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

Check out my latest online course: AngularJS Unit Testing in-depth with ngMock.

ngMock Fundamentals for AngularJS Unit Testing - Understanding Module

Part 1 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 24, 2015 on *javascript, angularjs, testing, ngmock*

The last post detailed how to get started with Karma for AngularJS Testing and concluded the first few stages necessary to get setup with the tooling. Now we can focus on taking a more in-depth look into the AngularJS framework, starting with the ngMock module.

ngMock was designed to make it easier to unit test modules. It also extends some of the core services to make testing asynchronous code easier.

This post is the first of two, that takes a deeper dive into two main concepts from ngMock that we used in the previous tests:

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

This post will focus on *angular.mock.module*.

We will touch upon AngularJS internals and will examine some of the Angular.js source code. It will be tricky, but the end-result will be a solid foundation of the ngMock framework and some of the core principles of AngularJS.

NB We talk about version 1.4 source code. Angular 2 is of course in the pipeline and set for release later this year, however there is still 1.4 code being written and an abundance of 1.x code around in the wild already, so there is value in learning these concepts.

Understanding Angular Modules and Mock.Module

To recap, an Angular module acts like a container for different components that make up an angular app. The *angular.module* is the global place for creating these containers. The basic functionality it offers, is to:

- Create a module
- Retrieve a module

(If you are a little rusty on this topic, it was covered in this post).

Loading Modules for the Full Application

If we were starting up a conventional angular application in the browser, we would need to initialise the application when the browser document is ready. There are two ways of doing this:

- Option 1. ngApp

This is the auto initialisation method using ng-app attribute:

```
<!doctype html>
<html ng-app="calculatorApp">
  <body>
    ...
    <script src="angular.js"></script>
  </body>
</html>
```

NB the attribute on the HTML element *ng-app="calculatorApp"* loads the module named *calculatorApp*. This calls the same bootstrap function that follows.

- Option 2. Bootstrap

This is the manual initialisation using angular.bootstrap that was used in our calculator example in an earlier post:

```
<!doctype html>
<html>
  <body>
    ...
    <script src="angular.js"></script>
    <script>
        angular.element(document).ready(function() {
            angular.bootstrap(document, ['calculatorApp']);
        });
    </script>
  </body>
</html>
```

NB the HTML does not use the ngApp directive.

Similar to the first mechanism, here we explicitly call the bootstrap function with the module name passed as an argument.

Loading Specific Modules for Unit Tests

When we unit test, we don't load a browser in the same way, so how can we initialise the app if we can't use the two options we just discussed? The *angular.mock.module* provides this mechanism to initialise Angular modules. Next we can look into how this works.

Note that the ngMock include makes *module* available on the window object, therefore *angular.mock.module* and *module* are the same thing. To avoid confusion between *angular.module*, *angular.mock.module* and *window.module* in this post, I will use the fully qualified version of: *angular.mock.module*.

The *angular.mock.module* allows us to pass one or many arguments of type:

- string - the name of an existing module.
- function() - creates an anonymous module.
- Object - creates an anonymous module.

Let's next examine each type.

string

You would use this to register an existing module via its string alias (name) that's part of an application.

For example, let's consider we had these modules that make up our app:

```
angular.module('products', []);
angular.module('categories', []);
```

NB these lines are not from the test files, but part of the application code.

If we wanted to just unit test features of the products module, we could register the module by its string alias i.e. name:

```
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?');
    });
});
```

If we wanted to include both modules in our tests, we could update the mock.module call to be:

```
angular.mock.module('products', 'categories');
```

We could also register the modules separately (but within the same spec/describe block):

```
angular.mock.module('products');
angular.mock.module('someOtherModule');
```

The concept of registering single or multiple modules (via comma separated list or across multiple lines) is the same for the remaining two argument types.

You can also mix types for registering modules, e.g:

```
var alias = 'products';
var objType = {}
var funcType = function() { return 1; }

angular.mock.module(alias, objType, funcType);
```

function()

You would use this to anonymously construct a module e.g. we don't have to register the modules in our main app as we did in the string type example. In the following code snippet, we are registering a nameless module with a $provider.constant service:

```
describe('function arg', function () {
    it('should load module with anonymous function', function () {
        angular.mock.module(function($provide) {
            $provide.constant('discountPercentage', 0.10);
            // We could register other provider services here... e.g.
            // $provide.value('apiKey', 'abc123');
        });

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

Object

In the same manner as the anonymous function, you would use an object literal to anonymously construct a module. In this form, the object key/value pairs will be used under the covers to create a $provide.value service, where the key and value of the object item are set accordingly to the name/value of the value arguments:

```
describe('object arg', function () {
    it('should load module with object', function () {
        angular.mock.module({
            'customerService': { getCustomerIds: function() { return [4,5,6];  }
  }
        });

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

function() vs Object

The examples we used were similar, so what's the key difference between the *function()* and *Object* form? As the Object type gets added as a *$provider.value*, it is therefore restricted to the capabilities of the value service, most importantly that a value service

cannot be injected with other services.

The Source Code

The following snippet is the *angular.mock.module* function itself from the [ngMock source code]():

```
window.module = angular.mock.module = function() {
  var moduleFns = Array.prototype.slice.call(arguments, 0);
  return isSpecRunning() ? workFn() : workFn;
  /////////////////////
  function workFn() {
    if (currentSpec.$injector) {
      throw new Error('Injector already created, can not register a module!');
    } else {
      var modules = currentSpec.$modules || (currentSpec.$modules = []);
      angular.forEach(moduleFns, function(module) {
        if (angular.isObject(module) && !angular.isArray(module)) {
          modules.push(function($provide) {
            angular.forEach(module, function(value, key) {
              $provide.value(key, value);
            });
          });
        } else {
          modules.push(module);
        }
      });
    }
  }
};
```

What this code does, is:

```
FOR EACH arg in arguments array:
    IF arg is the Object type:
        FOR EACH key in the obj:
            add to $provide as a value service (using obj.key as name, and obj.v
alue as value)
    ELSE
        add the arg, as-is to the list of modules
```

For example, if we had the following code in our test:

```
var alias = 'products';
```

```
var func = function($provide) {
    $provide.constant('discountPercentage', 0.10);
}
var obj = {
    'customerService': { getCustomerIds: function() { return [4,5,6];  }  }
}

angular.mock.module(alias, obj, func);
```

The modules array in the angular.mock.module function would result in the following:

```
modules = [
    'products',           // 0 array position
    function($provide) {  // 1 array position
        $provide.constant('discountPercentage', 0.10);
    },
    function($provide) {  // 2 array position
        $provide.value('getCustomerIds', function() { return [4,5,6];  } );
    }]
```

The three items added to the modules array need to be further processed before they can be used for tests, but by what? That's what we will cover in the next post: the injector!

Part 2 of this post is now available: ngMock Fundamentals for AngularJS - Understanding Inject

See more ngMock Tutorials

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

SHARE

Don't miss out on the free technical content:

Subscribe to Updates

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:

2 Comments bradoncode 1 Login

Recommend Share Sort by Best

Join the discussion…

fayjai • 3 months ago

Just to confirm that I understand the difference between the Object and Function form of angular.mock.module:

1) If you're trying to inject a service like CategoryService which has no dependencies on other services, then you can use the Object or Function form.

2) But if you're trying to inject a service like ProductsService which has a dependency on CategoryService, then you have to use the Function form.

Are these statements accurate?

• Reply • Share ›

Bradley Braithwaite Mod fayjai • 3 months ago

Yes that sounds about right. The object form creates a $provider.value: https://docs.angularjs.org/api... as in, it doesn't have a constructor.

1 • Reply • Share ›

ALSO ON **BRADONCODE** WHAT'S THIS?

How to Unit Test $http in AngularJS - Part 1 - ngMock Fundamentals

3 comments • 7 months ago

Stanislav Verjikovskiy — Thanks for helping understand expect vs when

How to Unit Test $http in AngularJS - Part 2 - ngMock Fundamentals

2 comments • 7 months ago

fayjai — There is a slight typo here: statusText - at time time of writing, this doesn't appear to work.

How to Unit Test Exceptions in AngularJS - ngMock Fundamentals

1 comment • 7 months ago

Anil Singh — Its very useful post. Thank you! http://www.code-sample.com/

Unit Testing $interval in AngularJS - ngMock Fundamentals

5 comments • 7 months ago

fayjai — In your test code for "should start the interval on click with user values", I believe you need to modify your

Home | Blog | Tutorials | About | RSS

© 2015 Bradley Braithwaite