Mastering MEAN: Introducing the MEAN stack

Develop modern, full-stack, twenty-first-century web projects from end-to-end

Scott Davis
Founder
ThirstyHead.com

09 September 2014

Build a modern web application with MongoDB, Express, AngularJS, and Node.js in this six-part series by web development expert Scott Davis. This first installment includes a demo, sample code, and full instructions for creating a basic MEAN application. You'll also learn about Yeoman generators that you can use to bootstrap a new MEAN application quickly and easily.

View more content in this series

In his 2002 book, David Weinberger described the burgeoning web's content as a collection of *Small Pieces Loosely Joined*. That metaphor stuck with me, because it's easy to get tricked into thinking of the web as a monolithic technology stack. Actually, every website you visit is the product of a unique mixture of libraries, languages, and web frameworks.

One of the earliest collections of open source web technologies to gain prominence is the LAMP stack: Linux® for the operating system, Apache for the web server, MySQL for the database, and Perl (or Python, or PHP) for the programming language used to generate the HTML-based web pages. These technologies weren't written to work together. They are discrete projects that one ambitious software engineer — and then another, and then another — cobbled together. Since then, we've witnessed a Cambrian explosion of web stacks. Every modern programming language seems to have a corresponding web framework (or two) that preassembles a motley crew of technologies to make it quick and easy to bootstrap a new website.

An emerging stack that's gaining much attention and excitement in the web community is the MEAN stack: MongoDB, Express, AngularJS, Node.js. The MEAN stack represents a thoroughly modern approach to web development: one in which a single language (JavaScript) runs on every tier of your application, from client to server to persistence. This series demonstrates what a MEAN web development project looks like end-to-end, going beyond simple syntax. This initial installment gets you started with an in-depth, hands-on introduction to the stack's component technologies, including installation and setup. See Download to get the sample code.

About this series

The MEAN (MongoDB, Express, AngularJS, Node.js) stack is a modern challenger to the long-popular LAMP stack for building professional websites with open source software. MEAN represents a major shift in architecture and mental models — from relational databases to NoSQL and from server-side Model-View-Controller to client-side, single-page applications. In this series, learn how the MEAN stack's technologies complement one another and how to use the stack to create modern, twenty-first century, full-stack JavaScript web applications.

“ Every website you visit is the product of a unique mixture of libraries, languages, and web frameworks. ”

From LAMP to MEAN

MEAN is more than a simple rearrangement of acronym letters and technology upgrades. Switching the base platform from an OS (Linux) to a JavaScript runtime (Node.js) brings OS independence: Node.js runs as well on Windows® and OS X as it does on Linux.

Node.js also replaces Apache in the LAMP stack. But Node.js is far more than a simple web server. In fact, you don't deploy your finished application to a stand-alone web server; instead, the web server is included in your application and installed automagically in the MEAN stack. The deployment process is dramatically simpler as a result, because the required version of the web server is explicitly defined along with the rest of your runtime dependencies.

More than MEAN

Although this series focuses on the four main planets in the MEAN solar system, you'll also fly by several smaller (yet no less important) satellite technologies in the MEAN stack:

- **Yeoman**: A three-headed combination of command-line development tools for scaffolding (Yo), build scripts (Grunt), and client-side dependency management (Bower).
- **Bootstrap**: A CSS library that gives you a mobile-ready responsive web design out of the box.
- **Testing libraries**: In addition to Mocha, Jasmine, and Karma, a whole galaxy of testing libraries for mocking out Ajax calls (Chai), showing test coverage (Istanbul), and automating your functional tests to run in actual browsers (Protractor).

The move from a traditional database such as MySQL to a NoSQL, schemaless, document-oriented persistence store such as MongoDB represents a fundamental shift in persistence strategy. You'll spend less time writing SQL and more time writing map/reduce functions in JavaScript. You'll also cut out huge swaths of transformation logic, because MongoDB emits JavaScript Object Notation (JSON) natively. Consequently, writing RESTful web services is easier than ever.

But the biggest shift from LAMP to MEAN is the move from traditional server-side page generation to a client-side single-page application (SPA) orientation. With Express, you can still handle server-side routing and page generation, but the emphasis is now on client-side views, courtesy of AngularJS. This change involves more than simply shifting your Model-View-Controller (MVC) artifacts from the server to the client. You'll also be taking the leap from a synchronous mentality to

one that is fundamentally event-driven and asynchronous in nature. And perhaps most important, you'll move from a page-centric view of your application to one that is component-oriented.

The MEAN stack isn't mobile-centric — AngularJS runs equally well on desktops and laptops, smartphones and tablets, and even smart TVs — but it doesn't treat mobile devices as second-class citizens. And testing is no longer an afterthought: With world-class testing frameworks such as MochaJS, JasmineJS, and KarmaJS, you can write thorough, comprehensive test suites for your MEAN app.

Ready to get MEAN?

Installing Node.js

You need a working installation of Node.js to work on the sample application in this series, so now is the time to install Node if you haven't already.

If you are on a UNIX®-like OS (Linux, Mac OS X, and so on), I recommend that you use Node Version Manager (NVM). (Otherwise, click **Install** on the Node.js home page to download the installer for your OS, and accept the defaults.) With NVM, you can easily download Node.js and switch among various versions from the command line. This helps me seamlessly move from one version of Node.js to the next as I move from one client project to the next.

After NVM is installed, type `nvm ls-remote` to see which versions of Node.js are available for installation, as shown in Listing 1.

Listing 1. Using NVM to list available versions of Node.js

```
$ nvm ls-remote

v0.10.20

v0.10.21
v0.10.22
v0.10.23
v0.10.24
v0.10.25
v0.10.26
v0.10.27
v0.10.28
```

Typing `nvm ls` shows which versions of Node.js you already have installed locally, and which version is currently in use.

At the time of writing, the Node website suggests that v0.10.28 is the most recent stable version. Type `nvm install v0.10.28` to install it locally.

After you install Node.js (either via NVM or the platform-specific installer), type `node --version` to verify that you are using the current version:

```
$ node --version

v0.10.28
```

What is Node.js?

Node.js is a headless JavaScript runtime. It is literally the same JavaScript engine (named V8) that runs inside of Google Chrome, except that with Node.js, you can run JavaScript from the command line instead of in your browser.

Accessing your browser's developer tools

Get comfortable with the developer tools in the browser of your choice. I'll use Google Chrome throughout this series, but feel free to use Firefox, Safari, or even Internet Explorer.

- In Google Chrome, click **Tools > JavaScript Console**.
- In Firefox, click **Tools > Web Developer > Browser Console**.
- In Safari, click **Develop > Show Error Console**. (If you don't have a Develop menu, click **Show Develop menu in menu bar** on the Advanced preferences page.)
- In Internet Explorer, click **Developer Tools > Script > Console**.

I've had students scoff at the idea of running JavaScript from the command line: "If there isn't HTML to manipulate, what is JavaScript good for?" JavaScript was introduced to the world in a browser (Netscape Navigator 2.0), so those naysayers can be forgiven for their shortsightedness and naiveté.

In fact, JavaScript the programming language has no native capabilities for **Document Object Model (DOM)** manipulation or for making Ajax requests. Browsers provide DOM APIs so that you can do that sort of thing with JavaScript, but outside of the browser JavaScript loses those capabilities.

Here's an example. Open a JavaScript console in your browser (see Accessing your browser's developer tools). Type `navigator.appName`. After you get a response, type `navigator.appVersion`. Your results are similar to those in Figure 1.

Figure 1. Using the JavaScript `navigator` object in a web browser

In Figure 1, the response to `navigator.appName` is `Netscape`, and the response to `navigator.appVersion` is the cryptic user agent string that seasoned web developers have come to know and either love or loathe. In Figure 1 (from Chrome on OS X), the string is `5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/35.0.1916.153 Safari/537.36`.

Now, create a file named test.js. Type the same commands in the file, wrapping each in a `console.log()` call:

```
console.log(navigator.appName);
console.log(navigator.appVersion);
```

Save the file and type `node test.js` to run it, as shown in Listing 2.

Listing 2. Viewing the `navigator is not defined` error in Node.js

```
$ node test.js

/test.js:1
ion (exports, require, module, __filename, __dirname) { console.log(navigator.
                                                                    ^
ReferenceError: navigator is not defined
    at Object.<anonymous> (/test.js:1:75)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:902:3
```

As you can see, `navigator` is available to you in the browser but not from Node.js. (Sorry for making your first Node.js script fail on you spectacularly, but I want to make sure that you are convinced that running JavaScript in the browser is different from running it in Node.js.)

According to the stack trace, the proper `Module` isn't loaded. (Modules are another major difference between running JavaScript in the browser and running it in Node.js. More on modules in a moment.) To get similar information from Node.js, change the contents of test.js to:

```
console.log(process.versions)
console.log(process.arch)
console.log(process.platform)
```

Type `node test.js` again, and you'll see output similar that in Listing 3.

Listing 3. Using the process module in Node.js

```
$ node test.js

{ http_parser: '1.0',
  node: '0.10.28',
  v8: '3.14.5.9',
  ares: '1.9.0-DEV',
  uv: '0.10.27',
  zlib: '1.2.3',

  modules: '11',
  openssl: '1.0.1g' }
x64
darwin
```

Now that you've successfully run your first script in Node.js, it's time to tackle the next major concept: modules.

What are modules?

You can create single-purpose functions in JavaScript, but — unlike in Java, Ruby, or Perl — you have no way to bundle multiple functions into a cohesive module or "package" that can be imported and exported. Of course, you can include any JavaScript source file by using the `<script>` element, but that time-honored method falls short of a proper module declaration in two key ways.

First, any JavaScript you include by using the `<script>` element is loaded into the global namespace. With modules, you can sandbox the imported functionality in a locally namespaced variable. Second, and more crucially, you can explicitly declare your dependencies with modules, whereas with the `<script>` element, you can't. As a result, importing Module A transitively imports dependent Modules B and C also. As your application gains complexity, transitive dependency management quickly becomes a vital requirement.

CommonJS

The CommonJS project, as the name implies, defines a common format for modules (among other outside-of-the-browser JavaScript specifications). Node.js is one of many CommonJS implementations out in the wild. RingoJS (an app server similar to Node.js that runs on the JDK's Rhino/Nashorn JavaScript runtime) is CommonJS-based, as are popular NoSQL persistence stores CouchDB and MongoDB.

Modules are a hotly anticipated feature of the next major version of JavaScript (ECMAScript 6), but until that version is widely adopted, Node.js is using its own version of modules based on the CommonJS specification.

You include a CommonJS module in your script by using the `require` keyword. For example, Listing 4 is a slightly modified version of the Hello World script you can find on the Node.js homepage. Create a file named example.js and copy the code from Listing 4 into it.

Listing 4. Hello World in Node.js

```
var http = require('http');
var port = 9090;
http.createServer(responseHandler).listen(port);
console.log('Server running at http://127.0.0.1:' + port + '/');

function responseHandler(req, res){
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('<html><body><h1>Hello World</h1></body></html>');
}
```

Type `node example.js` to launch your new web server, and visit http://127.0.0.1:9090 in your web browser.

Look at the first two lines in Listing 4. You've most likely written simple statements like `var port = 9090;` hundreds (or thousands) of times. This statement defines a variable named `port` and assigns the number `9090` to it. Importing a CommonJS module as you do in the first line (`var http = require('http');`) is really no different. It brings in the `http` module and assigns it to a local variable. All of the corresponding modules that `http` relies on are `required` in also.

The subsequent lines of example.js:

1. Create a new HTTP server.
2. Assign a function to handle the responses.
3. Start listening on the specified port for incoming HTTP requests.

So, in a handful of lines of JavaScript, you created a simple web server in Node.js. As you'll learn in future tutorials in this series, Express expands on this simple example to handle more-complex routes and serve up both static and dynamically generated resources.

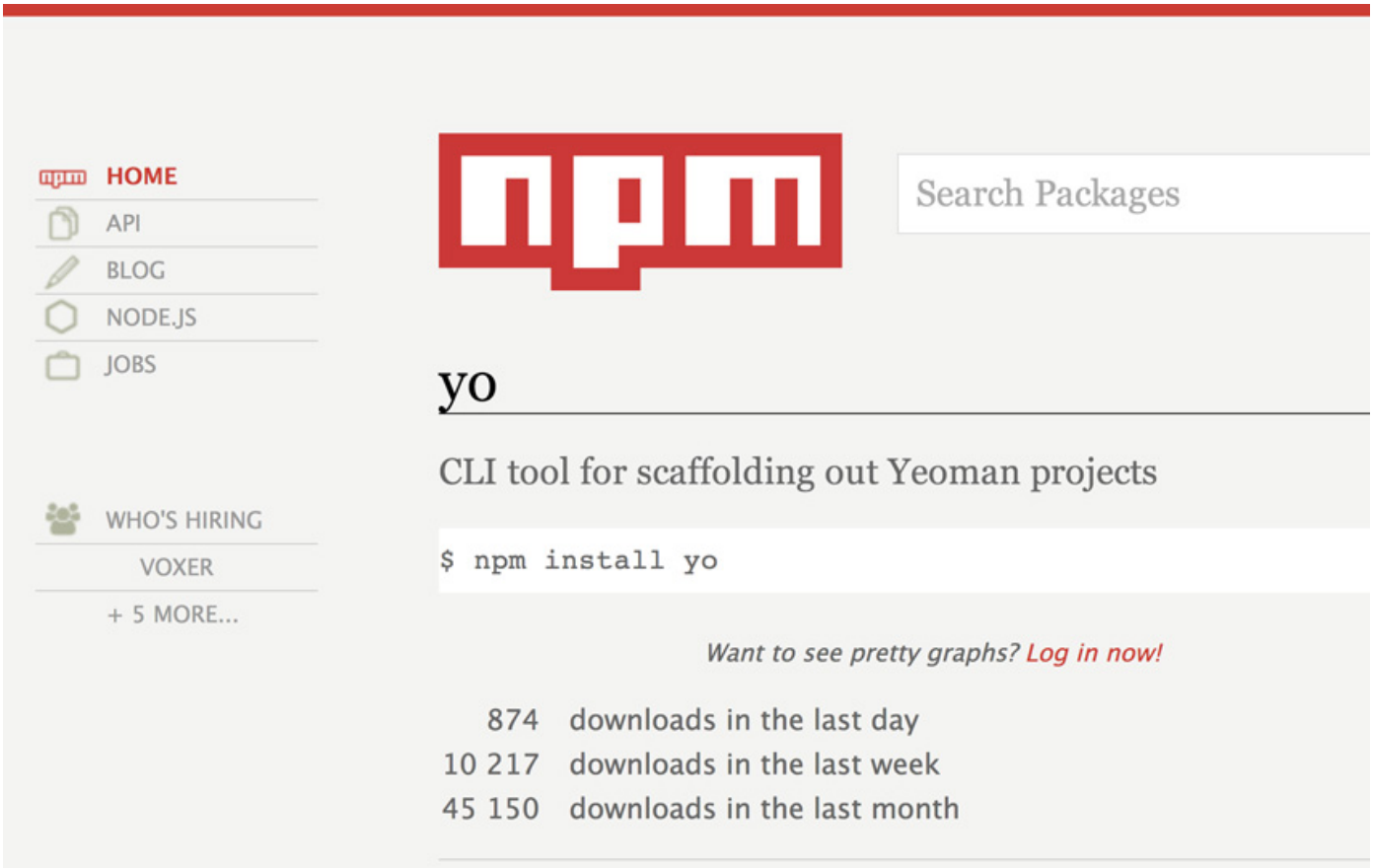
The `http` module is a standard part of any Node.js installation. Other standard Node.js modules enable file I/O, read command-line input from the user, handle lower-level TCP and UDP requests, and much more. Visit the Modules section of the Node.js documentation to see the full list of standard modules and read about their capabilities.

Although the list of included modules is impressive, it pales in comparison to the list of available third-party modules. To gain access to them, you need to familiarize yourself with another command-line utility: NPM.

What is NPM?

NPM is short for Node Packaged Modules. To see a list of more than 75,000 publicly available third-party Node modules, visit the `NPM` website. On the site, search for the `yo` module. Figure 2 shows the results.

Figure 2. Details of the yo module



The result page gives you a brief description of the module ("CLI tool for scaffolding out Yeoman projects"); how many times it was downloaded in the past day, week, and month; who wrote it; which other modules (if any) that it depends on; and much more. Most important, the result page gives you the command-line syntax for installing the module.

To get similar information about the `yo` module from the command line, type `npm info yo`. (If you didn't already know the official name of the module, you could type `npm search yo` to search for any module whose name contains the string `yo`.) The `npm info` command displays the contents of the module's package.json file.

Understanding package.json

Every Node.js module must have a well-formed package.json file associated with it, so it's worth getting familiar with the contents of this file. Listings 5, 6, and 7 show the contents of package.json for `yo`, divided into three parts.

The first elements, shown in Listing 5, are typically `name`, `description`, and a JSON array of available `versions`.

Listing 5. package.json, Part 1

```
$ npm info yo

{ name: 'yo',
  description: 'CLI tool for scaffolding out Yeoman projects',
  'dist-tags': { latest: '1.1.2' },
  versions:
   [
     '1.0.0',
     '1.1.0',
     '1.1.1',
     '1.1.2' ],
```

To install the latest version of a module, you type `npm install package` . Typing `npm install package@version` installs a specific version.

Next, as shown in Listing 6, are authors, maintainers, and the GitHub repository where you can find the source directly.

Listing 6. package.json, Part 2

```
author: 'Chrome Developer Relations',
repository:
 { type: 'git',
   url: 'git://github.com/yeoman/yo' },
homepage: 'http://yeoman.io',
keywords:
 [ 'front-end',
   'development',
   'dev',
   'build',
   'web',
   'tool',
   'cli',
   'scaffold',
   'stack' ],
```

In this case, you also find a link to the project's homepage and a JSON array of associated keywords. Not all of these fields are present in every package.json file, but users rarely complain about having too much metadata associated with a project.

Finally, you see a list of dependencies with explicit version numbers, as in Listing 7. These version numbers follow a common pattern of *major version*.*minor version*.*patch version* called SemVer (Semantic Versioning).

Listing 7. package.json, Part 3

```
engines: { node: '>=0.8.0', npm: '>=1.2.10' },
dependencies:
 { 'yeoman-generator': '~0.16.0',
   nopt: '~2.1.1',
   lodash: '~2.4.1',
   'update-notifier': '~0.1.3',
   insight: '~0.3.0',
   'sudo-block': '~0.3.0',
   async: '~0.2.9',
   open: '0.0.4',
   chalk: '~0.4.0',
```

```
   findup: '~0.1.3',
   shelljs: '~0.2.6' },
peerDependencies:
 { 'grunt-cli': '~0.1.7',
   bower: '>=0.9.0' },
devDependencies:
 { grunt: '~0.4.2',
   mockery: '~1.4.0',
   'grunt-contrib-jshint': '~0.8.0',
   'grunt-contrib-watch': '~0.5.3',
   'grunt-mocha-test': '~0.8.1' },
```

This package.json file indicates that it must be installed on a Node.js instance of version 0.8.0 or higher. If you try to `npm install` it on an unsupported version, the installation fails.

Shortcut syntax for SemVer

In Listing 7, notice the tilde (~) in many of the dependency versions. This is the equivalent of 1.0.x (also valid syntax), which means "the major version must be 1, the minor version must be 0, but you can install the latest patch version you find." Implicit in SemVer is the idea that patch versions *never* make breaking changes to an API (typically they are bug fixes to existing functions), and minor versions bring additional functionality (such as new function calls) without breaking existing functionality.

Beyond the platform requirement, this package.json file also provides several lists of dependencies:

- The `dependencies` block lists runtime dependencies.
- The `devDependencies` block lists modules that are required during the development process.
- The `peerDependencies` block enables the author to define a "peer" relationship between projects. This capability is often used to specify the relationship between a base project and its plugins, but in this case, it identifies the other two projects (Grunt and Bower) that comprise the Yeoman project along with Yo.

If you type `npm install` without a specific module name, `npm` looks in the current directory for a package.json file and installs all dependencies listed in the three blocks I just discussed.

The next step toward getting a working MEAN stack installed is to install Yeoman and the corresponding Yeoman-MEAN generator.

Installing Yeoman

As a Java developer, I couldn't imagine starting a new project without a build system such as Ant or Maven. Similarly, Groovy and Grails developers rely on Gant (a Groovy implementation of Ant) or Gradle. These tools can scaffold out a new directory structure, download dependencies on the fly, and prepare your project for distribution.

In the pure web development world, Yeoman fills this need. Yeoman is a collection of three Node.js and pure JavaScript tools for scaffolding (Yo), managing client-side dependencies (Bower), and preparing your project for distribution (Grunt). As you know from examining Listing 7, installing Yo brings its peers Grunt and Bower along for the ride, thanks to the `peerDependencies` block in package.json.

Normally, you'd type `npm install yo --save` to install the `yo` module and update the `dependencies` block in your package.json. (`npm install yo --save-dev` updates the `devDependencies` block.) But the three peer modules of Yeoman aren't really project-specific — they are command-line utilities, not runtime dependencies. To install a NPM package globally, you add the `-g` flag to the `install` command.

Install Yeoman on your system:

```
npm install -g yo
```

After the package is installed, type `yo --version` to verify that it's up and running.

With Yeoman and all of the rest of the infrastructure in place, you're ready to install the MEAN stack.

Installing MeanJS

You could meticulously install each part of the MEAN stack by hand. Thankfully, Yeoman offers an easier path via its *generators*.

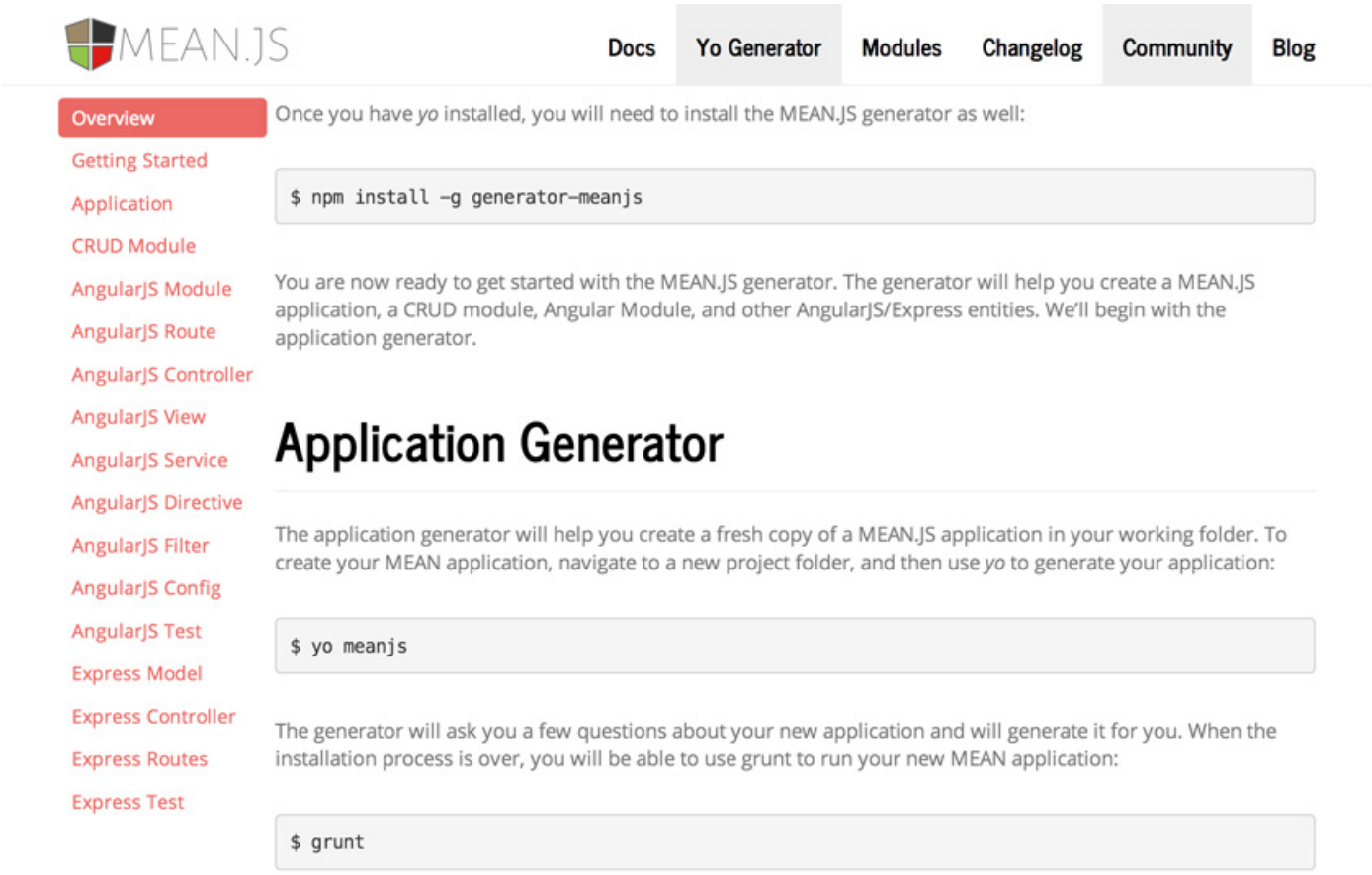
A Yeoman generator is the easiest way to bootstrap a new web project. The generator pulls in the base packages and all of their dependencies. And it typically includes a working build script with all of the associated plugins. Many times, the generator also includes a sample application, complete with tests.

There are over 1,000 Yeoman generators available. Several of them are written and maintained by the Yeoman core developers — look for The Yeoman Team in the Author column. But the majority of the generators are written by the community.

The community generator you'll use to bootstrap your first MEAN app is named, not surprisingly, MEAN.JS.

On the MEAN.JS home page, click the **Yo Generator** menu option or go directly to the Generator page, part of which is shown in Figure 3.

Figure 3. The MEAN.JS Yeoman generator



The page's instructions tell you to install Yeoman first, which you've already done. The next step is to install the MEAN.JS generator globally:

```
npm install -g generator-meanjs
```

When the generator is in place, you're ready to create your first MEAN app. Create a directory named test, `cd` into it, and type `yo meanjs` to generate the application. Answer the last two questions as shown in Listing 8. (You can provide your own answers for the first four.)

Listing 8. Using the MEAN.JS Yeoman generator

```
$ mkdir test
$ cd test
$ yo meanjs


     _-----_
    |       |
    |--(o)--|   .--------------------------.
   `---------#  |    Welcome to Yeoman,    |
    ( _#U`_ )   |   ladies and gentlemen!  |
    /___A___\   '__________________________'
     |  ~  |
   __'.___.'__
 #    `   |# # Y `
```

```
You're using the official MEAN.JS generator.
[?] What would you like to call your application?
Test
[?] How would you describe your application?
Full-Stack JavaScript with MongoDB, Express, AngularJS, and Node.js
[?] How would you describe your application in comma separated key words?
MongoDB, Express, AngularJS, Node.js
[?] What is your company/author name?
Scott Davis
[?] Would you like to generate the article example CRUD module?
Yes
[?] Which AngularJS modules would you like to include?
ngCookies, ngAnimate, ngTouch, ngSanitize
```

After you answer the final question, you see a flurry of activity as NPM downloads all of your server-side dependencies (including Express). After NPM is done, you see Bower download all of your client-side dependencies (including AngularJS, Bootstrap, and jQuery).

At this point, you have the EAN stack (Express, AngularJS, and Node.js) installed — all you are missing is the M (MongoDB). If you typed `grunt` right now to start up your application without MongoDB installed, you'd see an error message similar to the one in Listing 9.

Listing 9. Trying to start MeanJS without MongoDB

```
events.js:72
        throw er; // Unhandled 'error' event
              ^
Error: failed to connect to [localhost:27017]
    at null.<anonymous>
(/test/node_modules/mongoose/node_modules/mongodb/lib/mongodb/connection/server.js:546:74)

[nodemon] app crashed - waiting for file changes before starting...
```

If you started the app and see this error message, press Ctrl+C to stop the app.

Now you'll get MongoDB installed so that you can take your new MEAN app out for a ride.

Installing MongoDB

MongoDB is a NoSQL persistence store. It's not written in JavaScript, nor is it an NPM package. You must install it separately for your MEAN stack to be complete and functional.

Visit the MongoDB home page, download the platform-specific installer, and accept all of the defaults as you install MongoDB.

When the installation is complete, type `mongod` to start the MongoDB daemon.

The MeanJS Yeoman generator already installed a MongoDB client module called Mongoose; you can check your package.json file to verify this. I'll cover MongoDB and Mongoose in detail in a later installment.

With MongoDB installed and running, you are finally ready to launch your MEAN app and look around.

Launching the MEAN application

To start the newly installed MEAN application, make sure that you are in the test directory you created before running the MeanJS Yeoman generator. When you type `grunt`, you should see output similar to that shown in Listing 10.

Listing 10. Starting the MEAN.JS app

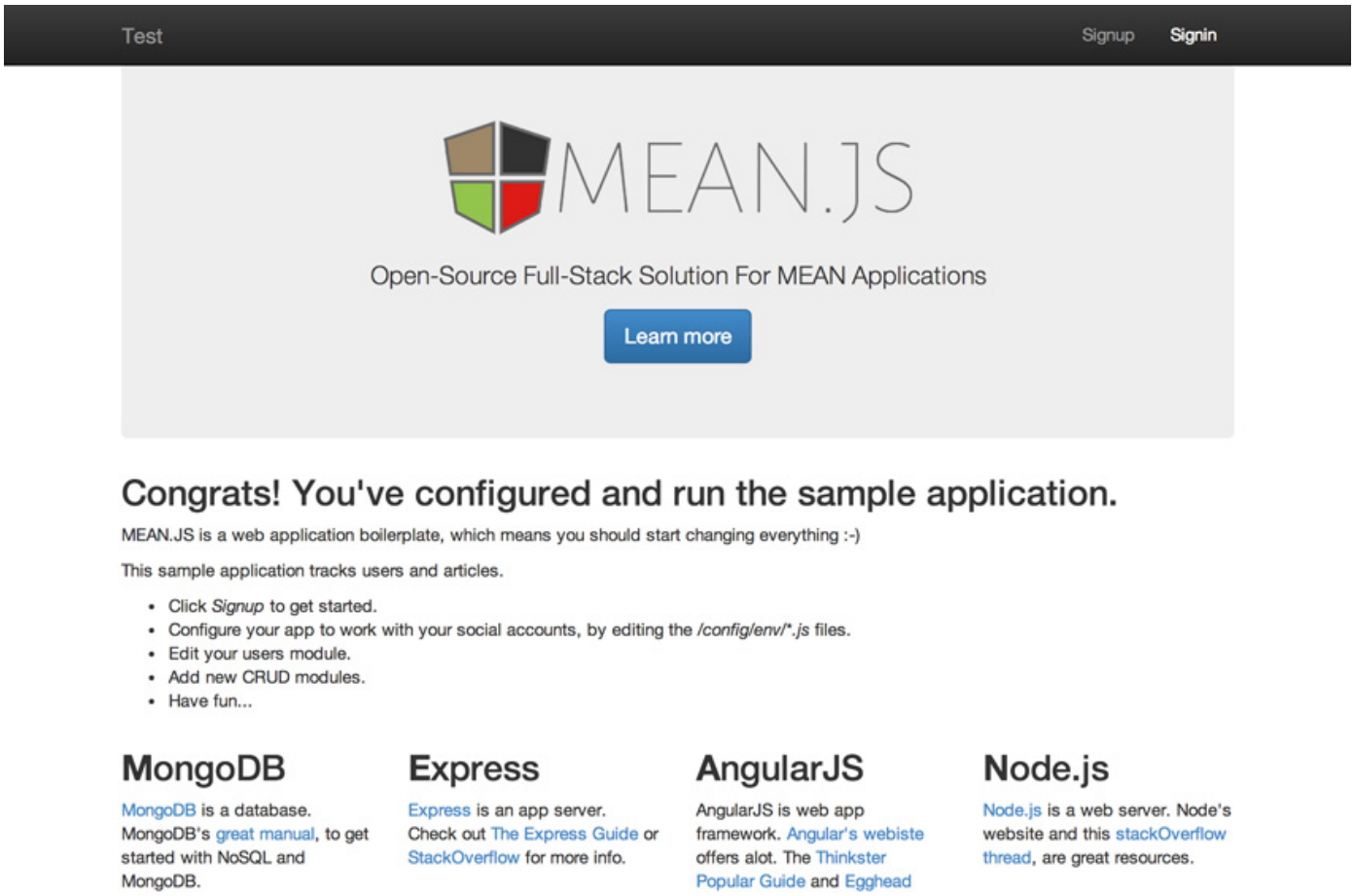
```
$ grunt

Running "jshint:all" (jshint) task
>> 46 files lint free.

Running "csslint:all" (csslint) task
>> 2 files lint free.

Running "concurrent:default" (concurrent) task
Running "watch" task
Waiting...
Running "nodemon:dev" (nodemon) task
[nodemon] v1.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: app/views/**/*.* gruntfile.js server.js config/**/*.js app/**/*.js
[nodemon] starting `node --debug server.js`
debugger listening on port 5858

 NODE_ENV is not defined! Using default development environment

MEAN.JS application started on port 3000
```

The `jshint` and `csslint` modules (both installed by the generator) ensure that the source code is syntactically and stylistically correct. The `nodemon` package watches the file system for changes to source code and automatically restarts the server when it detects any — a huge boon for developer velocity as you make rapid, frequent changes to your codebase. (The `nodemon` package only runs during the development phase; for production changes, you must redeploy your application and restart Node.js.)

As the console output suggests, visit http://localhost:3000 and take your new MEAN app out for a spin.

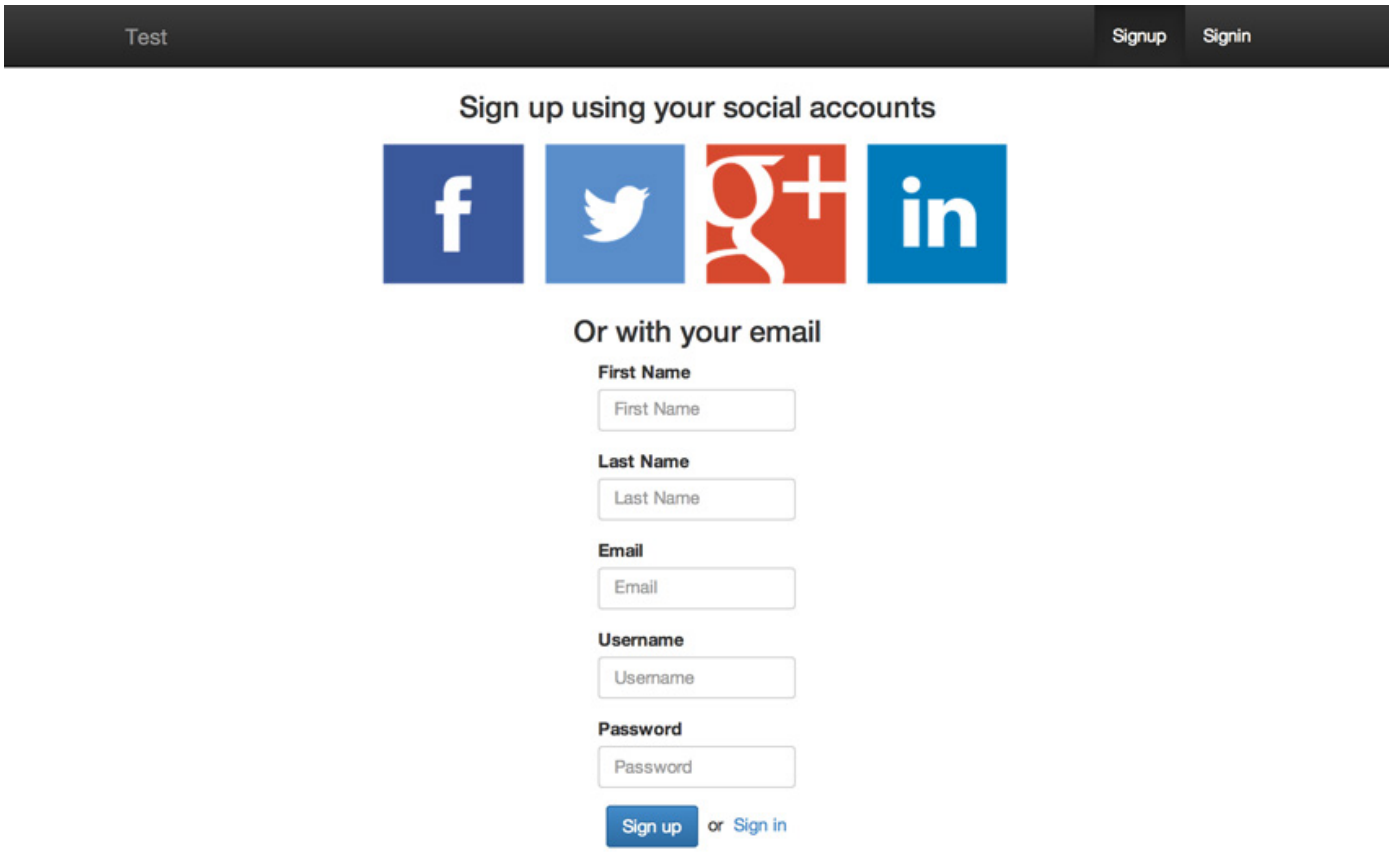
Figure 4 shows the MEAN.JS sample application home page.

Figure 4. The MEAN.JS sample application home page



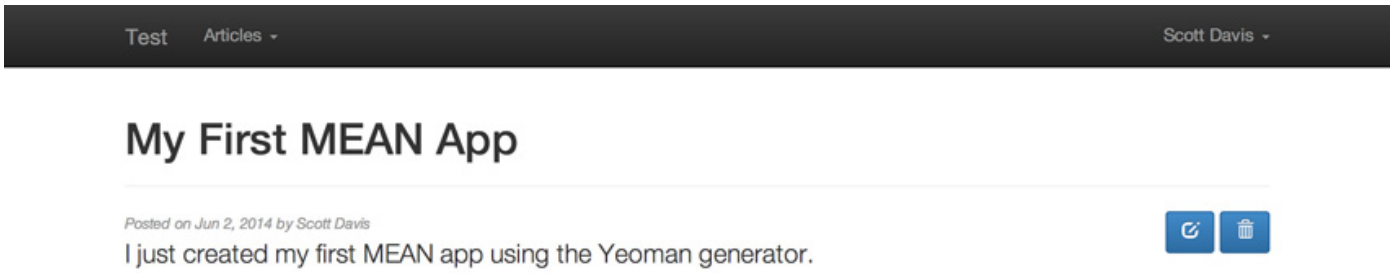
Click **Signup** in the menu bar to create a new user account. For now, fill in all of the fields on the Sign-up page (shown in Figure 5) and click **Sign up**. You'll enable OAuth logins via Facebook, Twitter, and so on in a later tutorial.

Figure 5. The MEAN.JS sample application Sign-up page



Now you have a set of user credentials stored in your local MongoDB instance and can begin creating new articles. Click the **Articles** menu option (which doesn't appear until you've logged in), and create a few sample articles. Figure 6 shows the Articles page.

Figure 6. The MeanJS article page



You have christened your first MEAN application. Welcome to the party!

Conclusion

You accomplished quite a bit in this tutorial. You installed Node.js and wrote your first Node.js script. You learned about modules and used NPM to install several third-party modules. You

installed Yeoman to give yourself a solid web development platform that includes a scaffolding utility (Yo), a build script (Grunt), and a utility to manage client-side dependencies (Bower). You installed the MeanJS Yeoman generator and used it to create your first MEAN application. You installed MongoDB and the Node.js client library Mongoose. And finally, you ran your first MEAN application.

Next time, you'll take a detailed walk through the source code of the sample application so that you can see how all four planets in the MEAN solar system — MongoDB, Express, AngularJS, and Node.js — interact with one another.

Downloads

Description	Name	Size
Sample code	wa-mean1src.zip	1.38MB

Resources

- "Build a real-time polls application with Node.js, Express, AngularJS, and MongoDB" (developerWorks, June 2014): Check out a MEAN development project that's deployed on IBM Bluemix™.
- "Node.js for Java developers" (developerWorks, November 2011): Get an introduction to Node.js and find out why its event-driven concurrency has sparked interest, even among die-hard Java developers.
- Getting started with Node.js (developerWorks, January 2014): View this 9-minute demo to get a quick introduction to Node.js and Express.
- "MongoDB: A NoSQL datastore with (all the right) RDBMS moves" (developerWorks, September 2010): Learn about MongoDB's custom API, interactive shell, and support for both RDBMS-style dynamic queries and quick, easy MapReduce calculations.
- "Get started with the JavaScript language" (developerWorks, April and August 2011): Learn JavaScript fundamentals in this two-part article.
- "JavaScript for Java developers" (developerWorks, April 2011): Find out why JavaScript is an important tool for the modern Java developer and get started with learning JavaScript variables, types, functions, and classes.
- "Introduction to LAMP technology" (developerWorks, May 2005): Compare MEAN to its predecessor stack.
- *Mastering Grails* (developerWorks, 2008-2009): Check out this series by Scott Davis on Grails, the Groovy-based web development framework.
- Get involved in the developerWorks Community. Connect with other developerWorks users while you explore developer-driven blogs, forums, groups, and wikis.

About the author

Scott Davis

Scott Davis is the founder of ThirstyHead.com, a training and consulting company that specializes in leading-edge technology solutions such as HTML5, mobile development, Node.js, smart TV development, web mapping, NoSQL, Groovy, and Grails. Scott cofounded the HTML5 Denver User Group in 2011. Scott has been writing about web development for more than 10 years. His books include *Getting Started with Grails*, *Groovy Recipes*, *GIS for Web Developers*, *The Google Maps API: Adding Where to Your Web Applications*, and *JBoss at Work*. Scott is also the author of two popular developerWorks article series: *Mastering Grails* and *Practically Groovy*.