

10/01/2015 • 8 minutes to read

In this article

[Get on the Express](#)

[Hello, Express](#)

[Debug, Express and You](#)

[Wrapping Up](#)

October 2015

Volume 30 Number 10

The Working Programmer - How To Be MEAN: Express Install

By [Ted Neward](#) | October 2015

The MongoDB, Express, AngularJS, Node.js (MEAN) stack is an alternative “Web stack” (some say complementary, others say supplementary) to the ASP.NET stack to which .NET developers are accustomed. This installment marks the third in the series in which I’ll cover the Express library that handles HTTP processing on the server.

In the last installment (msdn.microsoft.com/magazine/mt422588), I showed how to install a Node.js application into Microsoft Azure. Because doing this is essentially just committing to a Git repository (and then pushing those commits to the Azure remote repository), I’ll leave those out in this and future columns, at least until I start talking to other services (such as MongoDB) on the Azure platform. Alternatively, you can run all the examples on local machines in order to follow along. Azure certainly isn’t required (at least, not until I start talking about pushing to production).

Get on the Express

All train puns aside, Express is a fairly straightforward library. It’s easy to work with, once you embrace “the Node.js way.” Just to keep the cognitive load light, I’ll start from scratch (relatively speaking). I’ll assume you have a brand-new Azure site (the results of a successful “azure site create—git” command) and a simple Express application up and running.

In the first installment (msdn.microsoft.com/magazine/mt185576), I mentioned npm, the Node Package Manager. This is the library and dependency manager for all Node

applications. It's similar to the role NuGet plays in the Microsoft .NET Framework world. In fact, both were inspired by Ruby gems, so they share a number of the same characteristics.

As it turns out, for all of its information, npm depends on a JSON file that contains all of the dependencies—production and development-only packages—on a single file called `package.json`. So a logical first step is to create the file with Express as a dependency, as shown in **Figure 1**.

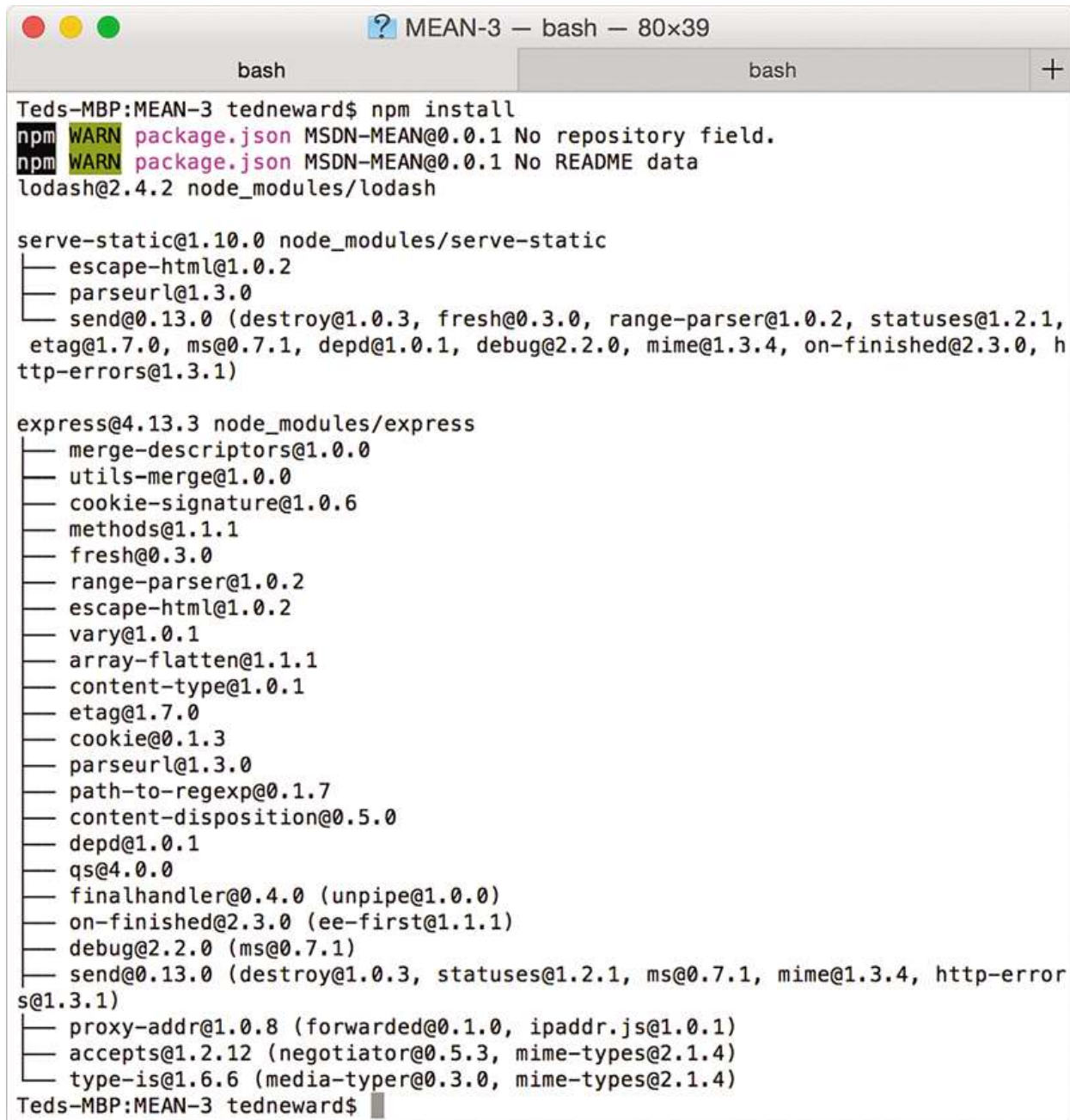
Figure 1 Create a File with Express as a Dependency

JavaScript Copy

```
{
  "name": "MSDN-MEAN",
  "version": "0.0.1",
  "description": "Testing out building a MEAN app from scratch",
  "main": "app.js",
  "scripts": {},
  "author": "Ted Neward",
  "license": "ISC",
  "dependencies": {
    "express": "^4.9.8",
    "lodash": "^2.4.1",
    "serve-static": "^1.7.0"
  },
  "devDependencies": {
  }
}
```

The contents of this file are mostly self-explanatory, but there are a few things worth pointing out. First, you'd often maintain this file by hand (although there are some tools that will manage it for you). That's in keeping with the "text editor and command line" mentality of the Node.js world. As a result, would-be Node developers should be comfortable editing by hand, regardless of the availability of any tools. Second, there are two sets of dependencies here. The dependencies are the packages the npm should install in a production-type environment. The devDependencies are packages used solely by developers.

For now, the easiest way to think about this is dependencies will be installed when the app is pushed (through Git) to Azure. The devDependencies are installed (along with the contents of dependencies) when installing packages on a developer laptop, as shown in **Figure 2**



```

Teds-MBP:MEAN-3 tedneward$ npm install
npm WARN package.json MSDN-MEAN@0.0.1 No repository field.
npm WARN package.json MSDN-MEAN@0.0.1 No README data
lodash@2.4.2 node_modules/lodash

serve-static@1.10.0 node_modules/serve-static
├── escape-html@1.0.2
├── parseurl@1.3.0
├── send@0.13.0 (destroy@1.0.3, fresh@0.3.0, range-parser@1.0.2, statuses@1.2.1,
  etag@1.7.0, ms@0.7.1, depd@1.0.1, debug@2.2.0, mime@1.3.4, on-finished@2.3.0, h
  ttp-errors@1.3.1)

express@4.13.3 node_modules/express
├── merge-descriptors@1.0.0
├── utils-merge@1.0.0
├── cookie-signature@1.0.6
├── methods@1.1.1
├── fresh@0.3.0
├── range-parser@1.0.2
├── escape-html@1.0.2
├── vary@1.0.1
├── array-flatten@1.1.1
├── content-type@1.0.1
├── etag@1.7.0
├── cookie@0.1.3
├── parseurl@1.3.0
├── path-to-regexp@0.1.7
├── content-disposition@0.5.0
├── depd@1.0.1
├── qs@4.0.0
├── finalhandler@0.4.0 (unpipe@1.0.0)
├── on-finished@2.3.0 (ee-first@1.1.1)
├── debug@2.2.0 (ms@0.7.1)
├── send@0.13.0 (destroy@1.0.3, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-error
s@1.3.1)
├── proxy-addr@1.0.8 (forwarded@0.1.0, ipaddr.js@1.0.1)
├── accepts@1.2.12 (negotiator@0.5.3, mime-types@2.1.4)
├── type-is@1.6.6 (media-typer@0.3.0, mime-types@2.1.4)
Teds-MBP:MEAN-3 tedneward$


```

Figure 2 Install Your Dependencies

Speaking of which, you should do that now. With just the package.json file in the current directory, enter “npm install” at the command line and watch npm pull down express, lodash (a handy library of functions and methods, including some functional tools like map and filter, as well as extensions to arrays and objects), and serve-static (which I’ll discuss later) onto your local machine.

Like NuGet, npm installs not just the package, but all its dependencies, as well. This is what causes the tree display in the terminal. For large umbrella packages that incorporate a large number of popular packages, this display can get quite long. The npm tool also notes that I’ve left out two entries in my package.json: the repository field where this package comes from and the README data. These are used mostly for npm packages installed into the central npm library. I don’t find them particularly appropriate for an application, but there’s certainly no harm in having them if you prefer warning-less npm activity.

As this series continues, I will add additional packages to package.json, so you should periodically run “npm install” to get the new packages. As a matter of fact, notice in the install output tree for express, the debug package (which I used in the last installment) is a dependency, but I didn’t put it into the package.json file directly. It’s been my habit that my package.json file should explicitly list all dependencies I use directly (as opposed to those used by the libraries, but never called directly), so go ahead and add it to package.json and do another “npm install”:

JavaScript	 Copy
<pre>"dependencies": { "debug": "^2.2.0", "express": "^4.9.8", "lodash": "^2.4.1", "serve-static": "^1.7.0" },</pre>	


This way I know which version of debug I’m actually using (as opposed to what version Express is using), in case that difference becomes important. By the way, the traditional way to install a library (now that you know what the file format looks like) is to use “npm install <package> --save.” Using the “--save” argument causes npm to modify the package.json file (as opposed to just installing it without record).

In fact, if you do an “npm install express --save,” npm will update the express entry to the latest version (which, as of this writing, is 4.13.3), even though it’s already there. There’s yet a third way, using another package called yeoman to scaffold out an Express app, but part of the goal here is to understand all the moving parts in MEAN, so I’ll leave that for later.

Hello, Express

Once the Express bits are installed, it’s a simple matter to write the ubiquitous “hello world” in Express. In a file called app.js (because that’s what package.json has as its entry for “main”), the code in **Figure 3** provides the simple-yet-necessary homage to the Gods of Computer Science.

Figure 3 The Code for the Hello World Express

JavaScript	 Copy
<pre>// Load modules var express = require('express'); var debug = require('debug')('app'); // Create express instance var app = express(); // Set up a simple route</pre>	

```
app.get('/', function (req, res) {
  debug("/ requested");
  res.send('Hello World!');
});
// Start the server
var port = process.env.PORT || 3000;
debug("We picked up",port,"for the port");
var server = app.listen(port, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('Example app listening at http://%s:%s', host, port);
});
```

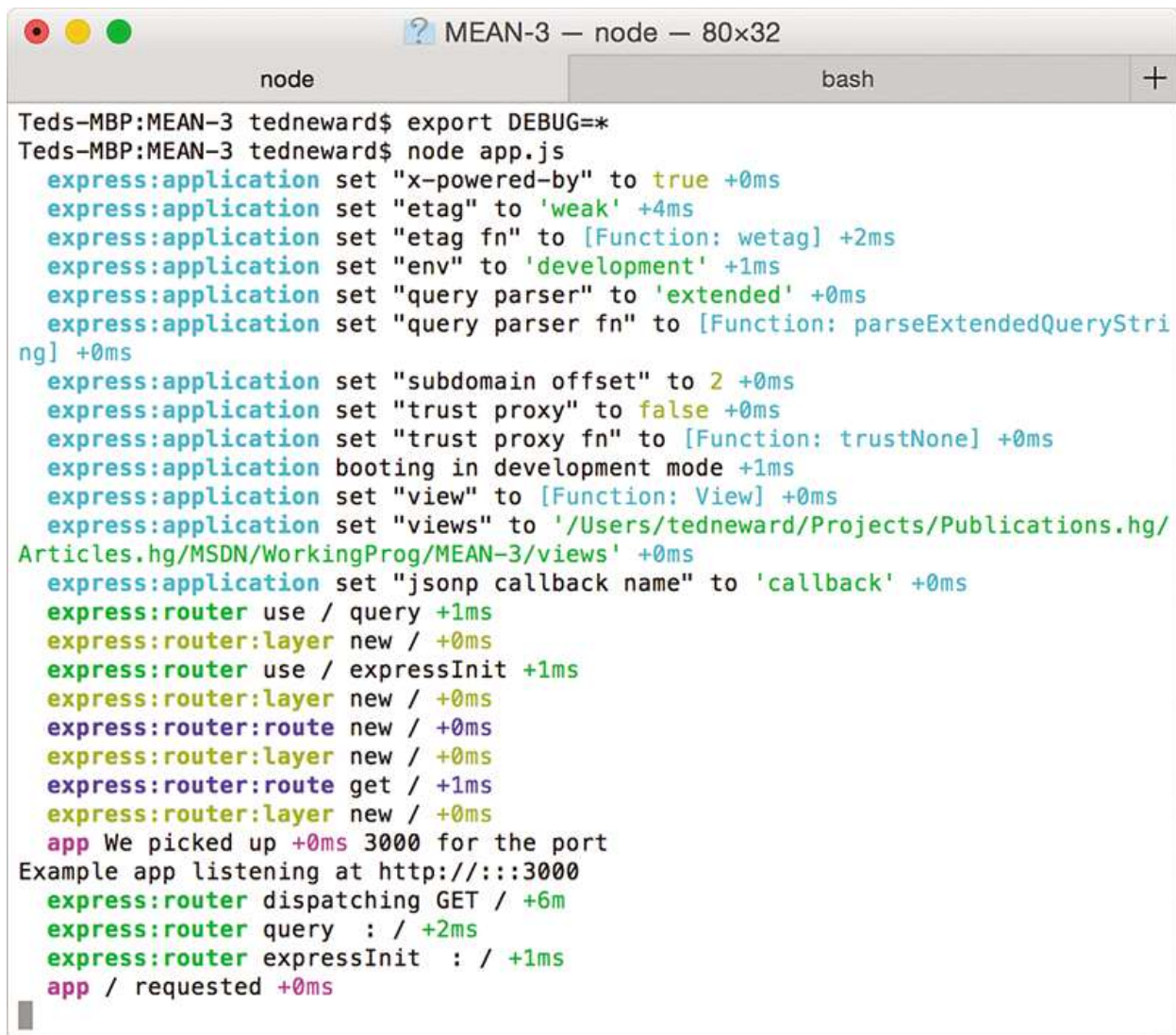
The first two lines are the require calls you saw in the previous installment, which is how Node.js loads modules at the moment. This will change in ECMAScript 6, once that's ratified and shipped. These give you access to the Express and debug packages, respectively. The Express module fetched by require is actually a function that, when called, yields the Express app itself—hence, it goes into app (by convention).

The next line is what Express calls a route. This is similar in many respects to the routing tables more popular in ASP.NET and ASP.NET MVC of late. This maps a request verb (get) with a relative URL path ("/" in this case) to a function. In this case, it's an anonymous function literal that responds with "Hello World." Note the function also uses the debug object to spew out a quick line after receiving a request.

Last, the code checks the current environment to see if there's a PORT environment variable specified (which there will be in Azure). If not, it assigns 3000 to port. Then the app object is told to listen, which is a blocking call. It will execute the anonymous function literal passed in when it begins listening. Then the process will just wait for an incoming request. Assuming the request is for "/", it replies with "Hello World."

Debug, Express and You

As mentioned in the previous installment, the debug output doesn't show up unless the DEBUG environment variable is set to the same string used in the require step. In this case, that's app. Express also uses debug, and if DEBUG is set to "*", then all of the Express diagnostic output will also display. It's probably a bit too voluminous to use for debugging on a regular basis, but it can be helpful to see it scroll by in the early days of working with Express. That will help give you a sense of the various parts and what's being invoked when. There's an example of its output (with a few requests) shown in Figure 4.




```

MEAN-3 — node — 80x32
node bash +
Teds-MBP:MEAN-3 tedneward$ export DEBUG=*
Teds-MBP:MEAN-3 tedneward$ node app.js
express:application set "x-powered-by" to true +0ms
express:application set "etag" to 'weak' +4ms
express:application set "etag fn" to [Function: wetag] +2ms
express:application set "env" to 'development' +1ms
express:application set "query parser" to 'extended' +0ms
express:application set "query parser fn" to [Function: parseExtendedQueryStri
ng] +0ms
express:application set "subdomain offset" to 2 +0ms
express:application set "trust proxy" to false +0ms
express:application set "trust proxy fn" to [Function: trustNone] +0ms
express:application booting in development mode +1ms
express:application set "view" to [Function: View] +0ms
express:application set "views" to '/Users/tedneward/Projects/Publications.hg/
Articles.hg/MSDN/WorkingProg/MEAN-3/views' +0ms
express:application set "jsonp callback name" to 'callback' +0ms
express:router use / query +1ms
express:router:layer new / +0ms
express:router use / expressInit +1ms
express:router:layer new / +0ms
express:router:route new / +0ms
express:router:layer new / +0ms
express:router:route get / +1ms
express:router:layer new / +0ms
app We picked up +0ms 3000 for the port
Example app listening at http://:::3000
express:router dispatching GET / +6m
express:router query : / +2ms
express:router expressInit : / +1ms
app / requested +0ms

```

Figure 4 Express with Debug Logging

To turn debugging off, simply set `DEBUG` to nothing. To view more than one debug stream, but not all of them, just comma-separate the names:

JavaScript	 Copy
DEBUG=app,express:router,express:router:layer	

That command will only show output from those three debug streams.

Wrapping Up

In many ways, Express is a lot like ASP.NET. That's true not only in how it handles HTTP traffic, but how it serves as a sort of "central hub" around which dozens (if not hundreds) of other packages depend and extend. The "serve-static" package, for example, gives an Express app a pre-built way to serve up a directory for static (that is, "not executing on the server side") assets, like images and fonts. In the next installment, I'll start looking at how to use the Express request and response objects. (If you can't

wait, check out the Express documentation at expressjs.com .) In the meantime ... happy coding!

Ted Neward *is the CTO at iTrellis, a consulting services company. He has written more than 100 articles and authored or co-authored a dozen books, including "Professional F# 2.0" (Wrox, 2010). He is 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.*

Thanks to the following technical expert for reviewing this article: Shawn Wildermuth
Shawn Wildermuth is a thirteen-time Microsoft MVP (ASP.NET/IIS), the author of eight books and dozens of articles on software development, and a Pluralsight author with over fifteen courses to his name. He is one of the Wilder Minds (<http://wilder minds.com>) and can be reached at his blog at <http://wildermuth.com> .