**Table des matières :**

1. Model
2. CRUD operations Sequelize
   1. Create
   2. Read
   3. Update
   4. Delete
   5. Autres filtres, …
3. CRUD operation SQL
4. Associations
5. **model**

Pour définir un modèle, il existe plusieurs façons :

**A) Etendre la class Model** (de sequelize)

Class User extends Model {}

User.init(

{ id{…}, {type : DataTypes.STRING,… }, … }, // attribut du model

{ options } // autre options du model (ex : timestamp du dernier changement, validator pour le model complet)

)

OU

**B) Appeler la classe Sequelize**

sequelize.define(

"user",

{id {…}, name {…}},

{Autre options }

)

Behind the scene, sequelize.define calls Model.init

**Définir les getters setters**

Il est possible de modifier les getters/setters des champs d’un objet ex :

Sequelize.define(

'user', {

name {

type: …,

get(){return this.getDataValue('name').toUpperCase() },

set(value) { this.setDataValue('password'), hash(value) }

}

}

)

NB : les getters setters ne sont pas du côté db, mais uniquement coté JS

**Le type Virtual**

C’est un champ dans un model qui ne sera pas enregistré en db. Par contre, coté JS, c’est comme si ce champs existais en DB.

fullname {

type : DataTypes.VIRTUAL,

get(){return `${this.firstName} ${this.lastName}` },

set(value) { throw error }

}

1. **CRUD operations Sequelize**
   1. **Create**

Pour créer une instance d’un modèle, il existe aussi deux manières différentes :

**A) Build & Save**

const Jane = User.build ({ name: "Jane" }); // créer une instance JS, mais pas en DB

await Jane.save() ; //sauvegarde l’instance en DB

**B) Create**

const jane = await User.create({ name: "Jane" });

* 1. **Read**

Il y a une grande richesse dans les méthodes offerte par sequelize sur les lectures :

findById, findAll, findAndCountAll, findOne, findOrCreate

* 1. **Update**

**Objet JS**

Un objet JS déjà présent en mémoire, peux être modifié via l’appel set({name :"fred", …}) ou une assignation classique ( jane.name = "Fred")

Après avoir modifié un objet en JS, il faut appeler à nouveau save() sur l’objet, pour effectuer les changement en BD (await jane.save())  
Il est possible de ne modifier que certains champs, dans ce cas, il préciser ce champs dans save() (ex : jane.save({ fields: ['name'] }) )

TODO [Model Instances | Sequelize](https://sequelize.org/docs/v6/core-concepts/model-instances/) incrementation bizz

**Objet SQL**

Il est aussi possible de modifier un objet en DB sans le récuperer au préalable. Cela se fait via la fonction update.

* 1. **Delete**

Pour supprimer un objet, il suffit d’appeler destroy sur l’objet (await jane.destroy())

* 1. **Autres filtres, …**

Il est souvent possible de passer des paramètres dans les différentes méthodes du CRUD :

* ne renvoyer que certains champs (ex : {attributes: {['name', 'age'] }} )
* ne pas inclure certains champs (ex : {attributes: { exclude : ['private'] }} )
* filtrer les recherches (ex : {where: {name : "Fred"}} ) (il existe de nombreux opérateurs (OR, AND, startsWith,…)
* faire des agrégations (ex : {attributes: { ['name', 'age', sequelize.fn(‘COUNT’, sequelize.col('prevJobs')), 'alias\_count' ] }} ) (l’alias permet de recup ce champs via jane.alias\_count) (sequelize.fn contient ne nombreuse fonctions d’agrégation varié)
* trier les résultats (ex : {order : ['name', 'DESC']})
* grouper les résultats (ex : {group : 'age'}) (groupe by age)
* limite, offset (ex : { offset: 5, limit: 5 } )

Ces filtre s’appel de cette manière User.findAll({where: {name : "Fred"}})

1. **CRUD operation SQL**

Il est aussi possible de se passer de la majorité des outils sequelize et de passer par des query SQL classique.

En voici un exemple :

const users = await sequelize.query("SELECT \* FROM `users`", { type: QueryTypes.SELECT });

TODO plein de truc : [Raw Queries | Sequelize](https://sequelize.org/docs/v6/core-concepts/raw-queries/)

1. **Associations**

Les associations ne sont peut-être pas le point fort de sequelize : là où en SQL, il ne faut les définir uniquement via contrainte foreign key une seule fois, avec sequelize, il va souvent falloir définir deux fois entre deux tables, et encore bien plus si on rajoute, par exemple, une table intermédiaire.

Voici les associations disponibles :

User.hasOne(IdCard); // (1 à 1) IdCard a une FK vers User

Dog.belongsTo(Owner); // (1 à 1) Dog a une FK vers Owner

Owner.hasMany(Dog); // (1 à N) un Owner possède plusieurs chien

Business.belongsToMany(Investor, { through: 'Biz\_Investor' }); // (M à N) Business appartient à plusieurs Investor via la table intermédiaire Biz\_Investor

// TODO [Associations | Sequelize](https://sequelize.org/docs/v6/core-concepts/assocs/)

// TODO [Advanced M:N Associations | Sequelize](https://sequelize.org/docs/v6/advanced-association-concepts/advanced-many-to-many/)