



JavaScript

Bases de Datos y ORM sequelize

Juan Quemada, DIT - UPM

Santiago Pavón, DIT - UPM

Bases de Datos y ORM sequelize - Índice

1.	<u>BBDD - Bases de Datos (JavaScript): Sequelize y Sqlite</u>	3
2.	<u>SGBDRs y ORMs para JavaScript: Sequelize y Sqlite</u>	8
3.	<u>Proyecto person: El proyecto y el paquete npm</u>	16
4.	<u>Proyecto person: El modelo de datos y la carga inicial</u>	23
5.	<u>Proyecto person: Interfaz CRUD y comandos</u>	32
6.	<u>Modelo, acceso y gestión de instancias</u>	38



BBDD - Bases de Datos (JavaScript): Sequelize y Sqlite

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
- Es también un almacén persistente y escalable de datos de gestión rápida, eficaz, fiable, ..

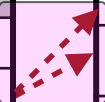
◆ **SGBD** - Sistema Gestor de BBDD (**DBMS** - Data Base Manag. Syst.)

- 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, ...
-

Base de Datos relacional



id	username	password	salt	isAdmin
1	admin	r34et5690y	"aaaa"	TRUE
2	pepe	56gh90op5	"bbbb"	FALSE
3

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

◆ Base de dato relacional y SGBDR (Sistema Gestor de BBDD 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/Entity_relationship_model

◆ SQL - Structured Query Language

- Lenguaje de definición, manipulación y control de datos en BBDDs relacionales
 - ♦ Permiten acceder a través de una API normalizada a cualquier BBDD relacional
 - <https://en.wikipedia.org/wiki/SQL>

◆ SGBDR mas habituales

- **MySQL, MariaDB, Microsoft SQL Server, Oracle, Firebird,**
 - ♦ Mas información en https://en.wikipedia.org/wiki/Relational_database_management_system

◆ Las BBDD relacionales han sido muy utilizadas

- Pero las **BBDDs NoSQL** están en auge en Internet: Google, Facebook, Amazon, ..

SGBDR: Modelo, clave primaria y clave externa

◆ Modelo relacional de datos

- Los datos se estructuran en tuplas de información relacionadas entre si
 - ◆ Cada tupla se suele implementar con una **tabla** (denominado modelo)

◆ Tabla (o modelo)

- Contiene una colección de registros de datos (tupla) con la misma estructura

◆ Clave primaria

- Clave que identifica unívocamente cada registro en la tabla
 - ◆ Una clave primaria no puede estar repetida en una tabla

◆ Clave externa y relación

- Las relaciones se 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

SGBDRs y ORMs para JavaScript:

Sequelize y Sqlite

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

ORM (Object Relational Mapping) Sequelize

◆ JavaScript es un lenguaje OO (Orientado a Objetos)

- SQL es un lenguaje orientado a comandos
 - ◆ Los lenguajes OO como JavaScript utilizan un ORM para acceder a una BBDD relacional
 - https://en.wikipedia.org/wiki/Object-relational_mapping
 - https://es.wikipedia.org/wiki/Mapeo_objeto-relacional

◆ ORM - Object Relational Mapping

- Adapta ordenes y tipos de un **lenguaje OO** a **comandos SQL**
 - ◆ Un ORM **simplifica** la utilización de un SGBDR desde **lenguajes OO**, por ejemplo JavaScript
 - Permite **acceder** a cualquier BBDD que soporte **SQL**: Oracle, Postgres, sqlite, MySQL,
- Existen diversos ORMs para node.js: **sequelize**, node-orm2, Bookshelf...
 - ◆ <https://www.codediesel.com/javascript/nodejs-mysql-orms/>

◆ Sequelize

- **ORM** muy popular de node.js basado en **promesas**
 - ◆ Instrucciones de instalación y uso: <http://sequelizejs.com/>

SGDBR y ORM JavaScript: instalar con npm

◆ En JS se deben instalar dos paquetes npm para acceder a una BBDD

- La **BBDD**: se utilizara **SQLite3** o **Postgres**
- El **ORM**: los ejemplos utilizan **sequelize** para acceso a BBDD desde JavaScript
 - ◆ <http://sequelizejs.com/>

◆ Los ejemplos utilizarán dos **SGBDRs** accesibles con **SQL**

- **SQLite v3**
 - ◆ SGBDR de **uso y configuración sencillos** que crea BBDDs relacionales
 - **SQLite v3** se utiliza en la fase de **desarrollo**: <https://www.npmjs.com/package/sqlite3>
- **Postgres (o PostgreSQL)**
 - ◆ SGBDR **completo y potente**, que permite acceso relacional, orientadas a objetos o clave-valor
 - Postgres se utiliza en **despliegue** en Internet: <https://www.postgresql.org/>

```
// instalar sequelize y sqlite3 en directorio node_modules con npm  
  
..$ npm install sequelize@3.30.2 // instala version 3.30.2 de sequelize*  
  
..$ npm install -g sqlite3@3.1.8 // instala version 3.1.8 de sqlite3*
```

*Nota. Estas versiones son las que se utilizaron para este proyecto.

Los paquetes se distribuyen con npm. El comando npm los instala en el subdirectorio **node_modules** del directorio de trabajo.

La opción **-g** instala el paquete en global y no en **node_modules**. sqlite3 debe instalarse con esta versión para que el programa de test funcione.

Objetos de acceso a BBDD y Tablas

◆ Sequelize genera objetos de acceso a BBDDs y tablas

- Estos objetos utilizan convenios de nombres preestablecidos que deben seguirse

◆ El objeto de acceso a una BBDD **sqlite** se define con

- `const sequelize = new Sequelize("sqlite:db.sqlite", options)`
 - ◆ **"sqlite:db.sqlite"**: es un URL que identifica la BBDD (Postgres utilizará otro URL)
 - **sqlite**: indica que es una BBDD de tipo sqlite
 - **db.sqlite**: es la ruta al **fichero** con la BBDD real

tabla people

name	age
Peter	22
Anna	23
John	30

◆ El **modelo** de la tabla **people** se define con el objeto

- `const person = sequelize.define('person', <object>)`
 - ◆ Los **campos de la tabla** tendrán el nombre y el tipo definido en el objeto **<object>**
- **Convenio** utilizado: **modelos** en **singular** (person) y **tablas** en **plural** (people)

◆ **sequelize.sync()**

- Configura las tablas definidas en el fichero **db.sqlite** (si no lo están ya)
 - ◆ Además crea el fichero **db.sqlite** (antes de configurar), si no existiese
 - Las tablas de una BBDD se pueden sincronizar también con **migraciones** (más versátil que **sync()**)

name	age
Peter	22
Anna	23
John	30

Acceso a las instancias de una tabla

- ◆ Los **objetos de instancia** representan los **filas** o **registros** de una **tabla**
 - Las **propiedades** de estos objetos **tienen el mismo nombre** que los **campos** de la tabla
- ◆ Los métodos de sequelize devuelven **promesas**
 - Así pueden sincronizarse temporalmente, aunque utilicen "callbacks" internamente
- ◆ **person.count()**
 - Promesa que devuelve el número de **instancias** (registros o filas) de una tabla
- ◆ **person.bulkCreate(<instance_array>)**
 - Promesa que añade a la tabla '**people**' un array de instancias
 - ◆ Asigna a los campos de la tabla los valores de las propiedades del mismo nombre de las instancias
 - La promesa retorna cuando la operación se ha completado (registros han sido añadidos a la tabla)
- ◆ **person.findAll()**
 - Promesa que devuelve un array con todas las instancias de la tabla '**people**'

Ejemplo 70-init.js: modelo e inicialización de la BBDD

```
const Sequelize = require('sequelize');
```

Importar sequelize.js

```
const options = { logging: false, operatorsAliases: false};
```

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

Estas opciones eliminan trazas por consola. Si se quitan aparecerán.

```
const person = sequelize.define('person', { name: Sequelize.STRING, age: Sequelize.INTEGER });
```

Define la **tabla people** con 2 campos de usuario: **name** (de tipo string) y **age** (de tipo integer).

Define que SGBDR utilizar (**sqlite**) y el fichero donde sqlite guardará los datos (**mi_db.sqlite**).

Borra el fichero **db.sqlite** con los datos.

Primera invocación: **sync()** crea un fichero **db.sqlite** nuevo y configura la tabla **people**.

```
sequelize.sync()
```

sync() sincroniza la BBDD en **mi_db.sqlite** con la estructura de tablas definida (**people**).

```
.then(() => person.count())
```

```
.then((count) => {
```

```
  if (count===0) {
```

```
    return (
```

```
      person.bulkCreate([
```

```
        { name: 'Peter', age: 22},
```

```
        { name: 'Anna', age: 23},
```

```
        { name: 'John', age: 30}
```

```
      ])
```

```
      .then( c => console.log(` DB created with ${c.length} elems`))
```

```
    )
```

```
  } else {
```

```
    return console.log(` DB exists & has ${count} elems`);
```

```
  }
```

```
})
```

```
.catch( err => console.log(` ${err}`));
```

Muestra errores o excepciones

Cadena de promesas: sincronizan las operaciones y devuelven el resultado al siguiente.

tabla people

name	age
Peter	22
Anna	23
John	30

count() devuelve el número de registros (filas) de la tabla.

bulkCreate(..) crea un registro (fila) en la **tabla people** por cada objeto del array del parámetro, creando los usuarios Peter, Anna y John.

Mensajes que indican si la BBDD se inicia o no.

Siguientes invocaciones: la BBDD esta ya creada en el fichero **db.sqlite** y solo se indica que existe.

```
$
$ rm db.sqlite
$
$ node 70-init.js
DB created with 3 elems
$
$ node 70-init.js
DB exists & has 3 elems
$
$ node 70-init.js
DB exists & has 3 elems
$
```

Ejemplo 71-all.js: modelo y listado de tabla

Definición del modelo de datos (BBDD y tabla people). Es similar al anterior para que acceda a la misma BBDD.

El comando **76-all.js** muestra el nombre y edad de todos los registrados en la tabla.

```
const Sequelize = require('sequelize');

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

const person = sequelize.define(
  'person',
  {
    name: Sequelize.STRING,
    age: Sequelize.INTEGER
  }
);
```

```
person.findAll()
  .then( people =>
    people.forEach(p => console.log(` ${p.name} is ${p.age} years old`))
  )
  .catch( err => console.log(err));
```

findAll() devuelve un array con todas las entradas de la tabla.

forEach() muestra los valores de **name** y **age** de todos los objetos del array **people**: [obj1, obj2, obj3].

```
$
$ node 71-all.js
Peter is 22 years old
Anna is 23 years old
John is 30 years old
$
```

tabla people

	name	age
obj1	Peter	22
obj2	Anna	23
obj3	John	30

Ejemplo 72-obj.js: tabla completa

Definición del modelo de datos (BBDD y tabla people). Es similar al anterior para que acceda a la misma BBDD.

**tabla
people
(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

El comando **72-obj.js** muestra los todos los campos de la tabla **people**, los dos definidos por el usuario (**name** y **age**) y los tres creados por el SGBDR (**id**, **createdAt** y **updatedAt**):

- **id**: clave pública
- **name**:
- **age**:
- **createdAt**: Fecha/hora de creación
- **updatedAt**: Fecha/hora de última modificación

```
const Sequelize = require('sequelize');

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

```
const person = sequelize.define(
  'person',
  {
    name: Sequelize.STRING,
    age: Sequelize.INTEGER
  }
);
```

```
person.findAll()
  .then( people =>
    people.forEach(p => console.log( p.get({ plain: true })))
  )
  .catch( err => console.log(err));
```

p.get({ plain: true }) muestra las propiedades y valores del objeto (en formato JSON) que corresponden con los campos de la tabla.

```
ej-BBDD - bash - 40x18
$
$ node 72-obj.js
{ id: 1,
  name: 'Peter',
  age: 22,
  createdAt: 2018-01-27T12:44:43.996Z,
  updatedAt: 2018-01-27T12:44:43.996Z }
{ id: 2,
  name: 'Anna',
  age: 23,
  createdAt: 2018-01-27T12:44:43.996Z,
  updatedAt: 2018-01-27T12:44:43.996Z }
{ id: 3,
  name: 'John',
  age: 30,
  createdAt: 2018-01-27T12:44:43.996Z,
  updatedAt: 2018-01-27T12:44:43.996Z }
$
```



Proyecto person:

El proyecto y el paquete npm

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

El proyecto educativo person

tabla people

name	age
Peter	22
Anna	23
John	30

- ◆ El proyecto educativo **person** tiene como objetivo
 - Ilustrar una aplicación con una BBDD con 1 tabla
- ◆ La aplicación son varios **comandos node.js**
 - Todos acceden a la misma BBDD en el fichero **db.sqlite**
 - ◆ El primer comando crea e inicializa la BBDD
 - comando **0_init**
 - ◆ El resto de comandos implementan la **interfaz CRUD**
 - Create: **2_create**
 - Read: **1_all** y **3_read**
 - Update: **4_update**
 - Delete: **5_delete**
- ◆ Es un proyecto **Git** que crea un paquete npm
 - La aplicación se construye en 5 commits
 - ◆ Accesible en: <https://github.com/CORE-UPM/person>
 - Tiene estructura de paquete npm
 - ◆ Para automatizar la instalación de dependencias con: **npm install**

```
$  
$ node 0_init.js  
DB created with 3 elems  
$  
$ node 0_init.js  
DB exists & has 3 elems  
$  
$  
$ node 1_all.js  
Peter is 22 years old  
Anna is 23 years old  
John is 30 years old  
$  
$ node 3_read.js Anna  
Anna is 23 years old  
$
```



```
$  
$ node 1_all.js  
Peter is 22 years old  
Anna is 23 years old  
John is 30 years old  
$  
$ node 2_create.js Eva 66  
Eva created with 66 years  
$  
$ node 4_update.js John 31  
John updated to 31  
$  
$ node 5_delete.js Peter  
Peter deleted from DB  
$  
$ node 1_all.js  
Anna is 23 years old  
John is 31 years old  
Eva is 66 years old  
$
```

Commits del proyecto person

◆ El proyecto person crea paquete **npm** en 5 commits **Git**

- GitHub: <https://github.com/CORE-UPM/person>

tabla people

name	age
Peter	22
Anna	23
John	30



```
$  
$ git clone -q https://github.com/CORE-UPM/person  
$ cd person  
$  
$ git log --oneline  
3282c00 Person v3: validation msgs  
5197308 Person v2: model validation  
374fa9d Person v1: CRUD interface  
6bb28be Add package.json & more  
ad73787 Initial commit  
$
```

Person v3: Añade **mensajes** asociados a la violación de la validación del modelo en el fichero model.js.

Person v2: Añade **validaciones** de los contenidos que se pueden introducir en la BBDD. Se añaden a las definiciones del modelo en el fichero model.js.

Person v1: Incluye los **6 comandos** (0_init, 1_all, 2_create, 3_read, 4_update, 5_delete) y el **modelo** de datos en el fichero model.js.

Commit inicial añadido al repositorio al crearlo en GitHub en (<https://github.com/CORE-UPM/person>) con los ficheros:

- **LICENSE:** con licencia MIT
- **README.md:** con una descripción del proyecto person
- **.gitignore:** con los ficheros ignorados típicamente en node.js.

Este commit añade el fichero **package.json** del paquete **npm** con:

\$ npm init

npm init solicita los parámetros a través de la consola al invocarlo.

Además se añaden las dependencias a los paquetes **sqlite3** y **sequelize** con los comandos:

\$ npm install --save sequelize@3.30.2

\$ npm install --save sqlite3@3.1.8

También añade mas ficheros a ignorar en **.gitignore** y mejora la descripción de **README.md**.



Commit 2: add packages.json & more



Commit 2: add packages.json & more

```
{
  "name": "person",
  "version": "1.0.0",
  "description": "Simple DB example program illustrating the CRUD interface",
  "main": "0_init.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/CORE-UPM/person.git"
  },
  "keywords": [
    "DB",
    "sqlite3",
    "sequelize"
  ],
  "author": "Juan Quemada",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/CORE-UPM/person/issues"
  },
  "homepage": "https://github.com/CORE-UPM/person#readme",
  "dependencies": {
    "sequelize": "^3.30.2",
    "sqlite3": "^3.1.8"
  }
}
```

◆ Este commit añade

- Nuevos ficheros a .gitnignore
- Una descripción mejor en README.md
- package.json
 - ◆ Se crea con: npm init
 - ◆ Se añaden dependencias con: npm install ...

El fichero **package.json** de un paquete **npm** se crea con el comando:

\$ npm init

Este comando pide los parámetros a través de la consola al invocarlo.

Las dependencias a los paquetes **sqlite3** y **sequelize** se añaden a package.json automáticamente al instalarlos con la opción --save:

\$ npm install --save sequelize@3.30.2
\$ npm install --save sqlite3@3.1.8



Commits 3, 4 y 5: Interfaz CRUD y validación

La aplicación: commits 3, 4 y 5

◆ Commit 3

- Define la BBDD con 1 tabla
 - ♦ Comando 0_init.js: crea la **BBDD** con una tabla en el **fichero db.sqlite**
- Incluye el interfaz CRUD con 5 comandos independientes
 - ♦ Create (2_create), Read (1_all, 3_read), Update (4_update), Delete (5_delete)

◆ Commit 4

- Introduce restricciones a la tabla con atributos y validaciones
 - ♦ **name** debe ser **único** y con **caracteres alfabéticos ASCII**
 - ♦ **age** debe ser mayor que **0** y menor que **150**

◆ Commit 5

- Añade mensajes a los atributos y validaciones

```
$ git clone https://github.com/CORE-UPM/person
$ cd person
```

```
$ npm install
```

npm install instala todas las dependencias y deja la aplicación lista para ejecutarse.

```
$ node 0_init          ## creates and initializes the DB
$ node 1_all           ## shows the age of all entries
$ node 2_create <name> <age>  ## new table entry
$ node 3_read <name>     ## lists the age of name
$ node 4_update <name> <n_age> ## updates to n_age
$ node 5_delete <name>    ## removes name from DB
```

tabla people

name	age
Peter	22
Anna	23
John	30

```
$
$ node 0_init.js
DB created with 3 elems
$
$ node 0_init.js
DB exists & has 3 elems
$
$
$ node 1_all.js
Peter is 22 years old
Anna is 23 years old
John is 30 years old
$
$ node 3_read.js Anna
Anna is 23 years old
$
```



```
$
$ node 1_all.js
Peter is 22 years old
Anna is 23 years old
John is 30 years old
$
$ node 2_create.js Eva 66
Eva created with 66 years
$
$ node 4_update.js John 31
John updated to 31
$
$ node 5_delete.js Peter
Peter deleted from DB
$
$ node 1_all.js
Anna is 23 years old
John is 31 years old
Eva is 66 years old
$
```



Proyecto person:

El modelo de datos y la carga inicial

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

El modelo sin validación (commit 3: Person_v1)

◆ Modelo

- Define la estructura de los datos de la BBDD y la tabla people
 - ◆ En las BBDDs relacionarles se refiere a la estructura de las tablas y relaciones entre ellas
 - <http://docs.sequelizejs.com/manual/tutorial/models-definition.html>
- El modelo permite acceder a los elementos de la BBDD
 - ◆ Se define en el fichero **model.js**, que importan todos los comandos del programa

model.js

```
const Sequelize = require('sequelize');
```

Importar el **ORM** sequelize.js (paquetes sqlite3 y sequelize deben instalarse antes con npm).

```
const options = { logging: false, operatorsAliases: false};
```

Estas opciones eliminan trazas por consola. Si se quitan aparecerán.

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

Define con el **URL "sqlite:db.sqlite"**, que SGBDR utilizar (**sqlite**) y la ruta al fichero donde sqlite guardará los datos (**db.sqlite**).

```
sequelize.define('person',  
  { name: Sequelize.STRING,  
    age: Sequelize.INTEGER  
  }  
);
```

Define una **tabla**:

- El parámetro 1 define nombre del modelo:
-> **person**
- El parámetro 2 define los campos de usuario de la tabla:
 - 1) **name** de tipo string
 - 2) **age** de tipo integer

tabla people

name	age
Peter	22
Anna	23
John	30

```
module.exports = sequelize;
```

Exporta el objeto de acceso a la **BBDD** y a sus tablas (modelos)

Algunos tipos de datos de las tablas

El tipo `STRING` permite guardar strings de hasta 255 caracteres.

tabla people

name	age
Peter	22
Anna	23
John	30

```
Sequelize.STRING // VARCHAR(255)
Sequelize.STRING(1234) // VARCHAR(1234)
Sequelize.STRING.BINARY // VARCHAR BINARY
Sequelize.TEXT // TEXT

Sequelize.INTEGER // INTEGER
Sequelize.BIGINT // BIGINT
Sequelize.BIGINT(11) // BIGINT(11)
Sequelize.FLOAT // FLOAT
Sequelize.FLOAT(11) // FLOAT(11)
Sequelize.FLOAT(11, 12) // FLOAT(11,12)

Sequelize.DECIMAL // DECIMAL
Sequelize.DECIMAL(10, 2) // DECIMAL(10,2)

Sequelize.DATE // DATETIME for mysql / sqlite, TIMESTAMP WITH TIME ZONE for postgres
Sequelize.BOOLEAN // TINYINT(1)

Sequelize.ENUM('value 1', 'value 2') // An ENUM with allowed values 'value 1', 'value 2'
Sequelize.ARRAY(Sequelize.TEXT) // Defines an array. PostgreSQL

Sequelize.BLOB // BLOB (bytea for PostgreSQL)
Sequelize.BLOB('tiny') // TINYBLOB (bytea for PostgreSQL)
// are medium and long

Sequelize.UUID // UUID datatype for PostgreSQL and SQLite, CHAR(36) BINARY for MySQL (use defaultValue: Sequelize.UUIDV1 or Sequelize.UUIDV4 to make sequelize generate the ids automatically)
```

```
const Person = sequelize.define(
  'people',
  {
    name: Sequelize.STRING,
    age: Sequelize.INTEGER
  }
);
```

Sequelize permite definir muchos tipos de campos en las tablas de la BBDD. Algunos son específicos de BBDD determinadas. Mas info:

<http://docs.sequelizejs.com/manual/tutorial/models-definition.html#data-types>

0_init: Sincronizar/iniciar la BBDD

◆ 0_init: crea la BBDD inicial

- añade 3 instancias (filas) a la BBDD vacía
 - ◆ Si no esta vacía informa del número de registros
 - <http://docs.sequelizejs.com/manual/tutorial/associations.html#creating-with-associations>

```
$  
$ node 0_init.js  
DB created with 3 elems  
$  
$ node 0_init.js  
DB exists & has 3 elems  
$
```

0_init.js

```
const sequelize = require("../model.js");  
const person = sequelize.models.person;
```

Importar la **BBDD (sequelize)** y extraer el **modelo person** de sus modelos.

```
sequelize.sync()  
.then(() => person.count())
```

El método **sync()** sincroniza la BBDD (archivo **bd.sqlite**) con el modelo (**model.js**). Crea la tabla **people**, cuando no existe.

```
.then((count) => {  
  if (count===0) {
```

Obtiene el **número** de elementos de la **tabla people**.

```
    return (  
      person.bulkCreate([  
        { name: 'Peter', age: 22},  
        { name: 'Anna', age: 23},  
        { name: 'John', age: 30}  
      ])
```

Si la tabla **people** esta **vacía (count === 0)**, se inicia con los 3 elementos.

```
    ).then( c => console.log(` DB created with ${c.length} elems`))
```

tabla people

name	age
Peter	22
Anna	23
John	30

Informar por consola que se han creado los 3 elementos iniciales.

```
  } else {  
    console.log(` DB exists & has ${count} elems`);
```

Informar por consola que la BBDD ya tiene elementos.

```
  }  
})  
.catch( err => console.log(` ${err}`));
```

Informar de promesas incumplidas.



Commits 4 y 5 (person v2 y v3): Modelo con validación y mensajes

Modelo y tabla

tabla people (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

```
{ name: {
  type: Sequelize.STRING,
  unique: true,
  validate: { is: /^[a-z]+$/i }
},
age: {
  type: Sequelize.INTEGER,
  validate: { min: 0, max: 150 }
}
}
```

◆ Una tabla tiene 3 columnas creadas por el SGBDR

- **id** o índice primario: identificador único de instancia
- **createdAt**: fecha de creación de la instancia
- **updatedAt**: fecha de última actualización

◆ El **modelo** define **columnas adicionales** a la tabla con **propiedades**

- Cada **propiedad** tiene asociado un tipo de **datos asociado**: STRING, NUMBER, ...
 - ♦ Las propiedades pueden tener además **opciones**

◆ Las **opciones** establecen requisitos sobre los campos de una columna

- Por ejemplo: **unique**, **allowNull**, **defaultValue**, **validate**, ...
 - ♦ <http://docs.sequelizejs.com/manual/tutorial/models-definition.html>

◆ Las **validaciones** son opciones especiales definidas con la propiedad **validate**

- Las validaciones permiten controlar el contenido de los campos con mayor precisión, e incluso pueden programarse controles a medida
 - ♦ <http://docs.sequelizejs.com/manual/tutorial/models-definition.html#validations>

Person_v2: validación del modelo

tabla people

name	age
Peter	22
Anna	23
John	30

```
const Sequelize = require('sequelize');

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

model.js

```
sequelize.define(
```

```
  'person',
```

```
  { name: {
```

```
    type: Sequelize.STRING,
```

```
    unique: true,
```

```
    validate: { is: /^[a-z]+$/i }
```

```
  },
```

```
  age: {
```

```
    type: Sequelize.INTEGER,
```

```
    validate: { min: 0, max: 150 }
```

```
  }
```

```
};
```

```
);

module.exports = sequelize;
```

El tipo de contenido asociado al campo se incluye en la propiedad type.

La propiedad **unique: true** añade una opción, que establece un requisito sobre la columna: dos campos no pueden tener el mismo contenido.

La propiedad **validate: {is: /^[a-z]+\$/i}** añade una validación: el contenido debe casar con la expresión regular y solo debe contener letras ASCII.

La propiedad **validate: {min: 0, max: 150}** indica que el valor numérico debe estar entre 0 y 150.

◆ Person v2 añade la validación

■ Comprueba los contenidos al introducirlos en los campos

- ◆ Si la condición definida no se cumple lanza un **Error**

- <http://docs.sequelizejs.com/manual/tutorial/models-definition.html>

- <http://docs.sequelizejs.com/manual/tutorial/models-definition.html#validations>

```
var ValidateMe = sequelize.define('foo', {
  foo: {
    type: Sequelize.STRING,
    validate: {
```

Sequelize incluye muchas funciones de validación que facilitan el rechazo de entradas no validas a la base de datos. Mas información:
<http://docs.sequelizejs.com/manual/tutorial/models-definition.html#validations>

Sequelize: validaciones

is: ["^[a-z]+\$", 'i'],	// will only allow letters
is: /^[a-z]+\$/i,	// same as the previous example using real RegExp
not: ["[a-z]", 'i'],	// will not allow letters
isEmail: true,	// checks for email format (foo@bar.com)
isUrl: true,	// checks for url format (http://foo.com)
isIP: true,	// checks for IPv4 (129.89.23.1) or IPv6 format
isIPv4: true,	// checks for IPv4 (129.89.23.1)
isIPv6: true,	// checks for IPv6 format
isAlpha: true,	// will only allow letters
isAlphanumeric: true	// will only allow alphanumeric characters, so "_abc" will fail
isNumeric: true	// will only allow numbers
isInt: true,	// checks for valid integers
isFloat: true,	// checks for valid floating point numbers
isDecimal: true,	// checks for any numbers
isLowercase: true,	// checks for lowercase
isUppercase: true,	// checks for uppercase
notNull: true,	// won't allow null
isNull: true,	// only allows null
notEmpty: true,	// don't allow empty strings
equals: 'specific value',	// only allow a specific value
contains: 'foo',	// force specific substrings
notIn: [['foo', 'bar']],	// check the value is not one of these
isIn: [['foo', 'bar']],	// check the value is one of these
notContains: 'bar',	// don't allow specific substrings
len: [2,10],	// only allow values with length between
isUUID: 4,	// only allow uuids
isDate: true,	// only allow date strings
isAfter: "2011-11-05",	// only allow date strings after a specific date
isBefore: "2011-11-05",	// only allow date strings before a specific date
max: 23,	// only allow values less than or equal to 23
min: 23,	// only allow values greater than or equal to 23
isArray: true,	// only allow arrays
isCreditCard: true,	// check for valid credit card number

```
sequelize.define(
  'Person',
  { name: {
    type: Sequelize.STRING,
    unique: true,
    validate: { is: /^[a-z]+$/i }
  },
    age: {
      type: Sequelize.INTEGER,
      validate: { min: 0, max: 150 }
    }
  }
);
```

Person_v3: mensajes de validación

tabla people

name	age
Peter	22
Anna	23
John	30

```
const Sequelize = require('sequelize');

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

model.js

```
sequelize.define(
  'person',
  {
    name: {
      type: Sequelize.STRING,
      unique: { msg: "Name already exists" },
      validate: {
        is: { args: /^[a-z]+$/i, msg: "name: invalid characters" }
      }
    },
    age: {
      type: Sequelize.INTEGER,
      validate: {
        min: { args: [0], msg: "Age less than 0" },
        max: { args: [150], msg: "Age higher than 150" }
      }
    }
  }
);
```

El mensaje se añade con la propiedad **msg**.

Si hay propiedad **msg**, el argumento debe ir en la propiedad **args**.

Si hay propiedad **msg**, el argumento debe ir en la propiedad **args** y además debe ser en este caso el primer elemento de un array.

◆ Person v3 añade los mensajes de validación

■ Los Errores lanzados incluirán los nuevos mensajes

- ◆ <http://docs.sequelizejs.com/manual/tutorial/models-definition.html>
- ◆ <http://docs.sequelizejs.com/manual/tutorial/models-definition.html#validations>

```
module.exports = sequelize;
```



Proyecto person: Interfaz CRUD y comandos

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

El interfaz CRUD

◆ Una BBDD suele utilizar el interfaz CRUD con cuatro tipos de operaciones

- **Create** - crear uno o varios registros de datos
- **Read** - leer el contenido de uno o varios registros de datos
- **Update** - actualizar el contenido de uno o varios registros de datos
- **Delete** - borrar uno o varios registros de datos

◆ El ejemplo person usa 5 comandos para gestionar la BBDD

```
$ node 1_all                ## shows the age of all entries
$ node 2_create <name> <age>  ## new table entry
$ node 3_read <name>        ## lists the age of name
$ node 4_update <name> <n_age> ## updates to n_age
$ node 5_delete <name>      ## removes name from DB
```

2_create: añadir usuario

◆ 2_create: añade usuario

- Crea una nueva instancia en la tabla people

tabla people

name	age
Peter	22
Anna	23
John	30
Eve	66

```
$  
$  
$ node 2_create.js Eve 66  
Eve created with 66 years  
$
```

2_create.js

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

```
if (process.argv.length !== 4){ // Wrong parameters?  
  console.log('    syntax: "node 2_create <name> <age>"');  
  process.exit(2);              // Finalizes node process  
}
```

```
const name = process.argv[2], age = process.argv[3];
```

```
person.create({ name, age })
```

```
.then(() =>
```

```
  console.log(`    ${name} created with ${age} years`)  
)
```

```
.catch( err => console.log(`    ${err}`));
```

Importar el **modelo person** directamente.

Comprobar que el comando se invoca con el número correcto de parámetros.

Asignar nombres legibles a los parámetros.

Crea nuevo usuario en la **tabla people** asignando al campo **name** el parámetro **process.argv[2]** y al campo **age** el parámetro **process.argv[3]**.

{age, name} en ES6 equivale a **{age: age, name: name}** en ES5.

Informar que el nuevo usuario se ha añadido.

Informar de promesas incumplidas.

Edad de uno (3_read) o de todos (1_all) los usuarios

3_read.js

Importar el **modelo person** directamente.

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

```
if (process.argv.length !== 3){ // Wrong parameters?
  console.log('  syntax: "node 3_read <name>"');
  process.exit(2); // Finalizes node process
}
```

Comprobar número de parámetros.

```
const name = process.argv[2];
```

Homogeneizar nombre de parámetro.

```
person.findOne( {where: {name}})
  .then( person => {
    if (!person) {throw new Error(` '${name}' is not in DB`)};
    console.log(` ${person.name} is ${person.age} years old`);
  })
  .catch( err => console.log(` '${err}'`));
```

Mostrar edad del usuario.

Busca el usuario en la tabla **people** cuyo campo **name** sea igual al parámetro `process.argv[2]`, contenido en la variable **name**.

{name} en ES6 es **{name: name}** en ES5.

Se **rechaza la promesa**:
usuario no existe!

tabla people

name	age
Peter	22
Anna	23
John	30

1_all.js

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

```
person.findAll()
  .then( people =>
    people.forEach( p => console.log(` ${p.name} is ${p.age} years old`))
  )
  .catch( err => console.log(` '${err}'`));
```

findAll() devuelve un array con todas las entradas de la tabla.

forEach() muestra los valores de las propiedades **name** y **age** de todos los objetos del array **people**.

```
$
$ node 1_all.js
Peter is 22 years old
Anna is 23 years old
John is 30 years old
$
$ node 3_read.js Anna
Anna is 23 years old
$
```

4_update: actualizar usuario

◆ 4_update: actualiza usuario

- asigna nueva edad a user

tabla people

name	age
Peter	22
Anna	66
John	30

```
$  
$  
$ node 4_update.js Anna 66  
Anna updated to 66  
$
```

4_update.js

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

```
if (process.argv.length !== 4){ // Wrong parameters?  
  console.log('  syntax: "node 4_update <name> <age>"');  
  process.exit(2); // Finalizes node process  
}
```

```
const name = process.argv[2], age = process.argv[3];
```

```
person.update( {age}, {where: {name}})
```

```
.then( n => {  
  if (n[0]!==0) { console.log(`  ${name} updated to ${age}`) }  
  else { throw new Error(`  ${name} not in DB`); }  
})
```

```
.catch( err => console.log(`  ${err}`));
```

Importar el **modelo person** directamente.

Comprobar que el comando se invoca con el número correcto de parámetros.

Asignar nombres legibles a los parámetros.

Actualiza campo **age** con **process.argv[3]**, si campo **name** es igual a la variable **name** (que contiene **process.argv[2]**).

{age} en ES6 equivale a **{age: age}** en ES5 y **{name}** en ES6 equivale a **{name: name}** en ES5.

Informar que el usuario ha sido actualizado.

Se **rechaza la promesa**: usuario no existe!

Informar de promesas incumplidas.

5_delete: borrar usuario

◆ 5_delete: borra el usuario

- eliminándolo de la tabla people

5_delete.js

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

```
if (process.argv.length !== 3){ // Wrong parameters?  
  console.log('    syntax: "node 5_delete <name>"');  
  process.exit(2); // Finalizes node process  
}
```

```
const name = process.argv[2];
```

```
person.destroy( {where: {name} })
```

```
.then( n => {
```

```
  if (n !== 0) {
```

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

```
  }
```

```
  else {
```

```
    throw new Error(`    ${name} not in DB`)
```

```
  };
```

```
})
```

```
.catch( err => console.log(`    ${err}`));
```

tabla people

name	age
Peter	22
John	30

```
$  
$ node 5_delete.js Anna  
Anna deleted from DB  
$  
$ node 5_delete.js Anna  
Error:    Anna not in DB  
$
```

Importar el **modelo person** directamente.

Comprobar que el comando se invoca con el número correcto de parámetros.

Asignar nombre legible al parámetro `process.argv[2]`.

Borrar los registros de la tabla `people` cuyo campo **name** coincida con el parámetro **`process.argv[2]`**.

`{name}` en ES6 equivale a **`{name: name}`** en ES5.

Informar que el usuario ha sido eliminado.

Se **rechaza la promesa**: usuario no existe!

Informar de promesas incumplidas.



Modelo, acceso y gestión de instancias

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

Algunos métodos de búsqueda

tabla people		
id	name	age
1	Peter	22
2	Anna	66
3	John	30

obj1 → 1
obj2 → 2
obj3 → 3

◆ Ejemplos de métodos de búsqueda de sequelize

- **person.findAll()** → [obj1, obj2, obj3]
 - ◆ Devuelve un array con todos los objetos de la tabla people
- **person.findOne()** → obj1
 - ◆ Devuelve el primer objeto que encuentra en la tabla people
- **person.findById(2)** → obj2
 - ◆ Devuelve el objeto con índice 2 en la tabla people
- **person.find()** → obj1
 - ◆ Se mantiene por compatibilidad hacia atrás, emula **findOne(..)**, **findById(2)**, ..
- **person.count()** → 3
 - ◆ Devuelve el número de objetos en la tabla people
- Documentación
 - ◆ <http://docs.sequelizejs.com/manual/tutorial/models-usage.html>

Métodos de actualización persistente

◆ Ejemplos de métodos para actualización persistente

- **person.create(<obj>)** y **person.bulkcreate(<obj array>)**
 - ♦ Crea uno o varios registros nuevos en la tabla people
- **let p = person.build()** y **p.save()**
 - ♦ **build()** crea una instancia no persistente con la estructura de la tabla people y **save()** la guarda de forma persistente en la tabla (equivale a **person.create(<obj>)**)
- **person.findOrCreate(<options>)**
 - ♦ Actualiza varios objetos de la tabla people
- **person.update(<values>, <options>)**
 - ♦ Actualiza varios objetos de la tabla people
- **let p = person.findOne(..)** y **p.save(<options>)**
 - ♦ **findOne()** busca una instancia en la tabla people y **save()** la guarda de forma persistente en la tabla las modificaciones que se hayan introducido en la instancia
- **person.destroy(<options>)**
 - ♦ Destruye varios objetos de la tabla people




Diagram illustrating the mapping of objects to database records. Three objects, labeled obj1, obj2, and obj3, are shown on the left. Arrows point from each object to a corresponding row in a table titled 'tabla people'. obj1 points to the first row (id 1, name Peter, age 22), obj2 points to the second row (id 2, name Anna, age 66), and obj3 points to the third row (id 3, name John, age 30).

tabla people		
id	name	age
1	Peter	22
2	Anna	66
3	John	30

◆ Documentación

- <http://docs.sequelizejs.com/manual/tutorial/instances.html>

Opciones de búsqueda: <options>

	id	name	age	
obj1	1	Peter	22	tabla people
obj2	2	Anna	66	
obj3	3	John	30	

◆ Las opciones de búsqueda son aplicables a diversos métodos

- **where:** fijar condiciones en la búsqueda
 - ◆ `person.findAll({ where: { id: [1, 3]}})` -> [obj1, obj3]
 - ◆ `person.delete({ where: { name: "Anna"}})` -> deletes obj2
 - ◆ `person.findAll({ where: { name: {$like: "%nna%"}}})` -> [obj2]
 - `{ answer: {$like: "%po%"}}` devuelve los quizzes donde name es *po*
 - ◆ `person.findOne({ where: { age: {$gt: 25}}})` -> obj2
 - `{ age: {$gt: 20}}` devuelve los quizzes con age > 20
 - ◆ `person.count({ where: { age: {$gt: 25}}})` -> 2
- **limit and offset:** límite de objetos y comienzo de búsqueda
 - ◆ `person.findAll({ limit:2, offset: 2})` -> [obj2, obj3]
- **order:** ordenar objetos devueltos
 - ◆ `person.findAll({ order: "age"})` -> [obj1, obj3, obj2]
 - ◆ `person.findAll({ order: ["id", "DESC"]})` -> [obj3, obj2, obj1]
- **Documentación**
 - ◆ <http://docs.sequelizejs.com/manual/tutorial/querying.html>



JavaScript

Final del tema

Muchas gracias!