QUnit JavaScript testing in the Enterprise

C. David Lee
david.lee@sftsrc.com

Abstract

Beyond the days when JavaScript changed minor facets of a web page, today's JavaScript creates full featured web applications. Technologies such as JSON, JQuery and AJAX move the playing field from server side to client-side with much of this functionality now delivered via JavaScript.

Programmatically unit testing this client-side code gives us the assurance that it will not break. The challenge is to implement JavaScript unit testing while keeping it relevant when server-side, client-side or HTML changes are made, and while keeping test failures visible.

What does this mean for quality? If an organization has a unit testing strategy they will detect errors quicker and easier than organizations that do not. When JavaScript unit tests are written well with code coverage then changes will no longer require hours of manual testing. Removing even a small portion of manual testing can save time and money, giving a QA organization the ability to focus on what's new rather than rehashing old tests.

SoftSource consulting has thought through how to implement JavaScript unit testing in a way that saves time, focusing on some of the drawbacks and positives of various approaches, and using custom development with the QUnit testing library.

Biography

David Lee is a Senior Software Developer at SoftSource Consulting (www.sftsrc.com), a consulting firm with a proven track record of helping companies engineer custom software solutions on the Microsoft platform. David has been developing and architecting software solutions for the past 13 years. Most of his experience has been in designing and maintaining enterprise portal applications in the financial/professional services industry. David graduated from Oregon State University with a BS in Liberal Studies.

1. Introduction – Why JavaScript Unit Testing?

Have you ever wondered how to make unit testing strategies apply to JavaScript? Technologies such as JSON, JQuery and AJAX provide the functionality via JavaScript that was once provided server-side testable code. This code is complex and rich. Through testing small "units" of functionality in an automated way this complicated JavaScript code can be made more solid, less breakable and with good test coverage make refactoring easy and inexpensive.

Programmatically unit testing client-side code gives us assurance that it will not break when changes are introduced. Those who work in an enterprise web environment know things change frequently and they may not always be aware of how or why the change took place.

In an enterprise, code originates from multiple layers (server-side, client-side), mixes of technologies (3rd party, legacy code, brand new controls) and the HTML itself can be dynamic changing once or twice on every layer. To survive all of this complexity a need for automated testing is in order. It keeps our code stable during maintenance, clear when investigated and saves time in rehashing old stuff. As we delve more and more into JavaScript, the stage for JavaScript unit testing clearly is set.

This paper has two sections. One portion of the paper will focus on some of the concepts of client-side testing and the other portion will show examples in how JavaScript testing is implemented.

1.1 Requirements for this paper… Assumptions regarding the reader

This paper is focused on the enterprise environment and primarily for those who have unit testing already well in hand. However, anyone who has a web site, who writes JavaScript code, or who has to test JavaScript web applications should strongly consider investigating JavaScript unit testing as described here.

The assumptions of this paper are that the reader first-off has a basic understanding of HTML and JavaScript. It will discuss JavasScript and HTML fluidly. An understanding of Unit testing processes, JQuery selectors [(2)] and the concepts of server-side versus client-side programming are helpful. Some Microsoft technologies are mentioned in this paper and one of example is written in Microsoft ASP.Net/C#.

1.2 In the Enterprise, some thoughts

How tests are run is often just as important as running the tests themselves. Preparing a proper unit testing strategy for your organization is only as complicated as your needs and requirements. It could be simple or it could be quite complicated. With that said, however, keeping things simple at first is usually the best strategy, then building up as the needs arises.

Here are some additional thoughts:

- piggyback upon your existing unit testing strategies.
- leverage your test or your development environment
- make sure your build process respects these tests, not placing them into production
- provide for reporting

One key to success is by adding visibility to JavaScript testing by making your test failure or success notifications visible as needed. These should not go on hidden, but rather be raised up to those who should take action.

2 Testing strategies: to HTML or not to HTML…

Will you add HTML into your JavaScript unit tests? This is not quite the simple question that some would assume it is. Some would easily say, "No way!", but HTML is so closely tied to JavaScript functionality that the waters soon become muddied and HTML may become a necessity.

2.1 A natural division between code and the UI

Experts agree that unit testing is best applied to the code that deals with logic, not to code that shows the User Interface itself. For example if you have a calculator, you generally do not try to unit test the size and position of the buttons. This should be left up to a graphic designer and made changeable. You should also not unit test the color of the display or the font size of the numbers displayed for the same reasons. These things should be left up to a design team and not fixed in code by programmers.

However, this is what HTML is. HTML is the size of the buttons, the number of buttons, the height and width of areas on the screen. The clear division between testable code and other code should say, naturally, that HTML is not testable because it ***is*** the user interface.

The problems come where there is major functionality implemented in JavaScript and where JavaScript has to rely on the fact that buttons, accordion controls, IFRAME objects and all manner of client-side HTML objects exist.

2.2 What we knew before we got started

When we looked at implementing JavaScript unit testing there were some different testing strategies considered.

Possibilities were:

1. separating JavaScript class libraries out so that they could exist completely independently of web pages
2. creating integrated testing where JavaScript is tested in the context of each page
3. copying HTML into our tests
4. creating static, never changing, test HTML

However, when we realized what we already knew, this simplified our lives. We already knew…

- **JavaScript is very close to the User Interface…** in fact it runs in the user interface. It manipulates HTML, reads from FORM fields, and is often designed to manipulate the display of web pages. Most JavaScript does something with HTML.

- **JQuery brings JavaScript even closer to the HTML…** with selectors. JQuery selectors read HTML elements from the HTML DOM. Most actions taken against JQuery is to collections of HTML objects found through selectors.

- **HTML is fluid…** and changeable. Most commonly used server-side controls create HTML and JavaScript. They might change structure right out from underneath us as programmers.

- **Designers and developers…** both change HTML markup**.** Developers may write JavaScript, but designers change the layout of HTML for design purposes. Look and functionality do not need to oppose each other, but often times the very structure of the HTML may affect hidden JavaScript

relying on specific formatting. This might lead to defects only visible by clicking through a web page.

2.2.1 An example of why it may be difficult to separate HTML from JavaScript

Here's an example:
To perform a simple task in JavaScript many use JQuery,… because it is easy. JQuery selectors, however, are really just "HTML queries" relying on HTML tags, class names and HTML ID's. The HTML itself is important to many JQuery operations.

Example: If you wanted to clean out the child tags under a certain set of parent HTML elements you could quite possibly do something like this:

```
$(".className > div").empty();
```

Basically, this says, "Give me every "<*div*>" tag that has a parent tag with a class assigned to it named "*className*"." Ouch!!! No separation here. In order to test this code we would have to have a small segment of HTML with a bunch of tags assigned the "*className*" class and that have "<*div*>" tags underneath them.

2.3 JavaScript tests may need HTML…

So, what does that mean for JavaScript testing? It means there are some underlying problems in the separation of testable code and the HTML that it changes. The HTML and the JavaScript it modifies are so closely tied together that at times JavaScript code will not work without HTML.

Our conclusion was that there is enough JavaScript that requires HTML that without HTML many of our JavaScripts will not be unit tested at all. If we were to test JavaScript without HTML then much of our code would be untested. Therefore our challenge was just how to put HTML into our JavaScript tests while honoring the fact that HTML is changeable, manipulated on a page in any number of ways. In other words we were asking, "How can we bring HTML from these pages directly into the tests we were running?"

2.4 Creating HTML ?

Now that we decided to use HTML we recognized that we had to do it "the right way!" This meant avoiding the outright creation of HTML for the tests. Instead we opted for using HTML directly from the page.

The reason we avoided creating HTML is illustrated in a simple example:

Suppose we copied a simple accordion control out of our web pages and pasted it into a test. Our tests were great and ran efficiently, always passing. Sometime later, when a programmer or designer came along to enhance the page (perhaps changing tags to tags) the copy of the accordion in the test will not get updated! This change introduced a problem that would go completely unnoticed by the unit tests because the tests were not aware of the problem!

> Static HTML can become old and irrelevant.

To all concerned, the updates were done and still the tests were passing, but the obvious problem was that we created static HTML in the first place. If you've ever studied for an exam and found out later that you studied for the wrong one… you now know why copying and pasting HTML into a test is a bad thing! Testing an old and stale copy is not the same as testing the real thing!

2.5 Directly testing the page?

The logical path we were following was taking us to someplace that was fairly unique.

Nowhere else in web development are we asked to:

1. create unit tests,
2. however, include a snippit of the actual UI of the page in those tests

This was now different than anyone had anticipated. The problem is that we now lived under one more constraint if we were to include actual HTML markup in our tests.

> We had to not impact the current web pages themselves!
> We had to keep the behavior, look and feel exactly the same.

We knew that our tests should have little to no impact on the page against which they ran.

For example, if an accordion control was to have a certain message when opened. Any tests that ran should not modify that message or change the message formatting in any way. So, our challenge was to create a process that used real HTML (from the page), but did not change that HTML on the page. It should not impact the page or cause undesired behavior.

What we chose to do was place Unit tests in a hidden IFRAME, and to put a small status dialog on each page that lets us know the state of each test. (See section 5)

3 What is QUnit?

In order to run tests first we need a framework in which to run them.

What is QUnit? To quote http://docs.jquery.com/Qunit,…

> "QUnit is a powerful, easy-to-use, JavaScript test suite. It's used by the jQuery project to test its code and plugins… [It] is capable of testing any generic JavaScript code."

To understand QUnit please look at http://docs.jquery.com/Qunit for a full API reference, but to get a quick preview of its capabilities…

3.1 QUnit Setup

- module(name, lifecycle)
  Separate tests into modules.

- test(name, expected, test)
  Add a test to run.

- asyncTest(name, expected, test)
  Add an asynchronous test to run. The test must include a call to start().

- expect(amount)
  Specify how many assertions are expected to run within a test.

3.2 QUnit Test Assertions

QUnit has several built-in functions that allow you to assert when something is true or false, therefore making your tests pass or fail.

Assertions possible with QUnit:

- ok(state, message)
  A boolean assertion, equivalent to JUnit's assertTrue. Passes if the first argument is true.

- equal(actual, expected, message)
  A comparison assertion, equivalent to JUnit's assertEquals.

- notEqual(actual, expected, message)
  A comparison assertion, equivalent to JUnit's assertNotEquals.

- deepEqual(actual, expected, message)
  A deep recursive comparison assertion, working on primitive types, arrays and objects.

- notDeepEqual(actual, expected, message)
  A deep recursive comparison assertion, working on primitive types, arrays and objects, with the result inverted, passing when some property isn't equal.

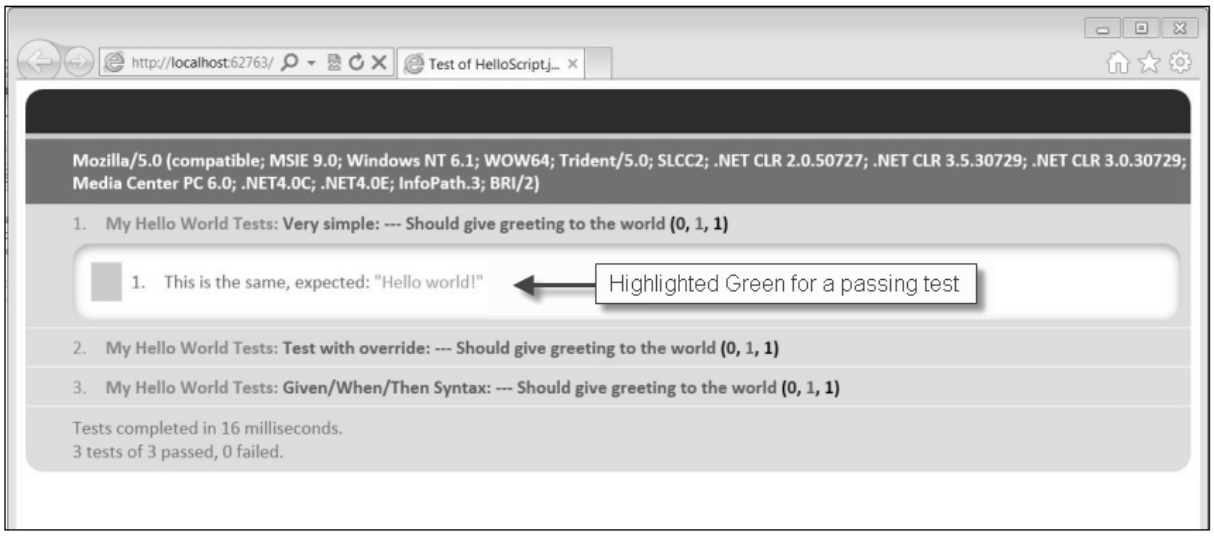
- strictEqual(actual, expected, message)
  A stricter comparison assertion then equal.

- notStrictEqual(actual, expected, message)
  A stricter comparison assertion then notEqual.

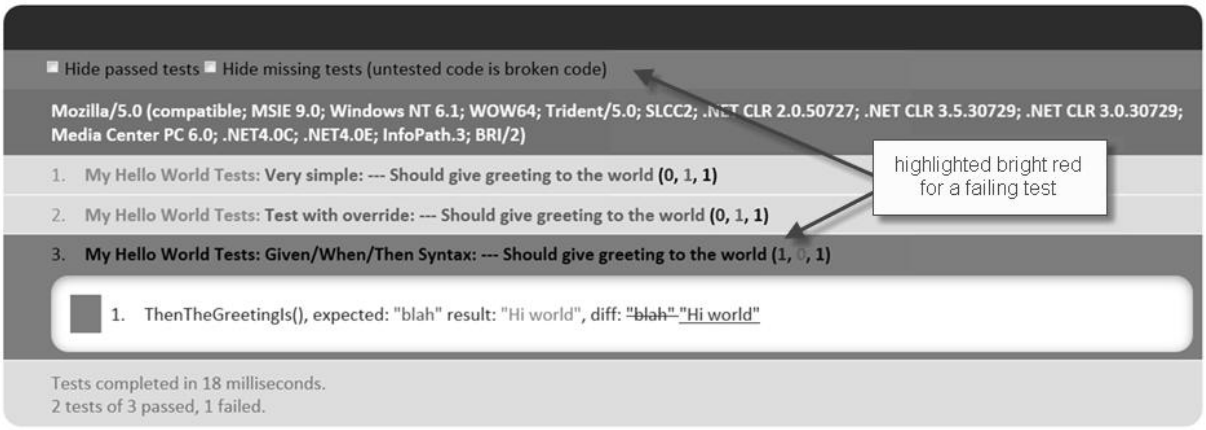
- raises(block, expected, message)
  Assertion to test if a callback throws an exception when run.

3.3 The Test Runner

3.3.1 Image of a passing test



3.3.2 Image of a failing test



4 Some examples of testing with QUnit

4.1 A Basic Test

This is a very simple example of testing with QUnit. In this example the test simply checks the output and asserts true if it is what is expected.

4.1.1 Web Page

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Sample Production page with external JavaScript.</title>
    <script src="HelloScript.js" type="text/javascript" language="javascript"></script>
</head>
<body>
<h2>Sample Production page with external JavaScript.</h2>

<div style="font-style:italic">
<script language="javascript" type="text/javascript">
    var greeting = GetHello();
    document.write(greeting);
</script>
</div>

<div style="padding-top:5px;">
    <div><a href="TestPage.htm">Click here</a> to view the tests for this page</div>
</div>

</body>
</html>
```

4.1.2 Script to be tested: HelloWorld.js

```
var helloWorld = "Hello world!";

function GetHello() {
    return helloWorld;
}
```

4.1.3 Test Runner Page

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

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

```
<head>
    <title>Test of HelloScript.js JavaScript file</title>
    <link type="text/css" href="../Common/qunit/qunit.css" rel="stylesheet" />

    <script type="text/javascript" language="javascript"
    src="../Common/scripts/jquery-1.4.1.js"></script>

    <script type="text/javascript" language="javascript"
    src="../Common/qunit/qunit.js"></script>

    <script src="HelloScript.js" type="text/javascript" language="javascript"></script>

</head>
<body>
<script type="text/javascript" language="javascript" >

    module("My Hello World Tests");

    test("Very simple:  ---  Should give greeting to the world", function () {

        var helloTest = GetHello()
        same("Hello world!", helloTest, 'This is the same');

    });

</script>
    <div>
                    <h1 id="qunit-header"></h1>
                    <h2 id="qunit-banner"></h2>
                    <div id="qunit-testrunner-toolbar"></div>
                    <h2 id="qunit-userAgent"></h2>
                    <ol id="qunit-tests"></ol>
                    <div id="qunit-fixture">
                           test markup, will be hidden
                           <div id="testHtml" style="display:none;" ></div>
                    </div>
    </div>
</body>
</html>
```

4.1.4 Output of test results

4.2 A test the override of a variable

To build upon the 1st example… sometimes it is appropriate to "mock" or "spoof" a variable or a class for testing purposes. This is very easy in JavaScript as seen in this example.

4.2.1 Unit test script to be placed in the test runner page

```
module("My Hello World Tests");

test("Test with override:  ---  Should give greeting to the world", function () {

    helloWorld = "Hi world!"
    var helloTest = GetHello()
    same("Hi world!", helloTest, 'This is the same');

});
```

4.2.2 Output of test results

4.3 Testing using a typical "Given/When/Then" testing syntax

Also building upon the previous examples… we can use good formatting. Today's testing, in many cases, uses a syntax called "Given/When/Then". This syntax is just as easy in JavaScript as it is in C# or any other server-side code.

4.3.1 Test Runner Page

```
test("Given/When/Then Syntax:  ---  Should give greeting to the world", function () {

    GivenTheGreetingIs('Hi world');
    WhenTheGreetingIsRequested();
    ThenTheGreetingIs('Hi world');

});


var _returnedGreeting;
```

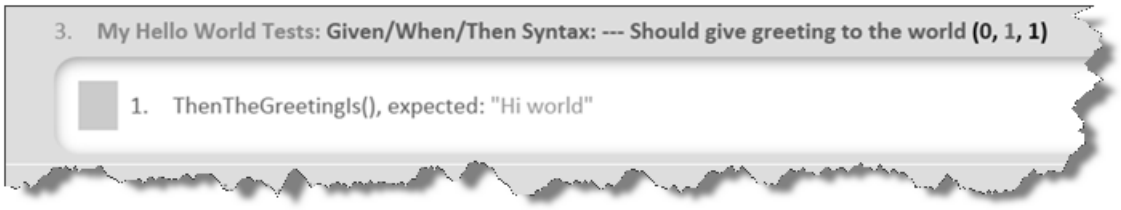
```
function Setup() {
}

function GivenTheGreetingIs(greeting) {
    helloWorld = greeting;
}

function WhenTheGreetingIsRequested() {
    _returnedGreeting = GetHello();
}

function ThenTheGreetingIs(greeting) {
    same(_returnedGreeting, greeting, "ThenTheGreetingIs()");
}
```

4.3.2 Output of test results



5 Implementation of JavaScript testing using our strategy

Our compromise, as mentioned in the end of section 2, was to include fresh HTML in our tests that is quickly copied out of the original page at the start of each test run. It runs tests after the page load has completed, placing HTML into a hidden IFRAME which runs our tests. Following test completion the status is reported back to a dialog.

Written in C# and ASP.Net, this example includes all of the strategies mentioned earlier.

Our solution strategy included 4 parts:

1. **Start from a codebase used site-wide:** To make the tests site-wide we wired our tests into a site-wide base class. This code verifies that it is only run in the development environment.

2. **Automatic wiring:** The test runner automatically shows up when a test file is put into a directory following the correct naming convention.

3. **Minimal footprint:** The test runner has a very low impact to the original page. It creates and destroys a *hidden* IFRAME that runs the tests. Then the results of the test are displayed in a tiny box in the upper left-hand corner.

4. **Easy to create tests:** To create a test the developer must simply create test runner page with the correct naming convention, using the provided test-runner master page.

5.1 The initial setup

A developer will need to take the sample code provided to integrate it into your site.

- We used a base class (*in the provided code samples it is a master page*) that includes both the JQuery script library and includes this code added to the OnPreRender event.
- Our production pages inherit from this base class (*again, in the test samples we are using a master page, but a similar concept*).
- Every path must be correct and changed to accommodate your site.

5.1.1 Example: Passing test

The result of the sample test is a small dialog showing success or failure. This shows at the top left-hand corner. By clicking "(show)" the testing dialog shows the QUnit test runner in an IFRAME.

5.1.2 Example: Failing test

When even one test is failing the test dialog shows as red. By clicking "(show)" you can also see failing tests.

5.2 The "using" namespace

In our example we created a JavaScript library for including things into our tests. This library is named "using." It load scripts HTML into a test automatically, based on the JQuery selector specified..

5.3 The "using" Class

- using.SelectedHtml(jquerySelector)
  Uses JQuery selectors to find HTML *on the parent page* and load it into the test page.

5.4 To implement a test using this strategy

Take a look at the sample code provided. Once this sample code is integrated into your site then the process of creating a test is fairly simple:

1.) Create the correct directory path for your tests
2.) Create an ASP.Net .aspx page in this new directory (following a naming convention)
3.) place your tests in the .aspx page.

6 Some Additional Testing Ideas

Here are some JavaScript testing ideas, thoughts on how you could use QUnit. These don't have corresponding examples in the code provided, but they are submitted as some food for thought.

1. **Make sure your web services are called** – By mocking up the return from JSON calls you could validate that your JSON return methods are coded correctly.

```
// loads the parent page's scripts into memory
using.ScriptsFromParentPage();

// a QUnit command to create a testing module
module("HealthConcierge", { setup: Setup });

test("Should Verify Autocomplete Webservice was called", function () {

    GivenAFakeSearchPrefix('fakeSearchPrefix');
    WhenAutoCompleteIsInvoked();
    ThenAutoCompleteServiceWasCalledWith('fakeSearchPrefix');
    ThenAutoCompleteServiceReturnedResults(_mockResults);

});


function GivenAFakeSearchPrefix(searchPrefix) {
    _searchPrefix.term = searchPrefix;
}

function WhenAutoCompleteIsInvoked() {
    _webService.CallService(_searchPrefix, TestResultsFunction);
    _mockedWebServiceOptions.success({
        d: _mockResults
    });
}

function ThenAutoCompleteServiceWasCalledWith(searchPrefix) {
            same(_mockedWebServiceOptions.data, "{\"searchText\":\"" +
searchPrefix + "\"" + ",\"numberOfTermsToReturn\":\"" + "20" + "\"}",
"ThenAutoCompleteServiceWasCalledWith('" + searchPrefix + "')");
}

function ThenAutoCompleteServiceReturnedResults(results) {
    same(_resultFromWebServiceCall, results,
"ThenAutoCompleteServiceReturnedResults('" + results + "')");
}



function Setup() {
    jQuery.ajax = function (param) {
        _mockedWebServiceOptions = param;
    }

    _webService = new WebService();
}
```

```
var _webService;
var _mockedWebServiceOptions = null;
function _searchPrefix() { term: null };
var _resultFromWebServiceCall;
var _mockResults = 'foo';

var TestResultsFunction = function (message) { _resultFromWebServiceCall =
message; }
```

2. **Validate the return from JavaScript functions –** QUnit is very similar to NUnit, so most of the assertions you are used to making can be made in JavaScript too.

```
// loads the parent page's scripts into memory
using.ScriptsFromParentPage();

module("PublicPage");

test("Is it Four?", function () {
        WhenTheCountIsCalled();
        ThenTheResultIsFour();
});

function WhenTheCountIsCalled() {
        countResult = new CountResults().Count();
}

function ThenTheResultIsFour() {
        same(countResult, 4, "ThenTheResultIsFour");
}


var countResult = 0;
```

3. **Validate JavaScript against Mocked objects –** JavaScript is very good at mocking objects because nothing is strongly typed.

```
using.Script("/HealthConcierge/ConditionResults/TreatmentCosts.js");

module("TreatmentCosts");

test("Should Call Counts", function () {
        WhenTreatmentCostsResultsAreRendered();
        ThenCountsAreRendered();
        ThenPresentIsCalled();
});
```

```
function WhenTreatmentCostsResultsAreRendered() {
        var treatmentCost = new ConditionResults.TreatmentCosts();
        treatmentCost.presenter = new MockPresenter();
        treatmentCost.Present(null);
}

function MockShowCount(resultLocationElement, results) { countMockCalled =
true; }
function MockPresent(treatmentCosts, template, resultLocationElement) {
presentMockCalled = true; }

function MockPresenter() { }
MockPresenter.prototype.ShowCount = MockShowCount
MockPresenter.prototype.Present = MockPresent;

function ThenPresentIsCalled() {
        ok(presentMockCalled, "ThenPresentIsCalled");
}

function ThenCountsAreRendered() {
        ok(countMockCalled, "ThenCountsAreRendered");
}

var presentMockCalled = false;
var countMockCalled = false;
```

4. **Validate objects in the page** – For example, if your scripts rely on several HTML objects in your page you could test to make sure these objects exist

```
// a QUnit command to create a testing module
module("HealthConcierge", { setup: Setup });


test("Does this HTML exist?", function () {

    DoesThisExist("body"); // the body tag

    DoesThisExist("#testHtml"); // a tag with a “testHtml” id

    DoesThisExist("#results > div"); // a tag with a “results” id with <div> tags
under it

});



function DoesThisExist(jQuerySelector) {
    ok($(jQuerySelector, window.parent.document).length, " DoesThisExist (" +
jQuerySelector + ")");
}
```

```
function Setup() {}
```

5. **Validate the loading of other scripts** – sometimes scripts get renamed or deleted. You could validate the objects instantiated by any other JavaScript reaching out to the parent page if desired.

```
// loads the parent page's scripts into memory
using.ScriptsFromParentPage();

// a QUnit command to create a testing module
module("HealthCareProvider", { setup: Setup });


test("Should Have ConditionResults Presenters", function () {

    ThenThisPresenterIsLoaded(parent.ConditionResults.Conditions,
"ConditionResults.Conditions");

    ThenThisPresenterIsLoaded(parent.ConditionResults.Discussions,
"ConditionResults.Discussions");
    ThenThisPresenterIsLoaded(parent.ConditionResults.DoctorQuestions,
"ConditionResults.DoctorQuestions");

    ThenThisPresenterIsLoaded(parent.ConditionResults.Drugs,
"ConditionResults.Drugs");
    ThenThisPresenterIsLoaded(parent.ConditionResults.HospitalQuality,
"ConditionResults.HospitalQuality");

    ThenThisPresenterIsLoaded(parent.ConditionResults.Provider,
"ConditionResults.Provider");
    ThenThisPresenterIsLoaded(parent.ConditionResults.TreatmentCosts,
"ConditionResults.TreatmentCosts");
});



function ThenThisPresenterIsLoaded(presenter, presenterName) {
    ok(presenter, "ThenThisPresenterIsLoaded(" + presenterName + ")");
}

function Setup() {}
```

7 Conclusion

JavaScript testing is not a necessity, but it can save you time and money as you properly plan an implement a JavaScript testing strategy. With the proper infrastructure and a few simple steps, programmers can add JavaScript tests to any page.

Definition of Terms

- HTML — HyperText Markup Language – the building blocks for web pages
- HTML DOM — the document object model of an HTML document; basically, the structure and objects within the document.
- Markup — In this paper the terms “markup” and “HTML” are used interchangeably, however, markup language itself is defined as a way to annotate text through using a specific syntax. (for example, specifying the creation of a table through using the syntax <table></table>
- JavaScript — JavaScript is a prototype-based, object-oriented scripting language that is dynamic and weakly typed.
- C# — (pronounced see sharp) is a programming language developed by Microsoft encompassing strong typing, object-oriented (class-based), and component-oriented programming.
- ASP.Net — ASP.NET is a web application framework developed and marketed by Microsoft to allow programmers to build dynamic web sites, web applications and web services.
- jQuery — jQuery is a cross-browser JavaScript library designed to simplify the client-side scripting of HTML. It was released in January 2006 and is used in over 46% of the most visited websites.
- UI — User Interface
- IFRAME — The IFRAME tag is an HTML markup designation that creates a frame. Frames allow a visual HTML Browser window to be split into segments, each of which can show a different document.

References

1. Code samples referenced here are found at: http://sftsrc.com/content/docs/PNSQC2011_QUnit.zip

2. www.jquery.com (JQuery)

3. http://api.jquery.com/category/selectors/ (JQuery Selectors)