Mongoose Reintroduced

Published Jun 12, 2014

#TUTORIAL #MONGODB #MONGOOSE

TL;DR: An interactive introduction to Mongoose, the ODM for Node.js and MongoDB.

In the recent months, regular readers will have noticed that we have had a lot of Node.js based examples. One reader asked why we were only using the MongoDB driver rather than their preferred method of using MongoDB and Node.js, Mongoose. The answer for them is that we were sticking to the minimum supported APIs but it did remind us that we really should talk about object modelling with Node.js as it can make for easier to read and manage code.

Mongoose brings some schema-like discipline to the dynamic world of JavaScript by allowing a programmer to create schema and models which take care of the grind in creating CRUD and other applications. If you want to follow along, remember to `npm install mongoose`, start the Node REPL and you can either type in or copy and paste each code block over. With Mongoose, rather than managing your data as an informal map of keys and value it is formalised in a schema. Before we can do that, we should start node (by running `node`) and get connected to MongoDB:

```
$ node
> var mongoose=require("mongoose");
> mongoose.connect(process.env.MONGOHQ_URL);
```

This simply pulls in the mongoose package and opens a connection to the database URL in the `MONGOHQ_URL`. If you are following along in the Node REPL, you will notice that that `connect` method comes back with something like…

```
{ connections:
   [ { base: [Circular],
       collections: {},
       models: {},
       replica: true,
       hosts: [Object],
       host: null,
       port: null,
       user: 'user',
       pass: 'pass',
       name: 'database',
       options: [Object],
       otherDbs: [],
       _readyState: 2,
       _closeCalled: false,
       _hasOpened: false,
       _listening: false,
       _events: {},
       db: [Object] } ],
  plugins: [],
  models: {},
  modelSchemas: {},
  options: { pluralization: true } }
```

That's the value of the mongoose variable and as you move on you will find there's some quite a lot of input generated by Mongoose methods which you may find distracting. A little general tip for working in the REPL though is that if you know you want to ignore the value returned by a command in the Node REPL is to append `;0;` to the end which will act like an extra statement and just return 0. So setting up that query you would do `mongoose.connect(process.env.MONGOHQ_URL);0;`.

Back to Mongoose and the REPL. We said we want to formalise out data in a schema, so we shall do that now. Let's copy in this block...

```
var Schema=mongoose.Schema;

var robotSchema=new Schema({
  name        : { type:String, index: { unique: true } },
  description : String,
  occupation  : { type:String, default:"General Purpose" },
  age         : { type: Number, min:18, max:60 },
  dangerous   : Boolean,
  added       : { type: Date, default: Date.now },
});
```

Looking at the code in this first block, we get Mongoose's Schema class which can hold a template for how we want the data stored, how we want it treated in terms of types, which fields we want indexed and any validation we want to perform on it. Indexes and defaults can also be incorporated. And thats what we do when we create a robotSchema, with fields which are typed (as String, Number, Boolean and Date), defaults and one uniquely indexed field (name). Thats not all we can add to our schema. We can also add methods to the Schema like this:

```
robotSchema.methods.causeHavoc= function() {
  if(this.dangerous) {
    console.log(this.name+" will Crush! Destroy!");
  } else {
    console.log("The Three Laws control "+this.name);
  }
};
```

Once we have a complete Schema we can turn it into a model which will be able to enact the Schema's demands and include a method to causeHavoc...

```
var Robot = mongoose.model('Robot',robotSchema);
```

A quick note here; the code would have to be cut and pasted into the REPL every time you started a session to initialise the connection and the schema. But, if you copied all the commands into a file called, say, `setup.js` then you can load those commands into the Node REPL by using the .load command, `.load setup.js`. But it gets better as you don't even have to create that file. As you have got to this point in the REPL, you can save your current session with `.save setup.js` and it will create that file for you.

Back with Mongoose, now we can use our newly created model to make some robots.

```
var robbie= new Robot( { name: "Robbie", dangerous:false });
var bender= new Robot( { name: "Bender", occupation:"Bending", dange
```

These documents, `robbie` and `bender`, have all the needed methods to be saved and modified. Modifications go through any validations defined in the schema. All you need to save them is to call the save method...

```
[robbie, bender].forEach(function(robot) {
  robot.causeHavoc();
  robot.save(function(err) {
    if (err) {
      console.log(robot.name + " unsaved "+err);
      }
    })
});
```

If you run this code twice, you'd note that the second time it was run the bots would not be saved and errors would be printed. That is because they were still new instances with the same name as existing bots and hence causing a unique constraint failure.

Lets retrieve one of the robots written to the database. We will get a version of the data which we can safely modify and save:

```
Robot.findOne( { name:"Bender"}, function(err,bot) {
  bot.description="has shiny metal";
  bot.save(function(err){
    if (err) {
     console.log(bot.name + " unsaved " + err);
   } else {
     console.log(bot.name + " updated");
   }
  })
});
```

We just looked for one matching record in this case but Mongoose has a whole query syntax to explore. Let's make some more robots in the database to work with:

```
for(var i=0;i<10000; i++) {
  var thxbot=new Robot( {
    name:"THX-"+i,
    description: "A generic THX robot",
    age: Math.floor(Math.random()*42+18),
    dangerous: Math.random()<.5
  } );
  thxbot.save();
}
```

(Astute readers will notice that there's no callback in the save. Any errors that occur now will be emitted as error events on the connection and can be captured there. We've just dropped the callback code for readability)

When you use a query method (find, findById, count, update) without a callback you get a query object back:

```
var query=Robot.find( { dangerous:true });
```

That query for dangerous robots won't run till the exec method is called on it and therefore you can have further qualifiers added to it so that you can then say only list ten dangerous robots over the age of thirty and only retrieve their name, age and description:

```
  query.where('age').gt(30);
  query.limit(10);
  query.select("name description age");
```

Then you can go execute it and process the results in a callback:

```
  query.exec(function(err,results) {
    results.forEach(function(bot) {
      console.log(bot.name+"/"+bot.age+"/"+bot.description);
    })
  });
```

There is something we are ignoring in this session within the REPL and it's that our interaction at the prompt is slowing things down enough that we don't have to worry (too much) about the asynchronous nature of Node and Mongoose. When running code, rather than typing in the REPL, you will have to remember that, for example, if run an update and then a query, the update may not have finished before the query. Updates in Mongoose are a special case of query so this, which says to increment the age of every dangerous robot:

```
Robot.update({ dangerous: true }, { $inc: { age: 1 } }, { multi: tru
```

would actually do nothing except return a query object as above. You can force it to execute without a callback by running:

```
Robot.update({ dangerous: true }, { $inc: { age: 1 } }, { multi: tru
```

But if the next thing we run is a query on the age of dangerous robots...

```
Robot.find({ dangerous: true }, function(err, results) {
      results.forEach(function(bot) {
console.log("Bot " + bot.name + " is " + bot.age);
      });
    });
```

Then you may be querying while the robots are still being updated. Better to pop the query in the callback for the update which gets called when the update is complete:

```
Robot.update({ dangerous: true }, { $inc: { age: 1 } }, { multi: tru
  function(err, num, raw) {
    Robot.find({ dangerous: true }, function(err, results) {
      results.forEach(function(bot) {
console.log("Bot " + bot.name + " is " + bot.age);
      });
    });
  });
```

Mongoose also supports the idea of promises, an alternative to callbacks for managing asynchronous actions in code.

Another example – Mongoose does a good job hiding that it establishes its connection in the background, if you want to be sure you have connected before proceeding, you'll want to register for the `connected` event on the connection:

```
mongoose.connection.on("connected", function() {
... the rest of the code...
});
```

Finally, before you tire of typing in the Node REPL, why not define a general purpose function to print results. Take counting the number of robots we now have. We can do this...

```
Robot.count(function(err,result) { console.log(result); });
```

Or we can pre-define a function:

```
function pr(err,results) { console.log(results); };
```

and use that when we want to just print result:

```
Robot.count(pr);
```

Thats surprisingly flexible too...

```
Robot.find( ).where('age').gt(30).limit(5).sort('-age name').exec(pr
```

Prints us five robots over the age of thirty sorted in descending age and ascending name.

As you can see, there is a lot to Mongoose and a lot more to explore such as middleware and plugins to make your models more potent, a lot more to the query syntax and schema definitions. But now you should be ready to start exploring its capabilities through the Node.JS REPL. You'll want work through the guide and make sure to have the API documentation to hand and you'll be well on your way to better managed data in your applications without giving up the power and flexibility of MongoDB. Try it out with a MongoHQ hosted database today.

Dj Walker-Morgan is Compose's resident Content Curator, and has been both a developer and writer since Apples came in II flavors and Commodores had Pets. Love this article? Head over to Dj Walker-Morgan's author page to keep reading.

RELATED ARTICLES

OCT 23, 2011

Introduction to Cloud9 & MongoDB

If you are not familiar, Cloud9 is a fully online editor plus PaaS (platform as a service) for building node.js projects. The...

Eric Redmond

MAY 18, 2018

NewsBits - Mongoose gets maps

Welcome to NewsBits where you'll find the database, and developer news from around the net for the week ending May 18th: Map...

Dj Walker-Morgan

JAN 19, 2018

Mongoose 5.0 emerges - NewsBits

Welcome to NewsBits where you'll find the database, cloud, and developer news from around the net for the week ending January...

Dj Walker-Morgan