Guía complementaria - Todo App (Parte II)

 ¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo recordar y repasar los contenidos que hemos visto en clase.

Dentro de los que se encuentran:

- Adaptar nuestro proyecto anterior "todo app" para utilizar una base de datos PostgreSQL.
- Leer e Insertar datos en una tabla utilizando el módulo pg de Node.js.
- Organizar el código progresivamente utilizando el patrón MVC.
- Crear variables de entorno para no compartir datos sensibles como la contraseña de la base de datos.

¡Importante! Esta guía es un apoyo para complementar ejercicios y prácticas, para un nivel más estructurado y avanzado. No entramos en detalles de conceptos que están abordados en la guía “Acceso a Base de Datos con Node (Parte I)”, por ende, si no entiendes algún concepto te recomendamos revisar dicha guía.

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

Contexto antes de iniciar

Seguiremos utilizando el proyecto "todo app" que creamos en la **Guía complementaria - Todo App (Parte I)** de la unidad 2 Introducción a Express, pero ahora lo adaptaremos para utilizar una base de datos PostgreSQL.

Asumimos que realizaste el ejercicio de la sección anterior, pero de todas formas te compartiremos la aplicación Backend y Frontend en la plataforma, con el nombre ***"Material de apoyo - Todo app (Parte I)"***

Al descargar el material de apoyo, solo ejecuta el comando npm install en ambas aplicaciones para que se instalen las dependencias.

¡Manos a la obra! - Agregando pg al Backend

A continuación, seguiremos trabajando en la aplicación Todo app, en esta ocasión realizaremos la instalación en el Backend del módulo pg, el comando es `npm i pg`. Una vez instalado, procedemos a realizar los siguientes pasos.

- **Paso 1:** En una terminal nueva accedemos a PostgreSQL y creamos una base de datos.

```
JavaScript
CREATE DATABASE db_app_todo;
```

```
JavaScript
DROP TABLE IF EXISTS todos;

CREATE TABLE todos (
  id SERIAL PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  done BOOLEAN NOT NULL DEFAULT false
);

-- insertar datos de ejemplo
INSERT INTO todos (title, done) VALUES ('Tarea 1', false);
INSERT INTO todos (title, done) VALUES ('Tarea 2', true);
INSERT INTO todos (title, done) VALUES ('Tarea 3', false);
```

- **Paso 2:** Conexión a la base de datos, para lograrlo debemos instalar previamente el módulo pg con el comando `npm install pg`. Luego, en el backend, creamos un nuevo archivo para la conexión database/connection.js e incorporamos el siguiente código:

```javascript
import pkg from "pg";
const { Pool } = pkg;

// cambia los datos de acuerdo a tu configuración de postgres
export const pool = new Pool({
  user: "postgres",
  host: "localhost",
  database: "db_app_todo",
  password: "root",
  port: 5432,
  allowExitOnIdle: true,
});

try {
  await pool.query("SELECT NOW()");
  console.log("Database connected");
} catch (error) {
  console.log(error);
}
```

Nota: Recuerda que en los datos de conexión debes definir lo que estén definidos en tu computador.

- El nombre pkg es un alias que se le asigna al paquete importado.
- La clase Pool se utiliza para administrar una agrupación de conexiones a la base de datos PostgreSQL.
- **allowExitOnIdle:** Cuando allowExitOnIdle se establece en true, indica que el programa puede finalizar y salir de forma segura si todas las conexiones en la agrupación (Pool) están inactivas, es decir, no se están utilizando.
- La carpeta database contiene el archivo connection.js que se encarga de realizar la conexión a la base de datos. Esto es una buena práctica para mantener el código organizado, pero existen muchas otras formas.

  Podemos comprobar la conexión corriendo el comando:

```
JavaScript
node database/connection.js
```

Si todo resulta bien, veremos el siguiente mensaje en consola o terminal.

```
JavaScript
PS D:\Node JS\U2\Backend> node .\database\connection.js
Database connected
```

Tip: No olvides exportar la constante pool para poder utilizarla en otros archivos.

Leer datos - findAll

- **Paso 3:** Ahora, realizaremos la lectura de datos con findAll. Para lograrlo, agruparemos el código en un archivo todo.model.js dentro de una carpeta /models. Esto es para mantener el código organizado utilizando el patrón MVC, del cual veremos más detalles en unidades posteriores.

  El código del archivo todo.model.js será el siguiente:

```
JavaScript
import { pool } from "../database/connection.js";

const findAll = async () => {
  const { rows } = await pool.query("SELECT * FROM todos");
  return rows;
};

export const todoModel = {
  findAll,
};
```

- **Paso 4:** Ahora en nuestro index.js del backend podemos importar el modelo y utilizarlo.

```javascript
import { todoModel } from "./models/todo.model.js";

// GET /todos
app.get("/todos", async (req, res) => {
  try {
    const todos = await todoModel.findAll();
    return res.json(todos);
  } catch (error) {
    console.log(error);
    return res.status(500).json({ message: "Internal server error" });
  }
});
```

Si probamos la ruta en el navegador, veremos como resultado el listado de tareas que contiene la base de datos.

Ruta: GET http://localhost:5000/todos

```javascript
[
  {
    id: 1,
    title: "Tarea 1",
    done: false,
  },
  {
    id: 2,
    title: "Tarea 2",
    done: true,
  },
  {
    id: 3,
    title: "Tarea 3",
    done: false,
  },
];
```

Leer datos - findById

- **Paso 5:** Ahora, insertaremos el código para probar la ruta de consulta a una tarea en específico por el parámetro del :id. En el archivo todo.model.js agregamos el siguiente código:

```
JavaScript
const findById = async (id) => {
  const query = "SELECT * FROM todos WHERE id = $1";
  const { rows } = await pool.query(query, [id]);
  return rows[0];
};

export const todoModel = {
  findAll,
  findById,
};
```

En el index.js del backend, escribimos el siguiente código para realizar la consulta a la base de datos.

```
JavaScript
// GET /todos/:id
app.get("/todos/:id", async (req, res) => {
  const id = req.params.id;

  try {
    const todo = await todoModel.findById(id);

    if (!todo) {
      res.status(404).json({ message: "Todo not found" });
    }
    res.json(todo);
  } catch (error) {
    console.log(error);
    return res.status(500).json({ message: "Internal server error" });
  }
});
```

Al ejecutar la consulta en la ruta: http://localhost:5000/todos/1 veremos como resultado la tarea 1.

En unidades posteriores, veremos cómo validar los datos de entrada para evitar errores en la base de datos.

Insertar datos - create

- **Paso 6:** Veamos el código en el archivo todo.model.js para insertar nuevos registros a la base de datos.

```
JavaScript
const create = async (todo) => {
    const query = "INSERT INTO todos (title, done) VALUES ($1, $2) RETURNING *";
  const { rows } = await pool.query(query, [todo.title, todo.done]);
  return rows[0];
};

export const todoModel = {
  findAll,
  findById,
  create,
};
```

No es necesario recibir el todo.done en el body ni en el modelo, ya que por defecto se insertará como false en la base de datos. Pero lo dejaremos por ahora para que puedas observar las rutas parametrizadas.

Ahora, modificamos el archivo index.js para realizar la petición a la base de datos con el verbo post.

```
JavaScript
// POST /todos
app.post("/todos", async (req, res) => {
 const { title } = req.body;

 if (!title) {
  return res.status(400).json({ message: "Title is required" });
 }
```

```
const newTodo = {
 title,
 done: false,
};
try {
 const todo = await todoModel.create(newTodo);
 return res.json(todo);
} catch (error) {
 console.log(error);
 return res.status(500).json({ message: "Internal server error" });
}
});
```

Recuerda que puedes probar la inserción de datos utilizando Postman o Thunder Client. Una vez seleccionada la herramienta, recuerda insertar mediante body el nuevo registro.

```
JavaScript
{
 "title": "Tarea 4"
}
```

Si compruebas esta ejecución en PostgreSQL, verás que el nuevo dato ha sido ingresado correctamente:

```
JavaScript
db_app_todo=# select * from todos;
 id | title | done
----+---------+------
  1 | Tarea 1 | f
  2 | Tarea 2 | t
  3 | Tarea 3 | f
  4 | Tarea 4 | f
(4 rows)
```

Remover datos - remove

- **Paso 7:** Ahora, debemos definir el código para la acción de eliminación de registros de la base de datos. Para lograrlo, debemos incorporar el siguiente código en el archivo todo.model.js

```javascript
const remove = async (id) => {
  const query = "DELETE FROM todos WHERE id = $1 RETURNING *";
  const { rows } = await pool.query(query, [id]);
  return rows[0];
};

export const todoModel = {
  findAll,
  findById,
  create,
  remove,
};
```

En el index.js del Backend agregamos el siguiente código para la acción del Delete.

```javascript
// DELETE /todos/:id
app.delete("/todos/:id", async (req, res) => {
  const id = req.params.id;
  try {
    const todo = await todoModel.remove(id);
    if (!todo) {
      return res.status(404).json({ message: "Todo not found" });
    }
    return res.json({ message: "Todo deleted" });
  } catch (error) {
    console.log(error);
    return res.status(500).json({ message: "Internal server error" });
  }
});
```

La acción del delete la puedes realizar mediante Postman o Thunder Client. Selecciona la acción Delete en la herramienta de consulta e ingresa la siguiente ruta http://localhost:5000/todos/4, esto eliminará la tarea con ID 4.

Actualizar datos - update

- **Paso 8:** Agregamos el código para la acción del Update, primero en el archivo todo.model.js

```javascript
const update = async (id) => {
  const query = "UPDATE todos SET done = NOT done WHERE id = $1 RETURNING
*";
  const { rows } = await pool.query(query, [id]);
  return rows[0];
};

export const todoModel = {
  findAll,
  findById,
  create,
  remove,
  update,
};
```

Ahora, el código del archivo index.js del Backend.

```javascript
// PUT /todos/:id
app.put("/todos/:id", async (req, res) => {
  const id = req.params.id;

  try {
    const todo = await todoModel.update(id);
    if (!todo) {
      return res.status(404).json({ message: "Todo not found" });
    }
    return res.json(todo);
  } catch (error) {
    console.log(error);
    return res.status(500).json({ message: "Internal server error" });
  }
});
```

SET done = NOT done

Modifica el valor de la columna done. La expresión NOT done invierte el valor actual de la columna done. Si done es true, después de la actualización será false, y viceversa.

Si probamos la acción PUT en la herramienta de consultas, veremos el siguiente resultado. El estado de la tarea cambiará a true.

Verbo PUT - Ruta http://localhost:5000/todos/1

```javascript
{
 "id": 1,
 "title": "Tarea 1",
 "done": true
}
```

Variables de entorno

Cuando trabajamos con bases de datos, es importante mantener un uso correcto de la información y velar por la seguridad de datos que pudieran ser sensibles. Las variables de entorno, nos permitirán resguardar la seguridad de esa información. Para lograrlo debemos incorporar en el backend una dependencia llamada dotenv. Ejecuta el comando `npm i dotenv`.

- **Paso 9:** Agregamos variables de entorno en un archivo .env. Este, lo ubicaremos en la raíz del proyecto.

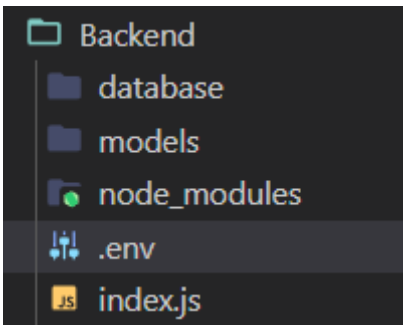

Imagen 1. Ubicación del archivo .env
Fuente: Desafío Latam

- **Paso 10:** Agregamos las variables de entorno en el archivo .env.

```
Unset
PGUSER="postgres"
PGHOST="localhost"
PGPASSWORD="root"
PGDATABASE="db_app_todo"
PGPORT=5432
```

- **Paso 11:** Seguidamente, modificamos el archivo connection.js y sustituimos los datos de conexión por las variables de entorno que definimos en el archivo .env. Para ello debemos importar primero la dependencia de dotenv.

```
JavaScript
import "dotenv/config";
```

Ahora el código de conexión a la base de datos.

```
JavaScript
export const pool = new Pool({
  user: process.env.PGUSER,
  host: process.env.PGHOST,
  database: process.env.PGDATABASE,
  password: process.env.PGPASSWORD,
  port: process.env.PGPORT,
  allowExitOnIdle: true,
});
```

Cuando hagas la conexión a github de este proyecto, recuerda agregar en el archivo .gitignore el archivo .env para que no se suba al repositorio. Así, este archivo no se compartirá en github y estas variables solo estarán disponibles en nuestro entorno local.

Dato interesante

En la documentación oficial de node-postgres puedes omitir llamar a las variables de entorno en tu archivo connection.js siempre y cuando los nombres de estas variables sean:

```
JavaScript
PGUSER="postgres"
PGHOST="localhost"
PGPASSWORD="root"
PGDATABASE="db_app_todo"
PGPORT=5432
```

En el archivo connection.js tendríamos entonces el siguiente código.

```
JavaScript
// Lo que está comentado podrías eliminarlo.
export const pool = new Pool({
  // user: process.env.PGUSER,
  // host: process.env.PGHOST,
  // database: process.env.PGDATABASE,
  // password: process.env.PGPASSWORD,
  // port: process.env.PGPORT,
  allowExitOnIdle: true,
});
```

Puerto mediante variable de entorno

Finalmente en el archivo index.js cambiamos el puerto que establecimos de forma estática por una variable de entorno. Ya que en producción el puerto será asignado por el servidor.

La variable PORT no es necesario agregarla en el archivo .env, puesto que la mayoría de los servidores la asignan automáticamente. Esto lo veremos con más detalles en la sección de Deploy en unidades posteriores.

```
JavaScript
import "dotenv/config";

const PORT = process.env.PORT || 5000;

app.listen(PORT, () => {
```

```
  console.log(`Server listening on port http://localhost:${PORT}`);
});
```

Probar la aplicación Frontend

Ahora que ya tenemos nuestra aplicación backend funcionando, podemos probarla con el frontend que creamos en la sección anterior.

Intenta pasar http://localhost:5000 a una variable de entorno en Vite, pero revisa la documentación oficial (aquí la documentación), ya que no es igual a la forma que lo hicimos en el backend.

Imagen 2. Probando el Front
Fuente: Desafío Latam

Al ingresar algunos datos, comprobamos la inserción en la base de datos:

```
JavaScript

db_app_todo=# select * from todos;
 id | title | done
----+---------+------
  3 | Tarea 3 | f
  6 | Tarea 5 | f
  7 | Tarea 6 | f
  2 | Tarea 2 | f
  1 | Tarea 1 | f
```

```
 5 | Tarea 4 | f
(6 rows)
```

¡Felicitaciones! Si llegaste a este punto de la guía, es porque lograste conectar una base de datos PostgreSQL con React.

Intenta replicar los pasos anteriores, recuerda que la práctica es fundamental durante estos aprendizajes.