

Apunte 11 - ORM con Sequelize

Introducción y Prerequisitos

Introducción

Este documento es parte de los materiales de estudio de la asignatura Desarrollo de Software. ¡Feliz aprendizaje!

Prerequisitos

Antes de tomar esta clase tenés que asegurarte de conocer y repasar los contenidos del taller de Bases de Datos Relacionales a saber:

- Fundamentos de **Bases de Datos** relacionales
- Conceptos de tablas, claves primarias y foráneas
- Lenguajes de definición de datos (DDL), manipulación de datos (DML) y consulta (SQL)
- Scripts DML de inicialización de la base de datos.
- Uso básico SQLite
- Uso de Node.js y manejo de dependencias con NPM

1. Necesidad de una base de datos

En general, sea cual sea el propósito de una aplicación, éstas necesitan manipular y almacenar datos. En un reducido número de casos, esta necesidad puede estar cubierta por una solución simple como un archivo o conjunto de archivos secuenciales, esta alternativa rápidamente queda limitada en cuanto a:

- Integridad de datos
- Acceso concurrente
- Eficiencia de búsqueda y filtrado
- Escalabilidad

entre otros aspectos. En la medida que la complejidad y el volumen de los datos aumentan se necesita de un gestor de base de datos para administrar esta información.

Los gestores de bases de datos (**DBMS - Database Management System**) nos permiten abstraernos de cómo los datos se almacenan físicamente, aseguran la integridad de los datos, permiten la concurrencia en el acceso a los datos, y nos permiten realizar operaciones con los datos de forma eficiente.

Existen distintos tipos de bases de datos según la organización de los datos: bases de datos relacionales, jerárquicas, orientadas a objetos, NoSQL, documentales, etc.

En Desarrollo de Software hemos decidido utilizar una bases de datos relacional para nuestros proyectos como complemento de lo que pueden estar estudiando en la asignatura Bases de Datos.

2. Bases de datos relacionales

Las bases de datos relacionales y sus productos de software encargados de administrarlas (**RDBMS - Relational Database Management System**) son el tipo de base de datos más extendido y de propósito más general. Si bien hoy hay muchas aplicaciones que se construyen con otros estilos de base de datos, más del 80% del software existente en el mundo está apoyado sobre bases de datos relacionales y por lo tanto no van a pasar a la historia en el futuro próximo.

Las bases de datos relacionales organizan los datos en forma de tablas y utilizan el lenguaje **SQL** (Structured Query Language) para la gestión y recuperación de los datos.

2.1 SQLite



En este curso vamos a utilizar **SQLite** que es un sistema de gestión de bases de datos relacional embebido de código abierto escrito en lenguaje C.

Este sistema tiene como ventajas su ubicuidad y disponibilidad en cualquier sistema ya que no es cliente servidor como la mayoría de los DBMS y sin embargo tiene la mayoría de las características de un sistema de gestión de base de datos relacionales más robusto como puede ser *MySQL*, *PostgreSQL* u otros. Esto quiere decir que todo lo que aprendas y desarrolles con SQLite vas a poder aplicarlo en otros DBMS relacionales con muy pocos o ningún cambio de por medio. Es muy utilizado en aplicaciones móviles y para actividades como **testing** ya que permite hasta disponer de bases de datos en memoria (sin persistencia física)

2.2 SQLite vs Motores Administrados

SQLite es un motor embebido: la base de datos se guarda como un único archivo en disco y su proceso se ejecuta dentro del mismo contexto de memoria que la aplicación que lo utiliza. Esto significa:

- No requiere instalación de servidor
- Es muy portátil y rápido para desarrollo local
- No soporta concurrencia a gran escala ni configuraciones distribuidas

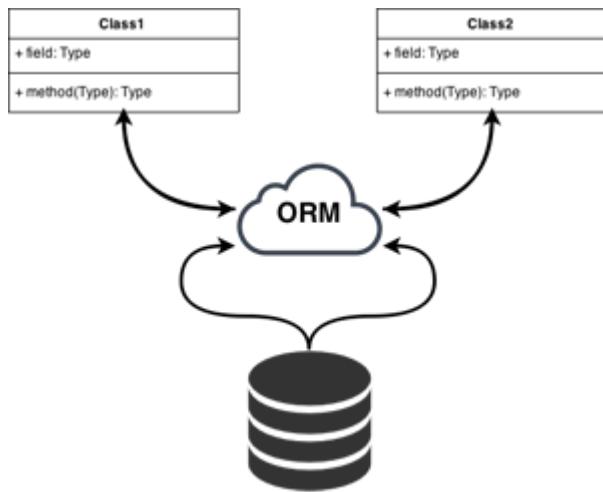
PostgreSQL, MySQL y otros motores independientes funcionan como servidores de base de datos (**DBMSs**):

- Pueden atender múltiples conexiones simultáneas
- Tienen herramientas avanzadas de administración
- Requieren instalación, configuración y administración de entorno

Para usar PostgreSQL en lugar de SQLite, consultar la documentación Sequelize en [Connection Examples](#)

3. ORM

ORM (*Object Relational Mapping*) o **Mapeo Objeto-Relacional** es una técnica que nos permite crear un "puente" entre las **entidades del modelo de negocios** de nuestro programa y el esquema de la base de datos relacional en que persisten los datos. Nos permite, entonces, trabajar con **objetos**, colecciones de objetos y propiedades en el lenguaje que estamos usando (en nuestro caso Javascript) en vez de hacerlo con **SQL** directamente.



En general no implementamos este mapeo "desde cero" sino que utilizamos herramientas (frameworks, bibliotecas) a las que por extensión llamamos también **ORM**

A modo de ejemplo veamos como se ve una operación sencilla como recuperar un usuario con id == 3 sin ORM y con ORM

Sin un ORM: necesitaríamos escribir código que ejecute una consulta sql a la base de datos, que recupere los resultados de dicha consulta y luego mapear los resultados a las propiedades del objeto. Y la consulta se vería como figura a continuación:

```
SELECT id, nombre, apellido, usuario, password, email FROM usuarios WHERE id = 3
```

Con un ORM simplemente le pedimos al framework de orm que se encargue de estas tareas y nos devuelva el objeto resultante.

```
Usuario.findOne({ where: { id: 3 } });
```

3.1. Ventajas

- Nos permite expresar todas las operaciones, incluyendo el acceso a datos, **en un mismo lenguaje**. Esto aporta facilidad y claridad a nuestro programa.
- Nos **abstiene** del RDBMS que estemos usando. Esto permite hacer cambios de RDBMS muy fácilmente. Por ejemplo, cambiar nuestra aplicación para que en vez de usar SQLite utilice MySql como motor de base de datos.
- Muchos de los ORMs nos brindan, además, **funcionalidad avanzada** como soporte para transacciones, pool de conexiones, migraciones, etc.

- En muchos casos (no en todos) las operaciones de datos pueden estar más optimizadas que si las escribieramos en SQL.

3.2. Desventajas

No todas son ventajas: hay una serie de factores que tenemos que tener en cuenta a la hora de tomar la decisión de si usar o no usar un ORM.

- Aquellos que tienen un muy buen dominio de SQL probablemente saquen mejor partido, a nivel de **eficiencia**, escribiendo las queries en SQL.
- Hay una **curva de aprendizaje** para comenzar a usar un ORM que debe ser considerada.
- La **configuración inicial** (mapeo) del ORM conlleva un esfuerzo y tiempo importante.
- El hecho de que se nos abstraiga, como programadores, de la complejidad de escribir el SQL trae aparejada la **dificultad para encontrar errores y problemas de performance**. Es muy importante mientras aprendemos a usar un ORM, adquirir el conocimiento y herramientas necesarias para realizar depuración y profiling. Seguramente no necesitaremos estas herramientas en todos los casos, pero habrá algunas situaciones límites en las que, de no contar con ellas, será un verdadero dolor de cabeza.

4. Sequelize



Sequelize es un ORM Typescript y Node.js basado en promesas para Oracle, Postgres, MySQL, MariaDB, SQLite, SQL Server, y otros. Tiene soporte para transacciones, relaciones, [carga diferida](#) y [precarga](#) (*lazy loading* y *eager loading*) y [replicación para la lectura](#)

Otras Características destacadas de Sequelize

- Basado en Promesas (async/await friendly) es decir pone en un hilo secundario de libuv el acceso a la base de datos.
- Permite definir modelos que representan las tablas y relaciones entre ellas
- Soporta validaciones, restricciones, asociaciones (1:1, 1:n, n:m)
- Como dijimos incluye manejo de transacciones, replicación, y estrategias de carga (lazy/eager loading)
- Permite ejecutar SQL puro cuando se necesita control total

Veamos cuales son los pasos fundamentales para utilizar **Sequelize** como ORM:

4.1. Instalar las dependencias

Dijimos que vamos a usar **SQLite** de modo que además de la dependencia para **Sequelize** instalaremos la que corresponde a esta base de datos.

```
npm install sqlite3
npm install sequelize
```

Para otros motores de base de datos:

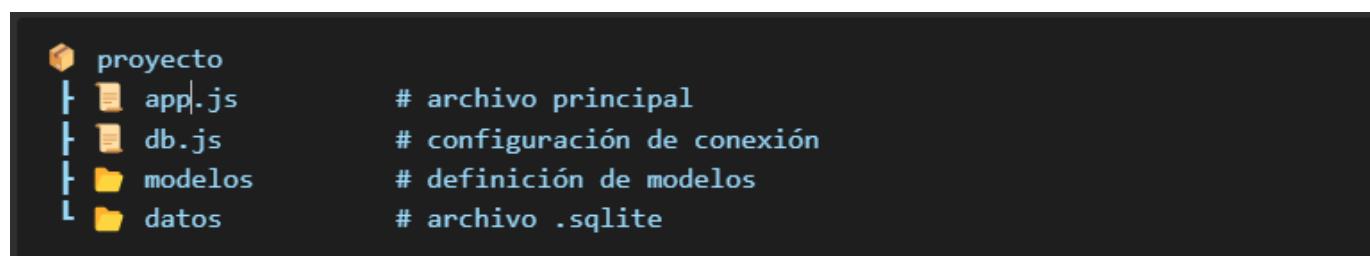
- **PostgreSQL:**

```
npm install sequelize pg pg-hstore
```

- **MySQL o MariaDB:**

```
npm install sequelize mysql2
```

Además, es recomendable crear una estructura básica de carpetas como la siguiente:



En db.js se suele colocar la configuración de Sequelize y la inicialización del objeto de conexión a la base de datos para ser reutilizado en toda la app.

Esta estructura también es la que vas a encontrar en los ejemplos paso a paso en la carpeta [pap_sequelize](#).

4.2. Conectar con la base de datos

Esta operación depende del motor de base de datos que estemos usando. Vamos a hacerlo con **SQLite**, como dijimos, en la documentación se pueden consultar el resto de las alternativas.

En **SQLite** tenemos la opción de conectarnos a una base de datos **en memoria**, es decir, que no se guarda en disco. Esta alternativa es comúnmente usada para testing, no es muy útil para desarrollo ni para producción ya que los cambios en la base de datos no estarán presentes una vez que termine de correr el programa.

Una vez instaladas las dependencias y estructurado el proyecto, debemos establecer la conexión entre Sequelize y la base de datos SQLite. Esto se realiza generalmente en un archivo db.js separado:

- archivo **db.js**

```
// db.js
import { Sequelize } from 'sequelize';

const sequelize = new Sequelize({
```

```
  dialect: 'sqlite',
  storage: './datos/db.sqlite',
});

export default sequelize;
```

Una alternativa de conexión que es clave para pruebas, es crear la base de datos en memoria:

```
// db.js
import { Sequelize } from 'sequelize';

const sequelize = new Sequelize("sqlite::memory:");
```

Este bloque crea una instancia de Sequelize utilizando SQLite como motor, y define la ubicación del archivo .sqlite que contendrá la base de datos persistente.

En la mayoría de los casos, también se recomienda probar la conexión al iniciar la aplicación, por ejemplo desde app.js:

- archivo **app.js**

```
// app.js
import sequelize from './db.js';

async function main() {
  try {
    await sequelize.authenticate();
    console.log('✓ Conexión establecida con la base de datos');
  } catch (error) {
    console.error('✗ Error al conectar con la base de datos:', error);
  }
}

main();
```

✍ También se puede usar `sequelize.sync()` para sincronizar los modelos definidos con la base de datos automáticamente:

```
await sequelize.sync();
```

Esto generará las tablas si no existen, lo cual es muy útil en etapas de desarrollo.

```
const { Sequelize } = require("sequelize");

const sequelize = new Sequelize("sqlite::memory:");
```

4.3. Tipos de Datos

Sequelize incluye una cantidad importante de tipos de datos, cuya lista exhaustiva puede ser consultada en la [documentación](#). Para tener disponibles los tipos de datos incluidos en **Sequelize** necesitamos importar **DataTypes**

```
import { DataTypes } from "sequelize";
```

Algunos de los tipos más usuales son:

DataTypes.STRING	// VARCHAR(255)
DataTypes.STRING(1234)	// VARCHAR(1234)
DataTypes.BOOLEAN	// TINYINT(1)
DataTypes.INTEGER	// INTEGER
DataTypes.BIGINT	// BIGINT
DataTypes.FLOAT	// FLOAT
DataTypes.DOUBLE	// DOUBLE
DataTypes.DATE	// FECHA y HORA
DataTypes.DATEONLY	// FECHA sin HORA

4.4. Definir un modelo

Un modelo es una abstracción que representa una tabla de la base de datos. Cuando definimos un modelos especificamos, el nombre de la tabla, las columnas de la tabla y sus [tipos de datos](#).

Además del nombre de la columna y su tipo de datos, podemos especificar ciertas características adicionales de las columnas mediante otras propiedades. Por ejemplo:

- **Clave primaria:** mediante **primaryKey: true** indicamos al modelo que la columna es clave primaria.
- **Columna autoincremental:** si agregamos el atributo **autoIncrement: true** la columna será autoincremental.
- **Valores nulos:** con el atributo **allowNull** podemos definir una columna para que acepte valores nulos (**allowNull:true**) o no los acepte (**allowNull:false**)
- **Valor por defecto:** podemos hacer que una columna tome un valor por defecto cuando este no es asignado explicitamente asignando la propiedad **defaultValue** por ejemplo **fecha_alta: { type: DataTypes.DATE, defaultValue: DataTypes.NOW }**

Hay dos esquemas de definición de modelos permitidos en sequelize, uno de ellos está basado en la función **define** que proporciona sequelize y el otro está basado en POO.

Comparativa de esquemas de definición de Modelos

Tendencia actual

- El enfoque **basado en clases con extends Model** y uso de **sintaxis ES6 (import/export)** es el **más recomendado y utilizado** en nuevos proyectos.

- Sequelize mismo, desde su documentación oficial, promueve el uso de clases como una forma más clara, extensible y alineada con prácticas modernas de programación orientada a objetos.

🔍 Comparación

Característica	<code>sequelize.define()</code> (funcional)	<code>class extends Model</code> (orientado a objetos)
Facilidad inicial	<input checked="" type="checkbox"/> Simple y directo	<input type="radio"/> Requiere definir clase y registrar modelo
Claridad para proyectos grandes	<input type="radio"/> Más difícil de escalar	<input checked="" type="checkbox"/> Mejora modularidad y legibilidad
Extensión con métodos personalizados	<input checked="" type="checkbox"/> Poco intuitivo	<input checked="" type="checkbox"/> Muy flexible
Alineación con patrones modernos	<input type="radio"/> Histórico pero menos usado	<input checked="" type="checkbox"/> Preferido en código profesional moderno

⌚ Recomendación

Conociendo estos datos cada uno puede optar por el esquema que prefiera.

4.4.1 Esquema de definición de Modelo con el enfoque Orientado a Objetos

En Sequelize, un modelo representa una tabla de la base de datos. Cada instancia del modelo representa una fila. A continuación se muestra cómo definir un modelo utilizando clases ES6 con sintaxis moderna y "type": "module" en el package.json.

- archivo modelos/Usuario.js

```
// modelos/Usuario.js
import { DataTypes, Model } from 'sequelize';
import sequelize from '../db.js';

class Usuario extends Model {}

Usuario.init({
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true
  },
  nombre: {
    type: DataTypes.STRING,
    allowNull: false
  },
  apellido: {
    type: DataTypes.STRING,
    allowNull: false
  }
}, {
  sequelize,
  modelName: 'Usuario'
})
```

```
},
usuario: {
  type: DataTypes.STRING,
  unique: true,
  allowNull: false
},
fecha_alta: {
  type: DataTypes.DATE,
  defaultValue: DataTypes.NOW
}
},
{
  sequelize,
  modelName: 'Usuario',
  tableName: 'usuarios',
  timestamps: false
});

export default Usuario;
```

💡 Este esquema basado en clases es más escalable, compatible con TypeScript (Un lenguaje basado en Javascript muy utilizado también) y facilita agregar métodos personalizados o integrarlo con un patrón de Repositorio más adelante.

💡 Nota técnica: Sequelize no infiere automáticamente los atributos existentes en la tabla de base de datos. Cada campo que querés usar desde JavaScript debe estar declarado explícitamente en el modelo, incluso si el nombre coincide con el de la columna en la tabla. Si no lo hacés:

- No podrás acceder ni manipular el campo desde instancias del modelo (usuario.nombre será undefined).
- No podrás aplicar validaciones ni valores por defecto.
- Sequelize ignorará ese campo en create, update o findAll.

Solo en casos especiales —como uso de consultas RAW— podrías trabajar con atributos no definidos en el modelo, pero eso va en contra del espíritu del ORM.

Luego se puede importar este modelo donde sea necesario:

```
// app.js
import Usuario from './modelos/Usuario';

await Usuario.sync() // crear la tabla si no existe

// crear un usuario en la base de datos (un insert en la tabla)
const nuevo = await Usuario.create({ nombre: 'Ana', apellido: 'García', usuario: 'ana2025' });

// mostrar el usuario creado
console.log('Nuevo usuario:', nuevo); //ver salida
```

Al ejecutar el proyecto hasta el paso anterior obtendremos:

```
[ philip@hpbook-Philip ~ ]$ main = ?4 ~1
$ npm start
> crud_usuarios@1.0.0 start
> node app.js

Executing (default): SELECT 1+1 AS result
✓ Conexión establecida con la base de datos
Executing (default): INSERT INTO `usuarios` (`id`, `nombre`, `apellido`, `usuario`, `fecha_alta`) VALUES (NULL,$1,$2,$3,$4);
Nuevo usuario: Usuario {
  dataValues: {
    fecha_alta: 2025-04-17T18:03:46.207Z,
    id: 1,
    nombre: 'Ana',
    apellido: 'García',
    usuario: 'ana2025'
  },
  _previousDataValues: {
    nombre: 'Ana',
    apellido: 'García',
    usuario: 'ana2025',
    id: 1,
    fecha_alta: 2025-04-17T18:03:46.207Z
  },
  _unique: 1,
  _changed: Set(0) {},
  _options: {
    isNewRecord: true,
    _schema: null,
    _schemaDelimiter: '',
    attributes: undefined,
    include: undefined,
    raw: undefined,
    silent: undefined
  },
  isNewRecord: false
}
philip@hpbook-Philip ~ ]$ main = ?4 ~1
```

Donde podemos revisar varias cosas

- En primer lugar ✓ Conexión establecida con la base de datos que nos puede servir como parámetro de que está todo Ok con la BD. Una buena prueba podría ser borrar o renombrar el archivo de datos y ver qué pasa.
- Luego vemos el Log de la sentencia que el **framework sequelize** ejecutó hacia la base de datos


```
Executing (default): INSERT INTO ``usuarios``(``id``,``nombre``,``apellido``,``usuario``,``fecha_alta``) VALUES (NULL,$1,$2,$3,$4);
```

 este log se puede configurar en la creación del objeto **sequelize** en el archivo **db.js**.
- Finalmente vemos la salida del objeto que devolvió el ORM donde podemos observar en primer lugar que la propiedad id que no enviamos a la base de datos contiene el id que se generó al insertar y también que la propiedad fecha contiene el valor por defecto que se le asignó al ejecutarse la operación.

A partir de aquí hay muchas más pruebas que se pueden hacer como trabajar con otros tipos de datos o con mayor cantidad de valores pero con esto tenemos una primera aproximación.

4.4.2 Esquema con **sequelize.define()**

Por otro lado también podemos seguir el esquema que utiliza la función **sequelize.define()** y en ese caso, definiremos el modelo **Usuario** con algunas propiedades usando el método **sequelize.define()**. Podríamos pasar algunos parámetros más como tercer argumento del método **define** pero por ahora veremos la forma más simple. La definición de modelos admite otra alternativa sintáctica basada en clases. Ilustraremos ésta en el ejercicio paso a paso que viene en el siguiente punto.

```
const { Sequelize, Model, DataTypes } = require("sequelize");

const sequelize = new Sequelize({
  dialect: "sqlite",
  storage: "path/to/database.sqlite",
});

const Usuario = sequelize.define("Usuario", {
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true
  },
  nombre: { type: DataTypes.STRING },
  apellido: { type: DataTypes.STRING },
  usuario: { type: DataTypes.STRING },
  fecha_alta: { type: DataTypes.DATE, defaultValue: DataTypes.NOW }
});
```

Se puede ver más sobre definir modelos [en la documentación de Sequelize](#)

4.5. Validaciones y restricciones

Sequelize permite implementar una serie de **validaciones** y **restricciones** para mantener la consistencia de los datos y asegurar las reglas de nuestro modelo de negocio. La diferencia entre estos conceptos radica en que las **validaciones** son establecidas a nivel de ORM e implementadas en JavaScript y por ende, además de usar las **validaciones** que **Sequelize** incluye, podemos implementar nuestras propias validaciones. Si una validación falla, los datos directamente no se enviarán a la Base de Datos. Las **restricciones** por otro lado, son implementadas en la Base de Datos, **Sequelize** enviará los datos al motor de base de datos y es éste quien se encargará de que las restricciones se cumplan.

4.5.1. Restricción `unique`

Permite asegurarnos que los valores que toma este campo son únicos.

```
usuario: {
  type: DataTypes.TEXT,
  unique: true
},
```

4.5.2. Permitir o no permitir nulos

Según especifiquemos el valor `true` o `false` para la clave `allowNull` el modelo admitirá o no valores nulos para ese atributo.

```
usuario: {  
    type: DataTypes.TEXT,  
    allowNull: false,  
},
```

Este caso es una combinación de validación y restricción.

4.5.3. Validadores por atributo

Mediante estos validadores que pueden ser los que se incluyen (implementados por [validator.js](#)) o validadores personalizados se pueden establecer una serie bastante amplia de validaciones.

Podés ver más detalles de los validadores disponibles [en la documentación de Sequelize](#), pero a modo de ejemplo veamos la especificación de un validador para el campo `genero` que permite solo dos valores `F, M`

```
genero: {  
    type: DataTypes.TEXT,  
    allowNull: false,  
    isIn: {  
        args: [['F', 'M']],  
        msg: "Debe ser (F)emenino o (M)asculino"  
    }  
},
```

4.5.4 Cómo queda nuestro ejemplo de usuario con algunas validaciones

Agreguemos algunas restricciones y validaciones a nuestro modelo:

```
// modelos/Usuario.js  
import { DataTypes, Model } from 'sequelize';  
import sequelize from '../db.js';  
  
class Usuario extends Model {}  
  
Usuario.init({  
    id: {  
        type: DataTypes.INTEGER,  
        autoIncrement: true,  
        primaryKey: true  
    },  
    nombre: {  
        type: DataTypes.STRING,  
        allowNull: false  
    },  
    apellido: {  
        type: DataTypes.STRING,  
    },
```

```

    allowNull: false
  },
  usuario: {
    type: DataTypes.STRING,
    allowNull: false, // Restricción a nivel DB
    unique: true, // Restricción de unicidad
    validate: {
      len: [4, 20], // Validación: longitud mínima y máxima
      is: /^[a-zA-Z0-9_-]+$/ // Validación: solo caracteres alfanuméricos y guiones
    }
  },
  email: {
    type: DataTypes.STRING,
    allowNull: false,
    validate: {
      isEmail: true // Validación de formato de correo electrónico
    }
  },
  genero: {
    type: DataTypes.STRING,
    allowNull: false,
    validate: {
      isIn: {
        args: [['F', 'M']],
        msg: 'Debe ser F o M'
      }
    }
  },
  {
    sequelize,
    modelName: 'Usuario',
    tableName: 'usuarios',
    timestamps: false
  });
}

export default Usuario;

```

⚠ Si una validación falla, Sequelize no enviará el dato a la base de datos y lanzará una excepción de validación. En cambio, si una restricción (como allowNull: false) se viola desde una fuente externa (por ejemplo, SQL crudo), será la base de datos la que generará el error.

```

// app.js
import sequelize from './db.js';
import Usuario from './modelos/usuario.js';

(async function main() {
  try {
    await sequelize.authenticate();
    console.log('✓ Conexión establecida con la base de datos');
  } catch (error) {

```

```
    console.error('✖ Error al conectar con la base de datos:', error);
}

await Usuario.sync(); // crea la tabla si no existe

// Crear un usuario válido (solo si no existe ya)
try {
    console.log("Primer intento de crear el usuario 'ana2025'");
    const nuevo = await Usuario.create({
        nombre: 'Ana',
        apellido: 'García',
        usuario: 'ana2025',
        email: 'ana@mail.com',
        genero: 'F'
    });
    console.log('✓ Usuario creado:', nuevo.usuario);
} catch (error) {
    console.error('✖ Error al crear usuario inicial:', error.errors?.[0]?.message || error.message);
}

// Intento de crear un usuario que ya existe
try {
    console.log("Segundo intento de crear el mismo usuario 'ana2025'");
    const nuevo = await Usuario.create({
        nombre: 'Ana',
        apellido: 'García',
        usuario: 'ana2025',
        email: 'ana@mail.com',
        genero: 'F'
    });
    console.log('✓ Usuario creado:', nuevo.usuario);
} catch (error) {
    console.error('✖ Error al crear usuario inicial:', error.errors?.[0]?.message || error.message);
}

// Intento fallido por validación (usuario con caracteres inválidos)
try {
    await Usuario.create({ nombre: 'Pepe', apellido: 'López', usuario: 'pe!', email: 'pepe@mail.com', genero: 'M' });
} catch (error) {
    console.error('🔴 Validación fallida:', error.errors?.[0]?.message);
}

// Intento fallido por restricción (usuario ya existente)
try {
    await Usuario.create({ nombre: 'Ana', apellido: 'García', usuario: 'ana2025', email: 'otraana@mail.com', genero: 'F' });
} catch (error) {
    console.error('🚫 Restricción violada:', error.errors?.[0]?.message || error.message);
}
})();
```

Nuevamente, si ejecutamos esta nueva prueba del paso04-ES6, obtendremos la siguiente salida:

Notar que el usuario ana2025 se puede crear nuevamente porque he cambiado la conexión de base de datos para que sea una base de datos en memoria mientras avanzamos con pruebas.

```
[ philip@hpZbook-Philip ~ % 8 ~1 ]
$ npm start

> crud_usuarios@1.0.0 start
> node app.js

SQL ejecutado:
SELECT
  1 + 1 AS result
✓ Conexión establecida con la base de datos

SQL ejecutado:
SELECT
  name
FROM
  sqlite_master
WHERE
  type = 'table'
  AND name = 'usuarios';

SQL ejecutado:
CREATE TABLE IF NOT EXISTS `usuarios` (
  `id` INTEGER PRIMARY KEY AUTOINCREMENT,
  `nombre` VARCHAR(255) NOT NULL,
  `apellido` VARCHAR(255) NOT NULL,
  `usuario` VARCHAR(255) NOT NULL UNIQUE,
  `email` VARCHAR(255) NOT NULL,
  `genero` VARCHAR(255) NOT NULL
);

SQL ejecutado:
PRAGMA INDEX_LIST (`usuarios`)

SQL ejecutado:
PRAGMA INDEX_INFO (`sqlite_autoindex_usuarios_1`)
Primer intento de crear el usuario 'ana2025'

SQL:
INSERT INTO `usuarios` (`id`, `nombre`, `apellido`, `usuario`, `email`, `genero`) VALUES (NULL,$1,$2,$3,$4,$5);
✓ Usuario creado: ana2025
Segundo intento de crear el mismo usuario 'ana2025'

SQL:
INSERT INTO `usuarios` (`id`, `nombre`, `apellido`, `usuario`, `email`, `genero`) VALUES (NULL,$1,$2,$3,$4,$5);
✖ Error al crear usuario inicial: usuario must be unique

Intento de crear un usuario inválido
● Validación fallida: Validation is on usuario failed

Intento de crear un usuario inválido
✖ Restricción violada: Debe ser F o M o X
philip@hpZbook-Philip ~ % 8 ~1 ]
```

Mucho para ver por aquí:

- En primer lugar podemos ver toda la estructura de conexión con la base en memoria y la creación de la tabla Usuarios que genera la sentencia `await Usuario.sync();` y las sentencias asociadas.
- Luego comenzamos con los intentos de ejecución, el primero que es el que veníamos utilizando resulta feliz y efectivamente crea el usuario.
- El segundo intento con el mismo nombre de usuario `ana20205`, falla e indica el error donde el atributo usuario debe ser único.
- Luego los intentos siguientes fallan por la validación en los caracteres permitidos en el nombre de usuario y el último por la validación en el valor de genero. **Nota: he realizado un agregado al archivo 'db.js' para que los logs de sequelize se impriman de forma legible, pueden revisar esa mejora en los ejemplos en pap_sequelize.

5 Inicialización de la base de datos

Para facilitar la carga de datos de prueba y trabajar con operaciones de consulta, es conveniente separar la lógica de inicialización en archivos reutilizables.

Para ello vamos a crear el archivo inicializar-bd.js que tendrá por función utilizar los archivos seeder que vayamos agregando al proyecto, creamos el archivo usuarioSeeder.js con la creación de datos en la tabla de usuarios y finalmente los utilizamos para la inicialización.

Los podemos utilizar ejecutando la inicialización con un script independiente en el package.json o bien importando la función desde el archivo seeder en el app.js si estamos trabajando con una base en memoria para pruebas. En el caso del paso 5 de las pruebas en la versión ES6, como estamos usando una base en memoria vamos a hacerlo desde el app.js.

```

crud_usuarios
├── modelos
│   └── Usuario.js      # Definición del modelo Usuario
├── scripts
│   └── inicializar-bd.js    # Script ejecutable para inicializar la base
(npm run init-db)
└── usuariosSeeder.js      # Función exportada que inserta los usuarios
├── datos
│   └── db.sqlite          # Archivo físico de la base de datos (si no se
usa en memoria)
└── db.js                  # Configuración de Sequelize y conexión a la
base
├── app.js                 # Script principal de la aplicación
├── package.json            # Configuración del proyecto y scripts
└── .gitignore              # Ignorar node_modules, datos temporales, etc.

```

5.1 Creación de filas en lote

El método `.bulkCreate()` de Sequelize permite crear múltiples registros en una sola operación. Es especialmente útil para inicializar una base de datos con datos de prueba o para realizar cargas masivas.

A diferencia del método `.create()` que recibe un único objeto con datos, `.bulkCreate()` espera un array de objetos, donde cada uno representa una fila a insertar.

→ SOON Más adelante veremos en profundidad cómo crear objetos individuales y cómo trabajar con instancias del modelo. En este punto usamos bulkCreate para centrarnos en el volumen de carga y la inicialización rápida.

Archivo `scripts/usuariosSeeder.js`

```

import sequelize from '../db.js'; // Importar Sequelize desde el archivo db.js
import { enableDbLog, disableDbLog } from '../db.js'; // Importar la función para
habilitar el logging
import Usuario from '../modelos/usuario.js';

```

```

export async function inicializarUsuarios() {
  const cantidad = await Usuario.count();
  if (cantidad > 0) {
    console.log('⚠ Ya existen usuarios, no se inicializa.');
    return;
  }
  disableDbLog(); // Desactivar el logging de Sequelize
  console.log('🌐 Inicializando usuarios... (Log Deshabilitado)');
  await Usuario.bulkCreate([
    { nombre: 'Ana', apellido: 'García', usuario: 'ana2025', email: 'ana@mail.com', genero: 'F' },
    { nombre: 'Luis', apellido: 'Martínez', usuario: 'lmartinez', email: 'luis@mail.com', genero: 'M' },
    { nombre: 'Sofía', apellido: 'López', usuario: 'sofial', email: 'sofia@mail.com', genero: 'F' },
    { nombre: 'Julían', apellido: 'Ruiz', usuario: 'jruiz', email: 'julian@mail.com', genero: 'M' },
    { nombre: 'Marta', apellido: 'Fernández', usuario: 'mfer', email: 'marta@mail.com', genero: 'F' },
    { nombre: 'Carlos', apellido: 'Domínguez', usuario: 'cdom', email: 'carlos@mail.com', genero: 'M' },
    { nombre: 'Lucía', apellido: 'Silva', usuario: 'lucas', email: 'lucia@mail.com', genero: 'F' },
    { nombre: 'Pedro', apellido: 'Sosa', usuario: 'psosa', email: 'pedro@mail.com', genero: 'M' },
    { nombre: 'Valentina', apellido: 'Paz', usuario: 'vpaz', email: 'valentina@mail.com', genero: 'F' },
    { nombre: 'Andrés', apellido: 'Cruz', usuario: 'acruz', email: 'andres@mail.com', genero: 'M' }
  ]);
  enableDbLog(); // Reactivar el logging de Sequelize

  console.log('✅ Usuarios inicializados');
}

```

Entre otras novedades podemos encontrar aquí que se utiliza el método `.count()` de sequelize para obtener la cantidad de filas de la tabla y realizar un control de la existencia de filas antes de inicializar.

📄 Archivo `scripts/inicializar-bd.js`

La idea del archivo `inicializar-bd.js` es la de generar un script independiente que permita ejecutar la inicialización de la base de datos que por ahora solo tiene la tabla de usuarios pero que a medida que avancemos irá agregando otras tablas y sus inicializaciones.

```
// inicializar-bd.js
```

```
import sequelize from '../db.js';
import { inicializarUsuarios } from './usuariosSeeder.js';

(async function main() {
    try {
        await sequelize.authenticate();
        console.log('✓ Conexión establecida con la base de datos');
    } catch (error) {
        console.error('✗ Error al conectar con la base de datos:', error);
    }
    try {
        // Sincronizar la base de datos con el modelo
        // await sequelize.sync(); // crea la tabla si no existe, al agregar {
force: true } se eliminan los datos existentes y se reinicializa la secuencia
        await sequelize.sync({ force: true });
        // Luego llamamos a la función que inicializa los usuarios
        await inicializarUsuarios();

        console.log('● Base de datos inicializada desde script');
    } catch (error) {
        console.error('✗ Error al inicializar la base de datos:', error);
    }

    process.exit();
})();
```

📄 package.json

Agregamos un script más en el archivo `package.json` para lograr ejecutar la inicialización de la base de datos de forma independiente.

```
"scripts": {
    "start": "node app.js",
    "init-db": "node scripts/inicializar-bd.js"
}
```

Esta estructura permite inicializar la base fácilmente tanto desde línea de comandos como desde la aplicación en tiempo de ejecución.

Una vez lograda esta versión podemos hacer una prueba de la inicialización para ver que todo funcione... La ejecución sería con el siguiente comando:

```
npm run init-db
```

📄 La salida obtenida será:

```
[ philip@hpZbook-Philip ~ ]$ npm run init-db
> crud_usuarios@1.0.0 init-db
> node scripts/inicializar-bd.js

SQL ejecutado:
SELECT
  1 + 1 AS result
✓ Conexión establecida con la base de datos

SQL ejecutado:
DROP TABLE IF EXISTS `usuarios`;

SQL ejecutado:
PRAGMA foreign_keys = OFF

SQL ejecutado:
DROP TABLE IF EXISTS `usuarios`;

SQL ejecutado:
PRAGMA foreign_keys = ON

SQL ejecutado:
DROP TABLE IF EXISTS `usuarios`;

SQL ejecutado:
CREATE TABLE IF NOT EXISTS `usuarios` (
  `id` INTEGER PRIMARY KEY AUTOINCREMENT,
  `nombre` VARCHAR(255) NOT NULL,
  `apellido` VARCHAR(255) NOT NULL,
  `usuario` VARCHAR(255) NOT NULL UNIQUE,
  `email` VARCHAR(255) NOT NULL,
  `genero` VARCHAR(255) NOT NULL
);

SQL ejecutado:
PRAGMA INDEX_LIST(`usuarios`)

SQL ejecutado:
PRAGMA INDEX_INFO(`sqlite_autoindex_usuarios_1`)

SQL ejecutado:
SELECT
  count(*) AS `count`
FROM
  `usuarios` AS `Usuario`;
● Inicializando usuarios... (Log Deshabilitado)
✓ Usuarios inicializados
● Base de datos inicializada desde script
[ philip@hpZbook-Philip ~ ]$ main = ?12 ~1
```

Dónde luego de observar las acciones que se llevaron a cabo para la creación de la tabla vacía que fueron provocadas por `sequelize.sync({ force: true })`; podemos observar que se inicializó la base de datos. **Nota:** vale aclarar que he desactivado el log para las acciones bulk mediante `disableDbLog()` que esencialmente ejecuta `sequelize.options.log = false;` para evitar un log de todos los inserts para cada uno de los valores. Y luego se vuelve a activar mediante `enableDbLog()` que lo vuelve a la estructura original. Estas funciones se puede observar en el archivo db.js desde el paso 05 en adelante.

Archivo app.js

Ahora bien, si además queremos comprobar la creación de las filas en la tabla de usuarios, podemos reaprovechar el método `.count()` que ya vimos en uso en el control de la existencia de filas previo a la inicialización. Un Script principal que solo inicialize los usuarios y luego controle en base a la cantidad de usuarios incializados podría ser como sigue:

```
// app.js
import sequelize from './db.js';
import { inicializarUsuarios } from './scripts/usuariosSeeder.js';
```

```
import Usuario from './modelos/usuario.js';

(async function main() {
  try {
    await sequelize.authenticate();
    console.log('✓ Conexión establecida con la base de datos');
  } catch (error) {
    console.error('✗ Error al conectar con la base de datos:', error);
  }

  try {
    // Crea todas las tablas desde cero y reinicia los autoincrementos
    await sequelize.sync({ force: true });

    // Carga de datos de prueba en tabla usuarios
    await inicializarUsuarios();

    console.log('🟡 Base de datos inicializada desde script');
  } catch (error) {
    console.error('✗ Error al inicializar la base de datos:', error);
  }

  try {
    // Comprobar la inicialización de los usuarios
    const cantidad = await Usuario.count();
    if (cantidad > 0) {
      console.log(`☑ Efectivamente se crearon ${cantidad} usuarios.`);
    }
    else {
      console.error('✗ No se encontraron usuarios en la base de datos.');
    }
  } catch (error) {
    console.error('✗ Error al contar los usuarios:', error);
  }
})();
```

Que al ejecutar con `npm start` como ya venimos haciendo con el script principal generará la siguiente salida por consola:

```
[ philip@hpZbook-Philip ~ ]$ npm start
> crud_usuarios@1.0.0 start
> node app.js

SQL ejecutado:
SELECT
  1 + 1 AS result
✓ Conexión establecida con la base de datos

SQL ejecutado:
DROP TABLE IF EXISTS `usuarios`;

SQL ejecutado:
PRAGMA foreign_keys = OFF

SQL ejecutado:
DROP TABLE IF EXISTS `usuarios`;

SQL ejecutado:
PRAGMA foreign_keys = ON

SQL ejecutado:
DROP TABLE IF EXISTS `usuarios`;

SQL ejecutado:
CREATE TABLE IF NOT EXISTS `usuarios` (
  `id` INTEGER PRIMARY KEY AUTOINCREMENT,
  `nombre` VARCHAR(255) NOT NULL,
  `apellido` VARCHAR(255) NOT NULL,
  `usuario` VARCHAR(255) NOT NULL UNIQUE,
  `email` VARCHAR(255) NOT NULL,
  `genero` VARCHAR(255) NOT NULL
);

SQL ejecutado:
PRAGMA INDEX_LIST (`usuarios`)

SQL ejecutado:
PRAGMA INDEX_INFO (`sqlite_autoindex_usuarios_1`)

SQL ejecutado:
SELECT
  count(*) AS `count`
FROM
  `usuarios` AS `Usuario`;
● Inicializando usuarios... (Log Deshabilitado)
✓ Usuarios inicializados
● Base de datos inicializada desde script

SQL ejecutado:
SELECT
  count(*) AS `count`
FROM
  `usuarios` AS `Usuario`;
✓ Efectivamente se crearon 10 usuarios.
[ philip@hpZbook-Philip ~ ]$ main ~ ?13 ~1
```

Donde se pueden observar las mismas acciones que en la inicialización además de la acción de conteo para el control de filas luego de la inicialización.

6 Acciones ORM con Sequelize

Bien, ya hemos realizado 2 de las 3 tareas necesarias cuando se trabaja con ORM:

1. Configuración y conexión a la base de datos, como lo hicimos al instalar las dependencias y luego en el archivo `db.js`, cabe aclarar que podríamos agregar varias configuraciones extra pero por ahora con esto nos alcanza.
2. Mapeo de relaciones en objetos de nuestra aplicación, como lo hicimos al estructurar el mapeo de la tabla Usuarios de la base de datos.
3. Y lo que nos queda es utilizar el framework para interactuar con esa base de datos *sin tener que escribir sentencias sql* para crear, obtener, modificar o eliminar datos de la base de datos.

Y a esto último nos vamos a decir ahora...

6.1. Recuperar datos

Al trabajar con Sequelize, la recuperación de datos es flexible y potente, permitiéndote realizar desde consultas simples hasta operaciones más complejas mediante el uso de operadores y funciones. Aquí exploraremos varias formas de recuperar datos que pueden ser útiles en distintos escenarios.

Recuperar todos los registros

Para recuperar todos los registros de una tabla, usamos el método `findAll()` sin parámetros, cabe aclarar que este procedimiento hay que realizarlo con mucha cautela en el código puesto que cualquier tabla que tenga gran cantidad de filas provocaría un excesivo uso de memoria al cargar todo ese contenido como respuesta de `findAll()`

De todos modos si fuera el caso podemos llevar a cabo esta acción mediante:

```
const usuarios = await Usuario.findAll();
```

Vale hacer notar el uso de `await` ya que la búsqueda de filas es una acción que bloque E/S por lo que Node.js la va a realizar de forma asincrónica.

Recuperar un registro específico

Para buscar un registro específico por clave primaria o cualquier otro criterio, utilizamos `findOne()`, es importante mencionar que debemos garantizar que el criterio utilizado devuelva una y solo una fila de la tabla ya que el método devuelve un objeto y no una colección de estos:

```
const usuario = await Usuario.findOne({ where: { id: 5 } });
```

En este caso estamos obteniendo el objeto `Usuario` que corresponde con la fila de la tabla que tiene la clave primaria igual a `5`.

Uso de operadores para filtrar datos

Sequelize proporciona varios operadores para realizar consultas más precisas, como el operador `like` para coincidencias parciales:

```
const usuarios = await Usuario.findAll({
  where: {
    apellido: { [Op.like]: '%ez' } // para traer todos los apellidos terminados en
    'ez'
  }
});
```

Notar que el símbolo de porcentaje % se utiliza como comin para representar cualquier cadena previa, es decir; cualquier cadena que termine con las letras "ez" cumple la condicion. Resultado esperado de la consulta, serían cuatro usuarios: 2 | Luis Martínez 3 | Sofía López 5 | Marta Fernández 6 | Carlos Domínguez

Combinar criterios de búsqueda

Puedes combinar varios criterios usando operadores lógicos como `and` y `not` para realizar consultas más complejas:

```
const usuarios = await Usuario.findAll({
  where: [
    [Op.and]: [
      { apellido: { [Op.like]: '%ez' } },
      { [Op.not]: { id: [1, 2, 3] } }
    ]
  ]
});
```

Ahora agregamos a la condición anterior un elemento extra de validación con `and` que es que el id no esté en el conjunto [1, 2, 3]. Resultado esperado de la consulta, serían dos usuarios ya que los dos primeros del resultado anterior no cumplen esta segunda condición: 5 | Marta Fernández 6 | Carlos Domínguez

📋 Principales operadores de `where` y operadores lógicos en Sequelize

En primer lugar vamos a necesitar importar el módulo que contiene todos los operadores

```
import { Op } from 'sequelize';
```

Y luego expresamos en la siguiente tabla un resumen de los principales operadoes con su función y ejemplo de uso.

Categoría	Operador Sequelize	Equivalente SQL	Ejemplo de uso <code>where</code> :
Comparación	<code>Op.eq</code>	=	<code>{ id: { [Op.eq]: 1 } }</code>
	<code>Op.ne</code>	!=	<code>{ nombre: { [Op.ne]: 'Ana' } }</code>
	<code>Op.gt</code>	>	<code>{ edad: { [Op.gt]: 18 } }</code>
	<code>Op.gte</code>	>=	<code>{ edad: { [Op.gte]: 18 } }</code>
	<code>Op.lt</code>	<	<code>{ edad: { [Op.lt]: 30 } }</code>
	<code>Op.lte</code>	<=	<code>{ edad: { [Op.lte]: 30 } }</code>

Categoría	Operador Sequelize	Equivalente SQL	Ejemplo de uso where:
Rango	Op.between	BETWEEN	{ edad: { [Op.between]: [18, 30] } }
	Op.notBetween	NOT BETWEEN	{ edad: { [Op.notBetween]: [18, 30] } }
Inclusión	Op.in	IN	{ id: { [Op.in]: [1, 2, 3] } }
	Op.notIn	NOT IN	{ id: { [Op.notIn]: [1, 2, 3] } }
Texto	Op.like	LIKE	{ nombre: { [Op.like]: '%ina' } }
	Op.notLike	NOT LIKE	{ nombre: { [Op.notLike]: '%ina' } }
Nulo	Op.is	IS [NOT] NULL	{ email: { [Op.is]: null } }
Lógico	Op.and	AND	{ [Op.and]: [{ edad: 25 }, { genero: 'F' }] }
	Op.or	OR	{ [Op.or]: [{ nombre: 'Ana' }, { nombre: 'Luis' }] }
Negación	Op.not	NOT (...)	{ [Op.not]: { id: [1, 2, 3] } }

Podemos aplicar un amplio espectro de operadores ([Ver una lista exhaustiva aquí](#)), **es sumamente importante investigar y ampliar el dominio sobre estos operadores puesto que en las evaluaciones vamos a trabajar sobre los condicionantes a la hora de recuperar datos**

Paginación de resultados

Para manejar grandes volúmenes de datos, puedes implementar paginación en tus consultas, paginar los resultados consiste en solicitar al servidor los primeros n resultados que cumplen la condición en primera instancia para luego pedir los siguientes n resultados que cumplen la condición en una petición diferente y así se puede seguir.

Esto permite que no se sobrecargue la memoria del servidor para tablas muchas filas. La estructura para pedirle a sequelize que recupere la página `pagina` con cantidad de filas igual a `tamanoPagina` es la siguiente:

```
const pagina = 2;
const tamanoPagina = 10;
const usuarios = await Usuario.findAll({
  offset: (pagina - 1) * tamanoPagina,
  limit: tamanoPagina
});
```

En este caso `offset` indica el punto de partida de la recuperación de filas y `limit` la cantidad de elementos a recuperar desde ese punto de partida.

Ordenar resultados

Para ordenar los resultados de tus consultas, puedes usar el atributo `order` del objeto de configuración de `findAll` y enviarle un array de arrays con los nombres de columna y la dirección de ordenamiento para cada una:

```
const usuariosOrdenados = await Usuario.findAll({
  where: {
    [Op.and]: [
      { apellido: { [Op.like]: '%ez' } },
      { [Op.not]: { id: [1, 2, 3] } }
    ]
  },
  order: [['apellido', 'ASC']]
});
```

Notar que en este caso el resultado que habíamos visto: 5 | Marta Fernández 6 | Carlos Domínguez cambiaría a: 6 | Carlos Domínguez 5 | Marta Fernández Por el ordenamiento alfabético ascendente por apellido.

Seleccionar atributos específicos

Puede ocurrir que en algún caso necesitemos obtener solo ciertos atributos de la tabla y no todos, cabe aclarar que esta práctica debe ser utilizada con precaución ya que los objetos obtenidos estarán incompletos pero puede darse la situación donde tenga sentido. Por ejemplo suele utilizarse cuando una de las columnas de la tabla contiene una imagen o un texto muy grande para evitar cargar en memoria estos datos que luego se buscan para el objeto para el que lo necesitamos.

Si solo necesitas algunos campos específicos de los registros, se puede especificarlos en el atributo `attributes` la lista de los atributos a cargar desde la tabla:

```
const usuarios = await Usuario.findAll({
  attributes: ['nombre', 'apellido'],
  where: {
    [Op.and]: [
      { apellido: { [Op.like]: '%ez' } },
      { [Op.not]: { id: [1, 2, 3] } }
    ]
  },
  order: [['apellido', 'ASC']]
});
```

Vale revisar aquí la estructura de los objetos resultantes **NO** será la que esperamos de un objeto `Usuario` sino simplemente la que se compone de los atributos solicitados. Mucho cuidado aquí si con estos objetos después se pretende realizar una acción de modificación hacia la base de datos. Por ejemplo, para la consulta del modelo el resultado será:

```
[  
  { nombre: 'Carlos', apellido: 'Domínguez' },  
  { nombre: 'Marta', apellido: 'Fernández' }  
]
```

Funciones de agregación

Las funciones de agregación son útiles para obtener datos resumidos, como contar usuarios, sumar valores, o encontrar el valor máximo en un conjunto de datos:

```
const totalUsuarios = await Usuario.count();  
const maxEdad = await Usuario.max('edad');  
const sumSalarios = await Usuario.sum('salario');
```

Ya hemos visto el uso de `.count()`, del mismo modo hay otras operaciones que pueden llevar a cabo agregación entre todas las filas de la tabla o entre las filas que correspondan con el filtro indicado en `where`

Sequelize proporciona funciones de agregación que permiten realizar operaciones como conteo, suma, promedio, máximo y mínimo sobre los datos de un modelo. Estas funciones se utilizan como métodos estáticos del modelo.

A continuación se muestra una tabla con las funciones más comunes:

Función	Método Sequelize	Descripción	Ejemplo de uso
Conteo	<code>Model.count()</code>	Cuenta la cantidad de registros	<code>await Usuario.count();</code>
Suma	<code>Model.sum('campo')</code>	Suma los valores de un campo numérico	<code>await Usuario.sum('edad');</code>
Promedio	<code>Model.avg('campo')</code>	Calcula el promedio de los valores numéricos	<code>await Usuario.avg('edad');</code>
Valor máximo	<code>Model.max('campo')</code>	Retorna el valor máximo del campo	<code>await Usuario.max('edad');</code>
Valor mínimo	<code>Model.min('campo')</code>	Retorna el valor mínimo del campo	<code>await Usuario.min('edad');</code>

💡 Estas funciones devuelven un valor único (no un arreglo de instancias), y son especialmente útiles para estadísticas o reportes.

También se pueden combinar con condiciones usando `where`:

```
import { Op } from 'sequelize';
```

```
const mujeresMayores = await Usuario.count({
  where: {
    genero: 'F',
    edad: { [Op.gte]: 18 }
  }
});
```

💡 El uso de corchetes [Op.gte] funciona igual que con filtros tradicionales: representa un nombre de propiedad dinámico evaluado en tiempo de ejecución.

Anexo 4.6 Modelado de relaciones

Para poder revisar nuevas alternativas de sequelize tenemos que agregar algo más de estructura a la base de datos de nuestro caso de estudio, y por ende, también tendremos que agregar nuevos modelos.

Modelando Perfiles

Comencemos por agregar el modelo del Perfil del usuario.

📄 Archivo `modelos/Perfil.js`

```
// Perfil.js

import { DataTypes, Model } from 'sequelize';
import sequelize from '../db.js';

class Perfil extends Model {}

Perfil.init({
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true
  },
  nombre: {
    type: DataTypes.STRING,
    allowNull: false
  },
  responsabilidades: {
    type: DataTypes.TEXT
  }
}, {
  sequelize,
  modelName: 'Perfil',
  tableName: 'perfiles',
  timestamps: false
});

export default Perfil;
```

Y luego tenemos que agregar su seeder a la inicialización de la base de datos.

Archivo scripts/perfilesSeeder.js

```
import Perfil from '../modelos/Perfil.js';

export async function inicializarPerfiles() {
    const cantidad = await Perfil.count();
    if (cantidad > 0) {
        console.log('⚠ Ya existen perfiles, no se inicializa.');
        return;
    }

    await Perfil.bulkCreate([
        { nombre: 'Analista Funcional', responsabilidades: 'Relevar requisitos, documentar funcionalidades, interactuar con el cliente.' },
        { nombre: 'Desarrollador', responsabilidades: 'Implementar funcionalidades, corregir errores, participar en revisiones de código.' },
        { nombre: 'Scrum Master', responsabilidades: 'Facilitar el proceso ágil, remover impedimentos, liderar las ceremonias Scrum.' },
        { nombre: 'Product Owner', responsabilidades: 'Definir backlog del producto, priorizar funcionalidades, representar al cliente.' },
        { nombre: 'Tester', responsabilidades: 'Diseñar casos de prueba, ejecutar pruebas manuales y automáticas, reportar bugs.' },
        { nombre: 'Stakeholder', responsabilidades: 'Interesado en el producto o proyecto, puede aportar requisitos o feedback.' }
    ]);

    console.log('✅ Perfiles inicializados');
```

Actualizando el modelo de Usuario

Además necesitamos actualizar el modelo del **Usuario** para incluir la relación con el perfil, para ello, tenemos que agregar los siguientes elementos en el modelo de Usuario y agregar al modelo del **Perfil** la relación:

```
// usuario.js
import Perfil from './perfil.js';

//Resto de código del modelo

// Definición de la relación entre Usuario y Perfil
// Un usuario pertenece a un perfil
Usuario.belongsTo(Perfil, {
    as: 'perfil', // nombre que toma el atributo que contiene el objeto referido
    foreignKey: 'perfilId', // nombre del atributo que contiene el valor de la clave foránea
    field: 'perfil_id' // nombre del atributo en la base de datos que contiene la clave foránea
```

```
});  
  
// Definición de la relación uno a muchos entre Perfil y Usuario  
// Un perfil puede tener muchos usuarios  
Perfil.hasMany(Usuario, {  
  foreignKey: 'perfilId', // idem  
  field: 'perfil_id' // idem  
});
```

Con esto el modelo **Usuario** (archivo `modelos/usuario.js`) va a quedar como sigue:

```
// modelos/Usuario.js
import { DataTypes, Model } from 'sequelize';
import sequelize from '../db.js';
import Perfil from './perfil.js';

class Usuario extends Model {}

Usuario.init({
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true
  },
  nombre: {
    type: DataTypes.STRING,
    allowNull: false
  },
  apellido: {
    type: DataTypes.STRING,
    allowNull: false
  },
  usuario: {
    type: DataTypes.STRING,
    allowNull: false, // Restricción a nivel DB
    unique: true, // Restricción de unicidad
    validate: {
      len: [4, 20], // Validación: longitud mínima y máxima
      // is permite utilizar expresiones regulares para validar la cadena
      is: /^[a-zA-Z0-9_]+$/ // Validación: solo caracteres alfanuméricos y guiones bajos
    }
  },
  email: {
    type: DataTypes.STRING,
    allowNull: false,
    validate: {
      isEmail: true // Validación de formato de correo electrónico
    }
  },
})
```

```

genero: {
  type: DataTypes.STRING,
  allowNull: false,
  validate: {
    isIn: {
      args: [['F', 'M', 'X']],
      msg: 'Debe ser F o M o X'
    }
  }
},
{
  sequelize,
  modelName: 'Usuario',
  tableName: 'usuarios',
  timestamps: false
});

// Definición de la relación entre Usuario y Perfil
// Un usuario pertenece a un perfil
Usuario.belongsTo(Perfil, {
  as: 'perfil',
  foreignKey: 'perfilId',
  field: 'perfil_id'
});

// Definición de la relación uno a muchos entre Perfil y Usuario
// Un perfil puede tener muchos usuarios
PerfilhasMany(Usuario, {
  foreignKey: 'perfilId',
  field: 'perfil_id'
});

export default Usuario;

```

En este punto la estructura del proyecto quedaría como sigue a continuación:

```

crud_usuarios
├── modelos
│   ├── psuario.js          # Definición del modelo Usuario
│   └── perfil.js           # Modelo de Perfil con relación 1:N a Usuario
├── scripts
│   ├── inicializar-db.js   # Script ejecutable para inicializar toda la base
│   ├── usuariosSeeder.js    # Seeder para datos iniciales de usuarios
│   └── perfilesSeeder.js    # Seeder para inicializar perfiles (Scrum roles)
└── datos
    └── db.sqlite            # Base de datos SQLite persistente
        # Configuración de conexión Sequelize con log
db.js                                # Script principal de la aplicación
package.json                           # Configuración del proyecto y scripts npm
.gitignore                            # Ignora node_modules, /datos y otros archivos
formateado
temporales

```

Esta estructura puede encontrarse con los demás detalles en la versión paso06-ES6 del ejemplo que acompaña este apunte pap-sequelize

Es decir que tenemos la aplicación conectada y mapeada a una base de datos que tiene la siguiente estructura:

Diagrama ER

```
erDiagram
    direction RL
    PERFIL ||--o{ USUARIO : tiene
    USUARIO {
        int id
        string nombre
        string apellido
        string usuario
        string email
        string genero
        date fecha_alta
        int perfil_id
    }
    PERFIL {
        int id
        string nombre
        text responsabilidades
    }
```

Continuando 6 Operaciones con ORM

Con la estructura de Base de Datos y Proyecto actualizados a la nueva estructura podemos continuar con las pruebas.

6.2 Recuperar filas de múltiples tablas

Recuperar con relaciones

Ahora que nuestros modelos representan una Base de Datos con 2 tablas relacionadas entre sí, podemos recuperar datos de ellas en conjunto con sus objetos relacionados, aquí hay que tomarlo con calma.

Si realizamos la consulta normal de usuarios, la novedad va a ser la propiedad `perfilId` en los usuarios, por ejemplo, aprovechemos para utilizar la versión simplificada de `.findOne(...)` para buscar un objeto por su clave primaria (`.findById()`):

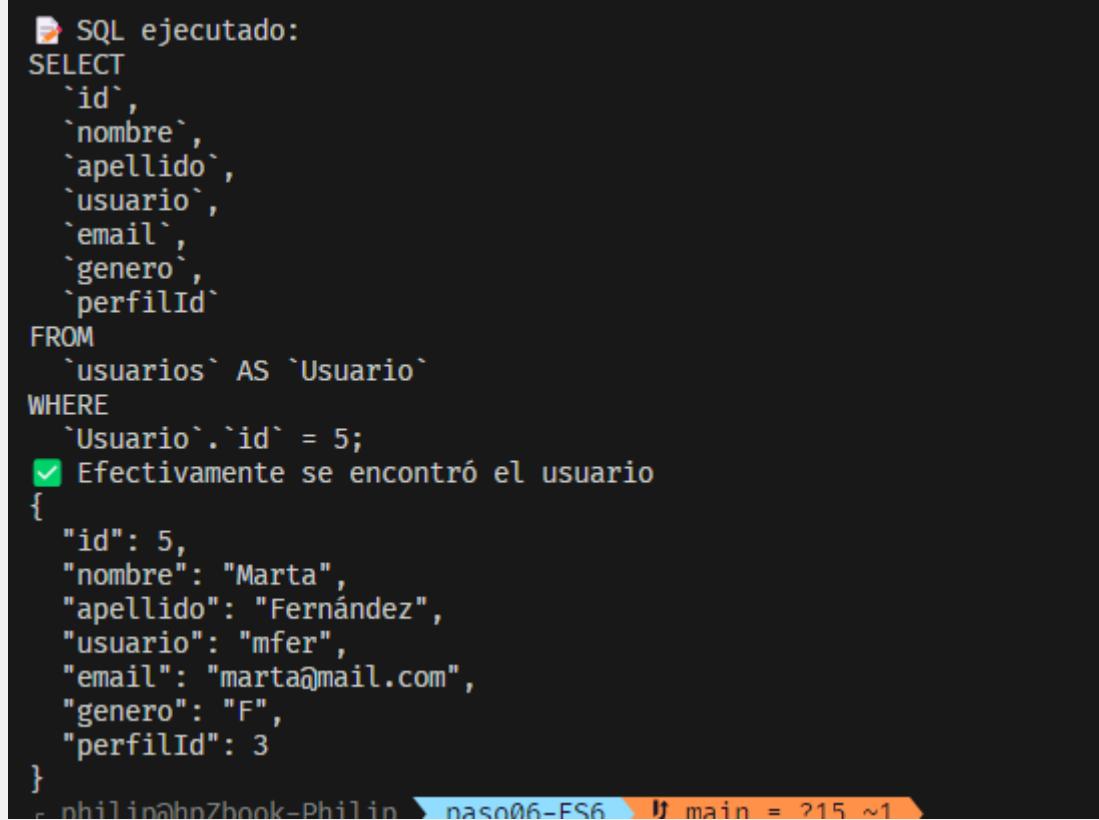
```
...
try {
    // Comprobar la inicialización de los usuarios
    const usuario = await Usuario.findById(5); // Busca el usuario con ID 5
    if (usuario) {
```

```

    console.log(`✅ Efectivamente se encontró el usuario
\n${JSON.stringify(usuario, null, 2)}`);
} else {
    console.error('❌ No se encontró el usuario en la base de datos.');
}
...

```

En este caso buscamos el usuario con clave primaria 5 y la salida obtenida sería:



SQL ejecutado:

```

SELECT
`id`,
`nombre`,
`apellido`,
`usuario`,
`email`,
`genero`,
`perfilId`
FROM
`usuarios` AS `Usuario`
WHERE
`Usuario`.`id` = 5;

```

✓ Efectivamente se encontró el usuario

```

{
  "id": 5,
  "nombre": "Marta",
  "apellido": "Fernández",
  "usuario": "mfer",
  "email": "marta@mail.com",
  "genero": "F",
  "perfilId": 3
}

```

philip@hnZbook-Philip ~ main = ?15 ~1

Pero acá viene lo interesante, podemos agregar la opción `include` para agregar directamente al objeto resultante el perfil asociado de acuerdo con la configuración, en el siguiente ejemplo modificamos `findByPk(...)` para que además de la clave incluya el objeto de configuración con la propiedad `include`:

```

try {
    // Comprobar la inicialización de los usuarios
    const usuario = await Usuario.findByPk(5,{
        include: [
            model: Perfil,
            as: 'perfil', // Alias definido en el modelo Usuario
            required: true, // Solo incluir usuarios con perfil asociado
        ]
    });
    // Busca el usuario con ID 5
    if (usuario) {
        console.log(`✅ Efectivamente se encontró el usuario
\n${JSON.stringify(usuario, null, 2)}`);
    } else {
        console.error('❌ No se encontró el usuario en la base de datos.');
    }
}

```

```
}
```

Ahora veamos como se modifica la consulta que sequelize envía a la base de datos y el objeto resultante:

```
SQL ejecutado:
SELECT
`Usuario`.`id`,
`Usuario`.`nombre`,
`Usuario`.`apellido`,
`Usuario`.`usuario`,
`Usuario`.`email`,
`Usuario`.`genero`,
`Usuario`.`perfilId`,
`perfil`.`id` AS `perfil.id`,
`perfil`.`nombre` AS `perfil.nombre`,
`perfil`.`responsabilidades` AS `perfil.responsabilidades`
FROM
`usuarios` AS `Usuario`
INNER JOIN `profiles` AS `perfil` ON `Usuario`.`perfilId` = `perfil`.`id`
WHERE
`Usuario`.`id` = 5;
✓ Efectivamente se encontró el usuario
{
  "id": 5,
  "nombre": "Marta",
  "apellido": "Fernández",
  "usuario": "mfer",
  "email": "marta@mail.com",
  "genero": "F",
  "perfilId": 3,
  "perfil": {
    "id": 3,
    "nombre": "Scrum Master",
    "responsabilidades": "Facilitar el proceso ágil, remover impedimentos, liderar las ceremonias Scrum."
  }
}
philip@hpZbook:~/Desktop/ES6$ ll main.js
```

Se puede observar la propiedad perfil con el objeto que incluye el id, el nombre y las responsabilidades del perfil asociado.

Y la cosa se pone interesante cuando extendemos esta opción a consultas que traen muchas filas, por ejemplo retomando la consulta con filtro y ordenamiento que habíamos trabajado antes, pero agregando la propiedad `include` para cargar los profiles:

```
...
try {

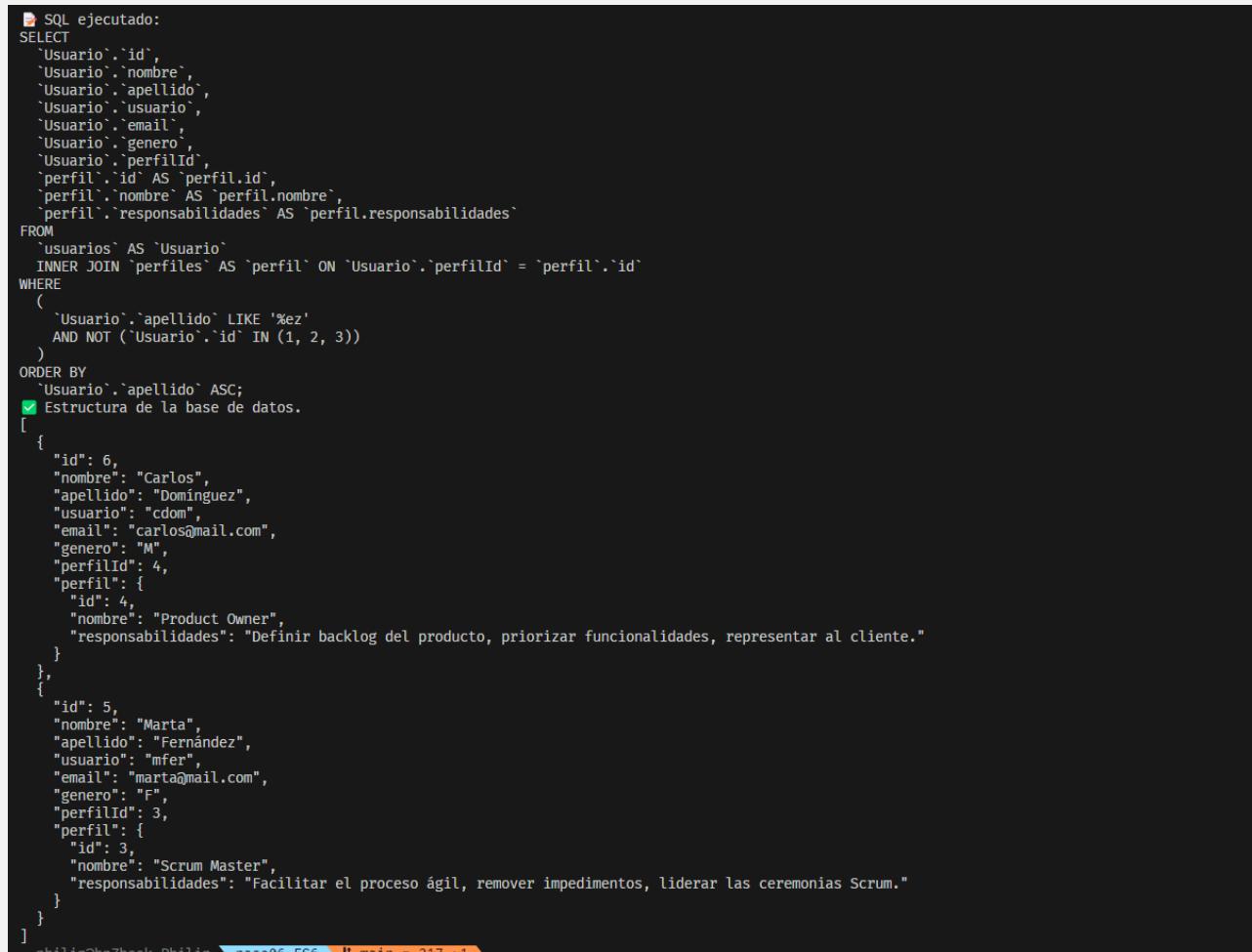
  const usuarios = await Usuario.findAll({
    where: {
      [Op.and]: [
        { apellido: { [Op.like]: '%ez' } },
        { [Op.not]: { id: [1, 2, 3] } }
      ]
    },
    order: [['apellido', 'ASC']],
    include: [
      {
        model: Perfil,
        as: 'perfil', // Alias definido en el modelo Usuario
        required: true, // Solo incluir usuarios con perfil asociado
      }
    ],
  });
  if (usuarios.length > 0) {
    console.log(`✓ Estructura de la base de datos.`);
    console.log(JSON.stringify(usuarios, null, 2));
  }
}
```

```

    }
    else {
        console.error('✖ Ops, algo salió mal.\nNo se encontraron usuarios en la
base de datos.');
    }
} catch (error) {
    console.error('✖ Ops, algo salió mal.\n', error);
}
...

```

Acá la salida empieza a realizar la unificación de todos los conceptos de consultas que venimos estudiando hasta aquí, veamos la consulta y la respuesta asociada:



```

SQL ejecutado:
SELECT
    `Usuario`.`id`,
    `Usuario`.`nombre`,
    `Usuario`.`apellido`,
    `Usuario`.`usuario`,
    `Usuario`.`email`,
    `Usuario`.`genero`,
    `Usuario`.`perfilId`,
    `perfil`.`id` AS `perfil.id`,
    `perfil`.`nombre` AS `perfil.nombre`,
    `perfil`.`responsabilidades` AS `perfil.responsabilidades`
FROM
    `usuarios` AS `Usuario`
    INNER JOIN `perfils` AS `perfil` ON `Usuario`.`perfilId` = `perfil`.`id`
WHERE
(
    `Usuario`.`apellido` LIKE '%ez'
    AND NOT (`Usuario`.`id` IN (1, 2, 3))
)
ORDER BY
    `Usuario`.`apellido` ASC;
 Estructura de la base de datos.
[

{
    "id": 6,
    "nombre": "Carlos",
    "apellido": "Dominguez",
    "usuario": "cdom",
    "email": "carlos@mail.com",
    "genero": "M",
    "perfilId": 4,
    "perfil": {
        "id": 4,
        "nombre": "Product Owner",
        "responsabilidades": "Definir backlog del producto, priorizar funcionalidades, representar al cliente."
    }
},
{
    "id": 5,
    "nombre": "Marta",
    "apellido": "Fernández",
    "usuario": "mfer",
    "email": "marta@mail.com",
    "genero": "F",
    "perfilId": 3,
    "perfil": {
        "id": 3,
        "nombre": "Scrum Master",
        "responsabilidades": "Facilitar el proceso ágil, remover impedimentos, liderar las ceremonias Scrum."
    }
}
]

```

phillip@hp7hook-Philipp ~nas06-ES6 main = 217 ~1

Es interesante ver por un lado la consulta SQL que se generó a partir de nuestra configuración de findAll y por otro lado el resultado como un array de objetos Usuario agregando en cada uno el Perfil asociado en una propiedad llamada perfil.

Y para finalizar veamos que pasa si lo hacemos al revés tomando la consulta desde el perfil para traer todos los desarrolladores:

```

try {
  // Buscar un usuario por ID...
  const perfil = await Perfil.findOne({
    where: {
      nombre: { [Op.eq]: 'Desarrollador' }
    },
    include: [
      { model: Usuario,
        as: 'usuarios', // Alias definido en el modelo Perfil
      }
    ]
  });
  if (perfil) {
    console.log(`✅ Resultados:`);
    console.log(JSON.stringify(perfil, null, 2));
  }
  else {
    console.error('✖ Ops, algo salió mal.\nNo se encontraron usuarios en la base de datos.');
  }
}
catch (error) {
  console.error('✖ Ops, algo salió mal.\n', error);
}
finally {
  // Cerrar la conexión a la base de datos
  await sequelize.close();
  console.log('● Conexión cerrada con la base de datos');
}

```

No hay mucha novedad ya en la sentencia pero veamos como reacciona el modelo al realizar la consulta desde el perfil, veamos primero la forma de la consulta generada:

```

SQL ejecutado:
SELECT
`Perfil`.*,
`usuarios`.`id` AS `usuarios.id`,
`usuarios`.`nombre` AS `usuarios.nombre`,
`usuarios`.`apellido` AS `usuarios.apellido`,
`usuarios`.`usuario` AS `usuarios.usuario`,
`usuarios`.`email` AS `usuarios.email`,
`usuarios`.`genero` AS `usuarios.genero`,
`usuarios`.`perfil_id` AS `usuarios.perfilId`
FROM
(
  SELECT
    `Perfil`.`id`,
    `Perfil`.`nombre`,
    `Perfil`.`responsabilidades`
  FROM
    `profiles` AS `Perfil`
  WHERE
    `Perfil`.`nombre` = 'Desarrollador'
  LIMIT
    1
) AS `Perfil`
LEFT OUTER JOIN `usuarios` AS `usuarios` ON `Perfil`.`id` = `usuarios`.`perfil_id`;
Resultados:

```

Donde más allá de la forma rebuscada de consultar la tabla `profiles` podemos observar el join y la recuperacion de los datos de los usuarios para generar luego el resultado que se verá como sigue:

```

 Resultados:
{
  "id": 2,
  "nombre": "Desarrollador",
  "responsabilidades": "Implementar funcionalidades, corregir errores, participar en revisiones de código.",
  "usuarios": [
    {
      "id": 2,
      "nombre": "Luis",
      "apellido": "Martínez",
      "usuario": "lmartinez",
      "email": "luis@mail.com",
      "genero": "M",
      "perfilId": 2
    },
    {
      "id": 4,
      "nombre": "Julián",
      "apellido": "Ruiz",
      "usuario": "jruiz",
      "email": "julian@mail.com",
      "genero": "M",
      "perfilId": 2
    },
    {
      "id": 8,
      "nombre": "Pedro",
      "apellido": "Sosa",
      "usuario": "psosa",
      "email": "pedro@mail.com",
      "genero": "M",
      "perfilId": 2
    }
  ]
}
● Conexión cerrada con la base de datos
φ philip@hpZbook-Philip ▶ paso06-ES6 ▶ main ≡ ?19 ~1

```

Consultas RAW

Para situaciones donde necesitas mayor flexibilidad, puedes ejecutar consultas SQL directas, sin embargo estas situaciones quizás deberían manejarse con otras alternativas como con la utilización de procedimientos almacenados, vistas u otras herramientas de la base de datos

De todos modos para dejar completo el material de consulta veamos cómo se implementa esta opción:

```

const resultado = await sequelize.query(
  "SELECT * FROM usuarios WHERE edad > 30",
  { type: QueryTypes.SELECT }
);

```

Donde el primer parámetro de `.query` es la consulta y el segundo indica que es un Select mediante la propiedad `type` del objeto de configuración que puede tener uno de los posibles valores especificados en `QueryType`.

Tipos de `QueryTypes` en Sequelize

Cuando se utiliza `sequelize.query()` para ejecutar SQL directo, se puede especificar el tipo de consulta usando la opción `type` con alguno de los siguientes valores disponibles en `Sequelize.QueryTypes`:

QueryType	Descripción
<code>SELECT</code>	Devuelve un array de resultados. Ideal para <code>SELECT * FROM ...</code>
<code>INSERT</code>	Devuelve el ID del nuevo registro insertado (si aplica)
<code>UPDATE</code>	Devuelve el número de filas afectadas

QueryType	Descripción
DELETE	Devuelve el número de filas eliminadas
UPSERT	Devuelve información del upsert , si está soportado
BULKUPDATE	Para operaciones de actualización en lote
BULKDELETE	Para operaciones de borrado en lote
SHOWTABLES	Devuelve la lista de tablas en la base de datos
DESCRIBE	Devuelve la descripción de una tabla (columnas y tipos)
RAW	Devuelve los resultados crudos sin procesar por Sequelize

6.3 Manipular Datos

Vamos a hora a revisar brevemente las operaciones de manipulación de datos: crear, actualizar y borrar datos de la base de datos.

Es importante tener en cuenta la estructura de nuestro modelo cuando llevamos a cabo estos cambios o modificaciones.

6.3.1. Creación o inserción de entidades

Para crear una instancia de una entidad utilizamos el método `.create()`, que recibe como parámetro un objeto con al menos los atributos obligatorios y sí valor por defecto del modelo.

El método `create` va a realizar la inserción de la fila en la tabla y luego va a completar los atributos con valor por defecto y/o la clave primaria en el caso que fuera un valor autoincremental.

Notar en el siguiente ejemplo que vamos a enviar los siguientes datos a la base de datos, en este primer caso vamos a crear un nuevo usuario para un perfil que ya existe por lo que enviamos los datos del usuario con la propiedad `perfilId` y el id del perfil a asociar:

```
try{
  // Crear un nuevo usuario con datos de prueba
  let usuario = {
    nombre: 'Cuti',
    apellido: 'Romero',
    usuario: 'cuti',
    email: 'cromero@mail.com',
    genero: 'M',
    perfilId: 2
  };
  console.log(` Creando un nuevo usuario..., antes de crear \n ${JSON.stringify(usuario, null, 2)}`);
  usuario = await Usuario.create(usuario, {
    include: {
      association: 'perfil' // el alias definido en belongsTo()
    }
  });
}
```

```

    console.log(`\n🕒 Creando un nuevo usuario..., después de crear \n
${JSON.stringify(usuario, null, 2)}`);

    usuario = await Usuario.findByPk(11);
    console.log(`🕒 Nuevo usuario encontrado..., después de crear \n
${JSON.stringify(usuario, null, 2)}`);
}
catch (error) {
    console.error('✖ Ops, algo salió mal.\n', error);
}

```

Para este caso si ejecutamos `app.js` con el fragmento anterior, obtendremos la salida que adjuntamos a continuación:

```

● Inicializando usuarios... (Log Deshabilitado)
✓ Usuarios inicializados
● Base de datos inicializada desde script
● Creando un nuevo usuario..., antes de crear
{
  "nombre": "Cuti",
  "apellido": "Romero",
  "usuario": "cuti",
  "email": "cromero@mail.com",
  "genero": "M",
  "perfilId": 2
}

● SQL:
INSERT INTO `usuarios` (`id`, `nombre`, `apellido`, `usuario`, `email`, `genero`, `fecha_alta`, `perfil_id`) VALUES (NULL,$1,$2,$3,$4,$5,$6,$7);

● Creando un nuevo usuario..., después de crear
{
  "fechaAlta": "2025-04-20T01:33:39.609Z",
  "id": 11,
  "nombre": "Cuti",
  "apellido": "Romero",
  "usuario": "cuti",
  "email": "cromero@mail.com",
  "genero": "M",
  "perfilId": 2
}

💡 SQL ejecutado:
SELECT
  `id`,
  `nombre`,
  `apellido`,
  `usuario`,
  `email`,
  `genero`,
  `fecha_alta` AS `fechaAlta`,
  `perfil_id` AS `perfilId`
FROM
  `usuarios` AS `Usuario`
WHERE
  `Usuario`.`id` = 11;

● Nuevo usuario encontrado..., después de crear
{
  "id": 11,
  "nombre": "Cuti",
  "apellido": "Romero",
  "usuario": "cuti",
  "email": "cromero@mail.com",
  "genero": "M",
  "fechaAlta": "2025-04-20T01:33:39.609Z",
  "perfilId": 2
}
● Conexión cerrada con la base de datos
philip@hpZbook-Philip: ~$ main ~1

```

En la que se observa la sentencia insert de la tabla usuarios por un lado. Y el objeto usuario resultante en el que se pueden apreciar las propiedades `id` y `fechaAlta` que no fueron enviadas y que las agregó sequelize al crear el usuario en la base de datos.

En cambio otra opción sería crear un usuario para un perfil nuevo que no existía previamente, en ese caso podemos enviar el objeto usuario con la propiedad `perfil` y el objeto del nuevo perfil a crear y sequelize creará

ambas filas en la base de datos en el orden correcto.

Por ejemplo:

```
try{
    // Crear un nuevo usuario con datos de prueba
    let usuario = {
        nombre: 'Cuti',
        apellido: 'Romero',
        usuario: 'cuti',
        email: 'cromero@mail.com',
        genero: 'M',
        perfil: {
            nombre: 'Coach Ágil', // aquí enviamos el nombre de un perfil que no
            existe en la base de datos.
            // y luego las responsabilidades del perfil.
            responsabilidades: 'Entrenar al equipo en la implementación de las
            prácticas y ceremonias de la agilidad.'
        }
    };
    console.log(`\n Creando un nuevo usuario..., antes de crear \n
${JSON.stringify(usuario, null, 2)}`);
    usuario = await Usuario.create(usuario, {
        include: {
            association: 'perfil' // el alias definido en belongsTo()
        }
    });

    console.log(`\n Creando un nuevo usuario..., después de crear \n
${JSON.stringify(usuario, null, 2)}`);

    usuario = await Usuario.findByPk(11);
    console.log(`\n Nuevo usuario encontrado..., después de crear \n
${JSON.stringify(usuario, null, 2)})`);
}
catch (error) {
    console.error('✖ Ops, algo salió mal.\n', error);
}
finally {
    // Cerrar la conexión a la base de datos
    await sequelize.close();
    console.log(`\n Conexión cerrada con la base de datos`);
}
```

Al ejecutar el `app.js` con el fragmento anterior (ejemplo: paso07-ES6) nos vamos a encontrar con la siguiente salida:

```

✓ Usuarios inicializados
● Base de datos inicializada desde script
● Creando un nuevo usuario..., antes de crear
{
  "nombre": "Cuti",
  "apellido": "Romero",
  "usuario": "cuti",
  "email": "cromero@mail.com",
  "genero": "M",
  "perfil": {
    "nombre": "Coach Ágil",
    "responsabilidades": "Entrenar al equipo en la implementación de las prácticas y ceremonias de la agilidad."
  }
}

● SQL:
INSERT INTO `perfiles` (`id`, `nombre`, `responsabilidades`) VALUES (NULL,$1,$2);

● SQL:
INSERT INTO `usuarios` (`id`, `nombre`, `apellido`, `usuario`, `email`, `genero`, `fecha_alta`, `perfil_id`) VALUES (NULL,$1,$2,$3,$4,$5,$6,$7);

● Creando un nuevo usuario..., después de crear
{
  "fechaAlta": "2025-04-20T01:01:34.292Z",
  "id": 11,
  "nombre": "Cuti",
  "apellido": "Romero",
  "usuario": "cuti",
  "email": "cromero@mail.com",
  "genero": "M",
  "perfil": {
    "id": 7,
    "nombre": "Coach Ágil",
    "responsabilidades": "Entrenar al equipo en la implementación de las prácticas y ceremonias de la agilidad."
  },
  "perfilId": 7
}

SQL ejecutado:
SELECT
  `id`,
  `id`,
  `nombre`,
  `apellido`,
  `usuario`,
  `email`,
  `genero`,
  `fecha_alta` AS `fechaAlta`,
  `perfil_id` AS `perfilId`
FROM
  `usuarios` AS `Usuario`
WHERE
  `Usuario`.`id` = 11;

● Nuevo usuario encontrado..., después de crear
{
  "id": 11,
  "nombre": "Cuti",
  "apellido": "Romero",
  "usuario": "cuti",
  "email": "cromero@mail.com",
  "genero": "M",
  "fechaAlta": "2025-04-20T01:01:34.292Z",
  "perfilId": 7
}
● Conexión cerrada con la base de datos
philipin@Phook-Philip: ~$ main ~1

```

Donde se puede observar entre otras salidas, las dos sentencias insert una para la tabla perfiles y la otra para la tabla de usuarios

Además, de objeto usuario creado como resultado que incluye la propiedad `perfilId` con el nuevo valor 7

6.3.2. Actualización de valores

Ahora bien, que pasa cuando ya existe la fila en la base de datos y necesitamos cambiar el valor de una columna/atributo, o bien establecer un valor por primera vez si este estuviera en null por ejemplo. La operación a realizar es la de actualización, pero antes de poder actualizar lo que necesitamos modificar tenemos que obtener el objeto desde la base de datos, y esto como ya lo vimos lo podemos hacer con `.findById(...)` o con `.findOne(...)` pero normalmente esta tarea se va a realizar por PK.

Una vez obtenido el objeto, modificamos el valor deseado y enviamos la modificación a la base de datos.: Por ejemplo:

```

try{
  // Buscar un usuario por Clave Primaria (ID) para luego actualizar el apellido

```

```
const idBuscar = 2;
const usuarioEncontrado = await Usuario.findByPk(idBuscar);

if (usuarioEncontrado) {
    console.log(`\nUsuario encontrado..., antes de crear \n${JSON.stringify(usuarioEncontrado, null, 2)}\n`);
    usuarioEncontrado.apellido = 'Martínez Crespo';

    await usuarioEncontrado.save(); // Guardar los cambios en la base de datos
    console.log(`\nUsuario Actualizado...\n`);

    const usuarioRecuperado = await Usuario.findByPk(2);
    console.log(`\nUsuario recuperado..., después de actualizar \n${JSON.stringify(usuarioRecuperado, null, 2)}\n`);

}
else {
    console.error('X Ops, algo salió mal.\nNo se encontrará el usuario en la base de datos.');
}

}
catch (error) {
    console.error('X Ops, algo salió mal.\n', error);
}
```

El bloque anterior efectivamente busca el usuario con el `id` = 2, luego modifica el apellido y finalmente guarda los cambios.

Hemos agregado algunos cambios para mostrar resultados y que se pueda seguir la ejecución

Del fragmento anterior del archivo `paso08-ES6/app.js` se obtiene la siguiente salida:

● Base de datos inicializada desde script

SQL ejecutado:

```
SELECT
`id`,
`nombre`,
`apellido`,
`usuario`,
`email`,
`genero`,
`fecha_alta` AS `fechaAlta`,
`perfil_id` AS `perfilId`
FROM
`usuarios` AS `Usuario`
WHERE
`Usuario`.`id` = 2;
```

● Usuario encontrado..., antes de crear

```
{
"id": 2,
"nombre": "Luis",
"apellido": "Martínez",
"usuario": "lmartinez",
"email": "luis@mail.com",
"genero": "M",
"fechaAlta": "2025-04-20T01:55:41.222Z",
"perfilId": 2
}
```

SQL:

```
UPDATE `usuarios` SET `apellido`=$1 WHERE `id` = $2
```

● Usuario Actualizado...

SQL ejecutado:

```
SELECT
`id`,
`nombre`,
`apellido`,
`usuario`,
`email`,
`genero`,
`fecha_alta` AS `fechaAlta`,
`perfil_id` AS `perfilId`
FROM
`usuarios` AS `Usuario`
WHERE
`Usuario`.`id` = 2;
```

● Usuario recuperado..., después de actualizar

```
{
"id": 2,
"nombre": "Luis",
"apellido": "Martínez Crespo",
"usuario": "lmartinez",
"email": "luis@mail.com",
"genero": "M",
"fechaAlta": "2025-04-20T01:55:41.222Z",
"perfilId": 2
}
```

● Conexión cerrada con la base de datos

```
[philip@hpZbook-Philip: ~]$ main.js
```

en la que nuevamente podemos hacer el seguimiento del estado del objeto y las sentencias que Sequelize genera hacia la base de datos y los datos del objeto luego de la aplicación de las sentencias.

Una alternativa menos orientada a ORM pero válida en el caso que hiciera falta hacer una actualización de más de una fila, es solicitar la actualización al modelo de todas las filas de la base de datos que cumplan una condición. Esta estructura está orientada a sentencias directas SQL/DML y no debería ser utilizada en un modelo con buenas prácticas de desarrollo. Para actualizar directamente filas en forma masiva, aplicando un criterio `where` para restringir la actualización a las filas que cumplen el criterio se puede realizar:

```
await Usuario.update(
  { apellido: "Perez" }, // Valores de actualización
  // TODAS LAS FILAS QUE CUMPLAN LA CONDICIÓN QUEDAN VALIENDO LO MISMO
{
  where: {
    id: 5 // Condición que las filas deben cumplir, en este caso se aplica sobre
el ID
    // que es la clave primaria y por lo tanto este no sería el mecanismo ideal
para implementar la acción
  }
});
```

Queda esta opción para completar el material más allá de las aclaraciones realizadas.

6.3.3. Borrado físico de filas

Sequelize Nos ofrece dos mecanismos de borrado de filas de la base de datos, por un lado permite que realicemos el borrado físico de una fila, es decir, que eliminemos la fila de la persistencia. O realizar un borrado lógico, para el que una opción sería utilizar la actualización vista en el apartado anterior para modificar una columna/atributo que indique que la fila está borrada o utilizar el modo **Paranoid** de **sequelize** que escapa al alcance de este curso pero del que dejamos una punta de ovillo hacia el final del presente apartado.

Para eliminar una fila de la base de datos de forma física, es decir, borrar los datos de la persistencia podemos usar la función **.destroy()** para la que previamente debemos haber obtenido el objeto desde la base de datos como lo hicimos para realizar la actualización:

```
try{
  // Buscar un usuario por Clave Primaria (ID) para luego actualizar el apellido
  const idBuscar = 2;
  const usuarioEncontrado = await Usuario.findByPk(idBuscar);

  if (usuarioEncontrado) {
    // Para validar vamos a contar las filas antes y después de borrar el
    usuario
    const cantidadAntes = await Usuario.count();
    console.log(` Filas antes de borrar: ${cantidadAntes}`);

    await usuarioEncontrado.destroy(); // Borrado de los datos de la base de
    datos

    // Para validar vamos a contar las filas antes y después de borrar el
    usuario
    const cantidadDespues = await Usuario.count();
    console.log(` Filas después de borrar: ${cantidadDespues}`);

  }
  else {
    console.error(' X Ops, algo salió mal.\nNo se encontrará el usuario en la
    base de datos');
  }
}
```

```

    base de datos.');
}

}

catch (error) {
  console.error('✖ Ops, algo salió mal.\n', error);
}

```

Allí observamos el uso de `.destroy()` para provocar la eliminación física en la base de datos

Podemos validar lo mencionado en la siguiente salida:

● Base de datos inicializada desde script

► SQL ejecutado:

```

SELECT
`id`,
`nombre`,
`apellido`,
`usuario`,
`email`,
`genero`,
`fecha_alta` AS `fechaAlta`,
`perfil_id` AS `perfilId`
FROM
`usuarios` AS `Usuario`
WHERE
`Usuario`.`id` = 2;

```

► SQL ejecutado:

```

SELECT
count(*) AS `count`
FROM
`usuarios` AS `Usuario`;

```

● Filas antes de borrar: 10

► SQL ejecutado:

```

DELETE FROM `usuarios`
WHERE
`id` = 2

```

► SQL ejecutado:

```

SELECT
count(*) AS `count`
FROM
`usuarios` AS `Usuario`;

```

● Filas después de borrar: 9

● Conexión cerrada con la base de datos

Donde podemos ver la cantidad de filas antes de borrar 10 y la cantidad de filas después de borrar 9
Además de la sentencia `DELETE` de base de datos que provoca el borrado.

Es importante tener en cuenta que **Sequelize** nos ofrece la posibilidad de realizar borrado lógico de filas como opuesto al borrado físico. En general esta es la alternativa que se usa en la mayor parte de los escenarios.

Sequelize llama a este modo: `Paranoid`, podés profundizar en la [documentación](#)

6.4 Manejo de transacciones

6.4 Manejo de Transacciones

En Sequelize, una **transacción** permite agrupar múltiples operaciones que se deben ejecutar como una unidad atómica: **si una falla, todas se revierten**. Esto es fundamental para mantener la integridad de los datos en operaciones críticas, como registrar un usuario y su actividad en simultáneo.

Ejemplo básico con control manual

```
import sequelize from './db.js';

const t = await sequelize.transaction();

try {
  const usuario = await Usuario.create({
    nombre: 'Nicolás',
    apellido: 'Juárez',
    usuario: 'njuarez',
    email: 'nico@mail.com',
    genero: 'M',
    perfilId: 2
  }, { transaction: t });

  await LogActividad.create({
    mensaje: `Alta de usuario ${usuario.usuario}`,
    usuarioId: usuario.id
  }, { transaction: t });

  await t.commit();
  console.log('☑️ Transacción completada');
} catch (error) {
  await t.rollback();
  console.error('☒️ Transacción revertida:', error.message);
}
```

💡 Todos los métodos dentro de la transacción deben recibir { transaction: t } como segundo parámetro para que estén incluidos en la operación atómica.

Alternativa: wrapper automático

Sequelize también permite usar un wrapper que administra commit y rollback automáticamente:

```
await sequelize.transaction(async (t) => {
  const usuario = await Usuario.create({ ... }, { transaction: t });
  await LogActividad.create({ ... }, { transaction: t });
});
```

Aquí podemos observar una versión más compacta de la configuración de una transacción.

⌚ Cuándo usar transacciones

- Al guardar datos en varias tablas relacionadas (por ejemplo: usuario + perfil, factura + ítems).
- Al realizar operaciones que deben ser consistentes (por ejemplo: transferencias, cancelaciones).
- Cuando se quiere garantizar que un conjunto de cambios se realice de forma segura o no se aplique en absoluto.

⚠ ¡Importante! SQLite (motor que usamos en este material) soporta transacciones, pero en motores como PostgreSQL o MySQL estas cobran aún más relevancia por el soporte avanzado de concurrencia y aislamiento.

7 Repositorios

7.1 🧠 ¿Qué es un Repositorio?

El **Patrón Repositorio** es una estrategia de diseño que permite **separar la lógica de acceso a datos** del resto de la aplicación. Su objetivo es encapsular las interacciones con la base de datos en una **interfaz orientada a objetos de alto nivel**, de modo que:

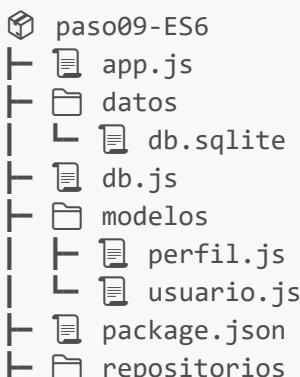
- El código que usa los datos (por ejemplo, controladores o scripts) **no necesita saber nada sobre Sequelize** ni cómo funciona internamente.
- Se puede **modificar o reemplazar** la lógica de acceso a datos sin afectar el resto de la aplicación. Por ejemplo se podría migrar la aplicación de SQLite a MongoDB y no tocar más que la capa de repositorios
- Favorece la **organización, reutilización, testeo y evolución** del código.

7.2 ⚙️ Implementemos una versión sobre el último ejemplo

En este ejemplo **paso09-ES6** vamos a trabajar con los siguientes objetivos:

- Ocultar los detalles específicos de Sequelize (como `where`, `findAll`, `findById`, etc.)
- Proveer una **interfaz simple** y legible para trabajar con los modelos
- Permitir la **extensión por herencia** de métodos básicos con comportamientos más avanzados
- Mostrar un enfoque modular que respeta el principio de **separación de responsabilidades**

💻 Estructura del ejemplo



```
|   └── repositoryBase.js
|   └── repositoryPerfil.js
|   └── repositoryUsuario.js
└── scripts
    ├── inicializar-bd.js
    ├── perfilesSeeder.js
    └── usuariosSeeder.js
```

❖ ¿Qué se logra?

- Reutilizar lógica de acceso común (obtener todos, obtener por ID, crear, actualizar, eliminar)
- Separar claramente el modelo Sequelize de la lógica de negocio o uso
- Facilitar futuras integraciones con controladores, APIs o pruebas unitarias
- Enseñar a los estudiantes un enfoque escalable y profesional para manejar persistencia

7.3 🔍 ¿Qué es RepositoryBase?

El archivo RepositoryBase.js define una clase base genérica que encapsula las operaciones comunes de persistencia para cualquier modelo Sequelize.

Su objetivo es que los repositorios concretos (como RepositoryUsuario o RepositoryPerfil) puedan heredar estas funcionalidades básicas sin tener que reescribirlas una y otra vez.

⌚ ¿Por qué usarlo?

- Centraliza lógica CRUD repetitiva
- Facilita la creación de nuevos repositorios en segundos
- Mantiene una API coherente para acceder a cualquier entidad del sistema
- Oculta los detalles del ORM para que el uso sea intuitivo

7.3.1 📜 Archivo `repositorys/respositorioBase.js`

```
export default class RepositoryBase {
  constructor(modelo) {
    this.modelo = modelo;
  }

  async obtenerTodos({ pagina = 1, limite = 10 } = {}) {
    const offset = (pagina - 1) * limite;
    return this.modelo.findAll({ limit: limite, offset });
  }

  async obtenerPorId(id) {
    return this.modelo.findByPk(id);
  }

  async crear(datos) {
    return this.modelo.create(datos);
  }
}
```

```

// Aquí hay algunas novedades
async actualizar(id, datos) {
  const instancia = await this.modelo.findByPk(id);
  if (!instancia) throw new Error(`Error: Instancia con id: ${id} no
encontrada`);

  // Actualiza los datos de la instancia
  // y guarda los cambios en la base de datos
  // Se puede usar Object.assign o el método set para asignar los datos en lote
  // Object.assign(instancia, datos);
  // instancia.set(datos);

  // y luego guardar los cambios en la base de datos
  // return instancia.save();

  // O simplemente usar el método update
  // y pasarle los datos a actualizar
  return instancia.update(datos);
}

async eliminar(id) {
  const instancia = await this.modelo.findByPk(id);
  if (!instancia) return 0;
  await instancia.destroy();
  return 1;
}
}

```

Comenzamos con el constructor que tiene por función inicializar el repositorios para un Modelo particular.

Luego tenemos el comportamiento `obtenerTodos` que tiene por función servir de interfaz genérica para el método `.findAll()` de `sequelize` pero como ya dijimos hay que tener cuidado por los casos de grandes volúmenes de datos, entonces en su versión por defecto solo va a traer una página de 10 filas.

Para usar este comportamiento bastaría con que en el lugar donde sea necesario solo realicemos 2 acciones:

```

// Importa el repositorio
import RepositorioBase from './repositorios/repositorioUsuario.js';
import Usuario from './modelos/usuarios.js';

// Por ser el repositorio base debería instanciarlo para con un Modelo específico
const repoBaseUsuarios = new RespositorioBase(Usuario);

// Usar el repositorio sin tener que preocuparme por detalles de sequelize
// Esta versión trae la primera página de 10 filas por los valores por defecto
const usuarios = await repoBaseUsuarios.obtenerTodos();

// pero también podría utilizar
// const usuarios = await repositorioUsuario.obtenerTodos({ pagina: 2, limite: 5
});
// para obtener la segunda página de 5 filas

```

```
// Luego usar la lista de usuarios como sirva
usuarios.forEach(u => console.log(` - ${u.nombre} ${u.apellido}`));
```

Esperamos que se pueda apreciar la diferencia con el código anterior que mezclaba los detalles de sequelize con el código que hace uso de los resultados a nivel de objetos. Incluso en el presente caso necesitamos crear la instancia del repositorio base, cosa que cuando armemos los repositorios concretos ya no será necesaria.

Del mismo modo podría utilizar los demás métodos básicos de creación, recuperación por clave, actualización y borrado, que en inglés se denominan normalmente como métodos CRUD, aquí ejemplos:

Este fragmento es el script `prueba-repo-base.js` del ejemplo [paso09-ES6](#).

```
// Importa el repositorio
import RepositorioBase from './repositorios/repositorioBase.js';

...

// Por ser el repositorio base debería instanciarlo para con un Modelo
específico
// Después veremos que esto no es necesario cuando usemos repositorios concretos
const repoBaseUsuarios = new RepositorioBase(Usuario);

// Obtener el usuario con la clave `id` = 1
const usuario1 = await repoBaseUsuarios.obtenerPorId(1);
console.log(`\nUsuario 1: ${usuario1.nombre} ${usuario1.apellido}`);

// Modificar el nombre del usuario 1 por "Felipe"
const updatedUsr = await repoBaseUsuarios.actualizar(1, { nombre: "Felipe" });
console.log(`\nUsuario 1 actualizado: ${updatedUsr.nombre}
${updatedUsr.apellido}`);

// Contar las filas de la tabla usuarios
let cantidad = await repoBaseUsuarios.contarFilas();
console.log(`\nCantidad de usuarios antes de crear: ${cantidad}`);

// Crear un usuario
let usuario = {
    nombre: 'Cuti',
    apellido: 'Romero',
    usuario: 'cuti',
    email: 'cromero@mail.com',
    genero: 'M',
    perfilId: 2
};
const newUsr = await repoBaseUsuarios.crear(usuario);
console.log(`\nUsuario creado: (${newUsr.id}) ${newUsr.nombre}
${newUsr.apellido}`);
```

```
cantidad = await repoBaseUsuarios.contarFilas();
console.log(`\nCantidad de usuarios después de crear: ${cantidad}`);

// Borrar el usuario creado
const deletedUsr = await repoBaseUsuarios.eliminar(newUsr.id);

cantidad = await repoBaseUsuarios.contarFilas();
console.log(`\nCantidad de usuarios después de borrar: ${cantidad}`);

...
```

💡 Nota explicativa Este bloque de código muestra cómo utilizar directamente el `RepositorioBase` con un modelo concreto, en este caso `Usuario`, para realizar todas las operaciones fundamentales de acceso a datos sin necesidad de escribir consultas manuales ni tratar directamente con `Sequelize`.

☑ ¿Qué conceptos estamos aplicando?

- Instanciación genérica del repositorio:
Al pasar el modelo `Usuario` al constructor de `RepositorioBase`, obtenemos una instancia que sabe cómo operar sobre esa tabla sin necesidad de que definamos métodos personalizados.
- Abstracción del ORM: Las llamadas a `obtenerPorId`, `actualizar`, `crear`, `eliminar` y `contarFilas` funcionan sin que tengamos que usar `findById`, `update`, `destroy` o `count`. Esto oculta la complejidad de `Sequelize` y permite trabajar con una interfaz más legible y orientada a objetos.
- Código reutilizable y coherente: Este mismo bloque podría funcionar exactamente igual con otros modelos (Perfil, Curso, etc.) simplemente cambiando el modelo que se pasa al repositorio, sin modificar ninguna línea de lógica.
- Separación de responsabilidades: Este enfoque separa la lógica de persistencia (repositorio) de la lógica de uso (como mostrar por consola o responder en una API), lo que hace que el código sea más fácil de mantener, testear y escalar.

💡 ¿Qué ventaja concreta aporta esto a nuestro código?

- Le permite enfocar la relación entre modelo, repositorio y lógica de uso sin distraerse con sintaxis complejas de SQL o `Sequelize`.
- La implementación de un patrón da una herramienta real que pueden reutilizar en cualquier proyecto.
- Propone escribir código que crece bien, sin depender de copiar y pegar `findAll` o `update` por todos lados.

Ahora bien, esta capa del `RepositorioBase` ya nos agrega una buena separación entre la lógica de acceso a base de datos que tiene que preocuparse por detalles de la base de datos y en este caso SQL y la lógica de aplicación que se preocupa más de entidades de dominio y sus interacciones. Tiene un problema cuando queremos un comportamiento particular que solo sea para una tabla.

Por ejemplo, en el caso de los usuarios, si bien el `obtenerPorId` nos sirve para trabajar por clave primaria y en muchos casos nos resuelve el problema, la tabla tiene una clave alternativa que tiene que ver con el

nombre de usuario (propiedad `usuario` del objeto) que es único y que debería permitirnos una búsqueda directa por este atributo.

Ahora bien, cómo hacemos esto, podríamos directamente esos métodos programarlos entre la lógica de aplicación pero volvemos a caer en el problema de antes. La solución son los **RepositoriosConcretos** es decir repositorios que ya están especializados para un modelo y en los que puedo escribir código específico para ese modelo.

En el ejemplo planteamos un repositorio concreto para cada modelo:

● RepositorioUsuario

- Extiende de `RepositorioBase` y agrega métodos como:
- `obtenerPorUsuario(nombreUsuario)`
- `buscarPorApellido(apellido)`

● RepositorioPerfil

- Extiende de `RepositorioBase` y podría tener:
- `usuariosAsociados(idPerfil)` (aunque esto también podría vivir en `Usuario`)

Y como veremos a continuación, estos repositorios concretos además van a resolver la necesidad de crear con `new` ante cada uso de los repositorios.

7.3.1 📜 Archivo `repositorios/repositorioUsuario.js`

```
import RepositorioBase from './repositorioBase.js';
import Usuario from '../modelos/usuario.js';

class RepositorioUsuario extends RepositorioBase { // Heredamos de la clase
    constructor() {
        super(Usuario); // Invocamos al contructor de la clase base pero con el modelo
        especificado
    }

    async obtenerPorUsuario(nombreUsuario) {
        return this.modelo.findOne({ where: { usuario: nombreUsuario } });
        // Aquí quedan los detalles de sequelize
    }

    async buscarPorApellido(apellido) {
        return this.modelo.findAll({ where: { apellido } });
        // O aquí...
    }
}

// creamos la instancia del repositorio y la exportamos una sola vez en toda la
// aplicación
export default new RepositorioUsuario();
```

💡 **Explicación del archivo `repositorioUsuario.js`**

Este archivo define una clase `RepositorioUsuario`, que hereda de `RepositorioBase`, y encapsula toda la lógica de acceso a los datos de la tabla `usuarios`.

A continuación, se detallan los puntos clave:

1. `import RepositorioBase from './repositorioBase.js';`

Se importa la clase base que contiene los métodos CRUD genéricos (`obtenerTodos`, `obtenerPorId`, `crear`, `actualizar`, `eliminar`).

Esto permite reutilizar lógica común sin duplicar código en cada repositorio.

2. `import Usuario from '../modelos/usuario.js';`

Se importa el modelo Sequelize asociado a la tabla `usuarios`, el cual será utilizado por este repositorio.

3. `class RepositorioUsuario extends RepositorioBase`

Se crea una clase especializada para el modelo `Usuario` que **hereda** de `RepositorioBase`.

Esto le da automáticamente acceso a todos los métodos base sin redefinirlos.

💡 **Este es un ejemplo de herencia clásica en Programación Orientada a Objetos.**

4. `constructor() { super(Usuario); }`

El constructor llama a `super(Usuario)`, lo cual significa que se pasa el modelo `Usuario` al repositorio base.

De esta forma, el repositorio base sabe operar sobre la tabla `usuarios`.

5. `async obtenerPorUsuario(nombreUsuario)`

Este método personalizado busca un usuario por su nombre de usuario (`usuario`).

Encapsula la lógica específica para que no esté repartida por la aplicación.

6. `async buscarPorApellido(apellido)`

Otro método personalizado que devuelve todos los usuarios que coincidan con un apellido.

Este tipo de comportamiento no está en el repositorio base, pero es fácil de agregar aquí.

7. `export default new RepositorioUsuario();`

Se **crea y exporta una única instancia** del repositorio.

Esto permite importar el repositorio directamente en otras partes del proyecto sin tener que instanciarlo y además garantiza que del repositorio siempre exista una y solo una instancia `singleton`

💡 **Este patrón reduce el boilerplate y garantiza que haya una sola instancia para acceder a los datos de usuarios.**

💡 **Ventajas de esta implementación**

- Reutilización de lógica CRUD genérica desde `RepositorioBase`
- Encapsulamiento de comportamientos específicos de `Usuario`

- Separación de responsabilidades entre datos y lógica de negocio
- Facilidad para testear, mantener y escalar
- Este enfoque es didáctico, profesional y completamente alineado con buenas prácticas de desarrollo backend.

Para finalizarse prestamos atención al código de uso del repositorioUsuario, encontramos código absolutamente trivial y completamente legible desde el punto de vista del dominio del problema, no nos queda mucho para decir pero esa es justamente la mayor ventaja de la división de responsabilidades puesto que las particularidades que hemos visto a lo largo de este material asociadas a ORM y Sequelize, quedan justamente encapsuladas en la capa de repositorios.

A continuación un fragmento del script [prueba-repo-usuario.js](#) del ejemplo [paso09-ES6](#).

```
// Cabe aclarar que no necesitamos crear ninguna instancia de repositorio
// porque el mismo repositorioUsuario que importamos es una instancia única
// (singleton)

// Obtener la primera página de 10 filas => usando el método del repositorioBase
const usuarios = await repositorioUsuario.obtenerTodos();
// Luego usar la lista de usuarios como sirva
console.log(`\n📝 Usuarios - Obtener todos (página 1 - 10 filas):`);
usuarios.forEach(u => console.log(` - ${u.nombre} ${u.apellido}`));

// Obtener el usuario con la clave `id` = 1 => usando el método del
repositorioBase
const usuario1 = await repositorioUsuario.obtenerPorId(1);
console.log(`\nUsuario 1: ${usuario1.nombre} ${usuario1.apellido}`);

// Modificar el nombre del usuario 1 por "Felipe" => usando el método del
repositorioBase
const updatedUsr = await repositorioUsuario.actualizar(1, { nombre: "Analía" });
console.log(`\nUsuario 1 actualizado: ${updatedUsr.nombre}
${updatedUsr.apellido}`);

// Contar las filas de la tabla usuarios => usando el método del repositorioBase
let cantidad = await repositorioUsuario.contarFilas();
console.log(`\nCantidad de usuarios antes de crear: ${cantidad}`);

// Crear un usuario => usando el método del repositorioBase
let usuario = {
    nombre: 'Cuti',
    apellido: 'Romero',
    usuario: 'cuti',
    email: 'cromero@mail.com',
    genero: 'M',
    perfilId: 2
};
const newUsr = await repositorioUsuario.crear(usuario);
console.log(`\nUsuario creado: (${newUsr.id}) ${newUsr.nombre}
${newUsr.apellido}`);

cantidad = await repositorioUsuario.contarFilas();
```

```
console.log(`\nCantidad de usuarios después de crear: ${cantidad}`);

// Borrar el usuario creado => usando el método del repositorioBase
const deletedUsr = await repositorioUsuario.eliminar(newUsr.id);

cantidad = await repositorioUsuario.contarFilas();
console.log(`\nCantidad de usuarios después de borrar: ${cantidad}`);

// O también podríamos usar el método del repositorioUsuario
// Por ejemplo, para obtener un usuario por nombre de usuario
const usrAna = await repositorioUsuario.obtenerPorUsuario("ana2025");
console.log(`\nUsuario 1: ${usrAna.nombre} ${usrAna.apellido}`);
```

Y ya para ir cerrando el presente material, nos queda mirar el resultado de ejecutar con:

```
node prueba-repo-usuario.js
```

la siguiente salida comprueba lo esperado ya sin demasiadas novedades más allá de la comprobación de lo esperado:

- ♦ Base de datos inicializada desde script

- ♦ Usuarios - Obtener todos (página 1 - 10 filas):

- Ana García
- Luis Martínez
- Sofía López
- Julián Ruiz
- Marta Fernández
- Carlos Domínguez
- Lucía Silva
- Pedro Sosa
- Valentina Paz
- Andrés Cruz

```
Usuario 1: Ana García
```

```
Usuario 1 actualizado: Analía García
```

```
Cantidad de usuarios antes de crear: 10
```

```
Usuario creado: (11) Cuti Romero
```

```
Cantidad de usuarios después de crear: 11
```

```
Cantidad de usuarios después de borrar: 10
```

```
Usuario 1: Analía García
```

8. Ejemplo con el código del presente material

CRUD (Create, Retrive, Update y Delete) con Sequelize

Paso a Paso: CRUD usando el ORM **Sequelize**

Bibliografía

[Documentación de Sequelize v6](#). Documentación online del framework.