# Tema 2. Acceso a datos con Node.js

## 2.5. Uso del ORM Sequelize

Sequelize es un popular ORM (*Object Relational Mapping*) que permite trabajar con bases de datos relacionales definiendo por encima nuestros propios modelos y esquemas, de forma que el código se asemeja más a lo que hemos estado haciendo con las bases de datos MongoDB: nos permitirá trabajar con los registros de las tablas como si fueran colecciones de objetos, y viceversa (almacenar objetos en tablas como registros). Actualmente soporta distintos SGBD relacionales, como MySQL/MariaDB, PostgreSQL o SQLite, entre otros.

Para el ejemplo que vamos a seguir en los siguientes apartados, vamos a crear un proyecto llamado "*PruebaLibrosSequelize*" en nuestra carpeta de pruebas, y una base de datos vacía llamada "libros_sequelize" desde phpMyAdmin.

### 2.5.1. Instalación y primeros pasos

La instalación de Sequelize es igual de sencilla que la de cualquier otro módulo de Express, a través del comando `npm`. Además, Sequelize se apoya en otras librerías para poder comunicarse con la base de datos correspondiente, y convertir así los registros en objetos y viceversa. Es lo que la propia librería denomina "dialectos" (*dialects*), y debemos incluir la(s) librería(s) del dialecto o SGBD que hayamos seleccionado.

En nuestro caso, vamos a trabajar con bases de datos MariaDB, por lo que necesitaremos incluir la librería con el *driver* correspondiente para conectar: `mariadb`, además de la propia `sequelize`. Para nuestro ejemplo, instalaremos también Express para implementar unos servicios REST básicos, y *body-parser* para procesar datos de las peticiones.

```
npm install sequelize mariadb express body-parser
```

Después, debemos incorporar Sequelize a los archivos fuente que lo necesiten en nuestro proyecto, con la correspondiente instrucción `require`. Creamos un archivo `index.js` en nuestro proyecto de pruebas creado anteriormente, y añadimos este código:

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

A continuación, debemos establecer los parámetros de conexión a la base de datos en cuestión:

```
const sequelize = new Sequelize('nombreBD', 'usuario', 'password', {
    host: 'nombre_host',
    dialect: 'mariadb'
});
```

En el último parámetro se admiten otros campos de configuración. Podemos, por ejemplo, configurar un *pool* de conexiones a la base de datos, de forma que se auto-gestionen las conexiones que quedan libres y se reasignen a nuevas peticiones entrantes.

En nuestro caso, si conectamos con una base de datos MariaDB llamada "libros_sequelize" usando el usuario y contraseña por defecto de *phpMyAdmin*, en el servidor local, nos quedaría una instrucción así (configurando un *pool* de 10 conexiones):

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

## 2.5.2. Definiendo los modelos

Para definir los modelos, vamos a crear una subcarpeta `models` en nuestro proyecto. Dentro, vamos a crear dos archivos, para los dos modelos que vamos a utilizar: uno llamado `libro.js` y otro llamado `autor.js`.

En el modelo de `autor.js`, vamos a definir la estructura que va a tener la tabla de "autores" de nuestra base de datos. De cada autor almacenaremos su nombre (texto, sin nulos) y su año de nacimiento (entero).

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

    let Autor = sequelize.define('autores', {
        nombre: {
            type: Sequelize.STRING,
            allowNull: false
        },
        nacimiento: {
            type: Sequelize.INTEGER
        }
    });

    return Autor;
};
```

En el modelo de `libro.js` definiremos los datos de los libros: su título y su editorial, ambos de tipo texto, y donde el título no admita nulos:

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

    let Libro = sequelize.define('libros', {
        titulo: {
            type: Sequelize.STRING,
            allowNull: false
        },
        editorial: {
            type: Sequelize.STRING
        }
    });

    return Libro;
};
```

Observa que, en ambos archivos, a la propiedad `module.exports` le asociamos una función que recibe dos parámetros, que hemos llamado `sequelize` y `Sequelize`. Serán dos datos que llegarán de fuera cuando carguemos estos archivos, y proporcionarán la conexión a la base de datos y el acceso a la API de Sequelize, respectivamente.

Como puedes ver, hemos exportado cada modelo para poder ser utilizado desde otros archivos de nuestra aplicación. En esta página puedes consultar los tipos de datos disponibles para definir los modelos en Sequelize. También puedes ver aquí algunos validadores que podemos aplicar a cada campo, como por ejemplo comprobar si es un e-mail, si tiene un valor mínimo y/o máximo, etc.

### 2.5.2.1. Relaciones entre modelos

Para establecer relaciones entre los distintos modelos, que a su vez se traduzcan en las correspondientes claves ajenas en la base de datos, Sequelize ofrece unos método de utilidad.

Partiendo del ejemplo anterior, lo que vamos a hacer es ir al programa principal `index.js` e importar ( `require` ) los dos modelos creados. Podemos hacerlo justo bajo la carga de librerías externas:

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

const ModeloAutor = require(__dirname + "/models/autor");
const ModeloLibro = require(__dirname + "/models/libro");
```

A continuación, creamos las instancias de cada modelo a partir de los elementos importados. Recordemos que dichos modelos necesitaban recibir como parámetro la conexión a la base de datos (parámetro `sequelize` , y el acceso a la librería (parámetro `Sequelize` )). Así que esto deberemos hacerlo después de establecer la conexión con la base de datos:

```
const sequelize = new Sequelize('libros_sequelize', 'root', '', {
    ...
});

const Autor = ModeloAutor(sequelize, Sequelize);
const Libro = ModeloLibro(sequelize, Sequelize);
```

Ya tenemos los modelos asociados a las correspondientes tablas en la base de datos. Ahora podemos establecer relaciones entre ellos.

**Uno a muchos**

En nuestro caso, nos interesa tener una relación *uno a muchos*, en la que un libro pertenece a un autor, y un autor puede tener muchos libros. Esto se puede hacer de dos formas diferentes: empleando el método `hasMany` desde el modelo del autor, o el método `belongsTo` desde el modelo del libro:

```
Libro.belongsTo(Autor);
Autor.hasMany(Libro);
```

En ambos casos, lo que se va a crear es una clave ajena en la tabla `libros` que se va a llamad `autorId` . Podemos elegir el nombre de dicha clave ajena añadiendo un parámetro de configuración llamado `foreignKey` . También se crearán unos métodos *get/set* para acceder al objeto vinculado (al autor en este caso, a partir del libro). El nombre de dichos

*get/set* se puede especificar con un atributo de configuración llamado `as`. Así, nuestra relación entre libros y autores podemos definirla así:

```
Libro.belongsTo(Autor, {foreignKey: 'idAutor', as: 'Autor'});
```

Lo que creará en cada objeto libro un método `getAutor` para obtener el autor, y otro `setAutor` para asignarlo.

**Otras relaciones**

Además de la relación *uno a muchos* que hemos empleado, también podemos utilizar una relación *uno a uno*, o una *muchos a muchos*. En el primer caso, para una relación **uno a uno**, se utiliza el método `hasOne` en lugar de `hasMany`, o en sentido contrario, también se utiliza el método `belongsTo`. Imaginemos, por ejemplo, una relación entre un paciente y un expediente médico: cada paciente tiene un expediente, y un expediente sólo puede pertenecer a un paciente. Esto podríamos representarlo de cualquiera de estas dos formas:

```
Paciente.hasOne(Expediente);
Expediente.belongsTo(Paciente);
```

En ambos casos, se crearía una clave ajena en la tabla de expedientes referenciando al paciente al cual pertenece.

En el caso de una relación **muchos a muchos**, se utiliza el método `belongsToMany`. Sería el caso, por ejemplo, de alumnos y asignaturas matriculadas: un alumno puede estar matriculado de muchas asignaturas, y una asignatura puede tener muchos alumnos matriculados. En este caso, podemos representarlo así:

```
Alumno.belongsToMany(Asignatura,
    {as: 'Asignaturas', through: 'matricula', foreignKey: 'idAlumno'});
Asignatura.belongsToMany(Alumno,
    {as: 'Alumno', through: 'matricula', foreignKey: 'idAsignatura'});
```

En una relación muchos a muchos se creará una nueva tabla que represente las relaciones entre ambas claves ajenas. Para poder indicar el nombre de la nueva tabla, así como el nombre de las claves ajenas que referenciarán a cada una de las tablas implicadas, podemos pasar varios parámetros a los métodos anteriores:

- `as` se utiliza, como en los casos anteriores, para indicar cómo se llamarán los *getters* y *setters* que referencien a la otra parte desde uno de los objetos implicados.

- `through` se utiliza para indicar el nombre de la nueva tabla que se creará (*matricula*, en el ejemplo anterior).
- `foreignKey` sirve para lo mismo que el caso anterior también: indicar cómo se llamará cada una de las claves ajenas que se creará para referenciar a las tablas implicadas.

### 2.5.2.2. Aplicando los cambios

Todos los pasos que hemos definido antes no se han materializado aún en la base de datos. Para ello, es necesario sincronizar el modelo de datos con la base de datos en sí, utilizando el método `sync` del objeto `sequelize`, una vez establecida la conexión. Podemos pasarle como parámetro un objeto `{force: true}` para forzar a que se creen de cero todas las tablas y relaciones, borrando lo que haya previamente. Si no se pone dicho parámetro, no se eliminarán los datos existentes, simplemente se añadirán o modificarán las estructuras nuevas que se hayan añadido al modelo.

```
const sequelize = new Sequelize('libros_sequelize', 'root', '', {
    ...
});


const Autor = ModeloAutor(sequelize, Sequelize);
const Libro = ModeloLibro(sequelize, Sequelize);

Libro.belongsTo(Autor, {foreignKey: 'idAutor', as: 'Autor'});

sequelize.sync(/*{force: true}*/)
.then(() => {
    // Aquí ya está todo sincronizado
}).catch (error => {
    console.log(error);
});
```

Tras sincronizar, observa que en cada tabla se han creado automáticamente:

- Un *id* autonumérico como clave primaria (no lo habíamos especificado en el esquema)
- Un par de campos adicionales de tipo fecha, que nos permiten almacenar la fecha de creación y de última modificación de cada registro. Estos datos se auto-actualizan cuando insertemos o modifiquemos registros utilizando los métodos proporcionados por Sequelize, que veremos más tarde.

> **Ejercicios propuestos:**
>
> **1.** Crea una carpeta llamada "**T2_CancionesSequelize**" en tu espacio de trabajo, en la carpeta "*Ejercicios*". Crea también una base de datos llamada "canciones" con *phpMyAdmin*.

> Ahora, vamos a definir una carpeta `models` en el proyecto, con dos archivos llamados `artista.js` y `cancion.js`. De cada artista vamos a almacenar su nombre (texto sin nulos) y su nacionalidad (que sí puede admitir nulos). De las canciones almacenaremos su título (texto sin nulos), su duración en segundos (sin nulos) y el álbum al que pertenece (texto que sí admite nulos).
>
> En el programa principal `index.js`, conecta con la base de datos, carga los modelos y establece una relación *uno a muchos* entre ambos modelos, de forma que una canción pertenece a un artista, y un artista puede tener muchas canciones. Finalmente, sincroniza los cambios para crear las tablas y relaciones en la base de datos.

### 2.5.3. Sequelize y Express

Vamos ahora a integrar un servidor Express con los modelos y estructura de la base de datos definida con Sequelize. Definiremos una serie de servicios REST para gestionar tanto los autores como los libros.

Para empezar, crearemos una carpeta `routes` con los archivos `libros.js` y `autores.js`. En el archivo de `autores.js` dejaremos el esqueleto básico de los servicios que vamos a proporcionar para autores:

```
module.exports = (express, Autor) => {

    let router = express.Router();

    // Servicio de listado
    router.get('/', (req, res) => {
    });

    // Servicio de búsqueda por id
    router.get('/:id', (req, res) => {
    })

    // Servicio de inserción
    router.post('/', (req, res) => {
    });

    // Servicio de modificación
    router.put('/:id', (req, res) => {
    });

    // Servicio de borrado
    router.delete('/:id', (req, res) => {
    });

    return router;
}
```

Hacemos algo similar para el caso de los libros ( `routes/libros.js` ):

```
module.exports = (express, Libro, Autor) => {

    let router = express.Router();

    // Servicio de listado
    router.get('/', (req, res) => {
    });

    // Servicio de búsqueda por id
    router.get('/:id', (req, res) => {
    });

    // Servicio de inserción
    router.post('/', (req, res) => {
    });

    // Servicio de modificación
    router.put('/:id', (req, res) => {
    });

    // Servicio de borrado
    router.delete('/:id', (req, res) => {
    });

    return router;
}
```

> **NOTA**: observad que en el caso de estos enrutadores, también reciben unos parámetros externos: la instancia de Express ya creada, y los modelos de autores y/o libros. Estos datos se los pasaremos desde el programa principal cuando incorporemos estos archivos.

En el programa principal `index.js`, cargamos las librerías de Express y *body-parser*, y los enrutadores asociados a las rutas `/autores` y `/libros`, respectivamente. Finalmente, ponemos en marcha el servidor express tan pronto como se produzca la sincronización con la base de datos. Nos quedará algo así:

```
const Sequelize = require('sequelize');
const express = require('express');
const bodyParser = require('body-parser');

const ModeloAutor = require(__dirname + "/models/autor");
const ModeloLibro = require(__dirname + "/models/libro");

const routerAutores = require(__dirname + "/routes/autores");
const routerLibros = require(__dirname + "/routes/libros");

const sequelize = new Sequelize('libros_sequelize', 'root', '', {
    ...
});

const Autor = ModeloAutor(sequelize, Sequelize);
const Libro = ModeloLibro(sequelize, Sequelize);

Libro.belongsTo(Autor, {foreignKey: 'idAutor', as: 'Autor'});

const autores = routerAutores(express, Autor);
const libros = routerLibros(express, Libro, Autor);

let app = express();

app.use(bodyParser.json());
app.use('/libros', libros);
app.use('/autores', autores);

sequelize.sync()
.then(() => {
    app.listen(8080);
}).catch (error => {
    console.log(error);
});
```

**Ejercicios propuestos:**

**2.** Sobre el ejercicio de canciones y artistas anterior, deja preparados los enrutadores para las operaciones sobre canciones y artistas, en la carpeta `routes`, y cárgalos en la aplicación principal, como en el ejemplo anterior. Estarán asociados a las rutas `/canciones` y `/artistas`, respectivamente. Inicia también el servidor Express en cuanto se haya sincronizado el modelo de Sequelize con la base de datos.

## 2.5.4. Operaciones sobre los modelos

Para terminar nuestro ejemplo, debemos definir las operaciones de los distintos servicios: listados, inserciones, borrados y modificaciones. Veamos cómo hacer cada una de ellas con

Sequelize:

#### 2.5.4.1. Inserciones

Para hacer una inserción de un objeto Sequelize, podemos emplear el método estático `create`, asociado a cada modelo. Recibe como parámetro un objeto JavaScript con los campos del objeto a insertar. Después, el método `create` se comporta como una promesa, por lo que podemos añadir las correspondientes cláusulas `then` y `catch`, o bien emplear la especificación *async/await*.

Por ejemplo, así realizaríamos la inserción de un autor, en el servicio `POST /autores`:

```
// Servicio de inserción
router.post('/', (req, res) => {
    Autor.create({
        nombre: req.body.nombre,
        nacimiento: req.body.nacimiento
    }).then(resultado => {
        if (resultado)
            res.status(200).send({ ok: true, resultado: resultado });
        else
            res.status(400).send({ ok: false,
                error: "Error insertando autor" });
    }).catch(error => {
        res.status(400).send({ ok: false,
            error: "Error insertando autor" });
    });
});
```

> **Ejercicios propuestos:**
>
> **3.** Sobre el ejercicio de canciones y artistas, añade ahora tú el código necesario para insertar nuevos artistas, en `POST /artistas`. Crea una colección llamada "Canciones" en Postman y añade este servicio para probarlo.

#### 2.5.4.2. Búsquedas

Para realizar búsquedas, Sequelize proporciona una serie de métodos estáticos de utilidad. Por ejemplo, el método `findAll` se puede emplear para obtener todos los elementos de una tabla, o bien indicar algún parámetro que permita filtrar algunos de ellos.

De esta forma implementaríamos el listado general de autores:

```
// Servicio de listado
router.get('/', (req, res) => {
    Autor.findAll().then(resultado => {
        res.status(200).send({ ok: true, resultado: resultado });
    }).catch(error => {
        res.status(500).send({ ok: false,
            error: "Error obteniendo autores" });
    });
});
```

Si quisiéramos, por ejemplo, quedarnos con los autores nacidos a partir de 1950, podríamos hacer algo así:

```
Autor.findAll({
    where: {
        nacimiento: { [Sequelize.Op.gte]: 1950 }
    }
})
```

Otra búsqueda que podemos hacer de forma habitual es la búsqueda por clave, a través del método `findByPk` (*buscar por clave primaria*). Le pasaremos como parámetro en este caso el *id* del objeto a buscar. Para obtener los datos de un autor a partir de su *id*, puede quedar así:

```
// Servicio de búsqueda por id
router.get('/:id', (req, res) => {
    Autor.findByPk(req.params['id']).then(resultado => {
        if (resultado)
            res.status(200).send({ ok: true, resultado: resultado });
        else
            res.status(400).send({ ok: false,
                error: "Autor no encontrado" });
    }).catch(error => {
        res.status(400).send({ ok: false,
            error: "Error buscando autor" });
    });
});
```

Aquí podéis consultar otros tipos de operadores y alternativas para hacer búsquedas filtradas.

> **Ejercicios propuestos:**

> **4.** Añade los servicios de listado de artistas y búsqueda de un artista a partir de su *id* en la aplicación de canciones y artistas. Añade las pruebas a la colección "Canciones" de Postman que has creado en el ejercicio anterior.

### 2.5.4.3. Modificaciones y borrados

Para realizar modificaciones y borrados, primero debemos obtener los objetos a modificar o borrar. En nuestro caso, al ser modificaciones y borrados individuales, emplearemos el método `findByPk`. Una vez obtenido el objeto, podemos directamente llamar al método `update` para cambiar los campos que queramos, o al método `destroy` para eliminar el objeto. En ambos casos, recogeremos los resultados de la promesa correspondiente.

Así quedaría el servicio de modificación de datos de autores:

```
// Servicio de modificación
router.put('/:id', (req, res) => {
    Autor.findByPk(req.params['id']).then(autor => {
        if (autor)
            return autor.update({ nombre: req.body.nombre,
                nacimiento: req.body.nacimiento });
        else
            reject ("Error actualizando autor");
    }).then(resultado => {
        res.status(200).send({ ok: true, resultado: resultado });
    }).catch(error => {
        res.status(400).send({ ok: false,
            error: "Error actualizando autor" });
    });
});
```

Y así quedaría el borrado:

```
// Servicio de borrado
router.delete('/:id', (req, res) => {
    Autor.findByPk(req.params['id']).then(autor => {
        if (autor)
            return autor.destroy();
        else
            reject ("Error borrando autor");
    }).then(resultado => {
        res.status(200).send({ ok: true, resultado: resultado });
    }).catch(error => {
        res.status(400).send({ ok: false,
            error: "Error borrando autor" });
    });
});
```

> **Ejercicios propuestos:**
>
> **5.** Añade los servicios de modificación y borrado de artistas en la aplicación de canciones y artistas, con las correspondientes peticiones de prueba en la colección de Postman.

### 2.5.4.4 Inserción de objetos con relaciones

Veamos ahora cómo podríamos insertar un libro asociándole un autor previamente insertado. En este caso, tenemos que seguir estos pasos:

1. Crear el libro con sus datos básicos (título y editorial)
2. Buscar el autor
3. Asociar el autor al libro

Para ayudarnos a realizar esta tarea de forma más sencilla, vamos a utilizar la especificación *async/await* para enlazar las operaciones: cuando tengamos el libro creado, buscaremos el autor, y cuando lo encontremos, lo asociaremos al libro. Esto nos evitará ir enlazando promesas y esperar a que termine una para pasar a la siguiente.

Crearemos un método asíncrono ( `async` ) que haga estas tres tareas de forma asíncrona ( `await` ):

```
let nuevoLibro = async (titulo, editorial, idAutor) => {
    try
    {
        let libro = await Libro.create({titulo: titulo, editorial: editor
        let autor = await Autor.findByPk(idAutor);
        let resultado = await libro.setAutor(autor);
        return resultado;
    } catch (error) {
        console.log(error);
        throw new Error(error);
    }
}
```

> **NOTA**: los *getters* y *setters* generados por Sequelize al relacionar modelos trabajan de forma asíncrona, por lo que es necesario tratarlos mediante promesas o con *await*.

Ahora, sólo tenemos que utilizar este método desde el servicio de inserción de libros ( `POST /libros` ):

```
// Servicio de inserción
router.post('/', (req, res) => {
    nuevoLibro(req.body.titulo, req.body.editorial, req.body.autor)
    .then(resultado => {
        res.status(200).send({ ok: true, resultado: resultado });
    })
    .catch (error => {
        res.status(400).send({ ok: false,
            error: "Error añadiendo libro" });
    });
});
```

A la hora de obtener datos vinculados de otra tabla con el correspondiente *getter*, debemos tener en cuenta que dicha operación es asíncrona. Así quedaría el servicio de ficha de un libro si quisiéramos adjuntar los datos del autor, por ejemplo:

```
router.get('/:id', (req, res) => {
    Libro.findByPk(req.params['id']).then(resultado => {
        if (resultado) {
            resultado.getAutor().then(autor => {
                res.status(200).send({ ok: true,
                    resultado: resultado, autor: autor });
            });
        }
        else
            res.status(400).send({ ok: false,
                error: "Libro no encontrado" });
    }).catch(error => {
        res.status(400).send({ ok: false,
            error: "Error buscando libro" });
    });
});
```

**Ejercicios propuestos:**

**6.** Termina de definir los servicios de búsqueda, modificación y borrado de libros en el ejemplo *PruebaLibrosSequelize*.

**7.** Define ahora todas las operaciones sobre las canciones (inserción, borrado, modificación y búsquedas), con sus correspondientes peticiones desde Postman. Para la inserción de canciones, utiliza una estrategia similar a la del ejemplo de los libros, para asociar cada canción con su artista.