Clase 5

Webservices con express

Temario

1. Qué es Express
2. Instalación
3. Creación de servidores
4. Middlewares
5. Enrutamiento
6. Knex/**Sequelize**
7. Sesiones, Cookies y JWT

Knex/Sequelize

El manejo de datos en una aplicación es muy importante, por ello, en esta sección aprenderemos a utilizar el ORM Sequelize y el constructor de consultas Knex (query builder).

- Qué es un ORM
- Qué es Knex y cómo se usa
- Qué es **Sequelize** y cómo se usa
- Servidor JSON con base de datos postgres

¿Qué es un ORM?

Un ORM (Object Relational Mappings) es un modelo de programación que permite mapear las estructuras de una base de datos relacional (SQL Server, Oracle, MySQL, etc.), sobre una estructura lógica de entidades con el objeto de simplificar y acelerar el desarrollo de nuestras aplicaciones.

Las estructuras de la base de datos relacional quedan vinculadas con las entidades lógicas o base de datos virtual definida en el ORM, de tal modo que las acciones CRUD (Create, Read, Update, Delete) a ejecutar sobre la base de datos física se realizan de forma indirecta por medio del ORM.

Ventajas:

- Evitar el uso de código redundante.
- No necesitamos escribir código SQL, nos abstraemos del motor de base de datos usado.
- Cada lenguaje de programación tiene sus propios ORMs con soporte a varias bases de datos.
- Se puede cambiar de motor de base de datos fácilmente.
- Los ORMs más completos ofrecen servicios para persistir todos los cambios en los estados de las entidades, previo seguimiento o tracking automático, sin escribir una sola línea de SQL.
- Te permite centrarte más en la lógica de negocio y no en la escritura de interfaces.

Uso:

- Proyectos grandes o complejos.

¿Qué es Sequelize?

Sequelize es un ORM que permite a los usuarios llamar a funciones javascript para interactuar con SQL DB sin escribir consultas reales. Es bastante útil para acelerar el tiempo de desarrollo.

Uso:

- Proyectos más complejos

Configurando Sequelize 1/3

1. Instalamos Sequelize y el cliente de Postgres

```
npm install sequelize pg pg-hstore --save
```

2. Instalamos Sequelize de manera global para usar algunos comandos desde la consola

```
sudo npm install sequelize-cli -g
```

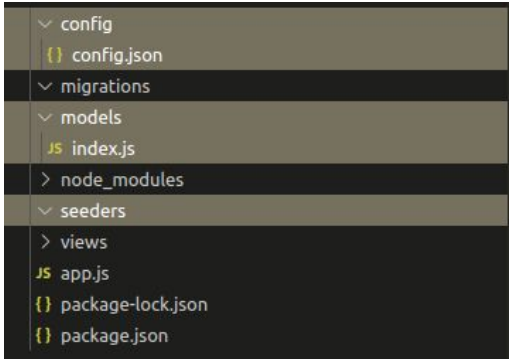
3. Iniciamos nuestro servicio de base de datos

```
npm run start-services
```

4. Iniciamos un proyecto de Sequelize

```
sequelize init
```

5. Editamos el archivo config/config.js y configuramos nuestra base de datos




```
// config/config.json
{
 "development": {
   "username": "postgres",
   "password": "1234",
   "database": "express-db-seq",
   "host": "127.0.0.1",
   "dialect": "postgres"
 }
}
```

Configurando Sequelize 2/3

6. Creamos nuestro modelo

```
sequelize model:generate --name students --attributes
name:string,last_name:string,date_of_birth:date
```

7. Observamos que ha creado un modelo student.js y una migración [TIMESTAMP]-create-students.js

8. Ejecutamos la migración

```
sequelize db:migrate
```

9. Probamos a eliminar la tabla haciendo rollback.

```
sequelize db:migrate:undo
```

Se puede deshacer todas las migraciones con:

```
sequelize db:migrate:undo:all
```

10. Volvemos a crear la tabla con el punto 8

Configurando Sequelize 3/3

11. Introducimos datos en la tabla con Sequelize

    ```
    sequelize seed:generate --name students
    ```

12. Editamos el archivo seeders/[TIMESTAMP]-students.js

13. Poblamos la base de datos

    ```
    sequelize db:seed:all
    ```

14. Si queremos borrar los datos usamos

    ```
    sequelize db:seed:undo:all
    ```

15. Comprobamos lo que ha ocurrido en PgAdmin: http://localhost:8081

16. Revisa el cheatsheet de queryInterface.

```
'use strict';

/** @type {import('sequelize-cli').Migration}*/
module.exports = {
 async up (queryInterface, Sequelize) {
   return queryInterface.bulkInsert('students', [
     {
       name: 'John',
       last_name: 'Doe',
       date_of_birth: '1987-04-27',
       createdAt: new Date(),
       updatedAt: new Date(),
     },
     {
       name: 'Joaquin',
       last_name: 'Torres',
       date_of_birth: '2014-09-15',
       createdAt: new Date(),
       updatedAt: new Date(),
     },
   ], {});
 },

 async down (queryInterface, Sequelize) {
   return queryInterface.bulkDelete('students', null, {});
 }
};
```

Aplicación Sequelize 1/2

1. Observamos que en el archivo **models/index.js** se exporta la base de datos

```
// models/index.js
. . .
module.exports = db;
```

2. Creamos nuestro archivo de consultas para nuestra tabla de estudiantes

3. Revisa el cheatsheet de Sequelize

```
// repositories/students.js
const db = require("../models"); // Forma abreviada de
models/index.js

module.exports = {
 getAll() {
   return db.students.findAll({});
 },
 insert(data) {
   return db.students.create(data);
 },
};
```

Aplicación Sequelize 2/2

1. Creamos una aplicación de express en **app.js** que escuche en el puerto 3000
2. Importamos nuestro repositorio de estudiantes
3. Definimos la ruta **GET /students** que devuelve todos los estudiantes de la tabla (getAll)
4. Definimos la ruta **POST /students** para dar de alta un nuevo estudiante vía json. Necesitamos usar el middleware para convertir los datos a json

```
const express = require("express");
const students = require("./repositories/students" );
const app = express();
const port = 3000;
app.use(express.json());
app.get("/students", (req, res) => {
 students.getAll().then((results) => res.json(results));
});
app.post("/students", (req, res) => {
 if (!(req.body.name && req.body.last_name &&
req.body.date_of_birth)) {
   res.status(422).send('All fields are required (name, last_name,
date_of_birth)' );
 } else {
   students.insert(req.body).then(result => {
     res.json({ success: true, message: 'Student was saved
successfully' });
   }).catch(err => {
     res.json({ success: false, message: err.detail });
   });
 }
});
app.listen(port, () => {
 console.log(`Example server listening on http://localhost: ${port}`);
});
```

Ejercicios

1. Añadir un endpoint GET /students dónde recibir un identificador y como respuesta devolver los datos del estudiante, si no existe, devolver un mensaje 404 “Student doesn’t exist”.
2. Añadir dos columnas a la tabla students (**email** y **active**) a través de una migración, introducir un nuevo registro por la consola que lleve esos campos. (Genera la migración con: ***sequelize migration:create –name [nombre_migración]***)
3. Validar que al dar de alta un estudiante (POST) el dato introducido en el campo **email** sea un email válido usando express-validator: https://express-validator.github.io/docs/. Si no lo es, devolver un error con estado 400 informando de ello.
4. Validar que el valor del campo **date_of_birth** sea una fecha, si no lo es, devolver un error con estado 400 informando de ello.
5. Antes de insertar un nuevo estudiante, comprobar que no existe otro con ese mismo email, devolver un error 422 en ese caso con el mensaje “**A user already exists with this email**”

ALBAÑILESDIGITALES

Developing your new career

servicio navarro de empleo

das·nano

VeriDas