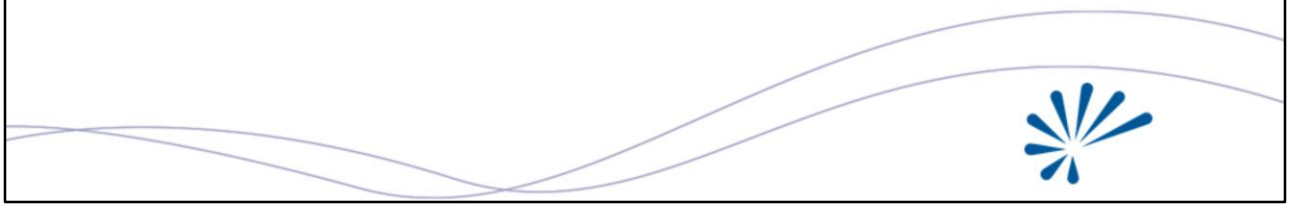
improving
It's what we do. ™

Don't Get Burned

A Hot Take on Javascript Unit Testing

Kevin Cavnar-Johnson



Introductions

- Hello!
- About Me
  - Kevin Cavnar-Johnson
  - Sr. Consultant @ Blinds.com
  - 10 years in software
- About you
  - Expectations for the talk

Second, I'd like to say thanks for coming out, despite the occasional still high waters. I hope everyone survived floodpocalypse intact.

This talk is inspired by several days I spent in the trenches getting javascript testing up and running on a huge codebase. I made some good decisions, and a few very bad decisions. And I was not the first person to try! I was the third, working from the lessons learned by the past.

One motto I have is that anybody promising a hard and fast rule that works all the time is probably selling you something. This talk is focused on providing a framework for making (and then justifying) decisions. We'll talk about the questions to ask yourself, and factors to consider to help make the decisions that are right for you. And hopefully provoke some good conversation, ideally with only a few religious wars.

After that, we're going to go over a couple demos that walk through the basics, showing some gotchas and unexpected results, and some examples of specific things that you are going to want to do:

1. Injecting module dependencies
2. Faking Ajax responses

3. And finally
3. jQuery/Dom manipulation

One thing I am not going to discuss, but I do provide a link at the end, is go tool by tool and list pros and cons of each one. That's super useful information but would be incredibly dull to listen to.
Depending on the number of people

General comfort level with Javascript?
Preferred frameworks?
AMD?

Step 1, possibly surprisingly, is not picking a test framework. Step 1 is to decide whether to test at all.

Now the purists are already going to be lighting torches and sharpening pitchforks, but this talk is going to focus more on the actual world, so

Deciding What To Test

- Should you start testing?
  - Greenfield
  - Untested Legacy
  - Weakly tested Legacy
- Build Process

In a greenfields scenario, you can live the disciplines of TDD. The power is yours!

You're working with an existing javascript codebase which is untested
You're starting a new project from scratch

Any other scenario, you don't have to make any decisions. If you have testing set up all the way through running as part of your CI build process, I don't care how much you dislike it, the right call is to stick with what you have that is working. Working tests >

What if your tests are so weak that they aren't providing value/are so brittle that their maintenance is costing more than their value. The main question I'd ask there is, when tests fail, are they because the actual requirements changed, or because an implementation detail changed. If the problem is that the code is just volatile, that's a fact of life. If the tests are coupled to the implementation: i.e. unrelated changes break tests, you probably need to torch them and start over.

Quick poll: what's more valuable, writing new code or deleting old code?

Tests are code, and have to be maintained

This talk is mostly about bringing tests to an untested codebase, because I'm guessing we've all written some Javascript, and haven't tested it

This is also where you should start to think about your build process. A testing framework is going to add a bunch of new dependencies, which means more places to break. We're gonna come back to this in a bit, but take a second to think about it.

Deciding What To Test

- Where to start?
- Factors to consider
  - Degree of Difficulty/Danger
  - Number of dependencies
  - Code Volatility

So there's a few ways you can work when it comes to identifying where to start testing.
Look for the most dangerous parts of your codebase. You kinda know these right? The pages that make you wince when you have to deal with them. Positives: these tests provide the most value. Drawback: These are very likely to be a giant pain to write, and combine it with the pain of setting up a testing framework from scratch, and you have a recipe for getting burnt out after working on it for two days and having nothing to show for yourself

Look for the easiest things to test. The pages with the fewest module dependencies. Relatively straightforward models and the like. Positive: you can generally get these done pretty easily. Drawback: you finish them and after two days of work, you are staring at a test file that tests little to no business logic.

Look for the most stable code to test. This is really more of a second dimension to consider along with degree of difficulty. You really want to avoid writing a bunch of incredibly difficult tests that then immediately get invalidated next sprint.

Obviously you want to look for a unicorn, a module that includes significant, stable logic with a minimum number of dependencies. Easier said than done, but in my experience this is crucial to the long term success of a testing initiative.

Deciding What To Test

- Things to watch out for
  - No constructor/does lots of work in constructor
  - jQuery selectors/ajax requests
- Things to look for
  - Low coupling, high cohesion
  - Functions with clear inputs and outputs (ex: computeds, validations)

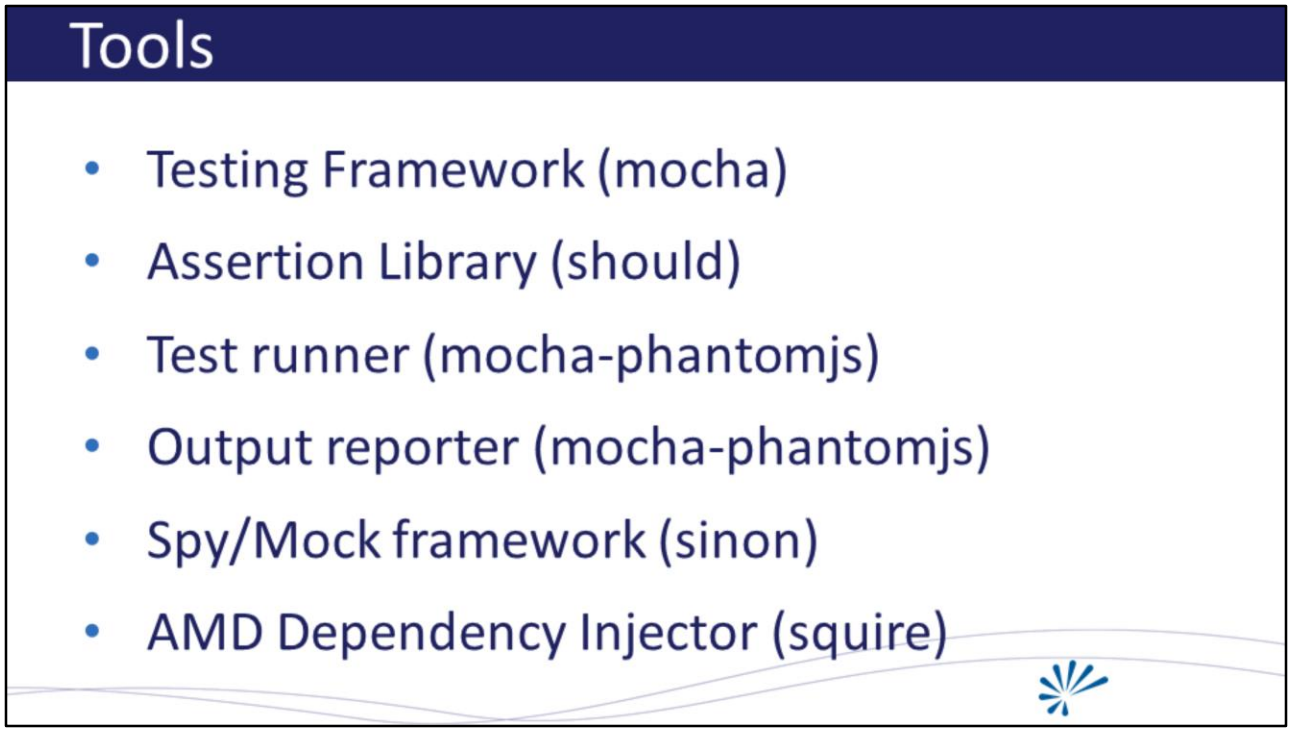
A warning sign on 'difficult to test' in an AMD is a module without a constructor, or one that does a ton of work in the constructor.
We're definitely going to go deep into handling jQuery/dom manipulation in tests in a later demo, but it's an extra layer to worry about. Similarly we're going to demo testing both the success and failure conditions of ajax requests, but again, an extra step.

A good sign is clear, cohesive functions. A knockout computed that does significant work is generally a great example. There may not be a ton of value in a Full Name computed, but one that say, converts measurements. For blinds for the ui we use two dropdowns, one for the whole inch, and one for fractions in eighths, but we obviously send a decimal back to the server. That might be worth testing, just in case someone fat fingered the comparisons. It was me. Thankfully I noticed it, that would have been an incredibly embarrassing bug.

One classic example of something to unit test is complex custom client side validation. While these might be relatively volatile, they are (hopefully) satisfying the open/closed principle. It's actually my go to recommendation, depending on your implementation.

Also, much like any kind of testing, a good candidate is a scenario that you've written for but is really difficult to actually do on the page. Ajax requests that check the error code are a great example. Distinguishing between a 500 error you want to hide vs an unauthorized you want to alert the user about.

Now that we have a good idea of what we're going to test, we can start getting to those exciting tooling decisions. This is the fun stuff, you get to go to the great coffee shop of the internet (NPM) and start ordering drinks. [That joke might sound super weird to some of you but javascript testing frameworks tend to be named after hot beverages]



This might look intimidating but many of the tools out there hit one or more of these. For example, Tap for NodeJS hits almost all of them, plus code coverage. That's a bit of a spoiler for suggestions. The names in parentheses are the ones that we'll be using in the demos. Anybody with experience or a strong opinion on these tools feel free to jump in here. There's more options for each of these than anybody could ever test out themselves. This is very much a personal preference, we can talk a little bit about the tradeoffs and things to look for, but your choice of framework is really unlikely to have much impact on the overall success or failure of your testing.

Holy Wars! Who thinks its ok, in the context of a test framework, to prototype Object? Well should.js does that, as do many other assertion libraries. If that'ss something you care about, go for it
Testing Framework – This is the framework for the test structure/organization. They're all pretty darn simple. Mocha/jasmine/tap are some big examples.

Assertion Library – this split is kind of handy. For example Chai works with anything and actually supports should, expect, and assert language. That can be a mixed bag though, how many people are on a team where everyone is in complete agreement on how tests should be named? How much would it annoy you to see multiple assertion syntaxes in the same test file?

Test runner – Exactly what it sounds like. Karma, Wallaby, and Chutzpah are all

prominent examples, some more fun to say than others. I use mocha-phantomjs in our demos. I've heard amazing things about Wallaby but it costs a surprisingly large amount of money and I'm a very cheap man, as you'll soon see from the fact that I'm using virtuabox for my VM. It's main claim to fame is that because it handles code coverage, it watches and re-runs only the applicable tests whenever you change your javascript. Also hooks into every IDE you can imagine.

Output Reporter – this you want to mainly look at what integrates with your build process. If you're using teamcity, for example, you definitely want to verify that whatever tools you use can output in a TC friendly way.

Spy/Mock – you can totally get away without this to be honest. Javascript is such a flexible language that, especially if your scripting-fu is strong, you can do a lot without a specific library. But it's easier and why reinvent the wheel?

AMD Injector – this is a huge one, and a huge problem with testing AMD modules. Once teams move to an AMD pattern they tend to go a little crazy and everything you want to test suddenly has a ton of dependencies on other modules. With a DI program, you can trick require calls into returning a simple shim.

This is a lot of info, but I wanted to at least go over tooling before we got into the live demo. One note: Code coverage - It's really difficult in Javascript. You can blame the language for being so flexible, or you could blame javascript developers for not caring enough.

[Run tests through psake -> testrunner.html -> require config -> cart view model tests
Javascript tests are fast

Build Process

- Don’t break the build
  - You will break the build
    - Bring donuts
- Dependencies

This is the part where I talk about some war stories. I didn’t specify the exact version of should.js in our packages.json, and then they released a new version that entirely changed the syntax, so every single one of our tests failed.

Phantomjs’s CDN is super flaky so our builds failed every once in a while

Protip: if you go to dunkin donuts, you can get 50 assorted donut holes for a reasonable price.

Gotchas

- Asynchronous tests will lie to you
- Unhandled exceptions will cause weird results
- Closures
- If you are seeing unexpected results in the command line, open your testrunner in a browser

Some of these I even hit when I was writing these demos, despite learning them the first time from painful experience. We're going to demo some of these explicitly in the next demo, but they're important enough I wanted to highlight them.

Tests succeed if and only if there are no exceptions thrown or assertions failed between the start and end of their test. With asynchronous testing, if you call done outside of an async block, it will not only report as passing, but it's entirely possible that if an exception is thrown in that block, it might get reported in a subsequent test. That's a nightmare to debug.

Also javascript doesn't recover from unhandled exceptions super well in general. I'd take any results after that with a grain of salt.

Code coverage - It's really difficult in Javascript. You can blame the language for being so flexible, or you could blame javascript developers for not caring enough.

[Run tests through psake -> testrunner.html -> require config -> cart view model tests
Javascript tests are fast

Further topics in testing

- End to End Testing
  - Selenium/Protractor
- Distributed testing – TestSwarm
- Browser specific quirks
- Further reading
  - This really good stack overflow post
  - The Art of Unit Testing – Roy Osherove

End to End testing - I am kind of a believer that the only way you REALLY know if your code works is to actually hit it end to end. I also hate selenium tests. This is directly contradictory, but that is why developers are like onions. I find them very flakey. Maybe I'm just bad at writing them!

This stack overflow post covers most of what I did not here, namely it goes tool by tool and gives pro and con lists. That stuff is super important but very boring to hear!