

Corso Sequelize

Cosa è?

Un **ORM** (Object Relational Mapper), ovvero una libreria che ti fa fare chiamate con JS al backend, senza andare di raw query.

Lezione 1 - Models

Passaggi **installazione**:

- 1) Crea Cartella progetto
- 2) Npm init per creare il package.json di configurazione
- 3) Installare pg
- 4) Installare pg-hstore
- 5) Installare sequelize

N.B. L'import deve essere eseguito **{ destrutturando }** il sequelize, così da permettere l'uso dell'intelliSense

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

Esempio:

```
31 Capital.belongsTo
32 // hasOne m
33 Country.has
34 let capital
35 sequelize.s
36 .then(() =>
37   console
38   return
39   .then((data) => {
40     country = data;
41     return Capital.findOne({where: {capitalName: 'Paris'}});
42   })
43   .then((data) => {
44     capital = data;
45     return capital.getCountry(country);
46   })
47   .then(() => {
48     console.log('Capital found:', capital);
49   })
50   .catch(err => console.log('Error:', err));
51 }
```

(method) belongsTo<M> extends Model<any, <any>, T extends Model<any, any>>(this: ModelStatic<M>, target: ModelStatic<T>, options?: BelongsToOptions): BelongsTo<...>

Creates an association between this (the source) and the provided target. The foreign key is added on the source.

Example: Profile.belongsTo(User) . This will add userId to the profile table.

@param target — The model that will be associated with hasOne relationship

@param options — Options for the association

Costruire sequelize e dargli i parametri personali (tipo database, user e password)

Effettuare la connessione in asincrono (es. con una promise)

Schema (Mappa di struttura di relazioni tra tabelle) == DB

Lezione 2 - Models

Schema == Model

|

+--- Tabella == Oggetto

Const Users = sequelize.define() → definiamo il model Users e creiamo la tabella

Templettiamolo:

```
Const Users = sequelize.define('nome tabella', {  
  colonna1:{  
    type: Sequelize.DataTypes.STRING // tipo di dato  
  }, colonna2:{  
    }, ecc...}  
{  
  Opzioni varie  
})
```

I DataType in Sequelize

```
const { DataTypes } = require("sequelize"); // Import the built-in data types
```

Strings

<code>DataTypes.STRING</code>	<code>// VARCHAR(255)</code>	
<code>DataTypes.STRING(1234)</code>	<code>// VARCHAR(1234)</code>	
<code>DataTypes.STRING.BINARY</code>	<code>// VARCHAR BINARY</code>	
<code>DataTypes.TEXT</code>	<code>// TEXT</code>	
<code>DataTypes.TEXT('tiny')</code>	<code>// TINYTEXT</code>	
<code>DataTypes.CITEXT</code>	<code>// CITEXT</code>	PostgreSQL and SQLite only.
<code>DataTypes.TSVECTOR</code>	<code>// TSVECTOR</code>	PostgreSQL only.

Boolean

<code>DataTypes.BOOLEAN</code>	<code>// TINYINT(1)</code>
--------------------------------	----------------------------

Numbers

<code>DataTypes.INTEGER</code>	<code>// INTEGER</code>	
<code>DataTypes.BIGINT</code>	<code>// BIGINT</code>	
<code>DataTypes.BIGINT(11)</code>	<code>// BIGINT(11)</code>	
<code>DataTypes.FLOAT</code>	<code>// FLOAT</code>	
<code>DataTypes.FLOAT(11)</code>	<code>// FLOAT(11)</code>	
<code>DataTypes.FLOAT(11, 10)</code>	<code>// FLOAT(11,10)</code>	
<code>DataTypes.REAL</code>	<code>// REAL</code>	PostgreSQL only.
<code>DataTypes.REAL(11)</code>	<code>// REAL(11)</code>	PostgreSQL only.
<code>DataTypes.REAL(11, 12)</code>	<code>// REAL(11,12)</code>	PostgreSQL only.

```
DataTypes.DOUBLE           // DOUBLE
DataTypes.DOUBLE(11)       // DOUBLE(11)
DataTypes.DOUBLE(11, 10)   // DOUBLE(11,10)
```

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

Dates

```
DataTypes.DATE             // DATETIME for mysql / sqlite, TIMESTAMP WITH TIME ZONE for
                             postgres
DataTypes.DATE(6)          // DATETIME(6) for mysql 5.6.4+. Fractional seconds support with
                             up to 6 digits of precision
DataTypes.DATEONLY         // DATE without time
```

Attributi:

allowNull == NOT NULL

defaultValue : il valore di default sarà sempre null, a meno che questo non venga settato.

NomeTabella.sync()...

// Viene usato per sincronizzare il database su postgres al model che lo astrae

NomeTabella.sync().promise

//...ad esso si associa una promise per restituire la tabella, se sincronizzato correttamente

N.B. : Sequelize **pluralizza** in automatico la tabella generata

Ma con l'oggetto opzioni possiamo freeze questa cosa (**freezeTableName: true**)

createdAt e **updatedAt**: sono aggiunti in automatico e servono ad indicare quando un campo è creato o aggiornato.

Per definire un **id autoincrementante** e con constraint da **primary key**:

```
person_id: {
  type: Sequelize.DataTypes.INTEGER,
  primaryKey: true,
```

```
    autoIncrement: true
  },
```

{force: true} : Se viene effettuata una sincronizzazione con un model con stesso nome di una tabella già esistente, quest'ultima verrà dropata in favore del model, se il force: true sarà presente.

// Distruzione e reinserimento

{alter: true} : Effettuerà un'alterazione della tabella presente in db con quella sincronizzata.

// Nuovo inserimento

- `User.sync()` - This creates the table if it doesn't exist (and does nothing if it already exists)
- `User.sync({ force: true })` - This creates the table, dropping it first if it already existed
- `User.sync({ alter: true })` - This checks what is the current state of the table in the database (which columns it has, what are their data types, etc), and then performs the necessary changes in the table to make it match the model.

Example:

```
await User.sync({ force: true });
console.log("The table for the User model was just (re)created!");
```

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

// Questo **forzerà** la **sostituzione** nel DB di ogni model implementato con quelli inseriti qui

```
sequelize.drop({match: /_test$/});
```

// Questo forzerà il drop di ogni tabella che finisce in "_test"

|

+--- Questo è un **RegEx Pattern***

*es. pizza_test, o tabella_test

Sequelize.model.NomeModel → Permette di entrare nel model che vogliamo

Es. : **Sequelize.model.Person**

Lezione 3 – Models Instance

Infiliamo i dati dentro il model

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

così da evitare:

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

e mettere:

```
type: DataTypes.INTEGER,
```

Model.build() → Ci permette di riempire il nostro model

Person.build()

Dati inseriti...

```
const person = Person.build({first_name: "Thomas", last_name: "Anderson", role: "user", email: "nostromo@gmail.com", password: "enter", WittRocks: "true"});
```

...e salvati

// Qui nel mezzo è possibile anche **modificare i valori prima del save!**

```
return person.save(); // Salverà in modo asincrono i dati nel database
```

model.create() // Crea e aggiunge nuove row per la tabella

Esempio:

```
return Person.create({
  first_name: "John",
  last_name: "Rambo",
  role: "admin",
  email: "rocky@gmail.com",
  password: "tunf",
  WittRocks: "true"
});
```

data.toJSON() // Restituisce in **formato JSON** il data

Aggiornare dati in corso di inserimento:

Person.sync({alter: true})

| SINC

.then(() => {

```
  console.log("Table and Model connected");
```

```

return Person.create({
  first_name: "John",
  last_name: "Rambo",
  role: "admin",
  email: "rocky@gmail.com",
  password: "tunf",
  WittRocks: "true"
});
})
.then((data) => {
  data.first_name = "Sam";
  return data.save(); *// Updatng di un valore dell'oggetto
})
.then((data) => {
  console.log("Person Aggiornato");
  console.log(data);
})
.catch((err) => {
  console.log(err);
  console.log("Table and Model Not connected");
})

```

| AGGIUNGE E SALVA ROW

| MODIFICA ROW

| LOGGA L'AGGIORNAMENTO

| CONTROLLA ERRORI

Distruggere un utente in fase di inserimento:

```
*data.destroy();
```

Ritornare il dato dell'utente a come era in fase di creazione:

```
*data.reload();
```

Salvare solo alcuni dei capi modificati:

```
*data.save({fields: ['first_person', 'last_name']});
```

// In questo caso saranno salvate le modifiche solo ai campi specificati

Aumentare e diminuire valori di numeri interi (integer):

```
data.decrement({ age: 2 }); // data.increment({ age: 2 });
```

// Decremento/Incremento di 2 unità alla colonna age

Molteplici aggiunte di più Models in una volta:

```
return Person.bulkCreate([oggetto1, oggetto2]);
```

N.B. Non segue certe limitazioni definite dal template del model iniziale a differenza del normale .create();

A meno che non si specifichi un oggetto dentro l'array che le imposta

Es.:

```
Model.bulkCreate([{}], { validate: true })
```

Lezione 4 – Models Querying

Person.findAll() : cerca e ritorna tutti gli elementi della tabella Person

Esempio:

```
Person.sync({alter: true})
.then(() => {
  console.log("Table and Model connected");
  return Person.findAll(); // Trova tutti gli oggetti inseriti della mia tabella
  Person e ti rimanda un array con essi dentro...
})
.then((data) => {
  data.forEach(element => {
    console.log(element.toJSON()); //...stampabile come un forEach
  });
})
.catch((err) => {
  console.log(err);
  console.log("Table and Model Not connected");
});
```

Return di solo alcuni campi?

```
Person.findAll({attributes: ['first_name', 'last_name']})
```

...cambiare nome key (labels) dei campi returnati?

```
Person.findAll({attributes: [['first_name', 'Nome'], ['last_name', 'Cognome']]});
```

Sequelize Aggregations con `sequelize.fn()`

```
Person.findAll({attributes: [[sequelize.fn('SUM', sequelize.col('age')), 'howOld']]});
```

Sintassi:

`sequelize.fn('METODO SQL', QUALI COLONNE USARE), LABEL ALL'OUTPUT`

Escludere dal return una colonna:

```
Person.findAll({attributes: {exclude: 'password'}});
```

Returnare solo oggetti dove il parametro ha un valore specifico:

```
Person.findAll({where: {age:17}}); // Returna solo oggetti con age = 17
```

Returnare solo nome oggetti dove il parametro ha un valore specifico:

```
return Person.findAll( { attributes: ['first_name'], where: {age:17} } );
```

LIMIT:

Returna solo un limitato numero di item

```
Person.findAll( { limit: 2 } ); // Returna solo 2 item LIMIT
```

ORDER BY:

```
Person.findAll( { order: [['age', 'DESC']] } );
```

// Ordina array per età discendente

GROUP BY:

Esempio:

Facciamo una somma delle età in base al first_name

```
return Person.findAll(  
  {  
    attributes: [  

```



```

        'first_name',
        [sequelize.fn('SUM', sequelize.col('age')), 'sum_age']
    ],
    group: 'first_name'
  }
);

```

In puro SQL:

```
SELECT "first_name", SUM("age") AS "sum_age" FROM "person" AS "person" GROUP BY "first_name";
```

Operatore logico **OR** e **AND**

const { DataTypes, Op } = require("sequelize"); // Importiamolo in sequelize direttamente

Esempio **OR**:

```

return Person.findAll( {
  where: {
    [Op.or]:{
      first_name:'John', age:42
    }
  }
}); // Trova tutti dove first_name = 'John' o age = 42

```

In puro SQL:

```
SELECT "person_id", "first_name", "last_name", "age", "role", "email", "password", "WittRocks",
"createdAt", "updatedAt" FROM "person" AS "person" WHERE ("person"."first_name" = 'John' OR
"person"."age" = 42)
```

Esempio **AND**:

```

return Person.findAll( {
  where: {
    [Op.and]:{
      first_name:'John', age:42
    }
  }
}); // Trova tutti dove first_name = 'John' e age = 42

```

In puro SQL:

```
SELECT "person_id", "first_name", "last_name", "age", "role", "email", "password", "WittRocks",
"createdAt", "updatedAt" FROM "person" AS "person" WHERE ("person"."first_name" = 'John' AND
"person"."age" = 42);
```

Number Comparator:

Op.gt → Comparatore che controlla se il valore in colonna sia maggiore al parametro inserito

Esempio:

```
return Person.findAll( {
  where: {
    age: {
      [Op.gt]: 25
    }
  }
} );
```

Esempio 2 age e or:

```
return Person.findAll( {
  where: {
    age: {
      [Op.or]: {
        [Op.lt]:18, // Minore a
        [Op.eq]:null, // Uguale a
      }
    }
  }
} ); // Trova tutti quelli con age minore a o uguale a
```

[Op.lt] = Minore a , [Op.eq] = Uguale a

Filtrare in base a lunghezza nome con **sequelize.where()**

Esempio:

```
return Person.findAll( {
  where:
    sequelize.where( sequelize.fn('char_length',
sequelize.col('first_name')), 3 ) // Ci permette di usare la sequelize.fn(funzione)
dentro il where
}); // Filtrare solo per lunghezza del first_name predefinita (in questo caso
uguale a 3)
// Filtrare solo per lunghezza del first_name predefinita (in questo caso uguale a 3)
```

sequelize.where() : permette di usare dentro di se il sequelize.fn()

Sintassi:

sequelize.fn(cosa, colonna, numero)

Update Query:

Uso dell'**update method** per aggiornare un oggetto in tabella.

Esempio:

```
return Person.update(
  { first_name: 'Lizzie' },
  {
```

```

        where: {
          age: 17
        }
      }
    )
  })

```

// Cambia name all'oggetto che ha age = 17

Sintassi:

Model.update()

Esempio 2:

```

return Person.update(
  { first_name: 'JohnOld' },
  {
    where: {
      age: {
        [Op.gt]: 41
      }
    }
  }
)
}) // Cambia name all'oggetto che ha age > 41 anni

```

Model.destroy() : Usato per distruggere un oggetto

{truncate: true} : Se usa come parametro {truncate: true} cancellerà le row, ma la tabella rimarrà, anche se vuota.

Model.max() : Usato per restituire il **valore massimo in una colonna**

Esempio: Person.max('age')

Model.sum() : Usato per sommare tutti i valori di una colonna

Esempio: Person.sum('age')

Lezione 5 – Esercitazione

(https://www.youtube.com/watch?v=zAY-lvbBXZc&list=PLkqiWyX-_Lov8qmMOVn4SEQwr9yOjNn3f&index=5)

Lezione 6 – Finder Methods

Finder Method: Metodi che generano SELECT query.

Ricorda:

Oggetto Sequelize === Row in SQL

“By default, the results of all finder methods are instances of the model class (as opposed to being just plain JavaScript objects). This means that after the database returns the results, Sequelize automatically wraps everything in proper instance objects”

I più importanti:

- `findAll()`
 - `findByPk()`
 - `findOne()`
 - `findOrCreate()`
 - `findAndCountAll()`
-

{raw : true} : parametro per stampare in **raw json** la **row** richiesta con il metodo.

- **findAll()**: Serve a richiamare tutto

Visto prima, serve a richiamare tutto

- **findByPK()**: Richiama oggetto con primary key specifica

```
Es.: return Person.findByPk(7); // Trova per primary key = 7
```

- **findOne()**: Ritorna il primo che trova (è possibile inserire dei parametri **where** all'interno)

```
Es.: return Person.findOne(); // Ritorna il primo che trova
```

- **findOrCreate()**: Se non lo trova, lo crea (attenzione ai **defaultValue!**) *

```
Es.: return Person.findOrCreate( {where: {first_name: 'Rocky', last_name: 'Balboa', age: 55 , role: 'Mod', email: 'Rocky@gmail.com', password: 'Adriana'}});
```

- **findAndCountAll()**: Conta rows con dati specifici

```
Es.: return Person.findAndCountAll(  
  {  
    where: {  
      first_name: 'JohnOld'  
    },  
    raw: true  
  }  
)
```

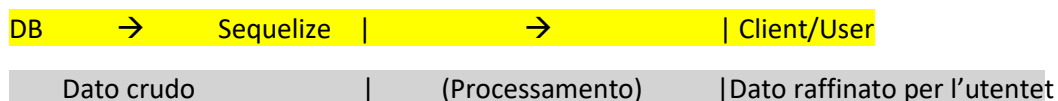
```
} // Mi conta le rows con quei dati
```

* The method `findOrCreate` will create an entry in the table unless it can find one fulfilling the query options. In both cases, it will return an instance (either the found instance or the created instance) and a boolean indicating whether that instance was created or already existed.

Quindi non verrà returnato soltanto il raw json dell'oggetto, ma molto di più.

Lezione 7 – Setters, Getters e Virtual Fields

Possiamo scegliere come far inviare i dati al nostro user a partire dal DB.



Getters e Setters non supportano le funzioni Asincrone, solo quelle sincrone.

Get(): Funzione che prende il valore della colonna, quando pronto.

Permette la modifica in post-chiamata.

Esempio in template del Model:

...

```
first_name: {  
  type: DataTypes.STRING,  
  allowNull: false,  
  validate: {
```

```

    len: [4,7] // Nome valido da inserire deve essere tra 4 e i 7 caratteri
  },
  get() {
    const rawValue = this.getDataValue('first_name'); // Prendo il valore
    return rawValue.toUpperCase(); // Lo rendere maiuscolo
  }
}

```

// Output: THOMAS (usando il findOne())

...

Set(): Funzione che prende il valore della colonna e lo modifica prima dell'inserimento in DB.

Esempio in template del Model:

...

```

password: {
  type: Sequelize.DataTypes.STRING,
  set(value) {
    this.setDataValue('password', hash);
  }
},

```

// Permette di prendere la password e usare un hashing per criptarla, il tutto prima di postarla dentro il DB

...

N.B. : Per usare questo hashing bisogna:

Installare il pacchetto (**npm install bcrypt**)

Importarlo **const bcrypt = require('bcrypt');**

```

set(value) {
  const salt = bcrypt.genSaltSync(12);
  const hash = bcrypt.hashSync(value, salt);
  this.setDataValue('password', hash);
}

```

Salt: particella che aiuta la generazione dell'hashing

Hashing \neq Encrypting

Risultato del setting:

```

return Person.create({
  first_name: 'James',
  last_name: 'Bond',
  password: 'colt'
});
}) //
.then((data) => {
  console.log(data.first_name);
  console.log(data.last);
  console.log(data.password);

```

```
} )
```

// Output:

JAMES

undefined

\$2b\$12\$z.Z7M0lLdf2wk91Zx36.6uaBSyYx2rhxUHn2LlifaFzi4pxYlJ8/e

I **get** e i **set** sono **combinabili** per fare molte cose.

Esempio: Utente posta un contenuto, questo è compresso (set prima di infilarlo nel DB), quando l'utente vuole leggerlo, questo verrà decompresso (get prima che venga sparato verso il client).

Prima installiamo un pacchetto per comprimere dati

npm install zlib

e importiamolo

```
const zlib = require('zlib'); // NPM per Compressione
```

Impostiamo set e get per comprimere e decomprimere

...

```
description: {
  type: DataTypes.STRING,
  set(value) {
    const compressed = zlib.deflateSync(value).toString('base64');
    this.setDataValue('description', compressed)
  },
  get() {
    const value = this.getDataValue('description')
    const uncompressed = zlib.inflateSync(Buffer.from(value, 'base64'));
    return uncompressed.toString();
  }
}
```

```
}  
}
```

...

```
Person.sync({alter: true})  
.then(() => {  
  console.log("Table and Model connected");  
  // return Person.findOne();  
  return Person.create({  
    first_name: 'Andy',  
    last_name: 'Wharol',  
    description: 'Good Man'  
  });  
}) //  
.then((data) => {  
  console.log(data.first_name);  
  console.log(data.last);  
  console.log(data.description);  
})
```

...

// Output: ANDY undefined Good Man

Virtual Fields : Campi volatili non inseriti fisicamente nel DB, ma derivati da combinazioni da campi esistenti fisicamente

Es:

first_name → Esiste

last_name → Esiste

first_name e last_name → Esistono “virtualmente” (**virtual field**).

```
aboutUser: {  
  type: DataTypes.VIRTUAL,  
  get() {  
    return `${this.first_name} ${this.description}`;  
  }  
}
```

Esempio:

```
console.log("Table and Model connected");  
// return Person.findOne();  
/* return Person.create({  
  first_name: 'Andy',  
  last_name: 'Wharol',  
  description: 'Good Man'
```



```

    }); */

    return Person.findOne({ where: {first_name: 'Andy'} });
  }) //
  .then((data) => {
    /*      console.log(data.first_name);
      console.log(data.last);
      console.log(data.description); */
    console.log(data.aboutUser)
  })

```

Lezione 8 – Validators e Constraints

Constraints: limitazioni poste sui valori delle colonne che ne limitano e monitorano l'attività.

Es: I nickname in colonna nicknames devono essere unici, ciò limita a poter inserire soltanto nickname univoci.

Le C. possono essere limitazioni anche relative al tipo di dato inseribile in DB.

Esempio:

```

nickname: {
  type: DataTypes.STRING,
  unique: true
}

```

Creiamo ed inseriamo una nuova row:

```

return Person.create(
  {
    first_name: "John",
    last_name: "Abbondio",
  }
)

```

```
        password: "nonsadafare",
        nickname: "DoomGuy"
    }
};
```

// Funziona

Inseriamo nuovamente una row **con stesso nickname**:

error: un valore chiave duplicato viola il vincolo univoco "person_nickname_key"

Errore, il nickname ha un cotraint limitante che lo rende univoco per row.

Sistema di validazione per età:

Inseriamo una funzione di validazione che controlla se age < 21 :

```
age: {
  type: DataTypes.INTEGER,
  allowNull: true,
  validate: {
    isOldEnough(value) {
      if (value < 21) {
        throw new Error("Too young!")
      }
    }
  }
},
```

Tentiamo di romperla inserendo un utente < 21 :

```
return Person.create({
  first_name: 'Junior',
  last_name: 'Namec',
  age: 14
})
```

// Verrà scatenato l'errore controllato e l'aggiunta al DB verrà bloccata.

Output: **original: Error: Too young!**

E' possibile fare validazioni per ogni cosa: inserimento del tipo corretto di dato, email di un certo tipo, ecc...

Come fare interagire NULL con i VALIDATORS:

Esempio:

Validator che da un errore controllato in caso di inserimento di un nickname null (vuoto)

```
nickname: {
```

```

    type: DataTypes.STRING,
    unique: true,
    allowNull: true,
    validator: {
        myNicknameValidator(value) {
            if (value === null) {
                throw new Error("Inserisci un Nickname");
            }
        }
    }
}
}

```

```

return Person.create({
    first_name: 'Big',
    last_name: 'Cave',
    age: 55,
    nickname: null
})

```

// Throw l'errore personalizzato

N.d.r. ATTENZIONE, ultimo script da ricontrollare.

VALIDATOR che controlla che first_name e password siano diversi:

Validator:

...

```

freezeTableName: true, // Evitiamo la pluralizzazione del nome del model
validate: {
    firstnamePassMatch() {
        if (this.first_name === this.password) {
            throw new Error("Il Name deve essere differente dalla password");
        } else {
            console.log("Passed");
        }
    }
}
}

```

...

```

return Person.create({
    first_name: 'BigJam',
    last_name: 'Cave',

```

```
    age: 55,  
    password: 'BigJam',  
  })
```

Essendo first_name e password uguali, non verrà creato

// Output: Error: Il Name deve essere differente dalla password

Lezione 9 – SQL Injection e Raw Queries

Come prevenire le SQL Injections:

`sequelize.query('COMANDO IN SQL PURO')`: Funzione che permette di inserire comandi in SQL puro

```
    return sequelize.query(`UPDATE person SET age = 37 WHERE first_name = 'Nick'`);  
  }) //  
  .then((data) => {  
    [result, metadata] = data; // L'output è il risultato e dei metadati, questi  
    ultimi cambiano in base al DBMS usato  
    console.log(data);  
  })
```

// L'output è il risultato e dei metadati, questi ultimi cambiano in base al DBMS usato

Replacements:

(Da rivedere)

Lezione 10 – Paranoid Table

paranoid: `true`

Parametro che permette di effettuare un `.destroy()` che non cancella totalmente la row, ma la tiene in record, effettuando un **soft deleting**.

Esempio:

```
return Person.destroy({where: {person_id: 24 }}) // Cancella row con id 24
```

In SQL puro:

```
UPDATE "person" SET "deletedAt"=$1 WHERE "deletedAt" IS NULL AND "person_id" = $2
```

// E' cancellato, ma tenuto nei record

MA è possibile forzare la cancellazione brutale tramite un **force: true** o una **Raw Query diretta**.

Esempio:

...

```
{
  where: {person_id: 24 },
  force: true // forza la cancellazione
}
```

...

Restaurare un soft deleted object:

```
return Person.restore({where: {person_id: 24}})
// Ripristina row con id 24 soft-cancellata prima
```

Usando **sequelize** (es. `findAll()`) la soft-deleted row verrà saltata, a meno che non venga inserito il parametro (`{paranoid: false}`)

Usando una **Raw Query** (`sequelize.query(SELECT * FROM person)`) anche quella soft-deleted verrà stampata.

Lezione 11 - Associations

(hasOne, belongsTo, hasMany, belongsToMany)

Tabella A ----- Tabella B

Primary Key: Identificativo univoco di una colonna per identificare univocamente rows

Tabella A__colonna | Tabella B__colonna

Foreign Key: Identificativo che associa una **Tabella Madre (A)** ad una **Tabella Figlia (B)** secondo un id univoco di riferimento

Tabella A

|

+--- Tabella B

1) **Associazione OnetoOne**: Una row di tabella A è legata a solo una row di una tabella B

Esempio:

Persona X ha un Codice fiscale X, Stato Y ha un Capitale Y, ecc...

La **funzione** che le lega è **Biunivoca**.

Con **Sequelize** è possibile legare con **4 tipi di funzione** le nostre **3 tipologie di Relazioni***:

- 1) **hasOne**
- 2) **belongsTo**
- 3) **hasMany**
- 4) **belongsToMany**

* Le Relazioni OnetoOne, OnetoMany, ManytoMany

Esempio:

```
5) /* ----- Associazione OnetoOne, Uno a Uno ----- */
```

Creazione Model di Country:

```
6) const Country = sequelize.define('country', {  
7)   countryName: {  
8)     type: DataTypes.STRING,  
9)     unique: true  
10)  }},  
11)  {  
12)    timestamps: false  
13)  }  
14));
```

Creazione Model di Capital:

```
15) const Capital = sequelize.define('capital', {  
16)   capitalName: {  
17)     type: DataTypes.STRING,
```

```

18)      unique: true
19)    }},
20)    {
21)      timestamps: false
22)    }
23));

```

Associazione con metodo `hasOne()`:

```

24) // hasOne method
25) Country.hasOne(Capital);

```

Sincronizzazione con DB:

```

26) sequelize.sync({alter: true})
27).then(() => {
28)   console.log("Connected");
29})
30).then((data) => {
31)   console.log(data);
32})
33).catch((err) => {
34)   console.log("Not connected");
35});

```

// Output: Creazione e legame tramite Foreign Key effettuato

E' possibile passare **parametri** aggiuntivi ad `hasOne()`:

`Country.hasOne(Capital, {foreignKey: 'soccer'})`

// serve a specificare il nome del Foreign Key in 'soccer'.

Riempiamo con `Model.bulkCreate([])`

Settiamo il **legame** tra **Capital** e **Country** usando i metodi `hasOneBelongTo()` e `hasManyBelongMany()`

Spain --- Madrid

England --- London

...

Set → Get → Create

1. Esempio:
2. Settiamo la capital
3. La otteniamo
4. Settiamo la countryId

5. La otteniamo

6. Visualizziamo il legame tra Capital e Country per quella row

...

Set Capital

```
return Capital.findOne({where: {capitalName: 'Madrid'}})
})
.then((data) => {
  capital = data;
  return Country.findOne({where: {countryName: 'Spain'}})
})
.then((data) => {
  country = data;
  country.setCapital(capital);
})
```

...

// Abbiamo settato il **countryId** di **Madrid** su Spain

—

Get Capital

...

```
return Country.findOne({where: {countryName: 'Spain'}})
})
.then((data) => {
  country = data;
  return country.getCapital();
})
.then((data) => {
  console.log(data.toJSON());
})
```

// In console otterremo la **capital** con relativo **countryId**

—

Creiamo capitale e country direttamente e li leghiamo contestualmente:

...

```
return Country.create(
  {countryName: 'USA'}
)
})
.then((data) => {
  country = data;
  return country.createCapital(
    {capitalName: 'Washington'}
  )
})
```

```
.then((data) => {
  console.log(data)
})
```

...

// Creerà una **country** e la **capitale**. Con `createModel()` (`createCapital()`) verranno legate da un **Foreign Key**.

Associazione con metodo `belongsTo()`*:

```
Capital.belongsTo(Country);
// Mette il nostro FK (countryId) nella nostra tabella Capitals
```

Simile a...

```
Country.hasOne(Capital)
```

...ma al contrario*.

N.B. Conviene **usare entrambi** così da permette le associazioni in entrambi i sensi delle tabelle!

Esempio esteso:

...

```
Country.hasOne(Capital); /*, {foreignKey: 'soccer'} | serve a specificare il nome
del Foreign Key */
Capital.belongsTo(Country); // Stesso di sopra, ma al contrario
let capital, country; // Ciò che unirà le colonne del Foreign Key
sequelize.sync({alter: true})
.then(() => {
  console.log("Connected");
  return Country.findOne({where: {countryName: 'France'}});
})
.then((data) => {
  country = data;
  return Capital.findOne({where: {capitalName: 'Paris'}});
})
.then((data) => {
  capital = data;
  return capital.setCountry(country);
})
.then((data) => {
  console.log(data);
})
```

...

*Entrambi accettano **parametri** opzionali dentro.

Parametri opzionali utili da usare con **hasOne()** e **belongsTo()**:

Sintassi:

onDelete: 'CASCADE'

Esempi:








```
Country.hasOne(Capital, {onDelete: 'CASCADE'});
Capital.belongsTo(Country, {onDelete: 'CASCADE'});
// Cancellando un elemento in tabella, pur essendoci un riferimento (che permette incancellabile
l'elemento nella tabella figlia, questo permetterà di eliminare il FK di legame)
```

...







```
.then(() => {
  console.log("Connected");
  return Country.destroy({where: {countryName: 'Spain'}})
})
```

...

Prima:

Data Output				Explain	Messages	Notifications	Data Output				Explain	Messages	Notifications
	id [PK] integer 		countryName character varying (255) 					id [PK] integer 		capitalName character varying (255) 		countryId integer 	
1	1		Spain				1	1		Madrid		1	
2	2		France				2	2		Paris		2	
3	3		Germany				3	3		Berlin		[null]	
4	4		England				4	4		London		[null]	
5	24		USA				5	5		Washington		24	

Dopo:

Data Output		Explain	Messages	Notifications	Data Output		Explain	Messages	Notifications
	id [PK] integer 		countryName character varying (255) 			id [PK] integer 		countryName character varying (255) 	
1	2		France		1	2		France	
2	3		Germany		2	3		Germany	
3	4		England		3	4		England	
4	24		USA		4	24		USA	

// Cancelli la Country, cancelli anche la Capital associata dal FK!

onUpdate():

```
Country.hasOne(Capital, {onUpdate: 'CASCADE'});
Capital.belongsTo(Country, {onDelete: 'CASCADE'});
```

```
    return Country.findOne({where: {countryName: 'France'}});
  })
  .then((data) => {
    country = data;
    return Capital.findOne({where: {capitalName: 'London'}});
  })
  .then((data) => {
    capital = data;
    return country.setCapital(capital);
  })
}
```

...poi...

```
    return Country.findOne({where: {countryName: 'France'}});
  })
  .then((data) => {
    country = data;
    // return Capital.findOne({where: {capitalName: 'London'}});
    return Capital.findOne({where: {capitalName: 'Paris'}});
  })
  .then((data) => {
    capital = data;
    return country.setCountry(country);
  })
}
```

// Aggiornamento dell'FK di London collegato a France, facendo così avremo una relazione **hasMany**,

poiché **belongsTo**, a differenza di **hasOne**, supporta tale tipo di **associazioni**!

| France |

|

+--- Paris (countryId : 2) |

+--- London (countryId : 2) |

2) Associazione **OnetoMany**: Una row di tabella A è legata a più di una row di una tabella B

Esempio:

Tabella A (User) --- Tabella B (Posts)

User

|
+--- Post 1
+--- Post 2
+--- Post 3

```
const User = sequelize.define('user', {
  username: {
    type: DataTypes.STRING
  },
  password: {
    type: DataTypes.STRING
  }},
{
  timestamps: false
});

const Post = sequelize.define('post', {
  message: {
    type: DataTypes.STRING
  }},
{
  timestamps: false
});

User.hasMany(Post);
Post.belongsTo(User);
// con onDelete, quando un utente verrà cancellato, ogni post suo verrà cancellato
```

Creiamo degli **utenti** e dei **posts** di prova:

...

```
User.bulkCreate([
```

...

```
Post.bulkCreate([
```

...

Tale tipo di **associazione** ha molti metodi (**helper methods**) che aiutano ad eseguire i legami:

Esempio: **user.addPosts(posts)**

```
    return User.findOne({where: {username: 'Otto'}});
  })
  .then((data) => {
    user = data;
    return Post.findAll();
  })
  .then((data) => {
    posts = data;
    return user.addPosts(posts);
  })
  .then((data) => {
    console.log(data);
  })
})
// Questo leggerà tutti i posts all'utente otto
```

Esempio: **user.countPosts(posts)**

Conta **quante rows** solo legate ad un item di un'altra tabella

```
    return User.findOne({where: {username: 'Otto'}});
  })
  .then((data) => {
    user = data;
    // return Post.findAll();
    return user.countPosts();
  })
})
// Output: 10
```

Esempio: `user.removePosts(posts)`

Cancella post associato allo user:

...

```
return User.findOne({where: {username: 'Otto'}});
})
.then((data) => {
  user = data;
  return Post.findOne()
})
.then((data) => {
  posts = data;
  return user.removePost(posts)
})
.then((data) => {
  console.log(data);
})
}
```

...

// In questo caso cancella il primo post (`findOne()` senza parametri prende il **primo post**, con `findAll()` avremmo cancellato i FK di tutti i posts legati)

Prima:

Data Output	Explain	Messages	Notifications
id [PK] integer	message character varying (255)	userid integer	
1	1 Sono il post 1	1	
2	2 Sono il post 2	1	
3	3 Sono il post 3	1	
4	4 Sono il post 4	1	
5	5 Sono il post 5	1	
6	6 Sono il post 6	1	
7	7 Sono il post 7	1	
8	8 Sono il post 8	1	
9	9 Sono il post 9	1	
10	10 Sono il post 10	1	

Dopo:

Data Output	Explain	Messages	Notifications
id [PK] integer	message character varying (255)	userid integer	
1	1 Sono il post 1	[null]	
2	2 Sono il post 2	1	
3	3 Sono il post 3	1	
4	4 Sono il post 4	1	
5	5 Sono il post 5	1	
6	6 Sono il post 6	1	
7	7 Sono il post 7	1	
8	8 Sono il post 8	1	
9	9 Sono il post 9	1	
10	10 Sono il post 10	1	

onDelete e **onUpdate** con **hasMany**:

...







```
User.hasMany(Post, {onDelete: 'CASCADE'});
Post.belongsTo(User, {onDelete: 'CASCADE'});
```

...

```
return User.destroy({where: {username: 'Otto'}});
})
```

...







// Ha cancellato l'utente e tutti i posts ad esso legati. Ne rimarrà uno solo, quello di cui avevamo **rimosso** lo **username_id**

Data Output	Explain	Messages	Notifications	Data Output	Explain	Messages	Notifications
 id [PK] integer		message character varying (255)	 userId integer	 id [PK] integer		username character varying (255)	 password character varying (255)
1	1	Sono il post 1	[null]	1	2	Mike	Zanzibar
				2	3	BigJim	excellent

belongsTo() method con **OneToMany**

```
return User.findOne();
})
.then((data) => {
  user = data;
  return Post.findOne();
})
.then((data) => {
  posts = data;
  posts.setUser(user);
})
```

// Abbiamo associato al primo user il primo post trovato con **posts.setUser(user)**

Data Output	Explain	Messages	Notifications	Data Output	Explain	Messages	Notifications
 id [PK] integer		message character varying (255)	 userId integer	 id [PK] integer		username character varying (255)	 password character varying (255)
1	1	Sono il post 1	2	1	2	Mike	Zanzibar
				2	3	BigJim	excellent

3) Associazione ManytoMany: Più rows di tabella A sono legate a più rows di una tabella B.

Tabella A --- Tabella B

Esempio: Consumatori --- Prodotti

Consumatore 1

|
+--- Prodotto 1 x2
+--- Prodotto 2 xN
+--- Prodotto 3 xN
|

Consumatore 2

Problema:

Di base i **DB non permettono direttamente** questo tipo di relazione,

Soluzione:

...ma è possibile aggirare il problema tramite più **OneToMany**.

Per effettuare ciò si effettueranno dei **JOIN**, che in sequenze daranno vita a dei **Junction Model**.

```
Customer.belongsToMany(Product, {through: 'customerproduct'});  
Product.belongsToMany(Customer, {through: 'customerproduct'});
```

{through: 'customerproduct'}: Un oggetto per creare la junction model definita dalla JOIN per legare le 2 tabelle.

È possibile aggiungere parametri aggiuntivi, come:

```
...through: 'customerproduct', foreignKey: 'customer_id'});  
...through: 'customerproduct', foreignKey: 'product_id'});  
// Che permetteranno di rinominare il FK.
```

È possibile creare una **custom Junction Table**:

```
// Custom junction Table
const CustomerProduct = sequelize.define('customerproduct', {
  customerproductId: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  {
    timestamps: false
  }
});
Customer.belongsToMany(Product, {through: 'CustomerProduct'});
Product.belongsToMany(Customer, {through: 'CustomerProduct'});
// Ricorda di cambiarla nei parametri di through!
```

Riempiamo i nostri model:

...

```
Customer.bulkCreate([
```

...

```
Product.bulkCreate([
```

...

Helper methods per belongsToMany(): customer.addProducts(product);

Esempio 1:

```
let customer, product;
sequelize.sync({alter: true})
.then(() => {
  console.log("Connected");

  return Customer.findOne({where: {customerName: 'Witt'}});
})
.then((data) => {
  customer = data;
  return Product.findAll();
})
.then((data) => {
  product = data;
  customer.addProducts(product);
})
```

// Il customer_id per ogni prodotto è 1 (ovvero Witt, il primo customer trovato)

	Data Output	Explain	Messages	Notifications
	createdAt timestamp with time zone	updatedAt timestamp with time zone	customerid [PK] integer	productid [PK] integer
1	2022-01-20 14:07:54.422+01	2022-01-20 14:07:54.422+01	1	1
2	2022-01-20 14:07:54.422+01	2022-01-20 14:07:54.422+01	1	2
3	2022-01-20 14:07:54.422+01	2022-01-20 14:07:54.422+01	1	3
4	2022-01-20 14:07:54.422+01	2022-01-20 14:07:54.422+01	1	4

"Tabella Junction creata dalla relazione customer-product"

Esempio 2:

```
return Product.findOne({where: {productName: 'laptop'}});
})
.then((data) => {
  product = data;
  return Customer.findAll();
})
.then((data) => {
  customer = data;
  product.addCustomers(customer);
})
// Il laptop sarà legato a tutti i customers
```

	Data Output	Explain	Messages	Notifications
	createdAt timestamp with time zone	updatedAt timestamp with time zone	customerid [PK] integer	productid [PK] integer
1	2022-01-20 14:07:54.422+01	2022-01-20 14:07:54.422+01	1	1
2	2022-01-20 14:07:54.422+01	2022-01-20 14:07:54.422+01	1	2
3	2022-01-20 14:07:54.422+01	2022-01-20 14:07:54.422+01	1	3
4	2022-01-20 14:07:54.422+01	2022-01-20 14:07:54.422+01	1	4
5	2022-01-20 14:13:13.222+01	2022-01-20 14:13:13.222+01	2	1
6	2022-01-20 14:13:13.222+01	2022-01-20 14:13:13.222+01	3	1
7	2022-01-20 14:13:13.222+01	2022-01-20 14:13:13.222+01	4	1

"Tabella Junction creata dalla neo-relazione customer-product"

onDelete e **onUpdate** in **ManyToMany**: In caso di ManyToMany sono "superflui" a causa dei legami già instaurati precedentemente.

```
return Customer.destroy({where: {customerName: 'Witt'}});
```

