Javascript Unit Test

February 28, 2013

Contents

1 Unit Test Frameworks

There are plenty of javascript unit test frameworks. Selecting one to use for your project is quite daunting. The purpose of this document is to help you to have a quick pick and a quick start on which javascript unit test framework to use so that writing unit test with mocking and automating your unit test become easy task. Two of the frameworks we used in AutoCAD team is Jasmine and QUnit. **Geolocation** map html dialog uses Jasmine and **New Tab Page** uses QUnit. Following sections will show you how to setup and start your unit test project quickly.

1.1 Jasmine Unit Test Framwork

1.1.1 Getting Jasmine

You can get the latest jasmine from `https://github.com/pivotal/jasmine/downloads`.

1.1.2 Setting up unit test project

Here is my own preference for creating test folder structure.

Listing 1: Folder Structure Suggestion

```
/..
/src
/lib
    /jasmine-1.3.1
        jasmine.css
        jasmine.js
        jasmine-html.js
/spec
        playerSpec.js
index.html
specRunner.html
```

Assume **/src** is your javascript implementation folder and index.html is the your html page. We need to add on folder **/lib** to contain jasmine library which you have downloaded, **/spec** to contain test code e.g **playerSpec.js**, and **specRunner.html** is entry of your test code.

Following is a boilerplate of SpecRunner.html which you can start with

Listing 2: SpecRunner.html boilerplate

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD␣HTML␣4.01␣Transitional//EN"
2    "http://www.w3.org/TR/html4/loose.dtd">
3  <html>
4  <head>
5  <title>Jasmine Spec Runner</title>
6  <link rel="shortcut␣icon" type="image/png" href="lib/jasmine-1.3.1/jasmine_favicon.png">
7  <link rel="stylesheet" type="text/css" href="lib/jasmine-1.3.1/jasmine.css">
8  <script type="text/javascript" src="lib/jasmine-1.3.1/jasmine.js"></script>
9  <script type="text/javascript" src="lib/jasmine-1.3.1/jasmine-html.js"></script>
10 <!-- include source files here... -->
11 <script type="text/javascript" src="src/Player.js"></script>
12 <script type="text/javascript" src="src/Song.js"></script>
13 <!-- include spec files here... -->
14 <script type="text/javascript" src="spec/PlayerSpec.js"></script>
15 <script type="text/javascript">
16 (function() {
17       var jasmineEnv = jasmine.getEnv();
18       jasmineEnv.updateInterval = 1000;
19       var htmlReporter = new jasmine.HtmlReporter();
20       jasmineEnv.addReporter(htmlReporter);
21       jasmineEnv.specFilter = function(spec) {
22         return htmlReporter.specFilter(spec);
23       };
24       var currentWindowOnload = window.onload;
25       window.onload = function() {
26         if (currentWindowOnload) {
27           currentWindowOnload();
28         }
29         execJasmine();
30       };
31       function execJasmine() {
32         jasmineEnv.execute();
33       }
34     })();
35 </script>
36 </head>
37 <body>
38 </body>
39 </html>
```

Line 11, 12 and 14 in Listing 2 shall be modified to include your javascript files and their specs. The sample code for **Player.js** and **Song.js** can be found in the appendix

We can start to add test for **PlayerSpec.js** as follow

Listing 3: Player.js

```
1  describe("Player", function() {
2    var player;
3    var song;
4
5    beforeEach(function() {
6      player = new Player();
7      song = new Song();
8    });
9
10   it("should␣be␣able␣to␣play␣a␣Song", function() {
11     player.play(song);
12     expect(player.currentlyPlayingSong).toEqual(song);
13
14     //demonstrates use of custom matcher
15     expect(player).toBePlaying(song);
16   });
17
```

```
18    describe("when song has been paused", function() {
19      beforeEach(function() {
20        player.play(song);
21        player.pause();
22      });
23
24      it("should indicate that the song is currently paused", function() {
25        expect(player.isPlaying).toBeFalsy();
26
27        // demonstrates use of 'not' with a custom matcher
28        expect(player).not.toBePlaying(song);
29      });
30
31      it("should be possible to resume", function() {
32        player.resume();
33        expect(player.isPlaying).toBeTruthy();
34        expect(player.currentlyPlayingSong).toEqual(song);
35      });
36    });
37
38    // demonstrates use of spies to intercept and test method calls
39    it("tells the current song if the user has made it a favorite", function() {
40      spyOn(song, 'persistFavoriteStatus');
41
42      player.play(song);
43      player.makeFavorite();
44
45      expect(song.persistFavoriteStatus).toHaveBeenCalledWith(true);
46    });
47
48    //demonstrates use of expected exceptions
49    describe("#resume", function() {
50      it("should throw an exception if song is already playing", function() {
51        player.play(song);
52
53        expect(function() {
54          player.resume();
55        }).toThrow("song is already playing");
56      });
57    });
58  });
```

describe('...', function()...) is use to declare a test suite and **it("...", function()...)** to declare a test case. Openning **SpecRunner.html** in a browser will exectue the test cases. To run the test cases in a console which is useful for continuous integration please refer to section **Automate Javascript Unit Testing**.

Jasmine provide very nice mock/spy mechanism and asynchronous support which are extremely helpful when writing your test. *For more information please check the detail on* ***http://pivotal.github.com/jasmine***

1.2 Jasmine vs QUnit

Jasmine is more preferable over QUnit for the following reasons:

- Spy/mock is not available in QUnit, in order to use spy/mock we need to use another framework(sinon-QUnit)
- Scoping differences of tests/specs within suites. Jasmine scoping in the suite is more handy for managing the code you need to run.
- Level of information displayed of assertion failure in jasmine is more informative automatically(BDD) than that of QUnit. For example :

Listing 4: Sample Jasmine Test Case

```
1  var callbackFinish = jasmine.createSpy("callback spy");
2  ...
```

```
3 expect(callbackFinish).not.toHaveBeenCalled();
```

The error displayed as 'Error: Expected spy callback spy not to have been called.'

Listing 5: Sample Qunit Test Case

```
1 var callbackFinish = sinon.spy();
2 ...
3 ok(!callbackFinish.called, "callback␣spy");
```

The error displayed is as 'callback spy'

- Creating spy/stub objects from scratch are easier to code in **jasmine** than **QUnit**

Listing 6: Sample Jasmine Test Case For Creating Spy/Mock

```
1  mockMap = jasmine.createSpyObj('mockMap',
2          ['createLocation',
3          'addPin',
4          'removeEntity',
5          'removeHandler', 'addHandler',
6          'createPoint',
7          'createInfoBox', 'addInfoBox', 'removeInfoBox',
8          'markPinStyle',
9          'getPinLocation',
10         'getHeightPin',
11         'panMapTo']);
```

Listing 7: Sample QUnit Test Case For Creating Spy/Mock

```
1  mockMap = sinon.stub({
2          createLocation : function(){},
3          addPin : function(){},
4          removeEntity : function(){},
5          removeHandler : function(){},
6          addHandler : function(){},
7          createPoint : function(){},
8          createInfoBox : function(){},
9          addInfoBox : function(){},
10         removeInfoBox : function(){},
11         markPinStyle : function(){},
12         getPinLocation : function(){},
13         getHeightPin : function(){},
14         panMapTo : function(){}
15 });
```

- Sometimes need to wrap function twice in QUnit. It's not allowed in some cases, we need to restore the mock first and then overwrite old stub.

Listing 8: Sample QUnit Test Case For Overwrite Old Stub

```
1 mockMap.addHandler.restore();
2 sinon.stub(mockMap, 'addHandler', function () {
3         mockMap.cntAddHandlerCall++;
4 });
```

Jasmine does not have this issue since it provides functions on the spy object to set which replacement function to call. It means the stub is more reusable

Listing 9: Sample Jasmine Test Case For Overwrite Old Spy/Mock

```
1 mockMap.addHandler.andCallFake(function () {
2         mockMap.cntAddHandlerCall++;
3 });
```

- QUnit does not have custom matchers support. Jasmine support custom matcher with pretty english deprived from matcher's name for error output.

Listing 10: Jasmine's custom matcher

```
this.addMatchers({
        toBeWatermarked: function () {
                return this.actual.css("opacity") == 0.5;
        },
        toBeNotWatermarked: function () {
                return this.actual.css("display") == "none" || this.actual.css('opacity') ==
        }
});
```

If there is an error with this matcher, it is displayed as 'Expected ... to be not watermarked'

- The structure of test narrative is readable and similar to Agile's user stories format.

2 Code Coverage For Javascript Unit Test

I suggest to use **JsCoverage** for code coverage. Detailed instruction can be found `http://siliconforks.com/jscoverage/manual.html`

3 Automate Javascript Unit Testing

There are many tools out there to help integrating javascript unit test with CI process. I choose to use **Headless Webkit PhantomJS** approach because it is easy to set up and it works nicely with Jasmine, QUnit, Mocha, etc. frameworks and the core module of PhantomJS and Chromium are Webkit so we do not worry much about the different of testing environment
Here is how I set it up with Jasmine unit test sample I provided previously on Window

1. Download PhantomJs `http://phantomjs.org/download.html` for window
2. Extract the zip file, and then copy **phantomjs.exe** and **examples**
   runjasmine.js to same folder as **SpecRunner.html**
3. Run **phantomjs.exe run-Jasmine.js SpecRunner.html**, basically phantom

The flow of run-Jasmine.js basically is requesting the webpage SpecRunner.html and once the html result is obtained, parse the result and put it in the console. So feel free to modify run-Jasmine.js to suit your need.

Listing 11: Player.js

```
var system = require('system');
function waitFor(testFx, onReady, timeOutMillis) {
 var maxtimeOutMillis = timeOutMillis ? timeOutMillis : 3001, //< Default Max Timeout is 3s
 start = new Date().getTime(),
 condition = false,
 interval = setInterval(function() {
     if ( (new Date().getTime() - start < maxtimeOutMillis) && !condition ) {
        // If not time-out yet and condition not yet fulfilled
        condition = (typeof(testFx) === "string" ? eval(testFx) : testFx()); //defensive code
     } else {
        if(!condition) {
          // If condition still not fulfilled (timeout but condition is 'false')
         console.log("'waitFor()' timeout");
         phantom.exit(1);
        } else {
         // Condition fulfilled (timeout and/or condition is 'true')
         //console.log("Test cases finished in " + (new Date().getTime() - start) + "ms.");
         // Do what it's supposed to do once the condition is fulfilled
         typeof(onReady) === "string" ? eval(onReady) : onReady();
```

```
          clearInterval(interval); //< Stop this interval
      }
 }}, 100); //< repeat check every 100ms
};
if (system.args.length !== 2) {
    console.log('Usage: run-Jasmine.js URL');
    phantom.exit(1);
}
var page = require('webpage').create();
// Route "console.log()" calls from within the Page context to the main Phantom context (i.e. c
page.onConsoleMessage = function(msg) {
    console.log(msg);
};
page.open(system.args[1], function(status){
    if (status !== "success") {
        console.log("Unable to access network");
        phantom.exit();
    } else {
        waitFor(function(){
            return page.evaluate(function(){
                return document.body.querySelector('.symbolSummary .pending') === null
            });
        }, function(){
            var exitCode = page.evaluate(function(){
                //Fail test cases
                var list = document.body.querySelectorAll('.results > #details > .specDetail.fa
                if (list && list.length > 0) {
                  console.log('');
                  console.log(list.length + ' test(s) FAILED:');
                  for (i = 0; i < list.length; ++i) {
                      var el = list[i],
                          desc = el.querySelector('.description'),
                          msg = el.querySelector('.resultMessage.fail');
                      console.log('');
                      console.log(desc.innerText);
                      console.log(msg.innerText);
                      console.log('');
                  }
                  return 1;
                } else {
                  console.log(document.body.querySelector('.alert > .passingAlert.bar').innerTe
                  return 0;
                }
            });
            phantom.exit(exitCode);
        });
    }
});
```

4 Appendix A

Listing 12: Player.js

```
function Player() {
}
Player.prototype.play = function(song) {
  this.currentlyPlayingSong = song;
  this.isPlaying = true;
};

Player.prototype.pause = function() {
  this.isPlaying = false;
};

Player.prototype.resume = function() {
```

```
13    if (this.isPlaying) {
14      throw new Error("song␣is␣already␣playing");
15    }
16
17    this.isPlaying = true;
18  };
19
20  Player.prototype.makeFavorite = function() {
21    this.currentlyPlayingSong.persistFavoriteStatus(true);
22  };
```

Listing 13: Song.js

```
1  function Song() {
2  }
3  Song.prototype.persistFavoriteStatus = function(value) {
4    // something complicated
5    throw new Error("not␣yet␣implemented");
6  };
```

... and here it ends.