```
import {
  Learning,
  JavaScript,
  AngularJS
} from 'bradoncode.com'
|
```


ngMock Fundamentals for AngularJS - Testing Controllers

An in-depth look at Unit Testing AngularJS Controllers with ngMock.

Bradley Braithwaite on June 5, 2015 on *javascript, angularjs, testing, ngmock*

In this post we take a deeper dive into unit testing AngularJS controllers. If you need an introduction into ngMock or the basics with controllers, you may wish to read How to Unit Test an AngularJS Controller first.

Back when we talked about the injector, we presented the following syntax for getting instances of angular objects for testing:

```
angular.mock.inject(function GetDependencies(categoryService) {
  service = categoryService;
});
```

Let's consider the following app as an example:

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

app.controller('ProductsController', function productsController($scope, ProductServ
        $scope.products = ProductService();
});

app.service('ProductService', function productService() {
        return function getProducts() {
                return [{ name: 'Chai' }, { name: 'Syrup' }];
        }
});
```

It would be logical to assume that we could do something similar to what we saw previously in order to get an instance of our controller for a unit test:

```
angular.mock.inject(function GetDependencies(ProductsController) {
        // this won't work!!
  controllerInstance = ProductsController;
});
```

But, it's not quite the same. We in fact need to get an instance of ngMock's $controller service, as follows:

```
var $controller;

beforeEach(inject(function(_$controller_) {
        $controller = _$controller_;
}));
```

NB the use of underscore wrapping is a convention in angular that allows us to use the same name for a variable.

Now that we have an instance of the *ngMock* controller service, we can use it to get a specific controller instance from our app via its name or by dynamically registering a new controller.

Using ngMock's $controller service

The controller service is a decorator for $controller with an additional bindings parameter, useful when testing controllers of directives that use bindToController.

It acts as decorator to angular's $controllerProvider. The $controller function accepts three arguments:

Param	Description
constructor	This can be either the name (string) of a controller, or a function that creates a new controller.
locals	This allows us to pass an object that maps by key names to the arguments of the controller constructor function e.g. $scope, ProductService.
bindings	This is optional, and allows us to pass in an object, where the values of the object e.g. properties or functions, will be bound to the controller's *this* binding.

We will see examples of each of these arguments in this post.

Getting a constructor instance by name

The most common scenario would be to get an instance of a controller by its name, as we demonstrate here:

```
it('should return products list on load', function () {
        var $scope = {};
        var productsController = $controller('ProductsController', { $scope: $scope
        expect($scope.products).toEqual([{ name: 'Tea' }, { name: 'Syrup' }]);
});
```

Here we show how the *$controller* service is used to locate a controller via its name, and we can also pass an object to represent the constructor arguments of the controller (the locals argument). We should also observe that the *ProductsController* has a dependency on *ProductService* which is resolved in the conventional angular way when the controller is invoked i.e. the *ProductService* code we set out in the example application will be used in this test.

Create controller instance via a function

It's also possible to create a controller in-line, by passing the controller function as an argument in place of a controller name. This could be useful when prototyping code. Here's an example:

```
it('should return products list on load', function () {
        var productsController = $controller(function inlineController($scope, Produ
                        $scope.products = ProductService();
        }, { $scope: $scope });

        expect($scope.products).toEqual([{ name: 'Chai' }, { name: 'Syrup' }]);
});
```

As we saw in the example that used the constructor name (string) we are also passing in a *locals* argument to represent the constructor arguments for the controller function.

Using Bindings

In this example, we make use of the 3rd optional argument called *bindings*. The type is a JavaScript object and will be attached to the controller's *this* binding for the test.

We will build upon the controller used in the previous example. Here we will pass the object *{ data: bindings }* for the *bindings* argument, which we will be able to access from within the controller function via *this.data*. Here's the code:

```
it('should return products list on load', function () {
        var bindings = { foo: 'bar' };

        var productsController = $controller(function inlineController($scope, Produ
                // not that we access the data object via this.
                expect(this.data).toEqual(bindings);

                $scope.products = ProductService();

        }, { $scope: $scope },  { data: bindings });

        expect($scope.products).toEqual([{ name: 'Chai' }, { name: 'Syrup' }]);
});
```

Primarily, this is used for isolate scope bindings in $compile. This is especially useful when unit testing directives that are linked to a controller, but we will leave discussing the internals of unit testing directives for a future post.

Mocking

The tests we have demonstrated so far, have all used the implementation of the *ProductService* defined in the example application we set out at the start of the post. Unit tests should test code in isolation, therefore we don't wish to use an implementation of the *ProductService* when testing the controller. So, how do we mock the service instance?

1. Mocking by Argument

Using my own unofficial terminology, I call this "mocking by argument", which involves passing in an instance of the thing we wish to mock via the *locals* argument. We have to use the same name for the object key as the argument, and this takes precedence when the injector finds the constructor arguments for the controller. Here's an example:

```
it('should return products list on load', function () {
        var $scope = {};
        var mockService = function ProductService() {
                return [{ name: 'Tea' }, { name: 'Syrup' }];
        }

        var productsController = $controller('ProductsController', { $scope: $scope,

        expect($scope.products).toEqual([{ name: 'Tea' }, { name: 'Syrup' }]);
});
```

I prefer to use this approach when mocking, as I find it more declarative than the alternative methods.

2. Using mock.module

Our test would look as before, but we would be required to register a new version of the *ProductService* via the *mock.module* function:

```
describe('mock.module mocking of service', function () {

        // we need to register our alternative version of ProductService, before we
        beforeEach(angular.mock.module(function($provide) {
                $provide.service('ProductService', function mockService() {
                        return function mockGetProducts() {
                                return [{ name: 'Tea' }, { name: 'Syrup' }];
                        }
                });
        }));

        beforeEach(inject(function(_$controller_) {
                $controller = _$controller_;
                $scope = {};
        }));

        it('should return products list on load', function () {
                var productsController = $controller('ProductsController', { $scope:
                expect($scope.products).toEqual([{ name: 'Tea' }, { name: 'Syrup' }]
        });

});
```

In this example we rely on angular's injector to find the ProductService instance, but we register the version we wish to be used via the *mock.module* function. Our second version of

ProductService will override the initial version registered by the example application code.

A Deeper dive into the $controller service

As with some of the earlier posts, now we will take a deeper dive into the internals of ngMock. By taking a look at the controller service, we can also understand a little more about angular's injector.

The controller service from angular is wrapped with ngMocks's own version, using the decorator pattern. If you dig into the source code of ngMock, you will see the following line of code that registers the $controller service, with the ControllerDecorator:

```
$provide.decorator('$controller', angular.mock.$ControllerDecorator);
```

The decorator sits in front of the $controllerProvider. The functionality it adds is to either call the controllerProvider as is, or to create a new instance in the presence of an object for the *bindings* (the argument is called *later*, in the source code), with the properties of the *bindings* object to be used for the controller instances' *this* binding:

```
angular.mock.$ControllerDecorator = ['$delegate', function($delegate) {
  return function(expression, locals, later, ident) {
    if (later && typeof later === 'object') {
      var create = $delegate(expression, locals, true, ident);
      angular.extend(create.instance, later);
      return create();
    }
    return $delegate(expression, locals, later, ident);
  };
}];
```

The *$delegate* function we see being called in this code, is an instance of angular's $controllerProvider.

We can also interact with the $controllerProvider directly, via ngMock's module function should we choose to. Here's an example of how it could be used:

```
module(function($controllerProvider) {
        // we could also access the $controllerProvider.allowGlobals() function,
        // which allows us to register a controller on the window object.
        $controllerProvider.register('ProductsController', function() {
                // logic of the controller...
        });
});
```

But keep in mind, that we cannot access the *$controllerProvider* via the *inject* function. That is to say that we can't do this:

```
beforeEach(inject(function($controllerProvider) {

        // do something interesting with the provider
        console.log($controllerProvider);

}));
```

If you try to do this, you will see the error: *Error: [$injector:unpr] Unknown provider: $controllerProviderProvider <- $controllerProvider*. Notice that the injector is trying to find "controllerProviderProvider"? The additional "Provider" string is appended onto the name!

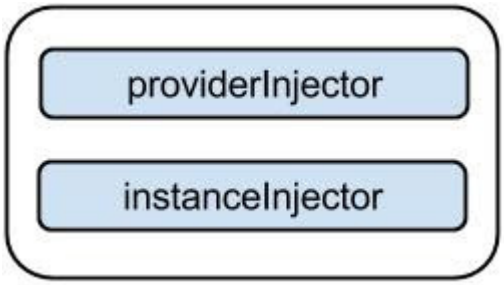
The inverse is also true, we cannot access an instance of the $controller service via the module function:

```
module(function($controller) {

        // do something interesting with the service
        console.log($controller);

});
```

In this case, we see the error: *Unknown provider: $controller*.

So what's happening? Isn't $controllerProvider and $controller the same thing? The key distinction between the two methods when using ngMock's module and inject functions, is that when we are using the module function, the injector has not yet been called. This is important, since it means that we can receive and uninstantiated $controllerProvider object. This means that we can access the register and allowGlobals functions that we saw in the code snippet.

As soon as mock.inject is called, something interesting happens. Before we talk more about the injector, let's remind ourselves that is has two main components:



Angular's injector is called, and an instance of $controllerProvider is created and saved within a "provider cache" by the injector. During this initialisation step for the injector, it iterates through any

objects/functions registered via a module and invokes them, also passing the necessary function arguments. During this process, the providerInjector is called directly:

```
providerInjector.invoke(module);
```

Once the injector is initialised, any subsequent calls to the inject function are handled by an instanceInjector and we cannot make direct calls to the providerInjector. The instanceInjector does make a call to the providerInjector, but it's hard coded to append the string "Provider" to the end of a service name.

This explains why we saw the error relating to *$controllerProviderProvider* when we asked the injector for *$controllerProvider*, and of course it also means that when we ask the injector for an instance of *$controller*, the instanceInjector makes a call to the providerInjector asking for *$controllerProvider*, which in turn will return the instance from its own cache.

Why would it be implemented this way? It's primarily for information hiding. Once the injector is created, we shouldn't be able to register new services. This enforces the convention that we already know, what we can register our services via modules and get instances via the injector.

In summary, we can configure the providerInjector via the module interface, but it's only read-only when using the injector.

Example Test Code

Full code example of the tests used in this post via a Github Gist.

CONNECT WITH BRADLEY

Bradley Braithwaite is a software engineer who works for the search engine duckduckgo.com. He is a published author at pluralsight.com. He writes about software development practices, JavaScript, AngularJS and Node.js via his website bradoncode.com. Find out more about Brad. Find him via:

You might also like:

5 Comments **bradoncode** **1 Login**

Recommend **Share** **Sort by Best**

Join the discussion…

Ankit Dwivedi • 7 months ago

Hi,

Could you please provide more insight on why...

```
beforeEach(module(function($controller) {
console.log($controller);

}));
```

will not work

Reply • Share ›

Bradley Braithwaite Mod → Ankit Dwivedi • 7 months ago

In this case, you need to use inject and not module:

```
beforeEach(inject(function(_$controller_) {
$controller = _$controller_;
console.log($controller);
}));
```

Reply • Share ›

Santy • 10 months ago

Thank you for nice explanation!

Reply • Share ›

Ruslan Vasylenko • 10 months ago

Hello, Brad!
At first thanks a lot for so useful topics and detailed explanation! It helps a lot!

I try to mock service by argument like in your example, and it seems that I do everything similary. But in my case Karma gives **TypeError: adminService.getHeader is not a function.** And I can't fix this. Can you see, what the problem is, please?

My controller:

```
angular.module("app.admin").controller("AdminsController", AdminsController);
AdminsController.$inject = ["adminService"];
function AdminsController(adminService) {
        var vm = this;
        vm.headElements = adminService.getHeader();
};
```

My service:

see more

Reply • Share ›

Bradley Braithwaite Mod ➚ Ruslan Vasylenko • 10 months ago

Sorry for the delay in responding. Did you fix this?

You need to pass:

```
var mockService = {
                getHeader: function() {
                        return [" ", "Логін", "E-mail", "Останній вхід", "Візити
                }
};
```

Reply • Share ›