GETTING STARTED WITH UNIT TEST FOR ANGULAR

Using Jasmine, Visual Studio and Chutzpah

By Vishal Kumar at https://bizsitegenie.com

BizSiteGenie

Contents

This E-Book will guide you step by step in writing your first Unit Test for Angular.

You can see it all in action at this YouTube video:

https://www.youtube.com/watch?v=D1HTzK92jqM.

Tools and framework we are using:-

As a starting point, we will need an Editor and a Test Runner. There are so many different options available for both, but we will be using the following:-

Visual Studio

Visual Studio is a very powerful editor primarily used by C# developers. But it has good support for Javascript as well.

Jasmine for writing the test

Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean obvious syntax to easily write tests.

Chutzpah for running the test.

Chutzpah is an open source JavaScript test runner which enables you to run unit tests using QUnit, Jasmine, Mocha, CoffeeScript and TypeScript.

Installation:-

The following softwares should be installed on your computer as prerequisites:

•Node.js

•Visual Studio

•Chutzpah Test Adapter for Visual Studio

•Chutzpah Test Runner Context Menu for Visual Studio

Once you have installed them, you will be ready to run Unit Tests for AngularJS.

Describe Function:-

A test suite begins with a call to the global Jasmine function **describe** with two parameters: a **string** and a **function**. The string is a name or title for a spec suite that explains what is being tested. The function is a block of code that implements the suite.

```
describe( ‘String’ ,Function(){

});
```

It Function(Specs) :-

Specs are defined by calling the global Jasmine function **it**, which is like ’describe’ in that it takes a **string** and a **function** as parameters. The string is the **title** of the spec and the **function** is the spec or test. A spec contains one or more expectations that test the state of the code. An expectation in Jasmine is an assertion that is either true or false. A spec with all true expectations is a passing spec. A spec with one or more false expectations is a failing spec.

```
describe( ‘String’ ,Function(){
            it( ‘Title’ ,Function(){
            });
});
```

Expect Function :-

Expectations are built with the function **expect** which takes a **value** field to match with actual result. It is **chained** with a **Matcher** function, which takes the expected value.

```
describe( 'String' ,Function(){
                it( 'Title' ,Function(){
                                expect(Value).toEqual(value);
                });
});
```

Matcher Function :-

Each matcher implements a boolean comparison between the actual value and the expected value. It is responsible for reporting to Jasmine if the expectation is true or false. Jasmine will then pass or fail the spec.

Any matcher can evaluate to a negative assertion by chaining the call to expect, with a not operator added before it.

Jasmine has a rich set of matchers included.

```
describe( 'String' ,Function(){
                it( 'Title' ,Function(){
                                expect(Value).toEqual(value);
                });
});
```

Example:-

Let’s go through a short example to understand the Unit Test runner by testing the sum of two numbers. When executed, it will assert whether we are getting a correct output from our summing function.

The below code snippet shows the use of describe, it, expect and matcher functions all laid out together.

```
describe('Our first test', function() {
            it('should be able to test summing of two values', function() {
                        var sum = 10 + 20;
                        expect(sum).toEqual(30);
            });
});
```

The above code will test summing of two values and output the test result.

We start the process of running by right clicking the file in Visual Studio and selecting “Run JS Test”. We can also run this test on the browser by selecting “Run on Browser”.

It will compare the expected value to the output value by using the **.toEqual** function.

If the value is equal to the output, we will see the result“1 Pass 0 Fail”.

If the value is not equal to the output, it will be “0 Pass”.

Unit Testing for Angular Code:-

For Angular Code testing, we have to include our angular files in the Project.

We can include these files by including the corresponding Visual Studio package by right clicking on the solution file and selecting References->Manage Nuget Packages.

Then we can search for “Angular “ to display all the packages related to Angular. Select the main Angular JS package and it will install the latest version of the relevant package. All the files will get downloaded in a scripts folder. Now we can write our Angular code.

Angular Controller:-

In Angular, a Controller is defined as a JavaScript **constructor function** that is used to augment the Angular Scope.

When a Controller is attached to the DOM via the ng-controller directive, Angular will instantiate a new Controller object, using the specified Controller's **constructor function**.

A new **child scope** will be created and made available as an injectable parameter to the Controller's constructor function as $scope.

Let’s write our Controller function. Create a new Javascript file that will store the controller code. Since this controller will implement the summing of numbers, we can save the file as “sumcontroller.js”.

The Controller should belong to a module. Define a module and name it as **summingModule**. There are no dependencies to the module, so we can keep the second parameter blank.

Then we define the Controller and and give it the name SumController.

The dependency of the controller is the $scope object.

After defining it, we add a function **sum** which takes two values and returns the sum of the two numbers.

Below is what our controller function code should look like:

```
angular.module('summingModule', [])
        .controller('SumController', function($scope){
                $scope.sum = function(first, second){
                        return first + second;
                };
        });
});
```

Test our controller function using Jasmine:-

Let’s create a test file to test the controller function.

Save the new file as “sumcontroller.test”.

We start by writing a **describe** function as explained previously.

We describe the operation as **“summingModule”** and write all the test code within the function.

Within this summing module, we also describe the controller we are testing as **SumController** which will be a nested describe function under the top one.

Then we define an **It function** within the describe for **SumController** module.

Below is the outline for this code:

```
angular.module('summingModule', [])
        describe(‘SumController’,function(){
                it('should be able to return the sum of two values', function() {

                });
        });
});
```

To test our Angular code, we will need to reference those files in our test code.

With Chutzpah, it is easy to include references. We can directly insert the links of references at the top of the test code.

We will reference three files:-

1.Angular.min.js (The main framework)

2.Angular.mocks.js(We can use this file to manually bootstrap our code).

3.SumController.js (File which we are testing).

Angular.mock.js is included to simulate injection of modules just like it would be injected in an actual html page.

For this, we will be using a module function provided by **angular.mock.js** which will instantiate the **summingModule.**

We will place it in a **beforeEach** statement to run the statement each time a test is executed.

```
beforeEach(module( ‘summingModule’ ));
```

Mock instance of “sumController” module:-

We defined the module for sumController in our basic framework.

Now we can create a mock instance for our SumController.

We will use the inject function for this. The **inject**() function creates new instance of $injector per test, which is then used for resolving references.

To the injection function, we provide the parameters which are to be injected.

In this case, **$rootScope** and **$controller** will be those parameters.

Then we define two variables for scope and controller in which we can store injected references.

We cannot inject the scope directly. So we inject a root scope to create a local instance of a scope.

```
scope=$rootScope.$new();
```

We can create a controller instance using **$controller**.

Pass the name of the controller we want to instantiate as the parameter. Add a $scope parameter, assigning it the value of scope that we created above.

```
controller=$controller( ‘SumController’ ,{

                $scope:scope

});
```

This is what our framework for the test code looks like

```
describe( ‘SumController’ ,function(){

                var scope,controller;

                beforeEach(inject(function($rootScope,$controller){

                scope=$rootScope.$new();

                controller=$controller( ‘SumController’ ,{

                                $scope:scope

                                });

                }));

});
```

Finally! The Unit Test:-

Now we can easily write a test in the **it function** after the beforeEach block to check the sum of numbers:

```
it(‘should be able to return sum of two numbers’,function {

                var result=scope.sum(10,20);

                expect(result).toEqual(30);

});
```

On running this test, the result will display the expected value ‘1 pass and 0 fail’

Here is the final unit test code in one place

```
describe(‘summingModule’,function(){
        describe(‘SumController’,function(){
                var scope,controller;
                beforeEach(inject(function($rootScope,$controller){
                        scope=$rootScope.$new();
                        controller=$controller(‘SumController’,{
                            $scope:scope
                        });
                }));
                it(‘should be able to return sum of two numbers’,function {
                        var result=scope.sum(10,20);
                        expect(result).toEqual(30);
                });
        });
});
```

That’s it!! We have written our first Angular test!

Hope this gives you enough to start writing your unit tests.

See it all in action here: https://www.youtube.com/watch?v=D1HTzK92jqM