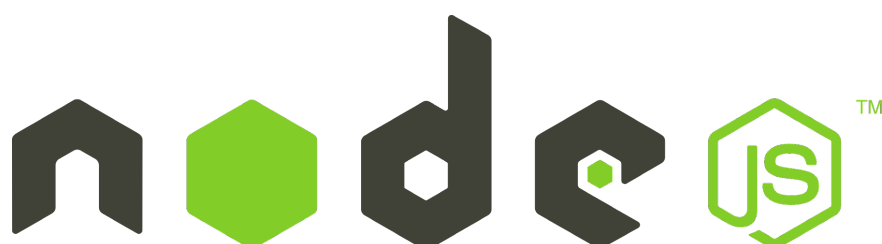


DESARROLLO DE SERVICIOS WEB CON NODEJS



Email: *julietalanciotti@gmail.com*

Módulo 4 - Bases de Datos

¿Qué es un ORM?

Ventajas

Paso a paso - ¿Cómo utilizar Sequelize?

Paso 1 - Instalación

Paso 2 - Archivo .sequelizerc

Paso 3 - Crear estructura de carpetas

Paso 4 - Crear base de datos

Paso 5 - Crear una tabla con Sequelize

Paso 6 - Sincronización de Modelos

Consultas con Sequelize

Asincronismo

¿Qué es un ORM?

Object-Relational Mapping (Mapeo Objeto-Relacional) es un modelo de programación que permite mapear las estructuras de una base de datos relacional (SQL), sobre una estructura lógica de entidades. De esta manera se logra simplificar y acelerar el desarrollo de nuestras aplicaciones.

Las estructuras de la base de datos relacional quedan vinculadas con la base de datos virtual definida en el ORM, de tal modo que las acciones ABM (alta, baja, modificación) a ejecutar sobre la base de datos física, se realizan de forma indirecta por medio del ORM.

Ventajas

- Nos “liberamos” de escribir consultas SQL.
- El ORM funciona como una capa intermedia totalmente separada de la base de datos. Esto permite que se pueda concentrar únicamente en el desarrollo de la aplicación.
- Las migraciones son más sencillas. Si se desea cambiar la aplicación, no se necesita reconstruir todas las consultas.
- No se necesita saber un lenguaje para cada base de datos: ORM unifica el lenguaje.
- Código más legible y con menos líneas.
- Brinda mayor seguridad.

Paso a paso - ¿Cómo utilizar Sequelize?

Sequelize es un ORM para Nodejs que permite agilizar los desarrollos que incluyen bases de datos relacionales como PostgreSQL y MySQL.

Este ORM permite manipular varias bases de datos SQL de una manera bastante sencilla.

i Todos estos pasos están explicado en el siguiente video donde les muestro cómo realizar todo: <https://youtu.be/0sUa5A80bbg>

Paso 1 - Instalación

1. `npm install --save sequelize` → instalación sequelize.
2. `npm install --save pg pg-hstore` → para poder utilizar una base de datos en postgres.
3. `npm install sequelize-cli` → nos permite usar funcionalidades de sequelize, como ejecutar seeders y migraciones (que veremos más adelante).
4. Por supuesto, en este punto, deberemos tener PostgreSQL también instalado en nuestra computadora.

i Por si lo necesitas, te dejo el link al video de la instalación de PostgreSQL:
https://youtu.be/VnUXm_lyDgw

Paso 2 - Archivo .sequelizerc

Es un archivo de configuración especial. Permite especificar diferentes opciones para que Sequelize sepa cómo crear y administrar parte de lo que necesitamos para operar la base de datos.

Este archivo se encontrará en la raíz del proyecto (junto con el package.json y los demás).

```
const path = require('path'); // importar path para trabajar con rutas

module.exports = {

  // configura la ruta del archivo de configuración de la base de datos
  "config": path.resolve("./src/database/config", "config.js"),

  // configura la ruta de los modelos de la base de datos
  "models-path": path.resolve("./src/database/models"),

  // configura la ruta de los seeders de la base de datos
  "seeders-path": path.resolve("./src/database/seeders"),

  // configura la ruta de las migraciones de la base de datos
  "migrations-path": path.resolve("./src/database/migrations"),

}
```

Paso 3 - Crear estructura de carpetas

Gracias al archivo de configuración, una vez que corramos el siguiente comando, veremos que se nos crean automáticamente diferentes carpetas

Comando: `npx sequelize-cli init`

Carpetas: /database/config , /database/migrations, /database/models,
/database/seeders

Una vez que se nos hayan creado las carpetas, modificaremos el archivo *index.js* de la carpeta **config** y le daremos la siguiente estructura:

```
module.exports = {  
  
  "development": {  
    "username": "root", // --> USUARIO QUE USAMOS CUANDO INSTALAMOS POSTGRESQL: postgres  
    "password": null, // --> CONTRASEÑA QUE USAMOS CUANDO INSTALAMOS POSTGRESQL  
    "database": "database_development", //--> NOMBRE DE LA BD  
    "host": "127.0.0.1",  
    "dialect": "postgres"  
  },  
  "test": {  
    "username": "root",  
    "password": null,  
    "database": "database_test",  
    "host": "127.0.0.1",  
    "dialect": "postgres"  
  },  
  "production": {  
    "username": "root",  
    "password": null,  
    "database": "database_production",  
    "host": "127.0.0.1",  
    "dialect": "postgres"  
  }  
}
```

Este archivo se va a encargar de decirle a Sequelize, cuál entorno de desarrollo estaremos utilizando y en cada uno, las credenciales de las bases de datos.

Paso 4 - Crear base de datos

Una vez configurado todo, crearemos la base de datos para luego comenzar a crear los modelos: **`npm sequelize-cli db:create`**

Como extra, para que luego sea más sencillo para alguien clonar el proyecto, podemos agregar el siguiente **script** en el archivo package.json

```
"db:create": "npm sequelize-cli db:create"
```

De esta forma, cuando queremos clonar el proyecto:

1. `npm start` → ejecuta el index.js con nodemon.
2. `npm run db:create` → crea la base de datos. Por ende, podemos ejecutar esta instrucción primero, y luego levantar la API con el start.

Paso 5 - Crear una tabla con Sequelize

Para crear modelos con Sequelize hay diferentes posibilidades. En este caso veremos cómo hacerlo de forma “manual” sin usar la consola. De esta manera evitamos errores y podemos realizar un modelo más completo.

Crearemos un archivo en la carpeta **models**. En este ejemplo, haremos la tabla médico con el campo id, nombre y DNI. Además le agregaremos 3 columnas más que guardarán la fecha de creación, de modificación y la de eliminación. Estas 3 últimas, es **necesario** que estén siempre en todas las tablas.

```
'use strict'

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

  let Medico = sequelize.define('medico', {
    id: {
      type: DataTypes.BIGINT,
      autoIncrement: true,
      primaryKey: true,
      allowNull: false
    },
    nombre: {
      type: DataTypes.STRING,
      allowNull: false
    },
    dni: {
      type: DataTypes.INTEGER,
      allowNull: false
    },
    createdAt: {
      type: DataTypes.DATE,
      field: 'created_at',
      defaultValue: DataTypes.NOW,
      allowNull: false
    },
    updatedAt: {
      type: DataTypes.DATE,
      field: 'updated_at',
      defaultValue: DataTypes.NOW,
      allowNull: false
    },
    deletedAt: {
      type: DataTypes.DATE,
      field: 'deleted_at'
    }
  }, { paranoid: true,
    freezeTableName: true,
  })

  Medico.associate = models => {
    Medico.hasMany(models.paciente)
  }

  return Medico
}
```


Analicemos las líneas de código de este archivo:

1. `'use strict'`

El modo estricto tiene varios cambios en la semántica normal de JavaScript:

- a. Elimina algunos errores silenciosos de JavaScript cambiándolos para que lancen errores.
- b. Corrige errores que hacen difícil para los motores de JavaScript realizar optimizaciones: a veces, el código en modo estricto puede correr más rápido que un código idéntico pero no estricto.

No es necesario que esté, pero de esta manera podemos evitar posibles problemas luego.

2. `let Medico = sequelize.define('medico')`

Creamos una variable llamada “Médico” donde estaremos llamando al módulo *define* de Sequelize para comenzar a crear nuestra tabla. El nombre de la tabla estará indicado entre comillas simples.

3. `let nombreVariable = sequelize.define('nombre_tabla')`

Definimos el primer campo de la tabla. Generalmente siempre tendrán un ID como clave primaria para evitar posibles errores en los cruces de las tablas.

```
id: { // --> NOMBRE DEL CAMPO
  type: DataType.BIGINT, // --> TIPO DEL CAMPO
  autoIncrement: true, // --> ES AUTOINCREMENTAL
  primaryKey: true, // --> CLAVE PRIMARIA DE LA TABLA
  allowNull: false // --> NO PERMITE QUE SEA NULL. SIEMPRE DEBE TENER UN DATO
},
```

```
dni: {  
  type: DataTypes.INTEGER, // --> TIPO ENTERO  
  allowNull: false // --> SI O SI DEBERÁ TENER UN VALOR ASIGNADO  
},
```

4.

```
createdAt: { // --> NOMBRE DEL CAMPO  
  type: DataTypes.DATE, // --> GUARDA UN TIPO DE DATO FECHA  
  field: 'created_at', // --> COMO SE LLAMARA REALMENTE EL CAMPO  
  defaultValue: DataTypes.NOW, // --> COMIENZA SIEMPRE CON LA FECHA ACTUAL  
  allowNull: false // --> NO ACEPTA EL CAMPO VACIO  
},
```

5.

6. **paranoid: true**

Sequelize admite el concepto de “*tablas paranoidas*”. Una tabla de este tipo, es aquella que, cuando se le dice que elimine un registro, no lo eliminará realmente. En su lugar, una columna especial llamada **deleteAt** guardará la fecha en que se pidió eliminar. De esta manera, se podrán evitar errores a futuro o pérdida accidental de información.

7. **freezeTableName: true**

Sequelize cambia automáticamente los nombres de sus tablas a plural. De esta manera, “congelamos” el nombre, y evitamos el cambio.

8. A continuación, si queremos comenzar a asociar nuestra tabla con otras, deberemos utilizar las directivas que nos brinda Sequelize.

```
Medico.associate = models => {  
  Medico.hasMany(models.paciente) // --> un medico tiene muchos pacientes  
}
```

- **HasOne**
A.hasOne(B) → cardinalidad (1,1), y la clave foránea está en B
- **BelongsTo**
A.belongsTo(B) → cardinalidad (1,1), y la clave foránea está en A
- **HasMany**
A.hasMany(B) → cardinalidad (1,N), y la clave foránea está en B

 Más información sobre relaciones:

<https://sequelize.org/docs/v6/core-concepts/assocs/>

Paso 6 - Sincronización de Modelos

sync() Esta instrucción permite sincronizar nuestro modelo con la base de datos. Internamente se corre la instrucción SQL **CREATE TABLE**. Esto nos evita tener que generar migraciones cada vez que queramos crear una nueva tabla.

Para poner en funcionamiento la sincronización de modelos, debemos agregar la siguiente línea al **archivo index.js de la carpeta models**: **sequelize.sync();** justo antes de la última línea.

 Para más información sobre sincronización:

<https://sequelize.org/docs/v6/core-concepts/model-basics/#model-synchronization>

En este punto, ya tenemos todo configurado y las tablas en nuestra base de datos creadas. Únicamente nos queda agregarle funcionalidad a nuestros endpoints.

Consultas con Sequelize

findAll	Obtiene todas las filas que encuentre que cumplan
findOne	Obtiene la primera fila que encuentre que cumpla
findOrCreate	Si no la encuentra, la crea

Información sobre consultas:

<https://sequelize.org/docs/v6/core-concepts/model-querying-basics/>

- `SELECT * FROM usuario;`
`const usuarios = await usuario.findAll()`

- SELECT * FROM usuario WHERE edad=10;

```
//OPCION 1
const usuarios = await usuario.findAll(
  where: {
    edad: 10
  }
)

//OPCION 2
const { Op } = require("sequelize")

const usuarios = await usuario.findAll({
  where:{
    edad:{
      [Op.eq]: 10
    }
  }
})
```


En la opción 2 se hace uso de los operadores de sequelize.

En este caso **eq** → **equals**

- `SELECT * FROM usuario WHERE edad=10 or edad=15;`

```
const { Op } = require("sequelize")

const usuarios = await usuario.findAll({
  where: {
    edad: {
      [Op.or]: [10,15]
    }
  }
})
```

 Más información sobre **operadores**:

<https://sequelize.org/docs/v6/core-concepts/model-querying-basics/#operators>

- Actualizar datos de una tabla. A la fila que tenga id=5 en la tabla usuario, le cambiará el nombre a Juan.

```
const usuarioModificado = await usuario.update({
  nombre: 'Juan'
},{
  where: {
    id: 5
  }
})
```

Asincronismo

Normalmente, el código de un programa determinado se ejecuta directamente, y solo sucede una cosa a la vez.

Si una **función A** se basa en el resultado de una **función B**, deberá esperar **a que la función B termine** y regrese, y hasta que esto suceda, todo el programa se detiene esencialmente desde la perspectiva del usuario.

Para solucionar este problema, surgió lo que es conocido como *Programación Asíncrona*.

Actualmente muchas funciones de API web utilizan código asíncrono para ejecutarse, especialmente aquellas que acceden o tienen algún tipo de recurso de un dispositivo externo (buscar un archivo de la red, acceder a una base de datos y devolver datos desde él, acceder a una transmisión de video con una cámara web, etc).

Las adiciones “más recientes” a JavaScript son las funciones asíncronas y la palabra clave **await**. Estas características actúan como “azúcar sintáctico” además de las promesas, lo que hace que el código asíncrono sea más fácil de escribir y leer después. (Lo veremos como un código sincrónico)

Es por esto, que cuando realizamos consultas a la base de datos, necesitamos que nuestras funciones sean asíncronas. Por eso utilizamos **async** y **await**.