Unit Testing Automation using Jasmine in SharePoint Applications

Table of Contents

[1. Introduction 4](#_Toc479171867)

[Objective 4](#_Toc479171868)

[2. Introduction to Jasmine 4](#_Toc479171869)

[Suites 4](#_Toc479171870)

[Specs 4](#_Toc479171871)

[It's Just Functions 4](#_Toc479171872)

[Asynchronous Support 4](#_Toc479171873)

[3. Required JavaScript file to run test cases 5](#_Toc479171874)

[4. Get Jasmine JS and CSS File for your App 5](#_Toc479171875)

[Node JS 5](#_Toc479171876)

[Nuget using Visual Studio 5](#_Toc479171877)

[5. Simple Example 6](#_Toc479171878)

[6. Example with SharePoint 10](#_Toc479171879)

[7. References 13](#_Toc479171880)

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision Log** | | | |
| **Date** | **Revision** | **Summary** | **Revised By** |
| 04/05/2017 | 1.0 | Jasmine Testing User Guide | Hanumant Patil |
|  |  |  |  |
|  |  |  |  |

# **Introduction**

## Objective

The overall objective of this document is to outline the process to writing JavaScript test cases using jasmine framework.

# **Introduction to Jasmine**

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 so that you can easily write tests.

## Suites

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 - usually what is being tested. The function is a block of code that implements the suite.

## Specs

Specs are defined by calling the global Jasmine function it, which, like describe takes a string and a function. 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.

## It's Just Functions

Since describe and it blocks are functions, they can contain any executable code necessary to implement the test. JavaScript scoping rules apply, so variables declared in a describe are available to any it blocks inside the suite.

## Asynchronous Support

Jasmine also has support for running specs that require testing asynchronous operations. Calls to **beforeEach**, **it**, and **afterEach** can take an optional single argument that should be called when the async work is complete.

This spec will not start until the done function is called in the call to beforeEach above. And this spec will not complete until its done is called.

By default jasmine will wait for 5 seconds for an asynchronous spec to finish before causing a timeout failure. If the timeout expires before done is called, the current spec will be marked as failed and suite execution will continue as if done was called.

# **Required JavaScript file to run test cases**

Below is the list of JavaScript file to run jasmine test cases

1. CSS file: -

<link rel="stylesheet" type="text/css" href="Content/jasmine/jasmine.css" />

1. JavaScript File: -

<script type="text/javascript" src="Scripts/jasmine/jasmine.js"></script>

<script type="text/javascript" src="Scripts/jasmine/jasmine-html.js"></script>

<script type="text/javascript" src="Scripts/jasmine/boot.js"></script>

1. Required DEV on html page

<div id="jasmine-specs"></div>

# **Get Jasmine JS and CSS File for your App**

Below is the method to get jasmine to you App

## Node JS

1. Make sure you have Node.js installed. This will already be the case if you’ve installed the Visual Studio Tools for Apache Cordova, otherwise visit [https://nodejs.org](https://nodejs.org/).
2. Open a command prompt and navigate to the folder you created for this exercise.
3. Run the following commands:

|  |  |
| --- | --- |
| Command | Purpose |
| *npm install --save-dev jasmine* | Installs Jasmine |

## Nuget using Visual Studio

1. To install Jasmine Test Framework, run the following command in the [Package Manager Console](https://docs.nuget.org/docs/start-here/using-the-package-manager-console)

Install-Package JasmineTest

# **Simple Example**

1. Below is the unit test sample function to convert a piece of JSON with one set of properties into an object with different properties. Save this code to **js/normalize.js**:
2. /\*\* @description Converts JSON data that may contain Name and PersonalIdentifier
3. \* properties to an object with the properties name (string) and id (positive
4. \* integer up to 9999999999.
5. \* @param {string} jsonIn The JSON data to normalize.
6. \* @return {object} An object with name (string) and id (integer) properties,
7. \* defaulting to "default" and 0, or null if the JSON is null or invalid.
8. \*/
9. function normalizeData(jsonIn) {
10. data = JSON.parse(jsonIn);
11. return {
12. name: data.Name,
13. id: data.PersonalIdentifier
14. };
15. }
16. Next, each unit test is a piece of code that validates the unit by:
17. Calling it with specific inputs, and,
18. Checking the output against the expected value.

A unit test must follow the conventions of the test framework you’re using. In this case, let’s use Jasmine. Save this code to **test/normalize\_tests.js**:

// First argument to 'describe' (which is defined by Jasmine) is the testing module that will

// appear in test reports. The second argument is a callback containing the individual tests.

describe("normalizeData", function () {

// The 'it' function of Jasmine defined an individual test. The first argument is

// a description of the test that's appended to the module name. Because a module name

// is typically a noun, like the name of the function being tested, the description for

// an individual test is typically written in an action-data format.

it("accepts golden path data", function () {

// Invoke the unit being tested as necessary

var json = '{"Name": "Maria", "PersonalIdentifier": 2111858}';

var norm = normalizeData(json);

// Check the results; "expect" and toEqual are Jasmine methods.

expect(norm.name).toEqual("Maria");

expect(norm.id).toEqual(2111858);

});

});

Notice how this individual unit test is **specific**: it calls the unit under test with *one* set of inputs and gives an *exact* name/description for the test with those inputs. This follows the best practice of isolating each unit test to an individual test case, creating a 1:1 mapping between the name of the test, as it appears in reports, and the exact test case (that is, the arguments used in the test). When the test runner reports a failure, then, you know exactly where to look in your code and can easily step through that one test in the debugger to isolate the failure.

1. Below is the html page that will display test result



Code: -

html xmlns="http://www.w3.org/1999/xhtml">

<head >

<title></title>

<link rel="stylesheet" type="text/css" href="Content/jasmine/jasmine.css" />

<script type="text/javascript" src="Scripts/jasmine/jasmine.js"></script>

<script type="text/javascript" src="Scripts/jasmine/jasmine-html.js"></script>

<script type="text/javascript" src="Scripts/jasmine/boot.js"></script>

<!-- include source files here... -->

<script src="js/normalize.js"></script>

<!-- include spec files here... -->

<script src="test/normalize\_tests.js"></script>

</head>

<body>

<div id="jasmine-specs"></div>

</body>

</html>

# **Example with SharePoint**

Below example user content editor web part to test jasmine test cases

* 1. Create document library
  2. Upload below JavaScript and CSS file to document library
     1. jquery-1.9.1.min.js
     2. jasmine/jasmine.css
     3. jasmine.js
     4. jasmine-html.js
     5. boot.js
  3. Create content editor web part to a SharePoint Page
  4. Add below references

<script type="text/javascript" src="Your Site URL/Style%20Library/Test demo/jquery-1.9.1.min.js"></script>

<link rel="stylesheet" type="text/css" href=" Your Site URL /Style%20Library/Test demo/Content/jasmine/jasmine.css" />

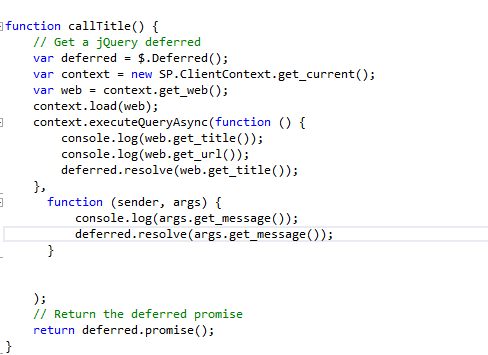
<script type="text/javascript" src=" Your Site URL /Style%20Library/Test demo/Scripts/jasmine/jasmine.js"></script>

<script type="text/javascript" src=" Your Site URL /Style%20Library/Test demo/Scripts/jasmine/jasmine-html.js"></script>

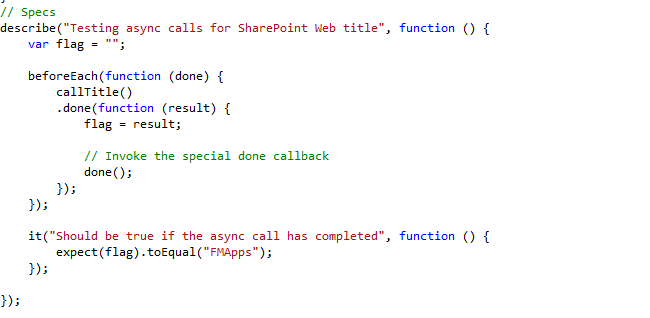
<script type="text/javascript" src=" Your Site URL /Style%20Library/Test demo/Scripts/jasmine/boot.js"></script>

* 1. Add below JavaScript code to content editor web part

1. <script>
2. // Method to get web title
3. function callTitle() {
4. // Get a jQuery deferred
5. var deferred = $.Deferred();
6. var context = new SP.ClientContext.get\_current();
7. var web = context.get\_web();
8. context.load(web);
9. context.executeQueryAsync(function () {
10. console.log(web.get\_title());
11. console.log(web.get\_url());
12. deferred.resolve(web.get\_title());
13. },
14. function (sender, args) {
15. console.log(args.get\_message());
16. deferred.resolve(args.get\_message());
17. }
18. );
19. // Return the deferred promise
20. return deferred.promise();
21. }
22. // Specs
23. describe("Testing async calls for SharePoint Web title", function () {
24. var flag = "";
25. beforeEach(function (done) {
26. callTitle()
27. .done(function (result) {
28. flag = result;
29. // Invoke the special done callback
30. done();
31. });
32. });
33. it("Should be true if the async call has completed", function () {
34. expect(flag).toEqual("FMApps");
35. });
36. });
37. // Code under test
38. function testAsyncWithDeferredReturnValue() {
39. // Get a jQuery deferred
40. var deferred = $.Deferred();
41. // Wait two seconds, then set the return true
42. setTimeout(function () {
43. // Resolve the deferred
44. deferred.resolve(true);
45. }, 2000);
46. // Return the deferred promise
47. return deferred.promise();
48. }
49. // Specs
50. describe("Testing async calls with beforeEach and invoking the special done callback in the promise's done callback and using the promise's return data", function () {
51. var flag = false;
52. beforeEach(function (done) {
53. testAsyncWithDeferredReturnValue()
54. .done(function (result) {
55. flag = result;
56. // Invoke the special done callback
57. done();
58. });
59. });
60. it("Should be true if the async call has completed", function () {
61. expect(flag).toEqual(true);
62. });
63. });
64. $(document).ready(function () {
65. //load sharepoint JS File to Work With CSOM
66. ExecuteOrDelayUntilScriptLoaded(callMethode, "sp.js");
67. });
68. function callMethode() {
69. callTitle();
70. }
71. </script>
    1. Screens of code
       1. Method to find web title of shrepoint site (Used client side objet model + jquery promises)



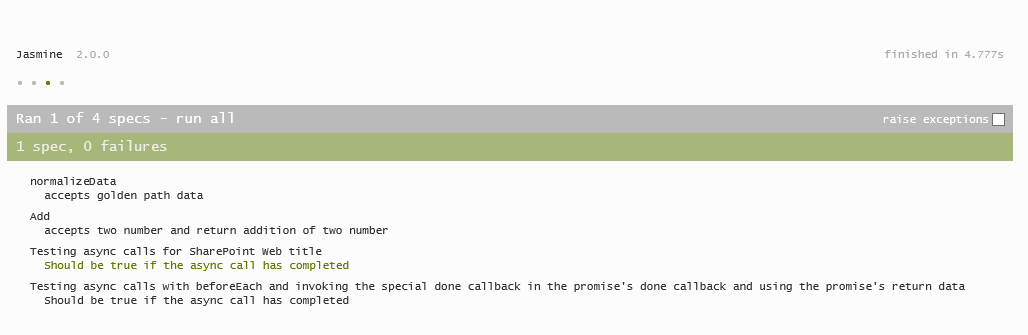
* + 1. Test case to check web title

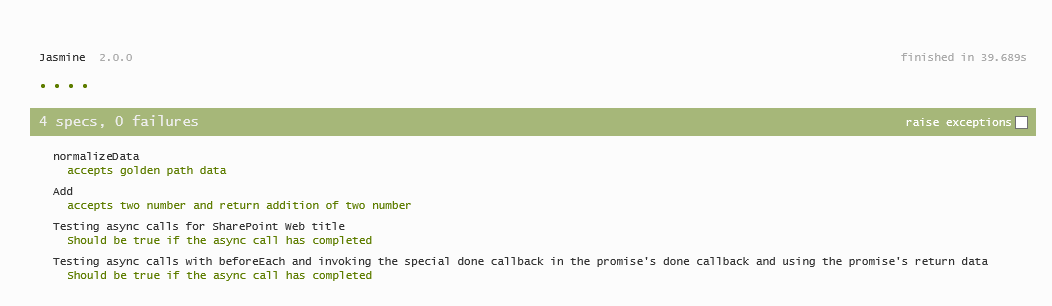


* 1. Add below DEV tag to content editor

<div id="jasmine-specs"></div>

* 1. Save the content editor web part
  2. OutPut:-





# **References**

[**https://taco.visualstudio.com/en-us/docs/unit-test-03-basic-testing/**](https://taco.visualstudio.com/en-us/docs/unit-test-03-basic-testing/)

[**https://volaresystems.com/blog/post/2014/12/09/Testing-async-calls-with-Jasmine**](https://volaresystems.com/blog/post/2014/12/09/Testing-async-calls-with-Jasmine)

[**https://jasmine.github.io/2.0/introduction.html**](https://jasmine.github.io/2.0/introduction.html)

[**https://volaresystems.com/blog/post/2014/12/09/Testing-async-calls-with-Jasmine**](https://volaresystems.com/blog/post/2014/12/09/Testing-async-calls-with-Jasmine)

[**https://blogs.msdn.microsoft.com/sharepointdev/2011/04/14/using-the-javascript-object-model-in-a-content-editor-web-part-kumar-abhishek-verma/**](https://blogs.msdn.microsoft.com/sharepointdev/2011/04/14/using-the-javascript-object-model-in-a-content-editor-web-part-kumar-abhishek-verma/)