The Code Barbarian

The software development blog of Valeri Karpov

Mistakes You're Probably Making With MongooseJS, And How To Fix Them

Posted on June 6, 2013 by vkarpov15

If you're familiar with Ruby on Rails and are using MongoDB to build a NodeJS app, you might miss some slick ActiveRecord features, such as declarative validation. Diving into most of the basic tutorials out there, you'll find that many basic web development tasks are more work than you like. For example, if we borrow the style of http://howtonode.org/express-mongodb (http://howtonode.org/express-mongodb), a route that pulls a document by its ID will look something like this:

```
1  app.get('/document/:id', function(req, res) {
2    db.collection('documents', function(error, collection) {
3      collection.findOne({ _id : collection.db.bson_serializer.ObjectID.cr
4          function(error, document) {
5            if (error || !document) {
6              res.render('error', {});
7            } else {
8              res.render('document', { document : document });
9            }
10         });
11   });
12 });
```

In my first post (http://thecodebarbarian.wordpress.com/2013/04/29/easy-web-prototyping-with-mongodb-and-nodejs/) I touched on MongooseJS, a schema and usability wrapper for MongoDB in NodeJS. MongooseJS was developed by LearnBoost, an education startup based in San Francisco, and maintained by 10gen. MongooseJS lets us take advantage of MongoDB's flexibility and performance benefits while using development paradigms similar to Ruby on Rails and ActiveRecord. In this post, I'll go into more detail about how The Ascot Project uses Mongoose for our data, some best practices we've learned, and some pitfalls we've found that aren't clearly documented.

Before we dive into the details of working with Mongoose, let's take a second to define the primary objects that we will be using. Loosely speaking, Mongoose's schema setup is defined by 3 types: Schema, Connection, and Model.

A Schema is an object that defines the structure of any documents that will be stored in your MongoDB collection; it enables you to define types and validators for all of your data items.

A Connection is a fairly standard wrapper around a database connection.

A Model is an object that gives you easy access to a named collection, allowing you to query the collection and use the Schema to validate any documents you save to that collection. It is created by combining a Schema, a Connection, and a collection name.

Finally, a Document is an instantiation of a Model that is tied to a specific document in your collection.

Okay, now we can jump into the dirty details of MongooseJS. Most MongooseJS apps will start something like this:

```
var Mongoose = require('mongoose');
var myConnection = Mongoose.createConnection('localhost', 'mydatabase');

var MySchema = new Mongoose.schema({
    name : {
      type : String,
      default : 'Val',
      enum : ['Val', 'Valeri', 'Valeri Karpov']
    },
  created : {
      type : Date,
      default : Date.now
    }
});var MyModel = myConnection.model('mycollection', MySchema);
var myDocument = new MyModel({});
```

What makes this code so magical? There are 4 primary advantages that Mongoose has over the default MongoDB wrapper:

1. MongoDB uses named collections of arbitrary objects, and a Mongoose JS Model abstracts away this layer. Because of this, we don't have to deal with tasks such as asynchronously telling MongoDB to switch to that collection, or work with the annoying createFromHexString function. For example, in the above code, loading and displaying a document would look more like:

```
app.get('/document/:id', function(req, res) {
  Document.findOne({ _id : req.params.id }, function(error, document) {
    if (error || !document) {
      res.render('error', {});
    } else {
      res.render('document', { document : document });
    }
  });
```

```
9 | });
```

2. Mongoose Models handle the grunt work of setting default values and validating data. In the above example myDocument.name = 'Val', and if we try to save with a name that's not in the provided enum, Mongoose will give us back a nice error. If you want to learn a bit more about the cool things you can do with Mongoose validation, you can check out my blog post on how to integrate Mongoose validation with AngularJS (http://thecodebarbarian.wordpress.com/2013/05/12/how-to-easily-validate-any-form-ever-using-angularjs/).

3. Mongoose lets us attach functions to our models:

```
1 | MySchema.methods.greet = function() { return 'Hello, ' + this.name; };
```

4. Mongoose handles limited sub-document population using manual references (i.e. no MongoDB DBRefs), which gives us the ability to mimic a familiar SQL join. For example:

```
1  var UserGroupSchema = new Mongoose.schema({
2    users : [{ type : Mongoose.Schema.ObjectId, ref : 'mycollection' }]
3  });
4
5  var UserGroup = myConnection.model('usergroups', UserGroupSchema);
6  var group = new UserGroup({ users : [myDocument._id] });
7  group.save(function() {
8    UserGroup.find().populate('users').exec(function(error, groups) {
9      // Groups contains every document in usergroups with users field popu
10     console.log(groups[0][0].name)
11   });
12 });
```

In the last few months, my team and I have learned a great deal about working with Mongoose and using it to open up the true power of MongoDB. Like most powerful tools, it can be used well and it can be used poorly, and unfortunately a lot of the examples you can find online fall into the latter. Through trial and error over the course of Ascot's development, my team has settled on some key principles for using Mongoose the right way:

1 Schema = 1 file

A schema should never be declared in app.js, and you should never have multiple schemas in a single file (even if you intend to nest one schema in another). While it is often expedient to inline everything into app.js, not keeping schemas in separate files makes things more difficult in the long run. Separate files lowers the barrier to entry for understanding your code base and makes tracking changes much easier.

Mongoose can't handle multi-level population yet, and populated fields are not Documents. Nesting schemas is helpful but it's an incomplete solution. Design your schemas accordingly.

Let's say we have a few interconnected Models:

```
var ImageSchema = new Mongoose.Schema({
  url : { type : String},
  created : { type : Date, default : Date.now }
});
var Image = db.model('images', ImageSchema);

var UserSchema = new Mongoose.Schema({
  username : { type : String },
  image : { type : Mongoose.Schema.ObjectId, ref : 'images' }
});

UserSchema.methods.greet = function() {
  return 'Hello, ' + this.name;
};

var User = db.model('users', UserSchema);

var Group = new Mongoose.Schema({
  users : [{ type : Mongoose.Schema.ObjectId, ref : 'users' }]
});
```

Our Group Model contains a list of Users, which in turn each have a reference to an Image. Can MongooseJS resolve these references for us? The answer, it turns out, is yes and no.

```
Group.
  find({}).
  populate('user').
  populate('user.image').
  exec(function(error, groups) {
    groups[0].users[0].username; // OK
    groups[0].users[0].greet(); // ERROR - greet is undefined

    groups[0].users[0].image; // Is still an object id, doesn't get popu
    groups[0].users[0].image.created; // Undefined
  });
```

In other words, you can call 'populate' to easily resolve an ObjectID into the associated object, but you can't call 'populate' to resolve an ObjectID that's contained in that object. Furthermore, since the populated object is not technically a Document, you can't call any functions you attached to the schema. Although this is definitely a severe limitation, it can often be avoided by the use of nested schemas. For example, we can define our UserSchema like this:

```
var UserSchema = new Mongoose.Schema({
  username : { type : String },
  image : [ImageSchema]
});
```

In this case, we don't have to call 'populate' to resolve the image. Instead, we can do this:

```
Group.
  find({}).
  populate('user').
```

```
4     exec(function(error, groups) {
5       groups[0].users[0].image.created; // Date associated with image
6     });
```

However, nested schemas don't solve all of our problems, because we still don't have a good way to handle many-to-many relationships. Nested schemas are an excellent solution for cases where the nested schema can only exist when it belongs to exactly one of a parent schema. In the above example, we implicitly assume that a single image belongs to exactly one user – no other user can reference the exact same image object.

For instance, we shouldn't have UserSchema as a nested schema of Group's schema, because a User can be a part of multiple Groups, and thus we'd have to store separate copies of a single User object in multiple Groups. Furthermore, a User ought to be able to exist in our database without being part of any groups.

Declare your models exactly once and use dependency injection; never declare them in a routes file.

This is best expressed in an example:

```
1  // GOOD
2  exports.listUsers = function(User) {
3    return function(req, res) {
4      User.find({}, function(error, users) {
5        res.render('list_users', { users : users });
6      });
7    }
8  };
9
10 // BAD
11 var db = Mongoose.createConnection('localhost', 'database');
12 var Schema = require('../models/User.js').UserSchema;
13 var User = db.model('users', Schema);
14
15 exports.listUsers = return function(req, res) {
16   User.find({}, function(error, users) {
17     res.render('list_users', { users : users });
18   });
19 };
```

The biggest problem with the "bad" version of listUsers shown above is that if you declare your model at the top of this particular file, you have to define it in every file where you use the User model. This leads to a lot of error-prone find-and-replace work for you, the programmer, whenever you want to do something like rename the Schema or change the collection name that underlies the User model.

Early in Ascot's development we made this mistake with a single file, and ended up with a particularly annoying bug when we changed our MongoDB password several months later. The proper way to do this is to declare your Models exactly once, include them in your app.js, and pass them to your routes as necessary.

In addition, note that the "bad" listUsers is impossible to unit test. The User schema in the "bad" example is inaccessible through calls to require, so we can't mock it out for testing. In the "good" example, we can write a test easily using Nodeunit:

```
1  var UserRoutes = require('./routes/user.js');
2
3  exports.testListUsers = function(test) {
4    mockUser.collection = [{ name : 'Val' }];
5    var fnToTest = UserRoutes.listUsers(mockUser);
6    fnToTest( {},
7      { render : function(view, params) {
8          test.equals(mockUser.collection, params.users); test.done();
9        }
10     });
11 };
```

And speaking of Nodeunit:

Unit tests catch mistakes, encourage you to write modular code, and allow you to easily make sure your logic works. They are your friend.

I'll be the first to say that writing unit tests can be very annoying. Some tests can seem trivial, they don't necessarily catch all bugs, and often you write way more test code than actual production code. However, a good suite of tests can save you a lot of worry; you can make changes and then quickly verify that you haven't broken any of your modules. Ascot Project currently uses Nodeunit for our backend unit tests; Nodeunit is simple, flexible, and works well for us.

And there you have it! Mongoose is an excellent library, and if you're using MongoDB and NodeJS, you should definitely consider using it. It will save you from writing a lot of extra code, it'll handle some basic population, and it'll handle all your validation and object creation grunt work. This adds up to more time spent building awesome stuff, and less time trying to figure out how to get your database interface to work.

Have any questions about the code featured in this post? Want to suggest a better approach? Feel like telling me why the MEAN Stack is the worst thing that ever happened in the history of the world and how horrible I am? Go ahead and leave a comment below, or shoot me an email at valkar207@gmail.com (mailto:valkar207@gmail.com) and I'll do my best to answer any questions you might have. You can also find me on github at https://github.com/vkarpov15 (https://github.com/vkarpov15). My current venture is called The Ascot Project, and you can find that over at http://www.AscotProject.com (http://www.AscotProject.com). Huge thanks to my partner William Kelly (@idostartups) for all of his work helping me get this post together.

Bookmark the permalink.

6 thoughts on "Mistakes You're Probably Making With MongooseJS, And How To Fix Them"

1. Pingback: Introduction to the MEAN Stack, Part One: Setting Up Your Tools | The Code Barbarian

2. Pingback: Introduction to the MEAN Stack, Part Two: Building and Testing a To-do List | The Code Barbarian

   Panos says:
   on November 13, 2013 at 2:06 pm
   Nice article! Thanks for sharing your experience and knowledge. I recently dived into the "MEAN" stack, and your blog is my guide. Keep up!

   Reply
   vkarpov15 says:
   on November 13, 2013 at 3:05 pm
   Thanks! Glad you're finding it useful

   Reply
   Jonathan David (@JonathanDavid) says:
   on June 23, 2014 at 11:53 am
   Thank you for that great article.

   How would you combine omni-di with sinon.stub?
   I would like to test a controller which is injected with a Model (for example User).
   Following your example here:https://github.com/vkarpov15/lean-mean-nutrition-sample, I see that I would have to assemble the di in server_setup.js as follows:

```
di.assemble([
[
{ name : 'User', factory : require('./user.mock.js').MockUser }
]
]);
```

in the unit test I would want to do something like this:

```
methodStub = sinon.stub(User, 'find');
…
methodStub.yields(null, [{user1},{user2}]);
```

I cant get this to work, since di is using a factory method MockUser and the yields call is ineffective. Is there a way to enjoy both worlds?

thanks

Reply

vkarpov15 says:
on September 4, 2014 at 9:43 pm
You can specify `obj` instead of `factory`. When you specify factory, omni-di will try to call the factory as a function. When you specify an `obj`, omni-di will just use the provided object as is.

Reply