

01/31/2019 • 14 minutes to read

In this article

[MongooseJS: Getting Started](#)

[Mongoosing a Person](#)

[Mongoosing in Action](#)

[Mongoose Validation](#)

[Mongoose Methoding](#)

[Mongoose Versioning](#)

[Wrapping Up](#)

March 2016

Volume 31 Number 3

[The Working Programmer] How To Be MEAN: Robust Validation with MongooseJS

By [Ted Neward](#) | March 2016



In my February 2016 column (msdn.com/magazine/mt632276), I transitioned to a MongoDB database. Doing so wasn't that hard thanks to the JSON-based nature of Node.js and the JSON-based nature of MongoDB. (Life is always easier when working with data if the transition is from apples to apples). MongoDB has a great advantage in that it will "scale up" and "scale out" easily, not to mention that it's trivial to get started. But it also has a significant drawback: Because MongoDB is a "schemaless" database (in that the schema is already predefined—a database holds collections, collections hold documents and documents basically are just JSON objects), within it lies the seeds of its own destruction.

First, recall that MongoDB queries are essentially query documents (that first parameter to the find call), containing the fields by which to scan the collection for matches. So if

the query `{'fristName': 'Ted'}` gets executed against the `"persons"` collection in the existing database, nothing will come back—the field name in the query document is misspelled (`"fristName"` instead of `"firstName"`), so it will match against no documents in the collection. (Unless there's a typo in the document in the collection, of course.) This is one of the biggest disadvantages of a schemaless database—a simple typo in the code or user input can create accidental bugs of the most furious head-scratching nature. This is where it would be nice to get some language support, whether that's by a compiler or an interpreter.

Second, notice that the code displayed in my last column is reminiscent of the `"two-tier"` applications that were popular for two decades during the `"client-server"` era of computing. There's a code layer that takes input directly from the user (or, in this case, from the API consumer), applies it and gets a simple data structure back from the database, which it hands directly back to the caller. There's certainly no sense of `"object-orientation"` in that code. While not a deal-breaker, it would be nice if we could get a stronger sense of `"object-ness"` to the server-side code, such that some validation of various properties could be centralized in one place, for example. Case in point: Is it legal for a person to have an empty `firstName` or `lastName`? Can he be doing absolutely anything in his status? Is there a `"default"` status, if he chooses not to provide one, or is an empty status acceptable?

The Node.js crowd has wrestled with these problems for quite a while (it was one of the first language ecosystems to adopt MongoDB on a large-scale basis) and, not surprisingly, it has come up with an elegant solution to the problem, called MongooseJS. It's a software layer that sits `"on top"` of MongoDB and provides not only a schema-like language-verified validation layer, but also an opportunity to build a layer of `"domain object"` into the server-side code. Hence, it's sort of `"the other 'M'"` in the MEAN stack.

MongooseJS: Getting Started

By now, the exercise should be getting pretty simple, straightforward and repetitive: `"Which 'thing' do you npm this time?"` The short answer is `"npm install --save mongoose,"` but if there's ever some confusion as to what the exact package might be (the Node folks tend to waffle between `"thing"` and `"thingjs"` as package names), an `"npm find"` or `"npm search"` will search the npm registry for packages that match whatever terms follow on the command line. Alternatively, typing `"Mongoose JS"` into the search engine of choice will yield up the Mongoose Web site (mongoosejs.com), which will have the correct npm incarnation, along with a ton of documentation on how to use Mongoose. (This makes it a handy thing to have bookmarked in the browser, for sure.)


Once installed, you can begin to define Mongoose “schema” objects, which will define the kind of objects to be stored in the MongoDB collection.

Mongoosing a Person

Mongoose uses some interesting terminology for what’s essentially a two-step process to defining a JavaScript object model on top of the MongoDB database API. First, we define a “schema,” which looks like a traditional class from a more traditional class-based language (C#, C++, Java or Visual Basic). This schema will have fields, define types for these fields and optionally include some validation rules around the fields for when assigning values to the fields. You can also add some methods, instance or static, which I’ll get to later. Then, once the schema object is defined, you “compile” it into a Model, which is what will be used to construct instances of these objects.

Take careful note here: This is still JavaScript, and more important, this is the version of JavaScript more accurately described as ECMAScript 5, which lacks any concept of “class” whatsoever. Therefore, even though the Mongoose approach creates the illusion that you’re defining a “class,” it’s still the prototype-based language that you know and love (or loathe). This is partly the reason for this two-step process—the first to define an object that will serve as the class, the second to define an object that will implicitly work as the “constructor” or “factory,” subject to the JavaScript/ECMAScript 5 rules around the “new” operator.

So, translating that into code, after “require()”ing the Mongoose library into the usual “mongoose” local variable at the top of the JavaScript code, you can use that to define a new Schema object:

JavaScript	 Copy
<pre>// Define our Mongoose Schema var personSchema = mongoose.Schema({ firstName: String, lastName: String, status: String }); var Person = mongoose.model('Person', personSchema);</pre>	


There’s a variety of things you can do with the fields in the personSchema, but for starters, let’s keep it simple; this will not only help get to working code faster, but will also highlight a few niceties about Mongoose along the way. Also, again, notice the two-step process by which the model is defined: first, you define the schema, using the Schema function/constructor, and then pass the schema object into the model function/constructor, along with the name of this model. Convention among Mongoose

users is that the returned object from the model call will be the same as the type being defined because that's what will help give it the illusion of being a class.

Mongoosing in Action

Having defined the model, now the only thing left to do is to rewrite the various route methods that will use the model. The easiest place to start is the `getAllPersons` route, as shown in **Figure 1**, because it returns the entire collection from the database, with no filters.


Figure 1 Rewriting the `getAllPersons` Route

JavaScript	 Copy
<pre>var getAllPersons = function(req, res) { Person.find(function(err, persons) { if (err) { debug("getAllPersons--ERROR:", err); res.status(500).jsonp(err); } else { debug("getAllPersons:", persons); res.status(200).jsonp(persons); } }); };</pre>	

Notice how the code is fundamentally similar to what was there before—the query is executed and invokes the passed-in callback function (again, with the convention “err, result” as its parameters) when the query is complete. However, now Mongoose provides a tiny bit more structure around the access by routing it through the `Person` model object, which will yield all kinds of powerful benefits, as you’ll see in a minute.

Next up is the `personId` middleware, because it’s used in almost all the rest of the routes, as shown in **Figure 2**.

Figure 2 Rewriting the `personId` Route

JavaScript	 Copy
<pre>app.param('personId', function (req, res, next, personId) { debug("personId found:", personId); if (mongoose.ObjectId.isValid(personId)) { Person.findById(personId) .then(function(person) { debug("Found", person.lastName); req.person = person; }); } next(); });</pre>	

```
        next();
    });
}
else {
    res.status(404).jsonp({ message: 'ID ' + personId + ' not found' });
}
});
```

Again, this is a middleware function, so the goal is to find the Person object out of the collection and store it into the request object (req). But notice how after validating that the incoming personId is a valid MongoDB ObjectId/OID; the Person object shows up again, this time calling findById, passing in the ObjectId/OID passed in. Of most interest here is that Mongoose supports the “promise” syntax/style that will be a fixture in future versions of JavaScript—the returned object from findById is a Promise object, upon which you can invoke “then” and pass in a callback to be executed when the query is finished configuring.

This two-step approach then lets you do a whole host of things with this query before its execution, such as sort the query results in a particular order based on fields within the results, or select only a subset of the fields (so as to hide any sensitive information the client shouldn’t receive). These would be method calls wedged right in between find and then in a fluid style, such as:

JavaScript

 Copy

```
Person.find({ })
    .sort({ 'firstName': 'asc', 'lastName': 'desc' })
    .select('firstName lastName status')
    .then(function(persons) {
        // Do something with the returned persons
    });
```

In this case, this would sort the results by firstName in ascending order, sort the results by lastName in descending order (not that doing that makes much sense) and then strip out anything except the “firstName,” “lastName” and “status” fields. The full list of query “modifiers” is impressive and includes a number of useful methods such as limit (cut off at a certain number of results), skip (to skip past the first n results returned), count (to return an aggregate count of the documents returned) and so on.

Before you get too distracted by that, however, I’ll round out the rest of the route methods.

Having used the middleware to obtain the Person object in question for update and delete, those become pretty straightforward uses of the save and delete methods provided by Mongoose on the objects themselves. As shown in **Figure 3**, inserting a

new Person just requires instantiating a new Person model and using the save method on it.

Figure 3 Instantiating a New Person Model

JavaScript  Copy

```
var updatePerson = function(req, res) {
  debug("Updating", req.person, "with", req.body);
  _.merge(req.person, req.body);
  // The req.person is already a Person, so just update()
  req.person.save(function (err, person) {
    if (err)
      res.status(500).jsonp(err);
    else {
      res.status(200).jsonp(person);
    }
  });
};

var insertPerson = function(req, res) {
  var person = new Person(req.body);
  debug("Received", person);
  person.save(function(err, person) {
    if (err)
      res.status(500).jsonp(err);
    else
      res.status(200).jsonp(person);
  });
};

var deletePerson = function(req, res) {
  debug("Removing", req.person.firstName, req.person.lastName);
  req.person.delete(function(err, result) {
    if (err) {
      debug("deletePerson: ERROR:", err);
      res.status(500).jsonp(err);
    }
    else {
      res.status(200).jsonp(req.person);
    }
  });
};
```

Conceptually, it looks a lot like the previous, non-Mongoose version, but there's a really important, if subtle, change: logic about Person is now being more localized to the Person model (and Mongoose's own persistence implementation).


Mongoose Validation

Now, just for grins, you want to add several new rules to the application: neither firstName nor lastName can be empty and status can only be one of several possible values (a la an enumerated type). This is where Mongoose's ability to create a domain

object model within the server side can be particularly powerful, because classic O-O thinking holds that these kinds of rules should be encapsulated within the object type itself and not the surrounding usage code.


With Mongoose, this is actually somewhat trivial. Within the schema object, the fields go from being simple name: type pairs to more complex name: object descriptor pairs and validation rules can be specified in these object descriptors. In addition to the usual raft of numeric validation (min and max) and String validation (min length and max length), you can specify an array of acceptable values for the "status" field and define a default value for each field when none is specified, which (not surprisingly) neatly fits the new requirements, as shown in **Figure 4**.

Figure 4 Mongoose Validation

JavaScript	 Copy
<pre>// Define our Mongoose Schema var personSchema = mongoose.Schema({ firstName: { type: String, required: true, default: "(No name specified)" }, lastName: { type: String, required: true, default: "(No name specified)" }, status: { type: String, required: true, enum: ["Reading MSDN", "WCFing", "RESTing", "VBing", "C#ing"], default: "Reading MSDN" }, }); var Person = mongoose.model('Person', personSchema);</pre>	

Nothing else needs to change anywhere else in the code base—all of these rules about Person-ness are captured exactly where they should be, in the definition of the schema object.


If you try to now insert JSON, that doesn't obey the right list of statuses, such as:

JavaScript	 Copy
------------	--


```
{
  "firstName": "Ted",
  "lastName": "Neward",
  "status": "Javaing"
}
```

The save method called inside of the insertPerson route will yield a 500 response, with the returned JSON body reading as shown in **Figure 5**.

Figure 5 Error Result from Failing Validation on a Save

JavaScript	 Copy
<pre>{ "message": "Person validation failed", "name": "ValidationError", "errors": { "status": { "properties": { "enumValues": ["Reading MSDN", "WCFing", "RESTing", "VBing", "C#ing"], "type": "enum", "message": "`{VALUE}` is not a valid enum value for path `{PATH}`.", "path": "status", "value": "Javaing" }, "message": "`Javaing` is not a valid enum value for path `status`.", "name": "ValidatorError", "kind": "enum", "path": "status", "value": "Javaing" } } }</pre>	

That pretty much nails what went wrong there.

Mongoose Methoding

Of course, object-orientation means combining state and behavior, which means that these domain objects won't really be objects unless you can attach methods to either object instances or the "class" as a whole. Doing so is actually fairly simple: You call a Mongoose method (either method for a traditional instance-based method or static for something class-based) that will define the method, and pass in the function to use as the method body, like so:

JavaScript

 Copy

```
personSchema.method('speak', function() {  
  console.log("Don't bother me, I'm", status);  
});
```

It's not exactly like writing a class in C#, but it's pretty close. Note that along with all the other changes to the Schema objects, these must all be done before the Schema is compiled into the model object, so this call has to appear before the `mongoose.model` call.

Mongoose Versioning

By the way, if you do a quick GET to `/persons` again, the resulting output contains something a little unexpected:

JavaScript

 Copy

```
[{"_id": "5681d8bfddb73cd9ff445ec2",  
  "__v": 0,  
  "status": "RESting",  
  "lastName": "Castro",  
  "firstName": "Miguel"}]
```

The `"__v"` field, which Mongoose silently slipped into the mix (and preserves all the way to the database) is a versioning field, known within Mongoose as the `versionKey` field, and it helps Mongoose recognize changes to the document; think of it as a version number on a source code file, as a way of detecting simultaneous changes. Normally, this is just an internal detail of Mongoose, but it can be a little surprising to see it show up there the first time.

Nevertheless, that raises a more interesting question: It's not uncommon for systems to want to track when a document was created or when changes were made to a document, and it would certainly be a pain to have to fill in fields every time any part of the surrounding code touched or saved one of these guys.

Mongoose can help with that, by letting you "hook" certain lifecycle methods to update fields right before the document is written to MongoDB, regardless of the call that's persisting it, as shown in **Figure 6**.

Figure 6 Hooking Lifecycle Methods with Mongoose

JavaScript

 Copy

```
// Define our Mongoose Schema
var personSchema = mongoose.Schema({
  created: {
    type: Date,
    default: Date.now
  },
  updated: {
    type: Date,
  },
  // ... as before
});
personSchema.pre('save', function(next) {
  // Make sure updated holds the current date/time
  this.updated = new Date();
  next();
});
var Person = mongoose.model('Person', personSchema);
```

Now, whenever a Person object is constructed, it will default the created field to hold the current date/time and the updated field will be set to the current date/time just prior to being sent to MongoDB for storage. Mongoose calls these “middleware,” because they are in spirit to the middleware functions defined by Express, but make no mistake, these are specific and contained entirely to the Mongoose-defined object.

Now, whenever a Person is created or updated, it’ll have those corresponding fields filled in and updated as necessary. Moreover, should something more sophisticated (like a full-blown audit log) be necessary, it’s fairly easy to see how this could be added. Instead of created/updated fields, there’s an auditLog field, which would be an array, and the “save” hook would simply append over and over to that array to describe what was done each time and/or who did it, depending on requirements.

Wrapping Up

This has been another relatively heavy piece, and certainly there’s a lot more of Mongoose that’s worth exploring. However, that’s half the fun of exploring a new technology stack and platform, and it would be extremely churlish of me not to include that kind of fun to my loyal readership. The key thing to realize is that this pairing of Mongoose + MongoDB provides a powerful combination: the schemaless nature of the MongoDB database is coupled with language-enforced validation of basic types, plus runtime-enforced validation of data values to give us something of the best of both the statically typed and dynamically typed worlds. It’s not without problems and rough spots, but overall, it’s hard for me to imagine doing any serious coding work in MongoDB without something like Mongoose keeping me from doing stupid things.

I'm almost done with the server-side of things, but I'm out of space, so for now ... happy coding!

Ted Neward is a Seattle-based polytechnology consultant, speaker and mentor. He has written more than 100 articles, is an F# MVP, INETA speaker, and has authored and coauthored a dozen books. Reach him at ted@tedneward.com if you're interested in having him come work with your team, or read his blog at blogs.tedneward.com .

Thanks to the following technical expert for reviewing this article: Shawn Wildermuth