

An Introduction To Unit Testing In AngularJS Applications

By Sébastien Fragnaud

Published on October 7th, 2014 in JavaScript, Programming, Testing with 18 Comments

AngularJS¹ has grown to become one of the most popular single-page application frameworks. Developed by a dedicated team at Google, the outcome is substantial and widely used in both community and industry projects.

One of the reasons for AngularJS' success is its outstanding ability to be tested. It's strongly supported by Karma¹¹² (the spectacular test runner written by Vojta Jína) and its multiple plugins. Karma, combined with its fellows Mocha¹⁷³, Chai¹⁸⁴ and Sinon²⁰⁵, offers a complete toolset to produce quality code that is easy to maintain, bug-free and well documented.

"Well, I'll just launch the app and see if everything works. We've never had any problem doing that."

– No one ever

The main factor that made me switch from "Well, I just launch the app and see if everything works" to "I've got unit tests!" was that, for the first time, I could **focus on what matters** and on what I enjoy in programming: creating smart algorithms and nice UIs.

I remember a component that was supposed to manage the right-click menu in an application. Trust me, it was a complex component. Depending on dozens of mixed conditions, it could show or hide buttons, submenus, etc. One day, we updated the application in production. I can remember how I felt when I launched the app, opened something, right-clicked and saw no contextual menu — just an empty ugly box that was definitive proof that something had gone really wrong. After having fixed it, re-updated the application and apologized to customer service, I decided to entirely rewrite this component in test-driven

development style. The test file ended up being twice as long as the component file. It has been improved a lot since, especially its poor performance, but it never failed again in production. Rock-solid code.

A Word About Unit Testing

Unit testing has become a standard in most software companies. Customer expectations have reached a new high, and no one accepts getting two free regressions for the price of one update anymore.

If you are familiar with unit testing, then you'll already know how confident a developer feels when refactoring tested code. If you are not familiar, then imagine getting rid of deployment stress, a "code-and-pray" coding style and never-ending feature development. The best part of? It's automatic.

Unit testing improves code's orthogonality. Fundamentally, code is called "orthogonal" when it's easy to change. Fixing a bug or adding a feature entails nothing but changing the code's behavior, as explained in *The Pragmatic Programmer: From Journeyman to Master*⁶. Unit tests greatly improve code's orthogonality by forcing you to write modular logic units, instead of large code chunks.

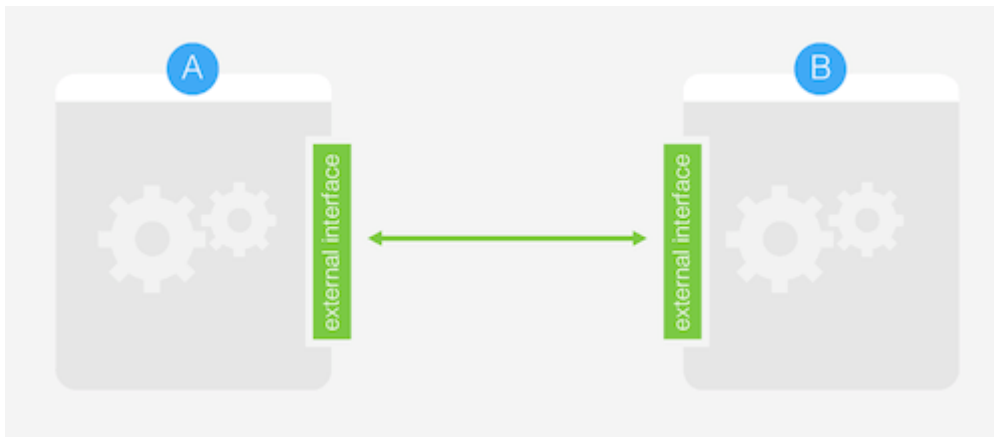
Unit testing also provides you with documentation that is always up to date and that informs you about the code's intentions and functional behavior. Even if a method has a cryptic name — which is bad, but we won't get into that here — you'll instantly know what it does by reading its test.

Unit testing has another major advantage. It forces you to actually use your code and detect design flaws and bad smells. Take functions. What better way to make sure that functions are uncoupled from the rest of your code than by being able to test them without any boilerplate code?

Furthermore, **unit testing opens the door to test-driven development**. While it's not this article's topic, I can't stress enough that test-driven development is a wonderful and productive way to write code.

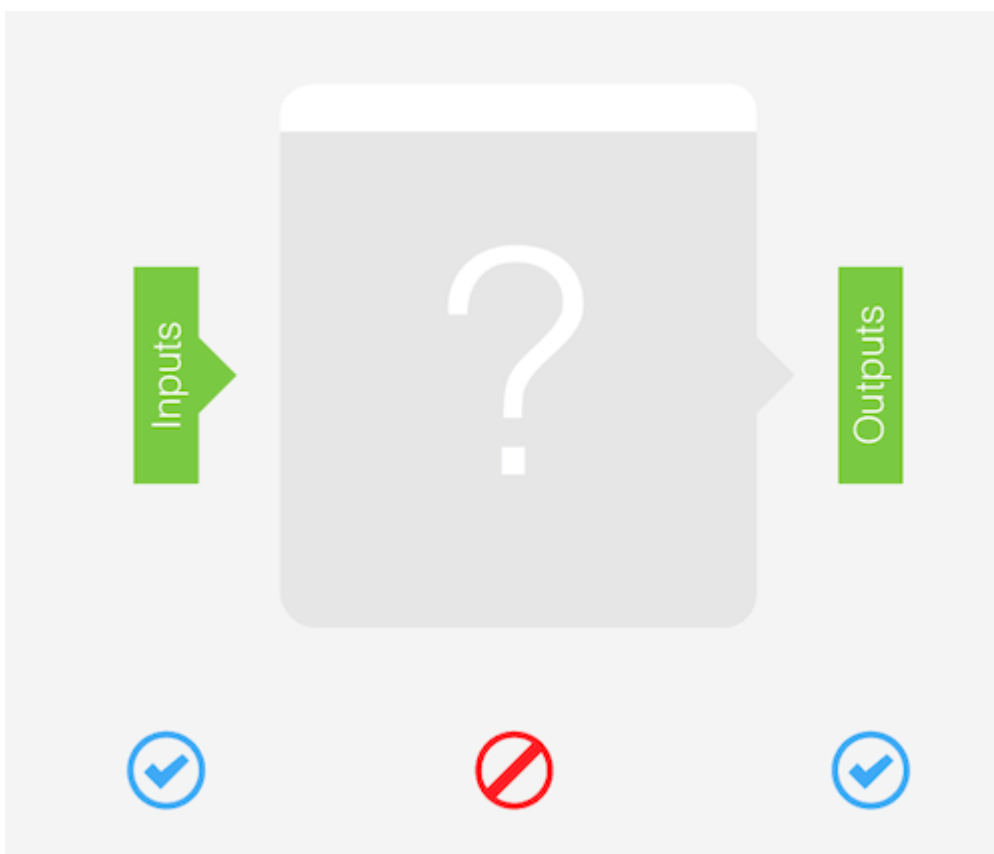
WHAT AND WHAT NOT TO TEST

Tests must define the code's API. This is the one principle that will guide us through this journey. An AngularJS application is, by definition, composed of modules. The elementary bricks are materialized by different concepts related to the granularity at which you look at them. At the application level, these bricks are AngularJS' modules. At the module level, they are directives, controllers, services, filters and factories. Each one of them is able to communicate with another through its external interface.



Everything is bricks, regardless of the level you are at. ([View large version](#)⁸)

All of these bricks share a common attribute. They behave as black boxes, which means that they have an inner behavior and an outer interface materialized by inputs and outputs. This is precisely what unit tests are for: to test bricks' outer interfaces.



Black box model (well, this one is gray, but you get the idea) ([View large version](#)¹⁰)

Ignoring the internals as much as possible is considered good practice. Unit testing — and testing in

general — is a mix of stimuli and reactions.

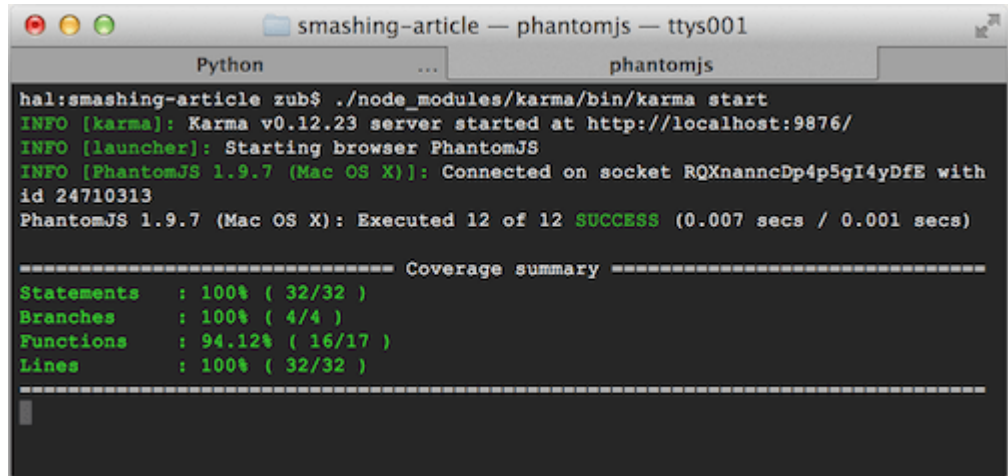
Bootstrapping A Test Environment For AngularJS

To set up a decent testing environment for your AngularJS application, you will need several npm modules. Let's take a quick glance at them.

KARMA: THE SPECTACULAR TEST RUNNER

Karma¹² is an engine that runs tests against code. Although it has been written for AngularJS, it's not specifically tied to it and can be used for any JavaScript application. It's highly configurable through a JSON file and the use of various plugins.

12

A terminal window titled "smashing-article — phantomjs — ttys001" with tabs for "Python" and "phantomjs". The terminal output shows the command `./node_modules/karma/bin/karma start` being executed. The output includes status messages from Karma and PhantomJS, indicating a successful test run. A coverage summary is displayed at the bottom of the output.

```
hal:smashing-article zub$ ./node_modules/karma/bin/karma start
INFO [karma]: Karma v0.12.23 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.7 (Mac OS X)]: Connected on socket RQXnanncDp4p5gI4yDfE with
id 24710313
PhantomJS 1.9.7 (Mac OS X): Executed 12 of 12 SUCCESS (0.007 secs / 0.001 secs)

===== Coverage summary =====
Statements : 100% ( 32/32 )
Branches   : 100% ( 4/4 )
Functions  : 94.12% ( 16/17 )
Lines      : 100% ( 32/32 )
=====
```

If you don't see this at some point in the process, then you might have missed something. ([View large version](#)¹³)

All of the examples in this article can be found in the [dedicated GitHub project](#)¹⁴, along with the following configuration file for Karma.

```

// Karma configuration
// Generated on Mon Jul 21 2014 11:48:34 GMT+0200 (CEST)
module.exports = function(config) {
  config.set({

    // base path used to resolve all patterns (e.g. files, exclude)
    basePath: '',

    // frameworks to use
    frameworks: ['mocha', 'sinon-chai'],

    // list of files / patterns to load in the browser
    files: [
      'bower_components/angular/angular.js',
      'bower_components/angular-mocks/angular-mocks.js',
      'src/*.js',
      'test/*.mocha.js'
    ],

    // list of files to exclude
    exclude: [],

    // preprocess matching files before serving them to the browser
    preprocessors: {
      'src/*.js': ['coverage']
    },

    coverageReporter: {
      type: 'text-summary',
      dir: 'coverage/'
    },

    // test results reporter to use
    reporters: ['progress', 'coverage'],

    // web server port
    port: 9876,

    // enable / disable colors in the output (reporters and logs)
    colors: true,

    // level of logging
    logLevel: config.LOG_INFO,

    // enable / disable watching file and executing tests on file changes
    autoWatch: true,

    // start these browsers
    browsers: ['PhantomJS'],

    // Continuous Integration mode
    // if true, Karma captures browsers, runs the tests and exits
    singleRun: false
  });
};

```

This file can be automagically generated by typing `karma init` in a terminal window. The available keys

are [described in Karma's documentation](#)¹⁵.

Notice how sources and test files are declared. There is also a newcomer: [ngMock](#)¹⁶ (i.e. `angular-mocks.js`). ngMock is an AngularJS module that provides several testing utilities (more on that at the end of this article).

MOCHA

[Mocha](#)¹⁷³ is a testing framework for JavaScript. It handles test suites and test cases, and it offers nice reporting features. It uses a declarative syntax to nest expectations into cases and suites. Let's look at the following example (shamelessly stolen from Mocha's home page):

```
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal(-1, [1,2,3].indexOf(5));
      assert.equal(-1, [1,2,3].indexOf(0));
    });
  });
});
```

You can see that the whole test is contained in a `describe` call. What is interesting about nesting function calls in this way is that the **tests follow the code's structure**. Here, the `Array` suite is composed of only one subsuite, `#indexOf`. Others could be added, of course. This subsuite is composed of one case, which itself contains two assertions and expectations. Organizing test suites into a coherent whole is essential. It ensures that test errors will be reported with meaningful messages, thus easing the debugging process.

CHAI

We have seen how Mocha provides test-suite and test-case capabilities for JavaScript. [Chai](#)¹⁸⁴, for its part, **offers various ways of checking things** in test cases. These checks are performed through what are called "assertions" and basically mark a test case as failed or passed. [Chai's documentation has more](#)¹⁹ on the different assertions styles.

SINON

[Sinon](#)²⁰⁵ describes itself as "standalone test spies, stubs and mocks for JavaScript." Spies, stubs and mocks all answer the same question: How do you efficiently replace one thing with another when running a test? Suppose you have a function that takes another one in a parameter and calls it. Sinon provides a smart and concise way to monitor whether the function is called and much more (with which arguments, how many times, etc.).

Unit Testing At The Application Level

The point of the external interface of a module in an AngularJS application is its ability to be injected into another module — that it exists and has a valid definition.

```
beforeEach(module('myAwesomeModule'));
```

This is enough and will throw an error if `myAwesomeModule` is nowhere to be found.

Unit Testing At The Module Level

An AngularJS module can declare several types of objects. Some are services, while others are more specialized. We will go over each of them to see how they can be bootstrapped in a controlled environment and then tested.

FILTERS, SERVICES AND FACTORIES: A STORY OF DEPENDENCY INJECTION

Filters, services and factories (we will refer to these as services in general) can be compared to static objects or singletons in a traditional object-oriented framework. They are easy to test because they need very few things to be ready, and these things are usually other services.

AngularJS links services to other services or objects using a very expressive dependency-injection model, which basically means asking for something in a method's arguments.

What is great about AngularJS' way of injecting dependencies is that mocking a piece of code's dependencies and injecting things into test cases are super-easy. In fact, I am not even sure it could be any simpler. Let's consider this quite useful factory:

```
angular.module('factories', [])
.factory('chimp', ['$log', function($log) {
  return {
    ook: function() {
      $log.warn('Ook.');
```

See how `$log` is injected, instead of the standard `console.warn`? While AngularJS will not print `$log` statements in Karma's console, avoid side effects in unit tests as much as possible. I once reduced by half the duration of an application's unit tests by mocking the tracking HTTP requests — which were all silently failing in a local environment, obviously.

```

describe('factories', function() {

  beforeEach(module('factories'));

  var chimp;
  var $log;

  beforeEach(inject(function(_chimp_, _$log_) {
    chimp = _chimp_;
    $log = _$log_;
    sinon.stub($log, 'warn', function() {});
  }));

  describe('when invoked', function() {

    beforeEach(function() {
      chimp.ook();
    });

    it('should say Ook', function() {
      expect($log.warn.callCount).to.equal(1);
      expect($log.warn.args[0][0]).to.equal('Ook. ');
    });
  });
});

```

The pattern for testing filters, services or other injectables is the same. Controllers can be a bit trickier to test, though, as we will see now.

CONTROLLERS

Testing a controller could lead to some confusion. What do we test? Let's focus on what a controller is supposed to do. You should be used to considering any tested element as a black box by now. Remember that AngularJS is a model-view-whatever (MVW) framework, which is kind of ironic because one of the few ways to define something in an AngularJS application is to use the keyword `controller`. Still, any kind of decent controller usually acts as a proxy between the model and the view, through objects in one way and callbacks in the other.

The controller usually configures the view using some state objects, such as the following (for a hypothetical text-editing application):


```
angular.module('textEditor', [])

.controller('EditionCtrl', ['$scope', function($scope) {
  $scope.state = {toolbarVisible: true, documentSaved: true};
  $scope.document = {text: 'Some text'};

  $scope.$watch('document.text', function(value) {
    $scope.state.documentSaved = false;
  }, true);

  $scope.saveDocument = function() {
    $scope.sendHTTP($scope.document.text);
    $scope.state.documentSaved = true;
  };

  $scope.sendHTTP = function(content) {
    // payload creation, HTTP request, etc.
  };
}]);
```

Chances are that the state will be modified by both the view and the controller. The `toolbarVisible` attribute will be toggled by, say, a button and a keyboard shortcut. Unit tests are not supposed to test interactions between the view and the rest of the universe; that is what end-to-end tests are for.

The `documentSaved` value will be mostly handled by the controller, though. Let's test it.

```

describe('saving a document', function() {

  var scope;
  var ctrl;

  beforeEach(module('textEditor'));

  beforeEach(inject(function($rootScope, $controller) {
    scope = $rootScope.$new();
    ctrl = $controller('EditionCtrl', {$scope: scope});
  }));

  it('should have an initial documentSaved state', function(){
    expect(scope.state.documentSaved).toEqual(true);
  });

  describe('documentSaved property', function() {
    beforeEach(function() {
      // We don't want extra HTTP requests to be sent
      // and that's not what we're testing here.
      sinon.stub(scope, 'sendHTTP', function() {});

      // A call to $apply() must be performed, otherwise the
      // scope's watchers won't be run through.
      scope.$apply(function () {
        scope.document.text += ' And some more text';
      });
    });

    it('should watch for document.text changes', function() {
      expect(scope.state.documentSaved).toEqual(false);
    });

    describe('when calling the saveDocument function', function() {
      beforeEach(function() {
        scope.saveDocument();
      });

      it('should be set to true again', function() {
        expect(scope.state.documentSaved).toEqual(true);
      });

      afterEach(function() {
        expect(scope.sendHTTP.callCount).toEqual(1);
        expect(scope.sendHTTP.args[0][0]).toEqual(scope.document.text);
      });
    });
  });
});

```

An interesting side effect of this code chunk is that it not only tests changes on the `documentSaved` property, but also checks that the `sendHTTP` method actually gets called and with the proper arguments (we will see later how to test HTTP requests). This is why it's a separated method published on the controller's scope. Decoupling and avoiding pseudo-global states (i.e. passing the text to the method, instead of letting it read the text on the scope) always eases the process of writing tests.

DIRECTIVES

A directive is AngularJS' way of teaching HTML new tricks and of encapsulating the logic behind those tricks. This encapsulation has several contact points with the outside that are defined in the returned object's `scope` attribute. The main difference with unit testing a controller is that directives usually have an isolated scope, but they both act as a black box and, therefore, will be tested in roughly the same manner. The test's configuration is a bit different, though.

Let's imagine a directive that displays a `div` with some string inside of it and a button next to it. It could be implemented as follows:

```
angular.module('myDirectives', [])
.directive('superButton', function() {
  return {
    scope: {label: '=', callback: '&onClick'},
    replace: true,
    restrict: 'E',
    link: function(scope, element, attrs) {

    },
    template: '<div>' +
      '<div>{{label}}</div>' +
      '<button ng-click="callback()">Click me!</button>' +
      '</div>'
  };
});
```

We want to test two things here. The first thing to test is that the label gets properly passed to the first `div`'s content, and the second is that something happens when the button gets clicked. It's worth saying that the actual rendering of the directive belongs slightly more to end-to-end and functional testing, but we want to include it as much as possible in our unit tests simply for the sake of failing fast. Besides, working with test-driven development is easier with unit tests than with higher-level tests, such as functional, integration and end-to-end tests.

```

describe('directives', function() {

  beforeEach(module('myDirectives'));

  var element;
  var outerScope;
  var innerScope;

  beforeEach(inject(function($rootScope, $compile) {
    element = angular.element('<super-button label="myLabel" on-click="myCallback()"></su

    outerScope = $rootScope;
    $compile(element)(outerScope);

    innerScope = element.isolateScope();

    outerScope.$digest();
  }));

  describe('label', function() {
    beforeEach(function() {
      outerScope.$apply(function() {
        outerScope.myLabel = "Hello world.";
      });
    })

    it('should be rendered', function() {
      expect(element[0].children[0].innerHTML).toEqual('Hello world.');
```

This example has something important. We saw that unit tests make refactoring easy as pie, but we didn't

see how exactly. Here, we are testing that when a click happens on the button, the function passed as the `on-click` attribute is called. If we take a closer look at the directive's code, we will see that this function gets locally renamed to `callback`. It's published under this name on the directive's isolated scope. We could write the following test, then:

```
describe('click callback', function() {
  var mySpy;

  beforeEach(function() {
    mySpy = sinon.spy();
    innerScope.callback = mySpy;
  });

  describe('when the directive is clicked', function() {
    beforeEach(function() {
      var event = document.createEvent("MouseEvent");
      event.initMouseEvent("click", true, true);
      element[0].children[1].dispatchEvent(event);
    });

    it('should be called', function() {
      expect(mySpy.callCount).to.equal(1);
    });
  });
});
```

And it would work, too. But then we wouldn't be testing the external aspect of our directive. If we were to forget to add the proper key to the directive's `scope` definition, then no test would stop us. Besides, we actually don't care whether the directive renames the callback or calls it through another method (and if we do, then it will have to be tested elsewhere anyway).

PROVIDERS

This is the toughest of our little series. What is a provider exactly? It's AngularJS' own way of wiring things together before the application starts. A provider also has a factory facet — in fact, you probably know the `$routeProvider` and its little brother, the `$route` factory. Let's write our own provider and its factory and then test them!

```

angular.module('myProviders', [])

.provider('coffeeMaker', function() {
  var useFrenchPress = false;
  this.useFrenchPress = function(value) {
    if (value !== undefined) {
      useFrenchPress = !!value;
    }

    return useFrenchPress;
  };

  this.$get = function () {
    return {
      brew: function() {
        return useFrenchPress ? 'Le café.': 'A coffee.';
      }
    };
  };
});

```

There's nothing fancy in this super-useful provider, which defines a flag and its accessor method. We can see the config part and the factory part (which is returned by the `$get` method). I won't go over the provider's whole implementation and use cases, but I encourage you to look at [AngularJS' official documentation about providers](#)²¹.

To test this provider, we could test the config part on the one hand and the factory part on the other. This wouldn't be representative of the way a provider is generally used, though. Let's think about the way that we use providers. First, we do some configuration; then, we use the provider's factory in some other objects or services. We can see in our `coffeeMaker` that its behavior depends on the `useFrenchPress` flag. This is how we will proceed. First, we will set this flag, and then we'll play with the factory to see whether it behaves accordingly.

```

describe('coffee maker provider', function() {
  var coffeeProvider = undefined;

  beforeEach(function() {
    // Here we create a fake module just to intercept and store the provider
    // when it's injected, i.e. during the config phase.
    angular.module('dummyModule', function() {})
      .config(['coffeeMakerProvider', function(coffeeMakerProvider) {
        coffeeProvider = coffeeMakerProvider;
      }]);

    module('myProviders', 'dummyModule');

    // This actually triggers the injection into dummyModule
    inject(function(){});
  });

  describe('with french press', function() {
    beforeEach(function() {
      coffeeProvider.useFrenchPress(true);
    });

    it('should remember the value', function() {
      expect(coffeeProvider.useFrenchPress()).to.equal(true);
    });

    it('should make some coffee', inject(function(coffeeMaker) {
      expect(coffeeMaker.brew()).to.equal('Le café.');
```

```

    }));
  });
});

```

```

describe('without french press', function() {
  beforeEach(function() {
    coffeeProvider.useFrenchPress(false);
  });

```

```

    it('should remember the value', function() {
      expect(coffeeProvider.useFrenchPress()).to.equal(false);
    });

```

```

    it('should make some coffee', inject(function(coffeeMaker) {
      expect(coffeeMaker.brew()).to.equal('A coffee.');
```

```

    }));
  });
});

```

HTTP REQUESTS

HTTP requests are not exactly on the same level as providers or controllers. They are still an essential part of unit testing, though. If you do not have a single HTTP request in your entire app, then you can skip this section, you lucky fellow.

Roughly, HTTP requests act like inputs and outputs at any of your application's level. In a RESTfully designed system, GET requests give data to the app, and PUT, POST and DELETE methods take some.

That is what we want to test, and luckily AngularJS makes that easy.

Let's take our factory example and add a `POST` request to it:

```
angular.module('factories_2', [])
.factory('chimp', ['$http', function($http) {
  return {
    sendMessage: function() {
      $http.post('http://chimps.org/messages', {message: 'Ook.'});
    }
  };
}]);
```

We obviously do not want to test this on the actual server, nor do we want to monkey-patch the `XMLHttpRequest` constructor. That is where `$httpBackend` enters the game.

```
describe('http', function() {

  beforeEach(module('factories_2'));

  var chimp;
  var $httpBackend;

  beforeEach(inject(function(_chimp_, _$httpBackend_) {
    chimp = _chimp_;
    $httpBackend = _$httpBackend_;
  }));

  describe('when sending a message', function() {
    beforeEach(function() {
      $httpBackend.expectPOST('http://chimps.org/messages', {message: 'Ook.'})
        .respond(200, {message: 'Ook.', id: 0});

      chimp.sendMessage();
      $httpBackend.flush();
    });

    it('should send an HTTP POST request', function() {
      $httpBackend.verifyNoOutstandingExpectation();
      $httpBackend.verifyNoOutstandingRequest();
    });
  });
});
```

You can see that we've defined which calls should be issued to the fake server and how to respond to them before doing anything else. This is useful and enables us to test our app's response to different requests' responses (for example, how does the application behave when the login request returns a 404?). This particular example simulates a standard `POST` response.

The two other lines of the `beforeEach` block are the function call and a newcomer, `$httpBackend.flush()`. The fake server does not immediately answer each request; instead, it lets you check any intermediary

state that you may have configured. It waits for you to explicitly tell it to respond to any pending request it might have received.

The test itself has two methods calls on the fake server (`verifyNoOutstandingExpectation` and `verifyNoOutstandingRequest`). AngularJS' `$httpBackend` does not enforce strict equality between what it expects and what it actually receives unless you've told it to do so. You can regard these lines as two expectations, one of the number of pending requests and the other of the number of pending expectations.

ngMock Module

The [ngMock module](#)²² contains various utilities to help you smooth over JavaScript and AngularJS' specifics.

\$TIMEOUT, \$LOG AND THE OTHERS

Using AngularJS' injectable dependencies is better than accessing global objects such as `console` or `window`. Let's consider `console` calls. They are outputs just like HTTP requests and might actually matter if you are implementing an API for which some errors must be logged. To test them, you can either monkey-patch a global object — yikes! — or use AngularJS' nice injectable.

The `$timeout` dependency also provides a very convenient `flush()` method, just like `$httpBackend`. If we create a factory that provides a way to briefly set a flag to `true` and then restore it to its original value, then the proper way to test it's to use `$timeout`.

```
angular.module('timeouts', [])

.factory('waiter', ['$timeout', function($timeout) {
  return {
    brieflySetSomethingToTrue: function(target, property) {
      var oldValue = target[property];

      target[property] = true;

      $timeout(function() {
        target[property] = oldValue;
      }, 100);
    }
  };
}]);
```

And the test will look like this:

```

describe('timeouts', function() {

  beforeEach(module('timeouts'));

  var waiter;
  var $timeout;

  beforeEach(inject(function(_waiter_, _$timeout_) {
    waiter = _waiter_;
    $timeout = _$timeout_;
  }));

  describe('brieflySetSomethingToTrue method', function() {
    var anyObject;

    beforeEach(function() {
      anyObject = {foo: 42};
      waiter.brieflySetSomethingToTrue(anyObject, 'foo');
    });

    it('should briefly set something to true', function() {
      expect(anyObject.foo).toEqual(true);
      $timeout.flush();
      expect(anyObject.foo).toEqual(42);
    });
  });
});

```

Notice how we're checking the intermediary state and then `flush()` 'ing the timeout.

MODULE() AND INJECT()

The `module()`²³ and `inject()`²⁴ functions help to retrieve modules and dependencies during tests. The former enables you to retrieve a module, while the latter creates an instance of `$injector`, which will resolve references.

```

it('should say Ook.', inject(function($log) {
  sinon.stub($log, 'warn', function() {});

  chimp.ook();

  expect($log.warn.callCount).toEqual(1);
  expect($log.warn.args[0][0]).toEqual('Ook.');
}));

```

In this test case, we're wrapping our test case function in an `inject` call. This call will create an `$injector` instance and resolve any dependencies declared in the test case function's arguments.

DEPENDENCY INJECTION MADE EASY

One last trick is to ask for dependencies using underscores around the name of what we are asking for.

The point of this is to assign a local variable that has the same name as the dependencies. Indeed, the `$injector` used in our tests will remove surrounding underscores if any are found. [StackOverflow has a comment²⁵](#) on this.

Conclusion

Unit testing in AngularJS applications follows a fractal design. It tests units of code. It freezes a unit's behavior by providing a way to automatically check its response to a given input. Note that unit tests do not replace good coding. AngularJS' documentation is pretty clear on this point: "Angular is written with testability in mind, but it still requires that you do the right thing."

Getting started with writing unit tests — and coding in test-driven development — is hard. However, the benefits will soon show up if you're willing to fully test your application, especially during refactoring operations.

Tests also work well with agile methods. User stories are almost tests; they're just not actual code (although some approaches, such as "[design by contract](#)²⁶", minimize this difference).

FURTHER RESOURCES

- "[The Pragmatic Programmer: From Journeyman to Master](#)²⁷", Andrew Hunt and David Thomas
- [AngularJS' documentation on unit testing](#)²⁸
- All examples can be found in the [GitHub repository](#)²⁹

(*al, ml*)

FOOTNOTES

¹ <https://angularjs.org>

² <http://karma-runner.github.io>

³ <http://visionmedia.github.io/mocha/>

⁴ <http://chaijs.com>

⁵ <http://sinonjs.org>

⁶ <https://pragprog.com/the-pragmatic-programmer>

⁷ <http://www.smashingmagazine.com/wp-content/uploads/2014/09/01-bricks-opt.png>

[8 http://www.smashingmagazine.com/wp-content/uploads/2014/09/01-bricks-opt.png](http://www.smashingmagazine.com/wp-content/uploads/2014/09/01-bricks-opt.png)
[9 http://www.smashingmagazine.com/wp-content/uploads/2014/09/02-blackbox-opt.png](http://www.smashingmagazine.com/wp-content/uploads/2014/09/02-blackbox-opt.png)
[10 http://www.smashingmagazine.com/wp-content/uploads/2014/09/02-blackbox-opt.png](http://www.smashingmagazine.com/wp-content/uploads/2014/09/02-blackbox-opt.png)
[11 http://karma-runner.github.io](http://karma-runner.github.io)
[12 http://www.smashingmagazine.com/wp-content/uploads/2014/09/03-karma-success-opt.png](http://www.smashingmagazine.com/wp-content/uploads/2014/09/03-karma-success-opt.png)
[13 http://www.smashingmagazine.com/wp-content/uploads/2014/09/03-karma-success-opt.png](http://www.smashingmagazine.com/wp-content/uploads/2014/09/03-karma-success-opt.png)
[14 https://github.com/lorem--ipsum/smashing-article](https://github.com/lorem--ipsum/smashing-article)
[15 http://karma-runner.github.io/0.8/config/configuration-file.html](http://karma-runner.github.io/0.8/config/configuration-file.html)
[16 https://docs.angularjs.org/api/ngMock](https://docs.angularjs.org/api/ngMock)
[17 http://visionmedia.github.io/mocha/](http://visionmedia.github.io/mocha/)
[18 http://chaijs.com](http://chaijs.com)
[19 http://chaijs.com/guide/styles/](http://chaijs.com/guide/styles/)
[20 http://sinonjs.org](http://sinonjs.org)
[21 https://docs.angularjs.org/guide/providers](https://docs.angularjs.org/guide/providers)
[22 https://docs.angularjs.org/api/ngMock](https://docs.angularjs.org/api/ngMock)
[23 https://docs.angularjs.org/api/ngMock/function/angular.mock.module](https://docs.angularjs.org/api/ngMock/function/angular.mock.module)
[24 https://docs.angularjs.org/api/ngMock/function/angular.mock.inject](https://docs.angularjs.org/api/ngMock/function/angular.mock.inject)
[25 http://stackoverflow.com/a/15318137/863119](http://stackoverflow.com/a/15318137/863119)
[26 http://en.wikipedia.org/wiki/Design_by_contract](http://en.wikipedia.org/wiki/Design_by_contract)
[27 https://pragprog.com/the-pragmatic-programmer](https://pragprog.com/the-pragmatic-programmer)
[28 https://docs.angularjs.org/guide/unit-testing](https://docs.angularjs.org/guide/unit-testing)
[29 https://github.com/lorem--ipsum/smashing-article](https://github.com/lorem--ipsum/smashing-article)



Sébastien Fragnaud

Sébastien Fragnaud is a front-end engineer at Metamarkets, a company specialized in real-time analytics for the advertising market. He loves JavaScript and is an active contributor to the AngularJS open-source ecosystem.

With a commitment to quality content for the design community.

Founded by Vitaly Friedman and Sven Lennartz. 2006-2014.

Made in Germany. 🇩🇪 – – <http://www.smashingmagazine.com>