

02/01/2019 • 11 minutes to read

In this article

[Getting Tested](#)

[Testing APIs](#)

[Wrapping Up](#)

January 2016

Volume 31 Number 1

[The Working Programmer] How To Be MEAN: Test Me MEANly

By [Ted Neward](#) | January 2016



Howdy, "Nodeists." Welcome back to the continuing saga of input, output, databases, users, UIs, APIs and parallel universes. (See the first part in this series [msdn.com/magazine/mt185576] if you missed that reference.)

In the previous installment (msdn.com/magazine/mt595757), the application had grown to the point where the API endpoint being built (a super-simplified "persons" database) rounded out its CRUD facilities to include the CUD, where the previous installment had provided just the "R." Building out in small steps is important, certainly, but it's nice to have something that can see changes to it and reflect them back when asked about them.

This brought up an uncomfortable point. When building a Web API such as this, "easy testing," in which a developer can just fire up the Web page and eyeball it to determine whether something is working correctly, isn't an option here. Well, sure, as pointed out last time, using tools such as cURL, Postman or Runscope can help make it easier to eyeball it, but frankly, trusting eyeballs isn't really the point. The application needs some automated tests to help make sure everything is running smoothly and correctly. It


should be a test suite that's easy to run, covers the code well and has all the other characteristics that so many others have documented far better than I can.

One note before I go too deep down this path, though. There's a bit of debate around "unit" tests, which have intricate knowledge of the inner workings of the code and (usually) require some level of mocking (in this case, the Express request and response objects, for example). These are supposed to be small, focused and execute quickly so they can be used as part of a build process (right after compilation, for compiled languages like C#) without slowing down programmers and jerking them out of the flow of their thoughts. Another view is that of "integration" or "feature" tests, which are more "external" tests, meaning the test relies on the frameworks to do their thing and treats the layer as a more opaque entity. For Web APIs, I tend to favor the latter; I've found that trying to write unit tests over Express controllers can often be more work than benefit to get all the mocking right. Given that controllers are generally intended to be single-purpose, anyway, and there are few, if any, bugs I've found in the Express layer that interferes with testing my own code, I've found external tests to be a better value in the bang-for-your-buck discussion.

With that out of the way, it's time to write some tests that exercise the persons API I've built so far. And to that end, there are two Node.js packages that immediately come to mind: mocha and supertest.

Getting Tested

Of course, as with all things Nodeish, life begins with npm; in this case, both mocha and supertest will need to be added to the project via npm install:

JavaScript	 Copy
<pre>npm install --save-dev mocha supertest</pre>	

(While it's traditional to do each separately, there's no reason not to do them together like that, particularly in this case where I already know I want to use both.)


Before I pass beyond that, note the `--save-dev` argument: this puts the two into the `devDependencies` section of the `package.json` file, meaning they're development-only dependencies and, as a result, when cloud hosts (such as Microsoft Azure, for example) prepare the code for execution in production, they'll leave those out.

The Node.js community treats its testing frameworks much as it does the rest of its projects: as small bits, looking to be composed together in developer-defined ways, to create a whole that is, presumably, pleasing to the developer. In this case, testing starts with the mocha package, which is a testing package designed to make "asynchronous

testing simple and fun,” according to its Web site. It will act as the basic “scaffolding,” if you will, of the testing effort. But because the things under test will be Web APIs (and because I don’t want to be writing all this low-level HTTP code), the second library, `supertest`, will add some useful HTTP-specific testing behavior. There’s one more optional part to the set, the `should` library, which I choose to add because I prefer Behavior-Driven Development (BDD)-style should assertions. But the `assert` package, which comes with `mocha`, is also there for those who prefer traditional assert styles.

Once both packages are installed, testing begins, not surprisingly, by writing the first test. Although not required, it’s common for Node.js projects to create a separate subdirectory in which the test code lives, called, not surprisingly, `test`. Into this directory goes the first test file, `getTest.js`, as shown in **Figure 1**.

Figure 1 Setting Up Some Baseline Tests Just for Fun

JavaScript	 Copy
<pre>var supertest = require('supertest'); var express = require('express'); var assert = require('assert'); var should = require('should'); describe('Baseline tests', function() { describe('#indexOf()', function () { it('should return -1 when the value is not present', function () { assert.equal(-1, [1,2,3].indexOf(5)); assert.equal(-1, [1,2,3].indexOf(0)); [1,2,3].indexOf(5).should.equal(-1); [1,2,3].indexOf(0).should.equal(-1); }); }); });</pre>	

There are a couple of things that require some explanation before I move on. The `describe` methods are simple output/execution pairs—the string is the message that should be displayed in the testing console (typically the Command Prompt or Terminal window) and the function is what should be run as the test. Notice that `describe` blocks can be nested—this can be helpful to split up tests along logical lines, particularly because you can then use the `skip` function to turn off whole blocks of them should that be necessary. The `it` function then describes a single test—in this case, to test that the JavaScript array’s `indexOf` method functions as it’s supposed to, and again, the string passed into “it” is the description of the test printed to the test console.

The nested function inside the `it` call is, not surprisingly, the actual test, and here I’ve chosen to show both the traditional `assert` style of assertions, as well as the more BDD-like `should`-style assertions, which is a more fluent API. Frankly, there are other options (the `mocha` Web page lists two more besides these two), but because I like `should`,


that's what I'm going to use. However, the `should` package will need to be installed via `npm`, whereas `assert` comes along for the ride already with `mocha` and `supertest`.

For a quick test, just type `"mocha"` into the directory containing `app.js` and it will pick up the tests in the `test` subdirectory automatically. But Node.js has a slightly better convention, which is to populate the `scripts` collection in the `package.json` file to have a set of command-line parameters to `npm` for easy execution; `"npm start"` can then start the server, and `"npm test"` can run the tests without the administrator (or cloud system) having to know what was used to build the code. This just requires filling out the `"scripts"` collection accordingly inside of `package.json`:

JavaScript	 Copy
<pre>{ "name": "MSDN-MEAN", "version": "0.0.1", "description": "Testing out building a MEAN app from scratch", "main": "app.js", "scripts": { "start": "node app.js", "test": "mocha" }, // ... }</pre>	

So now, a quick `"npm test"` will launch the tests and find out that, yes, `indexOf` works correctly.

Having done that, let's turn those tests off. (Obviously, you could remove them entirely, but I like having a baseline that I know works, in case something fails with my setup.) That's easily done by using `skip` as part of the chained `describe`, as shown here:


JavaScript	 Copy
<pre>describe.skip('Baseline tests', function() { describe('#indexOf()', function () { it('should return -1 when the value is not present', function () { assert.equal(-1, [1,2,3].indexOf(5)); assert.equal(-1, [1,2,3].indexOf(0)); [1,2,3].indexOf(5).should.equal(-1); [1,2,3].indexOf(0).should.equal(-1); }); }); });</pre>	

This will now skip these tests—they're still listed to the test console, but there's no checkmark next to them to indicate the tests were run.

Testing APIs

That was fun, but `indexOf` was already pretty well tested; the `GET /persons` endpoint isn't ... yet.

This is where `supertest` comes into play; using it, you can create a tiny HTTP agent that knows how to query the endpoint (in this case, using the server settings of "localhost" and port 3000, as established by the app code written before this) and verify its results. This means there are a couple of things that need to be added; first, the app code has to be loaded and run as part of the test and that's done by requiring the `app.js` file in the parent directory from the first line of the test:

JavaScript	 Copy
<pre>require("../app.js")</pre>	

This will get the app started (which is easy to spot if the `DEBUG` environment variable is set to `app`, remember) and running.

Then, using the `supertest` library to create a request object against the local host server and port 3000, you can use that to issue a request against the `/persons` endpoint with a person ID of 1 (making it `/persons/1`, remember) and verify that it comes back OK (a 200 status code), comes back as JSON and the body of the JSON contains what's expected, as shown in **Figure 2**.


Figure 2 Get Me All the Persons!

JavaScript	 Copy
<pre>var request = supertest("http://localhost:3000"); describe('/person tests', function() { it('should return Ted for id 1', function(done) { request .get('/persons/1') .expect(200) .expect('Content-Type', /json/) .expect(function(res) { res.body.id.should.equal(1) res.body.firstName.should.equal("Ted") res.body.lastName.should.equal("Neward") res.body.status.should.equal("MEANing") }) .end(done); }); });</pre>	

Notice how the request object has a fluent API of its own, using a verb method style similar to that of Express, so `get('/persons/1')` translates into a GET request against that URL. The successive expect methods, however, do exactly what it sounds like: They expect a particular kind of result, and if that isn't the case, it fails the entire test. However, the last end method is important to note because it makes use of the one parameter passed into "it": the "done" object, which is an opaque callback that signals that the test is finished. Because supertest does all of this testing serially but asynchronously (in the Node.js fashion), there needs to be some signal that the test is completed—if done isn't used, then after two seconds (per each test), supertest will assume the test timed out and signal an error.

Last, because space is getting short for this one, let's verify that you can add persons to the database by using the post method, and sending along a new JSON object containing the person instance that needs to be added to the system, as shown in **Figure 3**.

Figure 3 Verifying That Adding a Person Actually Works

JavaScript	 Copy
<pre>it('should allow me to add a person', function(done) { request .post('/persons') .send({ 'firstName': 'Ted', 'lastName': 'Pattison', 'status': 'SharePointing' }) .expect(200) .expect('Content-Type', /json/) .expect(function(res) { should.exist(res.body.id); res.body.firstName.should.equal("Ted"); res.body.lastName.should.equal("Pattison"); }) .end(done); })</pre>	

Easy, right? Note that the first line of the body-verification expect is to test that the id field exists. However, the should library can't trust that it won't be undefined, and calling a method or property on an undefined generates an exception, so you need to use should directly to verify it exists. Aside from that, though, it reads nicely.

The two other methods that require testing, PUT and DELETE, are pretty easy to write. The PUT endpoint test will use put instead of post, and the URL will need to point to a singular person URL (`/persons/1` for the person "Ted" "Neward," for example). The JSON body to send would look the same as the POST case, with differing values (because the test is to see if those updates will be accepted), and then test that the returned JSON body reflects those updates (because the test is also to see if those updates will be

reflected in the new object returned). And the DELETE endpoint test will use the delete method, passing in the URL for a specific person (/persons/1), and this time, pass in neither a JSON body nor expect one returned. (Although it seems a common API convention to return the deleted object from a DELETE endpoint, to be honest, I personally have never really seen much point to it, and have never used the returned JSON body for any reason.)

With those five tests in place, and armed with the newly enhanced npm script tag in package.json, it remains only to run npm test, watch the rest runner do its thing, see no errors, do a source code commit-and-push, and update the Azure environment with the new code.

Wrapping Up

There's a bunch of material about mocha, supertest and should that I don't have space to get into—for example, like most testing frameworks, mocha supports before and after hooks that will run around each test; also, the supertest library supports cookie-based requests, which is useful when working with HTTP endpoints that use cookies to maintain some kind of session state. Fortunately, the documentation around each of those libraries is pretty good, and examples floating around the Web are legion. Alas, for now, it's time to step back and say ... happy coding!

Ted Neward *is the principal at Neward & Associates, a Seattle-based polytechnology consulting firm. He has written more than 100 articles, is an F# MVP and has a dozen books to his name. Reach him at ted@tedneward.com if you're interested in having him come work with your team, or read his blog at blogs.tedneward.com .*

Thanks to the following technical expert for reviewing this article: Shawn Wildermuth