UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Desarrollo de Software
Notas de Clase

Apunte 11 - ORM con Sequelize

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:

- Nociones Generales de Bases de Datos Relacionales
  - Motor de Base de Datos
  - Tablas
  - Columnas
  - Índices
  - Comparación con Bases de Datos No SQL (opcional)
  - Clave Primaria
  - Claves Foráneas
  - Normalización desde una perspectiva práctica
  - Transacciones
  - Procedimientos almacenados, vistas y funciones (solo concepto)
- SQLite
  - Características
  - Cómo "instalar" y correr SQLite en un ambiente de desarrollo (Linux/Windows)
  - Herramientas (Gráficas/CLI) para trabajar con SQLite (en este punto sería bueno elegir herramientas no específicas de SQLite de modo que si se cambiase el motor por PostgreSQL o MySql el set de herramientas se pueda mantener) *¿DBeaver?*
- Definición de Datos (DDL) en SQLite
  - Cómo crear, modificar y eliminar una base de datos y los principales tipos objetos (esto no necesariamente tiene que ser hecho usando las sentencias DDL sino que pueden usarse herramientas gráficas o no que permitan realizar estas operaciones)
  - CREATE
  - ALTER
  - DROP
- Manipulación de Datos (DML)
  - Realizar consultas utilizando la sentencia SELECT
  - Realizar consultas sobre más de una tabla relacionada

    - Agregar filas utilizando INSERT
    - Actualizar datos con UPDATE
    - Eliminar filas usando DELETE
- Utilizar scripts para realizar tareas de inicialización o repetitivas
    - Cómo escribir de forma manual y/o generar de forma automática un script que permita inicializar una base de datos o algunos de sus objetos e insertar datos.
    - Cómo ejecutar estos scripts.
- ORM
    - Concepto
    - Ventajas
    - Desventajas
- Si hay tiempo...(opcional)
    - Si bien SQLite es una base de datos relacional completa, en la medida que alcance el tiempo, se podría realizar el mismo tipo de ejercicios con PostgreSQL o MySql de modo de tener una panorama de lo que es una base de datos relacional con un servidor.

1. Necesidad de una base de datos

En general, sea cual sea el propósito de una aplicación web, éstas necesitan manipular y almacenar datos. En un reducido número de casos, esta necesidad puede estar cubierta por una solución simple como un archivo secuencial. 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.

2. Bases de datos relacionales

Las bases de datos relacionales (**RDBMS** - **Relational Database Management System**) son el tipo de base de datos más extendido y de prppósito más general.

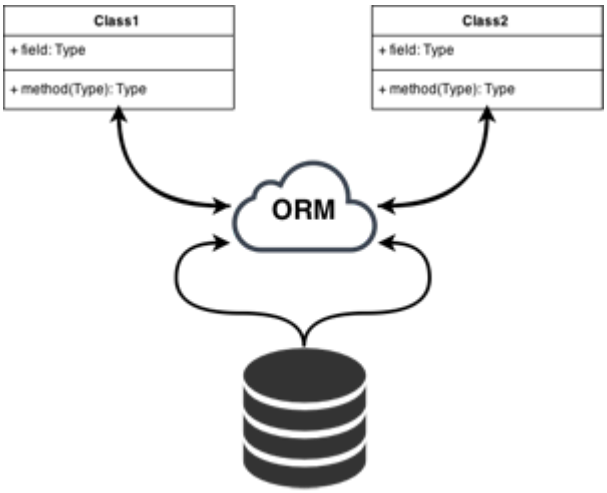
Las bases de datos relacionales organizan los datos en forma de tablas y utilizan el lenguaje `SQL` (Structured Query Language) para la gestión y recuperación de los datos.

2.1 SQLite

En este curso vamos a utilizar SQLite (habitualmente pronunciado */'si: kwə 'lait/*) que es un sistema de gestión de bases de datos relacional 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)

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:

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

Con un ORM

```
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 **abstrae** 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

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
```

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**, 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 comunmente 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 prgrama.

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

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

Pero lo más usual será usar una base de datos con persistencia. El valor de la clave `storage` contiene el path al archivo de la base de datos física.

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

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

Para probar la conexión con la base de datos:

```
try{
  // Probar la conexión
  await sequelize.authenticate();

  console.log("Base de Datos: lista");
}
catch (error){
  console.error("Ha ocurrido un error: ", error);
}
```

O para crear los objetos automáticamente:

```
try{
  // Sincronizar la base de datos con el modelo
  await sequelize.sync();
  console.log("Base de Datos: lista");
}
catch (error){
  console.error("Ha ocurrido un error: ", error);
}
```

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`

```
const { DataTypes  } = require("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
```

7.4. 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 }`

En este 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  }
});
```

Podés ver más sobre definir modelos acá

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 acá, 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"
  }
},
```

4.6 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.

4.6.1 Recuperar todos los registros

Para recuperar todos los registros de una tabla, usamos el método `findAll()` sin parámetros:

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

4.6.2 Recuperar un registro específico

Para buscar un registro específico por clave primaria o cualquier otro criterio, utilizamos `findOne()`:

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

4.6.3 Seleccionar atributos específicos

Si solo necesitas algunos campos específicos de los registros, puedes especificarlos en el atributo `attributes`:

```
const usuarios = await Usuario.findAll({
  attributes: ['nombre', 'apellido']
});
```

4.6.4 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]: '%cia'}
  }
});
```

4.6.5 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]: '%cia' } },
      { [Op.not]: { id: [1, 2, 3] } }
    ]
  }
});
```

Podemos aplicar operadores (Ver una lista más exhaustiva aquí)

4.6.7 Paginación de resultados

Para manejar grandes volúmenes de datos, puedes implementar paginación en tus consultas:

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

4.6.8 Ordenar resultados

Para ordenar los resultados de tus consultas, puedes usar el atributo `order`:

```
const usuariosOrdenados = await Usuario.findAll({
  order: [['apellido', 'ASC']]
});
```

4.6.9 Recuperar con relaciones

Si tus modelos están relacionados, puedes recuperar datos junto con sus relaciones usando `include`:

```
const usuariosConDetalles = await Usuario.findAll({
  include: [{
    model: Perfil,
    as: 'perfil'
  }]
});
```

4.6.10 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');
```

4.6.11 Joins y relaciones avanzadas

Sequelize facilita la realización de consultas que implican relaciones entre tablas, permitiendo incluso configuraciones avanzadas de joins:

```
const usuariosConDirecciones = await Usuario.findAll({
  include: [{
    model: Direccion,
    as: 'direcciones',
    required: true  // Realiza un INNER JOIN
  }]
});
```

4.6.12 Consultas RAW

Para situaciones donde necesitas mayor flexibilidad, puedes ejecutar consultas SQL directas:

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

Podés examinar la documentación acá y acá para tener una idea más completa de las posibilidades que existen a la hora de recuperar información almacenada mediante **Sequelize**

4.7. Persistir datos

4.7.1. Creación o inserción de entidades

Para crear una instancia de una entidad utilizamos el método `.create()`.

```
const juan_perez = await Usuario.create({
    nombre: "Juan",
    apellido: "Perez",
    usuario: "jperez",
    fecha_alta: new Date(),
});
```

4.7.2. Actualización

Podemos actualizar una entidad que ya hemos recuperado previamente en otra operación:

```
const idMaria = 5;
const maria = await Usuario.findOne({ where: { id: idMaria } });
maria.nombre = 'María';
maria.apellido = 'García';
await maria.save();
```

o actualizarla directamente aplicando un criterio where para restringir la actualización a las filas que cumplen el criterio:

```
await Usuario.update({ apellido: "Perez" }, {
  where: {
   id: 5
  }
});
```

4.7.3. Eliminación

De forma análoga a la modificación, podemos eliminar una instancia que ha sido recuperada en otra operación anterior:

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

O eliminar loas instancias que cumplen determinado criterio expresado como `where`

```
await Usuario.destroy({
  where: {
    id: 5
  }
});
```

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

4.7.4. Manejo de transacciones

Las transacciones son críticas para mantener la integridad de los datos, especialmente en operaciones complejas:

```
await sequelize.transaction(async (t) => {
  const usuario = await Usuario.create({ nombre: 'Juan' }, { transaction: t });
  const perfil = await Perfil.create({ usuarioId: usuario.id, bio: 'Desarrollador'
}, { transaction: t });
});
```

5. CRUD con Sequelize

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

Bibliografía

Podés examinar la documentación acá.

Ayuda colaborativa

Podés ver algunos tips sobre configuraciones recomendadas acá.