Trabajo Practico Base de Datos

Integración de Bases de Datos Relacionales y NoSQL

6 de Diciembre de 2024

Profesor		
Hernan Merlino		
Estudiantes		

INTRODUCCIÓN

El trabajo tiene como objetivo desarrollar una aplicación web que integre bases de datos relacionales y NoSQL, permitiendo realizar operaciones CRUD (crear, leer, actualizar y eliminar).

A lo largo de este trabajo, se abordará tanto la elección de las tecnologías utilizadas, se explorará la integración de ambas bases de datos en una misma aplicación, permitiendo entender cómo interactúan y se complementan entre sí en el ámbito del desarrollo web.

IMPLEMENTACIÓN

Elegimos **Node.js** para el backend debido a su simplicidad y legibilidad, lo que facilita el desarrollo rápido.

Los Frameworks:

- **FastAPI**: Utilizamos FastAPI para el desarrollo del backend debido a su alto rendimiento y su capacidad para manejar solicitudes asíncronas.
- **React:** Para el frontend por su capacidad para construir interfaces de usuario dinámicas y reactivas.

Bases de Datos:

- **PostgreSQL**: Base de datos relacional, robusta, escalable.
- **MongoDB**: Debido a su flexibilidad en el manejo de datos no estructurados y su capacidad para escalar horizontalmente.

La combinación de estás tecnologías permiten desarrollar la aplicación de forma sencilla y eficiente, además nos proporcionan la flexibilidad necesaria para manejar tanto datos estructurados como no estructurados.

OPERACIONES CRUD

Las operaciones crud en la Base de Datos Relacional y en Base de Datos NoSQL:

- Crear un Registro
- Obtener Registros
- Actualizar un Registro
- Eliminar un Registro

Las diferencias clave en el manejo de base de datos relacionales y NoSQL están en su:

- Estructura de Datos: En las bases de datos relacionales, los datos se almacenan en tablas con un esquema fijo, mientras que en las bases de datos NoSQL, los datos se almacenan en documentos JSON con un esquema flexible.
- Consultas: Las bases de datos relacionales utilizan SQL para realizar consultas, mientras que las bases de datos NoSQL utilizan consultas específicas del motor de base de datos (MongoDB en este caso).
- Escalabilidad: Las bases de datos NoSQL están diseñadas para escalar horizontalmente, mientras que las bases de datos relacionales suelen escalar verticalmente.

Create:

- **Base de Datos Relacional (SQL)**: Para la base de datos SQL, la operación de creación se realiza utilizando **Sequelize**, que es un ORM (Object-Relational Mapping) que nos permite interactuar con la base de datos mediante objetos JavaScript. Cuando se recibe una solicitud `POST` en la ruta `/sql/create`, los datos proporcionados por el usuario se envían a la base de datos usando el método

`Item.create()`. Este método crea un nuevo registro en la tabla `users` y guarda los datos en las columnas correspondientes, como `name`, `lastName`, `email`, etc. Además, Sequelize se encarga de las validaciones de datos como la unicidad de los campos `email` y `username`.

```
app.post('/sql/create', async (req, res) => {
  try {
    console.log("sql/create Received: ");
    console.log(req.body);
    const item = await Item.create(req.body);
    console.log("Item created: ", item);
    res.status(201).json(item);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

- **Base de Datos NoSQL**: En el caso de la base de datos NoSQL, utilizamos MongoDB y el controlador oficial de MongoDB en Node.js. Para crear un nuevo documento en la colección `items`, la operación se realiza mediante el método `insertOne()`. En este caso, los datos del cuerpo de la solicitud se insertan directamente en la colección sin las restricciones de tipo de dato o la normalización que se encuentran en las bases de datos relacionales. Esto permite una mayor flexibilidad, pero puede requerir más validaciones a nivel de la aplicación.

```
app.post('/nosql/create', async (req, res) => {
  try {
    console.log("nosql/create Received: ");
    console.log(req.body);
    const newItem = new MongoItem(req.body);
    await newItem.save();
    console.log("Item created in MongoDB: ", newItem);
    res.status(201).json(newItem);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

Read:

- **Base de Datos Relacional (SQL)**: La operación de lectura en la base de datos relacional se realiza con **Sequelize** usando el método `findAll()`, que recupera todos los registros de la tabla `users`. Además, podemos utilizar el método `findByPk()` para buscar un registro específico por su clave primaria (`id`). Los resultados de la consulta son devueltos en formato JSON. A diferencia de NoSQL, las consultas en SQL son estructuradas y permiten filtrar, ordenar y limitar los resultados mediante comandos SQL.

```
app.get('/sql/getAll', async (req, res) => {
  try {
    console.log("sql/getAll received: ");
    console.log(req.body);
    const items = await Item.findAll();
    console.log("Items: ", items);
    res.status(200).json(items);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

- **Base de Datos NoSQL**: En MongoDB, la operación de lectura se realiza usando el método `find()`, que devuelve todos los documentos de la colección `items`. Al igual que SQL, MongoDB permite aplicar filtros, pero con un enfoque más flexible y sin la necesidad de definir previamente un esquema de la base de datos.

```
app.get('/nosql/getAll', async (req, res) => {
  try {
    console.log("nosql/getAll received: ");
    const items = await MongoItem.find();
    console.log("MongoDB Items: ", items);
    res.status(200).json(items);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

Update:

- **Base de Datos Relacional (SQL)**: En la base de datos relacional, el proceso de actualización se realiza con **Sequelize** utilizando el método `update()`. Este método permite actualizar uno o varios campos de un registro específico utilizando su clave primaria (`id`). Si el registro no se encuentra, se responde con un error 404. Esta operación es estructurada y se realiza en una transacción controlada que garantiza la integridad de los datos.

```
app.put('/sql/update/:id', async (req, res) => {
  try {
    console.log("sql/update Received: ");
    console.log(req.body);
    const item = await Item.findByPk(req.params.id);
    if (!item) {
      return res.status(404).json({ error: 'Item not found' });
    }
    await item.update(req.body);
    res.status(200).json(item);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

- **Base de Datos NoSQL**: En MongoDB, el proceso de actualización se realiza utilizando el método `updateOne()`, donde se especifica el filtro de búsqueda y los nuevos datos que se desean actualizar. MongoDB no requiere un esquema estricto, por lo que los documentos pueden tener diferentes estructuras, y se pueden actualizar de manera más flexible, pero esto también puede llevar a inconsistencia de datos si no se controla adecuadamente.

```
app.put('/nosql/update/:id', async (req, res) => {
  try {
    console.log("nosql/update Received: ");
    const item = await MongoItem.findById(req.params.id);
    if (!item) {
      return res.status(404).json({ error: 'Item not found' });
    }
    await item.updateOne(req.body);
    res.status(200).json(item);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

Delete:

- **Base de Datos Relacional (SQL)**: En la base de datos SQL, la eliminación de un registro se realiza usando **Sequelize** con el método `destroy()`, que elimina un registro basado en su `id`. Al igual que con la actualización, si no se encuentra el registro, se responde con un error 404.

```
app.delete('/sql/delete/:id', async (req, res) => {
  try {
    console.log("sql/delete Received: ");
    console.log(req.body);
    const item = await Item.findByPk(req.params.id);
    if (!item) {
      return res.status(404).json({ error: 'Item not found' });
    }
    await item.destroy();
    res.status(204).send();
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

- **Base de Datos NoSQL**: En MongoDB, la eliminación de un documento se realiza con el método `deleteOne()`, que elimina un documento específico según su `id`. Al igual que en SQL, si no se encuentra el documento, se responde con un error.

```
app.delete('/nosql/delete/:id', async (req, res) => {
  try {
    console.log("nosql/delete Received: ");
    const item = await MongoItem.findById(req.params.id);
    if (!item) {
      return res.status(404).json({ error: 'Item not found' });
    }
    await item.remove();
    res.status(204).send();
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

BASE DE DATOS RELACIONALES Y NoSQL

Las bases de datos relacionales (como MySQL y PostgreSQL) se caracterizan por su estructura organizada y la capacidad de realizar consultas complejas. Es ideal para aplicaciones que requieren un alto nivel de integridad y relaciones bien definidas entre los datos. Sin embargo, presentan algunas desventajas, como una escalabilidad limitada y una rigidez en su esquema, lo que dificulta los cambios en la estructura de los datos una vez que están definidos.

Por otro lado, las bases de datos NoSQL (como MongoDB y Firebase) ofrecen una mayor flexibilidad, ya que no requieren un esquema fijo, lo que permite almacenar datos con estructuras variadas. Son también altamente escalables, especialmente en escenarios de crecimiento horizontal, y ofrecen un buen rendimiento al manejar grandes volúmenes de datos. Sin embargo, sus desventajas incluyen una menor consistencia y la falta de un control transaccional tan riguroso como el de las bases de datos relacionales.

Las bases de datos relacionales son más adecuadas cuando los datos están fuertemente estructurados y se requiere mantener relaciones complejas entre ellos. En cambio, las bases de datos NoSQL son más apropiadas para aplicaciones que gestionan grandes volúmenes de datos no estructurados, como redes sociales o plataformas que requieren escalabilidad masiva y flexibilidad en la estructura de los datos.

CONEXIÓN CON LAS BASES DE DATOS

NoSQL

1. Conexión de la Aplicación a la Base de Datos NoSQL

Para establecer la conexión entre la aplicación y la base de datos MongoDB, se siguieron los siguientes pasos:

a. Instalación de MongoDB y Dependencias

La aplicación utiliza MongoDB como sistema de gestión de base de datos NoSQL. En primer lugar, se instaló **MongoDB** en el servidor local y se aseguraron de que todas las dependencias necesarias, como el cliente de MongoDB para Node.js (`mongoose`), estuvieran disponibles. Para ello, se ejecutó el siguiente comando para instalar `mongoose`:

```
PS C:\Users\ASUS\Downloads\BDD_tp_grupal\backendNodeJs> npm mongoose
```

`mongoose` es un ODM (Object Data Modeling) para MongoDB y Node.js que facilita la interacción con la base de datos a través de esquemas y modelos.

2. Configuración del Archivo de Conexión

En el archivo de configuración, se importaron las dependencias necesarias, incluyendo `mongoose` y las variables de entorno. En el archivo `config/database.js`, se configuró la cadena de conexión de MongoDB utilizando la URL de conexión proporcionada.

```
const mongoose = require('mongoose');
const dotenv = require('dotenv');

// Cargar las variables de entorno desde un archivo .env
dotenv.config();

// Conexión a la base de datos MongoDB usando Mongoose
Qodo Gen: Options | Test this function
const connectDB = async () => {
  try {
    // Cadena de conexión obtenida de las variables de entorno
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('MongoDB connected successfully.');
  } catch (error) {
    console.error('Unable to connect to MongoDB:', error);
    process.exit(1);
  }
};

module.exports = connectDB;
```

En este código:

- **dotenv.config()** se usa para cargar las variables de entorno, como la URL de la base de datos, desde un archivo `.env`.
- **mongoose.connect()** establece la conexión a la base de datos, pasando como parámetro la URL de MongoDB y algunas opciones de configuración.

2. Configuración del Archivo `.env`

Para mantener las credenciales y configuraciones seguras, la URL de la base de datos MongoDB se guardó en el archivo `.env` de la aplicación. Ejemplo de un archivo `.env`:

```
MONGO_URI=mongodb://localhost:27017/tu_base_de_datos
```

4. Conexión en la Aplicación Principal

Una vez que el archivo de configuración está listo, la función `connectDB()` se importa en el archivo principal de la aplicación (por ejemplo, `index.js` o `app.js`) y se ejecuta al iniciar el servidor, lo que asegura que la conexión a la base de datos esté activa antes de que la aplicación comience a manejar solicitudes.

```
const { connectDB, sequelize } = require('./config/database');

connectDB();

Tabnine | Edit | Test | Explain | Document | Ask
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

5. Manejo de Errores en la Conexión

La función de conexión a la base de datos está envuelta en un bloque `try-catch`, lo que permite manejar errores durante el proceso de conexión. Si la conexión a MongoDB falla, se captura el error y se muestra un mensaje en la consola, y el proceso de la aplicación se detiene utilizando `process.exit(1)` para evitar que la aplicación se ejecute sin una conexión establecida.

6. Verificación de Conexión

Una vez que la conexión se establece correctamente, se muestra un mensaje de confirmación en la consola: "Conexión a MongoDB establecida". Esto asegura que el sistema está listo para interactuar con la base de datos.

SQL

1. Conexión de la Aplicación a la Base de Datos PostgreSQL

Para conectar la aplicación a la base de datos PostgreSQL, se siguieron los siguientes pasos:

a. Instalación de Dependencias

La aplicación utiliza **PostgreSQL** como sistema de gestión de base de datos relacional. Para interactuar con PostgreSQL desde Node.js, se utiliza el paquete `pg` (PostgreSQL client for Node.js). Para instalarlo, se ejecutó el siguiente comando:

```
PS C:\Users\ASUS\Downloads\BDD_tp_grupal> npm install pg
```

Este paquete proporciona las herramientas necesarias para ejecutar consultas SQL y manejar la interacción con la base de datos PostgreSQL.

2. Configuración del Archivo de Conexión

En la aplicación, se creó un archivo de configuración (por ejemplo, `config/database.js`) que maneja la conexión con la base de datos PostgreSQL. Se utilizaron las credenciales de la base de datos que se almacenan en un archivo de entorno `.env` para asegurar que la configuración esté protegida.

Aquí tienes un ejemplo del código para conectar la aplicación con PostgreSQL usando el paquete `pg`:

```
// config/database.js
const { Sequelize } = require('sequelize');
const dotenv = require('dotenv');

// Load environment variables from .env file
dotenv.config();

const sequelize = new Sequelize(process.env.DATABASE_URL, {
  dialect: 'postgres',
  logging: false,
});

Qodo Gen: Options | Test this function
const connectDB = async () => {
  try {
    await sequelize.authenticate();
    console.log('Connection has been established successfully.');
  } catch (error) {
    console.error('Unable to connect to the database:', error);
  }
};

module.exports = { sequelize, connectDB };
```

En este código:

- **dotenv.config()** carga las variables de entorno desde el archivo `.env`.
- **new Sequelize()** establece una conexión con PostgreSQL utilizando Sequelize, especificando las credenciales y el host desde el archivo `.env`.
- **sequelize.authenticate()** verifica que las credenciales son correctas y que la base de datos está accesible.
- **sequelize.sync()** sincroniza automáticamente los modelos con la base de datos, asegurándose de que las tablas coincidan con los modelos definidos.

3. Configuración del Archivo `.env`

Las credenciales necesarias para la conexión a la base de datos PostgreSQL, como el nombre de usuario, la contraseña y el host, se almacenan en el archivo `.env` de la aplicación para evitar exponer esta información en el código fuente. Aquí tienes un ejemplo del contenido del archivo `.env`:

```
DATABASE_URL=postgres://test_user:password@localhost:5432/test_db
```

4. Conexión en la Aplicación Principal

Una vez que el archivo de configuración está listo, la función `connectDB()` se importa en el archivo principal de la aplicación (por ejemplo, `index.js` o `app.js`) y se ejecuta al inicio de la aplicación para garantizar que la conexión a la base de datos esté activa antes de que se procesen las solicitudes.

```
const { connectDB, sequelize } = require('./config/database');

connectDB();

Tabnine | Edit | Test | Explain | Document | Ask
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

5. Manejo de Errores en la Conexión

Al igual que con MongoDB, la conexión con PostgreSQL también se maneja dentro de un bloque `try-catch`. Si la conexión falla, el error es capturado y se muestra un mensaje de error en la consola, y la aplicación se detiene usando `process.exit(1)` para evitar que la aplicación se ejecute sin conexión a la base de datos.

6. Verificación de Conexión

Cuando la conexión se realiza con éxito, se imprime un mensaje de confirmación en la consola: "Conexión a PostgreSQL establecida". Esto garantiza que la aplicación está lista para interactuar con la base de datos y manejar las operaciones relacionadas con los datos.

DIFICULTADES Y APRENDIZAJES

Durante el desarrollo del proyecto, enfrentamos varios desafíos significativos que nos permitieron aprender y mejorar nuestras habilidades. Uno de los principales desafíos fue la configuración y conexión de la base de datos relacional PostgreSQL con Node.js utilizando Sequelize. Para resolver este problema, utilizamos la librería dotenv para cargar las variables de entorno desde un archivo `.env`, lo que nos permitió manejar las credenciales de manera segura y eficiente. Configuramos Sequelize en el archivo database.js, lo que facilitó la conexión y la gestión de la base de datos.

La integración de una base de datos NoSQL (MongoDB) en la aplicación también presentó desafíos. Utilizamos la librería `mongoose` para manejar la conexión y las operaciones con MongoDB. La configuración se realizó en el archivo database.js y las operaciones CRUD se implementaron en `models/Item.js`. Esto nos permitió manejar datos no estructurados de manera eficiente y aprovechar la flexibilidad de MongoDB.

La integración del frontend y backend fue otro desafío. Asegurar que ambos se comunicaran correctamente, especialmente al realizar solicitudes a la API, fue crucial. Utilizamos `axios` en el frontend para realizar solicitudes HTTP al backend y configuramos rutas en el archivo `index.js` para manejar estas solicitudes.

CONCLUSIÓN

La integración de PostgreSQL y MongoDB en la misma aplicación nos enseñó a manejar diferentes tipos de bases de datos y a utilizar las mejores prácticas para cada una. Esto nos permitió aprovechar las ventajas de ambas tecnologías.

El desarrollo de este proyecto nos permitió identificar áreas de mejora en nuestras habilidades técnicas. La combinación de bases de datos relacionales y NoSQL nos proporcionó la flexibilidad necesaria para manejar tanto datos estructurados como no estructurados, y nos permitió escalar la aplicación de manera eficiente.