Mocking Dependencies in AngularJS Tests

By Ravi (https://www.sitepoint.com/author/rkiran/) October 30, 2014

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 = 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).

Was this helpful?

More: angular (https://www.sitepoint.com/tag/angular/), mocking (https://www.sitepoint.com/tag/mocking/), unit testing (https://www.sitepoint.com/tag/unit-testing/)

(http ://

Meet the author
Ravi (https://www.sitepoint.com/author/rkiran/) (https://twitter.com/sravi_kiran) (https://plus.google.com/+RaviKiran) (https://www.linkedin.com/profile/view?id=85014890) (https://github.com/sravikiran)

Rabi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like Angular JS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger (http://sravi-kiran.blogspot.com), an author at SitePoint and at DotNetCurry (http://www.dotnetcurry.com/author/ravi-kiran). He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.

22 Comments **SitePoint** 1 **Login**

Recommend 79 Share Sort by Best

Join the discussion…

David Tromblee • 2 years ago

For reference to any newcomers to this post, the jasmine framework has modified their functions slightly in versions >=2.0 (e.g. .andCallFake() is now .and.callFake()), so make sure to update those functions appropriately if copying the code.

No fault of Ravi of course (frameworks change), and might I say this is the BEST example set of mocking that I have found. Every other example set I've found so far doesn't take the time to directly override their services/factories/etc, and as Ravi pointed out, that could lead to the original firing and leading to weird results.

3 • Reply • Share ›

Frank van Wijk • 2 years ago

Ravi, this is an excellent post. You hit the nail about how to create a mock the correct way. Many blogs still mock every service by mocking the underlying $httpBackend and that's just it.

I've been mocking a lot of Angular services in some large projects (5000+ unit tests) and one of the drawbacks of mocking stuff is that it will take a lot of boilerplate code. Therefore I created a utility library that can mock dependencies automatically or just replaces the boilerplate code by a simple DSL.
You can check it out at Github: http://www.github.com/fvanw...

I hope you like it. Comments are welcome.

3 • Reply • Share ›

> **DannyGoncalves** Frank van Wijk • 2 years ago
>
> Great library! thanks a lot buddy
>
> • Reply • Share ›

blackfiredragon • 2 years ago

Ravi, this is an excellent article thank you very much. Building the different types of mocks for services, providers, controllers and directives can be daunting without examples such as this; much appreciated!

1 • Reply • Share ›

tennisgent • 2 years ago

Another option to help make mocking dependencies easier in Angular and Jasmine is to use QuickMock. It can be found on GitHub and allows you to create simple mocks in a reusable way. You can clone it from GitHub via the link below. The README is pretty self explanatory, but hopefully it might help others in the future. It does a lot of the above boilerplate for you, and allows you to spend less time setting up your tests and more time writing the actually test cases.

https://github.com/tennisge...

1 • Reply • Share ›

shon • a year ago

Am i missing something? If you're using Jasmine, don't you need to describe, beforeEach, and it methods? Not sure what a noob is supposed to do when you start your test with "module". Don't find this very helpful for learning to unit test Angularjs. Everyone says its better to unit test, and spend the time testing, but if you can get the project done in the time of trying to figure out "someones" idea of how to unit test, kind defeats the whole point... I'm not against testing, but when you spend more than a day trying to understand something so loosely put together like a client side language, how could one possibly figure out the best ways to unit test if there aren't any simple examples out there. I'm not a noob to programming or TDD (server side mostly), but when it comes to Angularjs and unit testing, i've never had more a difficult time understanding something... Hope it gets easier.

• Reply • Share ›

fayjai • a year ago

minor typo: should be jasmine.createSpy('isNumber').and.callFake(function(num) {}) instead of andCallFake.

• Reply • Share ›

> **Ravi Kiran** fayjai • a year ago
>
> It is not a typo. API of Jasmine was updated in a later version. Usage of the API in this post works well with the version referred in the demo.
>
> • Reply • Share ›
>
> > **fayjai** Ravi Kiran • a year ago
> >
> > Whoops sorry about that! I was trying out your examples and didn't realize which version of Jasmine to use :) Thanks again for your post!
> >
> > • Reply • Share ›

Jolo Bada • a year ago
how was dataSvcObj initialized and declared??
Reply • Share ›

Jolo Bada • a year ago
hey guys where was this variable initialize - dataSvcObj??
cant find it
Reply • Share ›

Ravi Kiran ➜ Jolo Bada • a year ago
Check promiseSpec.js file in the sample code (https://github.com/jsprodot..., you will find the answer
Reply • Share ›

NeverwinterMoon • 2 years ago
If you have something like this "angular.module('second',[]);" in your unit test and then you also have a separate spec (runs together with the first one) to test module 'second', the test mock module('second') will not test the actual code base as injector will fetch "angular.module('second',[]);" instead.
Reply • Share ›

Nico • 2 years ago
if (passPromise) {
return $q.when(items);
}
else {
return $q.reject('something went wrong');
}

if > return. Why else is needed here? Just use return like this!

if (passPromise) {
return $q.when(items);
}
return $q.reject('something went wrong');

Moreover, $rootScope.$digest() will fail if there is already a $digest in progress. You can prevent this error by using if (!$rootScope.$$phase) $rootScope.$digest().

angular.module('globalObjects',[])
.constant('toastr', toastr);

I like that :-)
Reply • Share ›

ajay • 2 years ago
Really good stuff keep on updating and posting new things for development .
Reply • Share ›

SquadWuschel • 2 years ago
ver cool article
Reply • Share ›

Joytas • 2 years ago
Hi Ravi, thanks for your great article! That's exactly what I've been searching for :)

What do you think about moving your service's mock declaration from beforeEach() method (specific only to your test suite) to another file? I think it would be convenient to have one declaration of a service's mock, which can be used in many tests. Especially if the service is often used as a dependency (e.g. authorization, common utils services).

I've combined together your tips with another approach ("Dailyjs: AngularJS: tests") that suggests creating mocked value as another service. In my solution I may have following files:

"commonServices.js" - original service (module('commonServices'), service 'util')
"commonServices.mock.js" - mocked service (module('commonServicesMock'), service 'util' with mocked values)
"myService.js" - service which depends on 'util' service (module('myService', ['commonServices']))
"myService.spec.js" - testing suite for 'myService'. Content:

beforeEach() {
module('myService'); //inject all needed services, including original 'util'
module('commonServicesMock'); //overwrite injected 'util' service with the mocked version
}

This solution works for me and I find it very convenient to have mock in one place, near the original service, so anytime original service is changed also mock can be easily changed.
What do you think about that? Maybe you can suggest some better solution? :)

(/) · Reply · Share ›

Get started, it's free

When you build in Bitbucket, you build with a purpose. Get the Git solution for professional teams, it's free.
(//srv.buysellads.com/ads/click/x/GTND4237C6YDE5QWCWA4YKQWF6YI42QMCYYIKZ3JCEAI4KJJC6SDPK7KC6BDLKQEF6YI6K3EHJNCLSIZ?segment=placement:sitepoint;)

Ravi Kiran ➦ Joytas • 2 years ago
Joytas,

I am glad that you found this article useful.
Your approach is correct. I have used this approach. This approach helps a lot in large apps where you would have a large number of dependencies in your components and you need to mock the same dependency at multiple places. In such case, creating a module of mocks saves a lot of time and helps in avoiding repeated code. But, it would be an overkill in relatively smaller apps.
· Reply · Share ›

Sandeep Pulikallu • 2 years ago
Hi Kiran, Very nice article.

As you have discussed sometimes code depend on some third party plugins etc... Mostly in directives... may be jQuery plugin etc... what can we do in such scenarios - mocking the same by creating a empty dummy object and adding spy's on required methods or any thing different can be done?
For example consider a directive which is dependent on charts.js library... what will be the good approach for mocking here?

Thank you for sharing this information.
· Reply · Share ›

Ravi Kiran ➦ Sandeep Pulikallu • 2 years ago
Sandeep,
There are a couple of ways to mock jQuery plugin methods. If you have jQuery loaded in your test environment, just create a spy for the method that the plugin adds. Otherwise, create a dummy jQuery object and add a spy method to it.
You can mock the object of chart.js using the technique I used for toastr in this article. But, it is going to be tricky as Chart's API is not as simple as toastr's API.
· Reply · Share ›

Srini Kusunam • 2 years ago
Nice article. Thanks for sharing.
· Reply · Share ›

Ravi Kiran ➦ Srini Kusunam • 2 years ago
Thanks Srini!
· Reply · Share ›

Subscribe Add Disqus to your site Add Disqus Add Privacy

LATEST THEMES ›
(/themes/)

PREMIUM THEME

SitePoint WordPress Restaurant Theme (https://www.sitepoint.com/wordpress-restaurant-theme/)

PREMIUM THEME

SitePoint WordPress Ecommerce Theme (https://www.sitepoint.com/wordpress-ecommerce-theme/)

PREMIUM THEME

LATEST COURSES (/premium/courses/)

PREMIUM COURSE

44m

Introducing TypeScript (https://www.sitepoint.com/premium/courses/introducing-typescript-2933)

PREMIUM COURSE

1h 3m

MS Bots (https://www.sitepoint.com/premium/courses/ms-bots-2939)

PREMIUM COURSE

3h 7m

Functional JavaScript Programming (https://www.sitepoint.com/premium/courses/functional-javascript-programming-2922)

LATEST BOOKS (/premium/books/)

PREMIUM BOOK

ECMAScript 2015: A SitePoint Anthology (https://www.sitepoint.com/premium/books/ecmascript-2015-a-sitepoint-anthology)

PREMIUM BOOK

Jump Start Git (https://www.sitepoint.com/premium/books/jump-start-git)

PREMIUM BOOK

Full Stack JavaScript Development with MEAN (https://www.sitepoint.com/premium/books/full-stack-javascript-development-with-mean)

Get the latest in JavaScript, once a week, for free.

Enter your email

Subscribe

(/)

About

Get started, it's free

When you build in Bitbucket, you build with a purpose. Get the Git solution for professional teams, it's free.

(//srv.buysellads.com/ads/click/x/GTND4237C6YDE5QWCWA4YKQWF6YI42QMCYYIKZ3JCEAI4KJJC6SDPK7KC6BDLKQEF6YI6K3EHJNCLSIZ?segment=placement:sitepoint;)

Our Story (/about-us/)
Advertise (/advertise/)
Press Room (/press/)
Reference (http://reference.sitepoint.com/css/)
Terms of Use (/legals/)
Privacy Policy (/legals/#privacy)
FAQ (https://sitepoint.zendesk.com/hc/en-us)
Contact Us (mailto:feedback@sitepoint.com)
Contribute (/write-for-us/)

Visit

SitePoint Home (/)
Themes (/themes/?utm_source=blog&utm_medium=footer)
Podcast (/versioning-show/)
Forums (https://www.sitepoint.com/community/)
Newsletters (/newsletter/)
Premium (/premium/)
References (/sass-reference/)
Versioning (https://www.sitepoint.com/versioning/)

Connect

(https://www.sitepoint.com/feed/) (/newsletter/) (https://www.facebook.com/sitepoint) (http://twitter.com/sitepointdotcom) (https://plus.google.com/+sitepoint)