Sequelize Table Associations (Joins)

Sep 12 2016

This post goes step-by-step through a basic example (a blog with users, posts, and post comments) of associating tables using the Node ORM, Sequelize. I will include the data used in the examples, as well as a GET API endpoint that interacts with our tables, sending back beautifully transformed responses. Three POST endpoints are found in the accompanying repo, but will not be covered here.

The examples below use Node version 6, and Sequelize 3.24.1. I encourage you to clone the aforementioned repo for reference.

The Sequelize docs on associations can be found here.

Introduction

Sequelize is a Node package that allows the developer to interact with a variety of SQL databases using a single API. Specifically, it performs Object Relational Mapping (ORM) between your backend code and a SQL database.

This means you, the developer, can write object-oriented code and Sequelize will translate it into a SQL dialect. Whether you're using MySQL, SQLite, MSSQL, or PostgreSQL, Sequelize has you covered. Indicate a database dialect in your configuration file, and Sequelize takes care of the rest.

Sequelize is promise-based, which is awesome!, so you can chain your functions for increased readability, and easy maintenance down the road.

If you'd like a nice overview of JavaScript promises, I suggest you check out David Walsh's post on the topic.

It's pretty easy to find basic Sequelize CRUD examples online, but I have yet to find a post with a straightforward explanation of table associations (joins). Here is that missing explanation.

There are two aspects of joining tables in Sequelize:

1. Declaring *associations*
2. Declaring `include` s tions using Sequel models.

Before we get started on number one, we'll need some Sequelize models to work with. For those new to ORMs, a model corresponds to a database table, but it doesn't have to be an exact match; it's okay to use only a limited number of columns in your model. That said, know you may use *fewer* of a table's columns in a Sequelize model, but you *cannot use more*.

Models

To define a model is to define a database table. This is the essence of an ORM like Sequelize. You define models, which you use in an object-oriented way in your code (using methods, chaining promises, etc). The actions you perform with these object-models are translated into an SQL dialect.

The blog API we'll discuss uses three models - users, posts, and comments.

Users

```
'use strict'

module.exports = (sequelize, DataTypes) => {
  const Users = sequelize.define('users', {
    id: {
      type: DataTypes.UUID,
      primaryKey: true,
      defaultValue: DataTypes.UUIDV4,
      allowNull: false
    },
    username: {
      type: DataTypes.STRING,
      required: true
    },
    role: {
      type: DataTypes.ENUM,
      values: ['user', 'admin', 'disabled']

    },
    created_at: {
      type: DataTypes.DATE,
      allowNull: false
    },
    updated_at:  DataTypes.DATE,
    deleted_at: DataTypes.DATE
  }, {
    underscored: true
  });
  return Users;
};
```

Posts


```
'use strict'

module.exports = (se
  const Posts = sequ
    id: {
      type: DataTypes.UUID,
      primaryKey: true,
      defaultValue: DataTypes.UUIDV4,
      allowNull: false
    },
    user_id: {
      type: DataTypes.UUID,
      allowNull: false
    },
    content: {
      type: DataTypes.TEXT,
      required: true
    },
    created_at: {
      type: DataTypes.DATE,
      allowNull: false
    },
    updated_at:  DataTypes.DATE,
    deleted_at: DataTypes.DATE
  }, {
    underscored: true
  });
  return Posts;
};
```

Comments

```
'use strict'

module.exports = (sequelize, DataTypes) => {
  const Comments = sequelize.define('comments', {
    id: {
      type: DataTypes.UUID,
      primaryKey: true,
      defaultValue: DataTypes.UUIDV4,
      allowNull: false
    },
    post_id: {
      type: DataTypes.UUID,
      allowNull: false
    },
    content: {
      type: DataTypes.TEXT,
      required: true
    },
    commenter_username: {
      type: DataTypes.STRING,
      required: true
    },
```

```
      type: DataType
      required: true
    },
    status: {
      type: DataType.ENUM
      values: ['appr
    },
    created_at: {
      type: DataTypes.DATE,
      allowNull: false
    },
    updated_at:  DataTypes.DATE,
    deleted_at: DataTypes.DATE
  }, {
    underscored: true
  });

  return Comments;
};
```


I won't go over datatypes or other information regarding the API for Sequelize models, as the focus here is associations. For more info on models, check out the docs. I will note, however, `underscored: true` indicates the the column names of the database tables are snake_case rather than camelCase.

Side note: Where I work, we have a convention of attaching all Sequelize models to a single `db` object and injecting this object into all routes and controllers. It's a convenient way to have access to everything. Using this convention, here's what the `db.js` file in the repo looks like:

```
const db = {};

db.Sequelize = Sequelize;
db.sequelize = sequelize;

db.users = require('../models/users.js')(sequelize, Sequelize);
db.comments = require('../models/comments.js')(sequelize, Sequelize);
db.posts = require('../models/posts.js')(sequelize, Sequelize);
```

Back to business - let's move on to the first aspect of associations.

1. Declaring associations in a config file

There are three Sequelize associations relevant to our example: `hasOne`, `belongsTo`, and `hasMany`.

The Sequelize docs tell us:

> *`hasOne` and `belongsT`* **Get notified about new content** *er. `hasOne` inserts the association key in **targe*** email address ***e source** model.*

Expanding on the above, in ✔ **targets**, we use `hasMany`.

In our case, the user_id is found in the Posts table. Because a single user usually writes multiple blog posts, there may be 10 posts in the Posts table that all have the same user_id. In this case a user `hasMany` posts.

Using my `db` object, we would declare this association (in `db.js`) like so, **`db.users.hasMany(db.posts)`**.

In the language of the Sequelize docs, the Users model is the **source** (i.e. it is the source of the id), and the Posts model is the **target** (i.e. it is the model that contains a foreign id/key).

Just as `hasMany` aligns with everyday speech, so to does `belongsTo`. In our case, posts belong to the user. In code we would declare the association as **`db.posts.belongsTo(db.users)`**.

The relationship of comments to posts is the same as that of posts to users. Just as each user can have multiple posts, so can each post have multiple comments. Here's the code: **`db.comments.belongsTo(db.posts)`** and **`db.posts.hasMany(db.comments)`**.

Associations can get hard to track, so here's the way I think about `belongsTo`:

In our tables, the user_id is a column in the posts table. The association of a foreign id in a table (like the user_id in the posts table) is a `belongsTo` relationship. I use the phrase "**the table belongs to the column**" to help me code this declaration correctly: (`db.posts.belongsTo(db.users)`).

For your reference:

```
sourceModel.hasOne(targetModel)
sourceModel.hasMany(targetModel)
targetModel.belongsTo(sourceTable)
```

The complete `db.js` file should look like this:

```
1  'use strict'
2
3  const Sequelize = require('sequelize');
4  const env = require('./env');
5  const sequelize = new Sequelize(env.DATABASE_NAME, env.DATABASE_USERNAME, env.DATABASE_PASSWORD, {
```

```
  port: env.DATABASE
  dialect: env.DATAB
  define: {
    underscored: tru
  }
});

// Connect all the models/tables in the database to a db object,
//so everything is accessible via one object
const db = {};

db.Sequelize = Sequelize;
db.sequelize = sequelize;

//Models/tables
db.users = require('../models/users.js')(sequelize, Sequelize);
db.comments = require('../models/comments.js')(sequelize, Sequelize);
db.posts = require('../models/posts.js')(sequelize, Sequelize);

//Relations
db.comments.belongsTo(db.posts);
db.posts.hasMany(db.comments);
db.posts.belongsTo(db.users);
db.users.hasMany(db.posts);

module.exports = db;
```


The data

Before turning to the GET route (that usesExpress.js), let's take a look at some sample data so our API response makes sense. (To see some more routes, check out the repo.)

Users table

id	username	role	created_at	updated_at	deleted_at
06896bd4-8cbc-48c6-8c46-9364a6d939c4	larrycool	user	09/11/2016	09/11/2016	null
92eeaac0-8845-4277-b5d6-b8adfc41ca03	jimmy_jonez	admin	09/11/2016	09/11/2016	null

Posts table

id	user_id	content	created_at	updated_at	deleted_at
55587382-1082-4ee8-ab9c-	06896bd4-8cbc-48c6-8c46-	This is larrycool's first	09/11/2016	09/11/2016	null

id				updated_at	deleted_at
04a657b7-ea72-41b6-9cd3-b0a9c420ea9c				09/11/2016	null
276ac6e9-5e9e-4ef2-96f9-ee7818f3f844	92eeaac0-8845-4277-b5d6-b8adfc41ca03	This is jimmy_jonez's first post.	09/11/2016	09/11/2016	null
32ed0999-1d54-4020-8845-5b13a452cc2d	92eeaac0-8845-4277-b5d6-b8adfc41ca03	This is jimmy_jonez's second post.	09/11/2016	09/11/2016	null

Comments table

id	post_id	commenter_username	commenter_email	content	status	created_at	updated_at	deleted_at
1	1	scuba_human	swim@gmail.com	Very interesting, but have you hear of shark week?	approved	09/11/2016	09/11/2016	null
2	1	jabber_jabs	sillystring@hotmail.com	I completely disagree, because bagels.	in review	09/11/2016	09/11/2016	null
3	2	terry_mcmuffin	teacherlady@yahoo.com	I think the children would devour this.	approved	09/11/2016	09/11/2016	null
4	4	vortex	blackmagic@gmail.com	Mixy, mix, the poison potion.	rejected	09/11/2016	09/11/2016	null

There's the data we'll be using in the API calls below. Let's turn to our routes.

2. Declaring includes in our actions

Now for the second aspect nd nested `include` s) while performing actions u

To illustrate this, let's take nt begins by querying the users table, but this table is linked to others by `include` s, so more than just users are being queried. Lines 6-8 join the posts table to the users table. Then lines 9-11 join the comments table to the posts table. The API will serve up all users, and all user posts, AND all comments attached to these posts.

```
1  //import the models (as noted above use a db object)
2  //import express and instantiate your app object
3
4  app.get('/users', (req, res) => {
5      db.users.findAll({
6        include: [
7          {
8            model: db.posts,
9            include: [
10             {
11               model: db.comments
12             }
13           ]
14         }
15       ]
16     }).then(users => {
17       const resObj = users.map(user => {
18
19         //tidy up the user data
20         return Object.assign(
21           {},
22           {
23             user_id: user.id,
24             username: user.username,
25             role: user.role,
26             posts: user.posts.map(post => {
27
28               //tidy up the post data
29               return Object.assign(
30                 {},
31                 {
32                   post_id: post.id,
33                   user_id: post.user_id,
34                   content: post.content,
35                   comments: post.comments.map(comment => {
36
37                     //tidy up the comment data
38                     return Object.assign(
39                       {},
40                       {
41                         comment_id: comment.id,
42                         post_id: comment.post_id,
43                         commenter: comment.commenter_username,
44                         commenter_email: comment.commenter_email,
45                         content: comment.content
46                       }
```

```
                }
                )
            })
        }
    )
});
res.json(resObj);
});
});
```


After the query, the data is transformed using `Object.assign()`. Each user is assigned two properties, `username` and a `posts` array. Within the posts array, each post is assigned an array of `comments`. (`Object.assign()` is a *very* useful JavaScript method introduced to the language in the ES2015 spec. Here is a quick overview.)

If you want to send the raw data, without transformations, to the client, then remove lines 19-54 above so the route only contains `res.json(users)`. I've used object assign to format our response because in most cases it isn't necessary to send data for `created_at`, `updated_at`, etc. Sometimes you may want this data, so transform (or don't transform!) your data accordingly.

Below is the JSON response sent by our GET request (using the dummy data in the tables above):


```
[
    {
        "user_id": "06896bd4-8cbc-48c6-8c46-9364a6d939c4",
        "username": "larrycool",
        "role": "user",
        "posts": [
            {
                "post_id": "55587382-1082-4ee8-ab9c-26eb738c0d87",
                "user_id": "06896bd4-8cbc-48c6-8c46-9364a6d939c4",
                "content": "This is larrycool's first post.",
                "comments": [
                    {
                        "comment_id": "d303a076-ec42-4c65-a49e-69666cbce193",
                        "post_id": "55587382-1082-4ee8-ab9c-26eb738c0d87",
                        "commenter": "jabber_jabs",
                        "commenter_email": "sillystring@hotmail.com",
                        "content": "I completely disagree, because bagels."
                    },
                    {
                        "comment_id": "099f3519-8737-45bc-90f8-2902e4cce1d1",
                        "post_id": "55587382-1082-4ee8-ab9c-26eb738c0d87",
                        "commenter": "scuba_human",
                        "commenter_email": "swim@gmail.com",
                        "content": "Very interesting, but have you hear of shark week?"
                    }
                ]
            },
            {
```

```
                "use
                "con
                "com
                                                                    ",
                        "commenter": "terry_mcmurfin",
                        "commenter_email": "teacherlady@yahoo.com",
                        "content": "I completely disagree, because bagels."
                    }
                ]
            }
        ]
    },
    {
        "user_id": "92eeaac0-8845-4277-b5d6-b8adfc41ca03",
        "username": "jimmy_jonez",
        "role": "admin",
        "posts": [
            {
                "post_id": "276ac6e9-5e9e-4ef2-96f9-ee7818f3f844",
                "user_id": "92eeaac0-8845-4277-b5d6-b8adfc41ca03",
                "content": "This is jimmy_jonez's first post.",
                "comments": []
            },
            {
                "post_id": "32ed0999-1d54-4020-8845-5b13a452cc2d",
                "user_id": "92eeaac0-8845-4277-b5d6-b8adfc41ca03",
                "content": "This is jimmy_jonez's second post.",
                "comments": [
                    {
                        "comment_id": "0c2d49f5-1d4d-417e-b3f6-62980072ae13",
                        "post_id": "32ed0999-1d54-4020-8845-5b13a452cc2d",
                        "commenter": "vortex",
                        "commenter_email": "blackmagic@gmail.com",
                        "content": "Mixy, mix, the poison potion."
                    }
                ]
            }
        ]
    }
]
```


That's it! We've joined three tables, and formatted a lovely response for anyone hitting this route.

To see some post requests in action - to create a user, a post, and a comment - check out the repo for this post.

If you'd like to be notified when I publish new content, sign up for my mailing list in the navbar.

TAGS

node.js express.js sequelize api JavaScript

NEXT POST

Sequelize CRUD 101

This tutorial covers the basics of CRUD operations using the Node ORM Sequelize. We will use Express.js to…

03 OCT 2016