# jQuery Test Driven Development

## Introduction

This article is a journey on how you might be able to implement Unit Tests for your JavaScript code. More specifically, we will be addressing how to write a jQuery Plugin using the Test Driven Development paradigm.

Many of you may be used to Unit Testing in other languages such as .NET, Ruby, Java, etc, but it doesn't seem to be as frequent practice in the JavaScript world.

Since more client-code is now being placed on the browser to support richer UI experience and better interaction with users, it makes sense that I would need to have Unit Test coverage for it as well.

## Test Driven Development

Test Driven Development (TDD), which is the process in which one writes tests before they actually write the code for their application. [Jeffery Palermo](http://twitter.com/jeffreypalermo) wrote a good introduction article on MSDN entitled [Guidelines for Test-Driven Development](http://msdn.microsoft.com/en-us/library/aa730844(VS.80).aspx). In that article he lays out the basic process of what TDD looks like. I'll summarize those steps from his post here, but I encourage you to read his full article.

1. Understand the scope of your feature or requirement
2. Create a test that you know will fail

* Call a method that doesn't exist or behave correctly yet
* This is key because you don't want your tests to pass by accident

1. Make your failed test pass by adding a minimal amount of code

* Change the code so that it will pass, but don't go overboard
* Writing minimal code helps enforce to not write things you don't need

1. Modify your code to make any improvements necessary

* This is where you can refactor into methods, rename variables, re-design, etc...

## Tools of The JavaScript Trade

In order to Unit Test your code effectively you need some tools to help you along. There are many different JavaScript tools available for you to choose from, but after quite a bit of research I have settled down to these three different tools.

### Unit Testing Tool

Since most of the JavaScript work I do is jQuery related, it made sense for me to pick [QUnit](http://docs.jquery.com/QUnit) as my Unit Testing tool. It was developed by [John Resig](http://twitter.com/Jeresig) and it was written to support the testing of both general JavaScript and also jQuery.

### Mocking Tool

I reviewed numerous Mocking frameworks and after a lot of research was able to find the [QMock](http://github.com/andybeeching/qmock) project that was able to suite most of my needs as a Mocking and Stubbing framework. I found that most of the Mocking framework either didn't support Stubbing or that if they did support Stubbing that they didn't' support callback methods, which is a highly used in jQuery. I had to make some minor changes to the code base to support the [jQuery.ajax()](http://api.jquery.com/jQuery.ajax/) method. I plan to commit my code as a patch on the [GitHub](http://github.com/andybeeching/qmock) project.

### Code Coverage Tool

As I was writing Unit Tests I began to get curious as to the Test Coverage I was having in my Unit Tests. In other words, I wanted to know how much of my project was being tested by my Unit Tests. I found a couple of tools for this, but the [JSCoverage](http://siliconforks.com/jscoverage/) tool has worked best for my needs. The examples I show later in this article will use this tool.

## Sample jQuery Plugin

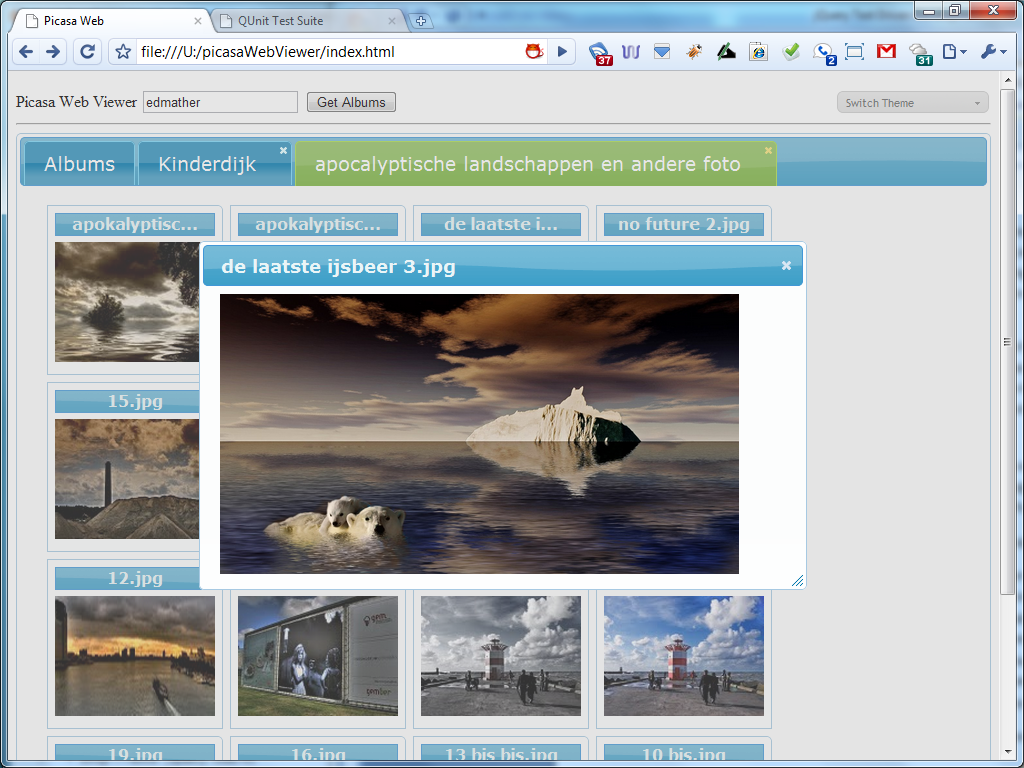
As a sample project to test these concepts I chose to develop a simple little jQuery Plug-in that is a Album and Photo viewer for Google's Picasa Web Albums. Since Google exposes JSONP support in their API it enables us to use jQuery AJAX to retrieve album and photo information.

Here are some basic features that the jQuery Plugin should support

* Default options that can be overridden by the user
* Should first should display all the public albums for the given user name inside a tab
* If the user clicks one of the albums, then a new tab will be created with the album's name as title and all of those public photos will be displayed in the new tab
* If the user clicks on one of the photos from an album, then a large version will be displayed in a modal dialog that can be resized and moved around
* The user should be able to close an album's tab by clicking a close image

**Note**: The plug-in will have a dependency on the jQuery UI library to enable tabs and modals.

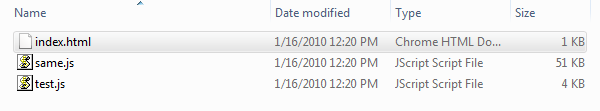
The following is a screenshot of what the jQuery Plug-in will look like after development.



### Setting Up Our Unit Tests

Since we are going to demonstrate TDD, we are going to first focus on setting up our Unit Tests and then branch out from there. If you don't already have QUnit installed on your box you can go to the [jQuery/QUnit GitHub Repository](http://github.com/jquery/qunit) and click the "Download Source" button to get the Zip or Tar.

After you extract the archived file, you'll notice two folders: qunit and test. The QUnit folder contains the QUnit framework and supporting CSS styles and the Test folder contains the test runner index.html and two JavaScript test files.



If you open the index.html file in your browser it will look something like this...



Now, open the "index.html" file in a text editor and remove the existing "test.js" and "same.js" scripts and replace them with all of your dependencies such as...

* jQuery Library
* jQuery UI Library
* QMock Library
* Picasa Web Viewer jQuery Plugin (empty file)
* Picasa Web Viewer Tests (empty file)

<!DOCTYPE html>

<html>

<head>

<title>QUnit Test Suite</title>

<link rel="stylesheet" href="../qunit/qunit.css" type="text/css" media="screen">

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.min.js"></script>

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-ui.min.js"></script>

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

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

<script type="text/javascript" src="jquery.picasawebviewer.js"></script>

<script type="text/javascript" src="jquery.picasawebviewer.tests.js"></script>

</head>

<body>

<h1 id="qunit-header">QUnit Test Suite</h1>

<h2 id="qunit-banner"></h2>

<div id="qunit-testrunner-toolbar"></div>

<h2 id="qunit-userAgent"></h2>

<ol id="qunit-tests"></ol>

</body>

</html>

Let's start our jQuery.PicasaWebViewer.Tests.js page by checking the values of the default options

module("Picasa Web Viewer");

test("Default Options", function() {

same($.picasaWebViewer.defaultOptions.urlFormat, 'http://picasaweb.google.com/data/feed/api/user/{0}?alt=json-in-script');

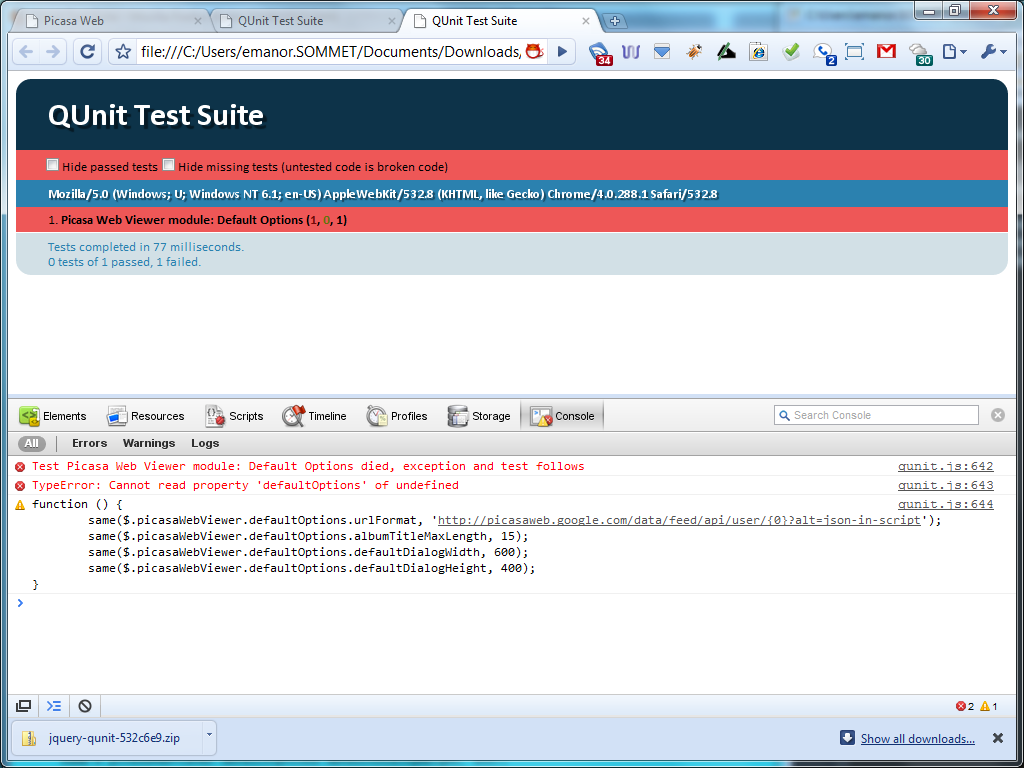
same($.picasaWebViewer.defaultOptions.albumTitleMaxLength, 15);

same($.picasaWebViewer.defaultOptions.defaultDialogWidth, 600);

same($.picasaWebViewer.defaultOptions.defaultDialogHeight, 400);

});

At this point, we don't have a jQuery Plug-in called PicasaWebViewer or any public methods that we can call, so if we ran the above tests we would get a "Red" result meaning our tests failed.



Now our task is to make our tests pass using the least effort possible. So, let's define the namespace and default options in our jQuery Plug-in.

(function($) {

var picasaWebViewer = $.picasaWebViewer = {};

picasaWebViewer.defaultOptions = {

urlFormat : "http://picasaweb.google.com/data/feed/api/user/{0}?alt=json-in-script",

albumTitleMaxLength : 15,

defaultDialogWidth : 600,

defaultDialogHeight : 400

};

$.fn.picasaWebViewer = function(options) {

return this.each(function() {

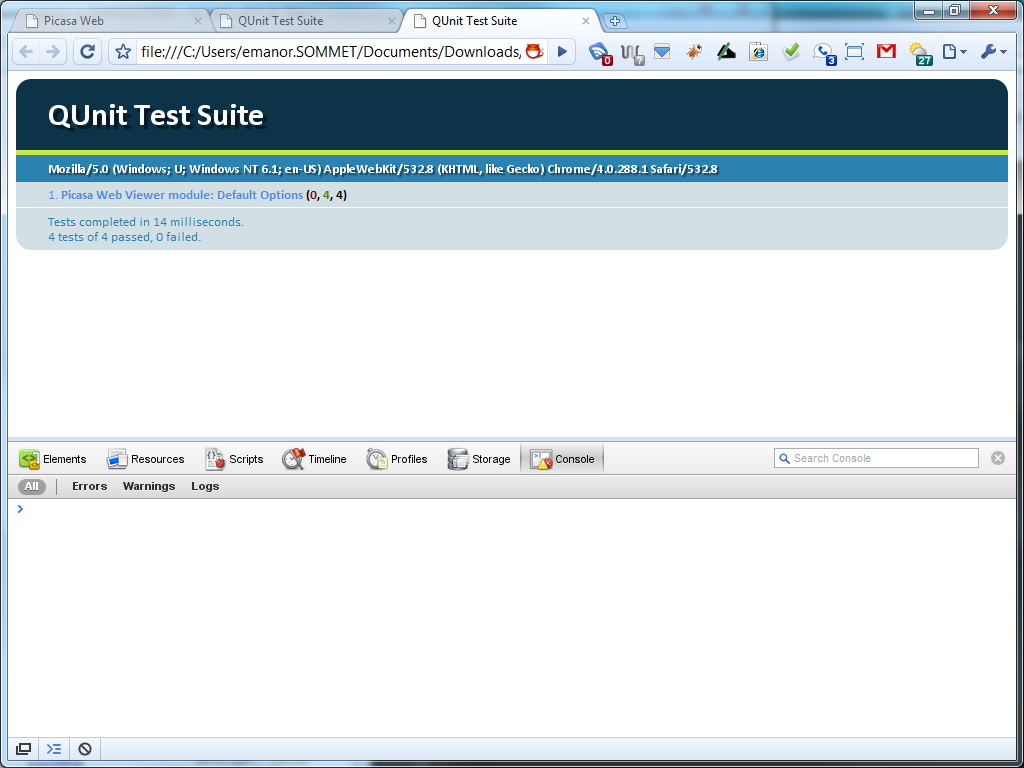
});

};

})(jQuery);

If we run our tests again we will get a green success bar returned from QUnit. At this point, there really isn't much to refactor, so let's focus on some other new tests.

Note: You will notice that I have made all of the functions in my jQuery plug-in public in order to facilitate Test Driven Development. This is not always required, but it does make things easier to fully Unit Test a piece of code. This can be a debated topic, but this article isn't intended to address those arguments.



We have shown that there are default values for our plug-in, but now let's check if we can override them. In this test we will call an "overrideOptions()" function that will override any default option to be used in the plug-in.

test("Override Deafult Options", function() {

$.picasaWebViewer.overrideOptions({

urlFormat : 'http://www.google.com',

albumTitleMaxLength : 25,

defaultDialogWidth : 400,

defaultDialogHeight : 300,

userName : 'BillGates'

});

same($.picasaWebViewer.options.urlFormat, 'http://www.google.com');

same($.picasaWebViewer.options.albumTitleMaxLength, 25);

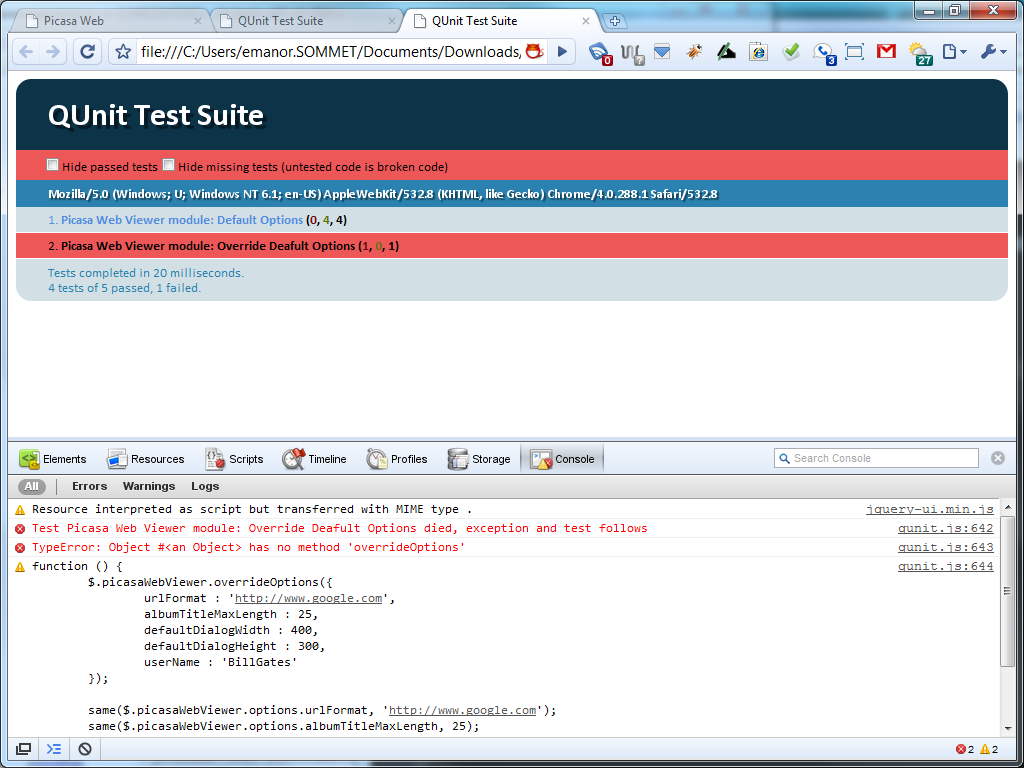
same($.picasaWebViewer.options.defaultDialogWidth, 400);

same($.picasaWebViewer.options.defaultDialogHeight, 300);

same($.picasaWebViewer.options.userName, 'BillGates');

});

As you could have probably guess, you'll see in the next screenshot that our tests come back as failed because we haven't defined the "overrideOptions" function yet.



To get our tests to pass we need to define the method and put a little logic in it to override the default options.

(function($) {

var picasaWebViewer = $.picasaWebViewer = {};

picasaWebViewer.defaultOptions = {

urlFormat : "http://picasaweb.google.com/data/feed/api/user/{0}?alt=json-in-script",

albumTitleMaxLength : 15,

defaultDialogWidth : 600,

defaultDialogHeight : 400

};

picasaWebViewer.options = null;

picasaWebViewer.overrideOptions = function(options) {

picasaWebViewer.options = $.extend(

{},

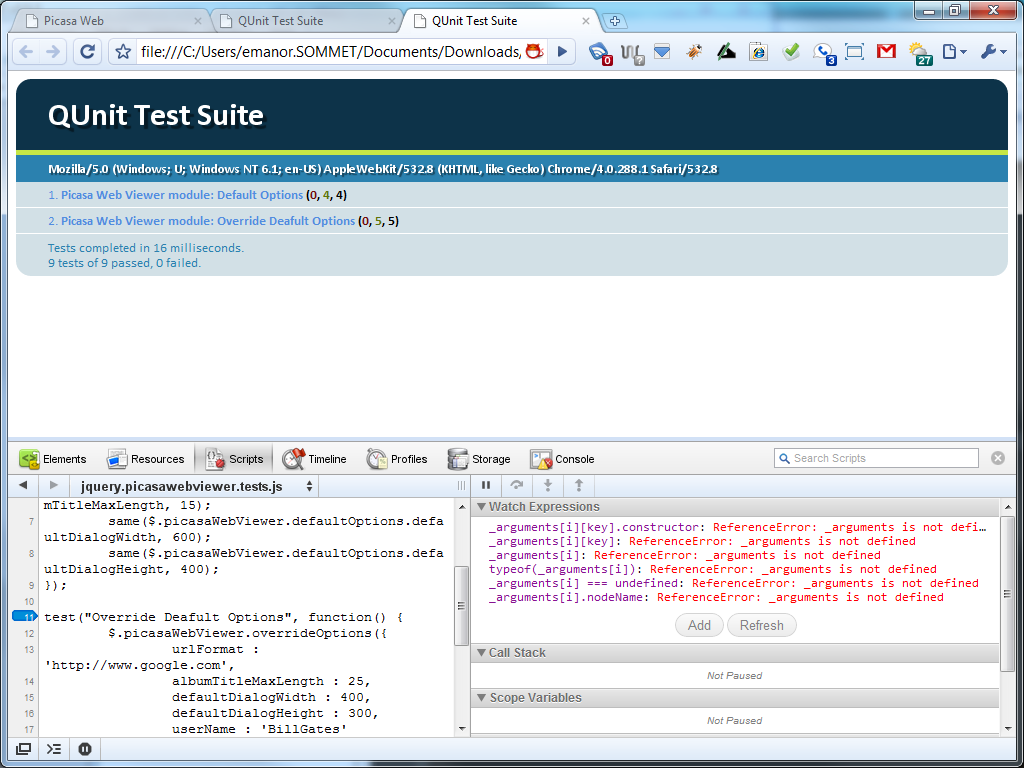
picasaWebViewer.defaultOptions,

options);

};

})(jQuery);

If we rerun our unit tests, you'll see, as in the next screenshot, the nice happy green bar of success.



Don't worry, I'm not going to keep making all of these failed and passed screenshots through the whole article. I just wanted to give you an idea of what you could expect from a TDD process.

You might have noticed that our code really isn't even a jQuery Plug-in yet. We have a namespace defined with some public properties and functions, but no Plug-in itself. So, let's make another test that calls the jQuery Plug-in and tests that the options passed to it override any default options.

test("Calling Plugin Override Options", function() {

$("#targetId").picasaWebViewer({

userName : "elijah.manor"

});

same($.picasaWebViewer.options.urlFormat, 'http://picasaweb.google.com/data/feed/api/user/{0}?alt=json-in-script');

same($.picasaWebViewer.options.albumTitleMaxLength, 15);

same($.picasaWebViewer.options.defaultDialogWidth, 600);

same($.picasaWebViewer.options.defaultDialogHeight, 400);

same($.picasaWebViewer.options.userName, 'elijah.manor');

});

We will also need to add an element to our index.html test runner with id of "targetId" for our plug-in to use. I went ahead and added "display: none" to the target element mainly so that the test results would not be disturbed.

<body>

<h1 id="qunit-header">QUnit Test Suite</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-target" style="display: none;"></div>

</body>

Running the above tests will fail of course, because we haven't defined the jQuery Plug-in yet. So, let's do that now and call the "overrideOptions" function.

$.fn.picasaWebViewer = function(options){

picasaWebViewer.overrideOptions(options);

return this;

};

The above code will make our tests pass, but it doesn't really look like much of a plug-in yet, does it? Well, as we continue to flush out the tests and functionality it will look more and more like a plug-in.

There is an important key to notice at this point in our testing. If I were to make another set of tests that used one of those options we've been testing I may get a result that I didn't intend. The problem is that we haven't been resetting our plug-in state between tests. So, our next step will be to add a teardown function to our module.

module("Picasa Web Viewer", {

setup: function() {

$.picasaWebViewer.overrideOptions({});

},

teardown: function() {

}

});

You can pass a module both a setup and teardown function that will be ran between each of your tests inside the module. We aren't using the teardown function yet, but we will later in our tests. In our setup function we are going to restore our options back to their default values in case we test one of our public functions directly. Technically, by adding the setup and teardown function already I have violated the TDD principles, but they are mostly guidelines and each developer tends to modify them to suit their needs.

Okay, it's about time we do something a little more exciting than checking for default options. The whole point of this plug-in is to show albums and pictures, so let's now focus on building up the proper HTML structures to house the albums and pictures.

var targetId = "#qunit-target";

test("Scafford Gallery", function() {

$.picasaWebViewer.scaffoldGallery($(targetId));

ok($(targetId).find('#gallery').length, 'Gallery created');

ok($(targetId).find('#tabs').length, 'Tabs created');

});

Since we will continue to need an HTML element to test with, I went ahead and pulled out the targetId out into its own variable so we can use it in the above test and future tests.

var tabs, gallery,

picasaWebViewer = $.picasaWebViewer = {};

picasaWebViewer.scaffoldGallery = function(element) {

var html =

"<div class='demo ui-widget ui-helper-clearfix'>" +

"<div id='tabs'>" +

"<ul>" +

"<li><a href='#tabs-0'>Albums</a></li>" +

"</ul>" +

"<div id='tabs-0'>" +

"<ul id='gallery' " +

"class='gallery ui-helper-reset ui-helper-clearfix' />" +

"</div>" +

"</div>" +

"</div>";

tabs = $(html).appendTo(element).find('#tabs').tabs();

gallery = $('#gallery');

};

Since we have added elements to the test runner DOM. We will need to clean this up for any further tests. This can easily resolve this by adding a simple jQuery function to clear out all DOM elements from our target element in the test runner.

module("Picasa Web Viewer", {

setup: function() {

$(targetId).empty();

$.picasaWebViewer.overrideOptions({});

},

teardown: function() {

}

});

Here comes the fun and often uncharted territory for many JavaScript developers. We are going to delve into Mocking. I like the way [Roy Osherove](http://twitter.com/RoyOsherove) put it in one of his [later blog posts](http://weblogs.asp.net/rosherove/archive/2005/06/30/416872.aspx)

Mocks are **spies in disguise**for you tests - double agents. they let you do what you want without your real code knowing about it, and tell you everything that happened to them like"your class \*should\* have called my 'authenticate' method with param x and Y but it actually called it with the wrong value.. Your test should \*fail\*".

As I mentioned previously, I will be using QMock to do JavaScript mocking. The following code is testing that when the "getAlbums" function is called and no results are returned that a call to "displayAblums" will not happen as well. There is also a test that the "scaffoldGallery" function should have been called. Since we tested that public function earlier, there is no need to test that again. We just want to know if it would have been called.

test("GetAndDisplayAlbums Returns Nothing so No Albums Displayed", function() {

var target = $(targetId)[0];

var mockRepository = new Mock();

mockRepository

.expects(1)

.method('getAlbums')

.withArguments(Function)

.callFunctionWith(null);

var mockPicasaWebViewer = new Mock();

mockPicasaWebViewer

.expects(1)

.method('scaffoldGallery')

.withArguments(target);

mockPicasaWebViewer

.expects(0)

.method('displayAlbums');

$.picasaWebViewer.setRepository(mockRepository);

$.picasaWebViewer.setPicasaWebViewer(mockPicasaWebViewer);

$.picasaWebViewer.getAndDisplayAlbums(target);

ok(mockRepository.verify(), 'Verify getAlbums was called');

ok(mockPicasaWebViewer.verify(), 'Verify displayAlbums was not called');

});

There are a couple things to note in the above code

1. When I am mocking the "getAlbums" function I am passing a Function as the argument. This represents that callback function that will be called with whatever is passed to the "callFunctionWith" function.
2. You might have noticed that I have introduced the concept of a Repository. I have done this for functions that are going to be using the AJAX functions off of jQuery. I don't want to actually make the AJAX request since it is an external dependency, so I am pushing these off to another variable so that I can mock them separately. We will cover an example of how to do that later in the article.
3. I am calling to set functions that are overriding the default behavior of the jQuery Plug-in. Instead of using the defined instances inside the Plug-in I am replacing them with these mock objects that are designed to record their behavior and return whatever I tell them. We will need to write these set function in our Plug-in.

To make this test pass, we'll need to write a new function in our jQuery Plug-in called getAndDisplayAlbums, which will actually be the starting point to kick off the rest of the plug-in functionality in $.fn.picasaWebViewer function.

var tabs, gallery, repository,

picasaWebViewer = $.picasaWebViewer = {};

picasaWebViewer.getAndDisplayAlbums = function(element) {

console.group('getAndDsiplayAlbums');

picasaWebViewer.scaffoldGallery(element);

repository.getAlbums(function(albums) {

if (albums) {

picasaWebViewer.displayAlbums(albums);

}

});

console.groupEnd('getAndDsiplayAlbums');

};

picasaWebViewer.setRepository = function(object) {

repository = object;

};

picasaWebViewer.setPicasaWebViewer = function(object) {

picasaWebViewer = object;

};

So, now the tests pass because we told the "getAlbums" function to return null to the callback function which never calls the displayAlbums function and viola our tests pass.

Since we are overriding the default behavior of the Repository and PicasaWebViewer, we should set them back to their original state again before running the next set of tests.

module("Picasa Web Viewer", {

setup: function() {

$(targetId).empty();

$.picasaWebViewer.overrideOptions({});

},

teardown: function() {

$.picasaWebViewer.setJquery(oldJquery);

$.picasaWebViewer.setPicasaWebViewer(oldPicasaWebViewer);

}

});

Let's try this mocking again, but against the jQuery.ajax function. We will write a test for the getAlbums function, which uses JSONP to request against Google's Picasa Web API to retrieve album information.

test("GetAlbums Repository", function() {

$.picasaWebViewer.overrideOptions({

urlFormat : 'http://www.google.com/{0}',

userName : 'BillGates'

})

var mockJquery = new Mock();

mockJquery

.expects(1)

.method('ajax')

.withArguments({

url: 'http://www.google.com/BillGates',

success: Function,

dataType: "jsonp"

})

.callFunctionWith({ feed : { entry : "data response" }});

$.picasaWebViewer.setJquery(mockJquery);

var albums = null;

$.picasaWebViewer.repository.getAlbums(function(data) {

albums = data;

});

ok(albums, "Album Data Was Returned");

same(albums, "data response");

ok(mockJquery.verify(), 'Verify ajax was called');

});

In this test we are validating that the jQuery.ajax function was called one time and that when the jQuery.ajax success callback gets fired with a complex JSON object that our getAlbums function will parse the JSON and return only the feed.entry property. The code to make the above test is defined below.

picasaWebViewer.repository = {

getAlbums : function(callback) {

console.group('getAlbums');

var updatedUrl = picasaWebViewer.options.urlFormat.replace("{0}", picasaWebViewer.options.userName);

$.ajax({

url: updatedUrl,

success: function(data) {

callback(data.feed.entry);

},

dataType: 'jsonp'

});

console.groupEnd('getAlbums');

}

};

Up until now, in all my Mock objects I have been either expecting 0 or 1 calls being made. Lets test our displayAblums function calling multiple displayAlbum functions.

var testAlbum = {

title : {

$t : "myTitle"

},

media$group : {

media$thumbnail : [{url : "myUrl"}]

},

link: [{href : "myHref"}]

};

test("Display Albums", function() {

var mockPicasaWebViewer = new Mock();

mockPicasaWebViewer

.expects(2)

.method('displayAlbum')

.withArguments(testAlbum);

$.picasaWebViewer.setPicasaWebViewer(mockPicasaWebViewer);

$.picasaWebViewer.displayAlbums([testAlbum, testAlbum]);

ok(mockPicasaWebViewer.verify(), 'Verify displayAlbum was called twice');

});

In our test we are calling the displayAlbums function passing it an array of test albums and in our mock object we are expecting that the displayAlbum function will be called 2 times corresponding to the number of albums. The following function can be added to your plug-in in for the test to pass.

picasaWebViewer.displayAlbums = function(albums) {

console.group('displayAlbums');

$.each(albums, function() {

picasaWebViewer.displayAlbum(this);

});

console.groupEnd('displayAlbums');

};

Now we need to test that the albums were actually displayed to the page correctly. In this test we are going to make sure that the album passed to the displayAlbum function is being accurately reflected in the DOM using the album properties.

test("Display Album", function() {

//Arrange

$.picasaWebViewer.scaffoldGallery($(targetId)[0]);

//Act

$.picasaWebViewer.displayAlbum(testAlbum);

//Assert

ok($('#gallery').find('li.ui-widget-content').length, 'Found listitem');

ok($('#gallery').find('h5').text() === testAlbum.title.$t, 'Title matches');

ok($('#gallery').find('img').attr('src') === testAlbum.media$group.media$thumbnail[0].url, 'Thumbnail Url matches');

});

Note: We are reusing the testAlbum private variable that we defined for use in the previous test.

In order to make the above tests pass we need to actually define the dispalyAlbum function.

picasaWebViewer.displayAlbum = function(album) {

console.group('displayAlbum');

var title = picasaWebViewer.truncateTitle(album.title.$t);

var html =

"<li class='ui-widget-content ui-corner-tr'>" +

"<h5 class='ui-widget-header'>" + title + "</h5>" +

"<a><img src='" + album.media$group.media$thumbnail[0].url + "' alt='" + album.title.$t + "' /></a>" +

"</li>";

$(html)

.appendTo(gallery)

.children("a").attr('href', album.link[0].href)

.click(picasaWebViewer.clickAlbum);

console.groupEnd('displayAlbum');

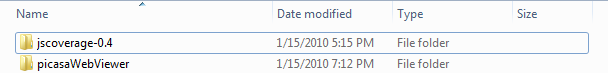
};

Instead of writing the whole program in this article, I will move on to another topic, but you are welcome to view the rest of my tests and jQuery plug-in on GitHub. Also if you are interested, you can run the jQuery Plug-in on my website.

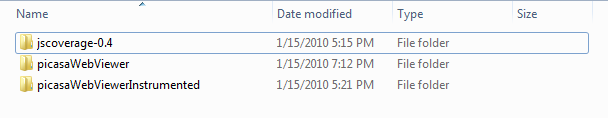
## Code Coverage

Like I mentioned above [JSCoverage](http://siliconforks.com/jscoverage/) is the code coverage tool that I use when unit testing.

The project hasn't been updated since for quite some time, but it still works well in my experience. Once you [download](http://siliconforks.com/jscoverage/download.html) the latest version from their website, you can follow the [documentation](http://siliconforks.com/jscoverage/manual.html) about how to get started with figuring out your code coverage.



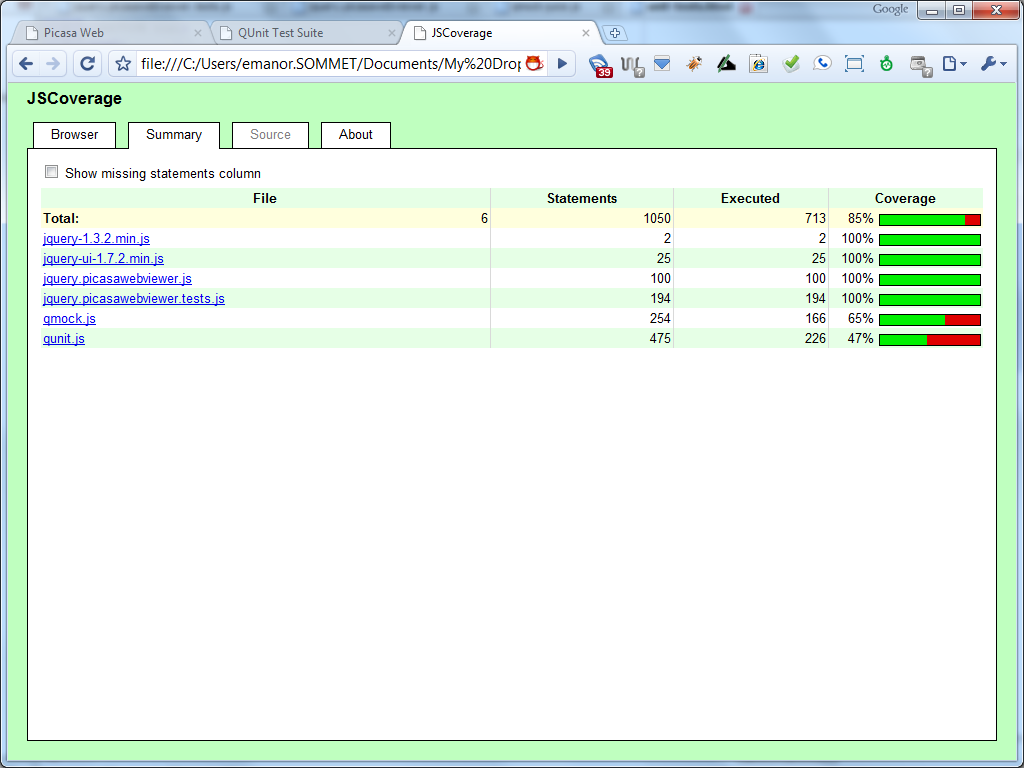
C:\...\jscoverage-0.4>jscoverage ../picasaWebViewer ../picasaWebViewerInstrumented



Then inside of your new Instrumented folder there will be the file named jscoverage.html that you'll launch to start you're code coverage analysis. You type in your QUnit html file and click the "Go" button to start analyzing your code.



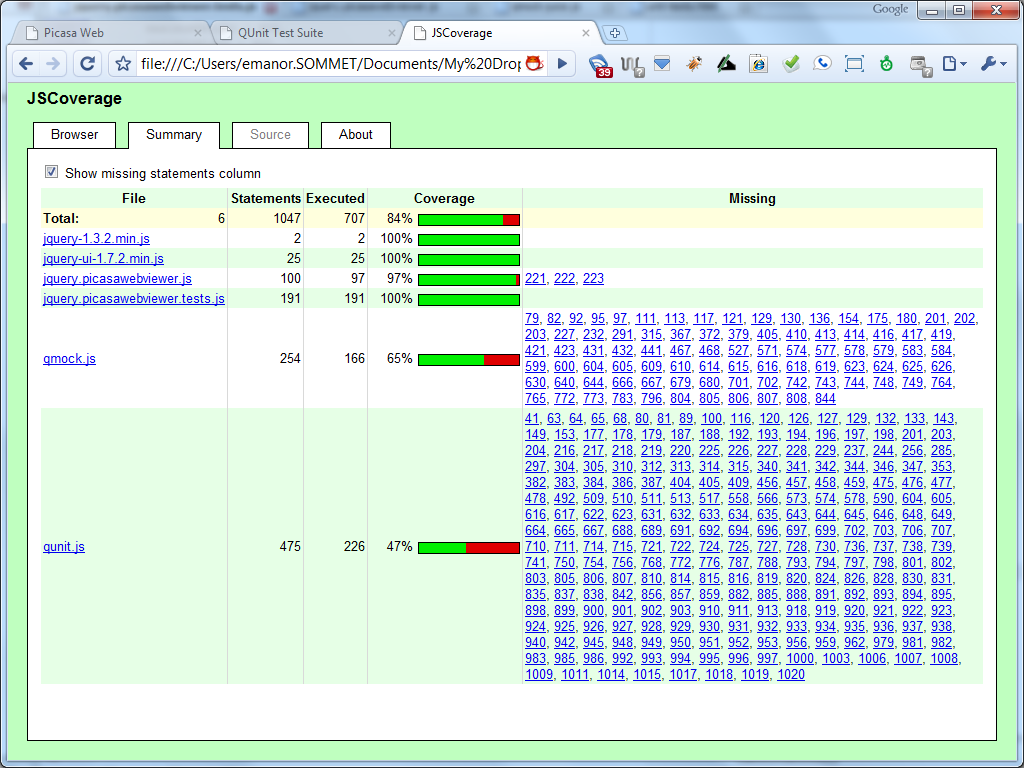
Once all the tests run, you can access the "Summary" tab to view details about the which files were ran, how many statements were executed, and the percentage of coverage your tests have.



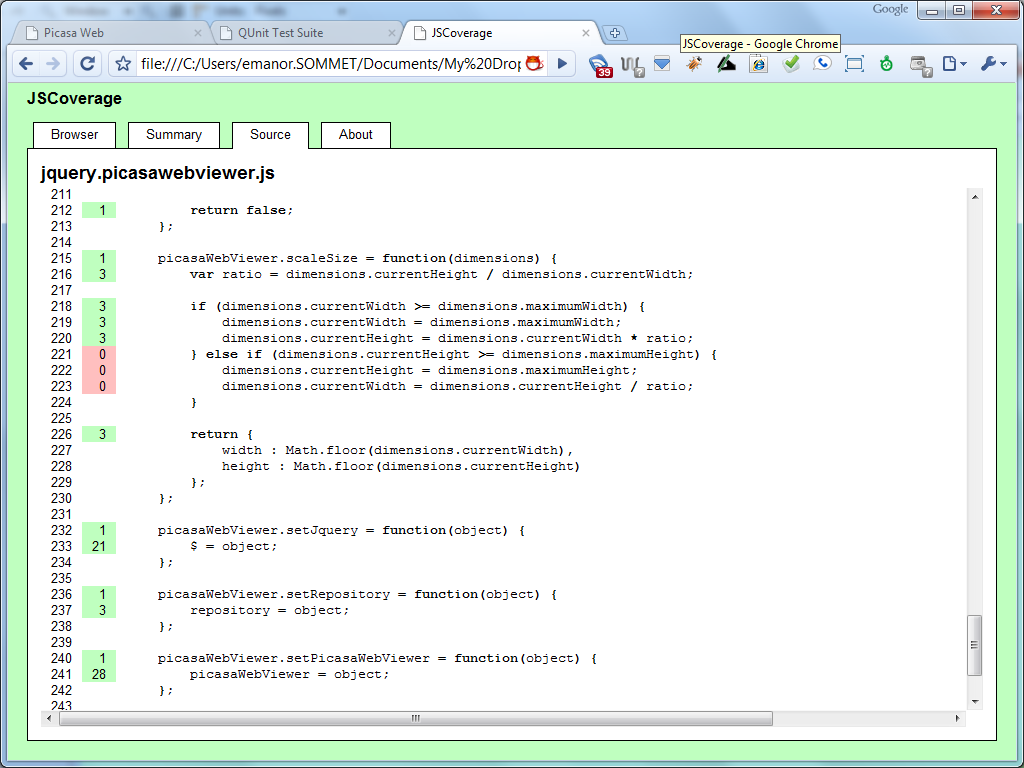
You can view the percentage of code coverage for each file in the right hand column represented in a progress bar. In our example, we have 100% test coverage, which means that our unit tests were able to execute each line in our jQuery plug-in.

Note: When you start getting into Unit Testing and Code Coverage you might get obsessed with getting 100% coverage on your tests, but that isn't necessary. The point is to test the important things and edge cases as much as you can, but there is a point where the effort of getting higher coverage is not in your best interest. Unit Testing is supposed to help give you assurance in your code's behavior and to make refactoring much easier and more reliable.

If you don't have 100% code coverage, then there are some nice features of JSCoverage to help you visually see the code that isn't covered. To see this in action, lets comment out one of our unit tests and run the code coverage tool again, but this time we will check the "Show missing statements column".



Now we can see that we only have 97% code coverage and the "Missing" column shows us which lines weren't unit tested. By clicking on one of the linked numbers you can view the source code and visually see which lines where executed.



Not only can we see the lines of code that weren't executed in our unit tests in red, but we can see how many times other lines of code were executed in green.

Note: If you decide to make additional unit tests to cover the lines in red, you will need to rerun the jscoverage command line tool to regenerate the instrumented folder to test the code coverage again.

## Conclusion

Either if you are interested in Unit Testing your JavaScript code or in JavaScript Test Driven Development, the above tools can help you gain assurance in your code's ability to complete the features you've worked developed. At the point you need to refactor your code, it is a good idea to have your code under Unit Tested so that you can rerun your tests and make sure you didn't break anything in the process.