Ús de l'ORM Sequelize


Sequelize és un popular ORM (*Object Relational Mapping*) que permet treballar amb bases de dades relacionals definint per damunt els nostres propis models d'objectes, de manera que, en el nostre codi, treballem amb les dades com si foren objectes, però internament s'emmagatzemen i extrauen de taules relacionals. Actualment, Sequelize suporta diferents SGBD relacionals, com MySQL/MariaDB, PostgreSQL o SQLite, entre altres.

Per a l'exemple que seguirem en els següents apartats, crearem un projecte anomenat "*ContactesSequelize*" en la nostra carpeta de proves, i també hem de crear una base de dades buida anomenada "contactes_sequelize". Executa també la comanda `npm init` en el projecte *ContactesSequelize* per a deixar l'arxiu `package.json` preparat.

1. Instal·lació i primers passos

La instal·lació de Sequelize és igual de senzilla que la de qualsevol altre mòdul de Node, a través de la comanda `npm`. A més, Sequelize es recolza en altres llibreries per a poder comunicar-se amb la base de dades corresponent, i convertir així els registres en objectes i viceversa. És el que la pròpia llibreria denomina "dialectes" (*dialects*), i hem d'incloure la(s) llibreria(s) del dialecte o SGBD que hàgem seleccionat.

En el nostre cas, treballarem amb bases de dades MySQL, per la qual cosa necessitarem incloure la llibreria amb el *driver* corresponent per a connectar: `mysql2`, a més de la pròpia `sequelize`.

```
npm install sequelize mysql2
```

> **NOTA**: en el cas d'usar bases de dades MariaDB, per exemple, el *driver* que hauríem d'instal·lar seria `mariadb`.

Després, hem d'incorporar Sequelize als arxius font que ho necessiten en el nostre projecte, amb la corresponent instrucció `require`. Creem un arxiu `index.js` en el nostre projecte de proves creat anteriorment, i afegim aquest codi:

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

A continuació, hem d'establir els paràmetres de connexió a la base de dades en qüestió:

```
const sequelize = new Sequelize('nomBD', 'usuari', 'password', {
    host: 'nom_host',
    dialect: 'mysql'
});
```

En l'últim paràmetre s'admeten altres camps de configuració. Podem, per exemple, configurar un *pool* de connexions a la base de dades, de manera que s'auto-gestionen les connexions que queden lliures i es reassignen a noves peticions entrants.

En el nostre cas, si connectem amb una base de dades MySQL anomenada "contactes_sequelize" en el servidor local, ens quedaria una instrucció així (configurant un *pool* de 10 connexions, i adaptant l'usuari i contrasenya pel qual tinguem configurat):

```
const sequelize = new Sequelize('contactes_sequelize', 'root', 'root', {
    host: 'localhost',
    dialect: 'mysql',
    pool: {
        max: 10,
        min: 0,
        acquire: 30000,
        idle: 10000
    }
});
```

Expliquem amb més detall cada paràmetre utilitzat:

- **host**: direcció del servidor de la base de dades. En este cas, la base de dades està en el mateix servidor (*localhost*), però podria ser diferent depenent de la teua configuració.
- **dialect**: el tipus de base de dades que Sequelize usarà. En este cas, MySQL.
- **pool**: configuració del grup de connexions (pool), que inclou:
- **max**: nombre màxim de connexions en el grup
- **min**: nombre mínim de connexions en el grup
- **acquire**: es refereix al temps màxim, en mil·lisegons, que Sequelize espera per a adquirir una connexió del grup de connexions
- **idle**: es refereix al temps màxim, en mil·lisegons, que una connexió pot estar inactiva en el grup abans de ser desconnectada

D'aquesta manera garantim un rang de connexions disponibles per als clients que, de manera simultània, vulguen accedir a la base de dades. A mesura que entren noves peticions es reserven connexions dins d'eixe rang (fins a 10 alhora, com a màxim), i a mesura que les deixen d'utilitzar o les alliberen, tornen a estar disponibles.

2. Definint els models

El model o models de la nostra aplicació defineixen les diferents classes o estructures de dades que necessitarem per a gestionar la informació d'aquesta aplicació. Per a definir el model del nostre exemple, crearem una subcarpeta `models` en el nostre projecte. Dins, crearem un arxiu `contacte.js`, amb l'estructura que tindrà la taula de contactes:

```
module.exports = (sequelize, Sequelize) => {

    let Contacte = sequelize.define('contactes', {
        nom: {
            type: Sequelize.STRING,
            allowNull: false
        },
        telefon: {
            type: Sequelize.STRING,
            allowNull: false
        }
    });

    return Contacte;
};
```

Observa que a la propietat `module.exports` li associem una funció que rep dos paràmetres, que hem anomenat `sequelize` i `Sequelize`. Seran dues dades que arribaran de fora quan carreguem aquests arxius, i proporcionaran la connexió a la base de dades i l'accés a l'API de Sequelize, respectivament.

Com pots veure, hem exportat cada model per a poder ser utilitzat des d'altres arxius de la nostra aplicació. En aquesta pàgina pots consultar els tipus de dades disponibles per a definir els models en Sequelize. També pots veure ací alguns validadores que podem aplicar a cada camp, com per exemple comprovar si és un e-mail, si té un valor mínim i/o màxim, etc.

Una vegada definit el model (o els models), podem importar-lo(s) des de l'arxiu principal `index.js` amb el corresponent `require`, encara que en aquest cas haurem de passar-li com a paràmetres els objectes `sequelize` i `Sequelize` creats prèviament:

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('contactes_sequelize', 'root', 'root', {
    ...
};
const ModelContacte = require(__dirname + '/models/contacte');
const Contacte = ModelContacte(sequelize, Sequelize);
```

L'objecte `Contacte` que obtindrem al final ens permetrà fer operacions sobre la taula "contactes" utilitzant objectes en lloc de registres, com veurem a continuació.

També és possible definir relacions entre models, en el cas de tindre varis, per a així establir connexions *un a un*, *un a molts* o *molts a molts*. És una cosa que no veurem en aquesta sessió per requerir més temps del qual disposem, però es pot consultar informació sobre aquest tema ací.

2.1. Aplicant els canvis

Tots els passos que hem definit abans no s'han materialitzat encara en la base de dades. Per a això, és necessari sincronitzar el model de dades amb la base de dades en si, utilitzant el mètode `sync` de l'objecte `sequelize`, una vegada establida la connexió i el model. Això ho farem des de l'arxiu principal `index.js`.

Podem passar-li com a paràmetre un objecte `{force: true}` per a forçar al fet que es creuen de zero totes les taules i relacions, esborrant el que hi haja prèviament. Si no es posa aquest paràmetre, no s'eliminaran les dades existents, simplement s'afegiran o modificaran les estructures noves que s'hagen afegit al model.

```
const sequelize = new Sequelize('contactes_sequelize', 'root', 'root', {
    ...
});
const ModelContacte = require(__dirname + '/models/contacte');
const Contacte = ModelContacte(sequelize, Sequelize);

sequelize.sync(/*{force: true}*/)
.then(() => {
    // Ací ja està tot sincronitzat
    // El nostre codi a continuació aniria ací
}).catch (error => {
    console.log(error);
});
```

Després de sincronitzar, observa que en cada taula que hàgem definit (en el nostre cas, només la taula de "contactes") s'han creat automàticament:

- Un *id* autonumèric com a clau primària (no ho havíem especificat en l'esquema)
- Un parell de camps addicionals de tipus data, que ens permeten emmagatzemar la data de creació i d'última modificació de cada registre. Aquestes dades s'acte-actualitzen quan inserim o modifiquem registres utilitzant els mètodes proporcionats per Sequelize, que veurem més tard.

3. Operacions sobre els models

Per a acabar el nostre exemple, vegem com realitzar diferents operacions sobre la base de dades amb Sequelize: llistats, insercions, esborrats i modificacions.

3.1. Insercions

Per a fer una inserció d'un objecte Sequelize, podem emprar el mètode estàtic `create`, associat a cada model. Rep com a paràmetre un objecte JavaScript amb els camps de l'objecte a inserir. Després, el mètode `create` es comporta com una promesa, per la qual cosa podem afegir les corresponents clàusules `then` i `catch`, o bé emprar l'especificació *async/await*.

Per exemple, així realitzaríem la inserció d'un contacte en la taula de contactes:

```
Contacte.create({
    nom: "Nacho",
    telefon: "966112233"
}).then(resultat => {
    if (resultat)
        console.log("Contacte creat amb aquestes dades:", resultat);
    else
        console.log("Error inserint contacte");
}).catch(error => {
    console.log("Error inserint contacte:", error);
});
```

De manera alternativa podem executar el mateix codi emprant l'especificació `async/await`:

```
// Definim una funció asíncrona que faça el treball
async function crearContacte() {
    try
    {
        const resultat = await Contacte.create({
            nom: "Nacho",
            telefon: "966112233"
        });

        if (resultat) {
            console.log("Contacte creat amb estes dades:", resultat);
        } else {
            throw new Error();
        }
    } catch (error) {
        console.log("Error inserint contacte");
    }
}

// Després pots cridar a la teua funció asíncrona
crearContacte();
```

3.2. Cerques

Per a realitzar cerques, Sequelize proporciona una sèrie de mètodes estàtics d'utilitat. Per exemple, el mètode `findAll` es pot emprar per a obtindre tots els elements d'una taula, o bé indicar algun paràmetre que permeta filtrar alguns d'ells.

D'aquesta manera implementaríem el llistat general de contactes:

```
Contacte.findAll().then(resultat => {
    console.log("Llistat de contactes: ", resultat);
}).catch(error => {
    console.log("Error llistant contactes: ", error);
});
```

Ací veiem el mateix exemple utilitzant `async/await`:

```
async function llistarContactes() {
    try
    {
        const resultat = await Contacte.findAll();
        console.log("Llistat de contactes: ", resultat);
    } catch (error) {
        console.log("Error llistant contactes: ", error);
    }
}

// Després pots cridar a la funció asíncrona
llistarContactes();
```

Si volguérem, per exemple, quedar-nos amb el contacte "Nacho", podríem fer una cosa així:

```
Contacte.findAll({
    where: {
        nom: "Nacho"
    }
}).then...
```

Una altra cerca que podem fer de manera habitual és la cerca per clau, a través del mètode `findByPk` (*buscar per clau primària*). Li passarem com a paràmetre en aquest cas l'*id* de l'objecte a buscar. Per a obtindre les dades d'un contacte a partir del seu *id*, pot quedar així:

```
Contacte.findByPk(1).then(resultat => {
    if (resultat)
        console.log("Contacte trobat: ", resultat);
    else
        console.log("No s'ha trobat contacte");
}).catch(error => {
    console.log("Error buscant contacte: ", error);
});
```

Ací podeu consultar altres tipus d'operadors i alternatives per a fer cerques filtrades.

3.3. Modificacions i esborrats

Per a realitzar modificacions i esborrats, primer hem d'obtindre els objectes a modificar o esborrar. Podem emprar els mètodes estàtics `update` i `destroy`.

- En `update`, passem com a primer paràmetre l'objecte amb les dades a actualitzar, i com segon paràmetre (opcional, però habitual) la condició que han de complir els objectes a actualitzar (la típica clàusula *where*)
- En `delete` passem com a primer paràmetre la condició *where* que han de complir els objectes a eliminar.

D'aquesta manera actualitzem el telèfon del contacte amb id = 1:

```
Contacte.update({telefono: "611223344"},
    {where: { id: 1 }}
).then(resultat => {
    console.log("Contacte actualitzat: ", resultat);
}).catch(error => {
    console.log("Error actualitzant contacte: ", error);
});
```

El mateix exemple amb `async/await`:

```
async function actualitzarContacte() {
    try
    {
        const resultat = await Contacte.update(
            { telefon: "611223344" },
            { where: { id: 1 } }
        );
        console.log("Contacte actualitzat: ", resultat);
    } catch (error) {
        console.log("Error actualitzant contacte: ", error);
    }
}

// Invocar a la funció asíncrona
actualitzarContacte();
```

Així esborraríem el contacte (o contactes) amb nom "Nacho"

```
Contacte.destroy({ where: { nom: "Nacho" }}
).then(resultat => {
    console.log("Contactes esborrats: ", resultat);
}).catch(error => {
    console.log("Error esborrant contactes: ", error);
});
```

El mateix exemple amb `async/await`:

```
async function esborrarContactes() {
    try
    {
        const resultat = await Contacte.destroy({ where: { nom: "Nacho" } });
        console.log("Contactes esborrats: ", resultat);
    } catch (error) {
        console.log("Error esborrant contactes: ", error);
    }
}

// Invocar a la funció asíncrona
esborrarContactes();
```

> **NOTA**: si afiges aquestes operacions una després d'una altra en l'arxiu `index.js` del nostre projecte de proves *ContactesSequelize*, has de tindre en compte que són asíncrones (treballen amb promeses), i per tant, no s'executaran seqüencialment. Dit d'una altra manera, si fem una inserció i a continuació un

> llistat, és possible que aquesta inserció no isca en el llistat perquè encara no s'ha executat del tot. És preferible que executes les instruccions una a una, deixant comentades la resta de proves, per a verificar el seu funcionament.

> **Exercici 1:**
>
> Crea una carpeta anomenada "**CancionesSequelize**" dins de la carpeta anterior "*ProjectesNode/Exercicis*". Crea també una base de dades anomenada "canciones" en MySQL.
>
> En el projecte, inicialitza l'arxiu `package.json` amb `npm init`, i instal·la els mòduls *sequelize* i *mysql2*. Ara definirem una carpeta `models` en el projecte, amb un arxiu anomenat `cancion.js`. De cada cançó emmagatzemarem el seu títol (text sense nuls), la seua duració en segons (sense nuls) i el nom de l'artista que la interpreta (text sense nuls).
>
> En el programa principal `index.js`, connecta amb la base de dades, carrega el model, sincronitza les dades i realitza les següents operacions:
>
> - Crea 2 noves cançons amb les dades que vulgues
> - Mostra les dades de la primera cançó
> - Modifica la duració d'alguna de les cançons
> - Esborra alguna de les dues cançons