Version: v6 - stable

Polymorphic Associations

Note: *the usage of polymorphic associations in Sequelize, as outlined in this guide, should be done with caution. Don't just copy-paste code from here, otherwise you might easily make mistakes and introduce bugs in your code. Make sure you understand what is going on.*

Concept

A **polymorphic association** consists on two (or more) associations happening with the same foreign key.

For example, consider the models `Image`, `Video` and `Comment`. The first two represent something that a user might post. We want to allow comments to be placed in both of them. This way, we immediately think of establishing the following associations:

- A One-to-Many association between `Image` and `Comment`:

```
Image.hasMany(Comment);
Comment.belongsTo(Image);
```

- A One-to-Many association between `Video` and `Comment`:

```
Video.hasMany(Comment);
Comment.belongsTo(Video);
```

However, the above would cause Sequelize to create two foreign keys on the `Comment` table: `ImageId` and `VideoId`. This is not ideal because this structure makes it look like a comment can be attached at the same time to one image and one video, which isn't true. Instead, what we really want here is precisely a polymorphic association, in which a `Comment` points to a single **Commentable**, an abstract polymorphic entity that represents one of `Image` or `Video`.

Before proceeding to how to configure such an association, let's see how using it looks like:

```
const image = await Image.create({ url: "https://placekitten.com/408/287" });
const comment = await image.createComment({ content: "Awesome!" });

console.log(comment.commentableId === image.id); // true

// We can also retrieve which type of commentable a comment is associated to.
// The following prints the model name of the associated commentable instance.
console.log(comment.commentableType); // "Image"

// We can use a polymorphic method to retrieve the associated commentable, without
// having to worry whether it's an Image or a Video.
const associatedCommentable = await comment.getCommentable();

// In this example, `associatedCommentable` is the same thing as `image`:
const isDeepEqual = require('deep-equal');
console.log(isDeepEqual(image, commentable)); // true
```

Configuring a One-to-Many polymorphic association

To setup the polymorphic association for the example above (which is an example of One-to-Many polymorphic association), we have the following steps:

- Define a string field called `commentableType` in the `Comment` model;
- Define the `hasMany` and `belongsTo` association between `Image`/`Video` and `Comment`:
  - Disabling constraints (i.e. using `{ constraints: false }`), since the same foreign key is referencing multiple tables;
  - Specifying the appropriate association scopes;
- To properly support lazy loading, define a new instance method on the `Comment` model called `getCommentable` which calls, under the hood, the correct mixin to fetch the appropriate commentable;
- To properly support eager loading, define an `afterFind` hook on the `Comment` model that automatically populates the `commentable` field in every instance;
- To prevent bugs/mistakes in eager loading, you can also delete the concrete fields `image` and `video` from Comment instances in the same `afterFind` hook, leaving only the abstract `commentable` field available.

Here is an example:

```
// Helper function
const uppercaseFirst = str => `${str[0].toUpperCase()}${str.substr(1)}`;

class Image extends Model {}
Image.init({
  title: DataTypes.STRING,
  url: DataTypes.STRING
}, { sequelize, modelName: 'image' });

class Video extends Model {}
Video.init({
  title: DataTypes.STRING,
  text: DataTypes.STRING
}, { sequelize, modelName: 'video' });

class Comment extends Model {
  getCommentable(options) {
    if (!this.commentableType) return Promise.resolve(null);
    const mixinMethodName = `get${uppercaseFirst(this.commentableType)}`;
    return this[mixinMethodName](options);
  }
}
Comment.init({
  title: DataTypes.STRING,
  commentableId: DataTypes.INTEGER,
  commentableType: DataTypes.STRING
}, { sequelize, modelName: 'comment' });
```

```
Image.hasMany(Comment, {
  foreignKey: 'commentableId',
  constraints: false,
  scope: {
    commentableType: 'image'
  }
});
Comment.belongsTo(Image, { foreignKey: 'commentableId', constraints: false });

Video.hasMany(Comment, {
  foreignKey: 'commentableId',
  constraints: false,
  scope: {
    commentableType: 'video'
  }
});
Comment.belongsTo(Video, { foreignKey: 'commentableId', constraints: false });

Comment.addHook("afterFind", findResult => {
  if (!Array.isArray(findResult)) findResult = [findResult];
  for (const instance of findResult) {
    if (instance.commentableType === "image" && instance.image !== undefined) {
      instance.commentable = instance.image;
    } else if (instance.commentableType === "video" && instance.video !== undefined) {
      instance.commentable = instance.video;
    }
    // To prevent mistakes:
    delete instance.image;
    delete instance.dataValues.image;
    delete instance.video;
    delete instance.dataValues.video;
  }
});
```

Since the `commentableId` column references several tables (two in this case), we cannot add a `REFERENCES` constraint to it. This is why the `constraints: false` option was used.

Note that, in the code above:

- The *Image* -> *Comment* association defined an association scope: `{ commentableType: 'image' }`
- The *Video* -> *Comment* association defined an association scope: `{ commentableType: 'video' }`

These scopes are automatically applied when using the association functions (as explained in the Association Scopes guide). Some examples are below, with their generated SQL statements:

- `image.getComments()`:

  ```
  SELECT "id", "title", "commentableType", "commentableId", "createdAt", "updatedAt"
  FROM "comments" AS "comment"
  WHERE "comment"."commentableType" = 'image' AND "comment"."commentableId" = 1;
  ```

  Here we can see that `` `comment`.`commentableType` = 'image' `` was automatically added to the `WHERE` clause of the generated SQL. This is exactly the behavior we want.

- `image.createComment({ title: 'Awesome!' })`:

  ```
  INSERT INTO "comments" (
    "id", "title", "commentableType", "commentableId", "createdAt", "updatedAt"
  ) VALUES (
    DEFAULT, 'Awesome!', 'image', 1,
    '2018-04-17 05:36:40.454 +00:00', '2018-04-17 05:36:40.454 +00:00'
  ) RETURNING *;
  ```

- `image.addComment(comment)`:

  ```
  UPDATE "comments"
  SET "commentableId"=1, "commentableType"='image', "updatedAt"='2018-04-17 05:38:43.948 +00:00'
  WHERE "id" IN (1)
  ```

Polymorphic lazy loading

The `getCommentable` instance method on `Comment` provides an abstraction for lazy loading the associated commentable - working whether the comment belongs to an Image or a Video.

It works by simply converting the `commentableType` string into a call to the correct mixin (either `getImage` or `getVideo`).

Note that the `getCommentable` implementation above:

- Returns `null` when no association is present (which is good);
- Allows you to pass an options object to `getCommentable(options)`, just like any other standard Sequelize method. This is useful to specify where-conditions or includes, for example.

Polymorphic eager loading

Now, we want to perform a polymorphic eager loading of the associated commentables for one (or more) comments. We want to achieve something similar to the following idea:

```
const comment = await Comment.findOne({
  include: [ /* What to put here? */ ]
});
console.log(comment.commentable); // This is our goal
```

The solution is to tell Sequelize to include both Images and Videos, so that our `afterFind` hook defined above will do the work, automatically adding the `commentable` field to the instance object, providing the abstraction we want.

For example:

```
const comments = await Comment.findAll({
  include: [Image, Video]
});
for (const comment of comments) {
  const message = `Found comment #${comment.id} with ${comment.commentableType} commentable:`;
```

```
  console.log(message, comment.commentable.toJSON());
}
```

Output example:

```
Found comment #1 with image commentable: { id: 1,
  title: 'Meow',
  url: 'https://placekitten.com/408/287',
  createdAt: 2019-12-26T15:04:53.047Z,
  updatedAt: 2019-12-26T15:04:53.047Z }
```

Caution - possibly invalid eager/lazy loading!

Consider a comment `Foo` whose `commentableId` is 2 and `commentableType` is `image`. Consider also that `Image A` and `Video X` both happen to have an id equal to 2. Conceptually, it is clear that `Video X` is not associated to `Foo`, because even though its id is 2, the `commentableType` of `Foo` is `image`, not `video`. However, this distinction is made by Sequelize only at the level of the abstractions performed by `getCommentable` and the hook we created above.

This means that if you call `Comment.findAll({ include: Video })` in the situation above, `Video X` will be eager loaded into `Foo`. Thankfully, our `afterFind` hook will delete it automatically, to help prevent bugs, but regardless it is important that you understand what is going on.

The best way to prevent this kind of mistake is to **avoid using the concrete accessors and mixins directly at all costs** (such as `.image`, `.getVideo()`, `.setImage()`, etc), always preferring the abstractions we created, such as `.getCommentable()` and `.commentable`. If you really need to access eager-loaded `.image` and `.video` for some reason, make sure you wrap that in a type check such as `comment.commentableType === 'image'`.

Configuring a Many-to-Many polymorphic association

In the above example, we had the models `Image` and `Video` being abstractly called *commentables*, with one *commentable* having many comments. However, one given comment would belong to a single *commentable* - this is why the whole situation is a One-to-Many polymorphic association.

Now, to consider a Many-to-Many polymorphic association, instead of considering comments, we will consider tags. For convenience, instead of calling Image and Video as *commentables*, we will now call them *taggables*. One *taggable* may have several tags, and at the same time one tag can be placed in several *taggables*.

The setup for this goes as follows:

- Define the junction model explicitly, specifying the two foreign keys as `tagId` and `taggableId` (this way it is a junction model for a Many-to-Many relationship between `Tag` and the abstract concept of *taggable*);
- Define a string field called `taggableType` in the junction model;
- Define the `belongsToMany` associations between the two models and `Tag`:
  - Disabling constraints (i.e. using `{ constraints: false }`), since the same foreign key is referencing multiple tables;
  - Specifying the appropriate association scopes;
- Define a new instance method on the `Tag` model called `getTaggables` which calls, under the hood, the correct mixin to fetch the appropriate taggables.

Implementation:

```
class Tag extends Model {
  getTaggables(options) {
    const images = await this.getImages(options);
    const videos = await this.getVideos(options);
    // Concat images and videos in a single array of taggables
    return images.concat(videos);
  }
}
Tag.init({
  name: DataTypes.STRING
}, { sequelize, modelName: 'tag' });

// Here we define the junction model explicitly
class Tag_Taggable extends Model {}
Tag_Taggable.init({
  tagId: {
    type: DataTypes.INTEGER,
    unique: 'tt_unique_constraint'
  },
  taggableId: {
    type: DataTypes.INTEGER,
    unique: 'tt_unique_constraint',
    references: null
  },
  taggableType: {
    type: DataTypes.STRING,
    unique: 'tt_unique_constraint'
  }
}, { sequelize, modelName: 'tag_taggable' });

Image.belongsToMany(Tag, {
  through: {
    model: Tag_Taggable,
    unique: false,
    scope: {
      taggableType: 'image'
    }
  },
  foreignKey: 'taggableId',
  constraints: false
});
Tag.belongsToMany(Image, {
  through: {
    model: Tag_Taggable,
    unique: false
  },
  foreignKey: 'tagId',
  constraints: false
});

Video.belongsToMany(Tag, {
  through: {
    model: Tag_Taggable,
```

```
    unique: false,
    scope: {
      taggableType: 'video'
    }
  },
  foreignKey: 'taggableId',
  constraints: false
});
Tag.belongsToMany(Video, {
  through: {
    model: Tag_Taggable,
    unique: false
  },
  foreignKey: 'tagId',
  constraints: false
});
```

The `constraints: false` option disables references constraints, as the `taggableId` column references several tables, we cannot add a `REFERENCES` constraint to it.

Note that:

- The *Image* -> *Tag* association defined an association scope: `{ taggableType: 'image' }`
- The *Video* -> *Tag* association defined an association scope: `{ taggableType: 'video' }`

These scopes are automatically applied when using the association functions. Some examples are below, with their generated SQL statements:

- `image.getTags()`:

```
SELECT
  `tag`.`id`,
  `tag`.`name`,
  `tag`.`createdAt`,
  `tag`.`updatedAt`,
  `tag_taggable`.`tagId` AS `tag_taggable.tagId`,
  `tag_taggable`.`taggableId` AS `tag_taggable.taggableId`,
  `tag_taggable`.`taggableType` AS `tag_taggable.taggableType`,
  `tag_taggable`.`createdAt` AS `tag_taggable.createdAt`,
  `tag_taggable`.`updatedAt` AS `tag_taggable.updatedAt`
FROM `tags` AS `tag`
INNER JOIN `tag_taggables` AS `tag_taggable` ON
  `tag`.`id` = `tag_taggable`.`tagId` AND
  `tag_taggable`.`taggableId` = 1 AND
  `tag_taggable`.`taggableType` = 'image';
```

  Here we can see that `` `tag_taggable`.`taggableType` = 'image' `` was automatically added to the `WHERE` clause of the generated SQL. This is exactly the behavior we want.

- `tag.getTaggables()`:

```
SELECT
  `image`.`id`,
  `image`.`url`,
  `image`.`createdAt`,
  `image`.`updatedAt`,
  `tag_taggable`.`tagId` AS `tag_taggable.tagId`,
  `tag_taggable`.`taggableId` AS `tag_taggable.taggableId`,
  `tag_taggable`.`taggableType` AS `tag_taggable.taggableType`,
  `tag_taggable`.`createdAt` AS `tag_taggable.createdAt`,
  `tag_taggable`.`updatedAt` AS `tag_taggable.updatedAt`
FROM `images` AS `image`
INNER JOIN `tag_taggables` AS `tag_taggable` ON
  `image`.`id` = `tag_taggable`.`taggableId` AND
  `tag_taggable`.`tagId` = 1;

SELECT
  `video`.`id`,
  `video`.`url`,
  `video`.`createdAt`,
  `video`.`updatedAt`,
  `tag_taggable`.`tagId` AS `tag_taggable.tagId`,
  `tag_taggable`.`taggableId` AS `tag_taggable.taggableId`,
  `tag_taggable`.`taggableType` AS `tag_taggable.taggableType`,
  `tag_taggable`.`createdAt` AS `tag_taggable.createdAt`,
  `tag_taggable`.`updatedAt` AS `tag_taggable.updatedAt`
FROM `videos` AS `video`
INNER JOIN `tag_taggables` AS `tag_taggable` ON
  `video`.`id` = `tag_taggable`.`taggableId` AND
  `tag_taggable`.`tagId` = 1;
```

Note that the above implementation of `getTaggables()` allows you to pass an options object to `getCommentable(options)`, just like any other standard Sequelize method. This is useful to specify where-conditions or includes, for example.

Applying scopes on the target model

In the example above, the `scope` options (such as `scope: { taggableType: 'image' }`) were applied to the *through* model, not the *target* model, since it was used under the `through` option.

We can also apply an association scope on the target model. We can even do both at the same time.

To illustrate this, consider an extension of the above example between tags and taggables, where each tag has a status. This way, to get all pending tags of an image, we could establish another `belognsToMany` relationship between `Image` and `Tag`, this time applying a scope on the through model and another scope on the target model:

```
Image.belongsToMany(Tag, {
  through: {
    model: Tag_Taggable,
    unique: false,
    scope: {
      taggableType: 'image'
    }
  },
  scope: {
    status: 'pending'
  },
  as: 'pendingTags'
```

```
  foreignKey: 'taggableId',
  constraints: false
});
```

This way, when calling `image.getPendingTags()`, the following SQL query will be generated:

```
SELECT
  `tag`.`id`,
  `tag`.`name`,
  `tag`.`status`,
  `tag`.`createdAt`,
  `tag`.`updatedAt`,
  `tag_taggable`.`tagId` AS `tag_taggable.tagId`,
  `tag_taggable`.`taggableId` AS `tag_taggable.taggableId`,
  `tag_taggable`.`taggableType` AS `tag_taggable.taggableType`,
  `tag_taggable`.`createdAt` AS `tag_taggable.createdAt`,
  `tag_taggable`.`updatedAt` AS `tag_taggable.updatedAt`
FROM `tags` AS `tag`
INNER JOIN `tag_taggables` AS `tag_taggable` ON
  `tag`.`id` = `tag_taggable`.`tagId` AND
  `tag_taggable`.`taggableId` = 1 AND
  `tag_taggable`.`taggableType` = 'image'
WHERE (
  `tag`.`status` = 'pending'
);
```

We can see that both scopes were applied automatically:

- `` `tag_taggable`.`taggableType` = 'image' `` was added automatically to the `INNER JOIN`;
- `` `tag`.`status` = 'pending' `` was added automatically to an outer where clause.

Edit this page

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

Previous
« Association Scopes

Next
Other topics »