Version: v6 - stable

Advanced M:N Associations

Make sure you have read the associations guide before reading this guide.

Let's start with an example of a Many-to-Many relationship between `User` and `Profile`.

```
const User = sequelize.define('user', {
  username: DataTypes.STRING,
  points: DataTypes.INTEGER
}, { timestamps: false });
const Profile = sequelize.define('profile', {
  name: DataTypes.STRING
}, { timestamps: false });
```

The simplest way to define the Many-to-Many relationship is:

```
User.belongsToMany(Profile, { through: 'User_Profiles' });
Profile.belongsToMany(User, { through: 'User_Profiles' });
```

By passing a string to `through` above, we are asking Sequelize to automatically generate a model named `User_Profiles` as the *through table* (also known as junction table), with only two columns: `userId` and `profileId`. A composite unique key will be established on these two columns.

We can also define ourselves a model to be used as the through table.

```
const User_Profile = sequelize.define('User_Profile', {}, { timestamps: false });
User.belongsToMany(Profile, { through: User_Profile });
Profile.belongsToMany(User, { through: User_Profile });
```

The above has the exact same effect. Note that we didn't define any attributes on the `User_Profile` model. The fact that we passed it into a `belongsToMany` call tells sequelize to create the two attributes `userId` and `profileId` automatically, just like other associations also cause Sequelize to automatically add a column to one of the involved models.

However, defining the model by ourselves has several advantages. We can, for example, define more columns on our through table:

```
const User_Profile = sequelize.define('User_Profile', {
  selfGranted: DataTypes.BOOLEAN
}, { timestamps: false });
User.belongsToMany(Profile, { through: User_Profile });
Profile.belongsToMany(User, { through: User_Profile });
```

With this, we can now track an extra information at the through table, namely the `selfGranted` boolean. For example, when calling the `user.addProfile()` we can pass values for the extra columns using the `through` option.

Example:

```
const amidala = await User.create({ username: 'p4dm3', points: 1000 });
const queen = await Profile.create({ name: 'Queen' });
await amidala.addProfile(queen, { through: { selfGranted: false } });
const result = await User.findOne({
  where: { username: 'p4dm3' },
  include: Profile
});
console.log(result);
```

Output:

```
{
  "id": 4,
  "username": "p4dm3",
  "points": 1000,
  "profiles": [
    {
      "id": 6,
      "name": "queen",
      "User_Profile": {
        "userId": 4,
        "profileId": 6,
        "selfGranted": false
      }
    }
  ]
}
```

You can create all relationship in single `create` call too.

Example:

```
const amidala = await User.create({
  username: 'p4dm3',
  points: 1000,
  profiles: [{
    name: 'Queen',
    User_Profile: {
      selfGranted: true
    }
  }]
}, {
  include: Profile
});

const result = await User.findOne({
  where: { username: 'p4dm3' },
  include: Profile
});
```

```
});

console.log(result);
```

Output:

```
{
  "id": 1,
  "username": "p4dm3",
  "points": 1000,
  "profiles": [
    {
      "id": 1,
      "name": "Queen",
      "User_Profile": {
        "selfGranted": true,
        "userId": 1,
        "profileId": 1
      }
    }
  ]
}
```

You probably noticed that the `User_Profiles` table does not have an `id` field. As mentioned above, it has a composite unique key instead. The name of this composite unique key is chosen automatically by Sequelize but can be customized with the `uniqueKey` option:

```
User.belongsToMany(Profile, { through: User_Profiles, uniqueKey: 'my_custom_unique' });
```

Another possibility, if desired, is to force the through table to have a primary key just like other standard tables. To do this, simply define the primary key in the model:

```
const User_Profile = sequelize.define('User_Profile', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
    allowNull: false
  },
  selfGranted: DataTypes.BOOLEAN
}, { timestamps: false });
User.belongsToMany(Profile, { through: User_Profile });
Profile.belongsToMany(User, { through: User_Profile });
```

The above will still create two columns `userId` and `profileId`, of course, but instead of setting up a composite unique key on them, the model will use its `id` column as primary key. Everything else will still work just fine.

Through tables versus normal tables and the "Super Many-to-Many association"

Now we will compare the usage of the last Many-to-Many setup shown above with the usual One-to-Many relationships, so that in the end we conclude with the concept of a "*Super Many-to-Many relationship*".

Models recap (with minor rename)

To make things easier to follow, let's rename our `User_Profile` model to `grant`. Note that everything works in the same way as before. Our models are:

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

const Profile = sequelize.define('profile', {
  name: DataTypes.STRING
}, { timestamps: false });

const Grant = sequelize.define('grant', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
    allowNull: false
  },
  selfGranted: DataTypes.BOOLEAN
}, { timestamps: false });
```

We established a Many-to-Many relationship between `User` and `Profile` using the `Grant` model as the through table:

```
User.belongsToMany(Profile, { through: Grant });
Profile.belongsToMany(User, { through: Grant });
```

This automatically added the columns `userId` and `profileId` to the `Grant` model.

Note: As shown above, we have chosen to force the `grant` model to have a single primary key (called `id`, as usual). This is necessary for the *Super Many-to-Many relationship* that will be defined soon.

Using One-to-Many relationships instead

Instead of setting up the Many-to-Many relationship defined above, what if we did the following instead?

```
// Setup a One-to-Many relationship between User and Grant
User.hasMany(Grant);
Grant.belongsTo(User);

// Also setup a One-to-Many relationship between Profile and Grant
Profile.hasMany(Grant);
Grant.belongsTo(Profile);
```

The result is essentially the same! This is because `User.hasMany(Grant)` and `Profile.hasMany(Grant)` will automatically add the

`userId` and `profileId` columns to `Grant`, respectively.

This shows that one Many-to-Many relationship isn't very different from two One-to-Many relationships. The tables in the database look the same.

The only difference is when you try to perform an eager load with Sequelize.

```
// With the Many-to-Many approach, you can do:
User.findAll({ include: Profile });
Profile.findAll({ include: User });
// However, you can't do:
User.findAll({ include: Grant });
Profile.findAll({ include: Grant });
Grant.findAll({ include: User });
Grant.findAll({ include: Profile });

// On the other hand, with the double One-to-Many approach, you can do:
User.findAll({ include: Grant });
Profile.findAll({ include: Grant });
Grant.findAll({ include: User });
Grant.findAll({ include: Profile });
// However, you can't do:
User.findAll({ include: Profile });
Profile.findAll({ include: User });
// Although you can emulate those with nested includes, as follows:
User.findAll({
  include: {
    model: Grant,
    include: Profile
  }
}); // This emulates the `User.findAll({ include: Profile })`, however
    // the resulting object structure is a bit different. The original
    // structure has the form `user.profiles[].grant`, while the emulated
    // structure has the form `user.grants[].profiles[]`.
```

The best of both worlds: the Super Many-to-Many relationship

We can simply combine both approaches shown above!

```
// The Super Many-to-Many relationship
User.belongsToMany(Profile, { through: Grant });
Profile.belongsToMany(User, { through: Grant });
User.hasMany(Grant);
Grant.belongsTo(User);
Profile.hasMany(Grant);
Grant.belongsTo(Profile);
```

This way, we can do all kinds of eager loading:

```
// All these work:
User.findAll({ include: Profile });
Profile.findAll({ include: User });
User.findAll({ include: Grant });
Profile.findAll({ include: Grant });
Grant.findAll({ include: User });
Grant.findAll({ include: Profile });
```

We can even perform all kinds of deeply nested includes:

```
User.findAll({
  include: [
    {
      model: Grant,
      include: [User, Profile]
    },
    {
      model: Profile,
      include: {
        model: User,
        include: {
          model: Grant,
          include: [User, Profile]
        }
      }
    }
  ]
});
```

Aliases and custom key names

Similarly to the other relationships, aliases can be defined for Many-to-Many relationships.

Before proceeding, please recall the aliasing example for `belongsTo` on the associations guide. Note that, in that case, defining an association impacts both the way includes are done (i.e. passing the association name) and the name Sequelize chooses for the foreign key (in that example, `leaderId` was created on the `Ship` model).

Defining an alias for a `belongsToMany` association also impacts the way includes are performed:

```
Product.belongsToMany(Category, { as: 'groups', through: 'product_categories' });
Category.belongsToMany(Product, { as: 'items', through: 'product_categories' });

// [...]

await Product.findAll({ include: Category }); // This doesn't work

await Product.findAll({ // This works, passing the alias
  include: {
    model: Category,
    as: 'groups'
  }
});

await Product.findAll({ include: 'groups' }); // This also works
```

However, defining an alias here has nothing to do with the foreign key names. The names of both foreign keys created in the through table are still constructed by Sequelize based on the name of the models being associated. This can readily be seen by inspecting the generated SQL for the through table in the example above:

```
CREATE TABLE IF NOT EXISTS `product_categories` (
  `createdAt` DATETIME NOT NULL,
  `updatedAt` DATETIME NOT NULL,
  `productId` INTEGER NOT NULL REFERENCES `products` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  `categoryId` INTEGER NOT NULL REFERENCES `categories` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  PRIMARY KEY (`productId`, `categoryId`)
);
```

We can see that the foreign keys are `productId` and `categoryId`. To change these names, Sequelize accepts the options `foreignKey` and `otherKey` respectively (i.e., the `foreignKey` defines the key for the source model in the through relation, and `otherKey` defines it for the target model):

```
Product.belongsToMany(Category, {
  through: 'product_categories',
  foreignKey: 'objectId', // replaces `productId`
  otherKey: 'typeId' // replaces `categoryId`
});
Category.belongsToMany(Product, {
  through: 'product_categories',
  foreignKey: 'typeId', // replaces `categoryId`
  otherKey: 'objectId' // replaces `productId`
});
```

Generated SQL:

```
CREATE TABLE IF NOT EXISTS `product_categories` (
  `createdAt` DATETIME NOT NULL,
  `updatedAt` DATETIME NOT NULL,
  `objectId` INTEGER NOT NULL REFERENCES `products` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  `typeId` INTEGER NOT NULL REFERENCES `categories` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  PRIMARY KEY (`objectId`, `typeId`)
);
```

As shown above, when you define a Many-to-Many relationship with two `belongsToMany` calls (which is the standard way), you should provide the `foreignKey` and `otherKey` options appropriately in both calls. If you pass these options in only one of the calls, the Sequelize behavior will be unreliable.

Self-references

Sequelize supports self-referential Many-to-Many relationships, intuitively:

```
Person.belongsToMany(Person, { as: 'Children', through: 'PersonChildren' })
// This will create the table PersonChildren which stores the ids of the objects.
```

Specifying attributes from the through table

By default, when eager loading a many-to-many relationship, Sequelize will return data in the following structure (based on the first example in this guide):

```
// User.findOne({ include: Profile })
{
  "id": 4,
  "username": "p4dm3",
  "points": 1000,
  "profiles": [
    {
      "id": 6,
      "name": "queen",
      "grant": {
        "userId": 4,
        "profileId": 6,
        "selfGranted": false
      }
    }
  ]
}
```

Notice that the outer object is an `User`, which has a field called `profiles`, which is a `Profile` array, such that each `Profile` comes with an extra field called `grant` which is a `Grant` instance. This is the default structure created by Sequelize when eager loading from a Many-to-Many relationship.

However, if you want only some of the attributes of the through table, you can provide an array with the attributes you want in the `attributes` option. For example, if you only want the `selfGranted` attribute from the through table:

```
User.findOne({
  include: {
    model: Profile,
    through: {
      attributes: ['selfGranted']
    }
  }
});
```

Output:

```
{
  "id": 4,
  "username": "p4dm3",
  "points": 1000,
  "profiles": [
    {
      "id": 6,
      "name": "queen",
      "grant": {
        "selfGranted": false
```

```
      }
    }
  ]
}
```

If you don't want the nested `grant` field at all, use `attributes: []`:

```
User.findOne({
  include: {
    model: Profile,
    through: {
      attributes: []
    }
  }
});
```

Output:

```
{
  "id": 4,
  "username": "p4dm3",
  "points": 1000,
  "profiles": [
    {
      "id": 6,
      "name": "queen"
    }
  ]
}
```

If you are using mixins (such as `user.getProfiles()`) instead of finder methods (such as `User.findAll()`), you have to use the `joinTableAttributes` option instead:

```
someUser.getProfiles({ joinTableAttributes: ['selfGranted'] });
```

Output:

```
[
  {
    "id": 6,
    "name": "queen",
    "grant": {
      "selfGranted": false
    }
  }
]
```

Many-to-many-to-many relationships and beyond

Consider you are trying to model a game championship. There are players and teams. Teams play games. However, players can change teams in the middle of the championship (but not in the middle of a game). So, given one specific game, there are certain teams participating in that game, and each of these teams has a set of players (for that game).

So we start by defining the three relevant models:

```
const Player = sequelize.define('Player', { username: DataTypes.STRING });
const Team = sequelize.define('Team', { name: DataTypes.STRING });
const Game = sequelize.define('Game', { name: DataTypes.INTEGER });
```

Now, the question is: how to associate them?

First, we note that:

- One game has many teams associated to it (the ones that are playing that game);
- One team may have participated in many games.

The above observations show that we need a Many-to-Many relationship between Game and Team. Let's use the Super Many-to-Many relationship as explained earlier in this guide:

```
// Super Many-to-Many relationship between Game and Team
const GameTeam = sequelize.define('GameTeam', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
    allowNull: false
  }
});
Team.belongsToMany(Game, { through: GameTeam });
Game.belongsToMany(Team, { through: GameTeam });
GameTeam.belongsTo(Game);
GameTeam.belongsTo(Team);
Game.hasMany(GameTeam);
Team.hasMany(GameTeam);
```

The part about players is trickier. We note that the set of players that form a team depends not only on the team (obviously), but also on which game is being considered. Therefore, we don't want a Many-to-Many relationship between Player and Team. We also don't want a Many-to-Many relationship between Player and Game. Instead of associating a Player to any of those models, what we need is an association between a Player and something like a *"team-game pair constraint"*, since it is the pair (team plus game) that defines which players belong there. So what we are looking for turns out to be precisely the junction model, GameTeam, itself! And, we note that, since a given *game-team pair* specifies many players, and on the other hand that the same player can participate of many *game-team pairs*, we need a Many-to-Many relationship between Player and GameTeam!

To provide the greatest flexibility, let's use the Super Many-to-Many relationship construction here again:

```
// Super Many-to-Many relationship between Player and GameTeam
const PlayerGameTeam = sequelize.define('PlayerGameTeam', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
```

```
    autoIncrement: true,
    allowNull: false
  }
});
Player.belongsToMany(GameTeam, { through: PlayerGameTeam });
GameTeam.belongsToMany(Player, { through: PlayerGameTeam });
PlayerGameTeam.belongsTo(Player);
PlayerGameTeam.belongsTo(GameTeam);
Player.hasMany(PlayerGameTeam);
GameTeam.hasMany(PlayerGameTeam);
```

The above associations achieve precisely what we want. Here is a full runnable example of this:

```
const { Sequelize, Op, Model, DataTypes } = require('sequelize');
const sequelize = new Sequelize('sqlite::memory:', {
  define: { timestamps: false } // Just for less clutter in this example
});
const Player = sequelize.define('Player', { username: DataTypes.STRING });
const Team = sequelize.define('Team', { name: DataTypes.STRING });
const Game = sequelize.define('Game', { name: DataTypes.INTEGER });

// We apply a Super Many-to-Many relationship between Game and Team
const GameTeam = sequelize.define('GameTeam', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
    allowNull: false
  }
});
Team.belongsToMany(Game, { through: GameTeam });
Game.belongsToMany(Team, { through: GameTeam });
GameTeam.belongsTo(Game);
GameTeam.belongsTo(Team);
Game.hasMany(GameTeam);
Team.hasMany(GameTeam);

// We apply a Super Many-to-Many relationship between Player and GameTeam
const PlayerGameTeam = sequelize.define('PlayerGameTeam', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
    allowNull: false
  }
});
Player.belongsToMany(GameTeam, { through: PlayerGameTeam });
GameTeam.belongsToMany(Player, { through: PlayerGameTeam });
PlayerGameTeam.belongsTo(Player);
PlayerGameTeam.belongsTo(GameTeam);
Player.hasMany(PlayerGameTeam);
GameTeam.hasMany(PlayerGameTeam);

(async () => {

  await sequelize.sync();
  await Player.bulkCreate([
    { username: 's0me0ne' },
    { username: 'empty' },
    { username: 'greenhead' },
    { username: 'not_spock' },
    { username: 'bowl_of_petunias' }
  ]);
  await Game.bulkCreate([
    { name: 'The Big Clash' },
    { name: 'Winter Showdown' },
    { name: 'Summer Beatdown' }
  ]);
  await Team.bulkCreate([
    { name: 'The Martians' },
    { name: 'The Earthlings' },
    { name: 'The Plutonians' }
  ]);

  // Let's start defining which teams were in which games. This can be done
  // in several ways, such as calling `.setTeams` on each game. However, for
  // brevity, we will use direct `create` calls instead, referring directly
  // to the IDs we want. We know that IDs are given in order starting from 1.
  await GameTeam.bulkCreate([
    { GameId: 1, TeamId: 1 },   // this GameTeam will get id 1
    { GameId: 1, TeamId: 2 },   // this GameTeam will get id 2
    { GameId: 2, TeamId: 1 },   // this GameTeam will get id 3
    { GameId: 2, TeamId: 3 },   // this GameTeam will get id 4
    { GameId: 3, TeamId: 2 },   // this GameTeam will get id 5
    { GameId: 3, TeamId: 3 }    // this GameTeam will get id 6
  ]);

  // Now let's specify players.
  // For brevity, let's do it only for the second game (Winter Showdown).
  // Let's say that that s0me0ne and greenhead played for The Martians, while
  // not_spock and bowl_of_petunias played for The Plutonians:
  await PlayerGameTeam.bulkCreate([
    // In 'Winter Showdown' (i.e. GameTeamIds 3 and 4):
    { PlayerId: 1, GameTeamId: 3 },   // s0me0ne played for The Martians
    { PlayerId: 3, GameTeamId: 3 },   // greenhead played for The Martians
    { PlayerId: 4, GameTeamId: 4 },   // not_spock played for The Plutonians
    { PlayerId: 5, GameTeamId: 4 }    // bowl_of_petunias played for The Plutonians
  ]);

  // Now we can make queries!
  const game = await Game.findOne({
    where: {
      name: "Winter Showdown"
    },
    include: {
      model: GameTeam,
      include: [
        {
          model: Player,
          through: { attributes: [] } // Hide unwanted `PlayerGameTeam` nested object from results
        },
        Team
      ]
```

```
    }
  });

  console.log(`Found game: "${game.name}"`);
  for (let i = 0; i < game.GameTeams.length; i++) {
    const team = game.GameTeams[i].Team;
    const players = game.GameTeams[i].Players;
    console.log(`- Team "${team.name}" played game "${game.name}" with the following players:`);
    console.log(players.map(p => `--- ${p.username}`).join('\n'));
  }

})();
```

Output:

```
Found game: "Winter Showdown"
- Team "The Martians" played game "Winter Showdown" with the following players:
--- s0me0ne
--- greenhead
- Team "The Plutonians" played game "Winter Showdown" with the following players:
--- not_spock
--- bowl_of_petunias
```

So this is how we can achieve a *many-to-many-to-many* relationship between three models in Sequelize, by taking advantage of the Super Many-to-Many relationship technique!

This idea can be applied recursively for even more complex, *many-to-many-to-...-to-many* relationships (although at some point queries might become slow).

Edit this page

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

Previous
« Creating with Associations

Next
Association Scopes »