APLICACIÓN CRUD PRODUCTOS

William Fernando Salamanca Barrera
José Daniel Rivas Soracá

I. INTRODUCCIÓN

Haciendo uso de diferentes herramientas conocidas dentro del entorno del desarrollo web, se logró construir una aplicación web que realiza las cuatro operaciones básicas de **Create, Read, Update** y **Delete.** Esta aplicación se desplegó de tal forma que se consiguió separar en nodos diferentes la base de datos y el servicio de lectura de un archivo de texto plano almacenado en **Google Drive.**

El despliegue se realizó de dos formas diferentes, una usando nuestra máquina local con **Ngrok** para acceder a este a través de internet desde cualquier otra máquina. Y con **Vercel** para no depender de la disponibilidad de nuestra máquina local y de esta forma tenerlo desplegado en la nube.

II. MARCO DE REFERENCIA

A. Requisitos para ejecutar en un entorno local

Tener instalado:

- NodeJS
- Algún manejador de paquetes (npm, yarn, pnpm).
- Git
- Ngrok

B. Dependencias de producción

- Axios, para hacer peticiones HTTP.
- NextJS, framework de React para la WEB.
- pg, para interactuar con una base de datos Postgresql desde NodeJS.
- React, para crear la UI.
- react-hot-toast, para tener componentes de notificación ya creados.

C. Desarrollo

- Typescript, para tener los beneficios de un lenguaje tipado.
- TailwindCSS, framework de CSS para dar estilos a nuestra UI.

D. Requisitos para despliegue con Ngrok y/o Vercel

- Cuenta en Ngrok
- Cuentra en Vercel
- Cuenta en GitHub

E. Diagramas de despliegue

El despliegue haciendo uso de Ngrok lo podemos ver en la Figura 1, en esta podemos ver que tenemos un nodo para representar el servicio de lectura de un archivo de texto plano a través de Google Drive. Hacia la izquierda vemos el nodo que representa la base de datos Postgresql alojada en Railway.

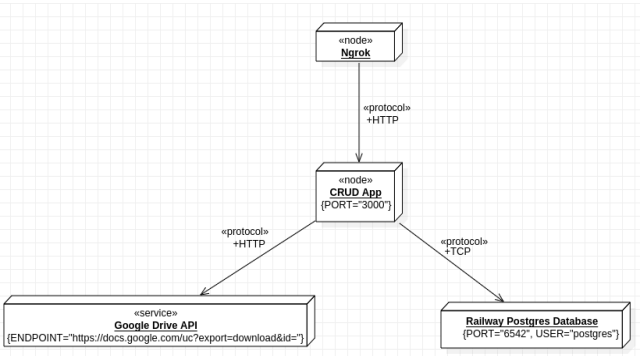

Figura 1. Diagrama de despliegue con Ngrok

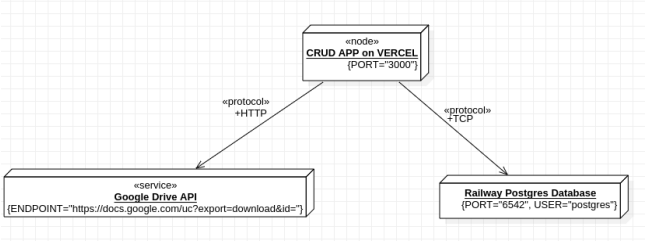

Figura 2. Diagrama de despliegue con Vercel

III. MÉTODOS Y DESCRIPCIÓN

A. CRUD de productos

```
- /.next
- /node_modules
- /public
- /src
    - /components
    - /config
        - /env
    - /database
    - /pages
        - /products
            - /edit
        - /api
    - /server
        - /controllers
        - /services
    - /styles
    - /types
- .env.example
- .eslintrc.json
- .gitignore
- next.config.js
- package.json
- postcss.config.js
- tailwind.config.js
- tsconfig.json
```

Figura 3. Estructura general de archivos y carpetas.

```
'./src/pages/api/products/[id].ts'

import { deleteProduct, getProduct, updateProduct } from "@app/server/controllers/product.controller";
import { NextApiRequest, NextApiResponse } from "next";  Calculating...

export default async function handler(req: NextApiRequest, res:NextApiResponse) {
  switch (req.method) {
    case "GET":
      return await getProduct(req, res);
    case "DELETE":
      return await deleteProduct(req, res);
    case "PUT":
      return await updateProduct(req, res);
    default:
      return res.status(400).json({ message: "bad request" });
  }
}
```

Figura 4. Handler de la API, para ruta con id

NextJS identifica todo lo que esté dentro de la carpeta "pages" como una ruta, con excepción de los archivos que nombremos con un "_" al inicio, como es el caso de "_document.tsx", dónde podemos de definir por ejemplo un título general para toda la aplicación o también por ejemplo los metadatos de la cabecera del documento.

Otro caso especial a tener en cuenta cuando nombramos archivos dentro de la carpeta "pages", es que para rutas dinámica hacemos uso de "[]", por ejemplo, si queremos acceder a un producto directamente desde la API con el id 12 lo haríamos a la ruta "/api/products/12" siendo "/src/pages/api/products/[id].ts" (Figura 4) el archivo que se encarga de manejar los valores dinámicos que vengan después del "/products/".

```
'./src/pages/api/index.ts'
import {
  getProducts,
  saveProduct,
} from "@app/server/controllers/product.controller";
import { NextApiRequest, NextApiResponse } from "next";

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {

  switch (req.method) {
    case "GET":
      return await getProducts(req, res);
    case "POST":
      return await saveProduct(req, res);
    default:
      return res.status(400).send("Method not allowed");
  }
}
```

Figura 5. Handler de la API, para ruta sin id

Siendo el caso contrario, al hacer una petición a "/api/products" sería el archivo "index.ts" (Figura 5) el que se encargue de manejar las peticiones.

```
'./src/server/controllers/product.controller.ts'
const getProducts = async (req: NextApiRequest, res: NextApiResponse) => {
  try {
    const results = await getAll();
    return res.status(200).json(results);
  } catch (error) {
    return res.status(500).json({ error });
  }
};
```

Figura 6. Controller de producto

Siguiendo el flujo de lo que sería una petición al endpoint que maneja el archivo de la Figura 5, podemos ver, que dependiendo del verbo HTTP con el que venga el request accederemos a una u otra lógica. Por convención decidimos que ya sea POST o GET la petición HTTP, nos dirigiremos a un mismo archivo que ejecutará una u otra función según sea el caso (Figura 6). Por ejemplo, si al archivo de la Figura 5 le llega una petición GET se dirigirá al archivo de la Figura 6, que es donde está definido el **controller** de la entidad producto. Siguiendo el ejemplo, en este archivo, ejecuta la función "getProducts" que se encarga de llamar a la función "getAll" definida en el archivo de servicios de producto (Figura 7). De esta forma el controller se encargará de resolver y responder peticiones HTTP solicitando información a

la base de datos a través de un servicio previamente definido.

```
'./src/server/servicies/product.service.ts'
import { pool } from "@app/config/db.connection";
import { Product, ProductToInsert } from "@app/types";

const getAll = async () => {
  return await pool.query("SELECT * FROM product");
};
```

Figura 7. Servicio, obtener todos los productos

La función getAll de la Figura 7 se encarga de usar el pool de conexión a la base de datos creada en el archivo de la Figura 8. El pool nos permite ejecutar queries, dónde inyectamos SQL al argumento de la función. En este caso, el que se ve en la Figura 7, seleccionamos todos los campos de la tabla **product.** Si nos fijamos, en el archivo de la Figura 8, además de crear el pool de conexión verificamos si existe la tabla de **product,** de no existir la creará, caso contrario omitirá esa instrucción.

```
'./src/config/db.connection.ts'
import { Pool } from "pg";

import { ENV } from "./env/env.variables";

const { DB_HOST, DB_NAME, DB_PORT, DB_USER, DB_USERPASS } = ENV;

let pool: any;

const connectToDB = async () => {
  pool = new Pool({
    host: DB_HOST,
    user: DB_USER,
    password: DB_USERPASS,
    port: DB_PORT,
    database: DB_NAME,
  });
  await pool.connect();
  await pool.query(
    "CREATE TABLE IF NOT EXISTS product (id SERIAL PRIMARY KEY, name VARCHAR(200), description VARC
  );
};

connectToDB();
```

Figura 8. Creando Pool de conexión con la base de datos

Cuando trabajamos con “pg” hay que tener en cuenta algo importante cuando tratamos de persistir datos en nuestra base de datos, y es que en el cuerpo de la query, donde normalmente van los valores, tenemos que indicar que ingresamos valores dinámicos, es decir, que no siempre serán los mismos, es por esto que dentro de “VALUES” (Figura 9) indicamos con el signo “$” y un número el orden en que van a ser ingresados los valores. Como segundo parámetro de la query va, en un arreglo, los valores en el orden que ya establecimos.

```
const saveOne = async (product: ProductToInsert) => {
  const { name, description, price } = product;

  const { fields } = await pool.query(
    "INSERT INTO product(name, description, price) VALUES ($1, $2, $3) RETURNING *",
    [name, description, price]
  );
  return { ...product, id: fields[0] };
};
```

Figura 9. Servicio, guardar un producto

Una vez entendido cómo interactuamos con la base de datos a través de nuestros servicios y controladores, podemos hacer peticiones a la API de productos desde la UI.

En la página principal de la aplicación (Figura 10) renderizamos los productos contenidos en la base de datos, además de mostrar el contenido del archivo de texto plano que creamos desde Google Drive del cual se hablará después.

NextJS trabaja de tal forma que nos permite interactuar con servicios externos desde nuestra UI a través de una función llamada “getServerSideProps”. En la Figura 11 vemos la lógica de esta función para obtener la información de todos los productos desde la base de datos y el acceso al contenido de nuestro archivo desde Google Drive.

```
'./src/pages/index.tsx'
function ProductsPage({ products = [], googleDocData }: Props) {
  const renderProducts = () => {
    if (products.length === 0) return <h1>No hay productos</h1>;
    return products.map((product) => (
      <ProductCard key={product.id} product={product} />
    ));
  };

  return (
    <Layout>
      <div className="max-w-md py-4 px-8 bg-white shadow-lg rounded-lg my-10">
        <h2 className="text-gray-800 text-2xl font-semibold">
          Contenido cargado desde Google Drive
        </h2>
        <p className="mt-2 text-gray-600">{googleDocData}</p>
      </div>
      <div className="grid gap-4 grid-cols-1 md:grid-cols-4">
        {renderProducts()}
      </div>
    </Layout>
  );
}
```

Figura 10. Página principal

```
'./src/pages/index.tsx'
export const getServerSideProps = async () => {

  const products = await axios
    .get(`${ENV.SERVER}/api/products`)
    .then(({ data }) => data)
    .catch((error) => console.error("ERROR AL ACCEDER AL ENDPOINT", error?.config?.url))

  const googleDocID = ENV.ID_ARCH_PLANO_DRIVE;

  let dataGoogleDrive = "Error al cargar el contenido";
  try {
    const res = await axios.get(
      `https://docs.google.com/uc?export=download&id=${googleDocID}`
    );

    dataGoogleDrive = res.data;
  } catch (error) {
    console.log("ERROR AL ENCONTRAR EL DOCUMENTO DE GOOGLE DRIVE", error);
  }

  return {
    props: {
      products: (products != undefined ? products.rows : []),
      googleDocData: dataGoogleDrive,
    },
  };
};
```

Figura 11. Obteniendo información de la API

Antes de pasar a lo que es la configuración con los servicios externos es importante ver primero qué variables de entorno usamos para poder conectarnos a estos. En la Figura 12 vemos los nombres de variables que están contenidos en el archivo ".env.example" en la ruta principal del proyecto (Figura 3). Con base a este archivo crearemos un nuevo archivo llamado ".env.local" en la ruta principal del proyecto. En este archivo definiremos las variables de entorno con sus valores una vez hayamos configurado los demás servicios.

```
DB_PORT=
DB_HOST=
DB_USER=
DB_PASSWORD=
DB_NAME=
ID_ARCH_PLANO_DRIVE=
SERVER=
```

Figura 12. Nombres de las variables de entorno

B. Configuración del acceso al archivo de texto plano alojado en Google Drive

Para crear un archivo en Google Drive es necesario tener una cuenta. Ya teniendo una tendremos dos formas de crear un archivo plano. La primera es subir un archivo ".txt" existente a Google Drive. La segunda es agregando un aplicación a Google Drive que nos permita editar y crear archivos de texto plano, en este caso usamos TextEditor.

Una vez creado o subido el archivo a Google Drive, tenemos que generar un enlace para compartirlo (Figura 13)

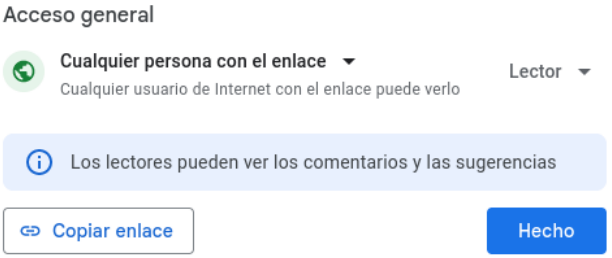

Figura 13. Compartiendo archivo de texto plano

Un ejemplo de lo que sería el enlace que nos genera es **"https://drive.google.com/file/d/1LSQzDmTCWe3aIL9fkve9lvSEjKdfCHQg/view"**. Para este caso (siendo 19 de Febrero de 2023), así es como se ve el enlace para la vista del documento. Pero este no nos sirve tal y como está, ya que lo que necesitamos es el ID de este para poder ingresarlo como una variable de entorno en nuestra aplicación, es por esto que a partir de esta ruta tenemos que inferir el ID.
El ID es el que se encuentra después del "d/" y va hasta el siguiente "/", en este caso sería **"1LSQzDmTCWe3aIL9fkve9lvSEjKdfCHQg"**.
Este va en la variable de entorno **"ID_ARCH_PLANO_DRIVE"** (Figura 12).

Con esto ya podremos ver el contenido de este documento desde nuestra aplicación.

C. Configuración de la base de datos en Railway.

Como ya se mencionó, se usará Postgresql como motor de base de datos, y para poder acceder a una de forma remota, en un nodo independiente, se hizo uso de una plataforma llamada Railway. Al ingresar a esta creamos un proyecto (Figura 14), entre las diferentes opciones que nos ofrece Railway se eligió el que necesitábamos, que era el de Postgresql como se ve en la Figura 15.

Una vez creado, en el panel de opciones del proyecto se accedió al de "Variables" (Figura 16) para poder ver las credenciales que necesitábamos para poder crear la conexión con éxito desde la aplicación de CRUD de productos.

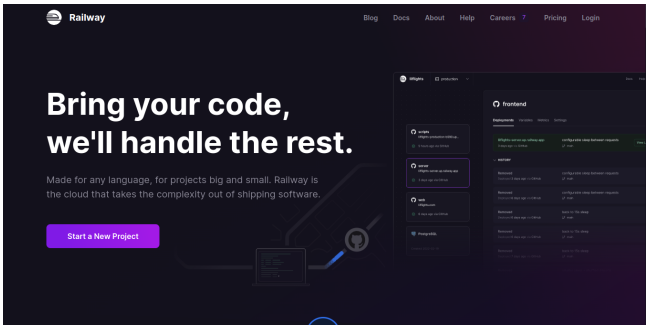

Figura 14. Página de inicio Railway

Ya teniendo el valor de las variables de entorno que nos da Railway simplemente las ingresamos en nuestro archivo de variables de entorno local, el que creamos a partir de ".env.example" (Figura 12) y que llamamos ".env.local".

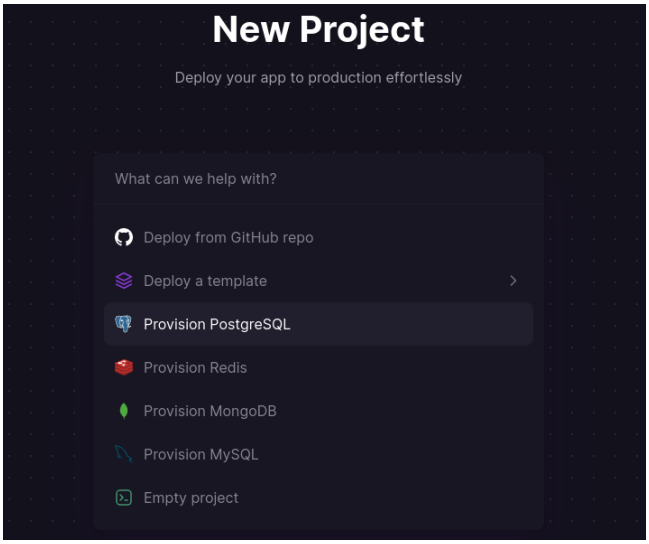

Figura 15. Selección de Postgresql como proyecto en Railway

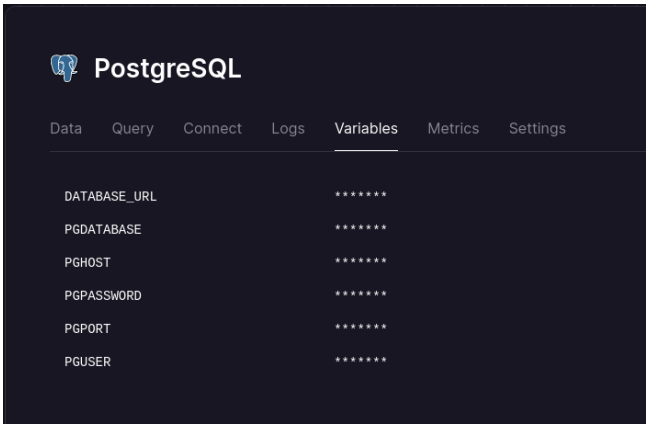

Figura 16. Variables de conexión de la base de datos

IV. CONCLUSIONES

- El tener desplegada una aplicación en diferentes nodos nos beneficia de varias formas, una de estas es el performance, ya que es diferente tener ejecutando dos o tres nodos en un mismo servidor a tenerlos en en servidores diferentes.

- Tener un sistema distribuido en general nos ayuda a aumentar la resiliencia de nuestro sistema, ya que el fallo de un nodo, dependiendo del contexto, puede no representar el fallo del sistema entero.

V. REFERENCIAS

"CRUD NextJS." *Crear una aplicación en NextJS*, https://dev.to/integridsolutions/building-a-crud-app-with-nextjs-react-query-react-hook-form-and-yup-46o9.

"Git." *Git*, https://git-scm.com/.

"NextJS." *NextJS*, https://nextjs.org/.

"NodeJS." *NodeJS*, https://nodejs.org/en/.

"PG en NodeJS." *pg con nodejs*, https://www.digitalocean.com/community/tutorials/how-to-use-postgresql-with-node-js-on-ubuntu-20-04.

"TailwindCSS." *TailwindCSS*, https://tailwindcss.com/.