js db.js empleadoController.js empleadoRoutes.js X TC
routes > JS empleadoRoutes.js > ...
1  const express = require('express')
2  const router = express.Router()
3  const empleadoController = require('../controllers/empleadoController')
4
5  // CRUD
6  router.get('/', empleadoController.obtenerTodos) // GET - listar
7  router.post('/', empleadoController.crear) // POST - crear
8  router.put('/:id', empleadoController.actualizar) // PUT - actualizar
9  router.delete('/:id', empleadoController.eliminar) // DELETE - eliminar
10
11  module.exports = router
12  |
```

-server.js: Configura el servidor y define las rutas base de la API.

```
JS server.js > ...
1  const express = require('express')
2  const cors = require('cors')
3
4  // Rutas
5  const empleadoRoutes = require('../routes/empleadoRoutes')
6
7  const app = express()
8  const PORT = process.env.PORT || 3000
9
10 // CONFIGURACIÓN
11 // Permisos CORS
12 app.use(cors({
13   origin: '*',
14   methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
15   credentials: true
16 }))
17 // JSON
18 app.use(express.json())
19
20 // Rutas API
21 app.use('/api/empleados', empleadoRoutes)
22
23 // Iniciar servidor
24 app.listen(PORT, () => {
25   console.log(`Servidor iniciado en http://localhost:${PORT}`)
26 })
27
```

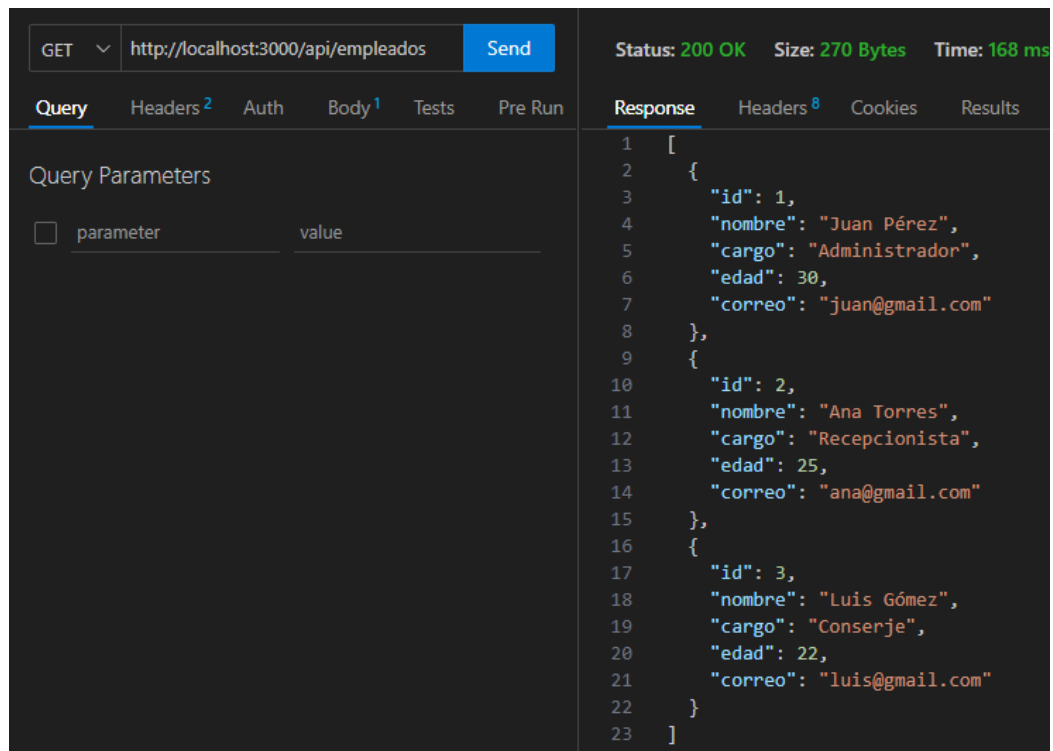
## EJECUCIÓN DEL SERVIDOR:

Una vez iniciado el servidor con el comando:

“nodemon server”

Se realizan las pruebas de los cuatro métodos principales del CRUD (**GET**, **POST**, **PUT** y **DELETE**) utilizando la herramienta **Thunder Client**.

### 1. GET - Listar empleados

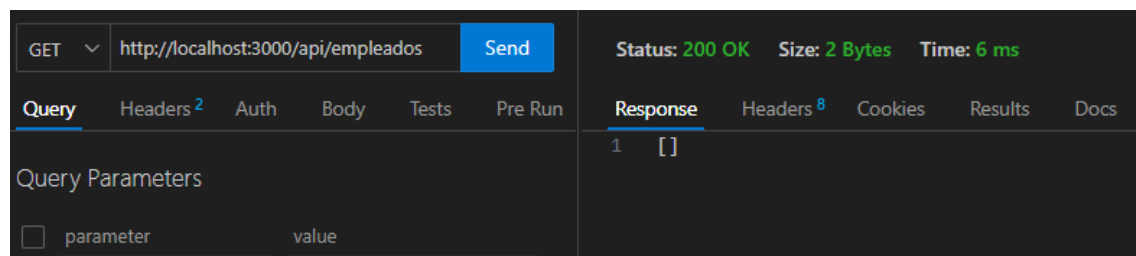


The screenshot shows the Thunder Client interface with a GET request to `http://localhost:3000/api/empleados`. The status is **200 OK**, size is **270 Bytes**, and time is **168 ms**. The response is a JSON array of three employee objects:

```
[
  {
    "id": 1,
    "nombre": "Juan Pérez",
    "cargo": "Administrador",
    "edad": 30,
    "correo": "juan@gmail.com"
  },
  {
    "id": 2,
    "nombre": "Ana Torres",
    "cargo": "Recepcionista",
    "edad": 25,
    "correo": "ana@gmail.com"
  },
  {
    "id": 3,
    "nombre": "Luis Gómez",
    "cargo": "Conserje",
    "edad": 22,
    "correo": "luis@gmail.com"
  }
]
```

### Prueba de error:

Si la base de datos está vacía, simplemente devuelve un arreglo vacío `[]`.



The screenshot shows the Thunder Client interface with a GET request to `http://localhost:3000/api/empleados`. The status is **200 OK**, size is **2 Bytes**, and time is **6 ms**. The response is an empty JSON array:

```
[ ]
```

## 2. POST - Crear empleado

The screenshot shows a REST client interface with a POST request to `http://localhost:3000/api/empleados`. The request body is a JSON object: `{ "nombre": "Diego Tasayco", "cargo": "Programador", "edad": 19, "correo": "tasayco@gmail.com" }`. The response status is `201 Created` with a size of `50 Bytes` and a time of `15 ms`. The response body is a JSON object: `{ "id": 4, "message": "Empleado creado correctamente" }`.

Method	URL	Status	Size	Time
POST	http://localhost:3000/api/empleados	201 Created	50 Bytes	15 ms

Request Body (JSON):

```
{
  "nombre": "Diego Tasayco",
  "cargo": "Programador",
  "edad": 19,
  "correo": "tasayco@gmail.com"
}
```

Response (JSON):

```
{
  "id": 4,
  "message": "Empleado creado correctamente"
}
```

### Prueba de error:

Si falta algún campo, el sistema devuelve:

The screenshot shows a REST client interface with a POST request to `http://localhost:3000/api/empleados`. The request body is a JSON object: `{ "nombre": "Carlos Rodriguez", "cargo": "Administrador", "edad": 21, "": "carlos@gmail.com" }`. The response status is `400 Bad Request` with a size of `45 Bytes` and a time of `6 ms`. The response body is a JSON object: `{ "error": "Todos los campos son obligatorios" }`.

Method	URL	Status	Size	Time
POST	http://localhost:3000/api/empleados	400 Bad Request	45 Bytes	6 ms

Request Body (JSON):

```
{
  "nombre": "Carlos Rodriguez",
  "cargo": "Administrador",
  "edad": 21,
  "": "carlos@gmail.com"
}
```

Response (JSON):

```
{
  "error": "Todos los campos son obligatorios"
}
```

### 3. PUT - Actualizar empleado

The screenshot shows a REST client interface with the following details:

- Method:** PUT
- URL:** http://localhost:3000/api/empleados/4
- Status:** 200 OK
- Size:** 48 Bytes
- Time:** 12 ms
- Body:**

```
{
  "nombre": "Alonso Garay",
  "cargo": "Administrador",
  "edad": 40,
  "correo": "garay@gmail.com"
}
```
- Response:**

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

```
10     "id": 2,
11     "nombre": "Ana Torres",
12     "cargo": "Recepcionista",
13     "edad": 25,
14     "correo": "ana@gmail.com"
15   },
16   {
17     "id": 3,
18     "nombre": "Luis Gómez",
19     "cargo": "Conserje",
20     "edad": 22,
21     "correo": "luis@gmail.com"
22   },
23   {
24     "id": 4,
25     "nombre": "Alonso Garay",
26     "cargo": "Administrador",
27     "edad": 40,
28     "correo": "garay@gmail.com"
29   }
}
```

#### Prueba de error:

Si el ID no existe:

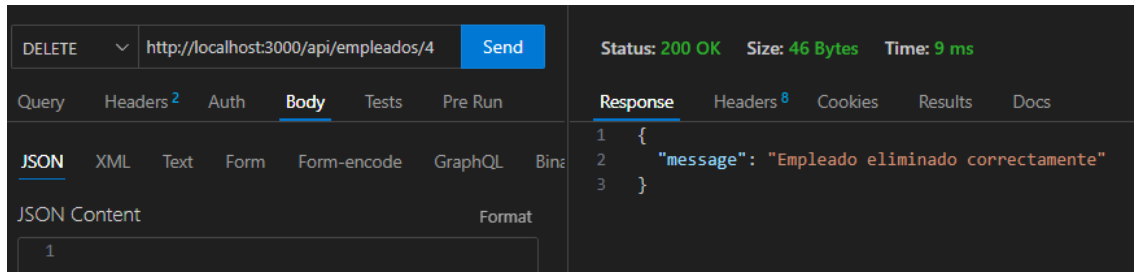
The screenshot shows a REST client interface with the following details:

- Method:** PUT
- URL:** http://localhost:3000/api/empleados/8
- Status:** 404 Not Found
- Size:** 34 Bytes
- Time:** 6 ms
- Body:**

```
{
  "nombre": "Alonso Garay",
  "cargo": "Administrador",
  "edad": 40,
  "correo": "garay@gmail.com"
}
```
- Response:**

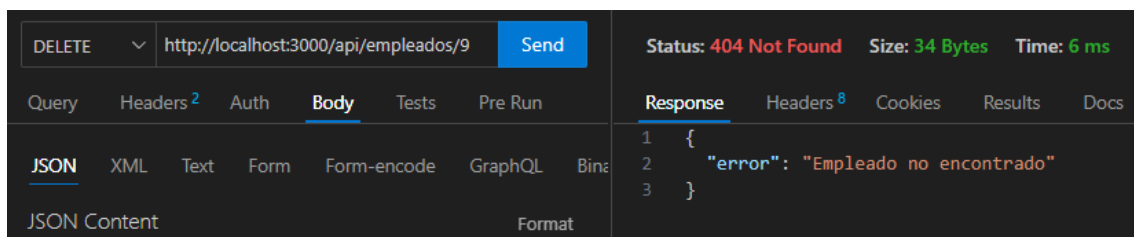
```
{
  "error": "Empleado no encontrado"
}
```

#### 4. DELETE - Eliminar empleado



#### Prueba de error:

Si se intenta eliminar un ID inexistente:



#### Carpetas creadas:

