Version: v6 - stable

Associations

Sequelize supports the standard associations: One-To-One, One-To-Many and Many-To-Many.

To do this, Sequelize provides **four** types of associations that should be combined to create them:

- The `HasOne` association
- The `BelongsTo` association
- The `HasMany` association
- The `BelongsToMany` association

The guide will start explaining how to define these four types of associations, and then will follow up to explain how to combine those to define the three standard association types (One-To-One, One-To-Many and Many-To-Many).

Defining the Sequelize associations

The four association types are defined in a very similar way. Let's say we have two models, `A` and `B`. Telling Sequelize that you want an association between the two needs just a function call:

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

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

They all accept an options object as a second parameter (optional for the first three, mandatory for `belongsToMany` containing at least the `through` property):

```
A.hasOne(B, { /* options */ });
A.belongsTo(B, { /* options */ });
A.hasMany(B, { /* options */ });
A.belongsToMany(B, { through: 'C', /* options */ });
```

The order in which the association is defined is relevant. In other words, the order matters, for the four cases. In all examples above, `A` is called the **source** model and `B` is called the **target** model. This terminology is important.

The `A.hasOne(B)` association means that a One-To-One relationship exists between `A` and `B`, with the foreign key being defined in the target model (`B`).

The `A.belongsTo(B)` association means that a One-To-One relationship exists between `A` and `B`, with the foreign key being defined in the source model (`A`).

The `A.hasMany(B)` association means that a One-To-Many relationship exists between `A` and `B`, with the foreign key being defined in the target model (`B`).

These three calls will cause Sequelize to automatically add foreign keys to the appropriate models (unless they are already present).

The `A.belongsToMany(B, { through: 'C' })` association means that a Many-To-Many relationship exists between `A` and `B`, using table `C` as junction table, which will have the foreign keys (`aId` and `bId`, for example). Sequelize will automatically create this model `C` (unless it already exists) and define the appropriate foreign keys on it.

Note: In the examples above for `belongsToMany`, a string (`'C'`) was passed to the through option. In this case, Sequelize automatically generates a model with this name. However, you can also pass a model directly, if you have already defined it.

These are the main ideas involved in each type of association. However, these relationships are often used in pairs, in order to enable better usage with Sequelize. This will be seen later on.

Creating the standard relationships

As mentioned, usually the Sequelize associations are defined in pairs. In summary:

- To create a **One-To-One** relationship, the `hasOne` and `belongsTo` associations are used together;
- To create a **One-To-Many** relationship, the `hasMany` and `belongsTo` associations are used together;
- To create a **Many-To-Many** relationship, two `belongsToMany` calls are used together.
  - Note: there is also a *Super Many-To-Many* relationship, which uses six associations at once, and will be discussed in the Advanced Many-to-Many relationships guide.

This will all be seen in detail next. The advantages of using these pairs instead of one single association will be discussed in the end of this chapter.

One-To-One relationships

Philosophy

Before digging into the aspects of using Sequelize, it is useful to take a step back to consider what happens with a One-To-One relationship.

Let's say we have two models, `Foo` and `Bar`. We want to establish a One-To-One relationship between Foo and Bar. We know that in a relational database, this will be done by establishing a foreign key in one of the tables. So in this case, a very relevant question is: in which table do we want this foreign key to be? In other words, do we want `Foo` to have a `barId` column, or should `Bar` have a `fooId` column instead?

In principle, both options are a valid way to establish a One-To-One relationship between Foo and Bar. However, when we say something like *"there is a One-To-One relationship between Foo and Bar"*, it is unclear whether or not the relationship is *mandatory* or optional. In other words, can a Foo exist without a Bar? Can a Bar exist without a Foo? The answers to these questions help figuring out where we want the foreign key column to be.

Goal

For the rest of this example, let's assume that we have two models, `Foo` and `Bar`. We want to setup a One-To-One relationship between them such that `Bar` gets a `fooId` column.

Implementation

The main setup to achieve the goal is as follows:

```
Foo.hasOne(Bar);
Bar.belongsTo(Foo);
```

Since no option was passed, Sequelize will infer what to do from the names of the models. In this case, Sequelize knows that a `fooId` column must be added to `Bar`.

This way, calling `Bar.sync()` after the above will yield the following SQL (on PostgreSQL, for example):

```
CREATE TABLE IF NOT EXISTS "foos" (
  /* ... */
);
CREATE TABLE IF NOT EXISTS "bars" (
  /* ... */
  "fooId" INTEGER REFERENCES "foos" ("id") ON DELETE SET NULL ON UPDATE CASCADE
  /* ... */
);
```

Options

Various options can be passed as a second parameter of the association call.

`onDelete` and `onUpdate`

For example, to configure the `ON DELETE` and `ON UPDATE` behaviors, you can do:

```
Foo.hasOne(Bar, {
  onDelete: 'RESTRICT',
  onUpdate: 'RESTRICT'
});
Bar.belongsTo(Foo);
```

The possible choices are `RESTRICT`, `CASCADE`, `NO ACTION`, `SET DEFAULT` and `SET NULL`.

The defaults for the One-To-One associations is `SET NULL` for `ON DELETE` and `CASCADE` for `ON UPDATE`.

Customizing the foreign key

Both the `hasOne` and `belongsTo` calls shown above will infer that the foreign key to be created should be called `fooId`. To use a different name, such as `myFooId`:

```
// Option 1
Foo.hasOne(Bar, {
  foreignKey: 'myFooId'
});
Bar.belongsTo(Foo);

// Option 2
Foo.hasOne(Bar, {
  foreignKey: {
    name: 'myFooId'
  }
});
Bar.belongsTo(Foo);

// Option 3
Foo.hasOne(Bar);
Bar.belongsTo(Foo, {
  foreignKey: 'myFooId'
});

// Option 4
Foo.hasOne(Bar);
Bar.belongsTo(Foo, {
  foreignKey: {
    name: 'myFooId'
  }
});
```

As shown above, the `foreignKey` option accepts a string or an object. When receiving an object, this object will be used as the definition for the column just like it would do in a standard `sequelize.define` call. Therefore, specifying options such as `type`, `allowNull`, `defaultValue`, etc, just work.

For example, to use `UUID` as the foreign key data type instead of the default (`INTEGER`), you can simply do:

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

Foo.hasOne(Bar, {
  foreignKey: {
    // name: 'myFooId'
    type: DataTypes.UUID
  }
});
Bar.belongsTo(Foo);
```

Mandatory versus optional associations

By default, the association is considered optional. In other words, in our example, the `fooId` is allowed to be null, meaning that one Bar can exist without a Foo. Changing this is just a matter of specifying `allowNull: false` in the foreign key options:

```
Foo.hasOne(Bar, {
  foreignKey: {
    allowNull: false
```

```
  }
});
// "fooId" INTEGER NOT NULL REFERENCES "foos" ("id") ON DELETE RESTRICT ON UPDATE RESTRICT
```

One-To-Many relationships

Philosophy

One-To-Many associations are connecting one source with multiple targets, while all these targets are connected only with this single source.

This means that, unlike the One-To-One association, in which we had to choose where the foreign key would be placed, there is only one option in One-To-Many associations. For example, if one Foo has many Bars (and this way each Bar belongs to one Foo), then the only sensible implementation is to have a `fooId` column in the `Bar` table. The opposite is impossible, since one Foo has many Bars.

Goal

In this example, we have the models `Team` and `Player`. We want to tell Sequelize that there is a One-To-Many relationship between them, meaning that one Team has many Players, while each Player belongs to a single Team.

Implementation

The main way to do this is as follows:

```
Team.hasMany(Player);
Player.belongsTo(Team);
```

Again, as mentioned, the main way to do it used a pair of Sequelize associations (`hasMany` and `belongsTo`).

For example, in PostgreSQL, the above setup will yield the following SQL upon `sync()`:

```
CREATE TABLE IF NOT EXISTS "Teams" (
  /* ... */
);
CREATE TABLE IF NOT EXISTS "Players" (
  /* ... */
  "TeamId" INTEGER REFERENCES "Teams" ("id") ON DELETE SET NULL ON UPDATE CASCADE,
  /* ... */
);
```

Options

The options to be applied in this case are the same from the One-To-One case. For example, to change the name of the foreign key and make sure that the relationship is mandatory, we can do:

```
Team.hasMany(Player, {
  foreignKey: 'clubId'
});
Player.belongsTo(Team);
```

Like One-To-One relationships, `ON DELETE` defaults to `SET NULL` and `ON UPDATE` defaults to `CASCADE`.

Many-To-Many relationships

Philosophy

Many-To-Many associations connect one source with multiple targets, while all these targets can in turn be connected to other sources beyond the first.

This cannot be represented by adding one foreign key to one of the tables, like the other relationships did. Instead, the concept of a Junction Model is used. This will be an extra model (and extra table in the database) which will have two foreign key columns and will keep track of the associations. The junction table is also sometimes called *join table* or *through table*.

Goal

For this example, we will consider the models `Movie` and `Actor`. One actor may have participated in many movies, and one movie had many actors involved with its production. The junction table that will keep track of the associations will be called `ActorMovies`, which will contain the foreign keys `movieId` and `actorId`.

Implementation

The main way to do this in Sequelize is as follows:

```
const Movie = sequelize.define('Movie', { name: DataTypes.STRING });
const Actor = sequelize.define('Actor', { name: DataTypes.STRING });
Movie.belongsToMany(Actor, { through: 'ActorMovies' });
Actor.belongsToMany(Movie, { through: 'ActorMovies' });
```

Since a string was given in the `through` option of the `belongsToMany` call, Sequelize will automatically create the `ActorMovies` model which will act as the junction model. For example, in PostgreSQL:

```
CREATE TABLE IF NOT EXISTS "ActorMovies" (
  "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL,
  "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL,
  "MovieId" INTEGER REFERENCES "Movies" ("id") ON DELETE CASCADE ON UPDATE CASCADE,
  "ActorId" INTEGER REFERENCES "Actors" ("id") ON DELETE CASCADE ON UPDATE CASCADE,
  PRIMARY KEY ("MovieId","ActorId")
);
```

Instead of a string, passing a model directly is also supported, and in that case the given model will be used as the junction model (and no model will be created automatically). For example:

```
const Movie = sequelize.define('Movie', { name: DataTypes.STRING });
```

```
const Actor = sequelize.define('Actor', { name: DataTypes.STRING });
const ActorMovies = sequelize.define('ActorMovies', {
  MovieId: {
    type: DataTypes.INTEGER,
    references: {
      model: Movie, // 'Movies' would also work
      key: 'id'
    }
  },
  ActorId: {
    type: DataTypes.INTEGER,
    references: {
      model: Actor, // 'Actors' would also work
      key: 'id'
    }
  }
});
Movie.belongsToMany(Actor, { through: ActorMovies });
Actor.belongsToMany(Movie, { through: ActorMovies });
```

The above yields the following SQL in PostgreSQL, which is equivalent to the one shown above:

```
CREATE TABLE IF NOT EXISTS "ActorMovies" (
  "MovieId" INTEGER NOT NULL REFERENCES "Movies" ("id") ON DELETE RESTRICT ON UPDATE CASCADE,
  "ActorId" INTEGER NOT NULL REFERENCES "Actors" ("id") ON DELETE RESTRICT ON UPDATE CASCADE,
  "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL,
  "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL,
  UNIQUE ("MovieId", "ActorId"),     -- Note: Sequelize generated this UNIQUE constraint but
  PRIMARY KEY ("MovieId","ActorId")  -- it is irrelevant since it's also a PRIMARY KEY
);
```

Options

Unlike One-To-One and One-To-Many relationships, the defaults for both `ON UPDATE` and `ON DELETE` are `CASCADE` for Many-To-Many relationships.

Belongs-To-Many creates a unique key on through model. This unique key name can be overridden using **uniqueKey** option. To prevent creating this unique key, use the ***unique: false*** option.

```
Project.belongsToMany(User, { through: UserProjects, uniqueKey: 'my_custom_unique' })
```

Basics of queries involving associations

With the basics of defining associations covered, we can look at queries involving associations. The most common queries on this matter are the *read* queries (i.e. SELECTs). Later on, other types of queries will be shown.

In order to study this, we will consider an example in which we have Ships and Captains, and a one-to-one relationship between them. We will allow null on foreign keys (the default), meaning that a Ship can exist without a Captain and vice-versa.

```
// This is the setup of our models for the examples below
const Ship = sequelize.define('ship', {
  name: DataTypes.TEXT,
  crewCapacity: DataTypes.INTEGER,
  amountOfSails: DataTypes.INTEGER
}, { timestamps: false });
const Captain = sequelize.define('captain', {
  name: DataTypes.TEXT,
  skillLevel: {
    type: DataTypes.INTEGER,
    validate: { min: 1, max: 10 }
  }
}, { timestamps: false });
Captain.hasOne(Ship);
Ship.belongsTo(Captain);
```

Fetching associations - Eager Loading vs Lazy Loading

The concepts of Eager Loading and Lazy Loading are fundamental to understand how fetching associations work in Sequelize. Lazy Loading refers to the technique of fetching the associated data only when you really want it; Eager Loading, on the other hand, refers to the technique of fetching everything at once, since the beginning, with a larger query.

Lazy Loading example

```
const awesomeCaptain = await Captain.findOne({
  where: {
    name: "Jack Sparrow"
  }
});
// Do stuff with the fetched captain
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);
// Now we want information about his ship!
const hisShip = await awesomeCaptain.getShip();
// Do stuff with the ship
console.log('Ship Name:', hisShip.name);
console.log('Amount of Sails:', hisShip.amountOfSails);
```

Observe that in the example above, we made two queries, only fetching the associated ship when we wanted to use it. This can be especially useful if we may or may not need the ship, perhaps we want to fetch it conditionally, only in a few cases; this way we can save time and memory by only fetching it when necessary.

Note: the `getShip()` instance method used above is one of the methods Sequelize automatically adds to `Captain` instances. There are others. You will learn more about them later in this guide.

Eager Loading Example

```
const awesomeCaptain = await Captain.findOne({
  where: {
    name: "Jack Sparrow"
  },
  include: Ship
});
```

```
// Now the ship comes with it
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);
console.log('Ship Name:', awesomeCaptain.ship.name);
console.log('Amount of Sails:', awesomeCaptain.ship.amountOfSails);
```

As shown above, Eager Loading is performed in Sequelize by using the `include` option. Observe that here only one query was performed to the database (which brings the associated data along with the instance).

This was just a quick introduction to Eager Loading in Sequelize. There is a lot more to it, which you can learn at the dedicated guide on Eager Loading.

Creating, updating and deleting

The above showed the basics on queries for fetching data involving associations. For creating, updating and deleting, you can either:

- Use the standard model queries directly:

```
// Example: creating an associated model using the standard methods
Bar.create({
  name: 'My Bar',
  fooId: 5
});
// This creates a Bar belonging to the Foo of ID 5 (since fooId is
// a regular column, after all). Nothing very clever going on here.
```

- Or use the *special methods/mixins* available for associated models, which are explained later on this page.

Note: The `save()` instance method is not aware of associations. In other words, if you change a value from a *child* object that was eager loaded along a *parent* object, calling `save()` on the parent will completely ignore the change that happened on the child.

Association Aliases & Custom Foreign Keys

In all the above examples, Sequelize automatically defined the foreign key names. For example, in the Ship and Captain example, Sequelize automatically defined a `captainId` field on the Ship model. However, it is easy to specify a custom foreign key.

Let's consider the models Ship and Captain in a simplified form, just to focus on the current topic, as shown below (less fields):

```
const Ship = sequelize.define('ship', { name: DataTypes.TEXT }, { timestamps: false });
const Captain = sequelize.define('captain', { name: DataTypes.TEXT }, { timestamps: false });
```

There are three ways to specify a different name for the foreign key:

- By providing the foreign key name directly
- By defining an Alias
- By doing both things

Recap: the default setup

By using simply `Ship.belongsTo(Captain)`, sequelize will generate the foreign key name automatically:

```
Ship.belongsTo(Captain); // This creates the `captainId` foreign key in Ship.

// Eager Loading is done by passing the model to `include`:
console.log((await Ship.findAll({ include: Captain })).toJSON());
// Or by providing the associated model name:
console.log((await Ship.findAll({ include: 'captain' })).toJSON());

// Also, instances obtain a `getCaptain()` method for Lazy Loading:
const ship = Ship.findOne();
console.log((await ship.getCaptain()).toJSON());
```

Providing the foreign key name directly

The foreign key name can be provided directly with an option in the association definition, as follows:

```
Ship.belongsTo(Captain, { foreignKey: 'bossId' }); // This creates the `bossId` foreign key in Ship.

// Eager Loading is done by passing the model to `include`:
console.log((await Ship.findAll({ include: Captain })).toJSON());
// Or by providing the associated model name:
console.log((await Ship.findAll({ include: 'Captain' })).toJSON());

// Also, instances obtain a `getCaptain()` method for Lazy Loading:
const ship = Ship.findOne();
console.log((await ship.getCaptain()).toJSON());
```

Defining an Alias

Defining an Alias is more powerful than simply specifying a custom name for the foreign key. This is better understood with an example:

```
Ship.belongsTo(Captain, { as: 'leader' }); // This creates the `leaderId` foreign key in Ship.

// Eager Loading no longer works by passing the model to `include`:
console.log((await Ship.findAll({ include: Captain })).toJSON()); // Throws an error
// Instead, you have to pass the alias:
console.log((await Ship.findAll({ include: 'leader' })).toJSON());
// Or you can pass an object specifying the model and alias:
console.log((await Ship.findAll({
  include: {
    model: Captain,
    as: 'leader'
  }
})).toJSON());

// Also, instances obtain a `getLeader()` method for Lazy Loading:
const ship = Ship.findOne();
console.log((await ship.getLeader()).toJSON());
```

Aliases are especially useful when you need to define two different associations between the same models. For example, if we have the

models `Mail` and `Person`, we may want to associate them twice, to represent the `sender` and `receiver` of the Mail. In this case we must use an alias for each association, since otherwise a call like `mail.getPerson()` would be ambiguous. With the `sender` and `receiver` aliases, we would have the two methods available and working: `mail.getSender()` and `mail.getReceiver()`, both of them returning a `Promise<Person>`.

When defining an alias for a `hasOne` or `belongsTo` association, you should use the singular form of a word (such as `leader`, in the example above). On the other hand, when defining an alias for `hasMany` and `belongsToMany`, you should use the plural form. Defining aliases for Many-to-Many relationships (with `belongsToMany`) is covered in the Advanced Many-to-Many Associations guide.

Doing both things

We can define and alias and also directly define the foreign key:

```
Ship.belongsTo(Captain, { as: 'leader', foreignKey: 'bossId' }); // This creates the `bossId` foreign key in Ship.

// Since an alias was defined, eager loading doesn't work by simply passing the model to `include`:
console.log((await Ship.findAll({ include: Captain })).toJSON()); // Throws an error
// Instead, you have to pass the alias:
console.log((await Ship.findAll({ include: 'leader' })).toJSON());
// Or you can pass an object specifying the model and alias:
console.log((await Ship.findAll({
  include: {
    model: Captain,
    as: 'leader'
  }
})).toJSON());

// Also, instances obtain a `getLeader()` method for lazy loading:
const ship = Ship.findOne();
console.log((await ship.getLeader()).toJSON());
```

Special methods/mixins added to instances

When an association is defined between two models, the instances of those models gain special methods to interact with their associated counterparts.

For example, if we have two models, `Foo` and `Bar`, and they are associated, their instances will have the following methods/mixins available, depending on the association type:

`Foo.hasOne(Bar)`

- `fooInstance.getBar()`
- `fooInstance.setBar()`
- `fooInstance.createBar()`

Example:

```
const foo = await Foo.create({ name: 'the-foo' });
const bar1 = await Bar.create({ name: 'some-bar' });
const bar2 = await Bar.create({ name: 'another-bar' });
console.log(await foo.getBar()); // null
await foo.setBar(bar1);
console.log((await foo.getBar()).name); // 'some-bar'
await foo.createBar({ name: 'yet-another-bar' });
const newlyAssociatedBar = await foo.getBar();
console.log(newlyAssociatedBar.name); // 'yet-another-bar'
await foo.setBar(null); // Un-associate
console.log(await foo.getBar()); // null
```

`Foo.belongsTo(Bar)`

The same ones from `Foo.hasOne(Bar)`:

- `fooInstance.getBar()`
- `fooInstance.setBar()`
- `fooInstance.createBar()`

`Foo.hasMany(Bar)`

- `fooInstance.getBars()`
- `fooInstance.countBars()`
- `fooInstance.hasBar()`
- `fooInstance.hasBars()`
- `fooInstance.setBars()`
- `fooInstance.addBar()`
- `fooInstance.addBars()`
- `fooInstance.removeBar()`
- `fooInstance.removeBars()`
- `fooInstance.createBar()`

Example:

```
const foo = await Foo.create({ name: 'the-foo' });
const bar1 = await Bar.create({ name: 'some-bar' });
const bar2 = await Bar.create({ name: 'another-bar' });
console.log(await foo.getBars()); // []
console.log(await foo.countBars()); // 0
console.log(await foo.hasBar(bar1)); // false
await foo.addBars([bar1, bar2]);
console.log(await foo.countBars()); // 2
await foo.addBar(bar1);
console.log(await foo.countBars()); // 2
console.log(await foo.hasBar(bar1)); // true
await foo.removeBar(bar2);
console.log(await foo.countBars()); // 1
await foo.createBar({ name: 'yet-another-bar' });
console.log(await foo.countBars()); // 2
```

```
await foo.setBars([]); // Un-associate all previously associated bars
console.log(await foo.countBars()); // 0
```

The getter method accepts options just like the usual finder methods (such as `findAll`):

```
const easyTasks = await project.getTasks({
  where: {
    difficulty: {
      [Op.lte]: 5
    }
  }
});
const taskTitles = (await project.getTasks({
  attributes: ['title'],
  raw: true
})).map(task => task.title);
```

`Foo.belongsToMany(Bar, { through: Baz })`

The same ones from `Foo.hasMany(Bar)`:

- `fooInstance.getBars()`
- `fooInstance.countBars()`
- `fooInstance.hasBar()`
- `fooInstance.hasBars()`
- `fooInstance.setBars()`
- `fooInstance.addBar()`
- `fooInstance.addBars()`
- `fooInstance.removeBar()`
- `fooInstance.removeBars()`
- `fooInstance.createBar()`

For belongsToMany relationships, by default `getBars()` will return all fields from the join table. Note that any `include` options will apply to the target `Bar` object, so trying to set options for the join table as you would when eager loading with `find` methods is not possible. To choose what attributes of the join table to include, `getBars()` supports a `joinTableAttributes` option that can be used similarly to setting `through.attributes` in an `include`. As an example, given Foo belongsToMany Bar, the following will both output results without join table fields:

```
const foo = Foo.findByPk(id, {
  include: [{
    model: Bar,
    through: { attributes: [] }
  }]
})
console.log(foo.bars)

const foo = Foo.findByPk(id)
console.log(foo.getBars({ joinTableAttributes: [] }))
```

Note: Method names

As shown in the examples above, the names Sequelize gives to these special methods are formed by a prefix (e.g. `get`, `add`, `set`) concatenated with the model name (with the first letter in uppercase). When necessary, the plural is used, such as in `fooInstance.setBars()`. Again, irregular plurals are also handled automatically by Sequelize. For example, `Person` becomes `People` and `Hypothesis` becomes `Hypotheses`.

If an alias was defined, it will be used instead of the model name to form the method names. For example:

```
Task.hasOne(User, { as: 'Author' });
```

- `taskInstance.getAuthor()`
- `taskInstance.setAuthor()`
- `taskInstance.createAuthor()`

Why associations are defined in pairs?

As mentioned earlier and shown in most examples above, usually associations in Sequelize are defined in pairs:

- To create a **One-To-One** relationship, the `hasOne` and `belongsTo` associations are used together;
- To create a **One-To-Many** relationship, the `hasMany` and `belongsTo` associations are used together;
- To create a **Many-To-Many** relationship, two `belongsToMany` calls are used together.

When a Sequelize association is defined between two models, only the *source* model *knows about it*. So, for example, when using `Foo.hasOne(Bar)` (so `Foo` is the source model and `Bar` is the target model), only `Foo` knows about the existence of this association. This is why in this case, as shown above, `Foo` instances gain the methods `getBar()`, `setBar()` and `createBar()`, while on the other hand `Bar` instances get nothing.

Similarly, for `Foo.hasOne(Bar)`, since `Foo` knows about the relationship, we can perform eager loading as in `Foo.findOne({ include: Bar })`, but we can't do `Bar.findOne({ include: Foo })`.

Therefore, to bring full power to Sequelize usage, we usually setup the relationship in pairs, so that both models get to *know about it*.

Practical demonstration:

- If we do not define the pair of associations, calling for example just `Foo.hasOne(Bar)`:

  ```
  // This works...
  await Foo.findOne({ include: Bar });

  // But this throws an error:
  await Bar.findOne({ include: Foo });
  // SequelizeEagerLoadingError: foo is not associated to bar!
  ```

- If we define the pair as recommended, i.e., both `Foo.hasOne(Bar)` and `Bar.belongsTo(Foo)`:

```
// This works!
await Foo.findOne({ include: Bar });

// This also works!
await Bar.findOne({ include: Foo });
```

Multiple associations involving the same models

In Sequelize, it is possible to define multiple associations between the same models. You just have to define different aliases for them:

```
Team.hasOne(Game, { as: 'HomeTeam', foreignKey: 'homeTeamId' });
Team.hasOne(Game, { as: 'AwayTeam', foreignKey: 'awayTeamId' });
Game.belongsTo(Team);
```

Creating associations referencing a field which is not the primary key

In all the examples above, the associations were defined by referencing the primary keys of the involved models (in our case, their IDs). However, Sequelize allows you to define an association that uses another field, instead of the primary key field, to establish the association.

This other field must have a unique constraint on it (otherwise, it wouldn't make sense).

For `belongsTo` relationships

First, recall that the `A.belongsTo(B)` association places the foreign key in the *source model* (i.e., in `A`).

Let's again use the example of Ships and Captains. Additionally, we will assume that Captain names are unique:

```
const Ship = sequelize.define('ship', { name: DataTypes.TEXT }, { timestamps: false });
const Captain = sequelize.define('captain', {
  name: { type: DataTypes.TEXT, unique: true }
}, { timestamps: false });
```

This way, instead of keeping the `captainId` on our Ships, we could keep a `captainName` instead and use it as our association tracker. In other words, instead of referencing the `id` from the target model (Captain), our relationship will reference another column on the target model: the `name` column. To specify this, we have to define a *target key*. We will also have to specify a name for the foreign key itself:

```
Ship.belongsTo(Captain, { targetKey: 'name', foreignKey: 'captainName' });
// This creates a foreign key called `captainName` in the source model (Ship)
// which references the `name` field from the target model (Captain).
```

Now we can do things like:

```
await Captain.create({ name: "Jack Sparrow" });
const ship = await Ship.create({ name: "Black Pearl", captainName: "Jack Sparrow" });
console.log((await ship.getCaptain()).name); // "Jack Sparrow"
```

For `hasOne` and `hasMany` relationships

The exact same idea can be applied to the `hasOne` and `hasMany` associations, but instead of providing a `targetKey`, we provide a `sourceKey` when defining the association. This is because unlike `belongsTo`, the `hasOne` and `hasMany` associations keep the foreign key on the target model:

```
const Foo = sequelize.define('foo', {
  name: { type: DataTypes.TEXT, unique: true }
}, { timestamps: false });
const Bar = sequelize.define('bar', {
  title: { type: DataTypes.TEXT, unique: true }
}, { timestamps: false });
const Baz = sequelize.define('baz', { summary: DataTypes.TEXT }, { timestamps: false });
Foo.hasOne(Bar, { sourceKey: 'name', foreignKey: 'fooName' });
Bar.hasMany(Baz, { sourceKey: 'title', foreignKey: 'barTitle' });
// [...]
await Bar.setFoo("Foo's Name Here");
await Baz.addBar("Bar's Title Here");
```

For `belongsToMany` relationships

The same idea can also be applied to `belongsToMany` relationships. However, unlike the other situations, in which we have only one foreign key involved, the `belongsToMany` relationship involves two foreign keys which are kept on an extra table (the junction table).

Consider the following setup:

```
const Foo = sequelize.define('foo', {
  name: { type: DataTypes.TEXT, unique: true }
}, { timestamps: false });
const Bar = sequelize.define('bar', {
  title: { type: DataTypes.TEXT, unique: true }
}, { timestamps: false });
```

There are four cases to consider:

- We might want a many-to-many relationship using the default primary keys for both `Foo` and `Bar`:

```
Foo.belongsToMany(Bar, { through: 'foo_bar' });
// This creates a junction table `foo_bar` with fields `fooId` and `barId`
```

- We might want a many-to-many relationship using the default primary key for `Foo` but a different field for `Bar`:

```
Foo.belongsToMany(Bar, { through: 'foo_bar', targetKey: 'title' });
// This creates a junction table `foo_bar` with fields `fooId` and `barTitle`
```

- We might want a many-to-many relationship using the a different field for `Foo` and the default primary key for `Bar`:

```
Foo.belongsToMany(Bar, { through: 'foo_bar', sourceKey: 'name' });
// This creates a junction table `foo_bar` with fields `fooName` and `barId`
```

- We might want a many-to-many relationship using different fields for both `Foo` and `Bar`:

```
Foo.belongsToMany(Bar, { through: 'foo_bar', sourceKey: 'name', targetKey: 'title' });
// This creates a junction table `foo_bar` with fields `fooName` and `barTitle`
```

Notes

Don't forget that the field referenced in the association must have a unique constraint placed on it. Otherwise, an error will be thrown (and sometimes with a mysterious error message - such as `SequelizeDatabaseError: SQLITE_ERROR: foreign key mismatch - "ships" referencing "captains"` for SQLite).

The trick to deciding between `sourceKey` and `targetKey` is just to remember where each relationship places its foreign key. As mentioned in the beginning of this guide:

- `A.belongsTo(B)` keeps the foreign key in the source model (`A`), therefore the referenced key is in the target model, hence the usage of `targetKey`.
- `A.hasOne(B)` and `A.hasMany(B)` keep the foreign key in the target model (`B`), therefore the referenced key is in the source model, hence the usage of `sourceKey`.
- `A.belongsToMany(B)` involves an extra table (the junction table), therefore both `sourceKey` and `targetKey` are usable, with `sourceKey` corresponding to some field in `A` (the source) and `targetKey` corresponding to some field in `B` (the target).

Edit this page

*Last updated on **Oct 13, 2022** by **Rik Smale***

Previous
« Raw Queries

Next
Paranoid »