```
1 import {
2   Learning,
3   JavaScript,
4   AngularJS
5 } from 'bradoncode.com'
6
```


ngMock Fundamentals for AngularJS - Understanding Inject

Final part of a two part series taking a deeper dive into the two key concepts of ngMock for Unit Testing - Module and Inject.

Bradley Braithwaite on May 27, 2015 on *javascript, angularjs, testing, ngmock*

This is the final post of a two part series that takes a deeper dive into two main concepts from *ngMock*:

- angular.mock.module
- angular.mock.inject

In the previous post, we looked at the *angular.mock.module*. In this post we will take a look into the *angular.mock.inject* funtionality and detail how this works in conjunction with *angular.mock.module*, and the Angular framework itself.

In the last post we established how we could load modules into unit tests via the *mgMock*

functionality. Next we need to fill in the blanks as to what we can assert in the tests and how we can go about obtaining instances of the things we wish to test. As a reminder, here's an example test from the last post:

```
describe('string alias arg', function () {
  it('should load module with string alias', function () {
    angular.mock.module('products');

    // TBC - we will build upon this!
    expect('what?').toBe('what?');
  });
});
```

To fill in the blanks, let's use the following module and simple service as an example. How would we go about unit testing this?

```
// create a new module
angular.module('products', []);

// register the service
angular.module('products').service('CategoryService', function CategoryService()
 {
  return {
    getCategories: function() {
      return { 1: 'Beverages', 2: 'Condiments' };
    }
  };
});
```

In the last post we covered three ways of registering modules via mock.module. It's important to keep these in mind, since this post will explain how these concepts link together in testing elements of an application in isolation. They are:

1. string
2. function()
3. Object

Using the String alias form to Register Modules

Starting with the string form of registering a module, let's update our test as follows:

```
describe('products category service', function () {
  it('should return the expected categories', function () {
```

```
    // registers module by its string alias
    angular.mock.module('products');

    // Where does this instance come from?
    var categories = CategoryService.getCategories();

    expect(categories).toEqual({ 1: 'Beverages', 2: 'Condiments' });
  });
});
```

The assertion is straight forward, we call the *getCategories* function from the service and check that the object returned is what we expect. The unanswered question is, how can we get hold of an instance of the *CategoryService*? This is where the *angular.mock.inject* functionality helps us.

Here's the same test with the angular.mock.inject functionality being used:

```
describe('products category service', function () {
  it('should return the expected categories', function () {
    angular.mock.module('products');

    var service;

    // Get the service from the injector
    angular.mock.inject(function GetDependencies(CategoryService) {
      service = CategoryService;
    });

    // call the function on our service instance
    var categories = service.getCategories();

    expect(categories).toEqual({ 1: 'Beverages', 2: 'Condiments' });
  });
});
```

What does inject do? At a high level, the mock.inject function acts a wrapper for the angular injector. The angular injector acts as a service locator for our application.

The mock.module and mock.injector we just saw work together as follows:

1. We registered the products module via *mock.module*.
2. We called the inject function, passing a function with an argument called *CategoryService*.
3. The injector looks for an object within the loaded module(s) (products, in our

example) for something called *CategoryService*.
4. It finds the *CategoryService* within the products module, and passes an instance to our *GetDependencies* function.
5. We can use this instance for our tests.

Taking our example further, let's introduce another dependency to the products module:

```
// create a new module
angular.module('products', []);

// register the service
angular.module('products').service('CategoryService', function CategoryService()
 {
  return {
    getCategories: function() {
      return { 1: 'Beverages', 2: 'Condiments' };
    }
  };
});

// register a service. **This has a dependency on the CategoryService**
angular.module('products').service('ProductsService', function ProductService (C
ategoryService) {
  return {
    getProducts: function () {
      var product1 = { name: 'Chai', categoryId: 1 };
      var product2 = { name: 'Aniseed Syrup', categoryId: 2 };
      var products = [product1, product2];

      var categories = CategoryService.getCategories();

      products.forEach(function (p) {
        // append the category name for the category service to each product.
        p.categoryName = categories[p.categoryId];
      });

      return products;
    }
  };
});
```

This new ProductsService relies on the new CategoryService to get the category names.

How would we write a test for this new service given the extra dependency? Here's the updated test code:

```
describe('products service tests', function () {

  // Note that we can move the call to module in the beforeEach block,
  // thus making it available for each test, keeping our tests DRY.
  beforeEach(angular.mock.module('products'));

  it('should append category names to products', function () {
    var service;

    // Get the service from the injector
    angular.mock.inject(function GetDependencies(ProductsService) {
      service = ProductsService;
    });

    var products = service.getProducts();
    expect(products[0].categoryName).toBe('Beverages');
    expect(products[1].categoryName).toBe('Condiments');
  });
});
```

The key point with this second test, is that the call to inject for the *ProductsService* also resolved the reference to the *CategoryService* that is used by the *ProductsService* itself. This is down to the magic of the angular injector.

Using the Object form to Register Modules

Unit testing means testing in isolation, so given that the test is for the *ProductsService*, we should be mocking the *CategoryService* that it uses. How can we do this? In this test we attach our own instance of the *CategoryService* service to the injector. Note the usage of the Object argument when registering the module via *mock.module*:

```
it('should append category names to products', function () {

  // Note that this version of the CategoryService overrides the version we adde
d to the products module.
  angular.mock.module({
    'CategoryService': {
      getCategories: function() {
        return { 1: 'Electronics', 2: 'DVDs' };
      }
    }
  });

  var service;
```

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

  var products = service.getProducts();
  expect(products[0].categoryName).toBe('Electronics');
  expect(products[1].categoryName).toBe('DVDs');
});
```

The result of this test demonstrates that we are using this new version of *CategoryService* and not the actual implementation as we saw with previous tests. This is a trivial example, but it demonstrates the concept simply.

The key point of this test, is that we can have multiple modules within an app **but there is only one injector**. This means that names of services can clash, even if they are in different modules! But of course it also allows us to override previously registered components with our own versions for testing.

Another point is that order matters. The products module had already been registered in the *beforeEach* block that we introduced in the test before last. Our mock service took precedence as it was registered after the products module.

Using the Anonymous Function form to Register Modules

Now let's introduce a controller that has a dependency on the *ProductsService*:

```
angular.module('products').controller('ProductsController', function ProductsCon
troller($scope, ProductsService) {
  $scope.products = [];
  $scope.load = function() {
    $scope.products = ProductsService.getProducts();
  };
});
```

How can we test this? We need to mock the *$scope* and *ProductsService* arguments, and then obtain an instance of the *ProductsController*. Here's the code:

```
describe('products module', function () {

  beforeEach(module('products'));

  beforeEach(angular.mock.module({
```

```
    'ProductsService': {
      getProducts: function() {
        return [{ name: 'Aniseed Syrup', categoryId: 2, categoryName: 'Condiment
s' }];
      }
    }
  }));

  var $controller;

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

  describe('load', function () {
    it('products should be loaded to scope', function () {
      var $scope = {};
      var controller = $controller('ProductsController', { $scope: $scope });
      $scope.load();
      expect($scope.products).toEqual([{ name: 'Aniseed Syrup', categoryId: 2, c
ategoryName: 'Condiments' }]);
    });

    it('products should default to empty array', function () {
      var $scope = {};
      var controller = $controller('ProductsController', { $scope: $scope });
      expect($scope.products).toEqual([]);
    });
  });

});
```

This first thing to note, is that the mechanism for injecting controllers is different from the method we've seen thus far. This is due to the fact that angular works differently under the covers when registering controllers.

We make use of the $controller decorator from ngMock. We can get an instance of the controller by passing its name to the $controller function instance. As a side note, if you noticed the use of underscores for the $controller function argument, it's a convention in angular that allows the use of underscores for the function arguments to inject. The injector unwraps the underscores thus allowing us to use the literal *$controller* for the variable assignment inside the function.

Secondly, note that we can easily provide a mock $scope argument, since it's a simple JavaScript object.

Lastly, as we registered a test version of the *ProductsService* before calling inject, the injector uses this version for the *ProductController* dependency.

If you recall from the previous post, we noted that the difference between the Object and function() method of creating an anonymous module, is that the Object cannot take any dependencies. This trivial example demonstrates this:

```
describe('products module', function () {

  // register the existing module from our app
  beforeEach(module('products'));

  describe('load', function () {

    it('products should be loaded to scope', function () {

      // Create an anonymous module with Object argument
      // NB this overrides the CategoryService from our products module
      angular.mock.module({
        'CategoryService': {
          getCategories: function() {
            return { 1: 'Electronics', 2: 'DVDs' };
          }
        }
      });

      // Create an anonymous module with function() argument
      // NB this overrides the ProductsService from our products module and
      // will also use our previously mocked CategoryService.
      angular.mock.module(function($provide) {
        $provide.service('ProductsService', function(CategoryService) {
          return {
            getProducts: function() {
              var mockCategories = CategoryService.getCategories();
              return [{ name: 'Aniseed Syrup', categoryId: 2, categoryName: mock
Categories[2] }];
            }
          };
        });
      });

      var $controller;

      angular.mock.inject(function(_$controller_){
        $controller = _$controller_;
```

```
      });

      var $scope = {};
      var controller = $controller('ProductsController', { $scope: $scope });
      $scope.load();
      expect($scope.products).toEqual([{ name: 'Aniseed Syrup', categoryId: 2, c
ategoryName: 'DVDs' }]);
    });
  });

});
```

As noted, this is a trivial example, but it shows how our mock ProductService can make use of the mock CategoryService.

Taking a Dive into the Source Code

Now that we have connected the dots between registering modules in tests via *angular.mock.module* and calling *angular.mock.inject*, we can take a deeper look at the source code.

Below is a version of the *angular.mock.inject* function with some elements removed for brevity:

```
window.inject = angular.mock.inject = function() {
  var blockFns = Array.prototype.slice.call(arguments, 0);
  return isSpecRunning() ? workFn.call(currentSpec) : workFn;
  ////////////////////
  function workFn() {
    var modules = currentSpec.$modules || [];
    modules.unshift('ngMock');
    modules.unshift('ng');
    var injector = currentSpec.$injector = angular.injector(modules, strictDi);
    for (var i = 0, ii = blockFns.length; i < ii; i++) {
      injector.invoke(blockFns[i] || angular.noop, this);
    }
  }
};
```

You can see the full source code of the function here.

This process is like the ngApp and/or bootstrap functionality for starting an angular application from the browser.

What this code does, is:

```
ADD ng, ngMock to the front of any existing modules registered in our tests
SET injector = call to angular.injector(modules)
FOR EACH fn in arguments array:
  call injector.invoke(fn)
```

The angular injector is a complex concept, but to explain it briefly, it takes all services, controllers etc from all modules that make up an app and holds them in a single injector container. When we ask for an instance of a service, controller etc we must ask the injector. Since the injector knows about all the components that make up our app, if we ask for a service that has a dependency on a different service, the injector will resolve that dependency for us.

The injector acts as a wrapper for the provider service, which we cannot access directly, we have to use the injector's API for this from the angular application code.

In our tests however, we can access the $provide service, that the injector calls. For example:

```
angular.mock.module(function($provide) {
  $provide.service('FooService', function() {
    return { ... }
  });
});
```

The call to *injector.invoke* in the source code takes our function argument to *mock.inject*, passes it through the injector and along the way any arguments to that function which match the name of a service registered to the injector will be populated.

A call to mock inject:

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

Would make an equivalent call to the angular injector:

```
injector.invoke(function(MyService){
  service = MyService;
});
```

Conclusion

This post concludes a first look at the ngMock fundamentals giving us a solid foundation to build upon. Expect a lot more to come relating to testing with AngularJS.

See more ngMock Tutorials

This is one of many ngMock tutorials. See the full list of AngularJS Unit Testing Tutorials.

CONNECT WITH BRADLEY

Bradley Braithwaite is a full-stack software engineer who works at duckduckgo.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:

This is great information, but I'm actually kind of glad I didn't find it before setting up our unit testing workflow. What we do is just have a directory called "mocks" that karma is pointed to and told to load all the js files in it. The mock files there register themselves as if they were the real deal, so they overwrite the original service without much fuss. This works well most of the time, and doesn't clutter up the test code with lots of repetitive mock code. The downside is that the mocked values aren't right there in the file for you to look at--you have to go look somewhere else--but this also contributes to making the mocks more reusable. And, of course, there are those two or three times that we've needed to mock different values for different tests, which is why I was looking today for alternative strategies. The above information is great for those situations.

• Reply • Share ›

fayjai • 4 months ago

Can you explain why you don't need to include the ProductsService in the object you pass to $controller in your code? Specifically, your code has the following when testing the ProductsController:

var controller = $controller('ProductsController', { $scope: $scope });

But why is it not the following:

var controller = $controller('ProductsController', { $scope: $scope: ProductsService: ProductsService });

Is it because the $controller function will know to resolve any services that are not included in the locals hash?

• Reply • Share ›

Bradley Braithwaite Mod ➚ fayjai • 4 months ago

Great question. It's because both $controller service and ProductsService are served by the same injector. Therefore when we ask for $controller('ProductsController', { $scope: $scope });, the injector goes and fetches an instance. The injector is clever enough to know that the ProductsController requires the ProductsService, so it also resolves that instance. If we did pass it in via the locals argument i.e. $controller('ProductsController', { $scope: $scope: ProductsService: ProductsService });, the injector would see we've already passed the ProductsService and not try to find it.

Hope I've explained well?

1 • Reply • Share ›

fayjai ➚ Bradley Braithwaite • 4 months ago

Yes this is super helpful and clear! I have a follow up question related to your comment that both $controller service and ProductsService are served by the same injector. When you register several modules like this:

beforeEach(angular.mock.module('products'))

```
beforeEach(angular.mock.module({
'ProductsService': {
getProducts: function() {
return [{ name: 'Aniseed Syrup', categoryId: 2, categoryName:
'Condiments' }];
}
}
}));
```

How does angular.mock.inject know to look across multiple modules for any Services that have been registered in those modules? The reason I ask is because I think for the normal angular.injector code, you have to specify which modules to look for the Services in (i.e. angular.injector(['products', 'someOtherModule']). I guess I'm curious if the inject functionality is slightly different across the mock and non-mock versions.

• Reply • Share ›

Bradley Braithwaite Mod ➚ fayjai • 4 months ago

You can have multiple modules, but only one injector per app. So if you have a service named 'ProductsService' in 'moduleA', and 'moduleB' also has a service called 'ProductsService', you would have a name clash if you were to use 'moduleA' and 'moduleB' together in the same app.

ngMock's injector adds some extra functionality to bootstrap the application, but the underlying functionality is the same.

• Reply • Share ›

fayjai ➚ Bradley Braithwaite • 4 months ago

Thanks so much for the explanation - this makes a ton of sense.

My guess is that when an angular app is bootstrapped, angular uses the modules that are declared as dependencies for the entire app to create the injector (i.e. angular.injector(['list', 'of', 'all', 'module', 'dependencies']). And if you have multiple modules with the same Service name, then the order in which you declare the dependencies will determine which Service gets overridden.

• Reply • Share ›

Bradley Braithwaite Mod ➚ fayjai • 4 months ago

Yep, exactly that.

1 • Reply • Share ›

fayjai ➚ Bradley Braithwaite • 4 months ago

Thank you!

• Reply • Share ›