



# Bases de Datos: Sequelize

5-3-2020 - V6

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Bases de Datos y ORM sequelize - Índice

1.	<a href="#"><u>Introducción a las Bases de Datos .....</u></a>	<a href="#"><u>3</u></a>
2.	<a href="#"><u>SGBDRs y ORMs para JavaScript: Sequelize y Sqlite .....</u></a>	<a href="#"><u>8</u></a>
3.	<a href="#"><u>Modelo, acceso y gestión de instancias .....</u></a>	<a href="#"><u>18</u></a>
4.	<a href="#"><u>Proyecto user-quiz: Tabla Users, Modelo y Comandos .....</u></a>	<a href="#"><u>23</u></a>



# Introducción a las Bases de Datos

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Base de Datos

## ◆ **BBDD** - Base de datos (**DB** - Data Base)

- Es una colección organizada de datos o de información
  - ◆ Son los almacenes donde se guardan las enormes cantidades de datos generados en Internet

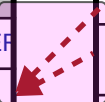
## ◆ **SGBD** - Sistema Gestor de BBDD (**DBMS** - Data Base Manag. Syst.)

- Conjunto de programas o librerías para definir, administrar y acceder a los datos de BBDDs

## ◆ BBDDs y SGBDs solucionan el **almacenamiento masivo** de datos, permitiendo

- Albergar y gestionar grandes repositorios de datos de forma **persistente**
- Representar **informaciones complejas** con las estructuras de datos mas adecuadas
- Garantizar la **integridad** y la **consistencia** de los datos
- **Compartir** los datos entre múltiples usuarios y aplicaciones
- Implementar soluciones de **seguridad** con control de acceso, encriptación, auditoría, ...
- ....

# SGDBR y SQL



id	username	password	salt	isAdmin
1	admin	r34et5690y	"aaaa"	VERDADE
2	pepe	56gh90op5	"bbbb"	FALSO
3	.....	.....	.....	.....

id	question	answer	authorId
1	Capital de Italia	Roma	2
2	Capital de Portugal	Lisboa	2
3	.....	.....	.....
4	.....	.....	.....

## ◆ SGBDR (Sistema Gestor de Base de Datos Relacional)

- Basadas en el modelo de Entidad-Relación y en el Calculo de Predicados de 1er orden
  - ♦ Representa los datos como **tuplas** (o **registros**) guardadas en las filas de **tablas relacionadas** entre sí
    - [https://en.wikipedia.org/wiki/Relational\\_database](https://en.wikipedia.org/wiki/Relational_database),
    - [https://en.wikipedia.org/wiki/Entity\\_relationship\\_model](https://en.wikipedia.org/wiki/Entity_relationship_model)

## ◆ SQL - Structured Query Language

- Lenguaje de definición, manipulación y control de datos en BBDDs relacionales
  - ♦ **SQL** es una **API normalizada** que permite acceder a cualquier **SGDBR**
    - <https://en.wikipedia.org/wiki/SQL>

## ◆ SGBDR mas habituales

- **MySQL, Postgres, SQLite, MariaDB, Microsoft SQL Server, Oracle, ....**
  - ♦ Mas información en [https://en.wikipedia.org/wiki/Relational\\_database\\_management\\_system](https://en.wikipedia.org/wiki/Relational_database_management_system)

# SGBDR: Modelo, clave primaria y clave externa

## ◆ Modelo **relacional** de datos

- Los datos se estructuran en tuplas de información relacionadas entre si
  - ♦ Las tuplas de un mismo tipo (misma estructura) se guardan en **tablas**

## ◆ **Tabla** (o modelo)

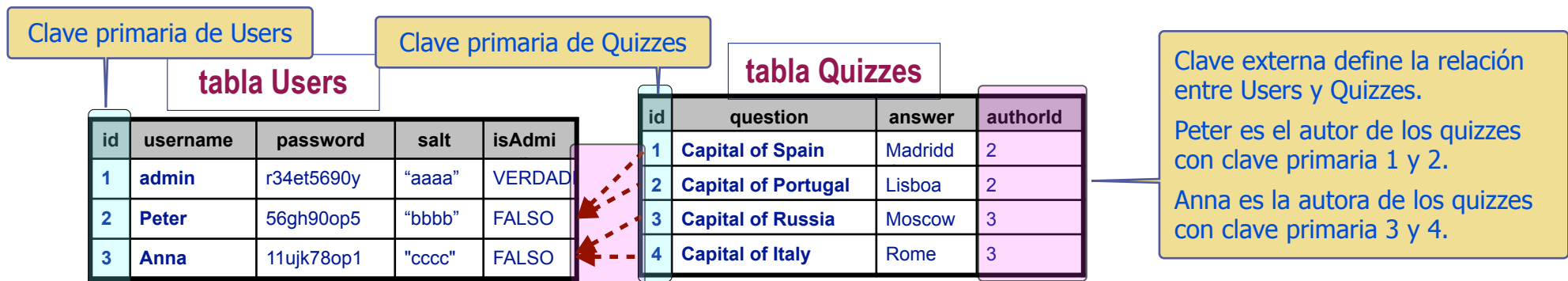
- Cada **fila** (o **registro**) de la tabla contiene los **campos** o **atributos** de una tupla de información

## ◆ **Clave primaria**

- Clave que identifica unívocamente cada fila (registro) de la tabla
  - ♦ Una clave primaria no puede estar repetida en una tabla

## ◆ **Relaciones y clave externa**

- Las relaciones asocian datos de unas tablas con los de otras tablas gestionan con claves externas que se añaden a la tabla
  - ♦ La clave externa contiene la clave primaria del elemento de otra tabla con el que está relacionado



# Bases de Datos NoSQL

## ◆ BBDDs NoSQL

- Son BBDDs que se acceden por APIs (Application Programming Interfaces) diferentes a SQL
  - ◆ Actualmente se prefiere utilizar el termino **NotOnlySQL**, porque muchas permiten también acceso SQL
    - <https://en.wikipedia.org/wiki/NoSQL>, <https://es.wikipedia.org/wiki/NoSQL>
- NoSQL incluye muchos tipos diferentes de BBDDs:
  - ◆ Clave-valor, de-grafos, de-documentos, multi-modales, de-objetos, tabulares, ...

## ◆ BBDDs clave-valor

- Son arrays asociativos (mapas o diccionarios) donde una clave única identifica cada valor
  - ◆ Algunos SGBDs: Redis, Oracle NoSQL, InfinityDB, ArangoDB, .....

## ◆ BBDDs de-grafos

- Están optimizadas para representar y procesar grafos de datos e información
  - ◆ Algunos SGBDs: Neo4j, AllegroGraph, ArangoDB, Oracle, FlockDB, InfiniteGraph, OrientDB, .....

## ◆ BBDDs de-documentos o de-objetos

- Almacenan y procesan documentos en formatos tipo JSON, XML, YAML, BSON, ....
  - ◆ Algunos SGBDs: MongoDB, ArangoDB, CouchDB, IBM Domino, InfiniteGraph, OrientDB, .....



JavaScript

# ORMs y SGBDRs para node.js:

## Sequelize y Sqlite

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM



# ORM y Modelo

## ◆ ORM - Object Relational Mapping

- Encapsula valores y comandos **SQL** como un **modelo** Orientado a Objetos (OO)
  - ◆ Permite **acceder** de forma sencilla a **SGBDRs** con **SQL**, p.e. Oracle, Postgres, sqlite, MySQL, ....
    - [https://en.wikipedia.org/wiki/Object-relational\\_mapping](https://en.wikipedia.org/wiki/Object-relational_mapping)
    - [https://es.wikipedia.org/wiki/Mapeo\\_objeto-relacional](https://es.wikipedia.org/wiki/Mapeo_objeto-relacional)
- Existen diversos ORMs para node.js: **sequelize**, node-orm2, Bookshelf...
  - ◆ <https://www.codediesel.com/javascript/nodejs-mysql-orms/>

## ◆ Modelo

- **Clase** (u objeto) que permite acceder a una **tabla** usando **SQL**

## ◆ Sequelize

- **ORM** mas popular para crear **modelos** en **node.js**
  - ◆ Instrucciones de instalación y uso: <http://sequelizejs.com/>

```
..$  
..$ npm install sequelize@5.21.3    // instala version 5.21.3 de sequelize  
..$                                // Si existe package.json añade una dependencia
```

# SQLite

## ◆ SQLite es un SGBDR muy extendido

- Es sencillo, fiable, completo, de pequeño tamaño y no **necesita configuración**
  - ◆ Soporta cualquier **transacción SQL**, incluyendo ACID (Atomicity, Consistency, Isolation, Durability)
    - <https://www.sqlite.org/index.html>
- **SQLite** se suele utilizar en las **fases de desarrollo** de un proyecto

## ◆ Existen implementaciones para prácticamente todos los entornos

- Para instalar **SQLite** en **Node.js** utilizamos el paquete npm **sqlite3**
  - ◆ Ver: <https://www.npmjs.com/package/sqlite3>

```
..$  
..$ npm install sqlite3@4.0.9    // instala version 4.0.9 de sqlite3  
..$                             // Si existe package.json añade una dependencia
```

## ◆ En despliegue y operación se utilizan otros **SGBDRs** mas escalables

- **Postgres** (o **PostgreSQL**) tiene acceso, tanto relacional (SQL), como clave-valor
  - ◆ **Postgres** se utilizará para **despliegue** en Heroku: <https://www.postgresql.org/>

# Importar Sequelize y conexión a la BBDD

## ◆ Importar Sequelize

- Se importa y se extraen estos 3 objetos por multi-asignación ES6
  - ♦ **Sequelize**: clase para crear la conexión a una base de datos: SQLite, Postgres, ....
  - ♦ **Model**: clase para construir el modelo por extensión
  - ♦ **DataTypes**: objeto con todos los tipos de atributos o campos de un registro de una tabla

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

## ◆ Definir la conexión a la **BBDD** SQLite

- **const sequelize = new Sequelize("sqlite:db.sqlite", <options>)**
  - ♦ **"sqlite:db.sqlite"**: URL de acceso a una BBDD SQLite
    - **sqlite** indica que es una BBDD de tipo SQLite
    - **db.sqlite** es la ruta relativa al fichero que albergará la BBDD
  - ♦ **<options>**: opciones de configuración de SQLite

```
const options = { logging: false};  
const sequelize = new Sequelize("sqlite:db.sqlite", options);
```

# Crear el modelo y construir la BBDD

## ◆ Crear la clase **User** por extensión de **Model**

- **class User extends Model {};**

- ♦ <https://sequelize.org/master/manual/model-basics.html#model-definition>

## ◆ Definir los **atributos** o **campos** de la **tabla**

- **User.init(<attributes>, <options>);**

- ♦ **<attributes>**: define los nombres de los **campos** y su tipo

- Los tipos de atributos se definen con ayuda de DataTypes  
[sequelize.org/master/manual/model-basics.html#data-types](https://sequelize.org/master/manual/model-basics.html#data-types)

- ♦ **<options>**:

- Debe incluir el **enlace** a la BBDD (sequelize),, además se hace explícito el nombre (por defecto) del modelo

```
class User extends Model {}

User.init(
  {
    name: DataTypes.STRING,
    age: DataTypes.INTEGER
  },
  { sequelize, modelName: "User" }
);
```

## ◆ Sincronizar la definición del modelo con la BBDD

- **sequelize.sync()**

- ♦ Construye (si no existe) la **tabla Users** en la BBDD (archivo **db.sqlite**)

- Una vez construida, la BBDD puede utilizarse

**tabla Users**

name	age

## ◆ Un **modelo** es una clase que permite gestionar una tabla

- Por defecto, el nombre de la **tabla** es el **plural** del nombre del modelo

## ◆ Todos los **modelos** definidos se guardan en **sequelize.models**

- Se guardan con el **nombre del modelo** o con lo indicado en la **propiedad "modelName"**

# Algunos tipos de datos de las tablas

```
DataType.STRING // VARCHAR(255)
DataType.STRING(1234) // VARCHAR(1234)
DataType.STRING.BINARY // VARCHAR BINARY
DataType.TEXT // TEXT
DataType.TEXT('tiny') // TINYTEXT
```

```
DataType.BOOLEAN // TINYINT(1)
```

```
DataType.INTEGER // INTEGER
DataType.BIGINT // BIGINT
DataType.BIGINT(11) // BIGINT(11)

DataType.FLOAT // FLOAT
DataType.FLOAT(11) // FLOAT(11)
DataType.FLOAT(11, 10) // FLOAT(11,10)

DataType.REAL // REAL
DataType.REAL(11) // REAL(11)
DataType.REAL(11, 12) // REAL(11,12)

DataType.DOUBLE // DOUBLE
DataType.DOUBLE(11) // DOUBLE(11)
DataType.DOUBLE(11, 10) // DOUBLE(11,10)

DataType.DECIMAL // DECIMAL
DataType.DECIMAL(10, 2) // DECIMAL(10,2)
```

```
DataType.DATE
DataType.DATE(6)
DataType.DATEONLY
```

**DataType** permite definir los diferentes tipos de campos de una tabla. Algunos son específicos de BBDDs determinadas como SQLite, PostgreSQL,... Ver: <https://sequelize.org/master/manual/model-basics.html#data-types>

tabla Users

name	age
Peter	22
Anna	23
John	30

```
class User extends Model {
```

```
  User.init(
    {
      name: DataTypes.STRING,
      age: DataTypes.INTEGER,
    },
    { sequelize }
  );
```

El tipo STRING permite guardar strings de hasta 255 caracteres.

El tipo INTEGER define números enteros con 32 bits de precisión.

Este ejemplo define la **tabla User** de una BBDD utilizando las facilidades de la clase **Model**. Ver: <https://sequelize.org/master/manual/model-basics.html#taking-advantage-of-models-being-classes>

Hay otras formas de definir tablas: <https://sequelize.org/master/manual/model-basics.html>

# Estructura de la tabla, validación y opciones

## ◆ Modelo: define las columnas de usuario

- **Propiedad:** nombre del campo (**name** y **age**)
- **Valor:** define el tipo (**STRING**, **INTEGER**, .. )

## ◆ Restricciones

- Establecen requisitos SQL a columnas del modelo
  - ♦ Por ejemplo: **unique**, **allowNull**, **defaultValue**, **validate**, ...
    - <https://sequelize.org/master/manual/validations-and-constraints>
  - ♦ Lanzas una **excepción** (con **msg**) si la restricción no se cumple

## ◆ Validaciones

- Restringen el contenido de un campo del modelo
  - ♦ Son funciones predefinidas definidas en la propiedad **validate**
    - <https://sequelize.org/master/manual/validations-and-constraints.html#validators>
  - ♦ Lanzas una **excepción** (con **msg**) si la restricción no se cumple

## ◆ Sequelize crea 3 columnas adicionales a las del modelo

- **id:** índice o clave primaria de una instancia
- **createdAt:** fecha de creación de la instancia
- **updatedAt:** fecha de última actualización

```
{ name: {
  type: DataTypes.STRING,
  unique: { msg: "Name already exists"},
  allowNull: false,
  validate: {
    isAlphanumeric: { args: true, msg: "name: invalid characters"}
  }
},
age: {
  type: DataTypes.INTEGER,
  allowNull: false,
  validate: {
    isInt: true,
    min: { args: [0], msg: "Age: less than 0"},
    max: { args: [140], msg: "Age: higher than 140"}
  }
}
},
}
```

tabla Users (completa)

id	name	age	createdAt	updatedAt
1	Peter	22	2017-12-26T19:29:09.286Z	2017-12-26T19:29:09.286Z
2	Anna	23	2017-12-26T22:03:12.616Z	2017-12-26T22:03:12.616Z
3	John	30	2017-12-26T24:05:12.616Z	2017-12-26T24:05:12.616Z

```
sequelize.define('foo', {
  bar: {
    type: DataTypes.STRING,
    validate: {
```

Sequelize incluye funciones de validación para rechazar entradas no validas (ver):  
<https://sequelize.org/master/manual/validations-and-constraints.html#per-attribute-validations>

# Sequelize: validaciones

```
is: /^[a-z]+$/i,
is: ["^[a-z]+$",'i'],
not: /^[a-z]+$/i,
not: ["^[a-z]+$",'i'],
isEmail: true,
isUrl: true,
isIP: true,
isIPv4: true,
isIPv6: true,
isAlpha: true,
isAlphanumeric: true,
isNumeric: true,
isInt: true,
isFloat: true,
isDecimal: true,
isLowercase: true,
isUppercase: true,
notNull: true,
isNull: true,
notEmpty: true,
equals: 'specific value',
contains: 'foo',
notIn: [['foo', 'bar']],
isIn: [['foo', 'bar']],
notContains: 'bar',
len: [2,10],
isUUID: 4,
isDate: true,
isAfter: "2011-11-05",
isBefore: "2011-11-05",
max: 23,
min: 23,
isCreditCard: true,
```

```
// matches this RegExp
// same as above, but constructing the RegExp from
// does not match this RegExp
// same as above, but constructing the RegExp from
// checks for email format (foo@bar.com)
// checks for url format (http://foo.com)
// checks for IPv4 (129.89.23.1) or IPv6 format
// checks for IPv4 (129.89.23.1)
```

```
{ name: {
  type: DataTypes.STRING,
  unique: { msg: "Name already exists"},
  allowNull: false,
  validate: {
```

**unique:** indica que el nombre no puede repetirse. Lanza excepción con msg, si e intenta introducir un nombre repetido.

Si se introduce **falsey**, lanza excepción.

```
isAlphanumeric: { args: true, msg: "name: invalid characters"}
```

Si el nombres no es alfanúmerico, se lanza excepción con msg.

```
age: {
  type: DataTypes.INTEGER,
  allowNull: false,
  validate: {
```

Si se introduce **falsey**, lanza excepción.

Si age no es entero, lanza excepción.

```
isInt: true,
min: { args: [0], msg: "Age: less than 0"},
max: { args: [140], msg: "Age: higher than 140"}
```

Si age es menor que 0 o mayor que 140, lanza excepción con msg correspondiente.

# Modelo y acceso a la BBDD

```
class User extends Model {}  
  
User.init(  
  { name: DataTypes.STRING,  
    age: DataTypes.INTEGER,  
  },  
  { sequelize }  
);
```

## ◆ La clase **User** es el modelo de la tabla **Users**

- Los **registros** de la tabla se representan en JavaScript como **objetos**, donde
  - ◆ Las **propiedades** de las instancias tienen el nombre de los **campos** de la tabla
    - Por ejemplo `{name:"Peter", age:"22"}`

## ◆ **User.count()**

- Promesa que devuelve el número de **registros** de la tabla **Users**

## ◆ **User.bulkCreate(<instances>)**

- Promesa que añade a la tabla '**people**' un array de instancias de User
  - ◆ <instances> -> [{name:"Peter", age:"22"}, {name:"Anna", age:"23"}, {name:"John", age:"30"}]

## ◆ **User.findAll()**

- Promesa que devuelve un array de objetos con todos los registros de la tabla **Users**

**tabla Users**

name	age
Peter	22
Anna	23
John	30



Importar la librería sequelize.js y extraer por multi-asignación:

- **Sequelize**: constructor para crear la BBDD (sequelize).
- **Model**: clase para crear modelos por extensión.
- **DataTypes**: tabla con los tipos de registros del modelo.

Crear enlace a la BBDD configurada para sqlite.

# Ejemplo: 60-users.js

Para poder utilizar promesas con async/await se crea esta función async cuyo bloque de código se ejecuta como un IIFE (Immediately invoked función expresión).

El bloque de código se incluye en el bloque try, para capturar los rechazos con catch.

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

```
const options = { logging: false};  
const sequelize = new Sequelize("sqlite:db.sqlite", options);
```

```
class User extends Model {}
```

Crear clase **User** extendiendo **Model**.

```
User.init(  
  { name: {  
    type: DataTypes.STRING,  
    unique: { msg: "Name already exists"},  
    allowNull: false,  
    validate: {  
      isAlphanumeric: { args: true, msg: "name: invalid"}  
    }  
  },  
  age: {  
    type: DataTypes.INTEGER,  
    allowNull: false,  
    validate: {  
      isInt: true,  
      min: { args: [0], msg: "Age: less than 0"},  
      max: { args: [140], msg: "Age: higher than 140"}  
    }  
  }  
},  
  { sequelize }  
);
```

Definir tabla **User** con campos **name** y **age**, junto con validaciones y restricciones.

```
$  
$ node 60-users.js  
DB exists & has 3 elems  
  
Peter is 22 years old  
Anna is 23 years old  
John is 30 years old  
$
```

```
(async () => {  
  try {
```

```
    // Initialize the database
```

```
    await sequelize.sync();  
    let count = await User.count();  
    if (count===0) {  
      let c = await User.bulkCreate([  
        { name: 'Peter', age: '22'},  
        { name: 'Anna', age: 23},  
        { name: 'John', age: 30}  
      ]);  
      process.stdout.write(` DB created with ${c.length} elems\n\n`);  
    } else {  
      process.stdout.write(` DB exists & has ${count} elems\n\n`);  
    }  
  };
```

Sincronizar la BBDD (archivo **db.sqlite**) con el modelo utilizando **sequelize.sync()**.

Si la BBDD está vacía (**count===0**), se introducen los 3 primeros usuarios con **User.bulkCreate(..)** y se informa por consola, que se ha creado la BBDD.

Si la BBDD ya existe se informa de ello por consola.

```
    // Show database content
```

```
    let users = await User.findAll()  
    users.forEach(u =>  
      console.log(` ${u.name} is ${u.age} years old`)  
    );
```

Se leen todos los registros de la **tabla Users** con **Users.findAll()**.

```
  } catch (err) {  
    console.log(err)  
  }  
})();
```

Mostrar por consola el array con todos los registros de la **tabla Users**.

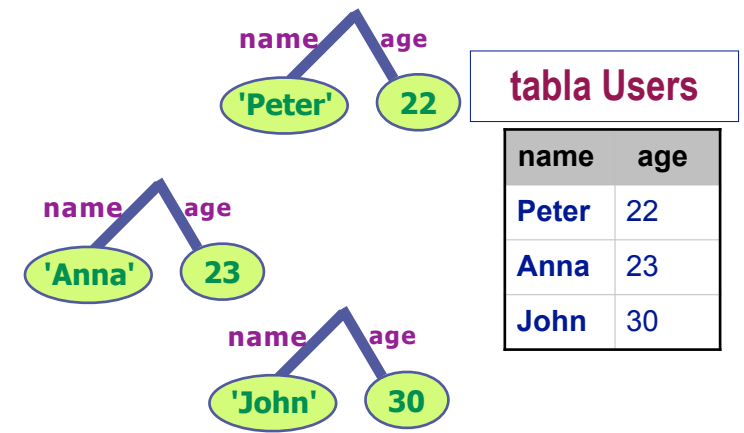
Captura rechazos de promesas, excepciones o errores.



# Modelo, acceso y gestión de instancias

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Modelo, instancias y consultas



## ◆ User es el modelo de la tabla Users

- La **instancia** de un **registro de la tabla** es un **objeto** donde
  - ◆ Los **nombres y contenidos** de las **propiedades** coinciden con los de los **atributos**

## ◆ Las **instancias** de **User** se crean con **build()** y se guardan con **save()**, p. e.

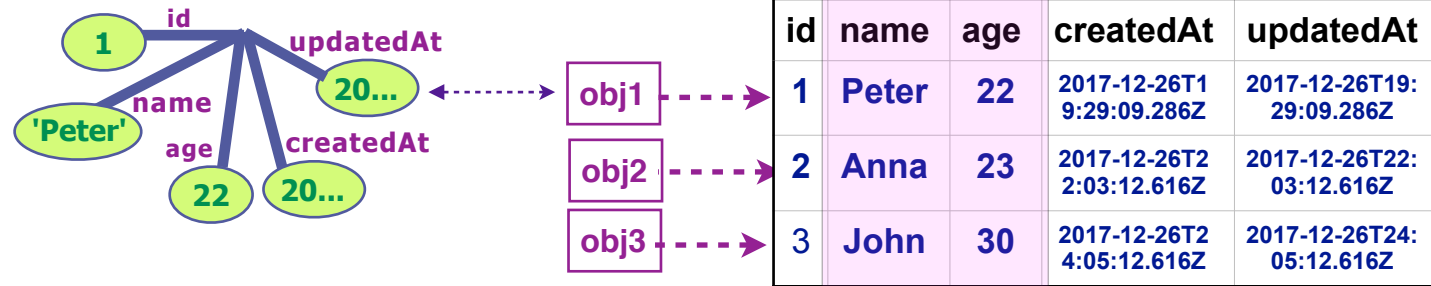
- `let user = await User.build({name:"Peter", age:"22"});` // crea instancia no persistente
- `user.save();` // guarda la instancia en la tabla

## ◆ **create()** permite crear instancia (**build()**) y guardarla (**save()**) en una operación

- `let user = await User.create({name:"Peter", age:"22"});` // crea instancia y la guarda

## ◆ **Ojo!** Los objetos de instancia **nunca** deben crearse con **new User(..)**

# Consultas a la tabla



## ◆ Ejemplos de métodos de búsqueda de sequelize

- **User.findAll()** -> [ obj1, obj2, obj3 ]
  - ◆ Devuelve un array con todos los objetos de la tabla Users
- **User.findOne()** -> obj1
  - ◆ Devuelve el primer objeto que encuentra en la tabla Users
- **User.findByPk(2)** -> obj2
  - ◆ Devuelve el objeto con la clave primaria 2 en la tabla Users
- **User.find()** -> obj1
  - ◆ Se mantiene por compatibilidad hacia atrás, emula **findOne(..)**, **findById(2)**, ..
- **User.count()** -> 3
  - ◆ Devuelve el número de objetos en la tabla Users
- Documentación
  - ◆ <https://sequelize.org/master/manual/model-querying-finders.html>

# Métodos de actualización persistente

	id	name	age	createdAt	updatedAt
obj1	1	Peter	22	2017-12-26T19:29:09.286Z	2017-12-26T19:29:09.286Z
obj2	2	Anna	23	2017-12-26T22:03:12.616Z	2017-12-26T22:03:12.616Z
obj3	3	John	30	2017-12-26T24:05:12.616Z	2017-12-26T24:05:12.616Z

tabla Users (completa)

## ◆ Métodos para actualización persistente

- **User.create(<obj>) y User.bulkcreate(<obj\_array>)**
  - ♦ Crea uno o varios registros nuevos en la tabla Users
- **let user = await User.build(<obj>) y user.save()**
  - ♦ **build()** crea una instancia no persistente con la estructura de la tabla Users y **save()** la guarda de forma persistente en la tabla (equivale a User.create(<obj>))
- **let user = await User.findOne(..) y user.save(<options>)**
  - ♦ **findOne()** busca una instancia en la tabla y **save()** la guarda de forma persistente en la tabla las modificaciones que se hayan introducido en la instancia
- **User.update(<values>, <options>)**
  - ♦ Actualiza varios objetos de la tabla Users
- **User.findOrCreate(<options>)**
  - ♦ Devuelve la instancia pedida o la crea y la devuelve si no existe
- **User.destroy(<options>)**
  - ♦ Destruye varios objetos de la tabla Users

## ◆ Documentación

- <https://sequelize.org/master/manual/model-instances.html>

# Opciones de búsqueda

	obj1	----->	1	Peter	23	2017-12-26T19:29:09.286Z	2017-12-26T19:29:09.286Z
	obj2	----->	2	Anna	22	2017-12-26T22:03:12.616Z	2017-12-26T22:03:12.616Z
	obj3	----->	3	John	30	2017-12-26T24:05:12.616Z	2017-12-26T24:05:12.616Z

tabla Users (completa)

◆ Las opciones de búsqueda mas habituales son

- **where:** fijar condiciones en la búsqueda
  - ◆ `User.findAll({ where: { name: {[Op.substring]: 'n'}}})` -> [ obj2, obj3 ] - contains string 'n'
  - ◆ `User.findAll({ where: { age: {[Op.lt]: 25}} })` -> [ obj1, obj2 ] - age lessThan 25
  - ◆ `User.count({ where: { age: {[Op.lt]: 25}} )` -> 2
  - ◆ `User.delete({ where: { name: "Anna"}})` -> deletes obj2
- **limit and offset:** comienzo de búsqueda y máximo número de objetos
  - ◆ `User.findAll({ offset: 2, limit:2})` -> [ obj2, obj3 ]
- **order:** ordenar objetos devueltos
  - ◆ `User.findAll({ order: "age"})` -> [ obj2, obj1, obj3 ]
  - ◆ `User.findAll({ order: [ "id", "DESC" ]})` -> [ obj3, obj2, obj1 ]
- Documentación
  - ◆ <https://sequelize.org/master/manual/model-querying-basics.html>



# Proyecto user-quiz:

## Tabla Users, Modelo y Comandos

Juan Quemada, DIT - UPM

# Users

**README.md:** doc e instrucciones.

**main.js:** programa principal.

**package.json:** Paquete npm

Name

README.md  
package.json  
package-lock.json  
model.js  
main.js  
cmds\_user.js

## Download, instalation and usage

This branch can be downloaded, installed and run as follows:

```
$ git clone -b user https://github.com/CORE-UPM/user_quiz  
$ cd user_quiz
```

```
$ npm install
```

```
$ npm start ## or 'node main'
```

```
....  
> [h]
```

Commands (params are requested after):

```
> [h] ## show help  
>  
> [lu] | [ul] | [u] ## users: list all  
> [cu] | [uc] ## user: create  
> [ru] | [ur] | [r] ## user: read (show age)  
> [uu] ## user: update  
> [du] | [ud] ## user: delete  
>  
> [e] ## exit & return to shell  
>
```

**git clone ....:** descarga un proyecto (directorio) con los ficheros de la captura.

**model.js:** Definición de la tabla **Users**.

**tabla Users**

name	age
Peter	22
Anna	23
John	30

**cmds\_user.js:** Implementación de comandos.

Paquete en GitHub: [https://github.com/CORE-UPM/user\\_quiz/tree/user](https://github.com/CORE-UPM/user_quiz/tree/user)

```
venus:user_quiz jq$  
venus:user_quiz jq$ node main  
> DB created with 3 elems  
>  
> [u]  
[ Peter is 22 years old  
[ Anna is 23 years old  
[ John is 30 years old  
>  
> [r]  
Enter name: Anna  
Anna is 23 years old  
>  
> [cu]  
Enter name: Eva  
Enter age: 11  
Eva created with 11 years  
>  
> [uu]  
Enter name to update: Anna  
Enter new name: Ana  
Enter new age: 23  
Anna updated to Ana, 23  
>  
> [du]  
Enter name: John  
John deleted from DB  
>  
> [u]  
Peter is 22 years old  
Ana is 23 years old  
Eva is 11 years old  
>  
> [e]  
Bye!  
venus:user_quiz jq$
```





# package.json

```
{
  "name": "user_quiz",
  "version": "1.0.0",
  "description": "Educational node.js project to start with DBs, sequelize & sockets",
  "main": "main.js",
  "scripts": {
    "start": "node main.js"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/CORE-UPM/user_quiz"
  },
  "author": "Juan Quemada",
  "license": "GNUv3 - General Public License version 3",
  "bugs": {
    "url": "https://github.com/CORE-UPM/user_quiz/issues"
  },
  "homepage": "https://github.com/CORE-UPM/user_quiz#readme",
  "dependencies": {
    "sequelize": "^5.21.3",
    "sqlite3": "^4.0.9"
  }
}
```

Un fichero **package.json** en raíz de un proyecto indica que es un paquete npm. Si no existe se crea con el comando:

**\$ npm init**

Este comando pide los parámetros a través de la consola. Desde node 5.2, se puede crear con **\$ npx create-react-app <ruta-a-proyecto>** que crea el directorio del proyecto y su package.json, si no existen.

Los scripts son comandos ejecutables (desde node 5.2) con:

**\$ npm run-script <comando>**

Como start es muy habitual permite la sintaxis abreviada:

**\$ npm start**

Ambas equivalen a ejecutar la shell el comando indicado:

**\$ node main.js**

Las dependencias de los paquetes **sqlite3** y **sequelize** se añaden a **package.json** con:

**\$ npm install sequelize@5.21.3 sqlite3@4.0.9**

Instala ambos paquetes en **node\_modules** y si existe **package.json** añade las dependencias. Así, cuando se copia o clona el paquete, la instalación se puede regenerar desde cero con:

**\$ npm install**

Este comando crea **node\_modules** e instala las dependencias y lo prepara para ejecutarlo. 25

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

```
const options = { logging: false};  
const sequelize = new Sequelize("sqlite:db.sqlite", options);
```

```
class User extends Model {}
```

```
User.init(  
  { name: {  
    type: DataTypes.STRING,  
    unique: { msg: "Name already exists"},  
    allowNull: false,  
    validate: {  
      isAlphanumeric: { args: true, msg: "name: invalid characters"}  
    }  
  },  
  age: {  
    type: DataTypes.INTEGER,  
    allowNull: false,  
    validate: {  
      isInt: true,  
      min: { args: [0], msg: "Age: less than 0"},  
      max: { args: [140], msg: "Age: higher than 140"}  
    }  
  },  
  { sequelize }  
);
```

Los comandos de esta app permiten todas las ops del interfaz CRUD.

```
>  
> h  
Commands (params are requested after):  
  > h          ## show help  
  >  
  > lu | ul | u  ## users: list all  
  > cu | uc      ## user: create  
  > ru | ur | r  ## user: read (show age)  
  > uu          ## user: update  
  > du | ud      ## user: delete  
  >  
  > e          ## exit & return to shell  
>
```

Crear enlace a la BBDD configurada para sqlite.

Importar la librería sequelize.js y extraer por multi-asignación:

- **Sequelize**: constructor para crear la BBDD (sequelize).
- **Model**: clase para crear modelos por extensión.
- **DataTypes**: tabla con los tipos de registros del modelo.

## tabla Users

name	age
Peter	22
Anna	23
John	30

## model.js

Para poder utilizar promesas con async/await se crea esta función async cuyo bloque de código se ejecuta como un IIFE (Immediately invoked función expresión).

Sincronizar la BBDD (archivo **db.sqlite**) con el modelo utilizando **sequelize.sync()**.

(**count===0**) indica BBDD vacía, introduciéndose los 3 primeros usuarios con **User.bulkCreate(..)**. Al final se informa por consola.

```
// Initialize the database  
(async () => {  
  try {  
    await sequelize.sync();  
    let count = await User.count();  
    if (count===0) {  
      let c = await User.bulkCreate([  
        { name: 'Peter', age: '22'},  
        { name: 'Anna', age: 23},  
        { name: 'John', age: 30}  
      ]);  
      process.stdout.write(` DB created with ${c.length} elems\n`);  
      return;  
    } else {  
      process.stdout.write(` DB exists & has ${count} elems\n`);  
    }  
  } catch (err) {  
    console.log(` ${err}`);  
  }  
})();
```

Si la BBDD se ha creado, se informa por consola.

Si la BBDD ya existe, se informa por consola.

Captura rechazos de promesas, excepciones o errores.

```
user_quiz — node main.js — 28x5  
$  
$ node main.js  
> DB created with 3 elems  
>  
>
```

```
user_quiz — node main.js — 28x5  
$  
$ node main.js  
> DB exists & has 3 elems  
>  
>
```

# cmds\_user.js: comandos help

Importa la tabla **User** del modelo.

```
const User = require("../model.js").models.User

exports.help = (rl) =>
  rl.log(
    `
    Commands (params are requested after):
    > h                ## show help
    >
    > lu | ul | u      ## users: list all
    > cu | uc          ## user: create
    > ru | ur | r      ## user: read (show age)
    > uu               ## user: update
    > du | ud          ## user: delete
    >
    > e                ## exit & return to shell`
  )
```

El comando **help** envía al stream de salida del interfaz **rl** un string con la ayuda. Este describe la lista de comandos y su función.

La función **help** no necesita ser **async** porque no tiene callbacks, ni sincronizaciones.

```
user@user:~$ node -i npm run dump_environment_flag=0 npm bin_path=/usr/bin/rlc/bin -- 48x17
>
> h
Commands (params are requested after):
  > h                ## show help
  >
  > lu | ul | u      ## users: list all
  > cu | uc          ## user: create
  > ru | ur | r      ## user: read (show age)
  > uu               ## user: update
  > du | ud          ## user: delete
  >
  > e                ## exit & return to shell
>
> ul
Peter is 22 years old
Ed is 17 years old
>
```

# cmds\_user.js: comando create

El comando **create** pide el nombre y la edad del usuario a crear con **rl.questionP("Enter ..")**, espera con **await** a que sean tecleados, los guarda en las variables **name** y **age**, y comprueba que no están vacíos. Lanza una excepción si alguno está vacío.

```
// Create user with age in the DB
```

```
exports.create = async (rl) => {
```

```
  let name = await rl.questionP("Enter name");  
  if (!name) throw new Error("Response can't be empty!");
```

```
  let age = await rl.questionP("Enter age");  
  if (!age) throw new Error("Response can't be empty!");
```

```
  await User.create(  
    { name, age }  
  );  
  rl.log(`    ${name} created with ${age} years`);  
}
```

Lanza una petición de creación de un nuevo registro en la tabla (**user.create({name, age})**) y si finaliza con éxito, envía un msj a consola indicándolo.

Si no se pudiese crear el usuario, sequelize rechazaría la promesa (**user.create(...)**). El rechazo sería capturado por catch, en el programa principal (main.js).

```
>  
> CU  
Enter name: Ed  
Enter age: 11  
Ed created with 11 years  
>
```

```
>  
> UC  
Enter name:  
Error: Response can't be empty!  
>
```

```
>  
> UC  
Enter name: Anna  
Enter age: 44  
SequelizeUniqueConstraintError:  
Name already exists  
>
```

# cmds\_user.js: comandos list

**list** lista el contenido de la tabla **User** por el stream de salida del interfaz **rl**.

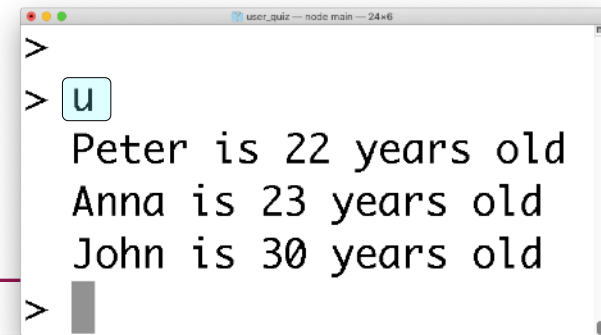
La función **list** necesita ser **async** porque los accesos a la BBDD deben ser sincronizados, porque incluyen esperas en los accesos a la BBDD.

**User.findAll()** retorna un array con todos los registros de la tabla User. Cada registro es un objeto con una propiedad para cada campo.

```
// Show all users in DB
exports.list = async (rl) => {
  let users = await User.findAll();

  users.forEach( u => rl.log(`  ${u.name} is ${u.age} years old`));
}
```

El iterador `forEach` envía una línea (con nombre y edad) por cada elemento del array al stream de salida.



A terminal window titled 'user\_quiz -- node main -- 24x6' showing a command prompt. The user enters 'u' (highlighted in a light blue box). The output is: 'Peter is 22 years old', 'Anna is 23 years old', and 'John is 30 years old'.

# cmds\_user.js: comandos read

El comando **read** pide el nombre del usuario a mostrar con **rl.questionP("Enter name")** y espera con **await** a que el usuario lo teclee.

Guarda el string tecleado en la variable **name** y si es el string vacío envía una excepción indicándolo y finaliza el comando.

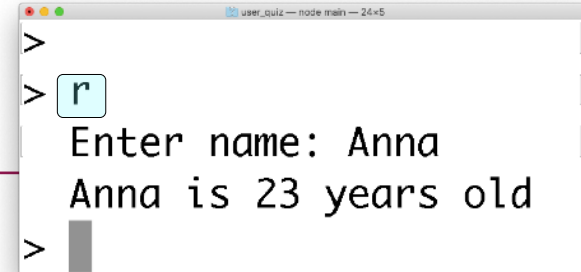
```
// Show user's age
exports.read = async (rl) => {
```

```
  let name = await rl.questionP("Enter name");
  if (!name) throw new Error("Response can't be empty!");
```

```
  let user = await User.findOne(
    { where: {name}, }
  );
  if (!user) throw new Error(` '${name}' is not in DB`);
```

```
  rl.log(` ${user.name} is ${user.age} years old`);
```

```
}
```



```
>
> r
Enter name: Anna
Anna is 23 years old
>
```

Petición de búsqueda por **nombre** a la BBDD (opción: **{ where:{name}}**). Recordar que **{name}** es equivalente en ES6 a **{name: name}**.

Si el nombre no está en la BBDD se devuelve **null**, y se lanza una excepción con un mensaje que lo indica.

Si se ha llegado a este punto es que el usuario está en la tabla **user** y se envía un mensaje indicando su edad.

# cmds\_user.js: comando update

```
// Update the user (identified by name) in the DB
```

```
exports.update = async (rl) => {
```

```
  let old_name = await rl.questionP("Enter name to update");  
  if (!old_name) throw new Error("Response can't be empty!");
```

```
  let name = await rl.questionP("Enter new name");  
  if (!name) throw new Error("Response can't be empty!");
```

```
  let age = await rl.questionP("Enter new age");  
  if (!age) throw new Error("Response can't be empty!");
```

```
  let n = await User.update(  
    {name, age},  
    {where: {name: old_name}}  
  );
```

```
  if (n[0]===0) throw new Error(` ${old_name} not in DB`);
```

```
  rl.log(` ${old_name} updated to ${name}, ${age}`);
```

```
}
```

El comando **update** pide el nombre del usuario a actualizar, así como los nuevos nombre y edad del usuario a actualizar con **rl.questionP("Enter ..")**, espera con **await** a que sean tecleados, los guarda en las variables **old\_name**, **name** y **age**, y comprueba que no están vacíos. Lanza una excepción si alguno está vacío.

```
user_quiz -- node main -- 28x7  
>  
> uu  
Enter name to update: Ed  
Enter new name: Eddy  
Enter new age: 11  
Ed updated to Eddy, 11  
>
```

**User.update({name, age}, where: {name: old\_name})**

Lanza una petición de actualización del registro en la tabla

Como la cláusula **where: {name: old\_name}** solicita la actualización de todos los registros cuyo campo **name** coincida con **old\_name**, el parámetro **n** indica el número de registros actualizados, por lo que si es cero, se lanza una excepción indicando que ese nombre no está en la BBDD.

Si no se lanza la excepción es porque el usuario indicado ha sido actualizado y el programa lo notifica por la consola.

Si no se pudiese crear el usuario, sequelize rechazaría la promesa (**user.create(...)**). El rechazo sería capturado por **catch**, en el programa principal (**main.js**).



# cmds\_user.js: comando delete

El comando **delete** pide el nombre del usuario a borrar con **rl.questionP("Enter name")** y espera con **await** a que el usuario lo teclee.

Guarda el string tecleado en la variable **name** y si vacío lanza una excepción y finaliza el comando.

```
// Delete user (identified by name) in the DB
```

```
exports.delete = async (rl) => {
```

```
  let name = await rl.questionP("Enter name");  
  if (!name) throw new Error("Response can't be empty!");
```

```
  let n = await User.destroy(  
    { where: {name}}  
  );
```

```
  if (n===0) throw new Error(`User ${name} not in DB`);
```

```
  rl.log(`  ${name} deleted from DB`);
```

```
}
```

Petición de borrar las entradas con el **nombre** dado (opción: **{ where:{name}}**). Este comando puede borrar varias entradas y retorna cuantas ha borrado.

(n===0) indica que no lo ha borrado y lanza una excepción con un mensaje indicándolo.

Si no se pudiese crear el usuario, sequelize rechazaría la promesa (**user.create(...)**). El rechazo sería capturado por catch, en el programa principal (main.js).

Si se ha llegado a este punto es que el usuario ha sido borrado tabla user y se envía un mensaje indicando.

```
>  
> du  
Enter name: John  
John deleted from DB  
>
```



# El Programa Principal: main.js

Importar módulos:

- **readline**: para crear interfaz.
- **cmds\_user**: comandos del programa.

Define el interfaz **rl** para acceso a los streams de entrada/salida en modo línea a línea. (ver doc node: <https://node.readthedocs.io/en/latest/api/readline/>)

Añadir el método **log** al interfaz **rl**. Este método utiliza **console.log(..)**, algo que deberá cambiarse para conectar a través de sockets.

Se añade **questionP(..)** al interfaz. Es similar al método **question** (basado en callbacks) del interfaz, pero devuelve lo tecleado con una **promesa**.

Manejador de **'line'**: invocado al teclear "retorno" (línea en parámetro line). El manejador se define como una función de tipo **async**, para que su código pueda contener expresiones **await** que sincronizan promesas.

Muestra el **prompt** ("> ") por el stream de salida.

El método **includes(<val>)** determina si un valor <val> está contenido en un array.

Si no se teclea nada, no se hace nada.

Al teclear alguno de los comandos se invoca el método asociado, que se ha importado del **módulo cmds\_user** que implementa el comando.

**exit** finaliza el programa y retorna a la shell de UNIX invocando: **process.exit(0)**

Si no coincide con ningún comando, se indica: **'Unsupported comand!'** y se muestra help().

Cualquier excepciones en cualquier comando se captura aquí.

```
> u
Peter is 22 years old
Ana is 23 years old
Eva is 11 years old
>
> r
Enter name: Ana
Ana is 23 years old
>
> cu
Enter name: Ed
Enter age: 17
Ed created with 17 year
> s
>
> lu
Peter is 22 years old
Ana is 23 years old
Eva is 11 years old
Ed is 17 years old
>
```

Muestra el **prompt** ("> ") después de responder.



JavaScript

# Final del tema

## Muchas gracias!