Full-Spectrum Testing with AngularJS and Karma

Learn how to fully test your AngularJS application with Karma

One of the major areas of AngularJS development which needs to be covered in better detail is *how to test your application*. But wait, how do you do that? How do you setup your test environment? How should you organize your code? How do you setup a runner and how to automate the process? The bigger question is how do you usually test your JavaScript code? Do you use a *browser-independent JavaScript tester* like *NodeJS*, *V8* or *Rhino*? Do you use a hidden browser like *PhantomJS* or *Headless Webkit*? Or do you go all out and test out your application in a *browser* or *multiple browsers*? You may also be wondering *what exactly to test* in your application? What should my tests really look for? How should I write my specs? As you can see there are *many questions to answer*.

It doesn't take much time to realize that *testing JavaScript is difficult and finicky*. However, if you wish to get a *robust understanding* of how to *properly test* your JavaScript code within *your AngularJS application*, then continue and read along... :)

Last Updated

This page was first published on *January 21st 2013* and was last updated on *November 20th 2013*.

Table of Contents

Karma

Karma
Spectacular Test Runner for JavaScript

The major challenge when it comes to testing JavaScript is that it requires you to have a runner setup to test your code. The issue with JavaScript runners is that you must properly set them up to include all of the JavaScript code (source and tests) in your application and then somehow wire that into an automated test suite. After that, you will need to open up each browser and test all the specs to see if they test properly with no errors or run a command line program to test your suite on call. If you run your suite from the command line then you'll need to write an automated browser tester into your test suite and this may require some tricky hacking. As you can see, this is too much of a pain and the time and effort to set this up should be saved for more important things like building your application.

Karma is an amazing testing tool which is designed to take *all the frustration out of setting up a working test runner* when testing JavaScript code. Karma works by *spawning up each browser that is specified within its configuration file* and then running JavaScript code against those browsers to see if they pass certain tests. Communication between Karma and each of the browsers is handled with the karma service running in the terminal using socket.io. Each time a test is run, Karma records its status and then tallies up which browsers have failed for each test and which ones passed and timed out. This makes each test work *100% natively in each browser* without the need to test individually. Also, since the Karma service runs on a port and keeps track of browsers by itself, you can easily hook up other browsers and devices to it just by visiting its broadcasting port. Oh and did I mention that Karma is fast? Yeah it's really fast... :)

So before we get started with Karma, lets figure out how testing works with AngularJS...

Two types of tests in AngularJS (plus one more)

Testing in AngularJS isn't as easy as it looks (but then again testing anything on the front-end is pretty difficult). AngularJS is a great framework for testing your application in all areas, but it requires a bit of configuration to get it to work. AngularJS provides *Unit testing* and *E2E (End-to-End) testing* both of which are very different from each other. Unit testing works by *isolating small "units" of code* so that it can be tested from every angle while E2E testing works by *testing full areas of the application* by running a test through its *entire stack* of operations *against a special HTTP server* from the start to the end (hence end to end). Think about E2E tests as a robot using a browser to run a test and then running an assertion against the page once it's ready. Basically, E2E is is just about the same idea as you refreshing your page and saying "It should say home somewhere" or "it should repeat this list five times".

The nice thing about AngularJS is that E2E tests are fully covered with use of the AngularJS Scenario Runner (which of course is *bundled into Karma*).

So that covers two of the major testing approaches, but what about the third? Well the third approach involves using a special AngularJS testing plugin which acts like a whitebox tester. This tester is used to test the raw code of your application and you have access to every level of the JavaScript code. This testing approach *falls in between Unit testing and E2E testing*. When Unit testing your application it sometimes *may get too complicated when you want to test an application-level operation* (like a page loading or XHR request) because you will need to use interceptors and mocks to make requests and templating work (basically anything *XHR'ish*). E2E testing may not also be the best option because it may be too *high level to capture to test for certain features*. For example, how do you test a controller on your website that performs a XHR request which downloads the user's session details on every page? ... You can't test this with E2E (since it's difficult to access scope members and application-level JavaScript code) and Unit testing requires lots of mocking and fake data which results in more code and more complexity for your tests. What we really need is a third testing approach which fits in the middle. Therefore, the third testing approach for testing AngularJS applications is known as *Midway Testing*.

Click here to view the *ngMidwayTester plugin*

The midway tester comes bundled along with the demo repository (so there is no need to download it) and all the code below called "*Midway Testing*" refers to the test spec code which the midway runner executes.

Here's a quick overview of the three different test approaches when testing AngularJS applications:

Unit Testing	Midway Testing	E2E Testing
1. Code-level testing	1. Application/Code-level testing	1. Web-level testing
2. Best for testing services, classes and objects	2. Can access all parts of an application	2. Requires its own special web server
3. Sandboxed & Isolated testing	3. Can interact directly with the web application code	3. The expect and should matchers are specific to AngularJS (MochaJS or Chai won't work)
4. Mocking & Stubbing required	4. Not really effective for stubbing & mocks	4. Perfect for integration tests
5. Fast	5. Easily breaks since it relies on application code to operate (but this may be good to catch code-level errors)	5. Works really well with assertions against future data
	6. Not possible to test anything inside of your index.html file	6. Unable to access Application JavaScript code (only rendered HTML and some AngularJS info)
	7. Fast, but slow for XHR requests	7. Slow
Click here to view the Unit Karma Configuration	Click here to view the Midway Karma Configuration	Click here to view the E2E Karma Configuration

So now that we know what each of the testing approaches are, lets figure out how to fully test out an AngularJS application. Inside each testing topic below, the *Unit*, *Midway* and/or *E2E* testing code specs are provided to demonstrate how to test key areas of your AngularJS application.

Installation & Configuration of the Demo Repository

Now that you understand the three different testing approaches, lets setup the test environment. It's *absolutely essential that you download this repository and follow along with the testing scripts* inside which fall inline with the tests featured alongside this article. Before you install it you must have *Node JS (with NPM)* and *Git* installed and accessible via the command line.

The following code below will download the demo repository which is a Youtube video library built with AngularJS. No special software is required since this is a static website.

Click here to view the *online demo website*

```
# install via Git
```

```
git clone git://github.com/yearofmoo-articles/AngularJS-Testing-Article.git

# then hop inside that directory
cd AngularJS-Testing-Article

# and get the dependencies
npm install
```

Now the demo application is created, *run this script to launch it as a web server* on port 8888.

```
# All you need to do is run this command to launch the website
grunt
```

You can now access the website via *http://localhost:8888 (or whatever port you used)*.

Now you're almost ready to to start testing. There is just one more command which is discussed next which you need to run to get karma ready to test your application.

Organizing your Code

The demo repo provides a large assortment of directories and files which are organized so that you can test out every aspect of your AngularJS application. Feel free to copy and paste the files from this repo or full out follow the structure of this repository layout for your new projects. The demo application contains the directories for the *Application Code* as well the directory layout and karma configuration for *Unit*, *Midway*, and *E2E* testing approaches.

Click here for a full *breakdown of the demo repository*

As you can see there are *three karma configuration files* which are each designed to handle the testing for the three types of testing approaches mentioned above. The karma.shared.conf.js is just a shared configuration file that all three other karma files read from. The *grunt test* command (which is mentioned next) runs the required to code to launch all three karma sessions to test the entire application via the testing scripts found within the test/ directory.

You can now execute each of the tests by running the commands below and leaving them open.

```
# This will test everything
grunt test
```

Each time you save any file inside of the *test/e2e*, *test/unit* or *test/midway* directories then run grunt test to test again. You can also run *grunt autotest:SUITE* (the SUITE value can unit, e2e or midway) to watch and test each time you update your files.

```
HH:MM:SS unit.1    |  INFO [watcher]: Changed file "...".
                      Chrome 24.0 (Mac): Executed 16 of 16 SUCCESS (0.366 secs / 0.035 secs)
HH:MM:SS midway.1 |  INFO [watcher]: Changed file "...".
                      Chrome 24.0 (Mac): Executed 24 of 24 SUCCESS (5.834 secs / 3.656 secs)
HH:MM:SS e2e.1     |  INFO [watcher]: Changed file "...".
                      Chrome 24.0 (Mac): Executed 15 of 15 SUCCESS (7.314 secs / 6.856 secs)
```

So now that everything is setup, lets start testing some AngularJS components!!

MochaJS, Chai.js and AngularJS Scenario Runner

For Unit and Midway tests, this demo repository uses MochaJS for running the test specs and Chai.js for the matchers and assertions.

For *E2E tests* AngularJS provides the AngularJS Scenario runner which is loosely designed around the Jasmine JavaScript Testing framework. Unfortunately, there is no way to use MochaJS or ChaiJS with E2E testing (trust me I tried...) so you need to stick around using the scenario runner. Here is some extra documentation about the runner.

Click here to view the *AngularJS Scenario Runner Documentation*

Isn't the AngularJS Scenario Runner just amazing?!!!

Interceptors & Mocks in AngularJS

Interceptors are built-in features in AngularJS where you can hop into a request or chain of callback methods to morph or break the flow of logic between end points. However, when *Unit* testing AngularJS applications, *interceptors are used to catch XHR requests to avoid external requests to services* which may occur inside routes (when resolving), controllers, services and even directives (please don't do this with directives :P). This is a really solid approach when testing AngularJS applications since any external request will be caught (so long as you use the $http service when doing them), but it will add more coupling and extra code into your tests. So if you want to avoid doing interceptors and (possibly) mocking then use *Midway tests* to test your routes, controllers, services and directives when XHR requests are going on in the background.

Another way to work around and/or avoid catching expensive XHR operations is by use of Mocks. Mocks are used to *provide and "stub" fake input and response information to functions and operations* inside test specs. So if you have a really expensive operation (like an external API call) that may take a long time to pull data then it's best to use mocks to "trick" your application code into doing its job without any expensive operations going on behind the scenes. Mocks are also a great way to decouple and remove redundant test code for parts of your application.

AngularJS provides its own mocking code *inside of the angular-mocks.js* file which does the trick. Mocks are designed to be used with *Unit tests* (not Midway or E2E tests). Each of the karma configuration files found under *test/* have all the files required to run each test. One thing to keep in mind is that, ~~if you plan on doing Unit testing using MochaJS in the future, be sure to check to see that you have the *updated version of the angular-mocks.js file* which contains the code...~~. Bower now properly manages the angular-mocks code so the file used within testing is the most recent and up to date version.

Testing Modules

What's our testing goal?
To test to see that the module is there and it works. To check to see if the dependencies are set for the module.

Modules are the topmost container objects for creating directives, controllers, templates, services and resources. Therefore, when testing modules, you are really just checking to see if the module exists. This can be achieved with both *Unit and Midway testing*. However, it's better to test it within midway testing because the code has already been executed and now you're just checking to see if the module is accessible via angular.

Midway Testing

```
//
// test/midway/appSpec.js
//
describe("Midway: Testing Modules", function() {
  describe("App Module:", function() {

    var module;
    before(function() {
      module = angular.module("App");
    });

    it("should be registered", function() {
      expect(module).not.to.equal(null);
    });

    describe("Dependencies:", function() {

      var deps;
      var hasModule = function(m) {
        return deps.indexOf(m) >= 0;
      };
      before(function() {
        deps = module.value('appName').requires;
      });

      //you can also test the module's dependencies
      it("should have App.Controllers as a dependency", function() {
        expect(hasModule('App.Controllers')).to.equal(true);
      });

      it("should have App.Directives as a dependency", function() {
        expect(hasModule('App.Directives')).to.equal(true);
      });

      it("should have App.Filters as a dependency", function() {
        expect(hasModule('App.Filters')).to.equal(true);
      });

      it("should have App.Routes as a dependency", function() {
        expect(hasModule('App.Routes')).to.equal(true);
      });

      it("should have App.Services as a dependency", function() {
        expect(hasModule('App.Services')).to.equal(true);
      });
    });
  });
});
```

Testing Routes

What's our testing goal?
To check to see that route works, doesn't work or redirects. To see that the correct controller handles the route.

When testing routes we want to ensure that the routes we're focussed on are routed properly in our application. Therefore we will need to check where the route is routed to and if the route itself is redirected or not found. A 404 page should be displayed and a $routeChangeError event should be fired if a route is not found. We should also check to see if the template is loaded for the route within the view. The best types of tests for this are *Midway tests* and *E2E tests*.

So lets get started with testing routes in AngularJS using *Midway* and *E2E* testing.

Midway Testing E2E Testing

```
//
// test/midway/routesSpec.js
//
describe("Midway: Testing Routes", function() {

  var tester;
  beforeEach(function() {
    tester = ngMidwayTester('App');
  });

  afterEach(function() {
    tester.destroy();
    tester = null;
  });

  it("should have a working videos_path route", function() {
    expect(ROUTER.routeDefined('videos_path')).to.equal(true);
    var url = ROUTER.routePath('videos_path');
    expect(url).to.equal('/videos');
  });

  it("should have a videos_path route that should goto the VideosCtrl controller", function() {
    var route = ROUTER.getRoute('videos_path');
    route.params.controller.should.equal('VideosCtrl');
  });

  it("should have a working video_path route", function() {
    expect(ROUTER.routeDefined('video_path')).to.equal(true);
    var url = ROUTER.routePath('video_path', { id : 10 });
    expect(url).to.equal('/videos/10');
  });

  it("should have a videos_path route that should goto the VideoCtrl controller", function() {
    var route = ROUTER.getRoute('video_path');
    route.params.controller.should.equal('VideoCtrl');
  });

  it("should have a working watched_videos_path route", function() {
    expect(ROUTER.routeDefined('watched_videos_path')).to.equal(true);
    var url = ROUTER.routePath('watched_videos_path');
    expect(url).to.equal('/watched-videos');
  });

  it("should have a videos_path route that should goto the WatchedVideosCtrl controller", function() {
    var route = ROUTER.getRoute('watched_videos_path');
    route.params.controller.should.equal('WatchedVideosCtrl');
  });

  it("should have a home_path route that should be the same as the videos_path route", function() {
    expect(ROUTER.routeDefined('home_path')).to.equal(true);
    var url1 = ROUTER.routePath('home_path');
    var url2 = ROUTER.routePath('videos_path');
    expect(url1).to.equal(url2);
  });

  it("should have a home_path route that should goto the VideosCtrl controller", function() {
    var route = ROUTER.getRoute('home_path');
    route.params.controller.should.equal('VideosCtrl');
  });

});
```

```
//
// test/e2e/routesSpec.js
//
describe("E2E: Testing Routes", function() {

  beforeEach(function() {
    browser().navigateTo('/');
  });

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

  it('should have a working /videos route', function() {
    browser().navigateTo('#/videos');
    expect(browser().location().path()).toBe("/videos");
  });

  it('should have a working /wathced-videos route', function() {
    browser().navigateTo('#/watched-videos');
    expect(browser().location().path()).toBe("/watched-videos");
  });

  it('should have a working /videos/ID route', function() {
    browser().navigateTo('#/videos/10');
    expect(browser().location().path()).toBe("/videos/10");
  });

});
```

Testing Requests / Pages

What's our testing goal?
To check to see that the entire page request works from start to end. To make sure that each page loads the data in the view properly.

Testing pages and requests involves the same testing approach as testing routes and controllers. However for this particular testing approach *you simply want to check to see that the route is accessed that the route works*, the controller is executed, the template is downloaded and the view is rendered and marked as ready. Any number of requests and downloads may occur in the background, therefore as long as your controller *somehow marks your page as ready* (something on the body tag, a html snippet to inside the page, etc...) then you know that the page is functional. This is really great for when you have this across your entire website for every route and every page.

This special type of testing approach can be accomplished with both *Midway Testing* and *E2E Testing*.

Midway Testing E2E Testing

```
//
// test/midway/requestsSpec.js
//
describe("Midway: Testing Requests", function() {

  var tester;
  beforeEach(function() {
    if(tester) {
      tester.destroy();
    }
    tester = ngMidwayTester('App');
  });

  it("should goto the videos_path by default", function(done) {
    tester.visit('/', function() {
      expect(tester.viewElement().html()).to.contain('app-youtube-listings');
      done();
    });
  });

  it("should have a working video_path request", function(done) {
    var url = ROUTER.routePath('video_path', { id : 10 });
    tester.visit(url, function() {
      var $params = tester.inject('$routeParams');
      expect(parseInt($params.id)).to.equal(10);

      expect(tester.viewElement().html()).to.contain('app-youtube-profile');
      done();
    });
  });

  it("should have a working other_path request", function(done) {
    var url = ROUTER.routePath('other_path');
    tester.visit(url, function() {
      expect(tester.viewElement().html()).to.contain('other page');
      done();
    });
  });

});
```

```
//
// test/e2e/requestsSpec.js
//
describe("E2E: Testing Requests", function() {

  beforeEach(function() {
    browser().navigateTo('/');
  });

  it('should have a working /videos page', function() {
    browser().navigateTo('#/');
    expect(browser().location().path()).toBe("/videos");
    expect(element('#ng-view').html()).toContain('data-app-youtube-listings');
  });

  it('should have a working /other page', function() {
    browser().navigateTo('#/other');
    expect(browser().location().path()).toBe("/other");

    //try removing the controller and this will fail
    expect(element('#ng-view').html()).toContain('success');
  });

});
```

Testing Controllers

What's our testing goal?
To check to see if the controller is executed properly. To check to see if any important $scope members are set. To capture any XHR requests in *Unit Testing* using mocks.

Testing controllers requires an idea of what data will be passed along from the controller to the template via its scope. Therefore, you will need to first test that the controller itself works and the data has been bound into the template. It's best to have an idea of what you expect the controller to do to the scope when it's run. If you were to use an *Unit test* then that test would be dependent on the logic of the controller. An *E2E test* works as well, but you can't guarantee that the controller is doing everything correctly. Therefore, the best approach for this would be to use a *Midway Test*.

Unit Testing Midway Testing E2E Testing

```
//
// test/unit/controllers/controllersSpec.js
//
describe("Unit: Testing Controllers", function() {

  beforeEach(module('App'));

  it('should have a VideosCtrl controller', function() {
    expect(App.VideosCtrl).not.to.equal(null);
  });

  it('should have a VideoCtrl controller', function() {
    expect(App.VideoCtrl).not.to.equal(null);
  });

  it('should have a WatchedVideosCtrl controller', function() {
    expect(App.WatchedVideosCtrl).not.to.equal(null);
  });

  it('should have a properly working VideosCtrl controller', inject(function($rootScope, $controller, $httpBackend) {
    var searchTestAtr = 'cars';
    var response = $httpBackend.expectJSONP(
      'https://gdata.youtube.com/feeds/api/videos?q=' + searchTestAtr + '&v=2&alt=json&callback=JSON_CALLBACK');
    response.respond(null);

    var $scope = $rootScope.$new();
    var ctrl = $controller('VideosCtrl', {
      $scope : $scope,
      $routeParams : {
        q : searchTestAtr
      }
    });
  }));

  it('should have a properly working VideoCtrl controller', inject(function($rootScope, $controller, $httpBackend) {
    var searchID = 'cars';
    var response = $httpBackend.expectJSONP(
      'https://gdata.youtube.com/feeds/api/videos/' + searchID + '?v=2&alt=json&callback=JSON_CALLBACK');
    response.respond(null);

    var $scope = $rootScope.$new();
    var ctrl = $controller('VideoCtrl', {
      $scope : $scope,
      $routeParams : {
        id : searchID
      }
    });
  }));

  it('should have a properly working WatchedVideosCtrl controller', inject(function($rootScope, $controller, $httpBackend) {
    var $scope = $rootScope.$new();

    //we're stubbing the onReady event
    $scope.onReady = function() { };
    var ctrl = $controller('WatchedVideosCtrl', {
      $scope : $scope
    });
  }));

});
```

```
//
// test/midway/controllers/controllersSpec.js
//
describe("Midway: Testing Controllers", function() {

  var tester;
  beforeEach(function() {
    if(tester) {
      tester.destroy();
    }
    tester = ngMidwayTester('App');
  });

  it('should load the VideosCtrl controller properly when /videos route is accessed', function(done) {
    tester.visit('/videos', function() {
      tester.path().should.eq('/videos');
      var current = tester.inject('$route').current;
      var controller = current.controller;
      var scope = current.scope;
      expect(controller).to.eql('VideosCtrl');
      done();
    });
  });

  it('should load the WatchedVideosCtrl controller properly when /watched-videos route is accessed', function(done) {
    tester.visit('/watched-videos', function() {
      tester.path().should.eq('/watched-videos');
      var current = tester.inject('$route').current;
      var controller = current.controller;
      var params = current.params;
      var scope = current.scope;

      expect(controller).to.equal('WatchedVideosCtrl');
      done();
    });
  });

});
```

```
//
// test/e2e/controllers/controllersSpec.js
//
describe("E2E: Testing Controllers", function() {

  beforeEach(function() {
    browser().navigateTo('/');
  });

  it('should have a working videos page controller that applies the videos to the scope', function() {
    browser().navigateTo('#/');
    expect(browser().location().path()).toBe("/videos");
    expect(element('#ng-view').html()).toContain('data-app-youtube-listings');
  });

  it('should have a working video page controller that applies the video to the scope', function() {
    browser().navigateTo('#/videos/WuiHuZq_cg4');
    expect(browser().location().path()).toBe("/videos/WuiHuZq_cg4");
    expect(element('#ng-view').html()).toContain('app-youtube-embed');
  });

});
```

Testing Services / Factories

What's our testing goal?
To isolate the service to find all bugs and errors. To check to see that the service can make XHR request without stubbing.

Services are the easiest blocks of code to test. A *Unit test* or a *Midway Test* is ideal when testing services because you can target the areas of the service that need to be tested from all angles (parameters, return types, exceptions, etc...). Keep in mind that a Unit test works similar to a Midway test for this example, but if your service has any XHR requests using the *$http* service then you will have to capture and intercept those requests and override the response data with mock data to make the Unit test work. Midway testing avoids this entirely.

Unit Testing Midway Testing

```
//
// test/unit/services/servicesSpec.js
//
describe("Unit: Testing Controllers", function() {

  beforeEach(module('App'));

  it('should contain an $appStorage service',
    inject(function($appStorage) {

    expect($appStorage).not.to.equal(null);
  }));

  it('should contain an $appYoutubeSearcher service',
    inject(function($appYoutubeSearcher) {
    expect($appYoutubeSearcher).not.to.equal(null);
  }));

  it('should have a working $appYoutubeSearcher service',
    inject(['$appYoutubeSearcher',function($yt) {

    expect($yt.prefixKey).not.to.equal(null);
    expect($yt.resize).not.to.equal(null);
    expect($yt.prepareImage).not.to.equal(null);
    expect($yt.getWatchedVideos).not.to.equal(null);
  }]));

  it('should have a working service that resizes dimensions',
    inject(['$appYoutubeSearcher',function($yt) {

    var w = 100;
    var h = 100;
    var mw = 50;
    var mh = 50;
    var sizes = $yt.resize(w,h,mw,mh);
    expect(sizes.length).to.equal(2);
    expect(sizes[0]).to.equal(50);
    expect(sizes[1]).to.equal(50);
  }]));

  it('should store and save data properly',
    inject(['$appStorage',function($storage) {

    var key = 'key', value = 'value';
    $storage.enableCaching();
    $storage.put(key, value);
    expect($storage.isPresent(key)).to.equal(true);
    expect($storage.get(key)).to.equal(value);

    $storage.erase(key);
    expect($storage.isPresent(key)).to.equal(false);

    $storage.put(key, value);
    $storage.flush();
    expect($storage.isPresent(key)).to.equal(false);
  }]));

});
```

```
//
// test/midway/services/servicesSpec.js
//
describe("Midway: Testing Services", function() {

  var tester;
  beforeEach(function() {
    if(tester) {
      tester.destroy();
    }
    tester = ngMidwayTester('App');
  });

  it('should perform a JSONP operation to youtube and fetch data',
    function(done) {

    var $yt = tester.inject('$appYoutubeSearcher');
    expect($yt).not.to.equal(null);

    //this is the first video ever uploaded on youtube
    //so I doubt it will be removed anytime soon
    //and should be a good testing item
    var youtubeID = 'jNQXAC9IVRw';

    $yt.findVideo(youtubeID, false,
      function(q, data) {
        expect(data).not.to.equal(null);
        expect(data.id).to.equal(youtubeID);
        done();
      }
    );
  });

});
```

Testing Filters

What's our testing goal?
To isolate all input and output operations with the filter using Unit testing. To check to see that the filter applies its logic to the scope or to the HTML within the DOM.

To test AngularJS filters you need to *include the $injector service*, then get the *$filter service from injector* and then from there call up each of the services you want to include into your page. Filter testing can be used with *all three testing approaches*, but each approach is much different from each other. Midway testing, however, falls in between where you can do both Unit testing and E2E testing methods.

Basically with a filter you want to just make sure that for a given input you get a given output (which in most cases is an array). Your AngularJS code will rely on working filters, so in your E2E tests just make sure to test to see that the filters *apply their logic to the page*.

Unit Testing Midway Testing E2E Testing

```
//
// test/unit/filters/filtersSpec.js
//
describe("Unit: Testing Filters", function() {

  beforeEach(module('App'));

  it('should have a range filter', inject(function($filter) {
    expect($filter('range')).not.to.equal(null);
  }));

  it('should have a range filter that produces an array of numbers',
    inject(function($filter) {

    var range = $filter('range')([], 5);
    expect(range.length).to.equal(5);
    expect(range[0]).to.equal(0);
    expect(range[1]).to.equal(1);
    expect(range[2]).to.equal(2);
    expect(range[3]).to.equal(3);
    expect(range[4]).to.equal(4);
  }));

  it('should return null when nothing is set',
    inject(function($filter) {

    var range = $filter('range')();
    expect(range).to.equal(null);
  }));

  it('should return the input when no number is set',
    inject(function($filter) {

    var range, input = [1];
    range = $filter('range')(input);
    expect(range).to.equal(input);

    range = $filter('range')(input, 0);
    expect(range).to.equal(input);

    range = $filter('range')(input, -1);
    expect(range).to.equal(input);

    range = $filter('range')(input, 'Abc');
    expect(range).to.equal(input);
  }));

});
```

```
//
// test/midway/filters/filtersSpec.js
//
describe("Midway: Testing Filters", function() {

  var tester;
  beforeEach(function() {
    tester = ngMidwayTester('App');
  });

  afterEach(function() {
    tester.destroy();
    tester = null;
  });

  it('should have a working range filter',
    function() {

    expect(tester.inject('$filter')('range')).not.to.equal(null);
  });

  it('should have a working filter that updates the DOM',
    function(done) {

    var id = 'temp-filter-test-element';
    var html = '<div id="' + id + '"><div class="repeated" ng-repeat="n in [] | range:10">...</div></div>';
    var element = angular.element(html);

    var scope = tester.rootScope().$new();
    tester.compile(element, scope);

    var elm = element[0];
    setTimeout(function() {
      var kids = elm.getElementsByTagName('div');
      expect(kids.length).to.equal(10);
      done();
    },1000);
  });

});
```

```
//
// test/e2e/filters/filtersSpec.js
//
describe("E2E: Testing Filters", function() {

  beforeEach(function() {
    browser().navigateTo('/');
  });

  it('should have a filter that expands the stars properly', function() {
    browser().navigateTo('#/videos/zogrnQjHZAM');
    expect(repeater('#app-youtube-stars > .app-youtube-star').count()).toBeGreaterThan(0);
  });

});
```

Testing Templates, Partials & Views

What's our testing goal?
To check to see that the template is set and is downloaded from the server. To check to see the correct template is assigned to a specific controller. To check to see that the template was assigned to the view and/or include.

Templates, partials and/or views are all essentially bodies of isolated HTML code which are injected into your AngularJS application. Whether templates are rendered with the use of the *ngView* directive or with *ngInclude*, templated HTML code can be injected quite easily into your webpage.

Views and includes work with the caching mechanisms provided inside of the *$templateCache* service. So if you were to mock template code then this can be achieved with simply adding HTML into the template cache with the cache key being the same as your templateUrl value.

Midway Testing E2E Testing

```
//
// test/midway/templates/templatesSpec.js
//
describe("Midway: Testing Templates", function() {

  it("should load the template for the videos page properly",
    function(done) {

    var tester = ngMidwayTester('App');
    tester.visit('/videos?123', function() {
      var current = tester.inject('$route').current;
      var controller = current.controller;
      var template = current.templateUrl;
      expect(template).to.match(/templates\/views\/videos\/index_tpl\.html/);
      tester.destroy();
      done();
    });
  });

});
```

```
//
// test/e2e/templates/templatesSpec.js
//
describe("E2E: Testing Templates", function() {

  beforeEach(function() {
    browser().navigateTo('/');
  });

  it('should redirect and setup the videos page template on root', function() {
    browser().navigateTo('#/');
    expect(element('#ng-view').html()).toContain('youtube_listing');
  });

  it('should load the watched videos template into view', function() {
    browser().navigateTo('#/watched-videos');
    expect(element('#ng-view').html()).toContain('youtube_listing');
  });

  it('should load the watched video template into view', function() {
    browser().navigateTo('#/videos/123');
    expect(element('#ng-view').html()).toContain('profile');
  });

  it('should redirect back to the index page if anything fails', function() {
    browser().navigateTo('#/something/else');
    expect(element('#ng-view').html()).toContain('youtube_listing');
  });
});
```

Testing Directives

What's our testing goal?
To check to see that the directive does its work to the view/DOM.

Directives are a major component of an AngularJS application and testing them is equally as important. When testing directives you want to see that when the directive is executed it does what you *expect it to do to the $scope and/or DOM*. Directives themselves may also, based on design, fire off XHR requests in the background, but this is quite unlikely and not a good practice to do. Therefore to test directives it's best to *test them in all three areas so that you get a full overview* to see if they're functioning properly.

Unit Testing Midway Testing E2E Testing

```
//
// test/unit/directives/directivesSpec.js
//
describe("Unit: Testing Directives", function() {

  var $compile, $rootScope;

  beforeEach(module('App'));

  beforeEach(inject(
    ['$compile','$rootScope', function($c, $r) {
      $compile = $c;
      $rootScope = $r;
    }]
  ));

  it("should display the welcome text properly", function() {
    var element = $compile('<div data-app-welcome>User</div>')($rootScope);
    expect(element.html()).to.match(/Welcome/i);
  })

});
```

```
//
// test/midway/directives/directivesSpec.js
//
describe("Midway: Testing Directives", function() {

  var tester;
  beforeEach(function() {
    tester = ngMidwayTester('App');
  });

  afterEach(function() {
    tester.destroy();
    tester = null;
  });

  it("should properly create the youtube listings with the directive in mind", function(done) {
    var $youtube = tester.inject('$appYoutubeSearcher');

    var html = '';
    html += '<div data-app-youtube-listings id="app-youtube-listings">';
    html += ' <div data-ng-include="\'templates/partials/youtube_listing_tpl.html\'" data-ng-repeat="video in videos"></div>';
    html += '</div>';

    $youtube.query('latest', false, function(q, videos) {
      var scope = tester.viewScope().$new();
      scope.videos = videos;
      var element = tester.compile(html, scope);
      setTimeout(function() {
        var klass = (element.attr('class') || '').toString();
        var hasClass = /app-youtube-listings/.exec(klass);
        expect(hasClass.length).to.equal(1);

        var kids = element.children('.app-youtube-listing');
        expect(kids.length > 0).to.equal(true);
        done();
      },1000);
    });
  });

});
```

```
//
// test/e2e/directives/directivesSpec.js
//
describe("E2E: Testing Directives", function() {

  beforeEach(function() {
    browser().navigateTo('/');
  });

  it('should have a working welcome directive apply it\'s logic to the page', function() {
    browser().navigateTo('#!/videos');
    expect(browser().location().path()).toBe("/videos");
    expect(element('#app-welcome-text').html()).toContain('Welcome');
  });

  it('should have a working youtube listing directive that goes to the right page when clicked', function() {
    browser().navigateTo('#!/videos');
    element('.app-youtube-listing').click();
    expect(browser().location().path()).toMatch(/\/videos\/.+/);
  });

});
```

Testing Resources

What's our testing goal?
To check to see the resource is available and that it is a resource. To check to see that the resource request methods work and return data. To check to see that the data is saved and persists between requests.

Resources are special object instantiations which are crafted together via the *$resource service* which is *provided via dependency injection*. Testing a resource requires all levels of testing and is a bit complicated. This particular testing approach can be very *low level* (testing every possible input and output of the resource) or it can be *high level* (testing to see that the resource is created and updated and that its data persists between requests).

Unfortunately, testing resources ties in heavily with your server side setup. Since the Github Repository is a static website, then this means that there is no persistence on the server (no database or back-end). Therefore, testing resources on this code would be pointless.

Testing resources itself requires its own article/tutorial and I'll be sure to blog about it in the near future...

Testing Animations

Animations can now be tested in AngularJS and this is explained in another yearofmoo article called Enhanced Animation in AngularJS .

Click here to view the *Animation Article Testing Code*

Testing 3rd Party Code (JavaScript)

What's our testing goal?
To test all public function calls for all 3rd party code. To hard code the version of the code into the tests and update that version each time the plugin is updated.

Testing 3rd party code isn't what is sounds like and doing so isn't a strict requirement for testing AngularJS applications. But it is highly important. Basically whenever you include a plugin or tool into your application and *interface with its API* you will need to keep track of what *public methods/functions you use as well as the type of data you expect to get back* (whether that be with JavaScript objects or function calls).

Why does this matter? Well normally it shouldn't (if you were using a compiled language), however, when it comes to JavaScript applications, there is no type checking or compile-time bind checking going on in the background. Therefore, you will need to keep track of what you use yourself. So lets say that you use a tool for enhancing your forms and integrate it into your application. Then, later on, you (or someone else from your team) updates the plugin to an older or newer version and your code breaks (because of API changes). You, or your client(s), then, sometime down the road, find out that this is broken and now you're in a rush to figure out if it's the plugin that's broken or your website, but then you end up finding out that it is really just your glue code that has failed.

To avoid this simple problem, *just test the inputs and outputs of your 3rd party code*.

To best test for 3rd party code, create a special test runner configuration file for karma inside the config directory of your AngularJS application (something like karma.3rd_party.conf.js). Inside that file include all the code required to emulate the input and output methods and code to run the 3rd party tools. Then just make sure that the API calls do what they should and setup your testing code to test for it.

Conclusion

As you can see, it's not too difficult to test your AngularJS application. *E2E* and *Unit* testing is built right in and they both work very well and as expected. *Midway* testing is more complicated, but provides you with more access to deeper parts of your application and it is very useful for testing external services. Karma is an incredible tool and now you have access to a github repository that provides full out configurations to get it to work 100%.

Keep testing and let me know how it works for you :) And thank you for taking the time to read the article. Please *share this article* and *spread the word*!