AngularJS was designed with testing in mind. The source code of the framework is tested really well and any code written using the framework is testable too. The built-in dependency injection mechanism makes every component written in AngularJS testable. Code in an AngularJS application can be unit tested using any JavaScript testing framework out there. The most widely used framework to test AngularJS code is Jasmine. All example snippets in this article are written using Jasmine. If you are using any other test framework in your Angular project, you can still apply the ideas discussed in this article.

This article assumes that you already have some experience with unit testing and testing AngularJS code. You need not be an expert in testing. If you have a basic understanding of testing and can write some simple test cases for an AngularJS application, you can continue reading the article.

**Role of Mocking in Unit Tests**

The job of every unit test is to test the functionality of a piece of code in isolation. Isolating the system under test can be challenging at times as dependencies may come from different sets of sources and we need to fully understand the responsibilities of the object to be mocked.

Mocking is difficult in non-statically typed languages like JavaScript, as it is not easy to understand structure of the object to be mocked. At the same time, it also provides a flexibility of mocking only part of the object that is currently in use by the system under test and ignore the rest.

**Mocking in AngularJS Tests**

As one of the primary goals of AngularJS is testability, the core team walked that extra mile to make testing easier and provided us with a set of mocks in the angular-mocks module. This module consists of mocks around a set of AngularJS services (viz, $http, $timeout, $animate, etc) that are widely used in any AngularJS application. This module reduces a lot of time for developers writing tests.

While writing tests for real business applications, these mocks help a lot. At the same time they are not enough for testing the entire application. We need to mock any dependency that is in the framework but not mocked – a dependency that came from a third party plugin, a global object, or a dependency created in the application. This article will cover some tips on mocking AngularJS dependencies.

**Mocking Services**

A service is the most common type of dependency in AngularJS applications. As you are already aware, service is an overloaded term in AngularJS. It may refer to a service, factory, value, constant, or provider. We will discuss providers in the next section. A service can be mocked in one of the following ways:

* Getting an instance of the actual service using an inject block and spying methods of the service.
* Implementing a mock service using $provide.

I am not a fan of the first approach as it may lead to calling actual implementation of the service methods. We will use the second approach to mock the following service:

angular.module('sampleServices', [])

.service('util', function() {

this.isNumber = function(num) {

return !isNaN(num);

};

this.isDate = function(date) {

return (date instanceof Date);

};

});

The following snippet creates a mock of the above service:

module(function($provide) {

$provide.service('util', function() {

this.isNumber = //此处应改为and.callFake(…) jasmine.createSpy('isNumber').andCallFake(function(num) {

//a fake implementation

});

this.isDate = jasmine.createSpy('isDate').andCallFake(function(num) {

//a fake implementation

});

});

});

//Getting reference of the mocked service

var mockUtilSvc;

inject(function(util) {

mockUtilSvc = util;

});

Though the above example uses Jasmine to create spies, you can replace it with an equivalent implementation using Sinon.js.

It is always good to create all mocks after loading all the modules that are required for the tests. Otherwise, if a service is defined in one of the modules loaded, the mock implementation is overridden by the actual implementation.

Constants, factories, and values can be mocked using *$provide.constant*, *$provide.factory*, and *$provide.value*, respectively.

**Mocking Providers**

Mocking providers is similar to mocking services. All rules that one has to follow while writing providers have to be followed while mocking them as well. Consider the following provider:

angular.module('mockingProviders',[])

.provider('sample', function() {

var registeredVals = [];

this.register = function(val) {

registeredVals.push(val);

};

this.$get = function() {

function getRegisteredVals() {

return registeredVals;

}

return {

getRegisteredVals: getRegisteredVals

};

};

});

The following snippet creates a mock for the above provider:

module(function($provide) {

$provide.provider('sample', function() {

this.register = jasmine.createSpy('register');

this.$get = function() {

var getRegisteredVals = jasmine.createSpy('getRegisteredVals');

return {

getRegisteredVals: getRegisteredVals

};

};

});

});

//Getting reference of the provider

var sampleProviderObj;

module(function(sampleProvider) {

sampleProviderObj = sampleProvider;

});

The difference between getting reference of providers and other singletons is, providers are not available in *inject()* lock as the providers are converted into factories by this time. We can get their objects using a *module()* block.

In the case of defining providers, an implementation of the *$get* method is mandatory in tests as well. If you don’t need the functionality defined in *$get* function in the test file, you can assign it to an empty function.

**Mocking Modules**

If the module to be loaded in the test file needs a bunch of other modules, the module under test can’t be loaded unless all the required modules are loaded. Loading all of these modules sometimes leads to bad tests as some of the actual service methods may get called from the tests. To avoid these difficulties, we can create dummy modules to get the module under test to be loaded.

For example, assume the following code represents a module with a sample service added to it:

angular.module('first', ['second', 'third'])

//util and storage are defined in second and third respectively

.service('sampleSvc', function(utilSvc, storageSvc) {

//Service implementation

});

The following code is the beforeEach block in the test file of the sample service:

beforeEach(function() {

angular.module('second',[]);

angular.module('third',[]);

module('first');

module(function($provide) {

$provide.service('utilSvc', function() {

// Mocking utilSvc

});

$provide.service('storageSvc', function() {

// Mocking storageSvc

});

});

});

Alternatively, we can add the mock implementations of the services to the dummy modules defined above as well.

**Mocking Methods Returning Promises**

It can be tough to write an end to end Angular application without using promises. It becomes a challenge to test a piece of code that depends on a method returning a promise. A plain Jasmine spy will lead to failure of some test cases as the function under test would expect an object with the structure of an actual promise.

Asynchronous methods can be mocked with another asynchronous method that returns a promise with static values. Consider the following factory:

angular.module('moduleUsingPromise', [])

.factory('dataSvc', function(dataSourceSvc, $q) {

function getData() {

var deferred = $q.defer();

dataSourceSvc.getAllItems().then(function(data) {

deferred.resolve(data);

}, function(error) {

deferred.reject(error);

});

return deferred.promise;

}

return {

getData: getData

};

});

We will test the *getData()* function in the above factory. As we see, it depends on the method *getAllItems()* of the service *dataSourceSvc*. We need to mock the service and the method before testing the functionality of the *getData()* method.

The $q service has the methods *when()* and *reject()* that allow resolving or rejecting a promise with static values. These methods come in handy in tests that mock a method returning a promise. The following snippet mocks the *dataSourceSvc* factory:

module(function($provide) {

$provide.factory('dataSourceSvc', function($q) {

var getAllItems = jasmine.createSpy('getAllItems').andCallFake(function() {

var items = [];

if (passPromise) {

return $q.when(items);

}

else {

return $q.reject('something went wrong');

}

});

return {

getAllItems: getAllItems

};

});

});

A $q promise finishes its action after the next digest cycle. The digest cycle keeps running in actual application, but not in tests. So, we need to manually invoke *$rootScope.$digest()* in order to force execution of the promise. The following snippet shows a sample test:

it('should resolve promise', function() {

passPromise = true;

var items;

dataSvcObj.getData().then(function(data) {

items=data;

});

rootScope.$digest();

expect(mockDataSourceSvc.getAllItems).toHaveBeenCalled();

expect(items).toEqual([]);

});

**Mocking Global Objects**

Global objects come from the following sources:

1. Objects that are part of global ‘window’ object (e.g, localStorage, indexedDb, Math, etc).
2. Objects created by a third party library like jQuery, underscore, moment, breeze or any other library.

By default, global objects can’t be mocked. We need to follow certain steps to make them mockable.

We may not want to mock the utility objects such as the functions of the Math object or *\_* (created by the Underscore library) as their operations don’t perform any business logic, don’t manipulate UI, and don’t talk to a data source. But, objects like $.ajax, localStorage, WebSockets, breeze, and toastr have to be mocked. Because, if not mocked these objects would perform their actual operation when the unit tests are executed and it may lead to some unnecessary UI updates, network calls, and sometimes errors in the test code.

Every piece of code written in Angular is testable because of dependency injection. DI allows us to pass any object that follows the shim of the actual object to just make the code under test not break when it is executed. Global objects can be mocked if they can be injected. There are two ways to make the global object injectable:

1. Inject $window to the service/controller that needs global object and access the global object through $window. For example, the following service uses localStorage through $window:

angular.module('someModule').service('storageSvc', function($window) {

this.storeValue = function(key, value) {

$window.localStorage.setItem(key, value);

};

});

1. Create a value or constant using the global object and inject it wherever needed. For example, the following code is a constant for toastr:

angular.module('globalObjects',[])

.constant('toastr', toastr);

I prefer using a constant over value to wrap the global objects as constants can be injected into config blocks or providers and constants cannot be decorated.

The following snippet shows mocking of localStorage and toastr:

beforeEach(function() {

module(function($provide) {

$provide.constant('toastr', {

warning: jasmine.createSpy('warning'),

error: jasmine.createSpy('error')

});

});

inject(function($window) {

window = $window;

spyOn(window.localStorage, 'getItem');

spyOn(window.localStorage, 'setItem');

});

});

**Conclusion**

Mocking is one of the important parts of writing unit tests in any language. As we saw, dependency injection plays a major role in testing and mocking. Code has to be organized in a way to make the functionality easily testable. This article lists mocking most common set of objects while testing AngularJS apps. The code associated with this article is available for download from [GitHub](https://github.com/jsprodotcom/source/blob/master/TestingTips-Mocking.zip).