
Unit Testing Best Practices in AngularJS

What should I be testing and when should I be writing tests? A frontend development article
by Andy Shora

← back to andyshora.com | [Tweet](#) 539

Why do you need to write tests?

Because you're not Chuck Norris. Chuck Norris' code tests itself, and it always passes, with 0ms execution time.

Ok, seriously, writing tests feels good! I think it's essential that you learn to love the feeling of having a water-tight app. One which you know is not going to, you know, suddenly die the moment someone else starts making changes.

It might just be me, but when you've written a nice function, and you've verified it works the way it should, it feels solid. You see green ticks on a command line, all tests passed. BOOM! You've just written some of the most solid, reliable code that's ever been written!

Level 1. Unit Testing

Welcome to the first level of testing. Everyone has to reach level 1! Breaking down the logic of your application into small chunks or 'units', and verifying that each unit works as desired, makes for a very happy and resilient app lifecycle!

Imagine the developer you were five years ago, imagine he or she inheriting your codebase, and imagine how it would be massacred into a small pile of excrement the moment urgent changes are made. Ok, maybe you weren't that bad. Still, you need to **write at least basic unit tests to make sure regressions don't spring up because some new developer doesn't understand what's going on.**

The core units which make up features should be verified with accompanying unit tests. In JavaScript apps, the smallest units of code you can test are usually individual functions.

The core units which make up features should be verified with accompanying unit tests. In JavaScript apps, the smallest units of code you can test are usually individual functions.

Recommended Unit Testing Tools

- [Jasmine](#) - *"Jasmine is a behavior-driven development framework for testing JavaScript code." (personal favourite, awesome docs)*
- [Mocha](#) - *"The fun, simple, flexible JavaScript test framework."*
- [QUnit](#) - *"A JavaScript Unit Testing framework, used by jQuery."*

Examples

- A function which displays an error message.
- A function which validates an email address.
- A function which sorts a list of names.
- A function which changes the state of a product to 'out of stock'.
- A function which adds a list of numbers together.

In the grand scheme of things, these units may be very abstracted away from the users of the app, they may even be unaware of this level of functionality happening under the hood. Still, it's our job to verify that each of the chunks of an app work individually, so when they are joined together they have a good chance of working as a whole.

...it's our job to verify that each of the chunks of an app work individually, so when they are joined together they have a good chance of working as a whole.

When is a good time to write Unit Tests?

It's easy for a semi-technical manager to say *"write all tests at the start"*, but the truth is, you often don't know all the small components which are required when presented with a high-level user story.

I usually write unit tests as I go along, as I develop all the 'units' required to implement a

user story. You're going to constantly be changing functions around, refactoring and abstracting parts away, tidying up what's going in and coming out of functions, trying to preempt any of this is going to cause a lot of wasted time.

Scenario

- User story: "As a user, when I login to my app, I am taken to the dashboard page."

How do we break it down into tests? Break it down like this: Story → Features → Units

I'll immediately select the 'login form' as a feature of this story which I can test, and I'll add the following test outlines for the important units that are required to implement this feature correctly:

```
describe('user login form', function() {  
  
  // critical  
  it('ensure invalid email addresses are caught', function() {});  
  it('ensure valid email addresses pass validation', function() {});  
  it('ensure submitting form changes path', function() { });  
  
  // nice-to-haves  
  it('ensure client-side helper shown for empty fields', function() { });  
  it('ensure hitting enter on password field submits form', function() { });  
  
});
```

Depending on your time constraints, you may decide to skip some of those tests, but hopefully you can see how critical the first three are, as if they failed, they could completely block people from using the app.

Consider building a car from start to finish, each of the parts which make the engine, the chassis, the wheels, must be individually verified to be in working order before they are to be assembled into a 'car'.

Consider building a car from start to finish, each of the parts which make the engine, the chassis, the wheels, must be individually verified to be in working order before they are to

be assembled into a 'car'.

If you've been told time is off the essence, less important parts can be de-prioritised. For example, tyres might not be checked for pressure, the stitching in the interior may not be checked for damage. This could result in a slightly shoddy car, but hey that's not your problem, you don't manage your time! (Always best to confirm it's ok to skip these things first though, could be dangerous!)

Writing Good Unit Tests for AngularJS

Let's go through some actual examples.

```
describe("Unit Testing Examples", function() {

    beforeEach(angular.mock.module('App'));

    it('should have a LoginCtrl controller', function() {
        expect(App.LoginCtrl).toBeDefined();
    });

    it('should have a working LoginService service', inject(['LoginService',
        function(LoginService) {
            expect(LoginService.isValidEmail).not.to.equal(null);

            // test cases - testing for success
            var validEmails = [
                'test@test.com',
                'test@test.co.uk',
                'test7341tylytkliytcryety9ef@jb-fe.com'
            ];

            // test cases - testing for failure
            var invalidEmails = [
                'test@testcom',
                'test@ test.co.uk',
                'ghgf@fe.com.co.',
                'tes@t@test.com',
                ''
            ];

            // you can loop through arrays of test cases like this
            for (var i in validEmails) {
                var valid = LoginService.isValidEmail(validEmails[i]);
                expect(valid).toBeTruthy();
            }
        }
    ]
});
```

```
    }  
    for (var i in invalidEmails) {  
        var valid = LoginService.isValidEmail(invalidEmails[i]);  
        expect(valid).toBeFalsy();  
    }  
    })  
    );  
});
```

Level 2. Integration Testing

Integration testing is where we write end-to-end tests, performing a sequence of a workflow or user journey, and verifying the state of the app/UI along the way.

This task is still mostly done manually by QA, and good QA engineers do automate tests using tools like [Selenium](#) or [PhantomJS](#) to make their life easier, but we're seeing a shift in responsibility now that developers have easy access to the tools required to automate end-to-end integration tests right from a task runner such as [Grunt](#).

Use Grunt Y'all

Quite simply, if you're not yet using [Grunt](#) for your test and build process, you either haven't heard of it yet, haven't heard exactly how awesome it is, or you're potentially an old, stubborn developer who doesn't like change.

Grunt allows us to install testing frameworks like Jasmine or Karma in a single command, and then execute all tests within a project when files change, or during the build process. Simply executing a command like **grunt test** can lint your code and perform all unit and integration tests. Also take a look at [Yeoman](#) as a complete solution for setting up an app skeleton as well as full build and testing procedures, which uses Grunt by default.

Anyway, back to testing... we're essentially going to program the interactions needed to perform a task, and our headless browser is going to do all the work.

Recommended Integration Testing Tools

- [Karma](#) - *"Spectacular Test Runner for JavaScript." (my favourite + uses real browsers!)*
- [CasperJS](#) - *"CasperJS is a navigation scripting & testing utility which uses headless browsers."*

Examples

- User logs into a website via the homepage, and sees a dashboard.
- User searches for a product, clicks on a search result, and is taken to a product page.
- User opens the basket, removes an item, and sees the number of items and the subtotal decrease.
- User clicks '2', then '+', then '3', then '=', and sees '5' in the result area.

Going back to our car example, integration testing is a bit like automating a robot to perform some function like drive the car. We consider the car as a whole, we tell the robot to get in, turn on the engine, and see if the car moved forward after a set sequence of operations.

When is a good time to write Integration Tests?

When user stories are being created, it's a good idea to flesh out integration tests which will correspond to user journeys. A good thing about higher level testing like this, is that the tests will be pretty resilient to changes in your code. As long as the elements rendered on a page, and the routing stays the same, if you change the underlying implementation one day, your E2E tests should still pass.

When user stories are being created, it's a good idea to flesh out integration tests which will correspond to user journeys.

Scenario

- User story: "As a user, when I login to my app, I am taken to the dashboard page."

At this point, I've literally just heard about the user story, I might decide my integration tests will look something like this:

```
describe('user login', function() {

  it('ensures user can log in', function() {
    // expect current scope to contain username
  });
  it('ensures path has changed', function() {
    // expect path to equal '/dashboard'
  });
});
```

```
});
```

What about verifying visual aspects?

If you wanted to programatically verify the design is pixel perfect when rendered, you're actually asking for the world! Although integration testing frameworks such as PhantomJS has become advanced enough to take screenshots at different points in a test, you're always going to need a manual stage to tests where someone can verify the rendered design against a visual design.

Developers have even started creating image diff tools to detect regressions in design, which is fantastic, but also comes with the burden of having to mock all content which is rendered, to prevent image diff libraries mistakenly detecting different content as visual regressions.

I recently saw a talk at LondonJS by [James Cryer](#) who wrote a really cool CSS Regression Testing tool, [PhantomCSS](#), which can produce screenshots of various parts of an app, overlayed with a bright colour indicating visual regressions. Amazing stuff, but perhaps a bit out of scope for this article.

Writing Good Integration/E2E Tests for AngularJS

Here's some examples of higher-level Integration Tests, which interact with the application level, as opposed to the actual functions in your app.

```
describe("Integration/E2E Testing", function() {

  // start at root before every test is run
  beforeEach(function() {
    browser().navigateTo('/');
  });

  // test default route
  it('should jump to the /home path when / is accessed', function() {
    browser().navigateTo('#/');
    expect(browser().location().path()).toBe("/login");
  });

  it('ensures user can log in', function() {
    browser().navigateTo('#/login');
```

```

expect(browser().location().path()).toBe("/login");

// assuming inputs have ng-model specified, and this combination will successf
input('email').enter('test@test.com');
input('password').enter('password');
element('submit').click();

// Logged in route
expect(browser().location().path()).toBe("/dashboard");

// my dashboard page has a label for the email address of the logged in user
expect(element('#email').html()).toContain('test@test.com');
});

it('should keep invalid logins on this page', function() {
  browser().navigateTo('#/login');
  expect(browser().location().path()).toBe("/login");

  // assuming inputs have ng-model specified, and this combination will successf
  input('email').enter('invalid@test.com');
  input('password').enter('wrong password');
  element('submit').click();

  expect(element('#message').html().toLowerCase()).toContain('failed');

  // Logged out route
  expect(browser().location().path()).toBe("/login");

});

});

```

Mocking Services and Modules in AngularJS

Instead of our tests talking to real services, it will often suffice to speak to the 'mock' versions of the services, which will respond to calls accurately, but rather unintelligently. We use mocks because if we started relying on a real service writing real data, or a service changing the location in the address bar, or an unreliable external API, then it may interfere with the execution of our test runner in its own potentially sandboxed environment.

In fact, it's a lot better to mock as we concentrate test code coverage purely on our module. Stubbing out endpoints which our module is interacting with is a really good way of ensuring the reliability of our tests is dependent purely on how well the code inside our

module performs.

The aim of mocking objects is to remove dependencies on any external factors that module X may be interfacing with, such as a browser's API, or an external API endpoint. We can assume this stuff has already been tested.

The important thing to remember is that we aren't testing this external service 'Y', it's an abstracted-away module which we assume has its own tests. If this service Y ships with a mock version, then we can assume that all essential logic is still included, and we are simply interfacing a dumb version which will act as a static endpoint to satisfy our tests.

In our car example, objects we may decide to mock out would be pedestrians, maybe other cars, even passengers are 'mocked out' in the form of crash test dummies!

Stubbing out endpoints which our module is interacting with is a really good way of ensuring the reliability of our tests is dependent purely on how well the code inside our module performs.

Consider the \$location service which should do the following things when it's path() function is called with a path string as a param:

- Change the path in the address bar.
- Fire a '\$routeChangeSuccess' event.

In this situation it would be ok for the mock \$location service to store the new path in the instance, and fire the '\$routeChangeSuccess' event when it's updated, but not actually redirect the browser. We assume this will already work, and has been tested by the person who wrote this service.

Mocking HTTP Requests with \$httpBackend

Here's an example showing how to mock a http request, using the \$httpBackend service to intercept the request if it meets certain conditions. Note that the LoginService.login method uses the \$http service to send a POST request, and this service relies on \$httpBackend in Angular.

```
it('should get login success',
  inject(function(LoginService, $httpBackend) {

    $httpBackend.expect('POST', 'https://api.mydomain.com/login')
      .respond(200, "[{ success : 'true', id : 123 }]");

    LoginService.login('test@test.com', 'password')
      .then(function(data) {
        expect(data.success).toBeTruthy();
      });

    $httpBackend.flush();
  });
```

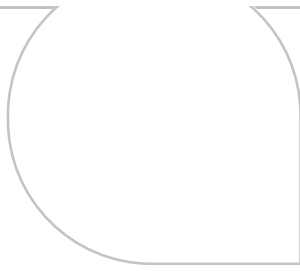
Summary

To round things up, remember to keep these rules in mind when writing Unit Tests and Integration Tests for your application:

- Try and flesh out outlines for the Integration Tests at the point where your user stories are becoming formalised.
- When writing E2E/Integration Tests, always bear in mind they should be application-level, so they should be pretty resistant to changes in the underlying implementation.
- Break components down into units/chunks (which will often be single functions), and identify the units which are critical to the reliability of your component. Write unit tests for these chunks.
- Remember to test for failure, as well as success. For example, check that an invalid email address doesn't pass an email validation function. Set up some test cases with really bizzare input as well as just a slightly invalid email address.
- Consider using [jasmine](#) for unit tests, and [Karma](#) for performing end-to-end integration tests.
- Mock external modules and services which simply act as endpoints for the code in your module. Don't test things like third party services which should have their own tests.


That's all for now!

Thanks for reading, as always please leave your questions and comments below!




Hi, I'm **Andy Shora**. I'm a **Frontend Web Developer** based in London and I currently work with some very talented people over at [R/GA](#). I founded [Stackey.com](#) - a place to stack things. All work here is my own.

 [andyshora](#)

 [fb.com/andyshora](#)

 [pinterest.com/andyshora](#)

 [github.com/andyshora](#)

 [contact me](#)

Sort by Best ▾

Share  Favorite ★**Daniel Eck** · a year ago

In 1.2 (currently in 3rd RC) and future releases ngScenario is being replaced by protractor for E2E testing.

12 ^ | ▾ · Reply · Share ›

**dmackerman** · a year ago

Great post. As a developer who's still very new to unit test and test driven development, this is a great start. Thanks!

12 ^ | ▾ · Reply · Share ›

**Andy Shora** Mod → **dmackerman** · a year ago

Thanks, Dave! Glad you found it useful.

^ | ▾ · Reply · Share ›

**Melanie Archer** · a year ago

In my experience, the colleagues most opposed to using testing workflows were the twenty-something "cowboy coders" who thought their every line of code was burnished perfection, and those most enthusiastic for tools like Grunt tasks were the forty- and fifty-something QA testers tired of the rubbish sent them from upstream--so your remark about "old, stubborn developer[s]" is not just ageist, it's inaccurate.

Great advice in the rest of your post, all the same.

7 ^ | ▾ · Reply · Share ›

**Rahul Guha** · a year ago

Great post ... unit tests are still in its childhood in JS world. Will help many people including me

6 ^ | ▾ · Reply · Share ›

**Andy Shora** Mod → **Rahul Guha** · a year ago

Thanks man! Much appreciated.

3 ^ | ▾ · Reply · Share ›

**Steve Clements** · a year ago

undefined !== null so your first unit test will pass incorrectly. It might be better to use `toBeDefined()`.

3 ^ | ▾ · Reply · Share ›

