Instituto Politécnico Nacional

Omar Montoya Romero

7CM1

WEB CLIENT AND BACKEND DEVELOPMENT FRAMEWORKS

Sistemas Computacionales

Sequelize, Prisma, Mongoose

Mtro. Efraín Arredondo Morales

06/10/2023

Sequelize

Sequelize es un ORM para Node.js que se utiliza para interactuar con bases de datos relacionales, como MySQL, PostgreSQL, SQLite, y MSSQL, utilizando objetos y JavaScript en lugar de consultas SQL directas. Sequelize fue diseñado para ser utilizado en entornos de Node.js y es ampliamente utilizado en aplicaciones web y proyectos de backend escritos en este lenguaje.

Sequelize a su vez, facilita la definición de modelo de datos, la validación, las relaciones e implementación de lógica de negocio en aplicaciones Nodejs. [1]

Características:

- Soporta varias bases de datos: MySQL, SQLite, PostgreSQL, …
- Definición de esquemas.
- Sincronización con la base de datos,
- Validaciones
- Métodos CRUD
- Relaciones 1-a-1, 1-a-N, N-a-N.
- Migraciones [2]

Ejemplo:

Instalación de nuestras dependencias

Para la instalación de nuestras dependencias ejecutar los siguientes comandos:

```
> mkdir api-authors-books
> cd api-authors-books
> npm -y init
> npm install express pg pg-hstore sequelize --save
```

Posterior a nuestras dependencias, crearemos un archivo base llamado index.js, que incluirá el siguiente código inicial.

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello world!');
});
app.listen(port, () => {
  console.log(`Server listen on http://localhost:${port}`);
});
```

Sí ejecutamos la instrucción node index.js en nuestra terminal, deberíamos poder visualizar nuestro servidor escuchando en el puerto 3000.

Para comenzar crearemos un fichero llamado connection.js en la raíz de nuestro proyecto para implementar nuestra conexión a PostgreSQL.

```
const { Sequelize } = require('sequelize');

const database = "test";
const username = "postgres";
const password = "";
const host = "localhost";

const sequelize = new Sequelize(database, username, password, {
  host: host,
  dialect: 'postgres',
});

module.exports = {
  sequelize
}
```

Implementando nuestros modelos

Posterior, crearemos un fichero llamado `models.js` en la raíz de nuestro proyecto para implementar nuestros modelos.

En este caso serían 2. El modelo de autor y el modelo de libros, en conjunto de sus relaciones 1:N.

```
const { DataTypes } = require('sequelize');
const { sequelize } = require('./connection');

const Author = sequelize.define('Author', {
  name: {
    type: DataTypes.STRING,
    allowNull: false
  },
  age: {
    type: DataTypes.INTEGER,
    allowNull: false
  },
  createdAt: {
    type: DataTypes.DATE,
    defaultValue: DataTypes.NOW
  }
}, { tableName: 'authors' });

const Book = sequelize.define('Book', {

  isbn: {
    type: DataTypes.STRING,
    allowNull: false
```

```
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false
    },
    cantPages: {
      type: DataTypes.INTEGER,
      allowNull: false,
      defaultValue: 0
    },
    createdAt: {
      type: DataTypes.DATE,
      defaultValue: DataTypes.NOW
    }
}, { tableName: 'books' });

Author.hasMany(Book, { as: 'books', foreignKey: 'authorId' });
Book.belongsTo(Author, {
  foreignKey: "authorId",
});

module.exports = {
  Author,
  Book
};
```

Implementando nuestras rutas y conexión a PostgreSQL
Posteriormente implementaremos nuestras rutas en el archivo index.js . Crearemos 4 endpoints referentes.

GET /authors — Obtención de todos los autores.
POST /authors — Creación de un nuevo autor.
GET /books — Obtención de todos los libros.
POST /books — Creación de un nuevo libro.

```
const express = require('express');
const app = express();
const port = 3000;
const { sequelize } = require('./connection');
const { Author, Book } = require('./models');

app.use(express.json());

app.get('/', (req, res) => {
  res.send('Hello world!');
});

app.get('/authors', async (req, res) => {
  try {
    const authors = await Author.findAll({
```

```
      include: [
        {
          model: Book,
          as: 'books',
          attributes: ['id', 'isbn', 'name', 'cantPages', 'createdAt'],
        },
      ],
    })
    return res.json({ authors });
  } catch (error) {
    console.log('Error', error);
    return res.status(500).json({ message: 'Internal server error' });
  }
});

app.post('/authors', async (req, res) => {
  try {
    const name = req.body?.name;
    const age = req.body?.age;

    if (!name || !age) {
      return res.status(400).json({ message: 'Bad request, name or age not found'
});
    }
    const save = await Author.create({
      name,
      age
    });
    return res.status(201).json({ author: save });
  } catch (error) {
    console.log('Error', error);
    return res.status(500).json({ message: 'Internal server error' });
  }
});

app.get('/books', async (req, res) => {
  try {
    const books = await Book.findAll()
    return res.json({ books });
  } catch (error) {
    console.log('Error', error);
    return res.status(500).json({ message: 'Internal server error' });
  }
});

app.post('/books', async (req, res) => {
  try {
    const isbn = req.body?.isbn;
    const name = req.body?.name;
    const cantPages = req.body?.cantPages;
    const author = req.body?.author;

    if (!name || !cantPages || !author || !isbn) {
      return res.status(400).json({ message: 'Bad request, isbn or name or
cantPages or author not found' });
    }
```

```
    const save = await Book.create({
      isbn,
      name,
      cantPages,
      authorId: author
    })
    return res.status(201).json({ book: save });
  } catch (error) {
    console.log('Error', error);
    return res.status(500).json({ message: 'Internal server error' });
  }
});

sequelize
  .authenticate()
  .then(() => {
    console.log('Connection success');
    return sequelize.sync();
  })
  .then(() => {
    console.log('Sync models');
    app.listen(port, () => {
      console.log(`Server listen on http://localhost:${port}`);
    });
  })
  .catch((error) => {
    console.error('Connection fail', error);
  });
```

Probando nuestra solución con curl

Creación de un autor.

```
// Crear autor

curl --location 'http://localhost:3000/authors' \
--header 'Content-Type: application/json' \
--data '{
    "name": "Diego",
    "age": 27
}'

// Respuesta 201

{
    "author": {
        "createdAt": "2023-08-04T21:31:12.659Z",
        "id": 1,
        "name": "Diego",
        "age": 27,
        "updatedAt": "2023-08-04T21:31:12.660Z"
    }
}
```

[1]

Prisma

El esquema Prisma es intuitivo y le permite declarar las tablas de su base de datos de una manera legible por humanos, lo que hace que su experiencia de modelado de datos sea un placer. Usted define sus modelos a mano o los examina a partir de una base de datos existente.

Normalmente se denomina esquema.prisma y consta de las siguientes partes:

- Fuentes de datos: especifique los detalles de las fuentes de datos a las que Prisma debe conectarse (por ejemplo, una base de datos PostgreSQL).
- Generadores: especifica qué clientes deben generarse en función del modelo de datos (por ejemplo, Prisma Client)
- Definición del modelo de datos: especifica los modelos de su aplicación (la forma de los datos por fuente de datos) y sus relaciones.
- Consulte la referencia de la API del esquema Prisma para obtener información detallada sobre cada sección del esquema.

Siempre que se invoca un comando de prisma, la CLI normalmente lee cierta información del archivo de esquema, por ejemplo:

- prisma generate: lee toda la información mencionada anteriormente del esquema Prisma para generar el código de cliente fuente de datos correcto (por ejemplo, Prisma Client).
- prisma migrar dev: lee las fuentes de datos y la definición del modelo de datos para crear una nueva migración.
- También puede utilizar variables de entorno dentro del archivo de esquema para proporcionar opciones de configuración cuando se invoca un comando CLI.

Ejemplo:

```
datasource db {
  provider = "mongodb"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model User {
  id        String   @id @default(auto()) @map("_id") @db.ObjectId
  createdAt DateTime @default(now())
  email     String   @unique
  name      String?
  role      Role     @default(USER)
```

```
  posts     Post[]
}

model Post {
  id        String   @id @default(auto()) @map("_id") @db.ObjectId
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  published Boolean  @default(false)
  title     String
  author    User     @relation(fields: [authorId], references: [id])
  authorId  String   @db.ObjectId
}

enum Role {
  USER
  ADMIN
}
```

[3]

Mongoose

Definiendo su esquema

Todo en Mongoose comienza con un esquema. Cada esquema se asigna a una colección de MongoDB y define la forma de los documentos dentro de esa colección.

```
import mongoose from 'mongoose';
const { Schema } = mongoose;


const blogSchema = new Schema({
  title: String, // String is shorthand for {type: String}
  author: String,
  body: String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs: Number
  }
});
```

Mongoose solo crea rutas de esquema reales para las hojas del árbol. (como meta.votes y meta.favs arriba), y las ramas no tienen rutas reales. Un efecto secundario de esto es que

el meta anterior no puede tener su propia validación. Si se necesita validación en el árbol, se debe crear una ruta en el árbol; consulte la sección Subdocumentos para obtener más información sobre cómo hacerlo. Lea también la subsección Mixta de la guía SchemaTypes para conocer algunos errores.

Los SchemaTypes permitidos son:

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array
- Decimal128
- Map
- UUID

Los esquemas no solo definen la estructura de su documento y la conversión de propiedades, sino que también definen métodos de instancia de documento, métodos de modelo estáticos, índices compuestos y enlaces del ciclo de vida del documento llamados middleware. [4]

Referencias

[1] D. Coder, «Conexión a una base de datos PostgreSQL con Node.js y Sequelize,» medium, [En línea]. Available: https://medium.com/@diego.coder/conexión-a-una-base-de-datos-postgreql-con-node-js-y-sequelize-d93b0546e4cc. [Último acceso: 05 10 2023].

[2] S. Pavon, «Sequize.key,» 28 02 2019. [En línea]. Available: https://www.dit.upm.es/~santiago/docencia/grado/core/Sequelize.pdf. [Último acceso: 05 10 2023].

[3] Prisma, «Prisma schema (Reference),» Prisma, [En línea]. Available: https://www.prisma.io/docs/concepts/components/prisma-schema. [Último acceso: 05 10 2023].

[4] Mongoose, «Mongoose v7.6.0: Schemas,» Mongoose ODM v7.6.0, [En línea]. Available: https://mongoosejs.com/docs/guide.html. [Último acceso: 05 10 2023].