



INSTALACIÓN DE SEQUELIZE Y CONEXIÓN A LA DB

```
1 npm install sequelize-cli -g
2 npm install sequelize
3 npm install mysql2
```

1 - Instalar sequelize-cli, sequelize y mysql2

```
node modules
public
src
  .env
  .gitignore
  .sequelizer
  data.sql
  database.sql
  package-lock.json
  package.json
  prueba
```

2 - Crear el archivo '.sequelizer' en la carpeta raíz del proyecto.

```
const path = require('path')
module.exports = {
  development: {
    username: "root",
    password: "contraseña",
    database: "nombre-base-de-datos",
    host: "127.0.0.1",
    dialect: "mysql",
    operatorsAliases: false,
    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"
    }
  }
}
```

3 - El archivo recién creado debe contener la siguiente información.

```
1 sequelize init
```

4 - Luego correr en terminal el comando sequelize init, y creará las carpetas detalladas en el archivo .sequelizer.

```
modals.exports = {
  "development": {
    "username": "root",
    "password": "contraseña",
    "database": "nombre-base-de-datos",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false,
    "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"
    }
  }
}
```

5 - En el archivo config.js en la ruta /database/config/config.js debemos modificar algunas cosas:

- a - Primero debemos poner todo el JSON inicial dentro del module.exports
- b - Luego debemos modificar el JSON dentro de development con los datos de la conexión a la db que creamos en el Workbench.

```
.env
1 USERNAME_DB = "root"
2 PASSWORD_DB =
3 NAME_DB = "mercado_libre_entregable"
config.js
1 require('dotenv').config()
2
3 module.exports = {
4   "development": {
5     "username": process.env.USERNAME_DB,
6     "password": process.env.PASSWORD_DB,
7     "database": process.env.NAME_DB,
8     "host": "127.0.0.1",
9     "dialect": "mysql",
10    "operatorsAliases": false,
11  },
12  "test": {
13    "username": "root",
14    "password": null,
15    "database": "database_test",
16    "host": "127.0.0.1",
17    "dialect": "mysql"
18  },
19  "production": {
20    "username": "root",
21    "password": null,
22    "database": "database_production",
23    "host": "127.0.0.1",
24    "dialect": "mysql"
25  }
26 }
```

OPCIONAL:

Se pueden definir los datos de la conexión a través de un archivo .env.

Para esto primero creamos un archivo .env en la carpeta raíz, con el siguiente contenido.

Luego en el archivo config.js requerimos el archivo .env.

RELACIONES ENTRE MODELOS

Existen 3 tipos de relaciones entre tablas de una base de datos. Y en los tres casos, se deben relacionar 2 modelos

Relación 1:1

En esta relación, un registro de un modelo se relaciona únicamente con un registro de otro modelo. Este tipo de relaciones no se usaron en el curso, por lo que no hay ejemplos de como relacionarlos.

Relación 1:M

En este caso, un registro de un modelo se relaciona con muchos registros de otro modelo. Por ejemplo, una categoría de producto se asocia con muchos productos.

En el modelo de productos

```
1 Product.belongsTo(models.Product_categories, {
2   as : 'categories',
3   foreignKey : 'category_id'
4 })
```

En el modelo de categorías

```
1 Product_category.associate = (models) => {
2   Product_category.hasMany(models.Products, {
3     as : "product",
4     foreignKey : 'category_id'
5   })
6 }
```

En la sintaxis podemos ver los siguientes campos:

=> as: es el alias de la relación. NO SE DEBE REPETIR EN OTROS MODELOS

=> foreignKey: es el nombre de la clave foránea de la relación, que se encuentra únicamente en un modelo.

Relación N:M

Para poder relacionar los registros de un modelo con muchos registros de otro modelo, es necesario hacerlo a través de una tabla pivot. Por ejemplo, el carrito de compras de un usuario tiene muchos productos, y a su vez, un producto puede estar en el carrito de muchos usuarios.

En este caso es opcional realizar un modelo de la tabla pivot. Pero de hacerlo, NO HAY QUE RELACIONARLO CON LOS MODELOS DE LA RELACION N:M. Al relacionar los modelos extremos ya quedan asociados al modelo pivot.

En el modelo de productos

```
1 Product.belongsToMany(models.Colors, {
2   as: 'colors',
3   through: 'Images',
4   foreignKey: 'product_id',
5   otherKey: 'color_id',
6   timestamps: true,
7   createdAt: 'created_at',
8   updatedAt: 'updated_at',
9   deletedAt: 'deleted_at'
10 })
```

En el modelo de colores

```
1 Colors.associate = function(models) {
2   Colors.belongsToMany(models.Products, {
3     as: 'product_color',
4     through: 'Images',
5     foreignKey: 'color_id',
6     otherKey: 'product_id',
7     timestamps: true,
8     createdAt: 'created_at',
9     updatedAt: 'updated_at',
10    deletedAt: 'deleted_at'
11  })
12 }
```

En este caso estamos relacionando un modelo de productos, con un modelo de colores, a través de un modelo pivot llamado "Images", en el cual se establece la ubicación de la imagen para esa combinación producto-color.

En la sintaxis podemos ver los siguientes campos:

=> as: es el alias de la relación. NO SE DEBE REPETIR EN OTROS MODELOS

=> through: es el alias del modelo pivot, en caso de no existir, debe ponerse el nombre de la tabla pivot

=> foreignKey: es el nombre de la clave foránea en el Modelo en el que se está escribiendo la relación.

=> otherKey: es el nombre de la clave foránea del otro Modelo extremo.

METODOS EN LOS CONTROLADORES

1 - Para poder acceder a la base de datos desde un controlador se debe requerir Sequelize, db y Op.

Al requerir db con esta ruta, se está llamando al archivo index.js

```
1 const Sequelize = require('sequelize');
2 const Op = Sequelize.Op;
3
4 db.Products.findAll({
5   where: {
6     year: { [Op.like]: "%"+req.query.buscador+"%" }
7   },
8   order: [
9     ['final_price', 'DESC']
10 ],
11 limit: 20,
12 include: [{association: 'brand'}, {association: 'colors'}, {association: 'categories'}]
13 })
```

2 - Todo método de sequelize requiere una sintaxis básica, ejemplificada en la imagen.

=> se debe llamar al método como db.Model.Method donde Model es el alias del Modelo al que se quiere acceder, y Method el nombre del método.

=> Luego de cada método debe hacerse un .then() y un .catch() para resolver la promesa del método. Dentro del .then() se realiza una función que resuelve en caso de no haber errores en el método. Dentro del .catch() se resuelve en caso de haber errores. Por ejemplo:

```
.catch( error => { res.send(error) } )
```

```
32 detail: (req, res) => {
33   db.Products.findById(req.params.id)
34   .then(productShow => {
35     res.render('../products/detail', {productShow});
36   })
37   .catch(error => {res.send(error)})
38 },
```

```
42 let brandRequest = db.Brands.findAll()
43 let categoryRequest = db.Categories.findAll()
44
45 Promise.all([brandRequest, categoryRequest])
46 .then(([brands, categories]) => {
47   res.render('../products/product-create-form', {brands, categories});
48 })
49 .catch(error => {res.send(error)})
50 },
```

3 - Como dijimos, sequelize trabaja con promesas, y a cada promesa hay que darle un .then. Lo que permite hacer un nuevo método ya teniendo el resultado del anterior. Pero si ambos métodos son independientes, sequelize permite simplificar la estructura y hacer varios request en paralelo, y a través de Promise.all() continuar con la lógica únicamente cuando todos los request hayan sido respondidos.

```
1 db.Users.findOne({
2   where: {
3     email: req.body.email
4   },
5   paranoid: false
6 })
```

4 - Si estamos utilizando paranoid: true, por defecto sequelize agrega a la query que escribamos un "WHERE deleted_at IS NULL". Esto es muy efectivo porque automáticamente descarta todos los productos eliminados. Pero si por alguna razón queremos acceder a los items eliminados, debemos ingresar en la query el pedido de paranoid:false como se muestra en la imagen.
Un ejemplo puede ser si queremos saber que productos fueron eliminados, para restaurarlos.

CREACIÓN DE LOS MODELOS Y REALCIONES

```
1 module.exports = (sequelize , DataTypes) =>{
2
3   let alias = 'Brands';
4   let cols = {
5     id : {
6       autoIncrement : true,
7       primaryKey : true,
8       type : DataTypes.INTEGER,
9       allowNull : false
10     },
11     name : {
12       type : DataTypes.STRING,
13       allowNull : false
14     }
15   }
16   let config = {
17     tableName: 'brands',
18     timestamps: false,
19   }
20   const Brands = sequelize.define(alias, cols, config);
21   Brands.associate = models =>{
22     Brands.belongsTo(models.Products, {
23       as : 'brands_products',
24       foreignKey: 'brand_id'
25     })
26   }
27
28   return Brands;
29 }
```

1 - Un modelo está creado por una función que se exporta con module.exports, y tiene como parámetros "sequelize" y "DataTypes".

Esta función tiene 6 elementos:

=> El alias con el cuál llamaremos al modelo desde el controlador.

=> La definición de todas sus columnas (incluidas 'createdAt' y otras, si corresponde).

=> La configuración del modelo. Se puede definir el nombre de la tabla, timestamps, paranoid, y modificar el nombre de las columnas de timestamps

=> Se define la variable del modelo con sequelize.define(alias, cols, config)

=> Luego se definen las relaciones con otros modelos. A continuación se desarrollaran los distintos tipos de relaciones y como es la sintaxis.

=> Luego se debe retornar la constante del modelo

```
32 detail: (req, res) => {
33   db.Products.findByPk(req.params.id)
34   .then(productShow => {
35     res.render('../products/detail', {productShow});
36   })
37   .catch(error => {res.send(error)})
38 },
```

```
42 let brandRequest = db.Brands.findAll()
43 let categoryRequest = db.Categories.findAll()
44
45 Promise.all([brandRequest, categoryRequest])
46 .then(([brands, categories]) => {
47   res.render('../products/product-create-form', {brands, categories});
48 })
49 .catch(error => {res.send(error)})
50 },
```

```
1 db.Users.findOne({
2   where: {
3     email: req.body.email
4   },
5   paranoid: false
6 })
```

4 - Si estamos utilizando paranoid: true, por defecto sequelize agrega a la query que escribamos un "WHERE deleted_at IS NULL". Esto es muy efectivo porque automáticamente descarta todos los productos eliminados. Pero si por alguna razón queremos acceder a los items eliminados, debemos ingresar en la query el pedido de paranoid:false como se muestra en la imagen.
Un ejemplo puede ser si queremos saber que productos fueron eliminados, para restaurarlos.

METODOS

En las imágenes siguientes se mostrará la sintaxis de ejemplo

CREATE

Sirve para crear un registro en una tabla

```
42 db.Products.create({
43   title: req.body.name,
44   description: req.body.description,
45   photo: req.body.photo,
46   price: req.body.price,
47   stock: req.body.stock,
48   brand_id: req.body.brand,
49   category_id: req.body.category
50 })
```

FIND ALL

Permite buscar todos los registros que cumplen los requisitos

```
1 db.Products.findAll({
2   where: {
3     category_id: CategorySearched.id
4   }
5 })
```

FIND BY PK

Permite buscar un registro por su primary key

```
1 db.Products.findById(req.params.id)
```

FIND OR CREATE

Permite realizar una búsqueda de un elemento, si no lo encuentra, lo crea

```
1 db.Carts.findOrCreate({
2   where: {
3     user_id: req.session.user.id,
4     status: null
5   },
6   include: [
7     {association: 'item'}
8   ]
9 })
```

DESTROY

Permite borrar un registro. Si paranoid está activado, realiza un soft-delete. Es decir, que no lo borra, solo completa la columna deleted_at.

IMPORTANTE: siempre poner el where, de lo contrario borra todos los registros

```
313 db.Users.destroy({
314   where: {
315     id: res.locals.user.id
316   }
317 })
```

Orden, asociaciones, operadores y limit

A continuación se muestra la sintaxis de como agregar orden, asociaciones, limit y operadores a una query. Si no queremos traer todas las columnas de una asociación, dentro del objeto literal que menciona la relación se escribe de la siguiente manera:

```
{ association: 'brand',
  attributes: ['array con las columnas que queremos] }
```

En la segunda imagen se puede ver la sintaxis de las funciones de agregación.

```
1 db.Products.findAll({
2   where: {
3     year: { [Op.like]: "%"+req.query.buscador+"%" }
4   },
5   order: [
6     ['final_price', 'DESC']
7   ],
8   limit: 20,
9   include: [{association: 'brand'}, {association: 'colors'}, {association: 'categories'}]
10 })
```

```
1 db.Cart_product.sum('subtotal', {
2   where: {
3     cart_id: usersCart.id
4   }
5 })
```

En la imagen de la derecha vemos la sintaxis de como crear un producto. Y dentro del .then() vemos la sintaxis de como crear una relación N:M con otro modelo, asociando el producto recién creado.

```
creado.addRelación ( id_otraTabla, {
  through: {
    columna: data
  }
})
```

Donde creado, es el registro recién creado. Relación es el alias de la relación, desde el modelo de creado. id_otraTabla es el id de la otra tabla con el que se quiere relacionar. Y through es una sintaxis que permite agregar información a la tabla pivot.

En el ejemplo de la imagen se está creando un producto, y luego se lo relaciona con un color de la tabla Colors, y en la tabla pivot se está agregando la ubicación de la imagen de ese producto que tiene ese color.

BONUS

Si en el controlador buscamos una película con sus relaciones con los actores. Desde la vista, la forma de acceder a cada nombre de actor es a través de la siguiente sintaxis

```
1 <ul>
2   <% for( let i = 0; i < peliculas.length; i++ ) { %>
3     <li>
4       <%= peliculas.actores[i].first_name %>
5     </li>
6   <% } %>
7 </ul>
```

Como regla para otros casos, la sintaxis es:

variable._del_controlador.alias._relación[i].columna

En un formulario de edición, queremos que ya aparezca seleccionada la opción que actualmente tiene el registro. En la imagen de abajo se muestra la sintaxis necesaria para cada select del formulario, que busque información en la db

```
1 <select name="genero" id="genero">
2   <% for( let i = 0; i < generos.length; i++ ) { %>
3     <% if( generos[i].id == pelicula.genre_id ) { %>
4       <option value="<%= generos[i].id %>" selected>
5         <%= generos[i].name %>
6       </option>
7     <% } else { %>
8       <option value="<%= generos[i].id %>">
9         <%= generos[i].name %>
10      </option>
11    <% } %>
12  <% } %>
13 </select>
```