Guía complementaria - Todo App (Parte IV)

¡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. Además, veremos los siguientes temas:

- Implementar limit y order by en la API.
- Implementar paginación en la API.
- Implementar HATEOAS en la API.
- Implementar paginación en el frontend.
- Implementar order by en el frontend.

¡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 **"Guía - Diseño avanzado de una API REST"**, 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 III)** de la unidad Acceso a una base de datos con Node y el paquete pg (Parte II).

Limit

El LIMIT nos permite limitar la cantidad de filas que devuelve una consulta SQL, por ejemplo:

```
JavaScript
SELECT * FROM todos LIMIT 5;
```

¡Manos a la obra! - Agregando limit, paginación y HATEOAS (Backend)

Antes de iniciar con el ejercicio debemos recordar el concepto de Limit. Este nos permite limitar la cantidad de filas que devuelve una consulta SQL.

```
JavaScript
SELECT * FROM todos LIMIT 5;
```

- **Paso 1:** Apliquemos en el modelo de datos un límite que sea configurable

models/todo.model.js

```
JavaScript
const findAll = async ({ limit = 5 }) => {
  // { limit = 5 } destructuring y asignamos 5 por defecto
  const query = "SELECT * FROM todos LIMIT $1";

  const { rows } = await pool.query(query, [limit]);
  return rows;
```

```
};
```

- **Paso 2:** Finalmente, en nuestro controlador podemos hacer que el límite sea configurable desde la query string.

controllers\todo.controller.js

```javascript
const read = async (req, res) => {
  const { limit = 5 } = req.query;

  try {
    const todos = await todoModel.findAll({ limit });
    return res.json(todos);
  } catch (error) {
    console.log(error);
    if (error.code) {
      const { code, message } = getDatabaseError(error.code);
      return res.status(code).json({ message });
    }
    return res.status(500).json({ message: "Internal server error" });
  }
};
```

Si ahora hacemos una petición a nuestra API, podemos ver que el límite es configurable.

```javascript
GET: http://localhost:5000/todos?limit=3
```

Como resultado veremos solamente 3 registros.

Order by

Para ordenar los resultados por la columna done, puedes usar la cláusula ORDER BY en tu consulta SQL. Aquí tienes un ejemplo de cómo podrías hacerlo:

```
JavaScript
SELECT * FROM todos
ORDER BY done ASC;
```

Esta consulta seleccionará todas las filas de la tabla todos y las ordenará en función de la columna done. Las filas con done establecido en false aparecerán primero, seguidas de las filas con done establecido en true.

Si deseas ordenar en orden descendente, puedes agregar la palabra clave DESC después de done:

```
JavaScript
SELECT * FROM todos
ORDER BY done DESC;
```

Esto colocará las filas con done establecido en true primero, seguidas de las filas con done establecido en false.

La URL debería verse así:

```
JavaScript
GET: http://localhost:5000/todos?limit=10&order=desc
```

Pero en PostgreSQL, no podemos parametrizar el nombre de la columna o la dirección de orden (ASC o DESC) directamente en una consulta utilizando marcadores de posición (1, 2, etc.) Los marcadores de posición se utilizan para valores, no para nombres de columnas o palabras clave.

pg-format

El paquete pg-format nos permite formatear consultas SQL de forma segura. Esta es una solución más robusta que la parametrización de cadenas, ya que también nos permite escapar de forma segura los identificadores y los valores literales SQL.

- **Paso 3:** Instalamos pg-format

```
JavaScript
npm install pg-format
```

En una consulta PostgreSQL, los símbolos % seguidos de una letra indican cómo se debe formatear un valor en la consulta.

- **%%:** se utiliza para imprimir un carácter % literal en la consulta.
- **%I:** se utiliza para imprimir un identificador SQL escapado en la consulta. Esto es útil para evitar problemas de seguridad al utilizar identificadores que contienen caracteres especiales o palabras reservadas de SQL.
- **%L:** se utiliza para imprimir un valor literal SQL escapado en la consulta. Esto es útil para evitar problemas de seguridad al utilizar valores que contienen caracteres especiales o comillas.
- **%s:** se utiliza para imprimir una cadena simple en la consulta. Este es el formato predeterminado para imprimir valores en la consulta.

```
JavaScript
SELECT * FROM %I WHERE name = %L;
```

Supongamos la siguiente consulta:

```
JavaScript
SELECT * FROM "users" WHERE name = 'John Doe';
```

En esta consulta, **%I** se utiliza para escapar el nombre de la tabla y **%L** se utiliza para escapar el valor literal de la cadena “John Doe”.

- **Paso 4:** Implementemos el escape de caracteres en nuestro todo.model.js.

models/todo.model.js

```javascript
import format from "pg-format";
import { pool } from "../database/connection.js";

const findAll = async ({ limit = 5, order = "ASC" }) => {
  const query = "SELECT * FROM todos ORDER BY done %s LIMIT %s";

  const formattedQuery = format(query, order, limit);
  const { rows } = await pool.query(formattedQuery);
  return rows;
};
```

controllers/todo.controller.js

```javascript
const read = async (req, res) => {
  const { limit = 5, order = "ASC" } = req.query;

  try {
    const todos = await todoModel.findAll({ limit, order });
    return res.json(todos);
  } catch (error) {
    console.log(error);
    if (error.code) {
      const { code, message } = getDatabaseError(error.code);
      return res.status(code).json({ message });
    }
    return res.status(500).json({ message: "Internal server error" });
  }
};
```

Ahora, prueba los resultados consultando la siguiente ruta:

```javascript
GET: http://localhost:5000/todos?limit=10&order=desc
```

Paginación

El OFFSET nos permite omitir una cantidad de filas en una consulta SQL. Paginar se logra con una combinación de offset y limit. El limit es el tamaño de la página y con el offset nos movemos a la página en específico, por ejemplo:

- Página 0 => Offset 0
- Página 1 => Offset 10
- Página 2 => Offset 20
- Página 3 => Offset 30

Ejemplo de consulta SQL

```JavaScript
SELECT * FROM todos
LIMIT 5
OFFSET 5;
```

La ruta de funcionamiento es la siguiente:

```JavaScript
GET: http://localhost:3000/todos?limit=5&page=2
```

- **Paso 5**: Agregamos la configuración de paginación en el modelo.

models\todo.model.js

```JavaScript
const findAll = async ({ limit = 5, order = "ASC", page = 1 }) => {
   const query = "SELECT * FROM todos ORDER BY done %s LIMIT %s OFFSET
%s";

  const offset = (page - 1) * limit;
  const formattedQuery = format(query, order, limit, offset);
  const { rows } = await pool.query(formattedQuery);
  return rows;
};
```

La fórmula `offset = (page - 1) * limit` se utiliza para calcular cuántas filas se deben omitir en la consulta. Aquí está la lógica detrás de esto:

- **page**: representa el número de la página que estás solicitando.
- **limit**: es la cantidad de resultados que deseas obtener por página.

Entonces, si estás en la primera página (page = 1), el offset será cero, lo que significa que no se omitirá ninguna fila. Si estás en la segunda página (page = 2), el offset será igual a limit, por lo que se saltarán las filas correspondientes a la primera página. Este patrón se repite para las páginas siguientes.

controllers\todo.controller.js

```JavaScript
const read = async (req, res) => {
  const { limit = 5, order = "ASC", page = 1 } = req.query;

  // Utilizar una expresión regular para verificar si 'page' es un número
válido
  const isPageValid = /^[1-9]\d*$/.test(page);

  // Validar el resultado de la expresión regular
  if (!isPageValid) {
    return res.status(400).json({ message: "Invalid page number, number >
0" });
  }

  try {
    const todos = await todoModel.findAll({ limit, order, page });
    return res.json(todos);
  } catch (error) {
    console.log(error);
    if (error.code) {
      const { code, message } = getDatabaseError(error.code);
      return res.status(code).json({ message });
    }
    return res.status(500).json({ message: "Internal server error" });
  }
};
```

Ruta de consulta para la paginación

```JavaScript
GET: http://localhost:5000/todos?limit=5&order=desc&page=2
```

En este caso, estamos utilizando una expresión regular para validar que el número de página sea mayor a cero. Recuerda que esto es solo una de las muchas formas de validar datos en JavaScript.

Aquí dejamos otra forma de hacerlo:

```JavaScript
const isPageValid = Number.isInteger(Number(page)) && Number(page) > 0;

if (!isPageValid) {
  return res.status(400).json({ message: "Invalid page number, page > 0"
});
}
```

La elección entre estas opciones depende de tus preferencias y de la complejidad de la validación que necesitas. La validación con Number.isInteger es clara y puede ser suficiente en muchos casos. Sin embargo, si necesitas reglas de validación más específicas o si prefieres un enfoque más flexible, podrías optar por otras opciones, como el uso de expresiones regulares o parseInt.

HATEOAS

Trabajar con HATEOSAS implica que la API REST proporciona enlaces (links) a otros recursos relacionados, permitiendo a los clientes de la API navegar y descubrir de manera dinámica las funcionalidades disponibles.

El beneficio principal de HATEOAS es que permite que los clientes de la API sean menos acoplados y más dinámicos. En lugar de que los clientes tengan que tener un conocimiento previo sobre la estructura de la API y las rutas específicas, pueden navegar y descubrir las rutas y acciones disponibles a medida que interactúan con la API.

Aquí dejamos una implementación de HATEOAS en nuestra API:

```JavaScript
import "dotenv/config";
import format from "pg-format";
import { pool } from "../database/connection.js";

// Creamos una constante con la URL de la aplicación según el entorno
// DOMAIN_URL_APP deberías crearla al momento de desplegar la aplicación
// PORT se debe agregar al archivo .env
```

```
const BASE_URL =
  process.env.NODE_ENV === "production"
    ? process.env.DOMAIN_URL_APP
    : `http://localhost::${process.env.PORT}`;

const findAll = async ({ limit = 5, order = "ASC", page = 1 }) => {
  // Consulta para contar el número total de filas en la tabla 'todos'
  const countQuery = "SELECT COUNT(*) FROM todos";
  const { rows: countResult } = await pool.query(countQuery);
  const total_rows = parseInt(countResult[0].count, 10);

  // Calcula el número total de páginas
  const total_pages = Math.ceil(total_rows / limit);

   const query = "SELECT * FROM todos ORDER BY done %s LIMIT %s OFFSET
%s";
  const offset = (page - 1) * limit;
  const formattedQuery = format(query, order, limit, offset);
  const { rows } = await pool.query(formattedQuery);

  // Devuelve un array con los resultados y un enlace a cada uno de ellos
  const results = rows.map((row) => {
    return {
      ...row,
      href: `${BASE_URL}/todos/${row.id}`,
    };
  });

  // Devuelve un objeto con los resultados, el número total de páginas y
los enlaces a la página siguiente y anterior
  return {
    results,
    total_pages,
    page,
    limit,
    next:
      total_pages <= page
        ? null
        : `${BASE_URL}/todos?limit=${limit}&page=${page + 1}`,
    previous:
       page <= 1 ? null : `${BASE_URL}/todos?limit=${limit}&page=${page -
1}`,
  };
};
```

Con los pasos ejecutados hasta este punto, hemos logrado implementar funcionalidad de limitación de retorno en los datos, ordenamos los registros e implementamos el sistema de paginación.

¡Importante! En la plataforma, te compartimos el código del Backend desarrollado hasta este momento con el nombre ***"Código finalizado - Todo app Backend (Parte IV)"***. Recuerda correr el comando `npm install` para la instalación de las dependencias.

¡Manos a la obra! - Agregando paginación (Frontend)

A continuación, vamos a incorporar algunas modificaciones en el Frontend, en este caso haremos funcionar la paginación que configuramos en el backend.

Recuerda que el Frontend lo venimos trabajando desde la unidad 2. Para este ejercicio, utilizaremos la misma configuración que dejamos hasta ese momento.

- **Paso 1:** Primero crearemos un componente llamado TodoFooter.jsx con el siguiente código:

```javascript
const TodoFooter = () => {
 return <div>TodoFooter</div>;
};

export default TodoFooter;
```

- **Paso 2:** En el App.jsx vamos a incorporar nuevas variables de estado para controlar las siguientes acciones:
  - Un estado inicial para la numeración de la página y su modificación.
  - Estado inicial para obtener el total de páginas que contenga.
  - Estado para controlar la acción de ir a la siguiente página.
  - Un estado para controlar la acción de ir a la página anterior.

- Un estado para controlar el orden en el que se muestran los registros de tareas.

```javascript
const [page, setPage] = useState(1);
const [totalPages, setTotalPages] = useState(1);
const [next, setNext] = useState(null);
const [previous, setPrevious] = useState(null);
const [order, setOrder] = useState("asc");
```

- **Paso 3:** Ahora, modificamos la función getTodos en la cual definiremos el código para la paginación.

```javascript
const getTodos = async (page = 1, order = "asc", limit = 5) => {
    const response = await fetch(
      `${BASE_URL}/todos?page=${page}&limit=${limit}&order=${order}`
    );

        const { results, total_pages, next, previous } = await
response.json();

    setTodos(results);
    setTotalPages(total_pages);
    setNext(next);
    setPrevious(previous);
  };
```

- **Paso 4:** Agregamos al useEffect la escucha a los cambios de paginación y ordenamiento de todos.

```javascript
useEffect(() => {
    getTodos(page, order);
  }, [page, order]);
```

- **Paso 5:** Ahora, modificamos las funciones de agregado, eliminación y actualización de todos que definimos en el componente App.jsx.

```JavaScript
const addTodo = async (title) => {
  const response = await fetch("http://localhost:5000/todos", {
   method: "POST",
   headers: { "Content-Type": "application/json" },
   body: JSON.stringify({ title }),
  });
  console.log({
   response,
  });
  await response.json();
  await getTodos();
 };


 const removeTodo = async (id) => {
  const response = await fetch(`http://localhost:5000/todos/${id}`, {
   method: "DELETE",
  });
  if (response.status !== 200) {
   return alert("Something went wrong");
  }
  await getTodos();
 };

 const updateTodo = async (id) => {
  const response = await fetch(`http://localhost:5000/todos/${id}`, {
   method: "PUT",
  });
  if (response.status !== 200) {
   return alert("Something went wrong");
  }
  await getTodos();
 };
```

- **Paso 6:** Seguidamente, modificamos el return del componente App.jsx

```JavaScript
return (
  <div className="container">
   <h1 className="my-5">Todos APP</h1>
   <TodoForm addTodo={addTodo} />
   <Todos
    todos={todos}
    removeTodo={removeTodo}
```

```
    updateTodo={updateTodo}
   />
   <TodoFooter
    page={page}
    setPage={setPage}
    totalPages={totalPages}
    next={next}
    previous={previous}
    order={order}
    setOrder={setOrder}
   />
  </div>
 );
```

Nótese que estamos llamando al componente TodoFooter.jsx y le estamos pasando mediante props un conjunto de funciones.

- **Paso 7:** En el componente TodoFooter.jsx completamos el código de la siguiente forma.

JavaScript

```
const TodoFooter = ({
  page,
  setPage,
  totalPages,
  next,
  previous,
  order,
  setOrder,
}) => {
  return (
    <div className="mt-2 d-flex">
      <nav className="me-auto">
        <ul className="pagination">
          <>
            <li
              className={`page-item ${!previous ? "disabled" : ""}`}
              onClick={() => {
                if (previous) {
                  setPage(page - 1);
                }
              }}
```

```
            style={
                    !previous ? { cursor: "not-allowed" } : { cursor:
"pointer" }
            }
          >
            <a
              className="page-link"
              href="#"
            >
              Previous
            </a>
          </li>

          {[...Array(totalPages)].map((_, index) => (
            <li
              key={index}
                 className={`page-item ${page === index + 1 ? "active" :
""}`}
              onClick={() => setPage(index + 1)}
            >
              <a
                className="page-link"
                href="#"
              >
                {index + 1}
              </a>
            </li>
          ))}

          <li
            className={`page-item ${!next ? "disabled" : ""}`}
            onClick={() => {
              if (next) {
                setPage(page + 1);
              }
            }}
                 style={!next ? { cursor: "not-allowed" } : { cursor:
"pointer" }}
          >
            <a
              className="page-link"
              href="#"
            >
```

```
              Next
            </a>
          </li>
        </>
      </ul>
    </nav>
    <div>
      <span className="me-2">Order by:</span>
      <button
        className={`btn btn-outline-primary me-2 ${
          order === "asc" ? "active" : ""
        }`}
        onClick={() => setOrder("asc")}
      >
        Active
      </button>
      <button
        className={`btn btn-outline-primary ${
          order === "desc" ? "active" : ""
        }`}
        onClick={() => setOrder("desc")}
      >
        Completed
      </button>
    </div>
  </div>
 );
};
export default TodoFooter;
```

El resultado hasta este punto es el siguiente

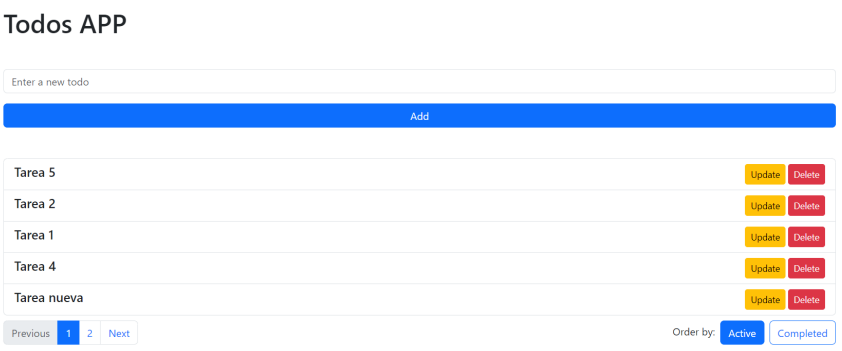

Imagen 1. Resultado de ejecución del frontend
Fuente: Desafío Latam

El código finalizado del frontend lo encontrarás en la plataforma con el nombre ***"Código finalizado - Todo app Frontend (Parte IV)"***