

01/31/2019 • 13 minutes to read

## In this article

[Hu-MongoDB-ous](#)

[Data Design](#)

[Data Access](#)

[Data Location](#)

[MongoDB + Node.js](#)

[Insert](#)

[Retrieve All](#)

[Middleware](#)

[Retrieve One, Delete and Update](#)

[Wrapping Up](#)

---

February 2016

Volume 31 Number 2

# [The Working Programmer] How to Be MEAN: Inside MongoDB

By [Ted Neward](#) | February 2016



Welcome back, “MEANies.” (I decided that sounds better than “Nodeists,” and, besides, starting with this piece, we’re moving past just Node.)

The code has reached an important transition point—now that there are some tests in place to verify that functionality remains the same, it’s safe to begin some serious refactoring. In particular, the current in-memory data storage system might be nice for quick demos, but it’s not going to scale up particularly well over time. (Not to mention all it takes is one VM reboot/restart, and you’ve lost all the data that’s not hardcoded in the startup code.) It’s time to explore the “M” in MEAN: MongoDB.

# Hu-MongoDB-ous

First off, it's important to recognize that according to popular lore, MongoDB actually gets its name from the word "humongous." Whether that bit of Internet trivia is true or not, it serves to underscore that MongoDB isn't built to provide the exact same feature set as your average relational database. To Mongo, scalability ranks high, and toward that end, MongoDB is willing to sacrifice a little consistency, trading off ACID transaction capabilities across the cluster in favor of "eventual consistency."

This system might not ever reach the scale of bazillions of records; in fact, I'd be quite shocked if it ever came within shouting distance of needing it. However, MongoDB has another feature that ranks equally high on the "this is worth exploring" scale of intrigue and that's its data model: It's a document-oriented database. This means that instead of the traditional relational tables-and-columns schema-enforced model, MongoDB uses a data model of "schemaless" documents gathered into collections. These documents are represented in JSON, and as such each document is made up of name-value pairs, where the values can be traditional data types (strings, integers, floating-point values, Booleans and so on), as well as more "composite" data types (arrays of any of the data types just listed, or child objects that in turn can have name-value pairs). This means, offhand, that the data modeling will be a bit different than you might expect if your only experience is with a relational database; this application is small enough now that these differences won't be very overt, but it's something to keep in mind when working with more complex storage needs.

Note: For a deeper look at MongoDB from a .NET developer's perspective, check out this column's 201 three-part series on MongoDB ([bit.ly/1J7DjOB](https://bit.ly/1J7DjOB) ).

## Data Design


From a design perspective, seeing how the "persons" data model will map against MongoDB is straightforward: there will be a "persons" collection and each document inside that will be a JSON-based bundle of name-value pairs and so on.

And that's pretty much it. Seriously. This is part of the reason that document-oriented databases are enjoying such favor in the development community—the startup curve to getting data into them is ridiculously low, compared to their schema-based relational counterparts. This has its own drawbacks, too, of course—one typo and suddenly all the queries that are supposed to be based on "firstName" are suddenly coming back empty, because no document has a field "firstName"—but I'll look at a few ways to mitigate some of these later.

For now, let's look at getting some data into and out of MongoDB.

# Data Access

The first step is to enable the application to talk to MongoDB; that involves, not surprisingly, installing a new npm package called "mongodb." So, by now, this exercise should seem almost automatic:

JavaScript	 Copy
<pre>npm install --save mongodb</pre>	

The npm tool will churn through its usual gyrations and when it returns, the Node.js MongoDB driver is installed into the node\_modules directory. If you get a warning from npm about a kerberos package not being installed ("mongodb-core@1.2.28 requires a peer of kerberos@~0.0"), this is a known bug and seems fixable by simply installing kerberos directly via npm ("npm install kerberos"). There shouldn't be any problems beyond that, but of course, this is all subject to the next release of any of these packages—such is the joy of developing on the bleeding edge.

Next, the code will need to open a connection to the MongoDB instance. Where the instance resides, however, deserves a little discussion.

## Data Location

As mentioned in the first article in this series, there's two easy options for MongoDB: one is to run it locally, which is great for the development experience but not so good for the production experience; and the other is to run it in the cloud, which is great for the production experience but not for development. (If I can't run the code while I'm on an airplane on my way to a conference, then it's not a great developer experience, in my opinion.) This is not an unusual state of affairs and the solution here is very much the same as it would be for any application: run it locally during development and from the cloud in production or testing.

Like most databases, connecting to MongoDB will require a server DNS name or IP address, a database name and (optionally) a port to use. Normally, in development, this will be "localhost," the database name and "27017" (the MongoDB default), but the settings for a MongoDB instance in the cloud will obviously be different than that. For example, the server and port settings for my Mongolab MongoDB instance called "msdn-mean" are "ds054308.mongolab.com" and "54308," respectively.

The easiest way to capture this divergence in the Node world is to create a standalone JS file (typically called config.js) and require it into the app.js code, like so:

	 Copy
--	--

```
JavaScript
// Load modules
var express = require('express'),
    bodyParser = require('body-parser'),
    debug = require('debug')('app'),
    _ = require('lodash');
// Go get your configuration settings
var config = require('./config.js');
debug("Mongo is available at", config.mongoServer, ":", config.mongoPort);
// Create express instance
var app = express();
app.use(bodyParser.json());
// ... The rest as before
```

What remains, then, is for the config file to determine the environment in which this application is running; the usual way to do this in the Node.js environment is to examine an environment variable, “ENV,” which will be set to one of “prod,” “dev,” or “test” (if a third, QA-centric, environment is in place). So the config code needs to examine the ENV environment variable and put the right values into the exported module object:

JavaScript

 Copy

```
// config.js: Configuration determination
//
var debug = require('debug')('config');
debug("Configuring environment...");
// Use these as the default
module.exports = {
  mongoServer : "localhost",
  mongoPort : "27017"
};
if (process.env["ENV"] === "prod") {
  module.exports.mongoServer = "ds054308.mongolab.com";
  module.exports.mongoPort = "54308";
}
```

Note the use of the “process” object—this is a standard Node.js object, always implicitly present inside any Node.js-running application, and the “env” property is used to look up the “ENV” environment variable. (Sharp readers will note that the ExpressJS code does exactly the same thing when deciding what port to use; you could probably refactor that snippet to use the config.js settings, as well, but I’ll leave that as an exercise to you, the reader.)

So far, so good. Actually, better; this has also implicitly created a nice separation of configuration code away from the main code base.

Let’s start adding and removing data.

# MongoDB + Node.js

Like most databases, you need to open a connection to MongoDB, hold on to that object, and use that for subsequent actions against the database. Thus, it would seem an obvious first step would be to create that object as the application is starting up and store it globally, as shown in **Figure 1**.

Figure 1 Creating an Object in MongoDB

JavaScript  Copy

```
// Go get your configuration settings
var config = require('./config.js');
debug("Mongo is available at ", config.mongoServer, ":", config.mongoPort);
// Connect to MongoDB
var mongo = null;
var persons = null;
var mongoURL = "mongodb://" + config.mongoServer +
    ":" + config.mongoPort + "/msdn-mean";
debug("Attempting connection to mongo @", mongoURL);
MongoClient.connect(mongoURL, function(err, db) {
    if (err) {
        debug("ERROR:", err);
    }
    else {
        debug("Connected correctly to server");
        mongo = db;
        mongo.collections(function(err, collections) {
            if (err) {
                debug("ERROR:", err);
            }
            else {
                for (var c in collections) {
                    debug("Found collection", collections[c]);
                }
                persons = mongo.collection("persons");
            }
        });
    }
});
// Create express instance
var app = express();
app.use(bodyParser.json());
// ...
```

Notice that the connect call takes a URL, and a callback—this callback takes an error object and the database connection object as its parameters, as is the Node.js convention. If the first is anything but undefined or null, it's an error, otherwise everything went swimmingly. The URL is a MongoDB-specific URL, using the "mongodb" scheme, but otherwise looking very much like a traditional HTTP URL.

However, there's a subtlety to this code that may not be apparent at first: The callback is invoked at some point well after the rest of the startup code completes, which becomes more obvious when you look at the debug-printed output, as shown in **Figure 2**.

 Debug Printed Output

### Figure 2 Debug Printed Output

See how the "Example app listening" message appears before the "Connected correctly to server" message from the callback? Given that this is happening on application startup, this concurrency issue isn't critical, but it's not going away, and this is, without question, one of the trickiest parts of working with Node.js. It's true that your Node.js code will never be executed simultaneously on two threads at the same time, but that doesn't mean you won't have some interesting concurrency problems going on here; they just look different than what you're used to as a .NET developer.

Also, just as a quick reminder, when this code is first run against a brand-new MongoDB database, the collections loop will be empty—MongoDB won't create the collections (or even the database!) until it absolutely has to, which usually occurs when somebody writes to it. Once an insert is done, then MongoDB will create the necessary artifacts and data structures to store the data.

Regardless, for the moment, we have a database connection. Time to update the CRUD methods to start using it.

## Insert

The `insertPerson` will use the `insert` method on the MongoDB collection object and, again, you need a callback to invoke with the results of the database operation:

JavaScript

 Copy

```
var insertPerson = function(req, res) {
  var person = req.body;
  debug("Received", person);
  // person.id = personData.length + 1;
  // personData.push(person);
  persons.insert(person, function(err, result) {
    if (err)
      res.status(500).jsonp(err);
    else
      res.status(200).jsonp(person);
  });
};
```

Notice the commented-out code (from the in-memory database version I'm migrating away from); I left it there specifically to prove a point. MongoDB will create an identifier

field, "\_id," that's the primary key for the document in the database, so my incredibly lame homegrown "id" generator code is not only no longer necessary, but entirely unwanted.

Also, notice that the last statement in the function is the insert method, with the associated callback. While it isn't necessary that this be the last statement in the function block, it's critical to understand that the insertPerson function will terminate before the database insert completes. The callback-based nature of Node.js is such that you don't want to return anything to the caller until you know the success or failure of the database operation—hence the calls to "res" don't happen anywhere outside the callback. (Skeptics should convince themselves of this by putting a debug call after the persons.insert call, and another one in the callback itself, and see the first one fire before the callback does.)

## Retrieve All

Inserts require validation, so while I'm here, I'll refactor getAllPersons, which just needs a quick query to the collection to find all the documents in that collection:

JavaScript

 Copy

```
var getAllPersons = function(req, res) {
  persons.find({}).toArray(function(err, results) {
    if (err) {
      debug("getAllPersons--ERROR:", err);
      res.status(500).jsonp(err);
    }
    else {
      debug("getAllPersons:", results);
      res.status(200).jsonp(results);
    }
  });
};
```

Before moving on, there's a couple of quick things to note: First, the find call takes a predicate document describing the criteria by which you want to query the collection, which in this case, I leave as empty; if this were a query by first name, that predicate document would need to look something like:

JavaScript

 Copy

```
"{ 'firstName': 'Ted' }"
```


Second, notice that the returned object from find isn't an actual result set yet, hence the need to call toArray to convert it into something of use. The toArray takes a callback

and, again, each branch of the callback must make sure to communicate something back to the caller using `res.status().jsonp`.

## Middleware

Before I can go on, recall from my previous columns that the `getPerson`, `updatePerson` and `deletePerson` functions all depend on the `personId` middleware function to look up a person by identifier. This means that that middleware needs to be updated to query the collection by its `_id` field (which is a MongoDB ObjectId, not a string!), instead of looking in the in-memory array, as shown in **Figure 3**.

Figure 3 Updating Middleware to Query the Collection

JavaScript	 Copy
<pre>app.param('personId', function (req, res, next, personId) {   debug("personId found:", personId);   if (mongodb.ObjectId.isValid(personId)) {     persons.find({"_id": new mongodb.ObjectId(personId)})       .toArray(function (err, docs) {         if (err) {           debug("ERROR: personId:", err);           res.status(500).jsonp(err);         }         else if (docs.length &lt; 1) {           res.status(404).jsonp(             { message: 'ID ' + personId + ' not found' });         }         else {           debug("person:", docs[0]);           req.person = docs[0];           next();         }       });   }   else {     res.status(404).jsonp({ message: 'ID ' + personId + ' not found' });   } });</pre>	

The MongoDB Node.js driver documents a `findOne` method, which would seem to be more appropriate, but the driver documentation notes that as a deprecated method.

Notice that the middleware, if it gets an invalid ObjectId, doesn't call `next`. This is a quick way to save some lines of code in the various methods that are depending on finding persons from the database, because if it's not a legitimate ID, it can't possibly be there, so hand back a 404. The same is true if the results have zero documents (meaning that ID wasn't in the database).



# Retrieve One, Delete and Update

Thus, the middleware makes `getPerson` trivial, because it handles all the possible error or document-not-found conditions:

JavaScript

 Copy

```
var getPerson = function(req, res) {  
  res.status(200).jsonp(req.person);  
};
```

And `deletePerson` is almost as trivial:

JavaScript

 Copy

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

Both of which make `updatePerson` pretty predictable:

JavaScript

 Copy

```
var updatePerson = function(req, res) {  
  debug("Updating", req.person, "with", req.body);  
  _.merge(req.person, req.body);  
  persons.updateOne({"_id":req.person._id}, req.person, function(err,  
result) {  
    if (err)  
      res.status(500).jsonp(err);  
    else {  
      res.status(200).jsonp(result);  
    }  
  });  
};
```

The `merge` call, by the way, is the same `Lodash` function used before to copy the properties from the request body over to the person object that was loaded out of the database.

# Wrapping Up

Wow. This has been a little heavier than some of the others in the series, but at this point, I have code that completely runs now against a MongoDB database, instead of the in-memory array I'd been using during the mocking out. But it's not perfect, not by a long shot. For starters, any typos in the code around those query predicates will create unanticipated runtime errors. More important, as .NET developers, we're accustomed to some kind of "domain object" to work with, particularly if there's some sort of validation on the various properties of the object that need to be done—it's not a good idea to spread that validation code throughout the Express parts of the code base. That's on the docket for next time. But for now ... happy coding!

---

**Ted Neward** *is the CTO of iTrellis, a Seattle-based polytechnology consulting firm. He has written more than 100 articles, is an F# MVP, INETA speaker, and has authored or co-authored a dozen books. Reach him at [ted@tedneward.com](mailto:ted@tedneward.com) if you're interested in having him come work with your team, or read his blog at [blogs.tedneward.com](http://blogs.tedneward.com) .*

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