

# 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 de objetos, de forma que, en nuestro código, trabajamos con los datos como si fueran objetos, pero internamente se almacenan y extraen de tablas relacionales. Actualmente, Sequelize 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 "PruebaContactosSequelize" en nuestra carpeta de pruebas, y también debemos crear una base de datos vacía llamada "contactos\_sequelize". Ejecuta también el comando `npm init` en el proyecto *PruebaContactosSequelize* para dejar el archivo `package.json` preparado.

## 1. Instalación y primeros pasos

La instalación de Sequelize es igual de sencilla que la de cualquier otro módulo de Node, 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 MySQL, por lo que necesitaremos incluir la librería con el *driver* correspondiente para conectar: `mysql2`, además de la propia `sequelize`.

```
npm install sequelize mysql2
```

**NOTA:** en el caso de usar bases de datos MariaDB, por ejemplo, el *driver* que tendríamos que instalar sería `mariadb`.

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: 'mysql'  
});
```

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 MySQL llamada "contactos\_sequelize" usando el usuario y contraseña de nuestra máquina virtual, en el servidor local, nos quedaría una instrucción así (configurando un *pool* de 10 conexiones):

```
const sequelize = new Sequelize('contactos_sequelize', 'alumno', 'alumno1234', {  
  host: 'localhost',  
  dialect: 'mysql',  
  pool: {  
    max: 10,  
    min: 0,  
    acquire: 30000,  
    idle: 10000  
  }  
});
```

## 2. Definiendo los modelos

---

El modelo o modelos de nuestra aplicación definen las distintas clases o estructuras de datos que vamos a necesitar para gestionar la información de dicha aplicación. Para definir el modelo de nuestro ejemplo, vamos a crear una subcarpeta `models` en nuestro proyecto. Dentro, vamos a crear un archivo `contacto.js`, con la estructura que va a tener la tabla de contactos:

```
module.exports = (sequelize, Sequelize) => {  
  
  let Contacto = sequelize.define('contactos', {  
    nombre: {  
      type: Sequelize.STRING,  
      allowNull: false  
    },  
    telefono: {  
      type: Sequelize.STRING,  
      allowNull: false  
    }  
  });  
  
  return Contacto;  
};
```

Observa que 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.

Una vez definido el modelo (o los modelos), podemos importarlo(s) al archivo principal `index.js` con el correspondiente `require`, aunque en este caso deberemos pasarle como parámetros los objetos `sequelize` y `Sequelize` creados previamente:

```
const Sequelize = require('sequelize');  
const sequelize = new Sequelize('contactos_sequelize', 'alumno', 'alumno1234', {  
  ...  
});  
const ModeloContacto = require(__dirname + '/models/contacto');  
const Contacto = ModeloContacto(sequelize, Sequelize);
```

El objeto `Contacto` que obtendremos al final nos permitirá hacer operaciones sobre la tabla "contactos" utilizando objetos en lugar de registros, como veremos a continuación.

También es posible definir relaciones entre modelos, en el caso de tener varios, para así establecer conexiones *uno a uno*, *uno a muchos* o *muchos a muchos*. Es algo que no veremos en esta sesión por requerir más tiempo del que disponemos, pero se puede consultar información al respecto [aquí](#).

## 2.1. 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 y el modelo. Esto lo haremos desde el archivo principal `index.js`.

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('contactos_sequelize', 'alumno', 'alumno1234', {
  ...
});
const ModeloContacto = require(__dirname + '/models/contacto');
const Contacto = ModeloContacto(sequelize, Sequelize);

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

Tras sincronizar, observa que en cada tabla que hayamos definido (en nuestro caso, sólo la tabla de "contactos") 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.

## 3. Operaciones sobre los modelos

Para terminar nuestro ejemplo, veamos cómo realizar distintas operaciones sobre la base de datos con Sequelize: listados, inserciones, borrados y modificaciones.

### 3.1. 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 contacto en la tabla de contactos:

```
Contacto.create({
  nombre: "Nacho",
  telefono: "966112233"
}).then(resultado => {
  if (resultado)
    console.log("Contacto creado con estos datos:", resultado);
  else
    console.log("Error insertando contacto");
}).catch(error => {
  console.log("Error insertando contacto:", error);
});
```

### 3.2. 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 contactos:

```
Contacto.findAll().then(resultado => {
  console.log("Listado de contactos: ", resultado);
}).catch(error => {
  console.log("Error listando contactos: ", error);
});
```

Si quisiéramos, por ejemplo, quedarnos con el contacto "Nacho", podríamos hacer algo así:

```
Contacto.findAll({
  where: {
    nombre: "Nacho"
  }
}).then...
```

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 contacto a partir de su *id*, puede quedar así:

```
Contacto.findByPk(1).then(resultado => {
  if (resultado)
    console.log("Contacto encontrado: ", resultado);
  else
    console.log("No se ha encontrado contacto");
}).catch(error => {
  console.log("Error buscando contacto: ", error);
});
```

[Aquí](#) podéis consultar otros tipos de operadores y alternativas para hacer búsquedas filtradas.

### 3.3. Modificaciones y borrados

Para realizar modificaciones y borrados, primero debemos obtener los objetos a modificar o borrar. Podemos emplear los métodos estáticos `update` y `destroy`.

- En `update`, pasamos como primer parámetro el objeto con los datos a actualizar, y como segundo parámetro (opcional, pero habitual) la condición que deben cumplir los objetos a actualizar (la típica cláusula *where*)
- En `delete` pasamos como primer parámetro la condición *where* que deben cumplir los objetos a eliminar.

De este modo actualizamos el teléfono del contacto con id = 1:

```
Contacto.update({telefono: "611223344"},
  {where: { id: 1 }
}).then(resultado => {
  console.log("Contacto actualizado: ", resultado);
}).catch(error => {
  console.log("Error actualizando contacto: ", error);
});
```

Y así borraríamos el contacto (o contactos) con nombre "Nacho"

```
Contacto.destroy({ where: { nombre: "Nacho" }
}).then(resultado => {
  console.log("Contactos borrados: ", resultado);
}).catch(error => {
  console.log("Error borrando contactos: ", error);
});
```

Tenéis más opciones disponibles de cómo realizar estas consultas [aquí](#)

**NOTA:** si añades estas operaciones una tras otra en el archivo `index.js` de nuestro proyecto de pruebas *PruebaContactosSequelize*, debes tener en cuenta que son asíncronas (trabajan con promesas), y por tanto, no se van a ejecutar secuencialmente. Dicho de otro modo, si hacemos una inserción y a continuación un listado, es posible que dicha inserción no salga en el listado porque aún no se ha ejecutado del todo. Es preferible que ejecutes las instrucciones una a una, dejando comentadas el resto de pruebas, para verificar su funcionamiento.