Integración de Node.js y Express.js con bases de datos relacionales (MySQL) y no relacionales (MongoDB)

Julian F. Latorre

Índice

1. Introducción

Este instructivo trata sobre el proceso de integración de Node.js y Express.js con bases de datos relacionales (MySQL) y no relacionales (MongoDB). Aprenderemos sobre cómo establecer conexiones con bases de datos, realizar operaciones CRUD, implementar patrones de diseño, y aplicar mejores prácticas de seguridad y optimización.

2. Integración con MySQL

2.1. Instalación de Dependencias

Instala el driver de MySQL para Node.js:

```
npm install mysql2
```

mysql2 es un driver de MySQL para Node.js que proporciona una interfaz para interactuar con bases de datos MySQL.

2.2. Configuración de la Conexión

Crea un archivo `db.js` en la raíz de tu proyecto:

```
const mysql = require('mysql2');

const pool = mysql.createPool({
  host: 'localhost',
  user: 'tu_usuario',
  password: 'tu_contraseña',
  database: 'tu_base_de_datos',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});

module.exports = pool.promise();
```

- Importamos el módulo mysql2.
- Creamos un pool de conexiones, que es más eficiente que crear conexiones individuales para cada consulta.
- Configuramos el pool con los detalles de conexión a la base de datos.
- Exportamos el pool con la interfaz de promesas para facilitar el uso de async/await.

2.3. Implementación de Operaciones CRUD

Crea un archivo `userModel.js`:

```
const db = require('./db');

class UserModel {
  static async getAllUsers() {
    const [rows] = await db.query('SELECT * FROM users');
    return rows;
  }

  static async getUserById(id) {
    const [rows] = await db.query('SELECT * FROM users WHERE id = ?
    ', [id]);
    return rows[0];
  }

  static async createUser(name, email) {
    const [result] = await db.query(
      'INSERT INTO users (name, email) VALUES (?, ?)',
      [name, email]
    );
    return result.insertId;
  }

  static async updateUser(id, name, email) {
    const [result] = await db.query(
      'UPDATE users SET name = ?, email = ? WHERE id = ?',
      [name, email, id]
    );
    return result.affectedRows;
  }

  static async deleteUser(id) {
    const [result] = await db.query('DELETE FROM users WHERE id = ?
    ', [id]);
    return result.affectedRows;
  }
}

module.exports = UserModel;
```

- Creamos una clase UserModel que encapsula todas las operaciones relacionadas con los usuarios.
- Cada método estático corresponde a una operación CRUD (Create, Read, Update, Delete).
- Utilizamos consultas parametrizadas para prevenir inyecciones SQL.
- Usamos async/await para manejar las promesas retornadas por el pool de conexiones.

2.4. Ejemplo de Uso en Express.js

En tu archivo principal `app.js`:

```
const express = require('express');
const UserModel = require('./userModel');

const app = express();
app.use(express.json());

app.get('/users', async (req, res) => {
  try {
    const users = await UserModel.getAllUsers();
    res.json(users);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.get('/users/:id', async (req, res) => {
  try {
    const user = await UserModel.getUserById(req.params.id);
    if (user) {
      res.json(user);
    } else {
      res.status(404).json({ message: 'Usuario no encontrado' });
    }
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.post('/users', async (req, res) => {
  try {
    const { name, email } = req.body;
    const userId = await UserModel.createUser(name, email);
    res.status(201).json({ id: userId, name, email });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.listen(3000, () => console.log('Servidor corriendo en el puerto
    3000'));
```

- Creamos una aplicación Express y configuramos el middleware para parsear JSON.
- Implementamos rutas para obtener todos los usuarios, obtener un usuario por ID y crear un nuevo usuario.
- Utilizamos try/catch para manejar errores y enviar respuestas apropiadas.

3. Integración con MongoDB

3.1. Instalación de Dependencias

Instala el driver oficial de MongoDB para Node.js:

```
npm install mongodb
```

3.2. Configuración de la Conexión

Crea un archivo `mongodb.js`:

```
const { MongoClient } = require('mongodb');

const url = 'mongodb://localhost:27017';
const dbName = 'tu_base_de_datos';

let db = null;

async function connect() {
  if (db) return db;

  const client = new MongoClient(url, { useUnifiedTopology: true })
    ;
  await client.connect();
  db = client.db(dbName);
  return db;
}

module.exports = { connect };
```

- Importamos MongoClient del módulo mongodb.
- Definimos la URL de conexión y el nombre de la base de datos.
- Implementamos una función connect que establece la conexión si no existe y la reutiliza si ya está establecida.

3.3. Implementación de Operaciones CRUD

Crea un archivo `userModelMongo.js`:

```
const { connect } = require('./mongodb');

class UserModelMongo {
  static async getAllUsers() {
    const db = await connect();
    return db.collection('users').find().toArray();
  }

  static async getUserById(id) {
    const db = await connect();
    return db.collection('users').findOne({ _id: id });
  }

  static async createUser(user) {
    const db = await connect();
    const result = await db.collection('users').insertOne(user);
    return result.insertedId;
  }
```

```
  static async updateUser(id, updateData) {
    const db = await connect();
    const result = await db.collection('users').updateOne(
      { _id: id },
      { $set: updateData }
    );
    return result.modifiedCount;
  }

  static async deleteUser(id) {
    const db = await connect();
    const result = await db.collection('users').deleteOne({ _id: id
     });
    return result.deletedCount;
  }
}

module.exports = UserModelMongo;
```

- Implementamos métodos similares a los de MySQL, pero utilizando las operaciones específicas de MongoDB.
- Utilizamos la función connect para obtener la conexión a la base de datos en cada operación.

4. Patrones de Diseño y Mejores Prácticas

4.1. Patrón Repositorio

El patrón repositorio ya está implementado en los ejemplos anteriores con las clases `UserModel` y `UserModelMongo`. Estas clases actúan como una capa de abstracción entre la lógica de negocio y el acceso a datos.

4.2. Manejo de Errores

Implementa un middleware para manejar errores de forma centralizada:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    message: 'Ocurrió un error en el servidor',
    error: process.env.NODE_ENV === 'production' ? {} : err
  });
});
```

- Este middleware captura errores no manejados en la aplicación.
- Registra el error en la consola y envía una respuesta JSON con un mensaje de error genérico.
- En producción, no se envían detalles del error al cliente por razones de seguridad.

5. Optimización y Escalabilidad

5.1. Implementación de Caché

Para implementar caché, puedes usar una base de datos en memoria como Redis. Aquí un ejemplo conceptual:

```
const cache = new Map();

app.get('/users/:id', async (req, res) => {
  const id = req.params.id;

  if (cache.has(id)) {
    return res.json(cache.get(id));
  }

  const user = await UserModel.getUserById(id);
  if (user) {
    cache.set(id, user);
    res.json(user);
  } else {
    res.status(404).json({ message: 'Usuario no encontrado' });
  }
});
```

- Usamos un Map como caché en memoria simple.
- Verificamos si el usuario está en caché antes de consultar la base de datos.
- Si el usuario se encuentra en la base de datos, lo almacenamos en caché para futuras consultas.

5.2. Paginación

Implementa paginación para manejar grandes conjuntos de datos:

```
// En userModel.js
class UserModel {
  // ... otros métodos ...

  static async getAllUsers(skip = 0, limit = 10) {
    const [rows] = await db.query('SELECT * FROM users LIMIT ?
    OFFSET ?', [limit, skip]);
    return rows;
  }

  static async countUsers() {
    const [result] = await db.query('SELECT COUNT(*) as total FROM
    users');
    return result[0].total;
  }
}
```

Ahora, la ruta de Express que utiliza estos métodos:

```
app.get('/users', async (req, res) => {
  try {
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const skip = (page - 1) * limit;

    const users = await UserModel.getAllUsers(skip, limit);
    const total = await UserModel.countUsers();

    res.json({
      users,
      currentPage: page,
      totalPages: Math.ceil(total / limit),
      totalUsers: total
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

- Implementamos paginación utilizando los parámetros page y limit de la consulta.
- Calculamos el número de elementos a omitir (skip) basado en la página actual.
- Devolvemos los usuarios de la página actual junto con metadatos de paginación.

6. Seguridad

6.1. Uso de Variables de Entorno

Utiliza variables de entorno para manejar información sensible:

```
// Asegúrate de tener las variables de entorno configuradas
const pool = mysql.createPool({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASS,
  database: process.env.DB_NAME,
  // ...
});
```

- Utilizamos process.env para acceder a las variables de entorno.
- Esto permite mantener la información sensible fuera del código fuente.

6.2. Prevención de Inyección SQL

Ya implementado en los ejemplos anteriores mediante el uso de consultas parametrizadas.

6.3. Implementación de CORS

```
npm install cors
```

```
const cors = require('cors');
app.use(cors());
```