

08/04/2015 • 9 minutes to read

In this article

[Get MEAN](#)

[Install Node.js](#)

[Install the NPM](#)

[Install MongoDB](#)

[Wrapping Up](#)

August 2015

Volume 30 Number 8

The Working Programmer - How To Be MEAN: Getting Started

By [Ted Neward](#) | August 2015



Parallel universes are all the rage, especially if you follow the stories in comic books. Clearly, I've stepped into one myself. When I first started reading this magazine, it was called Microsoft Systems Journal (MSJ for short). The languages of choice were C++ and Visual Basic. The underlying technology platform was a managed platform (COM) written with both unmanaged languages like C++ and managed languages like Visual Basic. The OS of choice was clearly, unfailingly, unquestionably and unhesitatingly Windows. Perhaps it was Windows 3.0, 3.1, Chicago or NT—but always Windows.

Look where we are today. It's clearly not a "Microsoft" world anymore, even within the Microsoft world. Forget, for a moment, the "competitors" against which Microsoft wrestles, such as Java or Ruby. The company has partnerships with those languages, and even supports them natively on the next-generation Microsoft Azure platform. Forget the languages that Microsoft "owns," like C# or Visual Basic or F#. They've all been made open source. As has its Web framework, and its data-access framework. And the new version of Visual Studio will ship with an Android emulator.

And just when we thought it couldn't get more different, Microsoft again went and did something entirely different. As of this writing, Microsoft just announced a partnership with Cyanogen, an Android distributor. Can somebody please tell me which portal I stepped through?

In the comics, whenever a hero steps into a new universe, there's a moment when the hero has to embrace the new world or risk being identified as the outsider—and possibly locked away in a sanitarium. Unless you fancy the sanitarium, you probably should embrace this new world.

Get MEAN

Let's talk about Node.js. Or, more precisely, let's talk about one of the favored software stacks for the Node.js platform: MEAN (MongoDB, Express, AngularJS, Node.js). It's quickly becoming one of the key players in the new technology world. Microsoft's support for Node.js and MongoDB on Azure (not to mention the fact you can easily run these on a standard Windows machine on-premises) means—if you'll pardon the pun—it's something every Microsoft developer should know.

You can do MEAN a couple of ways, including using Visual Studio. Start off by "embracing the Node Way." Use just the preferred set of Node-ish tools: a text editor, the command line and (if you have one lying around) a Mac. That's what I'll be using to start, although most of the command-line commands will be pretty adjustable to Windows without difficulty, it will be different.

Before I get too far into this parallel universe, take a look at the major players. The MEAN stack is a "full stack" quartet, meaning it covers front end, back end and storage. Starting from the front end, AngularJS provides a complete Single-Page Application Web client framework, complete with Model-View-Controller abstractions and two-way binding for the UI. AngularJS resides entirely on the front end, though, and requires a back end with which to communicate. This typically uses Web API calls. These are also known in some circles as RESTful endpoints, although that can lead to zealous debates about what REST is, so let's leave it as Web APIs for now.

Those Web API endpoints are built with the Express framework sitting on top of the Node.js platform. This is much the same way ASP.NET Web API sits on top of the ASP.NET pipeline and the Microsoft .NET Framework. These back-end Web APIs will clearly need a database for storage, which is where MongoDB comes into play. MongoDB is a schema-less, document-oriented data store (in contrast to SQL Server, which is a schemaed, relational-oriented data store) with some built-in sharding and map-reduce capabilities.

It may have already occurred to you these three parts—front end, back end and storage—are actually pretty interchangeable. For example, it wouldn't be hard to imagine using AngularJS to talk to ASP.NET Web APIs that in turn talk to MongoDB. Or to use a Windows Forms application as the front end, making HttpClient calls to Node.js that in turn talks to MongoDB. Or, just to round out the hat trick, AngularJS to Node.js to SQL Server.

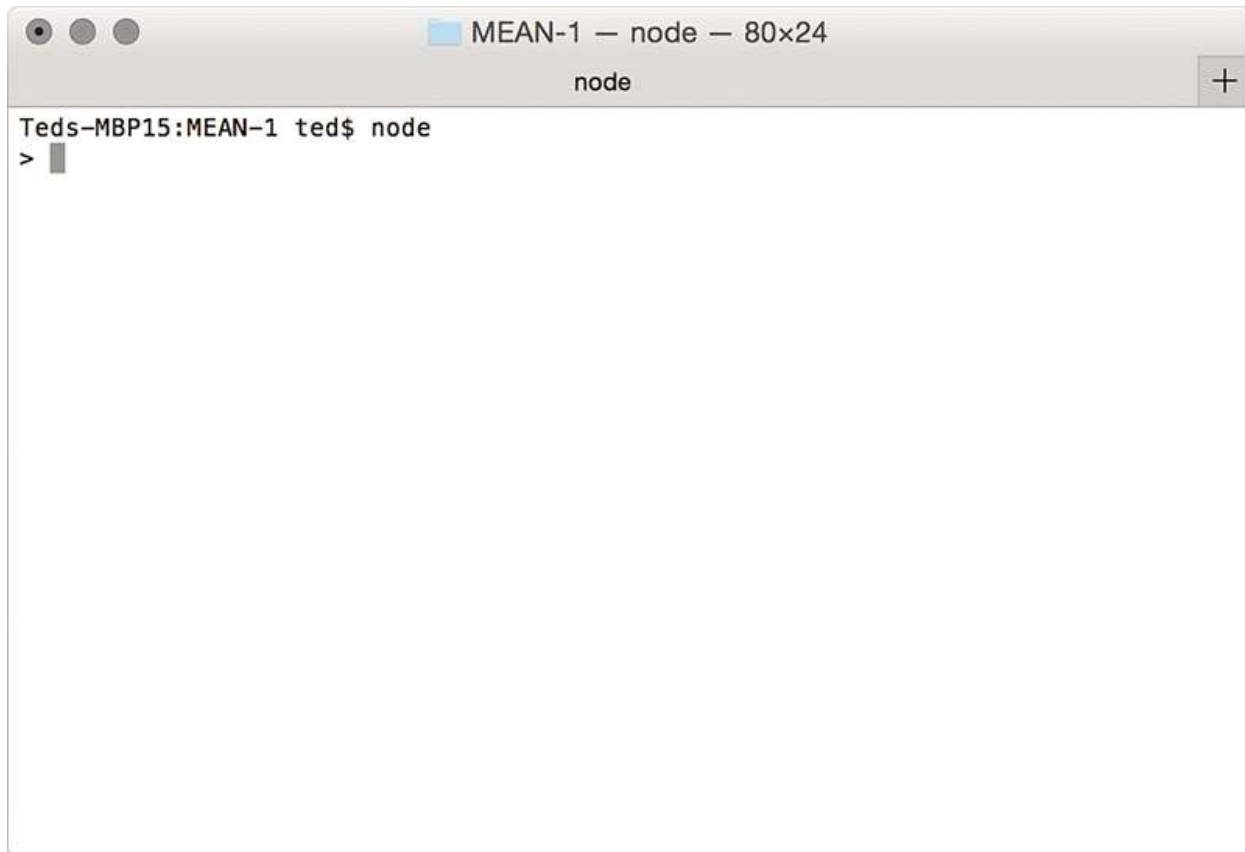
Any of these three components is easily “swappable,” so long as the front end uses HTTP (or something similarly platform-neutral) to talk to the back end. The back end also needs a driver to communicate to your choice of data store.

Install Node.js

Node.js is really a trivial beast to install, when you get right down to it. Developers who have the Azure SDK already installed have Node.js, and it's probably already on the PATH, to boot. Just type `node` in a command prompt to verify it's installed. Ctrl+C takes you out of the interactive shell that fires up if Node.js is installed, by the way.

You'll have to install Node.js on a fresh-from-the-factory Windows box, or a similarly new Mac OS box for that matter. On Windows, the best way to do this is either get the Azure SDK, or go to the Node.js Web site for an MSI installer for Node.js, which puts it on the PATH by default. There's also an installer for the Mac OS, but the better approach on the Mac is to install another package manager called Homebrew. It's available at brew.sh. Once installed, this becomes your “go-to tool” for installing anything on the Mac, including Node.js.

Homebrew has a simple “`brew install node`” that will pull down all the Node.js bits, install them to the right places (without requiring root access to do it), and put it implicitly on the PATH. Again, “`node`” at the command line will verify that the installation worked. When launched, it provides the lowest-noise response of any utility you'll ever install (as shown in **Figure 1**).

A screenshot of a macOS terminal window. The title bar at the top reads "MEAN-1 — node — 80x24". Below the title bar, the window title "node" is displayed. The terminal content shows the prompt "Teds-MBP15:MEAN-1 ted\$ node" followed by a new line starting with ">".

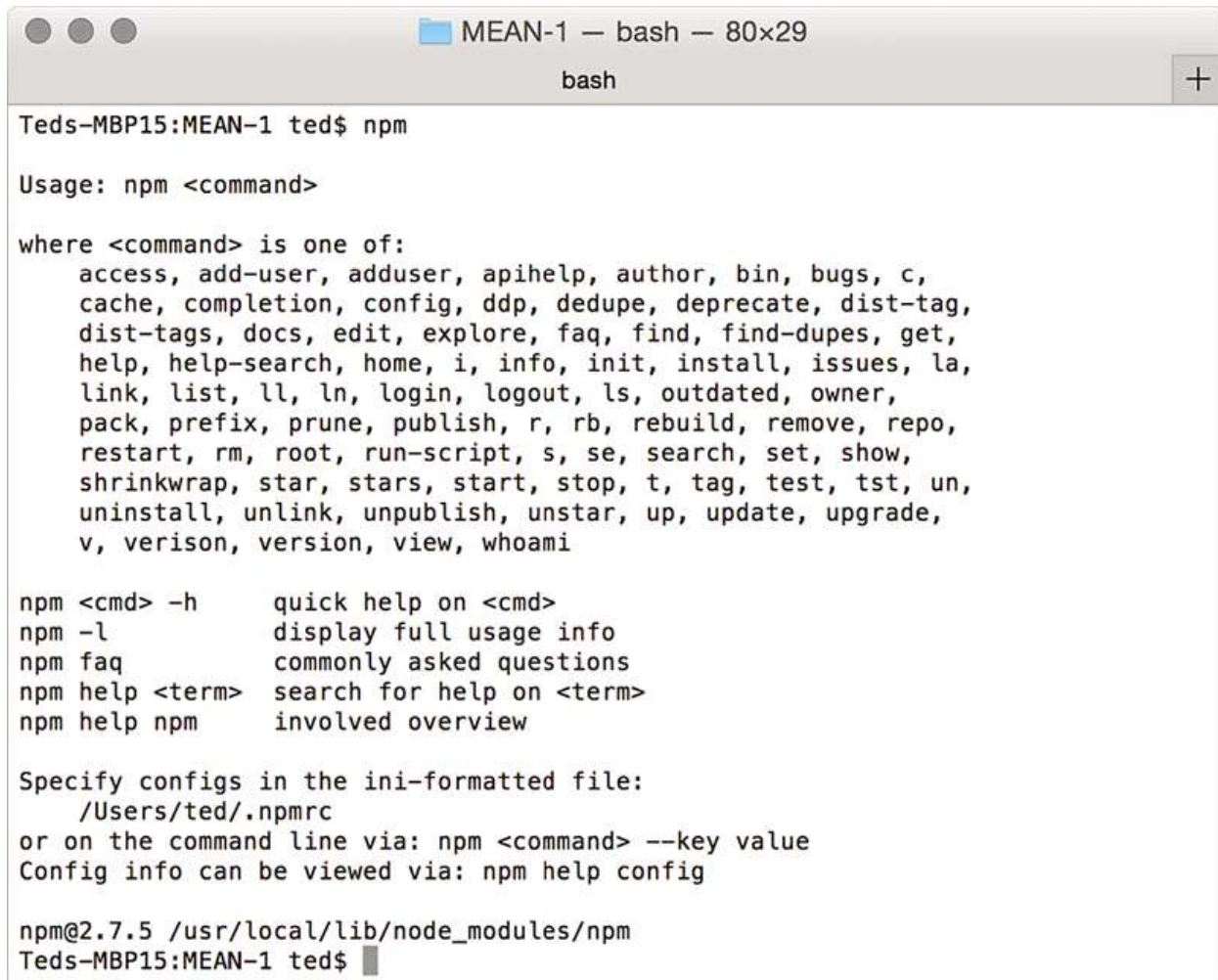
```
Teds-MBP15:MEAN-1 ted$ node
>
```

Figure 1 Node.js Running; No, Seriously, It's Running

Hit Ctrl+C (twice) to get Node.js to quit. You're started, but you're not done.

Install the NPM

The .NET universe has NuGet. The Ruby universe has gems. The Node.js universe has the Node Package Manager. It's called the npm and it installs as part of Node.js itself (npm is actually a small Node.js app that runs from the command line). Without having to do anything else once Node.js is installed, you should be able to fire up npm using just "npm" at the command line, as shown in **Figure 2**.

A screenshot of a macOS terminal window titled "MEAN-1 — bash — 80x29". The terminal shows the output of the command "npm" entered at the prompt "Teds-MBP15:MEAN-1 ted\$". The output displays the usage of npm, a list of available commands, and configuration options. The prompt "Teds-MBP15:MEAN-1 ted\$" is shown again at the bottom of the terminal.

```
MEAN-1 — bash — 80x29
bash
Teds-MBP15:MEAN-1 ted$ npm

Usage: npm <command>

where <command> is one of:
  access, add-user, adduser, apihelp, author, bin, bugs, c,
  cache, completion, config, ddp, dedupe, deprecate, dist-tag,
  dist-tags, docs, edit, explore, faq, find, find-dupes, get,
  help, help-search, home, i, info, init, install, issues, la,
  link, list, ll, ln, login, logout, ls, outdated, owner,
  pack, prefix, prune, publish, r, rb, rebuild, remove, repo,
  restart, rm, root, run-script, s, se, search, set, show,
  shrinkwrap, star, stars, start, stop, t, tag, test, tst, un,
  uninstall, unlink, unpublish, unstar, up, update, upgrade,
  v, verison, version, view, whoami

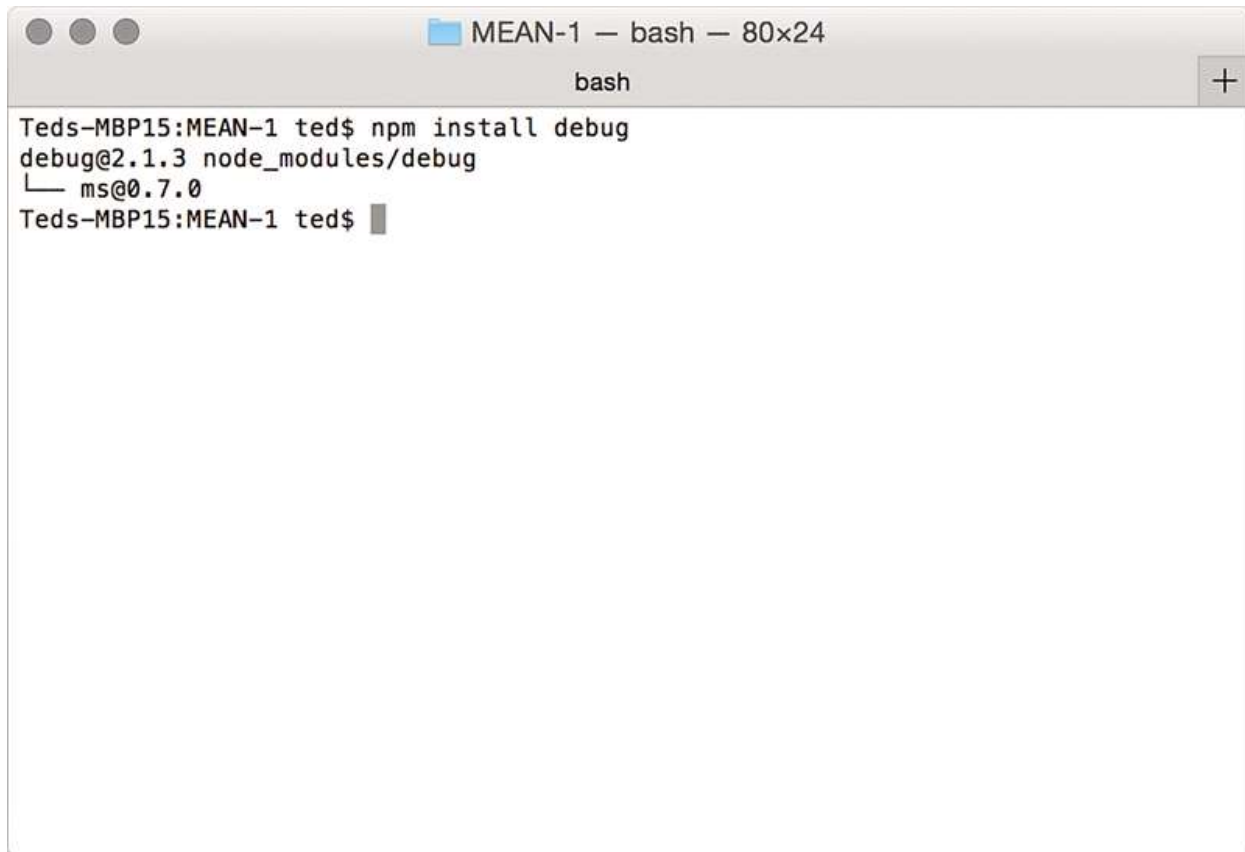
npm <cmd> -h      quick help on <cmd>
npm -l           display full usage info
npm faq          commonly asked questions
npm help <term>  search for help on <term>
npm help npm     involved overview

Specify configs in the ini-formatted file:
  /Users/ted/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

npm@2.7.5 /usr/local/lib/node_modules/npm
Teds-MBP15:MEAN-1 ted$
```

Figure 2 The Node Package Manager

The two commands you'll care most about are `npm install` and `npm update`. Each can take one or more npm package names as a parameter. When you install a package, npm will download the package from the npm Web site, just as NuGet does. It then installs locally on the hard drive under the current directory. So, for example, in an empty directory, tell npm to install the debug package (as shown in **Figure 3**).




```
MEAN-1 — bash — 80x24
bash
Teds-MBP15:MEAN-1 ted$ npm install debug
debug@2.1.3 node_modules/debug
└─ ms@0.7.0
Teds-MBP15:MEAN-1 ted$
```

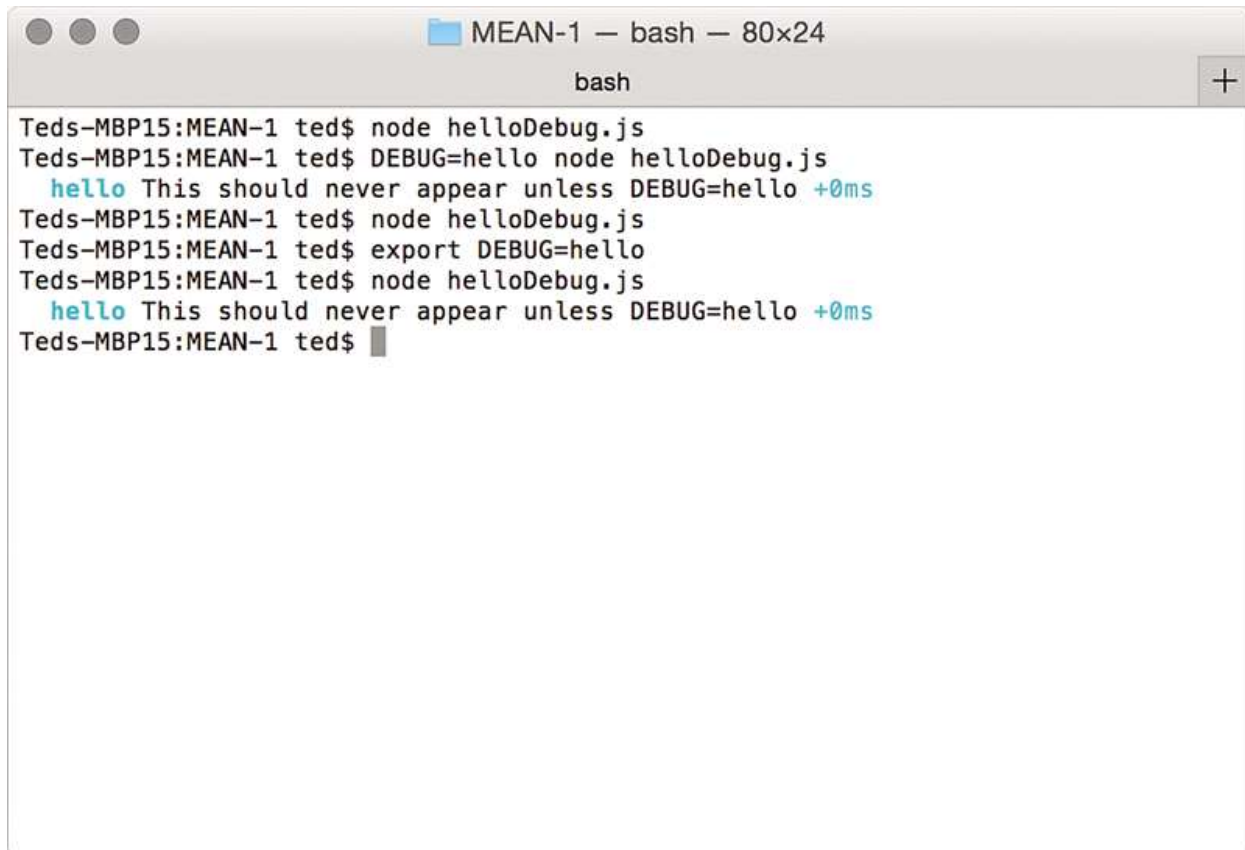
Figure 3 Install an npm Package

The npm tool responds by telling you it has downloaded version 2.1.3 of the debug package, which in turn depends on version 0.7.0 of the ms package. More important, both have been installed into a local directory called node_modules. This is the “local repository” of all Node.js packages you’ll use for this application. If for some reason you want an npm package installed globally (to a node_modules directory somewhere in a shared location), you’ll need to use `npm install -g debug`.

Once those packages are installed in the current directory, reference them using the “require” call in Node.js. It takes a string describing the package name, which Node expects to find inside the node_modules directory directly under the current directory. So the following code, in a file called helloDebug.js, loads the debug package, assigns it to a variable of the same name (the Node.js convention), and uses it to obtain a debug stream for emitting debug information (similar to `System.Diagnostics.Trace`):

JavaScript	 Copy
<pre>var debug = require('debug')('hello'); debug("This should never appear unless DEBUG=hello");</pre>	

Now when you run that code (node helloDebug.js), if there’s an environment variable named DEBUG set to “hello,” debug calls will print to the console. If not, nothing will appear. On a Mac or Unix system, you can temporarily set an environment variable for one run of Node.js by prefixing the assignment right in front of the node command (as shown in **Figure 4**).

A terminal window titled "MEAN-1 — bash — 80x24" with a "bash" tab. The terminal shows a sequence of commands and their outputs. The first command is `node helloDebug.js`, which outputs `hello This should never appear unless DEBUG=hello +0ms`. The second command is `DEBUG=hello node helloDebug.js`, which also outputs the same message. The third command is `export DEBUG=hello`. The fourth command is `node helloDebug.js`, which again outputs the same message. The prompt is `Teds-MBP15:MEAN-1 ted$`.

```
Teds-MBP15:MEAN-1 ted$ node helloDebug.js
Teds-MBP15:MEAN-1 ted$ DEBUG=hello node helloDebug.js
  hello This should never appear unless DEBUG=hello +0ms
Teds-MBP15:MEAN-1 ted$ node helloDebug.js
Teds-MBP15:MEAN-1 ted$ export DEBUG=hello
Teds-MBP15:MEAN-1 ted$ node helloDebug.js
  hello This should never appear unless DEBUG=hello +0ms
Teds-MBP15:MEAN-1 ted$
```

Figure 4 Hello, Debug World

It's not a lot, but it starts to give you a feel for how Node.js development works. Most important, you should realize a `require` call is looking to load a package out of the local `node_modules` directory. So if a `require` fails, it means the package was either corrupted locally or was never installed. In the next column, I'll talk about how to keep track of which npm packages are installed so you don't have to remember.

Did I forget to mention all of the Node.js code is JavaScript? If you're uncomfortable with JavaScript, now's a good time to brush up. Douglas Crockford's "JavaScript: The Good Parts" (O'Reilly Media, 2008) is a great place to start.

Install MongoDB

Getting MongoDB onto a local development system is indeed trivial. Download the .zip file appropriate for your system from the MongoDB Web site ([mongodb.org](https://www.mongodb.org)), unzip it and put the binaries on your PATH. MongoDB is also available in several "as-a-Service" flavors, such as MongoLab ([mongolab.com](https://www.mongolab.com)), which offers a forever-free tier for data loads less than half a gig (which is plenty for most introductory purposes). Either install MongoDB locally or create a MongoLab account.

The MongoDB download also has the "mongo" command-line client (similar in style and scope to the SQL Server command-line client). This is useful for accessing a MongoDB database from shell scripts and the like. If you're more GUI-centric, there are

a few free MongoDB GUI tools in the world. My favorite for the Mac is RoboMongo, and for MongoVue for Windows.

By default, assume Mongo is running locally (meaning the server is "localhost" and the default port is 27017). If you're unfamiliar with Mongo, you can either check out my earlier MongoDB column at msdn.microsoft.com/magazine/ee310029 , or spend a few minutes brushing up on any of the tens of thousands of articles on MongoDB online. Bing is your friend here.

Running MongoDB on a local machine is also trivial. Assume the MongoDB bin directory is on the PATH and just fire up "mongod." It will assume it can write to the "/var" directory to store data. That's usually not what you want, so pass a "--dbpath" argument (note the double hyphen) to specify the directory of choice in which to store data.

The MongoDB Web site also has instructions on how to run MongoDB as a Windows service or as a daemon on *nix platforms. However, launching it "by hand" in the foreground has the added advantage of being able to see the MongoDB logs being written to screen as they happen.

Wrapping Up

That's all the space for this month, but you have the core bits you need to get started. Next time, I'll start spinning up some HTTP endpoints on the server, use Express, talk a little bit more about how a Node.js application is structured, and how to build those endpoints to run in Azure. I'll also start looking at some of the tools Node.js applications use as part of their development. But for now ...

Happy coding!

Ted Neward is the CTO at iTrellis, a consulting services company. He has written more than 100 articles and authored and co-authored a dozen books, including "Professional F# 2.0" (Wrox, 2010). He's an F# MVP and speaks at conferences around the world. He consults and mentors regularly—reach him at ted@tedneward.com or ted@itrellis.com if you're interested.