

HANDBOOK

SEQUELIZE PARA TODOS



INCLUYE
COMENTARIOS
Y ENSEÑANZAS
DE CLIPPY

by pol & gonza

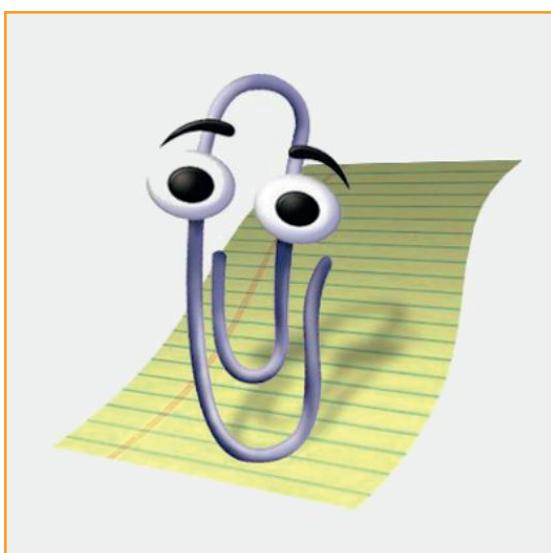
INTRODUCCIÓN

Sequelize es un **ORM** para Nodejs que te permitirá agilizar bastante tus desarrollos que incluyan bases de datos relacionales como MySQL o Postgress.

El **mapeo objeto-relacional** (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia.

Con sequelize vamos a poder crear Modelos que representarán nuestras tablas y sus relaciones.

¡Una vez configurado esto podremos utilizar grandes funcionalidades que nos permitirá hacer mucho más fácil el uso de db en nuestro proyecto!



CONFIGURACIÓN

Instalación

1. Instalamos el cli de sequelize de manera global

```
npm install sequelize-cli -g
```

2. Instalamos sequelize en nuestro proyecto

```
npm install sequelize
```

3. Instalamos el cliente de MySQL para node

```
npm install mysql2
```

Configuración

1. Crear `sequelizerc` en la raíz del proyecto con:

```
const path = require('path')

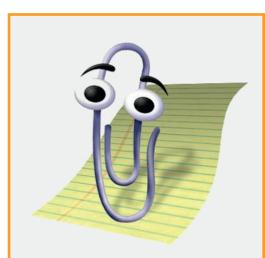
module.exports = {
  config: path.resolve('./database/config', 'config.js'),
  'models-path': path.resolve('./database/models'),
  'seeders-path': path.resolve('./database/seeders'),
  'migrations-path': path.resolve('./database/migrations'),
}
```

2. Correr el comando de inicialización

```
sequelize init
```

3. Esto creará las carpetas config, models, migrations, seeders.

Podemos omitir
la creación de `sequelizerc`
pero la ubicación de las carpetas
será la raíz del directorio.

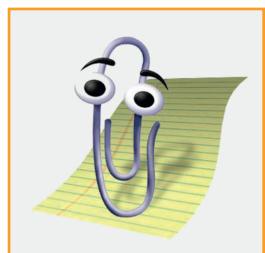


CONECTANDO LA BD

Dentro de database/config/config.js debemos poner los datos de nuestra conexión:

```
module.exports = {  
  "development": {  
    "username": "root",  
    "password": "root",  
    "database": "store",  
    "host": "localhost",  
    "dialect": "mysql",  
    "port": 8889  
  },  
  "test": {  
    "username": "root",  
    "password": null,  
    "database": "database_test",  
    "host": "127.0.0.1",  
    "dialect": "mysql"  
  },  
  "production": {  
    "username": "root",  
    "password": null,  
    "database": "database_production",  
    "host": "127.0.0.1",  
    "dialect": "mysql"  
  }  
}
```

A veces tenes que agregarle
module.exports =
al principio del archivo



En los otros ambientes
configuraremos sus bases de datos
¡OJO! nunca pongas los datos
de tu conexión de prod en este
archivo, siempre es mejor
levantarlo de variables de ambiente



CREANDO MODELOS

Creando nuestro primer modelo:

1. Debemos crear archivo en el path `/database/models/` un archivo js en CamelCase y singular:

Ejemplo:

Product.js

2. Definición del modelo. En nuestro archivo debemos exportar una función (la misma va a recibir sequelize y DataTypes)

```
module.exports = (sequelize, DataTypes) => {
  const alias = 'ModelName'
  const columns = {}
  const config = {}

  const Model = sequelize.define(alias, columns, config);
  return Model;
}
```

La función define recibe como primer parámetro el nombre (alias) del modelo, como segundo parámetro un objeto literal con la configuración de las columnas y como tercer parámetro otro objeto literal con configuración extra del modelo.

CREANDO MODELOS

Configuración de columnas:

```
const columns: {  
    id: {  
        autoIncrement: true,  
        primaryKey: true,  
        type: dataTypes.INTEGER  
    },  
    name: {  
        type: dataTypes.STRING  
        allowNull: true  
    },  
    price: {  
        type: dataTypes.DECIMAL,  
    },  
    image: {  
        type: dataTypes.STRING(200),  
    },  
    offer: {  
        type: dataTypes.BOOLEAN,  
        field: "otro_campo_field"  
    }  
}
```

Configuración extra:

```
const config: {  
    tableName: 'products',  
    timestamps: true, // buscará los campos createdAt/updatedAt  
    paranoid: true, // soft delete  
}
```

¿CON CUÁLES DATATYPES CONTAMOS?

```
Sequelize.STRING          // VARCHAR(255)
Sequelize.STRING(1234)     // VARCHAR(1234)
Sequelize.STRING.BINARY    // VARCHAR BINARY
Sequelize.TEXT             // TEXT
Sequelize.TEXT('tiny')     // TINYTEXT

Sequelize.INTEGER          // INTEGER
Sequelize.BIGINT           // BIGINT
Sequelize.BIGINT(11)        // BIGINT(11)

Sequelize.FLOAT             // FLOAT
Sequelize.FLOAT(11)         // FLOAT(11)
Sequelize.FLOAT(11, 10)      // FLOAT(11,10)

Sequelize.DOUBLE            // DOUBLE
Sequelize.DOUBLE(11)         // DOUBLE(11)
Sequelize.DOUBLE(11, 10)      // DOUBLE(11,10)

Sequelize.DECIMAL           // DECIMAL
Sequelize.DECIMAL(10, 2)      // DECIMAL(10,2)

Sequelize.DATE               // DATETIME for mysql / sqlite, TIMESTAMP WITH TIME ZONE for postgres
Sequelize.DATE(6)             // DATETIME(6) for mysql 5.6.4+. Fractional seconds support with up to 6 digits
of precision
Sequelize.DATEONLY           // DATE without time.
Sequelize.BOOLEAN            // TINYINT(1)

Sequelize.ENUM('value 1', 'value 2') // An ENUM with allowed values 'value 1' and 'value 2'
Sequelize.ARRAY(Sequelize.TEXT)    // Defines an array. PostgreSQL only.
Sequelize.ARRAY(Sequelize.ENUM)    // Defines an array of ENUM. PostgreSQL only.

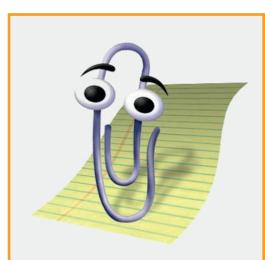
Sequelize.JSON                // JSON column. PostgreSQL, SQLite and MySQL only.
Sequelize.JSONB                // JSONB column. PostgreSQL only.

Sequelize.BLOB                  // BLOB (bytea for PostgreSQL)
Sequelize.BLOB('tiny')          // TINYBLOB (bytea for PostgreSQL. Other options are medium and long)
```

...

Podes ver todos los que faltan en
<https://sequelize.org/v5/manual/data-types.html>

La idea es que mapiemos
todas las columnas de nuestra
tabla a los DataTypes
que ofrese Sequelize



CONSULTAS BÁSICAS

Todas las operaciones que realizemos con Sequelize dentro de nuestros modelos retornarán una Promise.

Find

`db.Model.findAll()`

Devuelve todo el listado de rows

`db.Model.findOne()`

Devuelve la primer row

`db.Model.findPk(id)`

Busca la primaryId que matchee con el id proporcionado

Where

```
db.Model.findAll({  
  where: {  
    category: 'myCategory'  
  }  
})
```

Operadores Sequelize

```
const Op = db.Sequelize.Op
```

```
db.Model.findAll({  
  where: {  
    name: {  
      [Op.like]: '%clock'  
    }  
  }  
})
```

<https://sequelize.org/master/manual/model-querying-basics.html#operators>

CONSULTAS BÁSICAS

Order

Ordena por uno/varios campos en ASC o DESC

```
db.Model.findAll({  
    order: [  
        ['field_order', 'ASC']  
    ]  
})
```

LIMIT

Limita cantidad de resultados

```
db.Model.findAll({  
    limit: 10  
})
```

OFFSET

Omite cantidad de resultados

```
db.Model.findAll({  
    limit: 10  
})
```

creando un

CRUD

INSERTAR DATOS

Primero debemos crear una instancia del modelo y luego guardarlo.
El método `save()` nos devolverá la nueva instancia creada.

```
db.Product.create({
  name,
  category_id,
  price,
  order,
  image: file.filename
})
.then(newProduct => console.log("se creó el producto", newProduct))
.catch(error => console.log("Falló la creación del producto", error))
```

ACTUALIZAR DATOS

Tenemos varias opciones para hacer esto

1. Buscar la instancia y actualizarla:

```
Product.findOne({ where: { id: req.params.id } })
  .then(product => {
    product.update({
      name,
      price,
    })
  })
```

2. Utilizar el método update del modelo especificando un where:

```
Product.update({
  name,
  category_id,
  price,
  order,
},
{ where: { id: req.params.id } }
)
.then(result =>

)
```

ELIMINAR

```
Product.destroy({
  where: {
    id: req.params.id
  }
})
.then(() => {  
})
```

Dependiendo si el modelo Product tenemos `{ paranoid:true }` en la configuración, podrán pasar dos cosas, o borra el registro, o lo updatea poniéndole en **deletedAt** la fecha actual.

RELATIONS

1:N

AsociaciónhasMany <>> belongsTo

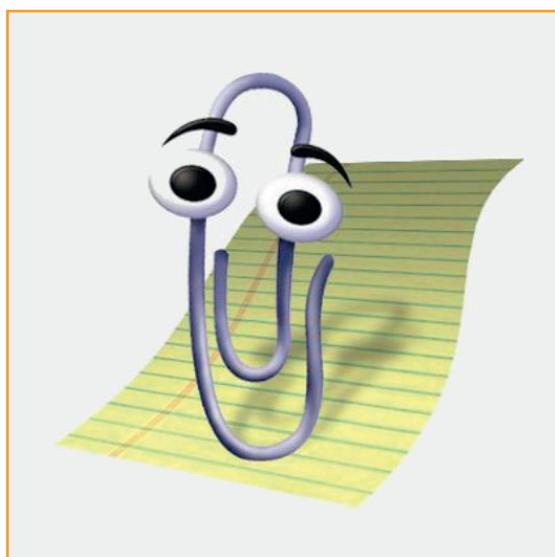
Cuando dos entidades están relacionadas de la forma que una tiene muchas de la otra hablamos de hasMany:

```
/*
desde users a addresses nos permitirá cargar las direcciones es un
usuario
*/
User.associate = models => {
    User.hasMany(models.Address, {
        as: 'addresses',
        foreignKey: 'address_id'
    })
}

/* desde addresses a users nos permitirá cargar el usuario de una
dirección */
Address.associate = models => {
    Address.belongsTo(models.User, {
        as: 'user',
        foreignKey: 'address_id'
    })
}
```

Truquito

Cuando no te acuerdes cual tiene el hasMany
y cuál el belongsTo, recordá que el que tiene
la fk es el que lleva el belongsTo



N:N

Asociación belongsToMany

Cuando trabajamos con una tabla pivot utilizamos belongsToMany

```
User.belongsToMany(models.Board, {
    as: 'boards',
    through: 'user_board',
    foreignKey: 'board_id',
    otherKey: 'user_id'
})
```

```
Board.belongsToMany(models.User, {
    as: 'users',
    through: 'user_board',
    foreignKey: 'user_id',
    otherKey: 'board_id'
})
```

ACCEDIENDO A LAS RELACIONES

Usando include:

```
// pasando un string solo permitirá una única
User.findAll({
    include: 'addresses'
})

/* pasando un array permitirá cargar más de una y más pa-
rametrización
*/
User.findAll({
    include: [
        { association: 'addresses' }
    ]
})
```

Usando magic method:

```
User.findOne()
    .then(user => {
        user.getBoard()
            .then(boards => console.log(boards))
    })
```

GUARDANDO N:N

Usando include:

```
// pasando un string solo permitirá una única
User.findByPk(1)
    .then(user => user.setBoards([board1, board2]))
```

MIGRATIONS

MIGRACIONES

CREAR NUEVA MIGRACIÓN

```
sequelize migration:generate --name nombre-migracion
```

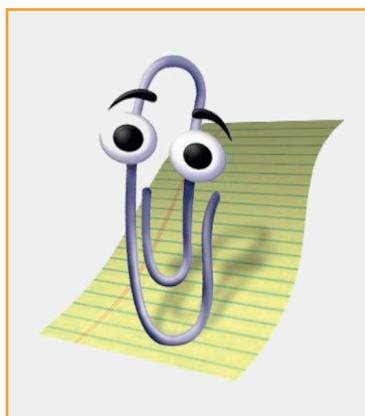
EJECUTAR MIGRACIONES PENDIENTES

```
sequelize db:migrate
```

EJEMPLO DE UNA MIGRACIÓN

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    return queryInterface.createTable('products', {
      name: Sequelize.STRING,
      category_id: {
        type: Sequelize.INTEGER,
        references: {
          model: 'categories',
          key: 'id'
        },
      }
    });
  },
  down: async (queryInterface, Sequelize) => {
    return queryInterface.dropTable('tags');
  }
};
```



Usando migrations no tendras más problemas de sincronización y exportar scripts para pasarle al resto de tu equipo.

MIGRACIONES

QUERY INTERFACE

Todas las acciones que podemos realizar es sobre cualquier modelo que podemos tener definido y también trabajar con queryInterface

<https://sequelize.org/master/class/lib/dialects/abstract/-query-interface.js~QueryInterface.html>

CREAR TABLA

`queryInterface.createTable`

ELIMINAR TABLA

`queryInterface.dropTable`

AGREGAR COLUMN

`queryInterface.addColumn`

ELIMINAR COLUMN

`queryInterface.removeColumn`

SEEDERS

Nos va a permitir cargar datos a nuestras columnas para que no empiecen vacías

CREAR SEED

```
sequelize seed:generate --name nombre-seed
```

CORRER SEEDS

```
sequelize db:seed:all
```

ROLLBACKEAR SEEDS

```
sequelize db:seed:undo:all
```