







 [arielZarate](#) / [henryAll](#) Public

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

 [main](#) ▾

⋮

[henryAll](#) / [FT-M4-master](#) / [03-sequelizeize](#) /

 arielZarate ...	3 days ago 
..	
 demo	3 days ago
 homework	3 days ago
 README.json	3 days ago
 README.md	3 days ago

☰ README.md



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

ORM

Un problema con las bases de datos relacionales (puede ser un gran problema en proyectos complejos) es que las cosas que guardamos en ella no mapean uno a uno a los objetos que tenemos en nuestra aplicación. De hecho, es probable que en nuestra App tengamos la clase *persona*, pero difícilmente tengamos la clase *ciudad* y que ambas estén relacionadas. Simplemente tendríamos una propiedad *ciudad* dentro de *persona*. Por lo tanto vamos a necesitar alguna capa de abstracción que nos oculte la complejidad de las tablas y sus relaciones y nosotros sólo veamos objetos desde la app. Para eso existen los **ORM** (Object relation mapping), que son librerías o frameworks que hacen este trabajo por nosotros. Sería lo mismo que `mongoose` , pero un poco al revés!

SEQUELIZE

Obviamente existen un montón de ORMs (de estos de verdad hay miles porque se vienen usando hace mucho). Nosotros vamos a utilizar `sequelize` , en particular. Este es un ORM para nodejs que soporta varios motores de bases de datos:

- PostgreSQL
- MySQL
- MariaDB
- SQLite
- MSSQL (Microsoft)

Por supuesto que ustedes deben probar y jugar con varios de ellos hasta que encuentren alguno que se acomode a su filosofía de programación, no hay *uno* que sea el mejor de todos.

Hay otras librerías que no llegan a ser ORMs pero nos ayudan a hacer queries a la base de datos, si les gusta tener el control al 100% de su base de datos le recomiendo probar con algunos de estos, por ejemplo: [MassiveJS](#)

Instalación

Como `sequelize` soporta varias bases de datos, vamos a necesitar primero instalar el módulo de `sequelize` per ser, y luego el módulo del conector a la base de datos que vayamos a usar. `sequelize` hace uso de estos últimos para conectarse a la base de datos.

```
$ npm install --save sequelize

# y uno de los siguientes
$ npm install --save pg pg-hstore
$ npm install --save mysql // Para MySQL y MariaDB
$ npm install --save sqlite3
$ npm install --save tedious // MSSQL
```

Igual que con Mongoose, primero vamos a importar el módulo y vamos a crear un objeto Sequelize pasándole como parámetro la string que indica a qué base de datos conectarse, con qué usuario y qué password:

```
var Sequelize = require("sequelize"); //requerimos el modulo
var sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname');
```

Modelos

Sequelize es muy parecido a Mongoose, primero vamos a tener que definir un modelo (las constraints y tipos de datos pueden diferir, pero el concepto es el mismo), los modelos se definen de la forma: `sequelize.define('name', {attributes}, {options})`. , veamos un ejemplo:

```
var User = sequelize.define('user', {
  firstName: {
    type: Sequelize.STRING,
    field: 'first_name' //el nombre en la base de datos va a ser first_name
  },
  lastName: {
    type: Sequelize.STRING
  }
});

User.sync({force: true}).then(function () { //tenemos que usar este método porque
  // Tabla creada antes de guardar datos en las tablas
  return User.create({ // estas tienen que estar definidas
    firstName: 'Guille',
    lastName: 'Aszyn'
  });
});
```

A diferencia de Mongoose, con sequelize vamos a tener que preocuparnos un poco por las tablas que haya en nuestra base de datos, ya que si no hay tablas creadas, no vamos a poder guardar datos (en mongodb nos creaba las colecciones solo, y no importaba la estructura para guardar documentos). Por eso usamos el método `sync()` que justamente sincroniza el modelo que tenemos en nuestra app con la BD (crea la tabla en la db si no existe), como vemos puede recibir un *callback*, en el cual usamos el método `create()` para crear un nuevo user y guardarlo en la tabla. Podemos ver más métodos y cómo funcionan en la [Documentación](#).

Fijense que en Sequelize para pasar un callback lo hacemos en la función `.then()` , eso es una promesa, por ahora usenlo como una callback normal, más adelante las veremos en detalle.

CRUD (Create, Read, Update, Delete)

Para insertar datos en la base de datos, vamos a usar la función `create()`, que básicamente lo que hace es crear una nueva instancia del modelo y lo guarda en la base de datos:

```
User.create({
  firstName: 'Juan',
  lastName: 'Prueba'
}).then(function(user) {
  console.log(user);
})
```

La función `create()` en realidad, encapsula dos comportamientos, como dijimos arriba: instanciar el modelo y guardar en la bd. De hecho, podríamos hacer ambas cosas por separado, usando la función `build()` y `save()`:

```
var user = User.build({
  firstName: 'Juan',
  lastName: 'Prueba'
})

user.save().then(function(user) {
  console.log(user);
})
```

Para buscar registros usamos la función `find`, que viene en distintos sabores: `findAll()`: Sirve para buscar múltiples registros, `findOne()`: sirve para buscar un sólo registro y `findById()`: es igual a `findOne()` pero podemos buscar sólo por el ID del registro.

```
User.findAll({ where: ["id > ?", 25] }).then(function(users) {
  console.log(users) //busca TODOS los usuarios
});

User.findOne({
  where: {firstname: 'Toni'},
  attributes: ['id', ['firstname', 'lastname']]
}).then(function(user) {
  console.log(user)
});

User.findById(2).then(function(user) {
  console.log(user) //El user con ID 2
});
```

Las búsquedas pueden ser más complejas que en Mongo, por la naturaleza de las relaciones, por lo tanto es importante que leamos bien la [documentación](#).

Para modificar un registro, o varios, tenemos que pasarle los nuevos atributos que queremos modificar, y además una condición de búsqueda, en este caso voy a cambiarle el nombre a todos los registros que tengan `id = 1` (sólo puede haber uno 😊):

```
User.update({
  firstname: 'Martin' ,
}, {
  where: {
    id: '1'
  }
});
```

Para borrar un registro, vamos a usar el método `destroy()` , que tambien recibe un parámetro de búsqueda, en el ejemplo vamos a borrar todos los registros que tengan id 1, 2 ,3 o 4:

```
User.destroy(
  { where: { id: [1,2,3,4] }
});
```

Homework

Completa la tarea descrita en el archivo [README](#)
