Tuesday, May 21, 2013

AngularJS - Unit Testing - Controllers

Updated Feb 6, 2014 - I'm changing this around a lot. A lot as changed since I wrote the original article and a lot has stayed the same. Most notably how Angular 1.2 handles promises on the $scope when processing the view, and some changes around testing promises. Since I see this gets some traffic and I *hate* the idea that I'm showing people the wrong thing, I'm going to try to keep this updated as time goes on. (Angular 2.0 will probably be a whold new post though)

Since Controllers carry the "business logic" of your Angular application, they're probably the single most important thing to unit test in your Application. I've run across a few tutorials on this subject, but most of them cover only the simplest scenarios. I'm going to try to add some slightly more complicated stuff in to my controller and test it, just to show examples. As I think of new examples as time goes on, I'll try to add those too.

For Unit Testing Services and Directives see these other Posts:

- AngularJS Unit Testing Services
- AngularJS Unit Testing Directives

The Controller: What are you testing?

First off, what is a controller? A controller is an instance of an object defined by executing the controller function as a class constructor. If you're new to JavaScript, that means it's calling the function, but in the context of creating an object. All of this aside, *most* of what a controller is doing is setting up your $scope object with properties and functions you can use to wire it to a view. This will be the lion's share of what you're testing.

Recommended Testing Suite: Jasmine

The recommended tool for testing Angular is Jasmine. You can, of course, use any unit testing tool you like, but for this blog entry, we'll be using Jasmine. To get started with Jasmine they have some really well annotated code on their site as a tutorial of sorts, but I'd recommend just going to Plunker and starting a new "Angular + Jasmine" Plunk and fiddling around until you get the hang of it.

To TDD or not to TDD? Yes.

I'm not going to go into the specifics of TDD, whether or not you should use it, the pros and cons of TDD, or even attack this blog entry from that angle. I'm going to assume that if you're here, you've probably written some Angular controller, or you know how to write an Angular controller, and you're thinking "how do I test this thing?". So we'll just cover some of those basics, mkay?

Recommended for Later: Karma or Grunt automated tests

Generally, I'd recommend using something like Karma or grunt-contrib-jasmine to run your unit tests automatically in Node... But that's probably another lesson for another day. For now, let's just learn some Jasmine (1.3.X) basics.

Ben Lesh

Follow 246

View my complete profile

Follow @BenLesh

Follow @blesh

View my profile on LinkedIn

Links

- ALE Framework on Github

Tags

- AJAX (1)
- ALE (7)
- AngularJS (23)
- Announcement (1)
- ASP.Net (5)
- Async (3)
- C# (13)
- Cassette (1)
- Dependency Injection (1)
- EmberJS (7)
- Embular (6)
- ES6 (1)
- File I/O (1)
- Form Validation (2)
- Fun (2)
- git (1)
- github (1)
- IIS (1)
- Install (1)
- Jasmine (1)
- Java (1)
- JavaScript (43)
- JQuery (3)
- JRE (1)
- Linux (1)
- Memory (1)
- MVC (3)
- MVC 3 (2)
- OpenJRE (1)
- Pittsburgh (1)
- Programming Languages (3)
- Rant (4)
- Security (3)
- Strings (1)
- SVG (2)
- Threading (6)

Right Now: Jasmine Basics

Let's start off with the basic Jasmine Set up. This is what's required to run the Jasmine specs you're going to write, and produce a report in HTML format you can read. So to do all of this, you will create some HTML file, we'll call it "index.html" for now. and this would be the basic content of it:

```
<!DOCTYPE html>
<html>

<head>
  <!-- jasmine -->
  <script src="//cdnjs.cloudflare.com/ajax/libs/jasmine/1.3.1/jasmine.js"></script>
  <!-- jasmine's html reporting code and css -->
  <script src="//cdnjs.cloudflare.com/ajax/libs/jasmine/1.3.1/jasmine-html.js"></script>
  <link href="//cdnjs.cloudflare.com/ajax/libs/jasmine/1.3.1/jasmine.css" rel="styleshee
  <!-- angular itself -->
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.11/angular.js"></script>
  <!-- angular's testing helpers -->
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.11/angular-mocks.js"></scri
  <!-- your angular app code -->
  <script src="app.js"></script>
  <!-- your Jasmine specs (tests) -->
  <script src="specs.js"></script>
</head>
<body>
  <!-- bootstrap jasmine! -->
  <script>
  var jasmineEnv = jasmine.getEnv();

  // Tell it to add an Html Reporter
  // this will add detailed HTML-formatted results
  // for each spec ran.
  jasmineEnv.addReporter(new jasmine.HtmlReporter());

  // Execute the tests!
  jasmineEnv.execute();
  </script>
</body>
</html>
```

index.html hosted with ❤ by **GitHub** **view raw**

An Example Controller

So now we'll need something to test. I'm going to make up a completely contrived controller to create some unit testing examples against. Nothing special, and nothing that might even make sense. It's just different things you might commonly do in an Angular controller, that you might need to test. In the specsRunner.html file above, this would be our "app.js".

```
var app = angular.module('myApp', []);

/* Set up a simple controller with a few
 * examples of common actions a controller function
 * might set up on a $scope. */
app.controller('MainCtrl', function($scope, someService) {

  //set some properties
  $scope.foo = 'foo';
  $scope.bar = 'bar';


  //add a simple function.
  $scope.test1 = function (){
    $scope.foo = $scope.foo + '!!!';
  };

  //set up a $watch.
  $scope.$watch('bar', function (v){
    $scope.baz = v + 'baz';
  });

  //make a call to an injected service.
```


- ThreeJS (1)
- Traceur (1)
- Ubuntu (1)
- Unit Testing (3)
- Validation (3)
- Web API (1)
- Web Development (34)
- Web Sockets (1)
- Workers (1)

Blog Archive

- ▶ 2014 (12)
- ▼ 2013 (16)
  - ▶ November (2)
  - ▶ October (1)
  - ▶ August (3)
  - ▶ June (3)
  - ▼ May (1)
    AngularJS - Unit Testing - Controllers
  - ▶ March (1)
  - ▶ February (4)
  - ▶ January (1)
- ▶ 2012 (34)

```
  $scope.test2 = fumction (){
    //an async call returning a promise that
    //inevitably returns a value to a property.
    someService.someAsyncCall($scope.foo)
      .then(fumction(data) {
        $scope.fizz = data;
      });
  };
});


/* Simple service example.
 * This is a service created just to use as an example of
 * some simple service that is making some asynchronous call.
 * A real-life example of something like this would be a
 * service that is making $http or $resource calls, perhaps. */
app.factory('someService', fumction ($timeout, $q){
  return {

    // simple method to do something asynchronously.
    someAsyncCall: fumction (x){
      var deferred = $q.defer();
      $timeout(fumction (){
        deferred.resolve(x + '_async');
      }, 100);
      return deferred.promise;
    }
  };
});
```

app.js hosted with ❤ by GitHub view raw

Unit Tests For Our Example Controller

So, given the above controller, here is a battery of unit tests that tests the behavior of this controller. Well, more importantly, it tests what has been set up on the $scope by the controller function. In the specsRunner html (above), this would be in our "specs.js":

```
describe('Testing a controller', fumction() {
  var $scope, ctrl, $timeout;

  /* declare our mocks out here
   * so we can use them through the scope
   * of this describe block.
   */
  var someServiceMock;


  // This function will be called before every "it" block.
  // This should be used to "reset" state for your tests.
  beforeEach(fumction (){
    // Create a "spy object" for our someService.
    // This will isolate the controller we're testing from
    // any other code.
    // we'll set up the returns for this later
    someServiceMock = jasmine.createSpyObj('someService', ['someAsyncCall']);

    // load the module you're testing.
    module('myApp');

    // INJECT! This part is critical
    // $rootScope - injected to create a new $scope instance.
    // $controller - injected to create an instance of our controller.
    // $q - injected so we can create promises for our mocks.
    // _$timeout_ - injected to we can flush unresolved promises.
    inject(fumction($rootScope, $controller, $q, _$timeout_) {
      // create a scope object for us to use.
      $scope = $rootScope.$new();

      // set up the returns for our someServiceMock
      // $q.when('weee') creates a resolved promise to "weee".
      // this is important since our service is async and returns
      // a promise.
      someServiceMock.someAsyncCall.andReturn($q.when('weee'));

      // assign $timeout to a scoped variable so we can use
      // $timeout.flush() later. Notice the _underscore_ trick
      // so we can keep our names clean in the tests.
      $timeout = _$timeout_;

```

```
43        // now run that scope through the controller function,
44        // injecting any services or other injectables we need.
45        // **NOTE**: this is the only time the controller function
46        // will be run, so anything that occurs inside of that
47        // will already be done before the first spec.
48        ctrl = $controller('MainCtrl', {
49          $scope: $scope,
50          someService: someServiceMock
51        });
52      });
53    });
54
55
56    /* Test 1: The simplest of the simple.
57     * here we're going to test that some things were
58     * populated when the controller function whas evaluated. */
59    it('should start with foo and bar populated', function() {
60
61      //just assert. $scope was set up in beforeEach() (above)
62      expect($scope.foo).toEqual('foo');
63      expect($scope.bar).toEqual('bar');
64    });
65
66
67    /* Test 2: Still simple.
68     * Now let's test a simple function call. */
69    it('should add !!! to foo when test1() is called', function (){
70      //set up.
71      $scope.foo = 'x';
72
73      //make the call.
74      $scope.test1();
75
76      //assert
77      expect($scope.foo).toEqual('x!!!');
78    });
79
80
81    /* Test 3: Testing a $watch()
82     * The important thing here is to call $apply()
83     * and THEN test the value it's supposed to update. */
84    it('should update baz when bar is changed', function (){
85      //change bar
86      $scope.bar = 'test';
87
88      //$apply the change to trigger the $watch.
89      $scope.$apply();
90
91      //assert
92      expect($scope.baz).toEqual('testbaz');
93    });
94
95
96    /* Test 4: Testing an asynchronous service call.
97       Since we've mocked the service to return a promise
98       (just like the original service did), we need to do a little
99       trick with $timeout.flush() here to resolve our promise so the
100      `then()` clause in our controller function fires.
101
102      This will test to see if the `then()` from the promise is wired up
103      properly. */
104   it('should update fizz asynchronously when test2() is called', function (){
105     // just make the call
106     $scope.test2();
107
108     // asser that it called the service method.
109     expect(someServiceMock.someAsyncCall).toHaveBeenCalled();
110
111     // call $timeout.flush() to flush the unresolved dependency from our
112     // someServiceMock.
113     $timeout.flush();
114
115     // assert that it set $scope.fizz
116     expect($scope.fizz).toEqual('weee');
117   });
118 });
```

specs.js hosted with ❤ by **GitHub** **view raw**

The Simple Tests

I don't want to dwell too much on the first two tests. They're fairly straight forward, and I don't want to patronize anyone that's made it this far. They're your basic, basic, unit tests. Make a call, assert a value, the end.

Testing a $watch()

Okay, here there's a little trick. If you have a $watch set up on a property, or on anything really, and you want to test it, all you need to do is update whatever you're watching on the $scope (or wherever it is), then call $scope.$apply(). Calling $apply will force a digest which will process all of your $watches.

Testing Service Calls

Testing services calls is easy: Mock the service, spy on it's methods, use expect(service.method).toHaveBeenCalled() or expect(service.method).toHaveBeenCalledWith(arg1, arg2) to verify it's been called. Pretty simple.

...and Asynchronous Service Calls

Testing async calls with Jasmine in Angular gets a little different than it might be with other frameworks. The first thing we did was isolate the controller from it's service with a mock, but that's not the end of it, since we have to handle the promise it returns by calling .then() on it. So there are a few things to make sure you're doing here:

1. Have your mock service return a resolved Angular promise by using $q.when('returned data here').
2. Use $timeout.flush() to force unresolved promises to resolve.

View the complete example on Plunker

Checkout the Gist as well

This is just a start

This blog entry really only covers the basics of testing controllers, there are a great many unique situations that can come up while you're unit testing.

Things to consider: If it's hard to test, maybe it needs refactored? Anything that's hard to test probably has issues with interdependence or functions that try to do too much in one go and a refactor should be considered.

Posted by Ben Lesh at 7:53 PM

Labels: AngularJS, Jasmine, JavaScript, MVC, Unit Testing, Web Development

22 comments:

Michael June 26, 2013 at 2:43 PM

This is a very clear, up to date tutorial. Thank you.

Reply

Arild Nilsen July 24, 2013 at 7:12 AM

Illustrative naming and well commented. Up to date. Great tutorial!

Reply

James Vanmeman July 27, 2013 at 8:42 AM

Thanks for the post, very well concise and well written

Reply

c. alex naser August 7, 2013 at 10:46 AM

this is brilliant. thanks so much, dude!

Reply

modernbacchanal August 13, 2013 at 6:56 PM

thank you nice tutorial

Reply

Sharondio August 13, 2013 at 7:23 PM

Nicest and clearest explanation I've seen so far.

Reply

Емил Табаков August 29, 2013 at 12:52 AM

Very nice walk-through for beginners like me. I have one question though - how can I test private functions that are part of controllers and are used in its methods? For example, I want to test the validate function directly ?:

app.controller('MainCtrl', function($scope, someService) {
//..... some stuff

//set up a $watch.
$scope.$watch('bar', function (v){
$scope.baz = v + 'baz';
});

//make a call to an injected service.
$scope.test2 = function (){
//an async call returning a promise that
//inevitably returns a value to a property.
validate();
$scope.fizz = someService.someAsyncCall($scope.foo);
};

function validate() {
//some logic to test
}
});

Reply

Replies

Benjamin Lesh August 29, 2013 at 6:43 AM

In that case, you'll want to make validate a member variable of the controller, by declaring it like: this.validate = function (){ /*...*/ };

After you do that, in the tests, you can call it right off of the ctrl variable:
it('should validate', function () {
expect(ctrl.validate()).toBe(true);
});

Benjamin Lesh August 29, 2013 at 6:44 AM

the primary point being, you can't test private functions/methods directly, no matter what language you're developing in... because they're "private" and inaccessible.

Reply

Kenn Brodhagen September 4, 2013 at 6:11 PM

I like how you avoid having to deal with the promise by having the controller stick it right into a scope variable and let the view deal with it. Are you able to stick to this pattern all the time or do you ever have to deal with then()s inside the controller? For example error handling. I've been creating mocks that return promises but it makes for a lot of test code

and a lot of people don't grock the rootScope.$apply() floating in the middle of the test since it doesn't seem as descriptive as the httpBackend.flush().

Thanks for the post!

Reply

Benjamin Lesh September 5, 2013 at 6:27 AM

You can do things this way right up to the point that the controller needs to filter or manipulate that returned value in some way. Then you're going to have to call .then() yourself and stick the value in a $scope property. So, for example, if you needed to have some functionality in your controller to aggregate values in a dynamic way, you'd want to be able to test that. It just becomes too much of a pain to deal with if you only have a promise in your scope property.

Reply

Dilip Yadav October 1, 2013 at 9:12 AM

Simple and great way of explaining Benjamin. This is the best article I've seen so far. Thanks!

Reply

Adam Napora October 31, 2013 at 2:58 AM

Very good, this makes sense and is helping a lot, thanks!

Reply

samsarum November 21, 2013 at 8:13 PM

Nice! This blog give me an idea on how to test controller. :)

Reply

moshevi December 2, 2013 at 8:12 AM

This tutorial give me a direction. Thank you very much for your work!

Reply

Logan December 21, 2013 at 10:40 AM

Great tutorial for getting me setup with writing my first unit test with Jasmine and Angular. Thanks!

Reply

Chris February 6, 2014 at 3:15 AM

I think your code is not valid anymore. as of angular > 1.2 assigning the promise of you someService.someAsyncCall directly to a $scope value (and eventually resolving to the value in the view) does not work anymore: https://github.com/angular/angular.js/commit/5dc35b527b3c99f6544b8cb52e93c6510d3ac577

Likewise mocking services to not return promises but raw data is no viable solution anymore? Am i wrong on this?

Reply

Replies

Ben Lesh February 6, 2014 at 7:09 AM

No, you're right. I haven't come back to blog entry since 1.2 was released. I'll be sure to update it.

Reply

Gurmeet singh August 8, 2014 at 8:17 AM

Very Good
But if u post some more post on unit testing with controller,private method etc.
Do some post on beforeeach and aftereach with example.
It s more helpfull to developer community.
At the end,Excellent Work

Reply

Gehan August 18, 2014 at 4:08 AM

This message is related to the previous comment.This is the unit test is implement to test this scenario.

Unit test used to test controller

```
"use strict";

describe("Controller : Login controller", function()
{
var $scope, ctrl, $location, $timeout, WebServiceMock;

beforeEach(module('Apps', function($provide)
{
WebServiceMock = jasmine.createSpyObj("WebService", ["loginRequest"]);
WebServiceMock.loginRequest.andReturn({ UserID:'1', SessionId:'1118430',
Status:'LOGIN_SUCCESS', EMail:'v@v.com', FirstName:'Viranga'});
$provide.value("WebServiceMock", WebServiceMock)

}));

beforeEach(inject(function($rootScope, $controller, $q, _$timeout_, _$location_,
_WebServiceMock_)
{
$scope = $rootScope.$new();
$timeout = _$timeout_;
$location = _$location_;

ctrl = $controller('loginCtrl',{
$scope: $scope,
$location: $location,
WebService: _WebServiceMock_,
$timeout: $timeout
});

$scope.$digest();
}));

it('should call loginRequest WebServiceMock', function ()
{
$scope.username = 'v@v.com';
$scope.password = 'viranga123';
$scope.$digest();

var result =$scope.login();
expect(WebServiceMock.loginRequest).toHaveBeenCalled();

//Validate recieved data and Check redirection
spyOn($scope, '$on').andCallThrough();
expect(result.Status).toEqual("LOGIN_SUCCESS");//Error here
});
});
```

Could you please suggest what changes should be made in order to implement this test correctly ?

Reply

Replies

Ben Lesh August 18, 2014 at 9:29 AM

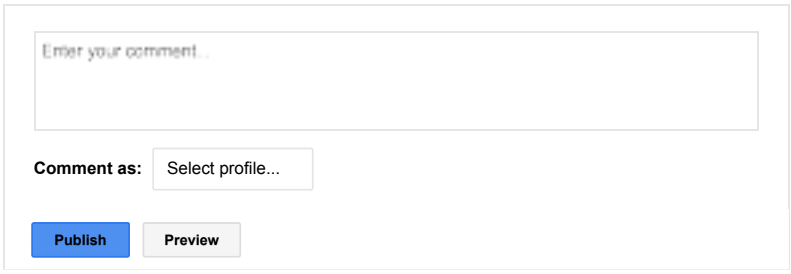
I'm happy to help, but It's probably best to post all of that on a site like StackOverflow, then just post the link. You'll get help from me, plus a million others.

Reply

Gehan August 20, 2014 at 5:48 AM

Sorry for the message clutter.I managed to complete this test by testing the Broadcast event.and i have posted the updated to StackOverflow - http://goo.gl/KB2ga6

Reply



This form allows some basic HTML. It will only create links if you wrap the URL in an anchor tag (Sorry, it's the Blogger default)

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)