Maria Burlando

114 Followers About Follow

A Guide to ORM Sequelize

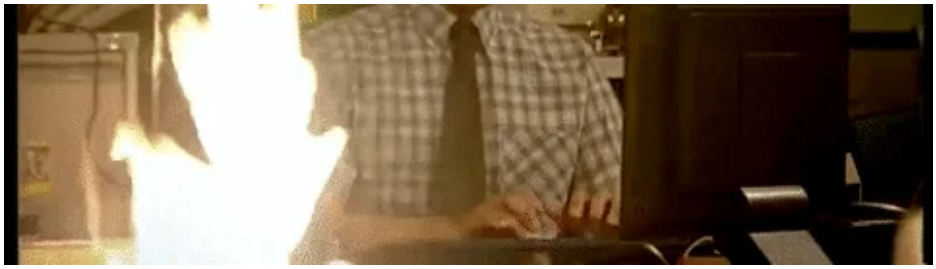
Maria Burlando Dec 17, 2018 · 7 min read

A few days ago, I started trying to figure out Sequelize, a promise-based ORM for NodeJS. I have always been a fan of JavaScript; it was the first programming language I started studying when I was only 15, along with — surprise, surprise — HTML5 and CSS3.

Suffice to say, I wanted to try and find the best ORM solution for NodeJS. After all, C# has Entity Framework and Java has Hibernate, so it was ripe time to find one for Node as well. I made my first search and found Sequelize to have around 250,000 downloads per week on npm. Not bad…

I started reading the documentation with its Getting Started section. It looked easy and fun, the code flowing out of me like a rivulet in spring, but then I arrived at a point where all my attempts just snagged on a barb in the documentation. I couldn’t for the life of me create a foreign key constraint in the PostgreSQL database I was using despite all the associations I wrote in my code. And so, for a whole morning, this is what I did to my computer:

When I finally got to the Migration section of the documentation, you can‘t imagine what my reaction was to find out I had started it all wrong. When I finally found how easy it was to use the Sequelize CLI, it all became a piece of cake. That is, if you’re ready to scour the Internet for every little information that is missing from the real Sequelize documentation; you have to forsake everything: how hungry you’re starting to grow, how the laundry starts piling up in the corner, how the house starts catching fire…



But in the end, I did it! And this is why I decided to share this tutorial in order to help others set up Sequelize with clear understanding of the process.

So, here we go!

How to set up a NodeJS application with Sequelize/Sequelize CLI

First things first, in an empty folder we initialize npm — *duh!*

```
npm init
```

Then we’re going to install Sequelize:

```
npm install --save sequelize
```

And the database we would like to use. In this case I’m using PostgreSQL.

```
npm install --save pg pg-hstore
```

This is described in the Getting Started section of the Sequelize documentation, which I would like to rename as *“Getting Stranded”*.

However, we want to do much better than play around with fancy little methods. We want to install the Sequelize CLI.

My advice is to install it locally because if you install it globally you’ll also have to install Sequelize and PostgreSQL globally because it will not be able to resolve them. So, in our application folder we write:

```
npm install --save sequelize-cli
```

And to initialize sequelize we write:

```
node_modules/.bin/sequelize init
```

This will do some Bibbidi-bobbidi-booing in our folder (I’ll call it applicationFolder) and create this schema for us:

```
applicationFolder
  |-config
     |-config.js
  |-models
     |-index.js
  |-migrations
  |-seeders
```

The index.js in models you don’t have to touch it because it’s all logic to render the models available, and other cool stuff. But the config.js is really interesting because there is where you can configure the databases you’re going to use for the three environments of development, testing and production.

```
{
  "development": {
    "username": "root",
    "password": null,
    "database": "database_development",
    "host": "127.0.0.1",
```

```
    "dialect": "mysql"
  },
  "test": {
    "username": "root",
    "password": null,
    "database": "database_test",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "production": {
    "username": "root",
    "password": null,
    "database": "database_test",
    "host": "127.0.0.1",
    "dialect": "mysql"
  }
}
```

config.js hosted with ♥ by GitHub view raw

For my development environment, I’ll be changing it like so:

```
"development": {
    "username": "postgres",
    "password": "secret",
    "database": "postgres",
    "host": "127.0.0.1",
    "dialect": "postgres"
}
```

And this is how you set up Sequelize and Sequelize CLI in your NodeJS application.

Create your models and migrations

Now, for this application we want to create two models in the database: a Product and a Category.

Sequelize CLI has lots of handy commands, but we want to start and create a new Product model with a sku, a name, and a price. To do this we use:

```
node_modules/.bin/sequelize model:generate --name Product --
attributes sku:string,name:string,price:double
```

Note: the name and attributes are compulsory.

This will create a product.js file in the models folder.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Product = sequelize.define('Product', {
    sku: DataTypes.STRING,
    name: DataTypes.STRING,
    price: DataTypes.DOUBLE
  }, {});
  Product.associate = function(models) {
    // associations can be defined here
  };
  return Product;
};
```

product.js hosted with ♥ by GitHub view raw

And a corresponding migration XXXXXXXXXXXXXX-create-product.js file in the migrations folder.

```
'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Products', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      sku: {
        type: Sequelize.STRING
      },
      name: {
        type: Sequelize.STRING
```

```
      },
      price: {
        type: Sequelize.DOUBLE
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Products');
  }
};
```

XXXXXXXXXXXXXX-create-product.js hosted with ♥ by GitHub view raw

Note: The up function is used to actuate the migration, whereas the down function is used to undo the migration.

Until now, we haven’t inserted anything in the database, so we’re going to migrate this cool stuff and write the model in our PostgreSQL database.

```
node_modules/.bin/sequelize db:migrate
```

As per documentation:

> *This command will execute these steps:*
>
> ** Will ensure a table called `SequelizeMeta` in database. This table is used to record which migrations have run on the current database*

> ** Start looking for any migration files which haven’t run yet. This is possible by checking* `SequelizeMeta` *table. In this case it will run* `XXXXXXXXXXXXX-create-product.js` *migration, which we created in last step.*
>
> ** Creates a table called* `Products` *with all columns as specified in its migration file.*

If we want to undo the migration, we type:

```
node_modules/.bin/sequelize db:migrate:undo
```

Moving on, we’re going to create the Category model with a name and a description.

```
node_modules/.bin/sequelize model:generate --name Category --
attributes name:string,description:string
```

As with the Product model we’ll get a category.js in the models folder and a XXXXXXXXXXXXX-create-category.js in the migration folders.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Category = sequelize.define('Category', {
    name: DataTypes.STRING,
    description: DataTypes.STRING
  }, {});
  Category.associate = function(models) {
    // associations can be defined here
  };
  return Category;
};
```

category.js hosted with ♥ by GitHub view raw

```
'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Categories', {
```

```
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      name: {
        type: Sequelize.STRING
      },
      description: {
        type: Sequelize.STRING
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Categories');
  }
};
```

XXXXXXXXXXXXXX-create-category.js hosted with ♥ by GitHub view raw

And again, to migrate this model to PostgreSQL, we type:

```
node_modules/.bin/sequelize db:migrate
```

Note: instead of doing two migrations we could have generated the two models first, and then executed the above command. Migrations run in order of creation.

Creating associations and foreign keys

What I want to do for these two models is to associate them.

Each product should belong to one category (hypothetically). So, we want a relationship one-to-many between category and products.

If you note in the product.js file in the models folder you’ll see a method like this:

```
Product.associate = function(models) {
    // associations can be defined here
};
```

As per Sequelize documentation we’ll be substituting the comment with this:

```
Product.associate = function(models) {
    Product.belongsTo(models.Category);
};
```

When you start studying Sequelize, you’re supposed to sync your table after defining it and after defining its associations with the `sync()` function, but that doesn’t work if you want to make a foreign key constraint in the database.

So, what you do instead, is create a new migration file and add a new column to the Product model which will write the `CategoryId` as foreign key in the database.

```
node_modules/.bin/sequelize migration:generate --name add-prod-cat-association
```

You’ll find a new empty migration file in the migrations folder called XXXXXXXXXXXXXX-add-prod-cat-association.js where you’ll write the following:

```
'use strict';

module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.addColumn(
```

```
      'Products', // name of Source model
      'CategoryId', // name of the key we're adding
      {
        type: Sequelize.INTEGER,
        references: {
          model: 'Categories', // name of Target model
          key: 'id', // key in Target model that we're referencing
        },
        onUpdate: 'CASCADE',
        onDelete: 'SET NULL',
      }
    );
  },

  down: (queryInterface, Sequelize) => {
    return queryInterface.removeColumn(
      'Products', // name of Source model
      'CategoryId' // key we want to remove
    );
  }
};
```

XXXXXXXXXXXXXX-add-prod-cat-association.js hosted with ♥ by GitHub view raw

Note: the type of the foreign key must be equal to the type of the key in the Target model, in this case `Sequelize.INTEGER`.

We run:

```
node_modules/.bin/sequelize db:migrate
```

And in your database underneath the rest of the Product properties, you’ll finally have this very satisfying result:

```
"CategoryId" integer,
  CONSTRAINT "Products_pkey" PRIMARY KEY (id),
  CONSTRAINT "Products_CategoryId_fkey" FOREIGN KEY ("CategoryId")
```

```
REFERENCES public."Categories" (id) MATCH SIMPLE
ON UPDATE CASCADE ON DELETE SET NULL
```

Ok, now you’re going to tell me, “This is really cool, but how am I going to write stuff inside it, and all the things a developer is supposed to do with databases? Do I have to write migration after migration?”

My dear, now we can go back to the *“Getting Stranded”* section because we’re finally able to write some good stuff.

Insert data into our database

First of all, we’re going to create an index.js file in our root folder. This is where, if needed, we’ll work with Express, body-parser, cookie-parser, and where we’ll write our routes. But for the time being, let’s suppose it’s just a simple JavaScript file that, like a child, likes to play around with a few basic functions.

In the index.js file we’re going to import the following:

```
const models = require('./models');
const Op = models.Sequelize.Op;
```

Note: Op is required to make operations in queries.

If you want to check that the models are now available, you can write a simple function like:

```
models.Product.findAll().then(pr => console.log(pr))
```

However, if you try to run it, you’ll get the error:

```
ReferenceError: Sequelize is not defined
```

To fix this you need to write the following in every file in the models folder. In this case in product.js and in category.js.

```
var Sequelize = require('sequelize');
```

Now, it should work, and of course the result will be an empty array because there is nothing in our database yet.

Let’s instead create a function that accepts an object category and an array of object products. We want this function to create a new row in the categories table in our PostgreSQL database with the new category object, while also creating new rows in the products table for each product object in the product array. Of course, this method should be called when every product in the array corresponds to the category object that is being passed.

```
1  const models = require('./models');
2  const Op = models.Sequelize.Op;
3
```

```
let createCatWithProds = async(catObj, prodArr) => {
    let catId = await models.Category.create({
        name: catObj.name,
        description:catObj.description
    }).then(cat => cat.id).catch(err => console.log(err));
    await prodArr.map(pr => {
        return models.Product.create({
            sku: pr.sku,
            name: pr.name,
            pr: pr.price,
            CategoryId:catId
        }).catch(err => console.log(err))
    });
  return;
}
```

index.js hosted with ♥ by GitHub view raw

What this function does is to create a category with the data the method receives, and save its newly generated id in `catId`. Then, we map the array of products and create new rows in the products table in the PostgreSQL database where we’ll also be setting the product’s `CategoryId` corresponding to `catId`.

An example of this function’s usage would be:

```
let newCategory = {
    name: "Fantasy",
    description:"Fantasy genre of the book"
};

let newProducts = [
    {
     sku: "nvw1",
     name: "Neverwhere",
     price: 10.99
    },
    {
     sku: "nrl2",
     name: "Northern Lights",
     price: 8.99
}
];
```

```
CreateCatWithProds(newCategory, newProducts)
```

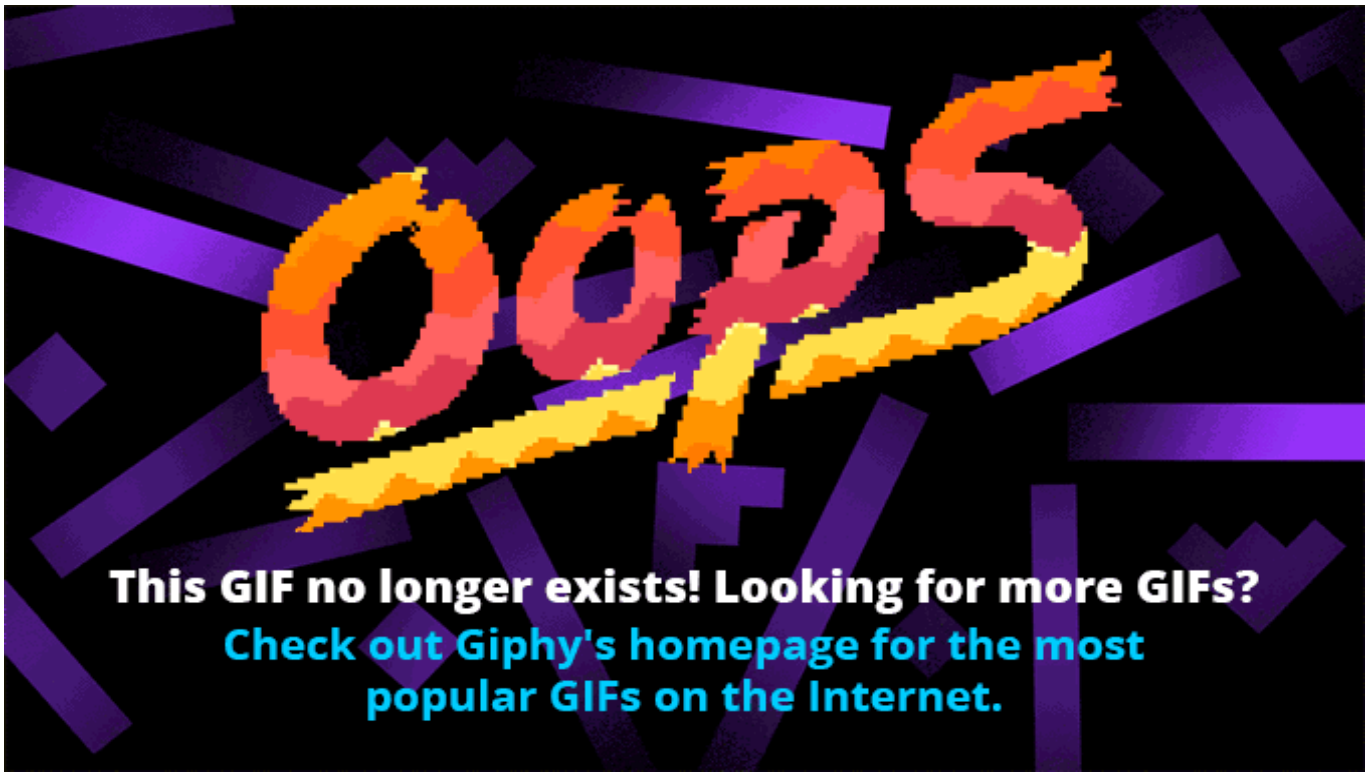
(This is just an example of course. It would have made more sense to create a Book model rather than a Product, but it doesn’t matter now.)

This is one way to do this, but according to the Sequelize documentation there are many more methods available.

For the sake of this tutorial I’ll be stopping here.

A final word

Sequelize is great, but it’s important that we understand migrations first if we want to write clear structural database operations, like adding tables, columns, etc; unless we don’t want to end up like this angry panda midway Sequelize “*Getting Stranded*” section.



So, I hope this tutorial will help you understand better the process of setting up Sequelize for your NodeJS application, and not run into unnecessary problems along the way — like I did.

Have a great day, Sequelizers!

JavaScript Sequelize Nodejs Postgresql Orm