Guía complementaria - Todo App (Parte V)

¡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:

- Crear y validar tokens con JWT.
- Generar rutas protegidas con JWT.
- Vincular Backend y Frontend con JWT.

¡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 - Autenticación y Autorización de usuarios con JWT"**, por ende, si no entiendes algún concepto, te recomendamos revisar dicha guía.

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

Contexto antes de iniciar

El material que veremos a continuación, servirá para conectar nuestra aplicación de todos con un sistema de validación de accesos y restricción de recursos como JWT. Para lograrlo, debemos modificar la base de datos en el Backend y agregar algunas características extras en el Frontend como Context, React Router, entre otros.

Te recomendamos seguir los pasos y anotar los puntos que te parezcan relevantes.

¡Manos a la obra! - Token con JWT (Backend)

A continuación, seguiremos trabajando en nuestra aplicación de todos, en esta ocasión añadiremos algunas configuraciones para hacer funcionar el sistema de JWT para la autenticación de usuarios y restricción de recursos.

- **Paso 1:** Añadiremos una configuración en la base de datos, esto dado que necesitamos incorporar una tabla de users.

  Vamos a crear la tabla users, toma en consideración que la tabla todos tiene una relación con la tabla users por lo que debemos crearla después de la tabla users. Te recomendamos reiniciar tu base de datos para que no tengas problemas con la creación de las tablas y los futuros registros que vamos a insertar.

```javascript
DROP TABLE IF EXISTS todos;
DROP TABLE IF EXISTS users;

-- Crear tabla "users"
CREATE TABLE users (
  user_id SERIAL PRIMARY KEY,
  email VARCHAR(100) NOT NULL UNIQUE,
  password VARCHAR(300) NOT NULL
);

-- crear tabla todos
CREATE TABLE todos (
  id SERIAL PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  done BOOLEAN NOT NULL DEFAULT false,
```

```
  user_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE
);
```

Como resultado veremos el siguiente mensaje en la consola.

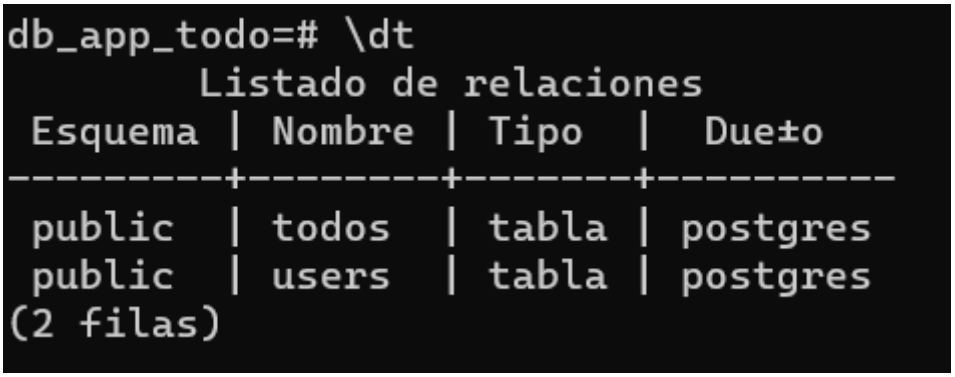


Imagen 1. Creando las tablas users y todos
Fuente: Desafío Latam

Login & Register

- **Paso 2:** Para crear y validar a los usuarios comenzaremos configurando las rutas de login y register en un nuevo archivo

routes\user.route.js

```javascript
import { Router } from "express";

const router = Router();

router.post("/login", (req, res) => {});
router.post("/register", (req, res) => {});

export default router;
```

- **Paso 3:** En nuestro index.js importaremos las rutas y las usaremos como middleware.

```javascript
import cors from "cors";
```

```
import "dotenv/config";
import express from "express";

import todoRoute from "./routes/todo.route.js";
import userRoute from "./routes/user.route.js";

const app = express();

app.use(express.json());
app.use(cors());
app.use("/todos", todoRoute);
app.use("/users", userRoute);

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

app.listen(PORT, () => {
 console.log(`Server listening on port http://localhost:${PORT}`);
});
```

- **Paso 4:** Para separar la lógica de las rutas, vamos a crear un archivo user.controller.js en la carpeta controllers y vamos a importar las funciones que vamos a usar en nuestro archivo routes.js.

controllers\user.controller.js

```
JavaScript
const login = async (req, res) => {};

const register = async (req, res) => {};

export const userController = {
  login,
  register,
};
```

- **Paso 5:** Volvemos al archivo de rutas y asignamos las funciones que definimos en el controlador de users.

```
JavaScript
import { Router } from "express";
```

```
import { userController } from "../controllers/user.controller.js";

const router = Router();

router.post("/login", userController.login);
router.post("/register", userController.register);

export default router;
```

- **Paso 6:** En nuestro modelo user.model.js vamos a crear las funciones que nos permitirán interactuar con la base de datos.

models\user.model.js

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

const findOneEmail = async (email) => {
  const query = "SELECT * FROM users WHERE email = $1";
  const { rows } = await pool.query(query, [email]);
  return rows[0];
};

const create = async ({ email, password }) => {
  const query =
    "INSERT INTO users (email, password) VALUES ($1, $2) RETURNING *";
  const { rows } = await pool.query(query, [email, password]);
  return rows[0];
};

export const userModel = {
  findOneEmail,
  create,
};
```

¡Importante! Tenemos dos entidades relevantes, Users y Todos. Estas entidades están relacionadas entre sí. Dentro de la lógica y estructura de nuestro proyecto creamos un archivo de rutas, controlador y modelo por separado, esto permite modularizar el código y tener una estructura entendible y mantenible en el tiempo.

Encriptar contraseñas

Para encriptar las contraseñas de nuestros usuarios vamos a usar la librería bcryptjs, para ello debemos instalar la dependencia con el comando:

```
JavaScript
npm i bcryptjs
```

- **Paso 7:** Agreguemos bcryptjs en el controlador de user.

controllers\user.controller.js

```
JavaScript
import bcript from "bcryptjs";
import { userModel } from "../models/user.model.js";

const register = async (req, res) => {
  const { email, password } = req.body;
  try {
    await userModel.create({
      email,
      password: bcript.hashSync(password, 10),
    });

     return res.status(201).json({ message: "User created successfully"
});
  } catch (error) {
    console.log(error);

     // recuerda que estos códigos de error los puedes modularizar como
vimos en todo.controller.js
    if (error.code === "23505") {
      return res.status(400).json({ message: "User already exists" });
    }

    return res.status(500).json({ message: "Internal server error" });
  }
};

const login = async (req, res) => {
  const { email, password } = req.body;
  try {
```

```
    const user = await userModel.findOneEmail(email);
    if (!user) {
      return res.status(400).json({ message: "User not found" });
    }

    const isMatch = bcript.compareSync(password, user.password);
    if (!isMatch) {
      return res.status(400).json({ message: "Invalid credentials" });
    }

    return res.status(200).json({ message: "User logged successfully" });
  } catch (error) {
    console.log(error);
    return res.status(500).json({ message: "Internal server error" });
  }
};

export const userController = {
  login,
  register,
};
```

- **Paso 8:** Probemos nuestro código hasta el momento utilizando Thunder Client o Postman:

Ruta para probar la acción de login

```
JavaScript
POST http://localhost:5000/users/register
```

En el cuerpo de la consulta insertamos los siguientes datos:

```
JavaScript
{
  "email": "test@test.com",
  "password": "123123"
}
```

Al ingresar esos datos en el body mediante JSON veremos la respuesta:

```JavaScript
{
  "message": "User not found"
}
```

“User not found” se genera dado que en nuestra base de datos no existe ningún usuario con estos valores. Para solucionarlo debemos entonces agregar un nuevo usuario, en este caso debemos utilizar la ruta de register

```JavaScript
POST http://localhost:5000/users/register
```

El cuerpo de la consulta mediante JSON será el siguiente:

```JavaScript
{
  "email": "test1@test.com",
  "password": "123123"
}
```

Al utilizar la ruta de /register para ingresar un nuevo usuario, veremos el siguiente resultado:

```JavaScript
{
 "message": "User created successfully"
}
```

Si vuelves a ejecutar ahora la ruta de /login, obtendrás el siguiente mensaje

```JavaScript
{
 "message": "User logged successfully"
```

```
}
```

- **Paso 9:** Vamos a probar acciones como registrar un usuario que ya existe y loguear un usuario con una contraseña distinta a la que se haya registrado:

Ruta

```
JavaScript
POST localhost:5000/users/register
```

Cuerpo de la consulta JSON

```
JavaScript
{
  "email": "test1@test.com",
  "password": "123123"
}
```

Mensaje obtenido

```
JavaScript
{
  "message": "User already exists"
}
```

Invalid credentials
Si intentamos hacer login con credenciales distintas en un usuario recibiremos el siguiente mensaje

Ruta

```
JavaScript
POST localhost:5000/users/login
```

Cuerpo de la consulta JSON

```
JavaScript
{
  "email": "test1@test.com",
  "password": "constraseña distinta"
}
```

Recuerda que para el usuario con el email test1@test.com definimos que su contraseña es "123123".

Mensaje obtenido

```
Java
{
  "message": "Invalid credentials"
}
```

Con estas validaciones, hemos logrado comprobar que el comportamiento de nuestro Backend es efectivo en cuanto a las siguientes funcionalidades:

- Registro de usuario.
- Login de usuarios.
- Evitar la duplicación de usuarios.
- Hacer login con credenciales distintas.

JSON Web Token

Ahora en nuestro proyecto vamos a configurar la librería JSON Web Token para poder generar los tokens.

- **Paso 10:** Instalamos la dependencia

```
JavaScript
npm i jsonwebtoken
```

- **Paso 11:** Llevaremos la palabra secreta a un archivo .env para que no se vea expuesta en nuestro código.

```
JavaScript
JWT_SECRET=increiblementeSecreto
```

.env

```
JavaScript
PGUSER="postgres"
PGHOST="localhost"
PGPASSWORD="123456"
PGDATABASE="db_app_todo"
PGPORT=5432
JWT_SECRET=increiblementeSecreto
```

- **Paso 12:** Ahora vamos a generar el token en nuestro controlador login.

controllers\user.controller.js

```
JavaScript
import "dotenv/config";
import bcript from "bcryptjs";
import jwt from "jsonwebtoken";
import { userModel } from "../models/user.model.js";

const register = async (req, res) => {
 const { email, password } = req.body;
 try {
  await userModel.create({
   email,
   password: bcript.hashSync(password, 10),
  });

  return res.status(201).json({ message: "User created successfully" });
 } catch (error) {
  console.log(error);

  // recuerda que estos códigos de error los puedes modularizar como vimos en
todo.controller.js
  if (error.code === "23505") {
   return res.status(400).json({ message: "User already exists" });
  }

  return res.status(500).json({ message: "Internal server error" });
```

```
 }
};

const login = async (req, res) => {
 const { email, password } = req.body;
 try {
  const user = await userModel.findOneEmail(email);
  if (!user) {
   return res.status(400).json({ message: "User not found" });
  }

  const isMatch = bcript.compareSync(password, user.password);
  if (!isMatch) {
   return res.status(400).json({ message: "Invalid credentials" });
  }

  // creación del payload
  const payload = {
   email,
   user_id: user.user_id,
  };

  // creación del token
  const token = jwt.sign(payload, process.env.JWT_SECRET);

  return res.status(200).json({
   message: "Login successfully",
   token,
   email,
  });
 } catch (error) {
  console.log(error);
  return res.status(500).json({ message: "Internal server error" });
 }
};

export const userController = {
 login,
 register,
};
```

- **Paso 13**: Para validar los token en las rutas, implementaremos un middleware que se encargue de realizar dicha tarea, crea el directorio para los middlewares e ingresa el siguiente código.

middlewares/auth.middleware.js

```javascript
import "dotenv/config";
import jwt from "jsonwebtoken";

export const authMiddleware = (req, res, next) => {
  const token = req.headers.authorization?.split(" ")[1];
  if (!token) {
    return res.status(401).json({ error: "No token provided" });
  }

  try {
    const payload = jwt.verify(token, process.env.JWT_SECRET);
    req.user = payload;
    next();
  } catch (error) {
    console.log(error);
    return res.status(401).send({ error: "Invalid token" });
  }
};
```

- **Paso 14:** Ahora vamos a usar nuestro middleware en las rutas que queremos proteger.

routes\todo.route.js

```javascript
import { todoController } from "../controllers/todo.controller.js";
import { authMiddleware } from "../middlewares/auth.middleware.js";

import { Router } from "express";

const router = Router();

// GET /todos
router.get("/", authMiddleware, todoController.read);

// GET /todos/:id
router.get("/:id", authMiddleware, todoController.readById);

// POST /todos
```

```
router.post("/", authMiddleware, todoController.create);

// PUT /todos/:id
router.put("/:id", authMiddleware, todoController.update);

// DELETE /todos/:id
router.delete("/:id", authMiddleware, todoController.remove);

export default router;
```

- **Paso 15:** Como nuestra tabla "todos" tiene una relación con la tabla "users" vamos a modificar el modelo todo.model.js para poder interactuar con la base de datos.

models\todo.model.js

JavaScript

```
import "dotenv/config";
import format from "pg-format";
import { pool } from "../database/connection.js";
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, user }) => {
  // Consulta para contar el número total de filas en la tabla 'todos'
  const countQuery = "SELECT COUNT(*) FROM todos WHERE user_id = $1";
     const { rows: countResult } = await pool.query(countQuery,
[user.user_id]);
  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 WHERE user_id = %s ORDER BY done %s LIMIT %s
OFFSET %s";
  const offset = (page - 1) * limit;
    const formattedQuery = format(query, user.user_id, 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}`,
  };
};

const findById = async (id, user) => {
  const query = "SELECT * FROM todos WHERE id = $1 WHERE user_id = $2";
  const { rows } = await pool.query(query, [id, user.user_id]);
  return rows[0];
};

const create = async (todo, user) => {
  console.log({ user });
  const query =
      "INSERT INTO todos (title, done, user_id) VALUES ($1, $2, $3)
RETURNING *";
  const { rows } = await pool.query(query, [
    todo.title,
    todo.done,
    user.user_id,
  ]);
```

```
  return rows[0];
};

const remove = async (id, user) => {
   const query = "DELETE FROM todos WHERE id = $1 AND user_id = $2
RETURNING *";
  const { rows } = await pool.query(query, [id, user.user_id]);
  return rows[0];
};

const update = async (id, user) => {
  const query =
     "UPDATE todos SET done = NOT done WHERE id = $1 AND user_id = $2
RETURNING *";
  const { rows } = await pool.query(query, [id, user.user_id]);
  return rows[0];
};

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

- **Paso 16:** Ahora vamos a modificar nuestro controlador todo.controller.js para que el modelo pueda recibir la información del usuario, en este caso lo estamos sacando del req.user que es donde lo guardamos en nuestro middleware.

controllers\todo.controller.js

JavaScript

```
import { getDatabaseError } from "../lib/errors/database.error.js";
import { todoModel } from "../models/todo.model.js";

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
```

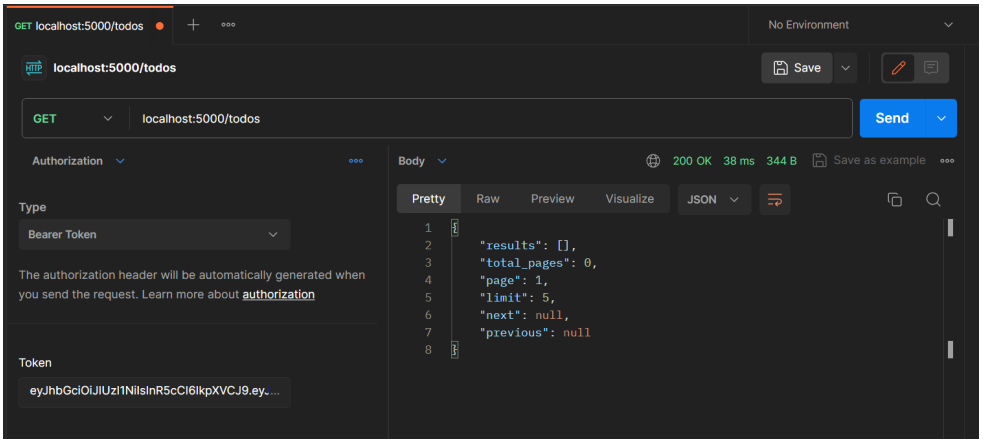
```
  const isPageValid = /^[1-9]\d*$/.test(page);

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

  try {
    const todos = await todoModel.findAll({
      limit,
      order,
      page,
      user: req.user,
    });
    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" });
  }
};

const readById = async (req, res) => {
  const id = req.params.id;

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

    if (!todo) {
      res.status(404).json({ message: "Todo not found" });
    }
    res.json(todo);
  } 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" });
  }
};

const create = 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, req.user);
    return res.status(201).json(todo);
  } 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" });
  }
};

const update = async (req, res) => {
  const id = req.params.id;

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

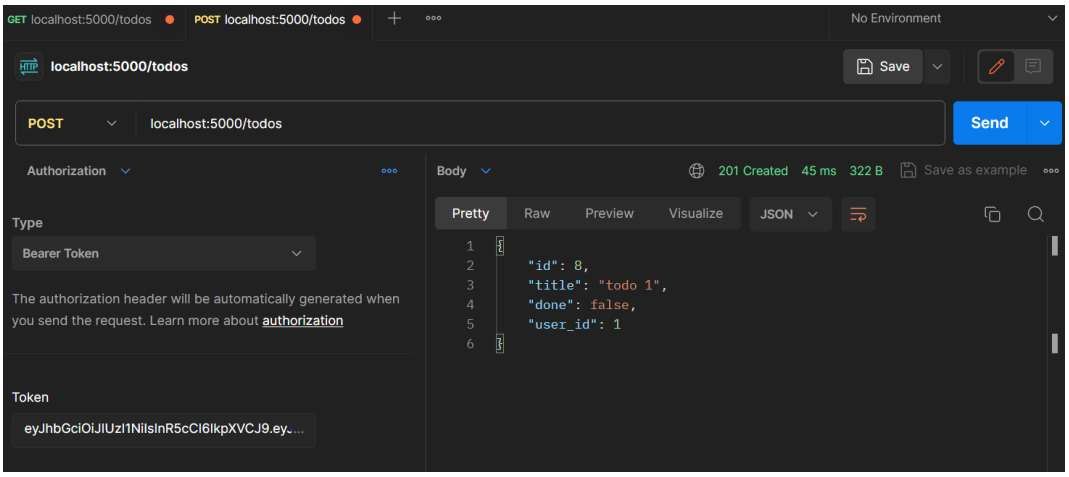
```
    }
    return res.status(500).json({ message: "Internal server error" });
  }
};

const remove = async (req, res) => {
  const id = req.params.id;

  try {
    const todo = await todoModel.remove(id, req.user);
    if (!todo) {
      return res.status(404).json({ message: "Todo not found" });
    }
    return res.json({ message: "Todo deleted" });
  } 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" });
  }
};

export const todoController = {
  read,
  readById,
  create,
  update,
  remove,
};
```

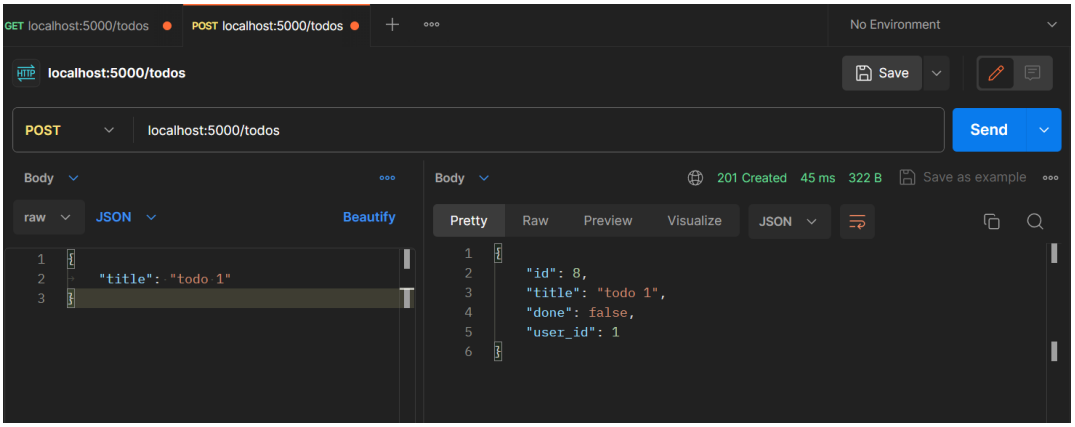
Prueba de rutas

GET /todos

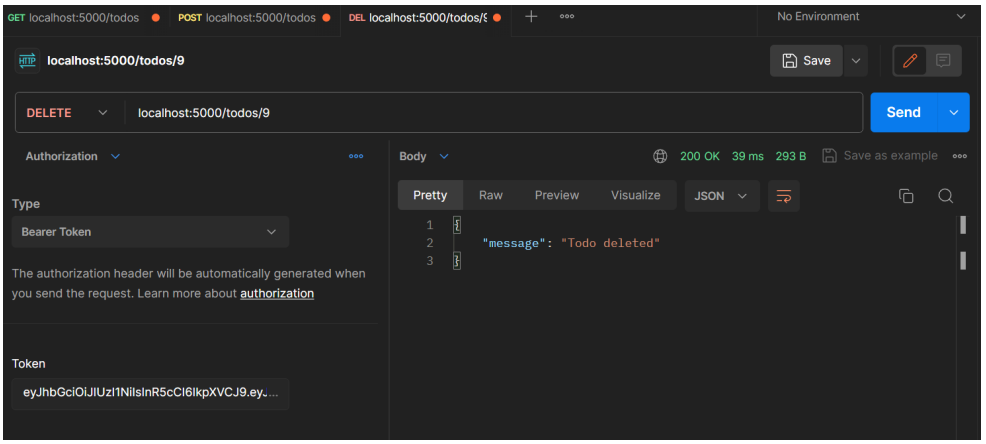


POST: /todos

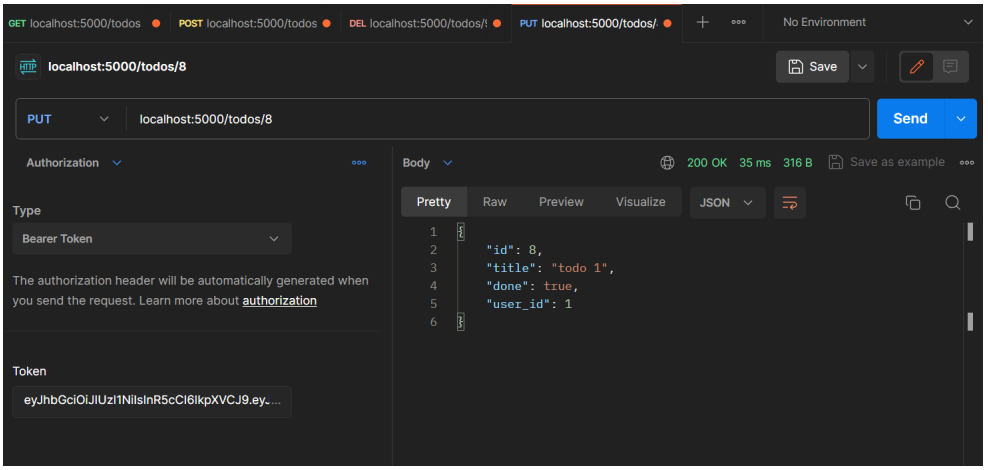




DELETE: /todos/:id



PUT: /todos/:id



¡Manos a la obra! - Token con JWT (Frontend)

Ahora trabajaremos con la integración de frontend, donde implementaremos el sistema de rutas con React Router y Context.

Configuración del router

- **Paso 1:** Para trabajar con diferentes rutas vamos a instalar la librería react-router-dom.

```
JavaScript
npm install react-router-dom@6.20.0
```

- **Paso 2:** Configuramos BrowserRouter en nuestro archivo main.jsx.

main.jsx

```
JavaScript
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.jsx";
import "./index.css";

import { BrowserRouter } from "react-router-dom";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

- **Paso 3:** Ahora vamos a crear tres páginas login, register y home.

pages/Login.jsx

```
JavaScript
const LoginPage = () => {
  return <div>LoginPage</div>;
};
export default LoginPage;
```

pages\Register.jsx

```
JavaScript
const RegisterPage = () => {
  return <div>RegisterPage</div>;
};
export default RegisterPage;
```

- **Paso 4:** En la página home trasladaremos el código de la página App.jsx. Ten en cuenta que las rutas de los componentes se modificaron.

pages/Home.jsx

```javascript
import { useEffect, useState } from "react";
import TodoFooter from "../components/TodoFooter";
import TodoForm from "../components/TodoForm";
import Todos from "../components/Todos";

const BASE_URL = import.meta.env.VITE_BASE_URL;

const HomePage = () => {
  const [todos, setTodos] = useState([]);
  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");

  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);
  };

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

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

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

  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>
  );
};
export default HomePage;
```

App.jsx

```JavaScript
import { Route, Routes } from "react-router-dom";

import HomePage from "./pages/Home";
import LoginPage from "./pages/Login";
import RegisterPage from "./pages/Register";

const App = () => {
  return (
    <div>
      <Routes>
        <Route
          path="/"
          element={<HomePage />}
        />
        <Route
          path="/login"
          element={<LoginPage />}
        />
        <Route
          path="/register"
          element={<RegisterPage />}
        />
      </Routes>
    </div>
  );
};
export default App;
```

Configuración de Context

Usaremos un context para poder compartir la información del usuario en toda la aplicación. Además, crearemos dos métodos para poder hacer el login (loginWithEmailAndPassword) y el registro (registerWithEmailAndPassword) de usuarios.

El token lo guardaremos en localstorage para poder mantener la sesión del usuario y hacer peticiones a las rutas protegidas.

Para aplicaciones avanzadas no se recomienda guardar el token en localstorage, sino por ejemplo en una cookie con el atributo httpOnly (entre varias otras formas) para evitar ataques XSS. Pero esto se escapa del alcance de este curso.

- **Paso 5:** Definimos el provider como UserProvider.jsx

src/providers/UserProvider.jsx

JavaScript

```
import { createContext, useEffect, useState } from "react";

export const UserContext = createContext();

const BASE_URL = import.meta.env.VITE_BASE_URL;

const initialStateToken = localStorage.getItem("token") || null;

const UserProvider = ({ children }) => {
  const [token, setToken] = useState(initialStateToken);

  useEffect(() => {
    if (token) {
      localStorage.setItem("token", token);
    } else {
      localStorage.removeItem("token");
    }
  }, [token]);

  const loginWithEmailAndPassword = async (email, password) => {
    const response = await fetch(`${BASE_URL}/users/login`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ email, password }),
    });

    const data = await response.json();
    setToken(data.token || null);

    return data;
```

```
  };

  const registerWithEmailAndPassword = async (email, password) => {
    const response = await fetch(`${BASE_URL}/users/register`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ email, password }),
    });
    const data = await response.json();
    return data;
  };

  const logout = () => {
    setToken(null);
  };

  return (
    <UserContext.Provider
      value={{
        loginWithEmailAndPassword,
        registerWithEmailAndPassword,
        token,

        logout,
      }}
    >
      {children}
    </UserContext.Provider>
  );
};

export default UserProvider;
```

- **Paso 6:** Ahora, desde el main.jsx habilitamos el Provider para hacerlo disponible en todos los niveles de la aplicación.

main.jsx

JavaScript

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.jsx";
import "./index.css";

import { BrowserRouter } from "react-router-dom";
import UserProvider from "./providers/UserProvider";
```

```
ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <UserProvider>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </UserProvider>
  </React.StrictMode>
);
```

Utilizando el token

Ya con el context configurado, es hora de comenzar a utilizar el token en nuestras páginas y componentes, aquí dejamos una implementación simple de como podrías hacerlo.

src/components/Navbar.jsx

JavaScript

```
import { useContext } from "react";
import { Link } from "react-router-dom";
import { UserContext } from "../providers/UserProvider";

const Navbar = () => {
  const { token, logout } = useContext(UserContext);

  return (
    <div className="navbar navbar-dark bg-dark">
      <div className="container">
        <span className="navbar-brand">Todo App</span>
        <div>
          {token ? (
            <>
              <Link
                to="/"
                className="btn btn-sm btn-outline-light me-2"
              >
                Home
              </Link>
              <button
                className="btn btn-sm btn-outline-danger"
                onClick={logout}
              >
```

```
              logout
            </button>
          </>
        ) : (
          <>
            <Link
              to="/login"
              className="btn btn-sm btn-outline-light me-2"
            >
              Login
            </Link>
            <Link
              to="/register"
              className="btn btn-sm btn-outline-light"
            >
              Register
            </Link>
          </>
        )}
      </div>
    </div>
  </div>
 );
};
export default Navbar;
```

src/App.jsx

JavaScript

```
import { useContext } from "react";
import { Navigate, Route, Routes } from "react-router-dom";

import { UserContext } from "./providers/UserProvider";

import Navbar from "./components/Navbar";
import HomePage from "./pages/Home";
import LoginPage from "./pages/Login";
import RegisterPage from "./pages/Register";

const App = () => {
  const { token } = useContext(UserContext);
```

```
  return (
    <div>
      <Navbar />
      <Routes>
        <Route
          path="/"
          element={token ? <HomePage /> : <Navigate to="/login" />}
        />
        <Route
          path="/login"
          element={token ? <Navigate to="/" /> : <LoginPage />}
        />
        <Route
          path="/register"
          element={<RegisterPage />}
        />
      </Routes>
    </div>
  );
};
export default App;
```

src/pages/Login.jsx

JavaScript

```
import { useContext, useState } from "react";
import { UserContext } from "../providers/UserProvider";

const LoginPage = () => {
  const { loginWithEmailAndPassword } = useContext(UserContext);

  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  const handleSubmit = async (e) => {
    e.preventDefault();
    const response = await loginWithEmailAndPassword(email, password);
    alert(response?.message || "Something went wrong");
  };

  return (
```

```
<div className="container">
  <h1>Login</h1>
  <form onSubmit={handleSubmit}>
    <div className="mb-3">
      <label
        htmlFor="email"
        className="form-label"
      >
        Email
      </label>
      <input
        type="email"
        id="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        className="form-control"
        placeholder="Enter your email"
      />
    </div>
    <div className="mb-3">
      <label
        htmlFor="password"
        className="form-label"
      >
        Password
      </label>
      <input
        type="password"
        id="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        className="form-control"
        placeholder="Enter your password"
      />
    </div>
    <button
      type="submit"
      className="btn btn-primary"
    >
      Login
    </button>
  </form>
```

```
    </div>
  );
};
export default LoginPage;
```

src/pages/Register.jsx

```javascript
import { useContext, useState } from "react";
import { UserContext } from "../providers/UserProvider";

const RegisterPage = () => {
  const { registerWithEmailAndPassword } = useContext(UserContext);

  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  const handleSubmit = async (e) => {
    e.preventDefault();
    const response = await registerWithEmailAndPassword(email, password);
    alert(response?.message || "Something went wrong");
  };

  return (
    <div className="container">
      <h1>Register</h1>
      <form onSubmit={handleSubmit}>
        <div className="mb-3">
          <label
            htmlFor="email"
            className="form-label"
          >
            Email
          </label>
          <input
            type="email"
            id="email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
            className="form-control"
            placeholder="Enter your email"
```

```
          />
        </div>
        <div className="mb-3">
          <label
            htmlFor="password"
            className="form-label"
          >
            Password
          </label>
          <input
            type="password"
            id="password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
            className="form-control"
            placeholder="Enter your password"
          />
        </div>
        <button
          type="submit"
          className="btn btn-primary"
        >
          Create account
        </button>
      </form>
    </div>
  );
};
export default RegisterPage;
```

src/pages/Home.jsx

```
JavaScript
import { useContext, useEffect, useState } from "react";
import TodoFooter from "../components/TodoFooter";
import TodoForm from "../components/TodoForm";
import Todos from "../components/Todos";
import { UserContext } from "../providers/UserProvider";

const BASE_URL = import.meta.env.VITE_BASE_URL;
```

```
const HomePage = () => {
  const { token } = useContext(UserContext);

  const [todos, setTodos] = useState([]);
  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");

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

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

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

  useEffect(() => {
    getTodos(page, order);
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [page, order]);

  const addTodo = async (title) => {
    const response = await fetch(`${BASE_URL}/todos`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${token}`,
      },
      body: JSON.stringify({ title }),
```

```
    });
    console.log({
      response,
    });
    await response.json();
    await getTodos();
  };

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

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

  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>
  );
};
export default HomePage;
```

Con este ejercicio hemos vinculado nuestro backend con nuestro frontend, creando rutas protegidas e interactuando con Postgres, ahora puedes seguir mejorando la aplicación agregando más funcionalidades.

Nota: En la plataforma tendrás acceso al Backend y el Frontend con el código finalizado, **"Código finalizado - Backend Todo App (Parte V)"** y **"Código finalizado - Frontend Todo App (Parte V)"**