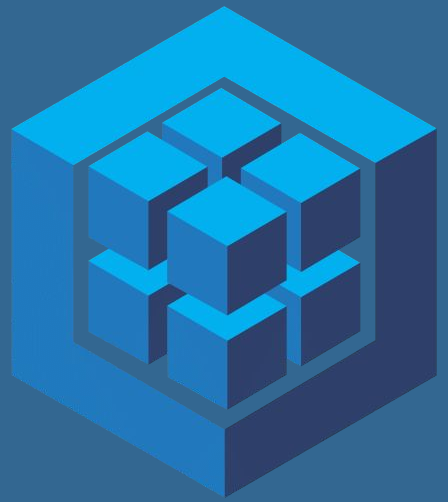
Sequelize

https://sequelize.org/



Sequelize ORM Object Relational Mapping

Que es **Sequelize**

Sequelize es un ORM orientado a NodeJS que funciona con distintos motores de Base de Datos (Oracle, Postgres, Mysql, y otros)

ORM (Object-Relational Mapping) es una técnica de programación que permite interactuar con bases de datos relacionales utilizando objetos en lugar de consultas SQL.

Esto permite a los desarrolladores trabajar con bases de datos de manera más intuitiva y eficiente dentro de lenguajes orientados a objetos.

Conectarse a **Sequelize**

Primero debemos instalar las librerías de Sequelize y postgres

npm i **pg**

npm i **sequelize**

Podemos utilizar el método **authenticate** para comprobar que estamos conectados correctamente

Para cerrar la conexión utilizamos el método **close**

```
const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname')
try {
    await sequelize.authenticate();
    console.log('Connection has been established successfully.');
  } catch (error) {
    console.error('Unable to connect to the database:', error);
  }
await sequelize.close()
```

Modelos

Un **modelo** es una abstracción de una entidad que está representado por una tabla en nuestra base de datos

En JavaScript ES6 los modelos son representados a través de **clases**.

Cada modelo tiene un nombre, que usualmente se pone en singular. La tabla asociada a ese modelo se suele representar en plural.

Ejemplo

Modelo: Usuario, Tabla: Usuarios

Para grabar el modelo que está en memoria en la base de datos se utiliza el método **sync**

Sequelize

Definición de un Modelo


```
class User extends Model {}
User.init(
  {
    id: {
        type: DataTypes.INTEGER,
        autoIncrement: true,
        primaryKey: true,
      },
    nombre: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    fechaNac: {
        type: DataTypes.DATE,
    }
  },
  {
    sequelize,
    modelName: 'User',
  },
);

await sequelize.sync({alter:"true"});
```

Instancias

Como ya sabemos, los **modelos** son **clases** en JavaScript

Una **instancia** de la clase es representada por un **objeto** que en Sequelize se mapea a un **row** (registro) de una **tabla** de la base de datos

Esto significa que al crear una **instancia** de un objeto, Sequelize internamente va a crear un **registro** en una tabla de nuestra base de datos

Crear instancias

Sequelize

A diferencia del operador **new** de JS, para crear una instancia de un objeto en Sequelize se debe utilizar el método **create**

Por ejemplo para crear una instancia de la clase User podemos hacer:

```
const leo =await User.create({nombre:"Lio",fechaNac:"06-22-1987"})
console.log(leo instanceof User) // True
console.log(leo.nombre) // Lio
```

Modificar instancias

Sequelize

Para modificar algún atributo de un objeto (modelo) se utiliza el operador “.” de javascript. Para que esto se vea reflejado en la base de datos es necesario llamar al método **save**()

```
leo.nombre = “Leonel”
console.log(leo.nombre) // “Leonel”
await leo.save()
```

Para directamente modificar el objeto en la base de datos y además agregar alguna condición existe el método **update**()

```
await User.update(
{‘edad’:’15’,’activo’:True},
{ where : {‘apellido’ : ‘Perez’}} )
```

Borrar instancias

Para borrar una instancia utilizamos el método **destroy()**

```
const leo =await User.create({nombre:"Lio",fechaNac:"06-22-1987"})
console.log(leo instanceof User) // True
console.log(leo.nombre) // Lio
await User.update({nombre:"Lionel"}, {where:{nombre:'Lio'}})
await leo.destroy()
```

Sequelize

Consultas

Para leer el contenido completo de una tabla utilizamos **findAll()**

```
const users = await User.findAll()
console.log("Usuarios:", JSON.stringify(users,null,2));
SELECT * from USUARIOS
```

Si queremos seleccionar solo algunos atributos podemos especificar

```
const users = await User.findAll({"attributes":['id','nombre']})
SELECT id, nombre from USUARIOS
```

Sequelize

Consultas

```
const users = await User.findAll({attributes:['id','nombre']})
console.log('Usuarios:', JSON.stringify(users,null,2));
```

```
Executing (default): SELECT "id", "nombre" FROM "Users" AS "User";
Usuarios: [
  {
    "id": 12,
    "nombre": "Anto"
  },
  {
    "id": 13,
    "nombre": "Thiago"
  },
  {
    "id": 11,
    "nombre": "Lionel"
  }
```

Sequelize

Consultas con WHERE

Veamos algunos ejemplos utilizando la propiedad ‘where’ del método findAll

const users = **await** User.**findAll**({“where”: {‘id’:3})

SELECT id, nombre from USUARIOS WHERE ID=3

const users = **await** User.**findAll**({“where”: {‘nombre’:’Leo’, ‘edad’:15})

SELECT id, nombre from USUARIOS WHERE nombre=’Leo’ and edad = 15

Consultas con WHERE


```
const users = await User.findAll(
    {attributes:['id','nombre'],
        where:{
            nombre:{[Op.like]:'%n%'}
        }
    })
console.log('Usuarios:', JSON.stringify(users,null,2));
```

```
Executing (default): SELECT "id", "nombre" FROM "Users" AS "User" WHERE "User"."nombre" LIKE '%n%';
Usuarios: [
  {
    "id": 12,
    "nombre": "Anto"
  },
  {
    "id": 11,
    "nombre": "Lionel"
```

Consultas con AND/OR

En el ejemplo anterior Sequelize asume que queremos hacer un “**and**”. Esto se puede especificar también de esta forma:

const users = await User.findAll({“where”:

[**Op.and**] : [{‘id’:3}, {‘edad’:15}])

Si queremos hacer un “or” se utiliza el Op de esta forma

El operador “op” nos permite hacer varias operaciones

const users = await User.findAll({“where”:

[**Op.or**] : [{‘id’:3}, {‘edad’:15}])

SELECT id, nombre from USUARIOS WHERE nombre=’Leo’ or edad = 15

Sequelize

Operadores

```
const { Op } = require("sequelize");
Post.findAll({
  where: {
    [Op.and]: [{ a: 5 }, { b: 6 }],            // (a = 5) AND (b = 6)
    [Op.or]: [{ a: 5 }, { b: 6 }],             // (a = 5) OR (b = 6)
    someAttribute: {
      // Basics
      [Op.eq]: 3,                              // = 3
      [Op.ne]: 20,                             // != 20
      [Op.is]: null,                           // IS NULL
      [Op.not]: true,                          // IS NOT TRUE
      [Op.or]: [5, 6],                         // (someAttribute = 5) OR (someAttribute = 6)

      // Using dialect specific column identifiers (PG in the following example):
      [Op.col]: 'user.organization_id',        // = "user"."organization_id"

      // Number comparisons
      [Op.gt]: 6,                              // > 6
      [Op.gte]: 6,                             // >= 6
      [Op.lt]: 10,                             // < 10
      [Op.lte]: 10,                            // <= 10
      [Op.between]: [6, 10],                   // BETWEEN 6 AND 10
      [Op.notBetween]: [11, 15],               // NOT BETWEEN 11 AND 15

      // Other operators

      [Op.all]: sequelize.literal('SELECT 1'), // > ALL (SELECT 1)

      [Op.in]: [1, 2],                         // IN [1, 2]
      [Op.notIn]: [1, 2],                      // NOT IN [1, 2]

      [Op.like]: '%hat',                       // LIKE '%hat'
      [Op.notLike]: '%hat',                    // NOT LIKE '%hat'
      [Op.startsWith]: 'hat',                  // LIKE 'hat%'
      [Op.endsWith]: 'hat',                    // LIKE '%hat'
      [Op.substring]: 'hat',                   // LIKE '%hat%'
```

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

Sequelize

Validaciones y constraints

Sequelize

Sequelize nos permite agregar **validaciones** y **constraints** a la definición de nuestros **modelos**.

Constraints son reglas definidas en lenguaje SQL. Por ejemplo se puede chequear que un valor sea único (unique constraint).

Ej: **unique, allowNull**

Validaciones son chequeos en que se escriben en JS. Se pueden utilizar para chequear formato de datos, tipos, valores válidos etc. Se resuelven en JS **sin interactuar con la base de datos.**

Ej: **equals, isInt, max**

Validaciones y constraints

Ejemplo

```
User.init(
  {
    id: {
        type: DataTypes.INTEGER,
        autoIncrement: true,
        primaryKey: true,
      },
    nombre: {
      type: DataTypes.STRING,
      allowNull: false,      // Constraint
      validate: {
        len: [5,10],         // Validation
        isUppercase: true    // Validation
    }
    },
    fechaNac: {
        type: DataTypes.DATE,
    }
  },
  {
    sequelize,
    modelName: 'User',
  },
);
```

Asociaciones

En sequelize las relaciones entre tablas (foreign keys) se llaman asociaciones.

Existen 4 tipos de asociaciones

```
const A = sequelize.define('A' /* ... */);
const B = sequelize.define('B' /* ... */);

A.hasOne(B); // A HasOne B
A.belongsTo(B); // A BelongsTo B
A.hasMany(B); // A HasMany B
A.belongsToMany(B, { through: 'C' }); // A BelongsToMany B through the junction table C
```

Asociaciones 1:1

Para crear una relación 1 a 1 se utiliza

hasOne y **belongsTo** recibiendo la otra tabla como parámetro para crear la asociación a nivel de modelo

```
class Piloto extends Model {}
class AutoF1 extends Model {}
AutoF1.belongsTo(Piloto)
Piloto.hasOne(AutoF1)
```

Sequelize

Asociaciones 1:N

Para crear una relación 1 a Muchos se utiliza

hasMany y **belongsTo**

La entidad que representa a muchos (N) se recibe como parámetro de hasMany y la que representa a 1 es parámetro de belongsTo

```
class Usuario extends Model {}

class Equipo extends Model {}

Usuario.belongsTo(Equipo)

Equipo.hasMany(Usuario)
```

Asociaciones 1:N

Sequelize

Una vez creada la relación, podemos insertar los datos a través de los métodos creados por Sequelize para nuestros modelos

a.setB(b) -> Asocia la instancia b a la instancia a en la BD

```
Equipo.hasMany(User)
User.belongsTo(Equipo)
await sequelize.sync({alter:"true"});

const inter = await Equipo.create({nombre:"Inter Miami"}) // Crea equipo
const jordi =await User.create({nombre:"Jordi",fechaNac:"5-11-2001"}) // Crea usuario
const suarez =await User.create({nombre:"Suarez",fechaNac:"5-1-1994"}) // Crea usuario
inter.addUsers([jordi, suarez]) // Asocia a Jordi y Suarez al equipo inter (setea FK en tabla Users)
```

a.getB() -> Me trae los objetos asociados a “a” de la BD

```
const equipo1 = await Equipo.findByPk(1)
const e1_users = await equipo1.getUsers()
console.log('Usuarios:', JSON.stringify(e1_users,null,2));
```

Asociaciones N:M

Para crear una relación Muchos a Muchos se utiliza

una doble asociación **belongsToMany**

En este caso hay que definir la tabla 'pivot', o de relación que se especifica con **through**.

Si la definimos como string, Sequelize va a crear la tabla automáticamente por nosotros conteniendo únicamente un id más las FK de las 2 otras tablas

class Canción extends Model {}

class Artista extends Model {}

Canción.**belongsToMany**(Artista, {**through**: 'CancionArtista'})

Artista.**belongsToMany**(Canción, {**through**: 'CancionArtista'})

Asociaciones N:M

Sin embargo si quisiéramos crear nosotros la tabla pivot (por ejemplo para agregar más datos a la misma) podemos definirla y pasar como parametro una variable que apunte al modelo.

class Canción extends Model {}

class Artista extends Model {}

class CancionArtista extends Model {} // Definimos manualmente la tabla y sus campos

Canción.**belongsToMany**(Artista, {**through**: CancionArtista})

Artista.**belongsToMany**(Canción, {**through**: CancionArtista})

Notar que CancionArtista es una variable ahora en vez de un string