Taller NodeJS – Express: Conexión a base de datos

Instructor: Abner Saavedra
Fecha: sept. de 2024
Email: ingenieroabnersaavedra@gmail.com
GracoSoft Centro Empresarial Plaza Madrid, piso 9, oficinas 9-7 a la 9-10

Agenda

- Introducción: Uso de ORM
- Preparación del entorno
- Conectando con la BBDD
- Creando el modelo
- Adaptando la lógica
- Conclusiones

Introducción: Uso de ORM

Para la conexión con las bases de datos existen los **ORM (Object Relational Mapping) Objeto de Mapeo Relacional**, que este nombre significa que creamos una representación de nuestras tablas, en el caso de base de datos relacionales, o documentos, en el caso de bbdd orientadas a documentos, y la convertimos en un objeto con el que podemos interactuar.

Con este objeto conseguimos una interacción mas integrada en la programación, ya que no es necesario usar otro lenguaje para las peticiones como puede ser SQL.

Para este proyecto vamos a usar uno de los ORM mas populares Sequelize.

Vas a necesitar tener instalado en local o acceso en alguna plataforma online un servidor de MySQL o MariaDB.

Preparación del entorno

Para centrarnos en el uso del ORM voy a usar un proyecto donde está implementada una API de Express que consume datos de dos arrays: users y posts, que migraremos a una base de datos MySQL.

Puedes usar cualquier proyecto que tengas con Express, pero si no tienes ninguno, este es el repositorio donde puedes descargar el que vamos a usar para el ejemplo: **API con Express** y así nos podemos centrar en implementar Sequelize.

Lo siguiente es clonar el repositorio de Github e instalar las dependencias con los siguiente comandos:

```
git clone https://github.com/raulalhena/medium-api-express.git
```

Accedemos al directorio del proyecto e instalamos las dependencias:

```
npm install
```

Una vez instaladas, vamos a necesitar instalar el paquete de Sequelize:

```
npm install sequelize
```

Ahora toca elegir el tipo de base de datos que vamos a usar, en el caso de que uses MySQL o MariaDB los paquetes son diferentes, tienes que elegir entre estos dos:

```
npm install mysql2
o
npm install mariadb
```

Con esto ya tenemos listo el entorno!

Conectando con la BBDD

Lo primero es configurar la conexión a la base de datos para poder realizar las peticiones, vamos a configurar Sequelize para que se conecte a nuestro servidor.

Creamos un directorio ***/src/db*** con el archivo ***db.config.js*** dentro:

Archivo: *src/db/db.config.js*

```
import { Sequelize }  from "sequelize";

export const sequelize = new Sequelize('bd_express_sequelize', 'root', 'root', {
  dialect: 'mysql',
  host: 'localhost'
});
```

La configuración básica es sencilla, tienes varias maneras de hacerlo con el constructor de ***Sequelize()***, aquí puedes ver que le pasamos, el nombre de la base de datos, el usuario, la contraseña del usuario y después un objeto con el dialecto, o sea con que base de datos se va a conectar y el host donde está el gestor de base de datos.

Para realizar la conexión necesitamos ejecutar la función ***authenticate()*** de la instancia de sequelize, esto lo haremos desde el archivo principal, antes de poner al servidor a escuchar, y usando el bloque try, catch, por si hay un error en la conexión, nuestro servidor siga funcionando.

```
import express from 'express';
import userRouter from './routes/users.js';
import postRouter from './routes/posts.js';
import { sequelize } from './db/db.config.js';

const app = express();

const PORT = 3000;

app.use(express.json());

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

// END POINT: users
app.use('/users', userRouter);

// END POINT: posts
app.use('/posts', postRouter);

try{
  await sequelize.authenticate();
  console.log('Connection with DB stablished');
} catch(error) {
  console.log('DB not connected', error);
}

export default app.listen(PORT | 3000, () => {
  console.log(`Server listening on port ${PORT}`);
});
```

Creando el modelo

Como ya he comentado al principio, al usar un ORM creamos una representación como objeto de nuestras tablas, así que le añadiremos las propiedades (variables) que corresponden a cada una de las columnas en MySQL.

En el proyecto tenemos dos entidades: **users** y **posts**, que corresponden a dos tablas con el mismo nombre en nuestra base de datos, vamos a crear los objetos con la definición de cada una de ellas.

Archivo: *src/users/entities/User.entity.js*

```
import { Sequelize } from "sequelize";
import { sequelize }  from "../../db/db.config.js";

export const User = sequelize.define('user',
  {
    id: {
      type: Sequelize.INTEGER,
      autoIncrement: true,
      primaryKey: true
    },
    name: {
      type: Sequelize.STRING,
      allowNull: false
    },
    email: {
      type: Sequelize.STRING,
      allowNull: false
```

```
    },
    createdAt: {
      type: 'DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL',
      defaultValue: () => new Date()
    },
    updatedAt: {
      type: 'DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP NOT NULL',
      defaultValue: () => new Date()
    }
  },
  {
    timestamps: false
  }
);
```

Lo primero que necesita este archivo es la clase Sequelize para sacar los tipos: **INTEGER, STRING**, etc y después la instancia de sequelize que hemos creado en el archivo ***db.config.js***.

La instancia la usamos para ejecutar la función ***define()*** con la que definimos la entidad '**user**' como primer argumento, y, con los campos (propiedades) que vamos a tener en la tabla users de la bbdd.

Las propiedades del objeto son:

id: Un entero definido como clave primaria y que se auto incrementa en 1 cada vez que añades un registro en la tabla.

name: Una cadena de caracteres que no permitimos que sea valor nulo.

email: Lo mismo que el campo name.

createdAt: Este campo se actualiza automáticamente por parte de la bbdd, indicando la fecha y hora cuando introducimos el registro en la tabla.

updatedAt: Es igual que el anterior pero se actualiza cada vez que realizamos un cambio en el registro de la tabla.

timestamp: Un booleano para indicar si queremos que sequelize se encargue de los timestamp en vez de la bbdd, está en false y por eso es MySQL quien los gestiona.

Archivo: *src/posts/entities/Post.entity.js*

```
import { Sequelize } from "sequelize";
import { sequelize }  from "../../db/db.config.js";

export const Post = sequelize.define('post',
  {
    id: {
      type: Sequelize.INTEGER,
      autoIncrement: true,
      primaryKey: true
    },
    title: {
      type: Sequelize.STRING,
      allowNull: false
    },
    content: {
      type: Sequelize.STRING,
      allowNull: false
    },
    userId: {
      type: Sequelize.INTEGER,
      allowNull: false
    },
    createdAt: {
      type: 'DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL',
      defaultValue: () => new Date()
    },
    updatedAt: {
      type: 'DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP NOT NULL',
      defaultValue: () => new Date()
    }
  },
  {
    timestamps: false
  }
);
```

Aquí puedes ver que hay poca diferencia con la entidad **User**, sólo los campos son los correspondientes a ***title*** y ***content***, y, encontramos un campo más, ***userId***. Esta propiedad es la que identifica al usuario que ha creado el **post** y en la bbdd se usa como clave foránea para relacionar las dos entidades.

Adaptando la lógica

El proyecto que usamos de ejemplo, API con Express, tiene tres elementos que intervienen el flujo de los datos, tanto las peticiones (requests) como las respuestas (response). Cada uno de ellos cumple una función especifica y responden a la arquitectura Modelo Vista Controlador (MVC).

- **Controller:** Es el encargado de recibir y gestionar el tipo de petición y devolver la respuesta al cliente.
- **Service**: Es el encargado de ejecutar la lógica necesaria para recuperar los datos, modificarlos y devolverlos al controller en la forma que necesaria.
- **Repository**: Es el encargado de conectar con la base de datos usando el modelo para recuperar los datos y devolverlos al service.

Al usar esta estructura con un elemento **repository**, podemos hacer una migración sencilla de usar arrays como hacíamos en el proyecto API con Express, a usar una base de datos, ya que no es necesario modificar toda la lógica del **service**, si no que, el service sigue usando las mismas funciones que facilita el **repository**, y es este el que ahora usa los métodos del modelo para interactuar con la bbdd.

Hay un cambio en la estructura, ya que antes el objeto **User** tenía los **posts** que había creado en un array, pero ahora es el ***id*** del user el que está en el objeto **post**, facilitando así la relación en la estructura de la base de datos.

Y por último tenemos que cambiar los métodos de los elementos para que sean asíncronos, ya que es necesario esperar a que la consulta a la base de datos devuelva los datos antes de continuar con la ejecución del resto del código.

Entidad User

Archivo: */src/users/controller.js*

```
import { userService } from './service.js';

// GET:

const findAll = async (req, res) => {
  res.status(200).json(await userService.findAll());
}

const findOneById = async (req, res) => {
  res.status(200).json(await userService.findOneById(+req.params.id));
}

// END GET

// POST:

const create = async (req, res) => {
  res.status(201).json(await userService.create(req.body));
}

// END POST

export const userController = {
  findAll,
  findOneById,
  create
}
```

Importamos el userService y definimos las tres funciones que vamos a exponer en los end points: ***findAll***, ***findOneById*** y ***create***.

Todas las funciones hacen lo mismo, de manera asíncrona llaman a una función con el mismo nombre del service en la función ***json()*** encadenada a la función ***status()*** del objeto response (res).

Y por último exportamos las funciones dentro del objeto ***userController*** para que se usen desde las routes que definen los end points.

Archivo: */src/users/service.js*

```
import { userRepository } from './repository.js';
import { postService } from '../posts/service.js';

const findAll = async () => {
    const users = await userRepository.findAll();
    return {
      users: users
    }
}

const findOneById = async (id) => {
  console.log('user id service ', id)
  const user = await userRepository.findOneById(id);
  const posts = await findUserPosts(id);
  return {
    id: user.id,
```

```
    name: user.name,
    email: user.email,
    posts: posts
  }
}

const findUserPosts = async (userId) => {
  const userPosts = await postService.findUserPosts(userId);
  return userPosts;
}

const create = async (userObj) => {
  const newUser = await userRepository.create(userObj);
  return {
    user: newUser
  }
}

export const userService = {
  findAll,
  findOneById,
  create
}
```

En este archivo tenemos las mismas funciones que en el controller y una mas: ***findUserPosts()***, que lo que hace es recuperar los posts del usuario que consultamos con la función ***findOneById()*** y así devuelve el usuario con todos los datos de los posts que ha creado.

findUserPosts() no se exporta en el objeto ***userService*** ya que es de uso interno del service.

Archivo: */src/users/repository.js*

```
import { User } from "./entities/User.entity.js";

const findAll = async () => {
    const users = await User.findAll();
    return users;
}

const findOneById = async (id) => {
  return await User.findOne({ where: { id: id }});
}

const create = async(user) => {
  const newUser = await User.create(user);
  return newUser;
}

export const userRepository ={
  findAll,
  findOneById,
  create
}
```

Para conectar con la bbdd necesitamos importar el modelo, en este caso **User**, que en él están integrados los métodos (funciones) que nos permiten interactuar con MySQL. Como ves usamos: ***User.findAll()***, **User.findOne()** y ***User.create()***.

Cada una tiene una finalidad, y por lo tanto necesita unos argumentos diferentes y devuelve unos valores diferentes.

findAll: Los argumentos no son necesarios, y nos devuelve un array con todos los registros de la tabla users, por lo tanto todos los usuarios guardados en la base de datos.

findOne: Nos permite filtrar que objeto queremos encontrar, puede ser por ***id*** (lo mas óptimo), ***name*** o ***email***, dependiendo de nuestras necesidades. Y devuelve un objeto con el usuario que ha encontrado.

create: El argumento es un objeto usuario "completo", quiere decir tenemos que pasarle los datos que podemos modificar, en nuestro caso: ***name*** y ***email***, ya que, ***id***, ***createdAt*** y ***updatedAt*** se encarga MySQL de gestionarlos. Devuelve un objeto con los datos del usuario que se acaba de guardar en la base de datos.

Y como en los otros elementos se exportan las funciones para que puedan ser usadas por el service.

Entidad Post

Archivo: *src/posts/controller.js*

```
import { postService } from './service.js';

const findAll = async (req, res) => {
  res.status(200).json(await postService.findAll());
}
```

```
const findOneById = async (req, res) => {
  res.status(200).json(await postService.findOneById(+req.params.id));
}

const create = async (req, res) => {
  res.status(201).json(await postService.create(req.body));
}

export const postController = {
  findAll,
  findOneById,
  create
}
```

En este archivo hay lo mismo que en el controller de **user**, únicamente que usa el service de **Post**.

Archivo: *src/posts/service.js*

```
import { postRepository } from './repository.js';

const findAll = async () => {
    const posts = await postRepository.findAll();
    return {
      posts: posts
    }
}

const findOneById = async (id) => {
  const post = await postRepository.findOneById(id);
```

```
  return {
    post: post
  }
}

const findUserPosts = async (userId) => {
  const userPosts = await postRepository.findUserPosts(userId);
  return userPosts;
}

const create = async (post) => {
  const newPost = await postRepository.create(post);
  return {
    post: newPost
  }
}

export const postService = {
  findAll,
  findOneById,
  create,
  findUserPosts
}
```

Al igual que el service de **User** tenemos las mismas funciones y una más, que es la de buscar los posts que pertenecen a un usuario concreto que exportamos para que lo pueda usar el service **User**, ya que como buena práctica es mejor que una entidad no acceda al modelo de la otra, si no que, pase por el service y sea este el que consulte con su modelo.

Archivo: */src/posts/repository.js*

```
import { Post } from "./entities/Post.entity.js";

const findAll = async () => {
    return await Post.findAll();
}

const findOneById = async (id) => {
  return await Post.findOne({ where: { id: id }});
}

const findUserPosts = async (userId) => {
  return await Post.findAll({ where: { userId: userId }});
}

const create = async (post) => {
  const newPost = await Post.create(post);
  return newPost;
}

export const postRepository ={
  findAll,
  findOneById,
  create,
  findUserPosts
}
```

Este repository contiene una función más que el de **User**, ***findUserPosts()*** y aquí puedes ver que usamos la función ***Post.findAll()*** pero pasándole argumentos, que va a filtrar los ***posts***, y sólo va a devolver en los que el campo ***userId*** coincida con el ***id*** de usuario que le enviamos, y los devuelve en forma de array.

Sincronizar sequelize

Vamos a modificar el archivo app.js que hemos visto al principio del artículo, para hacer uso de las propiedades de Sequelize es necesario indicarle que se sincronice con MySQL, esto lo hacemos desde el archivo de la aplicación principal.

Archivo: */src/app.js*

```
import express from 'express';
import userRouter from './routes/users.js';
import postRouter from './routes/posts.js';
import { sequelize } from './db/db.config.js';
import { User } from './users/entities/User.entity.js';
import { Post } from './posts/entities/Post.entity.js';

const app = express();

const PORT = 3000;

app.use(express.json());

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

```
});

// END POINT: users
app.use('/users', userRouter);

// END POINT: posts
app.use('/posts', postRouter);

// Configuración de la relación entre tablas y sincronización con la bbdd
try{
  User.hasMany(Post, {
    foreignKey: 'userId'
  });

  Post.belongsTo(User, {
    foreignKey: 'userId'
  });

  await sequelize.sync({ force: true });
  console.log('Connection with DB stablished');
} catch(error) {
  console.log('DB not connected', error);
}

// Iniciamos la escucha del servidor web
export default app.listen(PORT | 3000, () => {
  console.log(`Server listening on port ${PORT}`);
});
```

Lo importante de este archivo es la parte donde configuramos la relación que existe entre **Users** y **Posts**, que es **One to Many (Uno a Muchos)**, y lo hacemos con unos métodos que tienen los modelos:

User.hasMany(): Con este método que recibe, como primer argumento, el modelo al que está relacionado **Post** y como segundo argumento una serie de opciones, donde definimos cual es el campo que se usará para concretar la relación, y en nuestro caso es, ***userId***, que está en la tabla **posts** y por lo tanto es una propiedad del modelo **Post**.

Post.belongsTo(): La estructura es la misma, recibe el modelo al que pertenece y cómo segundo argumento las mismas opciones, la clave foránea userId.

Después de definir las relaciones ejecutamos la sincronización entre Sequelize y MySQL, **de forma asíncrona (await)**, y como argumento le podemos pasar la opción ***force***, que es un booleano que lo que hace es sobre escribir la estructura y datos de la base de datos con los definidos en el código, por lo tanto cuidado que cuando está en ***true***, **elimina los datos guardados**.

Conclusiones

Si ya tenías conocimientos de SQL y de cómo gestionar una base de datos, puedes ver que usando un ORM como Sequelize, nos abstrae del uso de otros lenguajes de programación que no sea JavaScript.

Este método nos facilita la interacción con la base de datos que puede hacerse muy tediosa en el momento que tenemos que usar peticiones SQL que debemos cambiar según lo que nos estén solicitando y también hace que por ejemplo ataques de inyección SQL ya no sean algo de lo que preocuparse.

Ejercicio

Construir una API utilizando Node.js y Express para gestionar productos. La API debe permitir realizar las operaciones básicas de creación, lectura, actualización y eliminación (CRUD) de productos

Requisitos:

1. **Creación de Producto:**
   - Implementa una ruta para agregar un nuevo producto a la base de datos.
   - El cuerpo de la solicitud (`body`) debe contener la información necesaria del producto, como nombre, precio y cantidad.
2. **Consulta de Producto:**
   - Implementa una ruta para consultar la información de un producto específico por su identificador único.
   - La información del producto debe incluir al menos el nombre, precio y cantidad.
3. **Modificación de Producto:**
   - Crea una ruta para actualizar la información de un producto existente.
   - El cuerpo de la solicitud debe contener los campos que se desean actualizar.
4. **Eliminación de Producto:**
   - Implementa una ruta para eliminar un producto según su identificador único.
5. **Filtrado de Información:**
   - Proporciona la capacidad de filtrar la información de los productos según ciertos criterios (por ejemplo, por precio o cantidad).
   - Los criterios de filtrado deben ser parámetros de la URL.

Notas Adicionales:

- Utiliza Express para crear el servidor web.
- Utiliza el módulo Express Router para crear las rutas de acceso e implementa controladores para modularizar el código.
- Usa Squelize para abstraer los modelos de base de datos y guardar los datos de productos.

GRACOSOFT ES EXCELENCIA EDUCATIVA