Take a walk through the small pieces, loosely joined of the MEAN stack's testing infrastructure. No application, MEAN or otherwise, is ready for production without passing a rigorous, comprehensive test suite. See how to use Karma, Mocha, Jasmine, and istanbul — with the help of PhantomJS — to test the UGLI app.

View more content in this series

The User Group List and Information (UGLI) app has come a long way since you started it. You're storing local data in the application and pulling in remote data through RESTful web services. The application sports a mobile-ready responsive web design, and it's semantically marked up to make the most of search engine optimization (SEO). For authentication, users can either create a local, dedicated account or (via OAuth) reuse an existing account stored elsewhere.

But you wouldn't feel comfortable putting the UGLI app into production without a solid testing suite as your safety net, would you? Of course not. That would be professionally irresponsible. I agree with Neal Ford (author and international speaker), who calls testing "the engineering rigor of software development." Whenever I start an engagement with new clients, I look at their test suite before anything else — even their design documents. The quality, quantity, and comprehensiveness of their tests has a direct correlation to the maturity of their software development process. A healthy, actively maintained test suite is the bellwether of a project's overall health. Similarly, any framework that places a premium on testability moves to the top of my list. AngularJS was written by testers, and I'm hard-pressed to think of another modern web framework that's easier to test. The MEAN.JS stack extends the out-of-the-box testability to include testing of server-side logic.

"AngularJS was written by testers, and I'm hard-pressed to think of another modern web framework that's easier to test."

At the beginning of this series, I walked you through the basic building blocks of the MEAN stack — the small pieces, loosely joined that make up the production components of your app. Now it's time to do the same for the various frameworks and libraries you'll use to test your app and make it production-ready. I'll introduce you to Karma: a pluggable test runner that makes it trivial to run tests written in any testing framework across any number of real web browsers (including
smartphones, tablets, and smart TVs) and return the results in a wide variety of formats. Along the way, you'll use Jasmine for client-side testing, Mocha for server-side testing, and istanbul for code coverage.

Running your tests

Because you've been using the Yeoman generator that ships standard with the MEAN.JS framework, you already have several generated tests in place. Type grunt test to run them. You should see results similar to those in Listing 1.

Listing 1. Running the generated tests

```bash
$ grunt test
Running "env:test" (env) task
Running "mochaTest:src" (mochaTest) task
  Application loaded using the "test" environment configuration
Running "karma:unit" (karma) task
INFO [karma]: Karma v0.12.31 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.8 (Mac OS X)]: Connected on socket 6zkU-H6qx_m2J6lY4z38 with id 51669923
PhantomJS 1.9.8 (Mac OS X): Executed 18 of 18 SUCCESS (0.016 secs / 0.093 secs)
Done, without errors.
```

Don't be concerned if you have errors or warnings; the tests are scaffolded out to match the models and controllers as they're initially implemented. If you've been making changes to the code under test (CUT) and not updating the corresponding tests, you can expect errors.

I'm thrilled every time a unit test fails. Unit tests are the circuit breakers of your code base. In your house, you put circuit breakers in between the power grid and your expensive personal electronics. That way when a potentially damaging power surge comes in over the wire, you stand to lose a 35-cent circuit breaker instead of a $3,500 laptop. Similarly, every breaking unit test is an error that you see and your users don't.

Take a moment to fix your broken tests if you can. A common source of errors is a test's reliance on deleted or changed field names. The server-side tests are in app/tests. The client-side tests are stored in the test directory of each public/module. If you can't immediately see the source of the error in your test, don't delete the test; simply move it out of the directory tree temporarily.

Now that you can make a clean test run, it's time to deconstruct the process.

Understanding the Grunt test task

As you typed grunt test, I hope that you asked yourself, "Hmm, I wonder how Grunt is running those tests." Grunt, as you know, runs your build script. Open gruntfile.js in your text editor and scroll all the way to the bottom of the file. You can see the test task being registered:

```javascript
// Test task.
grunt.registerTask('test', ['env:test', 'mochaTest', 'karma:unit']);
```
The first argument of grunt.registerTask is the name of the task — in this case, test. The next argument is an array of dependent tasks. The test task first sets up values specific to the test environment, then runs all of the server-side tests written in Mocha, and finally kicks off the client-side tests via Karma.

Scroll up a bit in gruntfile.js until you find the env task:

```javascript
env: {
  test: {
    NODE_ENV: 'test'
  }
},
```

This task does little more than set the NODE_ENV variable to test. Recall that this variable helps Grunt determine which environment-specific settings — in this case, config/env/test.js — to merge with the common settings in config/env/all.js.

If you look at config/env/test.js in a text editor (as shown in Listing 2), you'll see a custom MongoDB connection string, along with hooks for all of the Passport settings for various OAuth providers:

**Listing 2. config/env/test.js**

```javascript
'use strict';

module.exports = {
  db: 'mongodb://localhost/test-test',
  port: 3001,
  app: {
    title: 'Test - Test Environment'
  },
  facebook: {
    clientID: process.env.FACEBOOK_ID || 'APP_ID',
    clientSecret: process.env.FACEBOOK_SECRET || 'APP_SECRET',
    callbackURL: 'http://localhost:3000/auth/facebook/callback'
  },
  google: {
    clientID: process.env.GOOGLE_ID || 'APP_ID',
    clientSecret: process.env.GOOGLE_SECRET || 'APP_SECRET',
    callbackURL: 'http://localhost:3000/auth/google/callback'
  },
  // snip
};
```

This section would be an ideal place for you to point Passport to a mock implementation of the Meetup authentication strategy. That way, you don't need to rely on having actual users set up and make actual OAuth requests to Meetup.com during your test run.

After the test environment is configured, Grunt runs all of your server-side tests written in Mocha. Here's the mochaTest task:
mochaTest: {
    src: watchFiles.mochaTests,
    options: {
        reporter: 'spec',
        require: 'server.js'
    }
},

Knobs and dials
For more information on the knobs and dials you can twist when running your Mocha tests, see the plugin documentation for grunt-mocha-test.

Why are the server-side tests written in Mocha instead of Jasmine? Mocha's maturity, extensibility, and plugins make it one of my favorite testing frameworks. Mocha is a strong choice for testing things like Express routes, controllers, and MongoDB interaction. Although Mocha can easily run tests both in Node.js and in-browser, the AngularJS team prefers Jasmine for in-browser testing. Jasmine is more optimized for client-side testing, so the MEAN.JS developers took a best-of-breed approach and chose a strong server-side testing framework to test the server side, and a strong client-side testing framework to test the client side of things. You should feel comfortable swapping out either or both frameworks for the testing tools of your choice.

Because server-side tests are (by definition) not run in-browser, the Mocha tests are not kicked off by Karma. The Jasmine tests — the final part of the Grunt test dependencies — are triggered by the karma task:

karma: {
    unit: {
        configFile: 'karma.conf.js'
    }
}

Before I move on to deconstructing the karma.conf.js file, open package.json in a text editor. In addition to the runtime modules listed in the dependencies block, you can see several build-time dependencies listed in the devDependencies (short for developer dependencies) block. This block is specifically where the Grunt plugins related to Mocha and Karma are declared and installed when you type npm install:

"devDependencies": {
    "grunt-env": "~0.4.1",
    "grunt-mocha-test": "~0.10.0",
    "grunt-karma": "~0.8.2",
    "load-grunt-tasks": "~0.4.0",
    // snip
}

Introducing Karma
Karma is the only test runner I know of that is backed up by a master's thesis. A more succinct version of the thinking behind Karma is the project's mission statement:

    Things should be simple. We believe in testing and so we want to make it as simple as possible.

Mastering MEAN: Testing the MEAN stack
And simple it is. Karma allows you to write your tests in the framework of your choice. Whether you prefer the test-driven-development (TDD) style of QUnit or the behavior-driven-development (BDD) style of Jasmine, Karma will happily run tests written in any style. (Karma also offers first-class support for Mocha if you would prefer to use a single testing framework for writing both server- and client-side tests.)

Seasoned web developers know how important it is to test their apps across a wide variety of browsers. The core JavaScript language is remarkably consistent across browsers, but Document Object Model (DOM) manipulation and ways to make Ajax requests are far from standardized. Mainstream libraries like jQuery and AngularJS do a great job of polyfilling over browser incompatibilities, but that shouldn't lull you into a false sense of complacency. One test is worth a thousand opinions, and having proof that your app works as intended in a specific browser is far preferable to simply assuming that it will.

Karma offers several plugins that you can use to launch a real browser on demand, run the full test suite, and then shut down the browser upon completion. That capability is convenient for running the tests locally in the browser of your choice, but it can be limiting if the tests are being kicked off by a headless continuous integration server such as Jenkins, Hudson, or Strider.

Thankfully, you can stand up a long-running Karma server and capture browsers on remote devices. Capturing a browser is as simple as visiting the URL of your Karma server in the browser. If the browser supports Web Sockets (caniuse.com shows support in every mainstream, modern browser), the Karma server will maintain a long-running, durable connection with the device. As new tests are added to the suite, the Karma server will serialize them across the wire to the remote browser, run them, and return the results.

But what good is running the test suite if you can't quantify the results? Karma offers plugins for several different reporters. A reporter can be as simple as something that prints out a dot on the command line for each passing test. Or a reporter can yield fully formatted HTML, or emit raw JUnit-compatible XML that can be transformed into the output of your choice.

Testing frameworks, browser launchers, and results reporters are all defined in the karma.conf.js file.

**Understanding karma.conf.js**

Open karma.conf.js in a text editor, as shown in Listing 3. In the file, you'll find clearly labeled settings for frameworks, files, reporters, and browsers.

**Listing 3. karma.conf.js**

```javascript
'use strict';

/**
 * Module dependencies.
 */
var applicationConfiguration = require('./config/config');

// Karma configuration
module.exports = function(config) {
  config.set({
```
Refer back to package.json. In that file you can find corresponding entries in the `devDependencies` block for the various Karma plugins:

```json
"devDependencies": {
    // snip
    "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"
}
```

Because all of the scaffolded client-side tests are written in Jasmine, I recommend leaving the `frameworks` array as it stands. But as you’ll see later in this section, you can feel comfortable adding and removing browsers at will.
Introducing PhantomJS

If you're a web developer but aren't familiar with the PhantomJS browser, you're in for a treat. PhantomJS is one of a web tester's best friends.

It's easy to be tricked into thinking of web browsers as monolithic applications identified by familiar brand names: Firefox, Chrome, Safari, Opera, and Internet Explorer. Those brand names are merely a convenient way to describe a specific collection of technologies that include a rendering engine (for HTML and CSS), a scripting engine (for JavaScript), and a plugin subsystem.

"Now that you're a seasoned MEAN developer, you're already intimately familiar with plucking components out of a browser and running them headlessly. You should feel right at home running a headless render kit for testing."

Once you recognize browsers as a loose collection of rendering kits and scripting engines, a whole new level of understanding opens up. For instance, the Netscape Navigator browser had a 90+ percent market share when version 2.0 was released in the mid 1990s. IE took over that market lead just a few years later. But in recent years, a render kit — WebKit — rather than a browser enjoys a majority market share. That's because until recently (see the WebKit, meet Blink sidebar), WebKit powered Safari, Mobile Safari, Chrome, the Android browser, the BlackBerry browser, Kindle devices, PlayStation, Samsung smart TVs, LG smart TVs, Panasonic smart TVs, and more. Even though these applications and devices were all assembled by different companies and projects, they share a common render kit for displaying HTML and styling it with CSS.

WebKit, meet Blink

Google made news in 2013 with the announcement that it was forking the WebKit project and creating a separate render kit named Blink. Soon after, Opera announced that it would stop developing its custom render kit and adopt Blink instead. (This list shows the underlying render kit for all major browsers.)

Apple, which originally forked WebKit from the KHTML rendering engine in 2005, continues to actively develop and maintain the project. (See the WebKit entry in Wikipedia for more information.)

So, what does this have to do with PhantomJS? The PhantomJS website tells us: PhantomJS is a headless WebKit scriptable with a JavaScript API.

A headless service doesn't require a monitor or a GUI. That sounds perfect for running browser-based unit tests on a monitorless continuous integration server, doesn't it? (The SlimerJS project offers a similar capability: running a headless Gecko render kit to test the page rendering that occurs in the Firefox browser.)

Now that you're a seasoned MEAN developer, you're already intimately familiar with plucking components out of a browser and running them headlessly: Node.js is Google Chrome's scripting engine (V8) running headlessly. You should feel right at home running a headless render kit for testing.
Looking back at karma.conf.js, you can see that PhantomJS is in the browsers array. Now you understand how all of the Jasmine client-side tests could run and pass in a browser without you seeing a GUI launch.

**Configuring Karma to launch additional browsers**

Karma offers launchers for all major browsers. If you look back at the devDependencies block of package.json, you can see that launchers are already installed for Firefox and Chrome. If you have those browsers installed on your computer, add them to the browsers array in karma.conf.js and type grunt test to run your test suite in the newly added browsers.

I encourage you to visit the npm website and search for karma launcher to see a list of all supported browsers. You install each launcher and add it to package.json by typing npm install karma-xxx-launcher --save-dev. Once the launcher is installed, add it to the browsers array in karma.conf.js and rerun your tests.

**Capturing browsers that can't be launched**

Karma launchers are typically used to launch browsers that are co-located on the same computer. Recall that Karma can be used to run tests on remote browsers also — think smartphones, tablets, and smart TVs. Any browser that supports Web Sockets can be captured by Karma and used as a test target.

To capture a remote browser, you must first leave the Karma server up and running between test runs. To leave the Karma server running permanently, change the singleRun value to false in karma.conf.js:

```javascript
// Continuous Integration mode
// If true, it capture browsers, run tests and exit
singleRun: true
```

If you reboot either the Karma server or any of the captured browsers, they'll try to reconnect and rerun all of the tests.

Now that the Karma server is up and running, visit it in a remote browser at the URL http://your.server.ip.address:9876. That's all it takes to capture a nonlaunchable browser using Karma.

**Adding additional Karma reporters**

Now that you're comfortable adding additional tests and browsers, consider adding additional reporters to capture and display the results of the tests.

To start, add the dots reporter to the reporters array in karma.conf.js. The next time you type grunt test, you'll see a series of dots fly across your screen — one for every passing test.

The dots are cute but ephemeral. How will you know how many tests passed unless you're watching the screen as they run? Perhaps installing a reporter that's a bit more durable is in order.
The *karma-html-reporter* is most likely what you're looking for. As the example in Figure 1 shows, you get detailed verbal results for each test, nicely formatted in HTML.

**Figure 1. Report generated by karma-html-reporter**

![Report](image)

To install *karma-html-reporter*, type `npm install karma-html-reporter --save-dev`. Then to configure it, edit `karma.conf.js` like so:

```javascript
reporters: ['progress', 'html'],
htmlReporter: {
  outputDir: 'karma_html'
},
```

See the *karma-html-reporter* package details for the full set of configuration options.

If you would prefer raw XML output instead of polished HTML output, consider installing the *karma-junit-reporter*. To install it, type `npm install karma-junit-reporter --save-dev`. Then configure it in `karma.conf.js` as shown at the project site.

You typed `karma launcher` at the npm website to search for additional launchers. You should feel equally comfortable typing `karma reporter` to find additional Karma reporters.

**Showing code coverage with Karma and istanbul**

No testing infrastructure is complete without showing test code coverage. The previous reports only showed you the tests that passed and failed — they didn't show you the tests that you forgot
to write. A good code-coverage tool shows you line-by-line which parts of your code base were visited by unit tests, and more important, which lines of code haven’t been visited by a unit test yet.

If you install the `karma-coverage` plugin (which uses the `istanbul` library) by typing `npm install karma-coverage --save-dev` and configure it based on the `instructions`, you’ll get a set of beautiful reports that display every line of code in your application, as in Figure 2.

**Figure 2. Coverage report**

![Coverage report](image)

The green lines have been touched by a unit test, and the red lines are the lines waiting for a future unit test.

**Mocking dependencies**

A hallmark of well-written unit tests is their independence. They should never rely on actual databases or make actual HTTP calls to live web services. Thankfully, mocking out these dependencies is a time-honored way of running your tests.

Instead of making actual Ajax calls in your client-side Jasmine tests, consider using the `$httpBackend` mock service included with AngularJS.
Instead of relying on an actual MongoDB database for testing, consider using Mockgoose—a pure in-memory drop-in replacement for Mongoose (and MongoDB) written expressly for testing purposes.

**Running end-to-end tests with Protractor.js**

Up to this point, you've been running unit tests. Unit tests — by definition — are tests that don't rely on a GUI. Unit tests are for the non-UI parts of your code base.

But what about testing all of the typing and button clicking that typical users will perform when they're using your app? To test that type of behavior, you can install Protractor.js.

The Protractor home page has a full set of instructions and examples. Here's the short version:

```
type npm install protractor --save-dev to install the library. Next, you write Jasmine tests that visit specific URLs and interact with specific components on the page. Listing 4 shows an example of a Protractor test from the project's home page.
```

**Listing 4. A Protractor test**

```javascript
describe('angularjs homepage todo list', function() {
  it('should add a todo', function() {
    browser.get('http://www.angularjs.org');
    element(by.model('todoText')).sendKeys('write a protractor test');
    element(by.css('[value="add"]')).click();

    var todolist = element.all(by.repeater('todo in todos'));
    expect(todolist.count()).toEqual(3);
    expect(todolist.get(2).getText()).toEqual('write a protractor test');
  });
});
```

As you've probably surmised, this test visits the AngularJS home page, finds the todoText element, types in a test string, and clicks the add button. Then it runs a series of assertions to ensure that the expected values appear.

**Conclusion**

As I said earlier, and as I often say, one test is worth a thousand opinions. But cheeky rejoinders work only if you can back them up with solid software practices. If you put the lessons learned from this article into place, you'll be well on your way toward the "engineering rigor" required to be a part of this fast-paced, ever-changing software ecosystem.
Related topics

- **AngularJS unit testing**: Check out the unit testing section of the AngularJS Developer Guide.
- **MEAN.JS testing documentation**: Take a look at the testing sections in the MEAN.JS docs.
- **Karma**: Find out all about Karma and see the available plugins for Karma at the project site.
- **PhantomJS**: Learn more about this scriptable, headless browser.
- **Mocha**: Visit the Mocha home page for documentation and examples.
- **Jasmine**: Check out the Jasmine documentation.