Clase 4

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 Knex?

Es un constructor de consultas (query builder) para PostgreSQL, CockroachDB, MSSQL, MySQL, MariaDB, SQLite3, Better-SQLite3, Oracle y Amazon Redshift diseñado para ser flexible, portátil y fácil de usar.

Funcionalidades:

- Cuenta con devoluciones de llamada de estilo de Node tradicional (callbacks), así como una interfaz de promesa para el control de flujo asíncrono más limpio,
- Es un constructor completo de consultas y esquemas
- Tiene soporte de transacciones (con puntos de guardado)
- Permite la agrupación de conexiones y respuestas estandarizadas entre diferentes clientes de consulta y dialectos.
- Más ligero que un ORM

Uso:

- Proyectos más sencillos.

Configurando Knex 1/3

1. Instalamos Knex y el cliente de Postgres

   ```
   npm install knex pg --save
   ```

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

   ```
   sudo npm install knex -g
   ```

3. Copiamos todo el contenido del archivo class4_materials.zip en nuestra carpeta. Iniciamos nuestro servicio de base de datos.

   ```
   npm run start-services
   ```

4. Iniciamos un proyecto de Knex

   ```
   knex init
   ```

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

```
// Update with your config settings. knexfile.js

/**
* @type { Object.<string, import("knex").Knex.Config> }
*/

module.exports = {
 development: {
   client: "pg",
   connection: {
     database: "express-db",
     user: "postgres",
     password: "1234",
   },
   pool: {
     min: 0,
     max: 5,
   },
 },
};
```

Configurando Knex 2/3

6. Creamos una migración

   ```
   knex migrate:make students
   ```

7. Creamos la tabla students en el archivo migrations/[timestamp]_students.js

8. Ejecutamos la migración

   ```
   knex migrate:latest
   ```

9. Probamos a eliminar la tabla haciendo rollback

   ```
   knex migrate:rollback
   ```

10. Volvemos a crear la tabla con el punto 8

```
/**
* @param { import("knex").Knex } knex
* @returns { Promise<void> }
*/
exports.up = function (knex) {
 return knex.schema.createTable("students", (table) => {
   table.increments("id").primary();
   table.string("name");
   table.string("last_name");
   table.date("date_of_birth");
 });
};

/**
* @param { import("knex").Knex } knex
* @returns { Promise<void> }
*/
exports.down = function (knex) {
 return knex.schema.dropTableIfExists("students");
};
```

Configurando Knex 3/3

11. Introducimos datos en la tabla con Knex

    ```
    knex seed:make 01_students
    ```

12. Editamos el archivo seeds/01_students.js

13. Poblamos la base de datos

    ```
    knex seed:run
    ```

14. Comprobamos lo que ha ocurrido en PgAdmin: https://localhost:8081

```
/**
* @param { import("knex").Knex } knex
* @returns { Promise<void> }
*/
exports.seed = async function(knex) {
 // Deletes ALL existing entries
 await knex('students').del();
 await knex('students').insert([
   {name: 'Hugo', last_name: 'López', date_of_birth: '2018-10-01'},
   {name: 'María', last_name: 'Jiménez', date_of_birth:
'1982-11-04'},
   {name: 'Asier', last_name: 'Valencia', date_of_birth:
'2019-05-12'}
 ]);
 // await knex.raw('select setval(\'students_id_seq\', max(id)) from
students');
};
```

Aplicación Knex 1/2

1. Conectamos nuestra aplicación a la base de datos

```
// repositories/db.js
const environment = process.env.NODE_ENV || "development";
const configuration = require("../knexfile")[environment];
const db = require("knex")(configuration);

module.exports = db;
```

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

```
// repositories/students.js
const db = require("./db");

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

3. Revisa el cheatsheet: https://devhints.io/knex

Aplicación Knex 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.message });
   });
 }
});
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 una columna **email** a la tabla students a través de una migración, introducir un nuevo registro por la consola que lleve ese campo.
3. Validar que 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.

ALBAÑILESDIGITALES

Developing your new career

servicio navarro de empleo

das·nano

VeriDas