IBM

developerWorks®

Mastering MEAN: Tour a MEAN application

Follow an HTTP request from server side to client side

Scott Davis
Founder
ThirstyHead.com

11 September 2014

In the first *Mastering MEAN* installment, you used a Yeoman generator to bootstrap a MEAN application. Now take a walking tour of the application, tracing the first incoming HTTP request from the server side (Node.js and Express) through to the client side (AngularJS).

View more content in this series

In "*Mastering MEAN*: Introducing the MEAN stack," you installed and configured a MEAN development environment. In this article, I'll familiarize you further with the four key pieces of the MEAN stack — MongoDB, Express, AngularJS, and Node.js — by walking you through the sample MEAN.JS application that you created. As you tour the application, you'll follow an incoming HTTP request from the server side through to the client side.

Start your local MongoDB instance by typing `mongod`. (Or, on UNIX®-like operating systems, you can type `mongod &` to start the process in the background.) Next, `cd` to the test directory that you created in the previous article and type `grunt` to start the application that you created with the Yeoman generator. You'll see output similar to that shown in Listing 1.

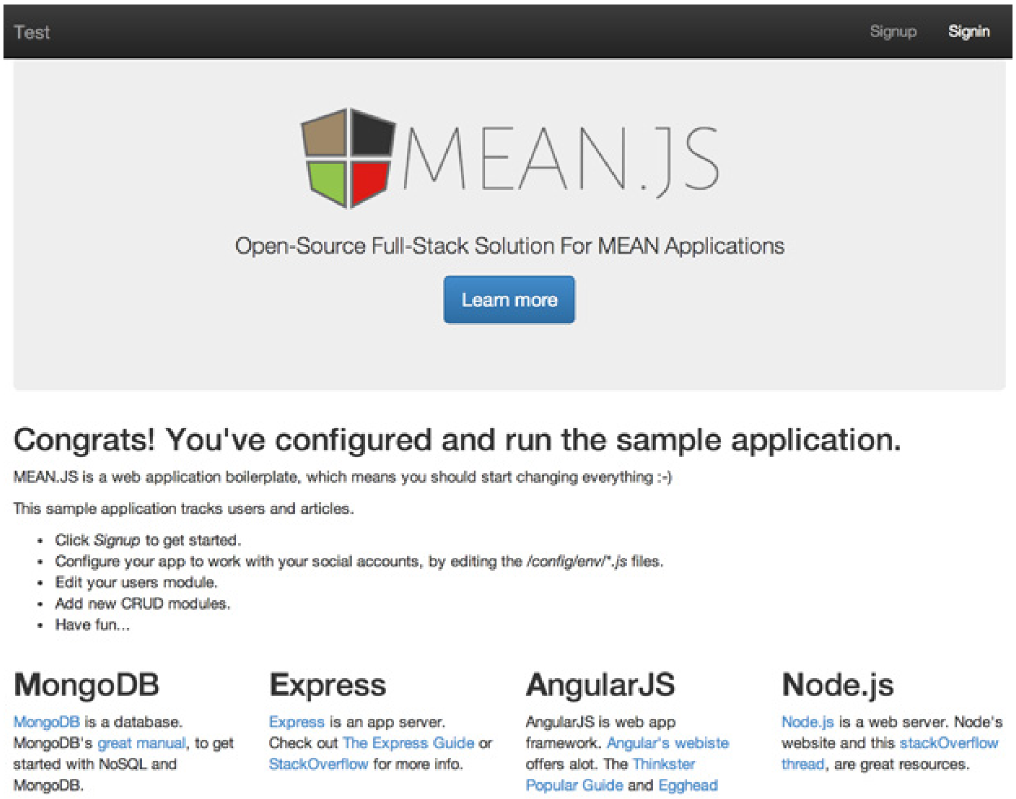
Listing 1. Starting your local MEAN.JS application

```
$ 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 "nodemon:dev" (nodemon) task
Running "watch" task
Waiting...
[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
```

Open http://localhost:3000 in a browser to see the application's home page, shown in Figure 1.

Figure 1. Your local MEAN.JS home page



Congrats! You've configured and run the sample application.

MEAN.JS is a web application boilerplate, which means you should start changing everything :-)

This sample application tracks users and articles.

- Click *Signup* to get started.
- Configure your app to work with your social accounts, by editing the */config/env/*.js* files.
- Edit your users module.
- Add new CRUD modules.
- Have fun...

MongoDB

MongoDB is a database. MongoDB's great manual, to get started with NoSQL and MongoDB.

Express

Express is an app server. Check out The Express Guide or StackOverflow for more info.

AngularJS

AngularJS is web app framework. Angular's webiste offers alot. The Thinkster Popular Guide and Egghead

Node.js

Node.js is a web server. Node's website and this stackOverflow thread, are great resources.

Now you'll poke around the directory structure to see what makes this application tick. (See the Folder Structure page in the MEAN.JS documentation for more information.)

“ Now you'll poke around the directory structure to see what makes this application tick. ”

Understanding the Node.js and Bower configuration files

I'll get to the source code in a moment. First, I'll quickly revisit package.json, the Node.js configuration file that you learned about in the previous article. And I'll add a word about its client-side counterpart, bower.json. Both of these files are in the root directory of the project.

package.json

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.

Arguably, the most important configuration file in any Node.js application is package.json. In this file, you'll find the metadata about your application that you provided to the Yeoman generator — such as name, description, and author — as in the portion of package.json shown in Listing 2.

Listing 2. package.json, Part 1

```
{
  "name": "test",
  "description": "Full-Stack JavaScript with MongoDB, Express, AngularJS, and Node.js",
  "version": "0.0.1",
  "author": "Scott Davis",
  "engines": {
    "node": "0.10.x",
    "npm": "1.4.x"
  },
```

Next, you see a series of commands that you can type at the command prompt, as shown in Listing 3.

Listing 3. package.json, Part 2

```
"scripts": {
  "start": "grunt",
  "test": "grunt test",
  "postinstall": "bower install --config.interactive=false"
},
```

You've already typed `grunt` to start the application. Later, you'll type `grunt test` to run the unit tests. The `postinstall` hook is your first hint at the divide between server-side dependencies and client-side ones.

But the most important part of this most important file lists the application's dependencies, as shown in Listing 4. These CommonJS modules all run on the server side of the house.

Listing 4. package.json, Part 3

```
"dependencies": {
  "express": "~4.2.0",
  "mongoose": "~3.8.8"
},
"devDependencies": {
  "grunt-mocha-test": "~0.10.0",
  "grunt-karma": "~0.8.2",
  "karma": "~0.12.0",
  "karma-jasmine": "~0.2.1",
  "karma-coverage": "~0.2.0",
  "karma-chrome-launcher": "~0.1.2",
  "karma-firefox-launcher": "~0.1.3",
  "karma-phantomjs-launcher": "~0.1.2"
}
```

Runtime dependencies (such as Express for routing and Mongoose for connecting to MongoDB) are declared in the `dependencies` block. Developer and build-time dependencies (including testing frameworks such as Mocha, Jasmine, and Karma) are declared in the `devDependencies` block.

bower.json

Now, switch your focus to the client side. The JavaScript libraries that are loaded in the browser are defined in bower.json, shown in Listing 5.

Listing 5. bower.json

```
{
  "name": "test",
  "version": "0.0.1",
  "description": "Full-Stack JavaScript with MongoDB, Express, AngularJS, and Node.js",
  "dependencies": {
    "bootstrap": "~3",
    "angular": "~1.2",
    "angular-resource": "~1.2",
    "angular-mocks": "~1.2",
    "angular-cookies": "~1.2",
    "angular-animate": "~1.2",
    "angular-touch": "~1.2",
    "angular-sanitize": "~1.2",
    "angular-bootstrap": "~0.11.0",
    "angular-ui-utils": "~0.1.1",
    "angular-ui-router": "~0.2.10"
  }
}
```

As you can see, bower.json is similar to package.json. It holds some of the same metadata fields, and it uses a `dependencies` block to define client-side dependencies such as Bootstrap (for look and feel plus responsive web design) and AngularJS (for your client-side single-page application).

The application's source code, likewise, is split between two key directories: one for the server side and one for the client side.

Understanding the directory structure

This MEAN app has four main directories, as shown in Listing 6.

Listing 6. The MEAN directory structure

```
$ ls -ld */
drwxr-xr-x+  7 scott  staff   238 Jun  6 14:06 app/
drwxr-xr-x+  8 scott  staff   272 Jun  6 14:06 config/
drwxr-xr-x+ 49 scott  staff  1666 Jun  6 14:07 node_modules/
drwxr-xr-x+  8 scott  staff   272 Jun  6 14:06 public/
```

app
: Contains server-side source code.

config
: Contains configuration files.

node_modules
: Contains the server-side modules specified in package.json.

public
Contains client-side source code — including the lib directory, which contains the client-side libraries specified in bower.json.

You'll focus your efforts in the app and public directories. The hunt for the elusive source code for the application's home page begins in the app directory.

Exploring the server side of the MEAN stack

Listing 7 shows the app directory structure.

Listing 7. The app (server-side) directory structure

```
$ tree app
app
|--- controllers
|## |--- articles.server.controller.js
|## |--- core.server.controller.js
|## |--- users.server.controller.js
|--- models
|## |--- article.server.model.js
|## |--- user.server.model.js
|--- routes
|## |--- articles.server.routes.js
|## |--- core.server.routes.js
|## |--- users.server.routes.js
|--- tests
|## |--- article.server.model.test.js
|## |--- user.server.model.test.js
|--- views
    |--- 404.server.view.html
    |--- 500.server.view.html

    |--- index.server.view.html
    |--- layout.server.view.html
```

If you've spent any time writing server-side MVC applications, you know the typical workflow:

1. The incoming HTTP request hits a router.
2. The router finds the appropriate controller to hand the request to.
3. The controller builds up a model (or a list of models) from a database and passes it to a view.
4. The view builds up the HTML page by combining the model with a template and then passes the finished output to the waiting HTTP response.

The app/routes/core.server.routes.js file (part of the Express framework), shown in Listing 8, holds the key entry point to the application.

Listing 8. app/routes/core.server.routes.js

```
'use strict';

module.exports = function(app) {
  // Root routing
  var core = require('../../app/controllers/core');
  app.route('/').get(core.index);
};
```

Strictly speaking

Strict mode is a part of the ECMAScript 5 specification, which is the latest mainstream version of JavaScript. (For more information, see the "Strict mode" article on the Mozilla Developer Network.) Strict mode is backward-compatible. Older browsers that don't understand the `'use strict'`statement simply ignore it; all modern browsers treat it with respect. So if your code runs in newer browsers with strict mode enabled, it runs in older browsers also.

This router defines a single route —`/`— that is handled by the `index` function of the core controller. Notice that the core controller is a CommonJS module that is `require`d in.

The `'use strict';` statement at the start of Listing 8 puts your JavaScript runtime into strict mode, which is less forgiving than the syntactic "anything goes" attitude of JavaScript runtimes of the past. In strict mode, the JavaScript runtime treats honest mistakes — such as accidentally making a variable global, or trying to use a variable that wasn't previously defined — as syntax errors. Strict mode, coupled with JSHint, goes a long way toward ensuring that syntax errors are caught in development rather than production. (The final key to a bug-free release is, of course, a healthy amount of code coverage with unit tests.)

Next, look at app/controllers/core.server.controller.js (also a part of the Express framework), shown in Listing 9.

Listing 9. app/controllers/core.server.controller.js

```
'use strict';

/**
 * Module dependencies.
 */
exports.index = function(req, res) {
  res.render('index', {
    user: req.user || null
  });
};
```

The `index` function accepts an incoming HTTP request and an outgoing HTTP response. Because this particular request doesn't need anything from the database, no models are instantiated. The `index` template is rendered to the response, along with a JSON block of variables that will replace identically named placeholders in the template.

Next up is app/views/index.server.view.html, shown in Listing 10.

Listing 10. app/views/index.server.view.html

```
{% extends 'layout.server.view.html' %}

{% block content %}
  <section data-ui-view></section>
{% endblock %}
```

There's not much to see here, although the link to app/views/layout.server.view.html, shown in Listing 11, looks promising.

Listing 11. app/views/layout.server.view.html

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>
  <title>{{title}}</title>

  <!-- General META -->
  <meta charset="utf-8">
  <meta http-equiv="Content-type" content="text/html;charset=UTF-8">

  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <meta name="viewport" content="width=device-width,initial-scale=1">

  <!-- Semantic META -->
  <meta name="keywords" content="{{keywords}}">
  <meta name="description" content="{{description}}">
```

Express template engines

You can use a variety of template engines with Express. A CommonJS module called ConsolidateJS even adapts non-Express template engines and makes them Express-compliant. I'll stick with Swig for this article series; feel free to experiment with other template libraries on your own.

Now you're beginning to see some HTML that looks familiar. The `{{}}` delimiters surrounding `title`, `keywords`, and `description` identify them as Swig placeholders that are meant to be replaced by actual values. Swig is the template engine installed by the MEAN.JS Yeoman generator.

If you look back at the `core` controller in Listing 9, though, you can see that the only value being passed into this template is `user`. If you suspect that the other placeholders are default values defined in a configuration file somewhere, you're on the right track.

Understanding configuration and environments

Take a look in config/env/all.js, shown in Listing 12, for the `title`, `description`, and `keywords` variables. (I did a search across the directory structure to find where these values are defined — a technique you might want to add to your exploratory toolbox as get to know the MEAN stack.)

Listing 12. config/env/all.js

```
'use strict';

module.exports = {
  app: {
    title: 'Test',
    description: 'Full-Stack JavaScript with MongoDB, Express, AngularJS, and Node.js',
    keywords: 'MongoDB, Express, AngularJS, Node.js'
  },
  port: process.env.PORT || 3000,
  templateEngine: 'swig',
```

In addition to the keywords that the template is expecting, some other interesting values, such as `port` and `templateEngine`, are in this file.

Environment variables

`PORT` and `NODE_ENV` are two examples of environment variables you can set outside of your application to alter behavior inside it. For example, notice the `port` setting in config/env/all.js:

```
port: process.env.PORT || 3000,
```

This setting says to your application, "Set the internal `port` variable to either the value of the `PORT` environment variable or the default value `3000` if `PORT` cannot be found."

To test this setting, press Ctrl+C to stop your application. Instead of restarting it with a naked `grunt` command, try `PORT=4000 grunt`. Your application now runs on port 4000.

You can code Node.js apps to behave differently depending on the type of runtime environment (development, production, staging, testing, and so on) that you run them in. As with the `PORT` variable, Express supplies a default value —`development`— for `NODE_ENV` if you don't explicitly set one to specify the runtime environment. This explains the mild complaint that your application issued when you restarted it:

```
NODE_ENV is not defined! Using default development environment
```

Externalizing runtime configuration in environment variables is a clever way to add flexibility. Variables such as `PORT` and `NODE_ENV` can be set in an ephemeral manner on the command line. This capability makes it trivial to flip the variables' values around for development and testing purposes. (Of course, you can give them more longevity by adding them to .bash_profile or, in Windows®, setting them in the Control Panel.)

You might consider using environment variables for security reasons also. Storing user names, passwords, and connection URLs in environment variables keeps them out of configuration files (and, by extension, source control) where they could be compromised. This approach also makes it easy to deploy a common set of configuration files across multiple developer or production machines and allow each machine to inject unique values or credentials via local environment variables.

You aren't limited to the `PORT` and `NODE_ENV` environment variables. Your Platform as a Service (PaaS) provider typically offers several service-specific variables for you to set.

Named environments

Setting individual environment variables is fine, but you might have a collection of related variables that all must change in unison. For example, you want to avoid the easy mistake of changing a username but not the corresponding password. Thankfully, this MEAN application supports the concept of *named environments*. (This idea isn't unique to MEAN applications. Rails, Grails, and many other popular web frameworks offer similar capabilities.)

Look in config/env in the directory tree in Listing 13, and you'll see several named environment files in place.

Listing 13. The config directory structure

```
$ tree config/
config/
|--- config.js
|--- env
|## |--- all.js
|## |--- development.js
|## |--- production.js
|## |--- test.js
|--- express.js
|--- init.js
|--- passport.js
|--- strategies
    |--- facebook.js
    |--- google.js
    |--- linkedin.js
    |--- local.js
    |--- twitter.js
2 directories, 13 files
```

In config/env, development.js, production.js, and test.js all specify named environments. If you guessed that all.js contains values common to all environments, give yourself a pat on the back.

To see where these files are read in and merged, look in config/config.js, shown in Listing 14.

Listing 14. config/config.js

```
/**
 * Module dependencies.
 */
var _ = require('lodash');

/**
 * Load app configurations
 */
module.exports = _.extend(
  require('./env/all'),
  require('./env/' + process.env.NODE_ENV) || {}
);
```

Lo-Dash and Underscore.js

Lo-Dash is similar to the Underscore.js library. If you've done any BackboneJS development, you're intimately familiar with Underscore.js — Backbone's only hard dependency.

Lo-dash is a CommonJS module that offers convenience functions for arrays, objects, and JSON structures. In Listing 14, the developer's intent is to set some base values in all.js and allow them to be overridden by values in development.js (or production.js or test.js).

You saw config/env/all.js in Listing 12. Listing 15 shows config/env/development.js.

Listing 15. config/env/development.js

```
'use strict';

module.exports = {
  db: 'mongodb://localhost/meanjs-dev',
  app: {
    title: 'MeanJS - Development Environment'
  },
```

Ideally, the `lodash.extend` function would merge the two JSON blocks to yield this result:

```
app: {
  title: 'MeanJS - Development Environment',
  description: 'Full-Stack JavaScript with MongoDB, Express, AngularJS, and Node.js',
  keywords: 'MongoDB, Express, AngularJS, Node.js'
}
```

Unfortunately, that's not the output you get. Add a line of code to print out the merged structure to config/config.js, as shown in Listing 16:

Listing 16. Logging the actual merged results to the console

```
/**
 * Load app configurations
 */
module.exports = _.extend(
    require('./env/all'),
    require('./env/' + process.env.NODE_ENV) || {}
);
console.log(module.exports)
```

Rerun the app by typing `PORT=4000 NODE_ENV=development grunt`. The console shows:

```
app: { title: 'MeanJS - Development Environment' }
```

It appears that the JSON structure in config/env/development.js is overwriting the structure in config/env/all.js rather than merging with it. Luckily, you can make a quick change to config/config.js to get the results you expect.

Change the function call from `_.extend` to `_.merge`. When you rerun the app again, you should see the results you expect:

```
app:
   { title: 'MeanJS - Development Environment',
     description: 'Full-Stack JavaScript with MongoDB, Express, AngularJS, and Node.js',
     keywords: 'MongoDB, Express, AngularJS, Node.js' },
```

If you click **View > Source** on the home page in your browser, you can see the config values merged with the HTML template, as shown in Listing 17.

Listing 17. The corrected merged results in HTML

```
<head>
  <title>MeanJS - Development Environment</title>

  <!-- Semantic META -->
  <meta name="keywords" content="MongoDB, Express, AngularJS, Node.js">
  <meta name="description" content="Full-Stack JavaScript with MongoDB, Express, AngularJS, and Node.js">
```

Now it's time to move from the server side to the client side to complete your walking tour of this MEAN application.

Exploring the client side of the MEAN stack

Key pieces of the home page (as defined in app/views/layout.server.view.html, shown in Listing 18) are filled in by AngularJS on the client side.

Listing 18. app/views/layout.server.view.html

```
<body class="ng-cloak">
  <header data-ng-include="'/modules/core/views/header.client.view.html'"
  class="navbar navbar-fixed-top navbar-inverse"></header>
  <section class="content">
    <section class="container">
      {% block content %}{% endblock %}
    </section>
  </section>
```

Recall that the app directory contains the Express server-side portion of the MEAN application. You have two hints that the `header` is managed on the client side by AngularJS. First, any time you see an HTML attribute with `ng` in it, that's an indication that A**ng**ularJS manages it. Second, and more pragmatically, the app directory that contains all of your server-side code doesn't include a modules directory. By eliminating the server side as a possible solution, that leaves the client-side source code that's in the public directory. The modules directory is clearly under the public directory, as shown in Listing 19.

Listing 19. The public (client-side) directory structure

```
$ tree -L 1 public/
public/

|--- application.js
|--- config.js
|--- lib
|--- modules
```

If you look under lib, you'll see several third-party libraries:

Listing 20. The public/lib directory for third-party libraries

```
$ tree -L 1 public/lib
public/lib
|--- angular
|--- angular-animate
|--- angular-bootstrap
|--- angular-cookies
|--- angular-mocks
|--- angular-resource
|--- angular-sanitize
|--- angular-touch
|--- angular-ui-router
|--- angular-ui-utils
|--- bootstrap
|--- jquery
```

Recall that these are the same libraries specified in bower.json.

But if you go snooping around the modules directory, you'll find the modules/core/views/header.client.view.html template specified in app/views/layout.server.view.html.

Listing 21. modules/core/views/header.client.view.html

```
<div class="container" data-ng-controller="HeaderController">
  <div class="navbar-header">
    <button class="navbar-toggle" type="button" data-ng-click="toggleCollapsibleMenu()">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a href="/#!/" class="navbar-brand">MeanJS</a>
  </div>
```

If you change the value of the `class="navbar-brand"` anchor from `MeanJS` to something else, the change is reflected immediately in the browser after you save the file. But the path to the main payload — the primary content of the home page — is more circuitous. Look once more at app/views/layout.server.view.html, shown in Listing 22.

Listing 22. app/views/layout.server.view.html

```
<body class="ng-cloak">
  <header data-ng-include="'/modules/core/views/header.client.view.html'"
  class="navbar navbar-fixed-top navbar-inverse"></header>
  <section class="content">
    <section class="container">
      {% block content %}{% endblock %}
    </section>
  </section>
```

Inside the `container` section is a `block` named `content`. Remember the innocuous app/views/index.server.view.html:

```
{% extends 'layout.server.view.html' %}

{% block content %}
  <section data-ui-view></section>
{% endblock %}
```

This `block content` contains an empty section with a `data-ui-view` attribute. This attribute is used by the client-side AngularJS router. Look at public/modules/core/config/core.client.routes.js, shown in Listing 23.

Listing 23. app/views/index.server.view.html

```
'use strict';

// Setting up route
angular.module('core').config(['$stateProvider', '$urlRouterProvider',
  function($stateProvider, $urlRouterProvider) {
    // Redirect to home view when route not found
    $urlRouterProvider.otherwise('/');

    // Home state routing
    $stateProvider.
    state('home', {
      url: '/',
      templateUrl: 'modules/core/views/home.client.view.html'
    });
  }
]);
```

It's not intuitively obvious, but this client-side router injects the modules/core/views/home.client.view.html template (shown in Listing 24) into the section of app/views/index.server.view.html containing the `data-ui-view` attribute, when the URL is `/`. The contents of this template should match what you see in your browser when you're on the home page of the MEAN application.

Listing 24. modules/core/views/home.client.view.html

```
<section data-ng-controller="HomeController">
    <h1 class="text-center">THANK YOU FOR DOWNLOADING MEAN.JS</h1>
    <section>
        <p>
          Before you begin we recommend you read about the basic building
          blocks that assemble a MEAN.JS application:
        </p>
```

Conclusion

In this article, you walked through the key pieces of a MEAN application, step-by-step. On the server side, you learned that the path started with Express routes, which in turn invoke Express controller functions, which in turn merge JSON data with a Swig template and return it to the client. But the process doesn't end there. On the client side, AngularJS routes pick up and inject HTML templates into the main page.

Next time, you'll take a deeper look at the role MongoDB plays along with AngularJS by exploring the articles portion of the application. You'll also see how testing on both the server side and client side ensures that unexpected behavior in production is minimized.

Until then, have fun mastering the MEAN stack.

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.
- 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*.