Associations

This section describes the various association types in sequelize. When calling a method such as `User.hasOne(Project)`, we say that the `User` model (the model that the function is being invoked on) is the **source** and the `Project` model (the model being passed as an argument) is the **target**.

One-To-One associations

One-To-One associations are associations between exactly two models connected by a single foreign key.

BelongsTo

BelongsTo associations are associations where the foreign key for the one-to-one relation exists on the **source model**.

A simple example would be a **Player** being part of a **Team** with the foreign key on the player.

```
const Player = this.sequelize.define('player', {/* attributes */});
const Team  = this.sequelize.define('team', {/* attributes */});

Player.belongsTo(Team); // Will add a teamId attribute to Player to hold the primary key value for Team
```

Foreign keys

By default the foreign key for a belongsTo relation will be generated from the target model name and the target primary key name.

The default casing is `camelCase` however if the source model is configured with `underscored: true` the foreignKey will be `snake_case`.

```
const User = this.sequelize.define('user', {/*
attributes */})
```

```
*/});

User.belongsTo(Company); // Will add companyId
to user

const User = this.sequelize.define('user', {/*
attributes */}, {underscored: true})
const Company  =
this.sequelize.define('company', {
  uuid: {
    type: Sequelize.UUID,
    primaryKey: true
  }
});

User.belongsTo(Company); // Will add
company_uuid to user
```

In cases where `as` has been defined it will be used in place of the target model name.

```
const User = this.sequelize.define('user', {/*
attributes */})
const UserRole  =
this.sequelize.define('userRole', {/*
attributes */});

User.belongsTo(UserRole, {as: 'role'}); // Adds
roleId to user rather than userRoleId
```

In all cases the default foreign key can be overwritten with the `foreignKey` option. When the foreign key option is used, Sequelize will use it as-is:

```
const User = this.sequelize.define('user', {/*
attributes */})
const Company  =
this.sequelize.define('company', {/* attributes
*/});

User.belongsTo(Company, {foreignKey:
'fk_company'}); // Adds fk_company to User
```

Target keys

The target key is the column on the target model that the foreign key column on the source model points to. By default the target key for a belongsTo relation will be the target model's primary key. To define a custom column, use the `targetKey` option.

```
const User = this.sequelize.define('user', {/*
attributes */})
const Company  =
this.sequelize.define('company', {/* attributes
*/});
```

```
fk_companyname to User
```

HasOne

HasOne associations are associations where the foreign key for the one-to-one relation exists on the **target model**.

```
const User = sequelize.define('user', {/* ...
*/})
const Project = sequelize.define('project', {/*
... */})

// One-way associations
Project.hasOne(User)

/*
  In this example hasOne will add an attribute
projectId to the User model!
  Furthermore, Project.prototype will gain the
methods getUser and setUser according
  to the first parameter passed to define. If
you have underscore style
  enabled, the added attribute will be
project_id instead of projectId.

  The foreign key will be placed on the users
table.

  You can also define the foreign key, e.g. if
you already have an existing
  database and want to work on it:
*/

Project.hasOne(User, { foreignKey:
'initiator_id' })

/*
  Because Sequelize will use the model's name
(first parameter of define) for
  the accessor methods, it is also possible to
pass a special option to hasOne:
*/

Project.hasOne(User, { as: 'Initiator' })
// Now you will get Project.getInitiator and
Project.setInitiator

// Or let's define some self references
const Person = sequelize.define('person', { /*
... */})

Person.hasOne(Person, {as: 'Father'})
// this will add the attribute FatherId to
Person

// also possible:
Person.hasOne(Person, {as: 'Father',
foreignKey: 'DadId'})
// this will add the attribute DadId to Person
```

```
Person.setFather
Person.getFather

// If you need to join a table twice you can
double join the same table
Team.hasOne(Game, {as: 'HomeTeam', foreignKey :
'homeTeamId'});
Team.hasOne(Game, {as: 'AwayTeam', foreignKey :
'awayTeamId'});

Game.belongsTo(Team);
```

Even though it is called a HasOne association, for most 1:1 relations you usually want the BelongsTo association since BelongsTo will add the foreignKey on the source where hasOne will add on the target.

Difference between HasOne and BelongsTo

In Sequelize 1:1 relationship can be set using HasOne and BelongsTo. They are suitable for different scenarios. Lets study this difference using an example.

Suppose we have two tables to link **Player** and **Team**. Lets define their models.

```
const Player = this.sequelize.define('player',
{/* attributes */})
const Team  = this.sequelize.define('team', {/*
attributes */});
```

When we link two models in Sequelize we can refer them as pairs of **source** and **target** models. Like this

Having **Player** as the **source** and **Team** as the **target**

```
Player.belongsTo(Team);
//Or
Player.hasOne(Team);
```

Having **Team** as the **source** and **Player** as the **target**

```
Team.belongsTo(Player);
//Or
Team.hasOne(Player);
```

HasOne and BelongsTo insert the association key in different models from each other. HasOne inserts the association key in **target** model whereas BelongsTo inserts the association key in the **source** model.

```
const Player = this.sequelize.define('player',
{/* attributes */})
const Coach  = this.sequelize.define('coach',
{/* attributes */})
const Team  = this.sequelize.define('team', {/*
attributes */});
```

Suppose our `Player` model has information about its team as `teamId` column. Information about each Team's `Coach` is stored in the `Team` model as `coachId` column. These both scenarios requires different kind of 1:1 relation because foreign key relation is present on different models each time.

When information about association is present in **source** model we can use `belongsTo`. In this case `Player` is suitable for `belongsTo` because it has `teamId` column.

```
Player.belongsTo(Team)  // `teamId` will be
added on Player / Source model
```

When information about association is present in **target** model we can use `hasOne`. In this case `Coach` is suitable for `hasOne` because `Team` model store information about its `Coach as coachId` field.

```
Coach.hasOne(Team)  // `coachId` will be added
on Team / Target model
```

One-To-Many associations (hasMany)

One-To-Many associations are connecting one source with multiple targets. The targets however are again connected to exactly one specific source.

```
const User = sequelize.define('user', {/* ...
*/})
const Project = sequelize.define('project', {/*
... */})

// OK. Now things get more complicated (not
really visible to the user :)).
// First let's define a hasMany association
Project.hasMany(User, {as: 'Workers'})
```

`getWorkers` and `setWorkers`.

Sometimes you may need to associate records on different columns, you may use `sourceKey` option:

```
const City = sequelize.define('city', {
countryCode: Sequelize.STRING });
const Country = sequelize.define('country', {
isoCode: Sequelize.STRING });

// Here we can connect countries and cities
base on country code
Country.hasMany(City, {foreignKey:
'countryCode', sourceKey: 'isoCode'});
City.belongsTo(Country, {foreignKey:
'countryCode', targetKey: 'isoCode'});
```

So far we dealt with a one-way association. But we want more! Let's define it the other way around by creating a many to many association in the next section.

Belongs-To-Many associations

Belongs-To-Many associations are used to connect sources with multiple targets. Furthermore the targets can also have connections to multiple sources.

```
Project.belongsToMany(User, {through:
'UserProject'});
User.belongsToMany(Project, {through:
'UserProject'});
```

This will create a new model called UserProject with the equivalent foreign keys `projectId` and `userId`. Whether the attributes are camelcase or not depends on the two models joined by the table (in this case User and Project).

Defining `through` is **required**. Sequelize would previously attempt to autogenerate names but that would not always lead to the most logical setups.

This will add methods `getUsers`, `setUsers`, `addUser`, `addUsers` to `Project`, and `getProjects`, `setProjects`, `addProject`, and `addProjects` to `User`.

and projects as tasks by using the alias (`as`) option. We will also manually define the foreign keys to use:

```
User.belongsToMany(Project, { as: 'Tasks',
through: 'worker_tasks', foreignKey: 'userId'
})
Project.belongsToMany(User, { as: 'Workers',
through: 'worker_tasks', foreignKey:
'projectId' })
```

`foreignKey` will allow you to set **source model** key in the **through** relation. `otherKey` will allow you to set **target model** key in the **through** relation.

```
User.belongsToMany(Project, { as: 'Tasks',
through: 'worker_tasks', foreignKey: 'userId',
otherKey: 'projectId'})
```

Of course you can also define self references with belongsToMany:

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

If you want additional attributes in your join table, you can define a model for the join table in sequelize, before you define the association, and then tell sequelize that it should use that model for joining, instead of creating a new one:

```
const User = sequelize.define('user', {})
const Project = sequelize.define('project', {})
const UserProjects =
sequelize.define('userProjects', {
    status: DataTypes.STRING
})

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

To add a new project to a user and set its status, you pass extra `options.through` to the setter, which contains the attributes for the join table

```
user.addProject(project, { through: { status:
'started' }})
```

By default the code above will add projectId and userId to the UserProjects table, and *remove any previously defined* is no reason to have other PK columns. To enforce a primary key on the `UserProjects` model you can add it manually.

```
const UserProjects =
sequelize.define('userProjects', {
  id: {
    type: Sequelize.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  status: DataTypes.STRING
})
```

With Belongs-To-Many you can query based on **through** relation and select specific attributes. For example using `findAll` with **through**

```
User.findAll({
  include: [{
    model: Project,
    through: {
      attributes: ['createdAt', 'startedAt',
'finishedAt'],
      where: {completed: true}
    }
  }]
});
```

Scopes

This section concerns association scopes. For a definition of association scopes vs. scopes on associated models, see Scopes.

Association scopes allow you to place a scope (a set of default attributes for `get` and `create`) on the association. Scopes can be placed both on the associated model (the target of the association), and on the through table for n:m relations.

1:m

Assume we have tables Comment, Post, and Image. A comment can be associated to either an image or a post via `commentable_id` and `commentable` - we say that Post and Image are `Commentable`

```
  title: Sequelize.STRING,
  commentable: Sequelize.STRING,
  commentable_id: Sequelize.INTEGER
});

Comment.prototype.getItem = function(options) {
  return this['get' +
this.get('commentable').substr(0,
1).toUpperCase() +
this.get('commentable').substr(1)](options);
};

Post.hasMany(this.Comment, {
  foreignKey: 'commentable_id',
  constraints: false,
  scope: {
    commentable: 'post'
  }
});
Comment.belongsTo(this.Post, {
  foreignKey: 'commentable_id',
  constraints: false,
  as: 'post'
});

Image.hasMany(this.Comment, {
  foreignKey: 'commentable_id',
  constraints: false,
  scope: {
    commentable: 'image'
  }
});
Comment.belongsTo(this.Image, {
  foreignKey: 'commentable_id',
  constraints: false,
  as: 'image'
});
```

`constraints: false,` disables references constraints - since the `commentable_id` column references several tables, we cannot add a `REFERENCES` constraint to it. Note that the Image -> Comment and Post -> Comment relations define a scope, `commentable: 'image'` and `commentable: 'post'` respectively. This scope is automatically applied when using the association functions:

```
image.getComments()
SELECT * FROM comments WHERE commentable_id =
42 AND commentable = 'image';

image.createComment({
  title: 'Awesome!'
})
INSERT INTO comments (title, commentable_id,
commentable) VALUES ('Awesome!', 42, 'image');

image.addComment(comment);
```

The `getItem` utility function on `Comment` completes the picture - it simply converts the `commentable` string into a call to either `getImage` or `getPost`, providing an abstraction over whether a comment belongs to a post or an image. You can pass a normal options object as a parameter to `getItem(options)` to specify any where conditions or includes.

n:m

Continuing with the idea of a polymorphic model, consider a tag table - an item can have multiple tags, and a tag can be related to several items.

For brevity, the example only shows a Post model, but in reality Tag would be related to several other models.

```
const ItemTag = sequelize.define('item_tag', {
  id : {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  tag_id: {
    type: DataTypes.INTEGER,
    unique: 'item_tag_taggable'
  },
  taggable: {
    type: DataTypes.STRING,
    unique: 'item_tag_taggable'
  },
  taggable_id: {
    type: DataTypes.INTEGER,
    unique: 'item_tag_taggable',
    references: null
  }
});
const Tag = sequelize.define('tag', {
  name: DataTypes.STRING
});

Post.belongsToMany(Tag, {
  through: {
    model: ItemTag,
    unique: false,
    scope: {
      taggable: 'post'
    }
  },
  foreignKey: 'taggable_id',
  constraints: false
});
Tag.belongsToMany(Post, {
  through: {
    model: ItemTag,
  foreignKey: 'tag_id',
  constraints: false
});
```

Notice that the scoped column (`taggable`) is now on the through model (`ItemTag`).

We could also define a more restrictive association, for example, to get all pending tags for a post by applying a scope of both the through model (`ItemTag`) and the target model (`Tag`):

```
Post.hasMany(Tag, {
  through: {
    model: ItemTag,
    unique: false,
    scope: {
      taggable: 'post'
    }
  },
  scope: {
    status: 'pending'
  },
  as: 'pendingTags',
  foreignKey: 'taggable_id',
  constraints: false
});

Post.getPendingTags();
```

```
SELECT `tag`.*  INNER JOIN `item_tags` AS
`item_tag`
ON `tag`.`id` = `item_tag`.`tagId`
  AND `item_tag`.`taggable_id` = 42
  AND `item_tag`.`taggable` = 'post'
WHERE (`tag`.`status` = 'pending');
```

`constraints: false` disables references constraints on the `taggable_id` column. Because the column is polymorphic, we cannot say that it `REFERENCES` a specific table.

Naming strategy

By default sequelize will use the model name (the name passed to `sequelize.define`) to figure out the name of the model when used in associations. For example, a model named `user` will add the functions `get/set/add User` to instances of the associated model, and a property named `.user` in eager loading, while a model named

As we've already seen, you can alias models in associations using `as`. In single associations (has one and belongs to), the alias should be singular, while for many associations (has many) it should be plural. Sequelize then uses the inflection library to convert the alias to its singular form. However, this might not always work for irregular or non-english words. In this case, you can provide both the plural and the singular form of the alias:

```
User.belongsToMany(Project, { as: { singular:
'task', plural: 'tasks' }})
// Notice that inflection has no problem
singularizing tasks, this is just for
illustrative purposes.
```

If you know that a model will always use the same alias in associations, you can provide it when creating the model

```
const Project = sequelize.define('project',
attributes, {
  name: {
    singular: 'task',
    plural: 'tasks',
  }
})

User.belongsToMany(Project);
```

This will add the functions `add/set/get Tasks` to user instances.

Remember, that using `as` to change the name of the association will also change the name of the foreign key. When using `as`, it is safest to also specify the foreign key.

```
Invoice.belongsTo(Subscription)
Subscription.hasMany(Invoice)
```

Without `as`, this adds `subscriptionId` as expected. However, if you were to say `Invoice.belongsTo(Subscription, { as: 'TheSubscription' })`, you will have both `subscriptionId` and `theSubscriptionId`, because sequelize is not smart enough to figure that the calls are two sides of the same relation. 'foreignKey' fixes this problem;

```
'subscription_id' })
Subscription.hasMany(Invoice, { foreignKey:
'subscription_id' )
```

Associating objects

Because Sequelize is doing a lot of magic, you have to call `Sequelize.sync` after setting the associations! Doing so will allow you the following:

```
Project.hasMany(Task)
Task.belongsTo(Project)

Project.create()...
Task.create()...
Task.create()...

// save them... and then:
project.setTasks([task1, task2]).then(() => {
  // saved!
})

// ok, now they are saved... how do I get them
later on?
project.getTasks().then(associatedTasks => {
  // associatedTasks is an array of tasks
})

// You can also pass filters to the getter
method.
// They are equal to the options you can pass
to a usual finder method.
project.getTasks({ where: 'id > 10'
}).then(tasks => {
  // tasks with an id greater than 10 :)
})

// You can also only retrieve certain fields of
a associated object.
project.getTasks({attributes:
['title']}).then(tasks => {
  // retrieve tasks with the attributes "title"
and "id"
})
```

To remove created associations you can just call the set method without a specific id:

```
// remove the association with task1
project.setTasks([task2]).then(associatedTasks
=> {
  // you will get task2 only
})

// remove 'em all
project.setTasks([]).then(associatedTasks => {
  // you will get an empty array
// or remove 'em more directly
project.removeTask(task1).then(() => {
  // it's gone
})

// and add 'em again
project.addTask(task1).then(function() {
  // it's back again
})
```

You can of course also do it vice versa:

```
// project is associated with task1 and task2
task2.setProject(null).then(function() {
  // and it's gone
})
```

For hasOne/belongsTo it's basically the same:

```
Task.hasOne(User, {as: "Author"})
Task.setAuthor(anAuthor)
```

Adding associations to a relation with a custom join table can be done in two ways (continuing with the associations defined in the previous chapter):

```
// Either by adding a property with the name of
the join table model to the object, before
creating the association
project.UserProjects = {
  status: 'active'
}
u.addProject(project)

// Or by providing a second options.through
argument when adding the association,
containing the data that should go in the join
table
u.addProject(project, { through: { status:
'active' }})


// When associating multiple objects, you can
combine the two options above. In this case the
second argument
// will be treated as a defaults object, that
will be used if no data is provided
project1.UserProjects = {
    status: 'inactive'
}

u.setProjects([project1, project2], { through:
{ status: 'active' }})
// The code above will record inactive for
project one, and active for project two in the
join table
```

instance:

```
u.getProjects().then(projects => {
  const project = projects[0]

  if (project.UserProjects.status === 'active')
{
    // .. do magic

    // since this is a real DAO instance, you
can save it directly after you are done doing
magic
    return project.UserProjects.save()
  }
})
```

If you only need some of the attributes from the join table, you can provide an array with the attributes you want:

```
// This will select only name from the Projects
table, and only status from the UserProjects
table
user.getProjects({ attributes: ['name'],
joinTableAttributes: ['status']})
```

Check associations

You can also check if an object is already associated with another one (N:M only). Here is how you'd do it:

```
// check if an object is one of associated
ones:
Project.create({ /* */ }).then(project => {
  return User.create({ /* */ }).then(user => {
    return project.hasUser(user).then(result =>
{
      // result would be false
      return project.addUser(user).then(() => {
        return
project.hasUser(user).then(result => {
          // result would be true
        })
      })
    })
  })
})

// check if all associated objects are as
expected:
// let's assume we have already a project and
two users
project.setUsers([user1, user2]).then(() => {
  return project.hasUsers([user1]);
}).then(result => {
  // result would be true
  return project.hasUsers([user1, user2]);
})
```

Foreign Keys

When you create associations between your models in sequelize, foreign key references with constraints will automatically be created. The setup below:

```
const Task = this.sequelize.define('task', { 
title: Sequelize.STRING })
const User = this.sequelize.define('user', { 
username: Sequelize.STRING })

User.hasMany(Task)
Task.belongsTo(User)
```

Will generate the following SQL:

```
CREATE TABLE IF NOT EXISTS `User` (
  `id` INTEGER PRIMARY KEY,
  `username` VARCHAR(255)
);

CREATE TABLE IF NOT EXISTS `Task` (
  `id` INTEGER PRIMARY KEY,
  `title` VARCHAR(255),
  `user_id` INTEGER REFERENCES `User` (`id`) ON 
DELETE SET NULL ON UPDATE CASCADE
);
```

The relation between task and user injects the `user_id` foreign key on tasks, and marks it as a reference to the `User` table. By default `user_id` will be set to `NULL` if the referenced user is deleted, and updated if the id of the user id updated. These options can be overridden by passing `onUpdate` and `onDelete` options to the association calls. The validation options are `RESTRICT, CASCADE, NO ACTION, SET DEFAULT, SET NULL`.

For 1:1 and 1:m associations the default option is `SET NULL` for deletion, and `CASCADE` for updates. For n:m, the default for both is `CASCADE`. This means, that if you delete or update a row from one side of an n:m association, all the rows in the join table referencing that row will also be deleted or updated.

Adding constraints between tables means that tables must be created in the database in a certain order, when using `sequelize.sync`. If Task has a reference to User, the User sequelize cannot find an order in which to sync. Imagine a scenario of documents and versions. A document can have multiple versions, and for convenience, a document has a reference to its current version.

```
const Document =
this.sequelize.define('document', {
  author: Sequelize.STRING
})
const Version =
this.sequelize.define('version', {
  timestamp: Sequelize.DATE
})

Document.hasMany(Version) // This adds
document_id to version
Document.belongsTo(Version, { as: 'Current',
foreignKey: 'current_version_id'}) // This adds
current_version_id to document
```

However, the code above will result in the following error: `Cyclic dependency found. 'Document' is dependent of itself. Dependency Chain: Document -> Version => Document` . In order to alleviate that, we can pass `constraints: false` to one of the associations:

```
Document.hasMany(Version)
Document.belongsTo(Version, { as: 'Current',
foreignKey: 'current_version_id', constraints:
false})
```

Which will allow us to sync the tables correctly:

```
CREATE TABLE IF NOT EXISTS `Document` (
  `id` INTEGER PRIMARY KEY,
  `author` VARCHAR(255),
  `current_version_id` INTEGER
);
CREATE TABLE IF NOT EXISTS `Version` (
  `id` INTEGER PRIMARY KEY,
  `timestamp` DATETIME,
  `document_id` INTEGER REFERENCES `Document`
(`id`) ON DELETE SET NULL ON UPDATE CASCADE
);
```

Enforcing a foreign key reference without constraints

Sometimes you may want to reference another table, without adding any constraints, or associations. In that

```
// Series has a trainer_id=Trainer.id foreign
reference key after we call
Trainer.hasMany(series)
const Series = sequelize.define('series', {
  title:        DataTypes.STRING,
  sub_title:    DataTypes.STRING,
  description:  DataTypes.TEXT,

  // Set FK relationship (hasMany) with
`Trainer`
  trainer_id: {
    type: DataTypes.INTEGER,
    references: {
      model: "trainer",
      key: "id"
    }
  }
})

const Trainer = sequelize.define('trainer', {
  first_name: DataTypes.STRING,
  last_name:  DataTypes.STRING
});

// Video has a series_id=Series.id foreign
reference key after we call
Series.hasOne(Video)...
const Video = sequelize.define('video', {
  title:        DataTypes.STRING,
  sequence:     DataTypes.INTEGER,
  description:  DataTypes.TEXT,

  // set relationship (hasOne) with `Series`
  series_id: {
    type: DataTypes.INTEGER,
    references: {
      model: Series, // Can be both a string
representing the table name, or a reference to
the model
      key:   "id"
    }
  }
});

Series.hasOne(Video);
Trainer.hasMany(Series);
```

Creating with associations

An instance can be created with nested association in one step, provided all elements are new.

Creating elements of a "BelongsTo", "Has Many" or "HasOne" association

```
const Product =
this.sequelize.define('product', {
  title: Sequelize.STRING
});
const User = this.sequelize.define('user', {
  first_name: Sequelize.STRING,
  last_name: Sequelize.STRING
});
const Address =
this.sequelize.define('address', {
  type: Sequelize.STRING,
  line_1: Sequelize.STRING,
  line_2: Sequelize.STRING,
  city: Sequelize.STRING,
  state: Sequelize.STRING,
  zip: Sequelize.STRING,
});

Product.User = Product.belongsTo(User);
User.Addresses = User.hasMany(Address);
// Also works for `hasOne`
```

A new `Product` , `User` , and one or more `Address` can be created in one step in the following way:

```
return Product.create({
  title: 'Chair',
  user: {
    first_name: 'Mick',
    last_name: 'Broadstone',
    addresses: [{
      type: 'home',
      line_1: '100 Main St.',
      city: 'Austin',
      state: 'TX',
      zip: '78704'
    }]
  }
}, {
  include: [{
    association: Product.User,
    include: [ User.Addresses ]
  }]
});
```

Here, our user model is called `user` , with a lowercase u - This means that the property in the object should also be `user` . If the name given to `sequelize.define` was `User` , the key in the object should also be `User` . Likewise for `addresses` , except it's pluralized being a `hasMany` association.

Creating elements of a "BelongsTo" association with an alias

```
const Creator = Product.belongsTo(User, {as:
'creator'});

return Product.create({
  title: 'Chair',
  creator: {
    first_name: 'Matt',
    last_name: 'Hansen'
  }
}, {
  include: [ Creator ]
});
```

Creating elements of a "HasMany" or "BelongsToMany" association

Let's introduce the ability to associate a product with many tags. Setting up the models could look like:

```
const Tag = this.sequelize.define('tag', {
  name: Sequelize.STRING
});

Product.hasMany(Tag);
// Also works for `belongsToMany`.
```

Now we can create a product with multiple tags in the following way:

```
Product.create({
  id: 1,
  title: 'Chair',
  tags: [
    { name: 'Alpha'},
    { name: 'Beta'}
  ]
}, {
  include: [ Tag ]
})
```

And, we can modify this example to support an alias as well:

```
const Categories = Product.hasMany(Tag, {as:
'categories'});

Product.create({
  id: 1,
  title: 'Chair',
  categories: [
    {id: 1, name: 'Alpha'},
    {id: 2, name: 'Beta'}
  ]
}, {
    as: 'categories'
  }]
})
```