

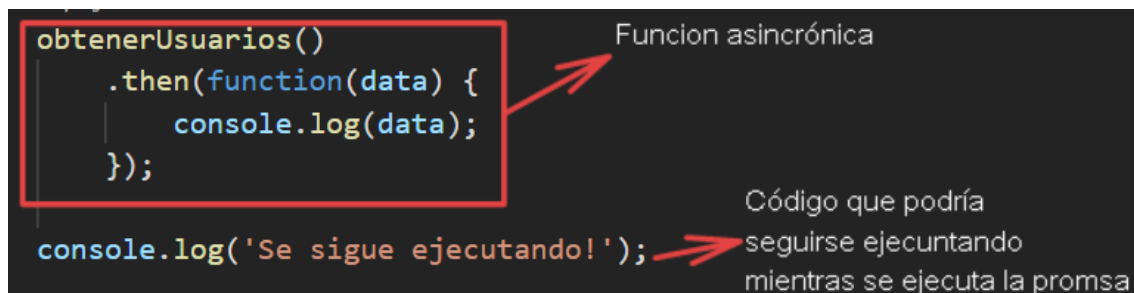
Introducción, modelos, y consultas

Promesas

Las promesas son funciones que permiten ejecutar **código asíncrono** de forma eficiente.

Un **pedido asíncrono** es un conjunto de instrucciones que se ejecutan mediante un mecanismo específico, como puede ser un **callback**, una **promesa**, o un **evento**. Esto hace posible que la respuesta sea procesada en otro momento.

Este comportamiento es **no bloqueante**, ya que el pedido se ejecuta en paralelo con el resto del código.



En este caso, el **console.log(data)** se ejecutará SOLO SI **obtenerUsuarios()** devuelve un resultado. Este resultado lo recibe el **.then()** dentro de su callback.

Hay casos donde tendremos **pedidos anidados**, es decir, que los **.then()** tengan promesas dentro. En este caso, utilizamos otro **.then()** que entre en ejecución apenas se resuelva el anterior

```
obtenerUsuarios()
  .then(function(data) {
    return filtrarDatos(data);
  })
  .then(function(dataFiltrada){
    console.log(dataFiltrada);
  });

console.log('Se sigue ejecutando!');
```

Aquí, **dataFiltrada** es la información que devuelve el return del primer **.then()**.

Por último, en caso de NO obtener un resultado, se genera un error. Podemos interceptarlo con **.catch()**. Es dentro de este método donde decidiremos que hacer con el error. Este llega como un parámetro dentro del callback del **.catch()**.

```
obtenerUsuarios()
  .then(function(data) {
    console.log(data);
  })
  .catch(function(error){
    console.log(error);
  });

console.log('Se sigue ejecutando!');
```

A veces necesitaremos que dos o más promesas se resuelvan para hacer una acción. Para eso usamos **Promise.all()**. Se usa de la siguiente forma:

```
let promesaPeliculas = obtenerPeliculas();
let promesaGeneros = obtenerGeneros();

Promise.all([promesaPeliculas, promesaGeneros])
  .then(function([resultadoPeliculas, resultadoGeneros]){
    console.log(resultadoPeliculas, resultadoGeneros);
  })
```

Primero, debemos guardar en variables todas las promesas a cumplir:

```
let promesaPeliculas = obtenerPeliculas();
let promesaGeneros = obtenerGeneros();
```

Luego, usamos el **Promise.all()** para evaluarlas todas, y en el callback del **.then()** enviamos un array con los resultados de las promesas como parámetro (y ejecutamos el código que precisemos)

```
Promise.all([promesaPeliculas, promesaGeneros])
  .then(function([resultadoPeliculas, resultadoGeneros]){
    console.log(resultadoPeliculas, resultadoGeneros);
  })
```

Documentación respecto a promesas:

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Sequelize

Sequelize es un ORM (Object Relational Mapper) que nos ayuda a conectarnos e interactuar con bases de datos relacionales. Esto nos permite escribir comandos en javascript en vez de tener que escribir queries para acceder a la base de datos

Para instalarlo:

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

```
npm install sequelize
```

Debemos instalar un paquete para decirle a sequelize que tipo de motor de base de datos estamos utilizando, y se pueda conectar al mismo:

```
npm install mysql2
```

Luego, debemos crear dentro de la carpeta **src** del proyecto el archivo **.sequelizerc**, con lo siguiente:

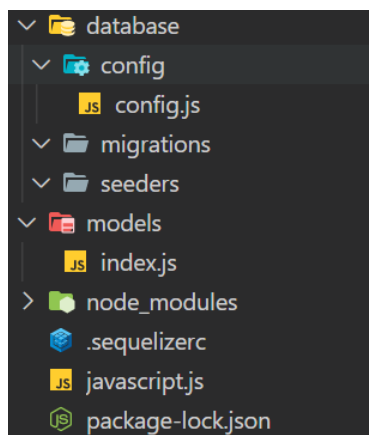
```
const path = require('path')

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'),
}
```

A continuación, con el comando `cd` nos paramos en **src**, y en la terminal escribimos:

```
sequelize init
```

Esto creara las carpetas **database** y **models**, cada una con varias cosas adentro.



Es importante ir a **database > config > config.js** y cambiar el código agregando `module.exports` al principio:



The diagram illustrates the modification of a database configuration file. On the left, a JSON-like object is shown with fields for 'development' and 'test' environments, including 'username', 'password', 'database', 'host', and 'dialect'. An arrow points to the right, where the same object is wrapped within a `module.exports = { ... }` structure, indicating it is being exported as a module.

```
{
  "development": {
    "username": "root",
    "password": null,
    "database": "database_development",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "test": {
    "username": "root",

```

```
module.exports = {
  "development": {
    "username": "root",
    "password": null,
    "database": "database_development",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "test": {
    "username": "root",

```

Aquí mismo debemos configurar todos los campos de acuerdo a nuestra base de datos (nombre de la base de datos en database, y password si tienen Mac).

Por último, en **models > index.js** al final se exporta una variable llamada **db**. Esta es la variable que utilizaremos en el proyecto para interactuar con la base de datos.

Documentación:

<https://sequelize.org/>

Modelos

El modelo es la M en el patrón de diseño MVC. Este contiene únicamente los datos puros de aplicación. No contienen lógica que describa como pueden presentarse los datos a un usuario. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.

En resumen, un modelo es la **representación de nuestra tabla en código**. Así podremos realizar consultas e interacciones con la base de datos de manera simplificada, en este caso, con Sequelize.

Crear un modelo para una tabla:

Siempre el modelo debemos crearlo en la ruta **/database/models**. Es un archivo .js, y su nombre debe estar escrito en UpperCamelCase y en singular.



peliculas.js



Pelicula.js

Dentro de este archivo deberemos hacer lo siguiente (choclozo incoming):

```
module.exports = (sequelize, DataTypes) => {  
  ...  
  let alias = 'Peliculas';  
  let cols = {  
    id: {  
      type: DataTypes.INTEGER,  
      primaryKey: true,  
      autoIncrement: true  
    },  
    title: {  
      type: DataTypes.STRING  
    },  
    length: {  
      type: DataTypes.INTEGER  
    }  
  };  
  let config = {  
    tableName: 'movies',  
    timestamps: false  
  }  
  
  const Pelicula = sequelize.define(alias, cols, config);  
  return Pelicula;  
}
```

Bueno, a desmenuzarlo.

Primero, debemos exportar una función que tendrá dos parámetros, **sequelize**, y **dataTypes**:

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

Dentro de esta función, tenemos que crear una constante que será la que se exportará, con tres parámetros: **alias**, **cols**, y **config (opcional)**. Importante aclarar que esta constante debe tener el nombre de la tabla en singular y comenzando con mayúscula:

```
const Pelicula = sequelize.define(alias, cols, config);  
return Pelicula;
```

Bueno, estos tres parámetros los tenemos definidos previamente. Vamos uno por uno, del mas sencillo al más complejo (alias, config, cols).

```
let alias = 'Películas';
```

En **alias** se suele poner el nombre del modelo en plural.

```
let config = {
  tableName: 'movies',
  timestamps: false
}
```

config es opcional pero recomendado. El **tableName** no es necesario ya que sequelize tomará el plural del nombre del archivo y lo usará para referirse a la tabla. Si por algún motivo la tabla se llamara distinto, aquí debemos aclararlo.

Por otro lado, **timestamps** sirve para aclarar si usamos o no dos columnas llamadas **createdAt**, y **updatedAt**. Estas sirven para llevar registro de las modificaciones en la tabla. Sequelize asume que las usamos, así que si no es el caso, deberemos aclarárselo con **timestamps: false**.

```
let cols = {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  title: {
    type: DataTypes.STRING
  },
  length: {
    type: DataTypes.INTEGER
  }
};
```

Bien, aquí le definimos la estructura de nuestra tabla. **id**, **title**, **length** son los nombres de las tablas en la base de datos. Dentro es OBLIGATORIO escribir el type, para que sepa qué tipo de dato se guarda. Pueden ser los siguientes:

```
DataType.STRING           // VARCHAR(255)
DataType.STRING(400)      // VARCHAR(400)

DataType.INTEGER          // INTEGER
DataType.BIGINT           // BIGINT
DataType.FLOAT            // FLOAT
DataType.DOUBLE           // DOUBLE
DataType.DECIMAL          // DECIMAL

DataType.DATE             // DATE
```

Luego se pueden aclarar otras cosas, como si es un primary key, una foreign key, si debe ser no nulo, etc.

findAll(), findByPk() y findOne()

Para utilizar estos métodos debemos siempre importar donde queramos usarlo el modelo:

```
const db = require ('../database/models');
```

Luego escribiremos db.nombreTabla.funcion para interactuar con la tabla que queramos, por ejemplo:

```
db.Usuario.findAll()
```

Esto es una promesa, así que tenemos que utilizarlo con el .then()

```
db.Usuario.findAll()
  .then((resultados) => {
    console.log(resultados);
  })
```

findAll()

Es el equivalente a SELECT * en mysql. Se hace de la siguiente forma:

```
db.Usuario.findAll()
  .then((resultados) => {
    console.log(resultados);
  })
```

findByPk()

Sirve para buscar un registro cuya clave primaria sea el valor del parámetro pasado:

```
db.Auto.findByPk(42)
  .then((resultado) => {
    console.log(resultado);
  })
```

findOne()

Permite buscar resultados que coincidan con los atributos indicados en el objeto literal que recibe el método:

```
db.Usuario.findOne(
  { where: { name: 'Tony' } })
  .then((resultado) => {
    console.log(resultado)
  })
```

Documentación:

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

Where y operadores

El where de mysql se hace pasando un objeto literal (lo vimos en el findOne) con la consulta:

```
db.Usuario.findOne(  
  { where: { name: 'Tony' } })  
  .then((resultado) => {  
    console.log(resultado)  
  })
```

Esto solo nos sirve para buscar por “igual”. Para buscar con estilo LIKE, mayor, menor, etc, debemos primero importar los operadores:

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

Luego, para usar un LIKE deberíamos hacerlo así (apellidos que tengan una s):

```
db.Usuario.findAll(  
  {where:{apellido:[Op.like]: '%s%'}}})
```

Aquí tenemos todos los operadores (atentos con la versión):

<https://sequelize.org/v4/manual/tutorial/querying.html#operators>

```
[Op.and]: {a: 5}           // AND (a = 5)  
[Op.or]: [{a: 5}, {a: 6}]  // (a = 5 OR a = 6)  
[Op.gt]: 6,                // > 6  
[Op.gte]: 6,              // >= 6  
[Op.lt]: 10,              // < 10  
[Op.lte]: 10,             // <= 10  
[Op.ne]: 20,              // != 20  
[Op.eq]: 3,               // = 3  
[Op.not]: true,           // IS NOT TRUE  
[Op.between]: [6, 10],    // BETWEEN 6 AND 10  
[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15  
[Op.in]: [1, 2],          // IN [1, 2]  
[Op.notIn]: [1, 2],       // NOT IN [1, 2]  
[Op.like]: '%hat',        // LIKE '%hat'  
[Op.notLike]: '%hat'      // NOT LIKE '%hat'  
[Op.iLike]: '%hat'        // ILIKE '%hat' (case insensitive) (PG only)  
[Op.notILike]: '%hat'     // NOT ILIKE '%hat' (PG only)  
[Op.regexp]: '^h|a|t$',   // REGEXP/~/ '^h|a|t$' (MySQL/PG only)  
[Op.notRegexp]: '^h|a|t$', // NOT REGEXP/!~/ '^h|a|t$' (MySQL/PG only)  
[Op.iRegexp]: '^h|a|t$',  // ~* '^h|a|t$' (PG only)  
[Op.notIRegexp]: '^h|a|t$', // !~* '^h|a|t$' (PG only)  
[Op.like]: { [Op.any]: ['cat', 'hat'] }  
// LIKE ANY ARRAY['cat', 'hat'] - also works for iLike and notLike  
[Op.overlap]: [1, 2]      // && [1, 2] (PG array overlap operator)  
[Op.contains]: [1, 2]     // @> [1, 2] (PG array contains operator)  
[Op.contained]: [1, 2]    // <@ [1, 2] (PG array contained by operator)  
[Op.any]: [2, 3]         // ANY ARRAY[2, 3]::INTEGER (PG only)
```

Order y Limit

El ORDER se implementa de la siguiente manera:

```
db.Usuario.findOne({
  where: {
    name: 'Tony'
  },
  order: [
    ['desempeño', 'ASC']
  ]
})

.then((resultado) => {
  console.log(resultado)
})
```

LIMIT y OFFSET son bastante directos:

```
db.Usuario.findOne({
  where: {
    name: 'Tony'
  },
  order: [
    ['desempeño', 'ASC']
  ],
  limit: 5,
  offset: 5
})

.then((resultado) => {
  console.log(resultado)
})
```