jQuery Test Driven Development

Elijah Manor

This article takes you on a journey of how you might implement unit tests for your JavaScript code. More specifically, I’ll describe how to write a jQuery plugin using the test-driven development paradigm.

Many of you may be familiar with unit testing in other languages, such as .NET, Ruby, and Java, but this practice isn’t as frequently used in the JavaScript world. Because more client code is being placed on the browser to support richer UI experiences and better interaction with users, having unit tests that cover these areas of development makes sense as well.

# Test Driven Development

Test-driven development (TDD) is a process in which you write tests before you actually write the code for your application. [Jeffery Palermo](http://twitter.com/jeffreypalermo) wrote a good introductory article on MSDN, [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 important 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 not writing code you don't need

1. Modify your code to make any necessary improvements

* This is the point at which you can refactor into methods, rename variables, redesign, and so on

# Tools of the JavaScript Trade

To unit test your code effectively you need some tools to help you along. Many different JavaScript tools are available for you to choose from, but after quite a bit of research I have settled on these three.

## Unit Testing Tool

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

## Mocking Tool

I reviewed numerous mocking frameworks, and after a lot of research I found the [QMock](http://github.com/andybeeching/qmock) project, which suits most of my needs for a mocking and stubbing framework. I found that most of the mocking frameworks either didn't support stubbing or, if they did support stubbing, that they didn't support callback methods, which are used often 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 about the test coverage I was achieving. 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 [JSCoverage](http://siliconforks.com/jscoverage/) has worked best for my needs. The examples I show later in this article use this tool.

# Sample jQuery Plugin

For the sample project to test these concepts, I developed a simple jQuery plugin that is an album and photo viewer for Google's Picasa Web Albums. Because Google exposes JSONP support in their API, you can use jQuery AJAX to retrieve album and photo information. Here are some basic features that the jQuery plugin should support:

* Provides default options that can be overridden by users
* First displays all the public albums for the given user name inside a tab
* If the user clicks an album, a new tab should be created with the album's name as its title, and all the public photos in that album should be displayed in the new tab
* If the user clicks a photo from an album, an enlarged image should be displayed in a modal dialog box that can be resized and moved
* The user should be able to close an album's tab by clicking a close image

**Note**: The plugin has a dependency on the jQuery UI library to enable tabs and modals.

Figure 1 shows what the jQuery plugin looks like after development.

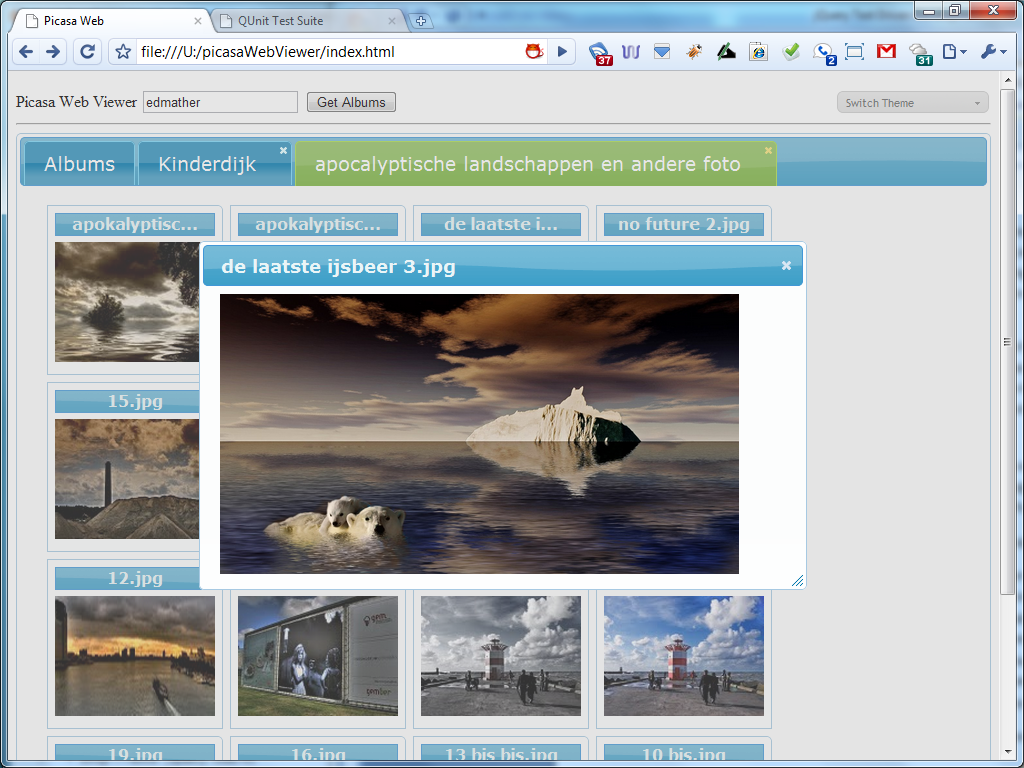


Figure 1 The sample jQuery plugin

## Setting Up Our Unit Tests

I’ll first focus on setting up unit tests and then branch out from there. If you don't have QUnit installed on your box, you can go to [jQuery/QUnit GitHub Repository](http://github.com/jquery/qunit) and click the Download Source button to get the Zip or the Tar file.

After you extract the archived file, you'll see 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 Figure 2.



Figure 2 The index.html file from QUnit

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

* jQuery library
* jQuery UI library
* QMock library
* Picasa Web Viewer jQuery plugin (empty file)
* Picasa Web Viewer tests (empty file)

Here’s an example of the markup:

<!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>

I start building the 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 plugin called PicasaWebViewer or any public methods that we can call, so if I run these tests, I get a red result, meaning that the tests failed. (See Figure 3.)

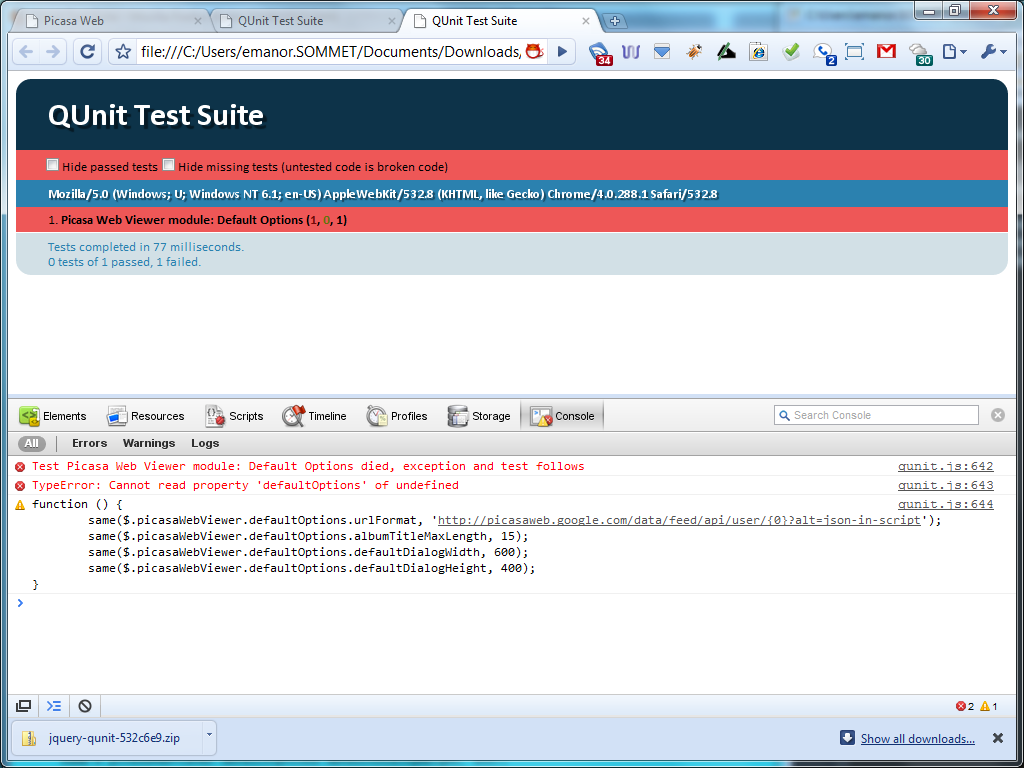


Figure 3 QUnit shows that the first tests failed.

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

(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);

When we run the tests again, QUnit returns a green success bar. (See Figure 4.) At this point, there really isn't much to refactor, so let's focus on some new tests.

**Note:** I made all the functions in my jQuery plugin public to facilitate TDD. This is not always required, but I think it makes it easier to fully unit test a piece of code. Developers debate this topic, but this article isn't intended to address those arguments.

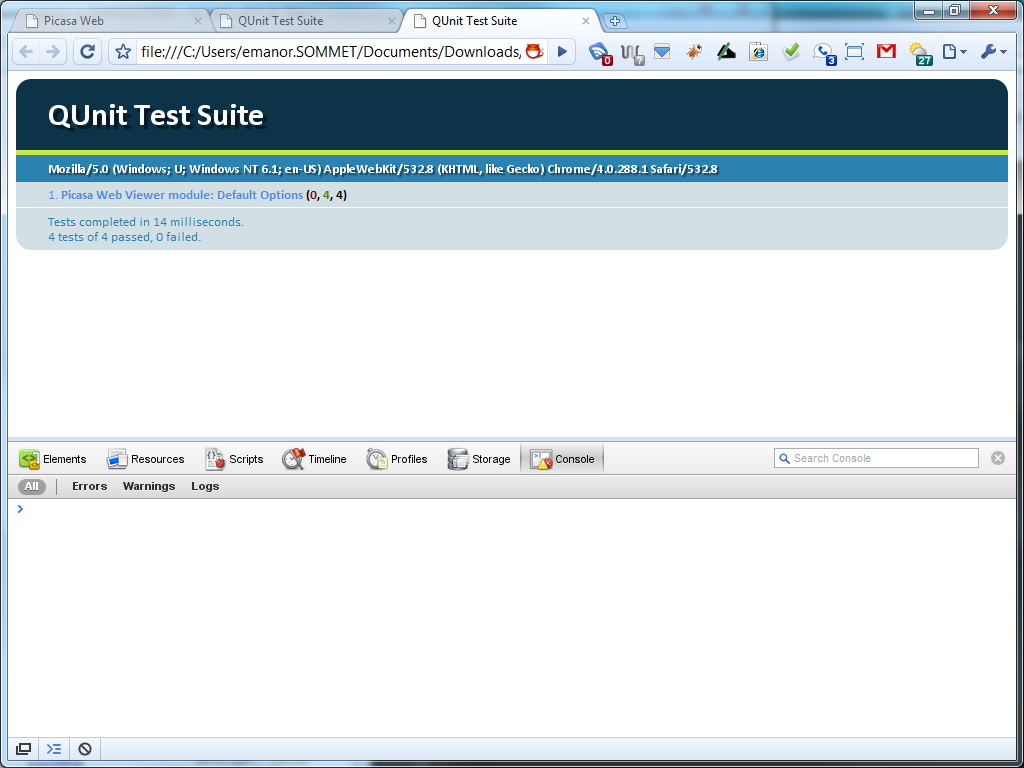


Figure 4 A green bar indicates that tests succeeded.

I’ve shown that the plugin has default values, so now let's check whether we can override them. In this test, I call the *overrideOptions* function that overrides any default option used in the plugin:

test("Override Default 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 can guess, these tests fail because I haven't defined the *overrideOptions* function yet. To get the tests to pass, I need to define the method and put a little logic in it to override the default options. Here is the code:

(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);

When I rerun the unit tests, I see the green bar of success.

You might have noticed that the code isn't really a jQuery plugin yet. I have a namespace defined with some public properties and functions, but no plugin itself. Here’s another test that calls the jQuery plugin and tests whether 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');

});

I also need to add an element to the index.html test runner with an id of "targetId" for the plugin to use. I 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 these tests fails, of course, because I haven't defined the jQuery plugin yet. I’ll do that now and call the *overrideOptions* function.

$.fn.picasaWebViewer = function(options){

picasaWebViewer.overrideOptions(options);

return this;

};

This code makes our tests pass, but it doesn't really look like much of a plugin yet, does it? Well, as we continue to flush out the tests and functionality, it will look like one more and more.

There is an important key to notice at this point in the testing. If I created another set of tests that used one of the options we've been testing, I might get a result that I didn't intend. The problem is that I haven't been resetting the plugin state between tests. The next step is 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 a *teardown* function that will run between each of your tests inside the module. We aren't using the *teardown* function yet, but we will in our later tests. In the *setup* function, I restore the options to their default values in case we test one of the public functions directly. Technically, by adding the *setup* and *teardown* functions at this point, I have violated TDD principles, but these principles are mostly guidelines, and each developer tends to modify them to suit his or her needs.

Okay, now it's time to do something a little more exciting than checking for default options. The whole point of this plugin is to show albums and pictures, so here I’ll focus on building 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');

});

Because we will continue to need an HTML element to test with, I pulled the *targetId* into its own variable so that we can use it in the following 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');

};

Because I added elements to the test runner DOM, I need to clean this up for any further tests. I can resolve this easily by adding a simple jQuery function to clear out all DOM elements from the target element in the test runner.

module("Picasa Web Viewer", {

setup: function() {

$(targetId).empty();

$.picasaWebViewer.overrideOptions({});

},

teardown: function() {

}

});

Here comes some fun in often-uncharted territory for many JavaScript developers. I am going to delve into mocking. I like the way [Roy Osherove](http://twitter.com/RoyOsherove) puts 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[r] tests—double agents. [T]hey 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 use QMock to do JavaScript mocking. The following code tests whether when the getAlbums function is called and no results are returned, that a call to displayAlbums will not occur. There is also a test to determine whether the scaffoldGallery function should have been called. Because we tested that public function earlier, we don’t need to test it again. We just want to know whether 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');

});

You should note a couple things in this code:

1. When I mock the *getAlbums* function, I pass a *Function* as the argument. This represents the callback function that is called with whatever is passed to the *callFunctionWith* function.
2. I introduced the concept of a *Repository*. I added this for functions that are going to use the AJAX functions off of jQuery. I don't actually want to make the AJAX request because it is an external dependency. Instead, I push these off to another variable so that I can mock them separately. I will cover an example of how to do this later in the article.
3. I calling to *set* functions that are overriding the default behavior of the jQuery plugin. Instead of using the defined instances inside the plugin, I replace them with the mock objects that are designed to record their behavior and return whatever I tell them to. We need to write these *set* functions in our plugin.

To make this test pass, I need to write a new function in our jQuery plugin called *getAndDisplayAlbums*. This function will actually be the starting point for the rest of the plugin functionality in the *$.fn.picasaWebViewer* function.

var tabs, gallery, repository,

picasaWebViewer = $.picasaWebViewer = {};

picasaWebViewer.getAndDisplayAlbums = function(element) {

console.group('getAndDisplayAlbums');

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;

};

Now the tests pass because I told the *getAlbums* function to return null to the callback function, which never calls the *displayAlbums* function. Viola!

And, because 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 mocking again, but this time we’ll do it against the *jQuery.ajax* function. I’ll write a test for the *getAlbums* function, which uses JSONP to send a request to 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, I verify whether the *jQuery.ajax* function was called one time and that when the *jQuery.ajax* success callback is fired with a complex JSON object, that the *getAlbums* function will parse the JSON and return only the *feed.entry* property. The code to create this test is shown here:

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');

}

};

In all the mock objects to this point, I have been expecting either 0 or 1 calls to be made. Now let’s test the *displayAblums* function by 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 this test, I call the *displayAlbums* function, passing it an array of test albums, and in the mock object we expect that the *displayAlbum* function will be called two times, corresponding to the number of albums. The following function can be added to the plugin so that the test will pass:

picasaWebViewer.displayAlbums = function(albums) {

console.group('displayAlbums');

$.each(albums, function() {

picasaWebViewer.displayAlbum(this);

});

console.groupEnd('displayAlbums');

};

The next step is to test whether the albums were actually displayed to the page correctly. In this test, I check to be sure that the album passed to the *displayAlbum* function is being accurately reflected in the DOM by 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:** I am reusing the *testAlbum* private variable that was defined in the previous test.

To make the previous tests pass, I need to actually define the *displayAlbum* function, which is shown here:

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);

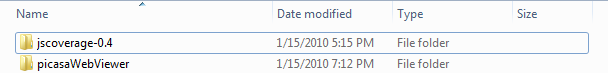
console.groupEnd('displayAlbum');

};

I won’t present the entire program in this article, but you are welcome to view the rest of my tests and jQuery plugin on GitHub. Also, if you are interested, you can run the jQuery plugin on my website.

# Code Coverage

Now I want to move on to another topic. As I mentioned earlier, [JSCoverage](http://siliconforks.com/jscoverage/) is the code-coverage tool that I use when unit testing. The project hasn't been updated 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 the JSCoverage website, you should review the [documentation](http://siliconforks.com/jscoverage/manual.html) about how to get started figuring out your code coverage.



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

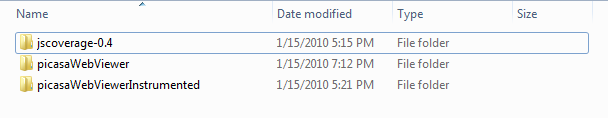


Figure 5: EM: Please add caption

Inside your new Instrumented folder, you’ll find a file named jscoverage.html, which you can launch to start analyzing your code coverage. Type in the name of your QUnit html file, and then click the Go button to start analyzing your code. You’ll see results like those shown in Figure 6.



Figure 6 Code-coverage results in JSCoverage.

After all the tests run, click the Summary tab to view details about which files were run, how many statements were executed, and the percentage of coverage your tests have. Figure 7 shows an example.

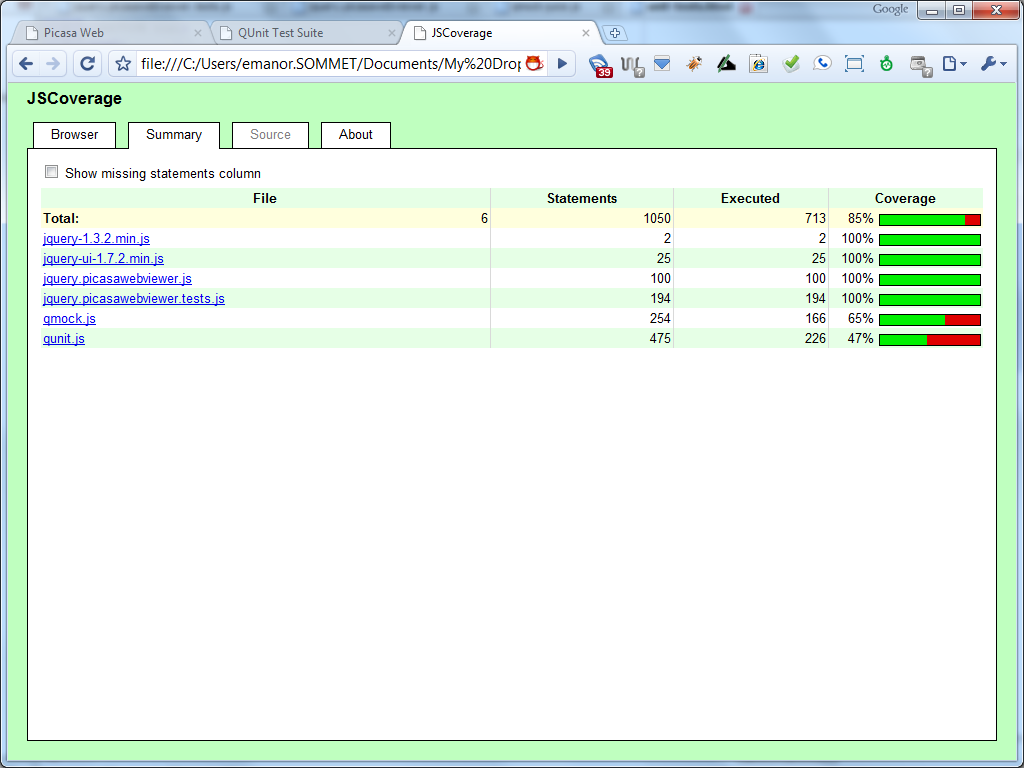


Figure 7 A summary of code coverage analysis

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

**Note:** When you start getting into unit testing and code coverage, you might become obsessed with achieving 100 percent coverage on your tests, but that isn't necessary. The point is to test the important features and requirements for sure and the edge cases as much as you can, but there is a point at which the effort of getting higher coverage is not in your best interest. Unit testing is supposed to help give you assurance about your code's behavior and to make refactoring much easier and more reliable.

If you don't have 100 percent code coverage, you can use some of the nice features of JSCoverage to help you see the code that isn't covered. To see this in action, let’s comment out one of our unit tests and run the code coverage tool again, but this time we’ll check the Show Missing Statements Column option. The results are shown in Figure 8.

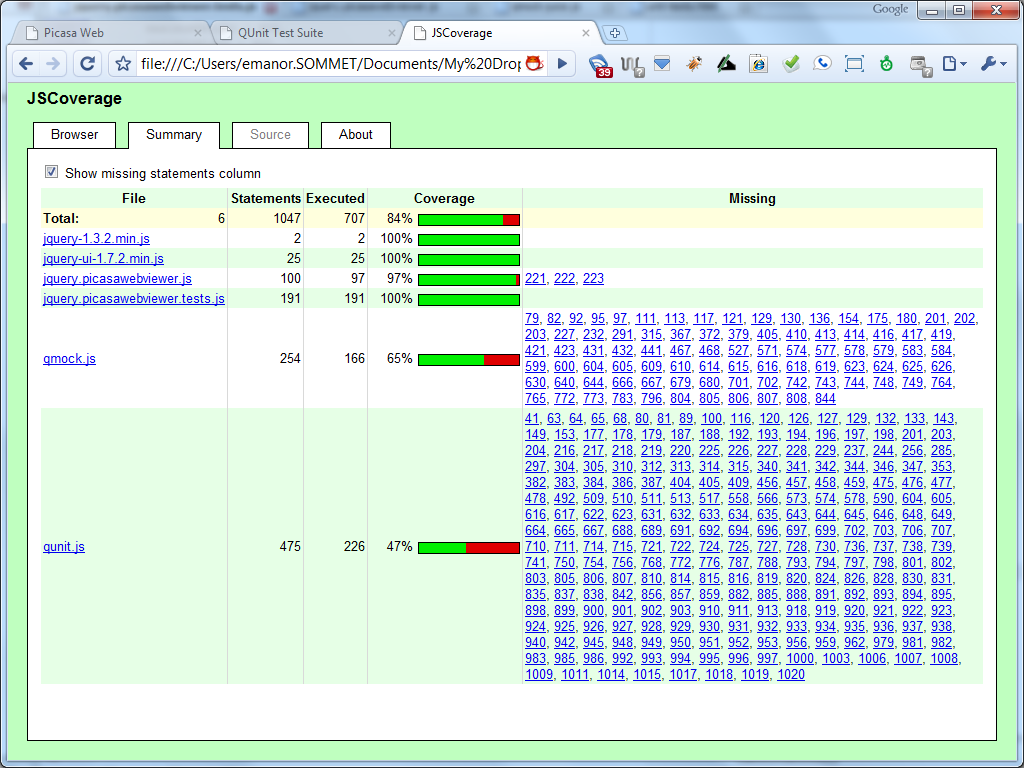
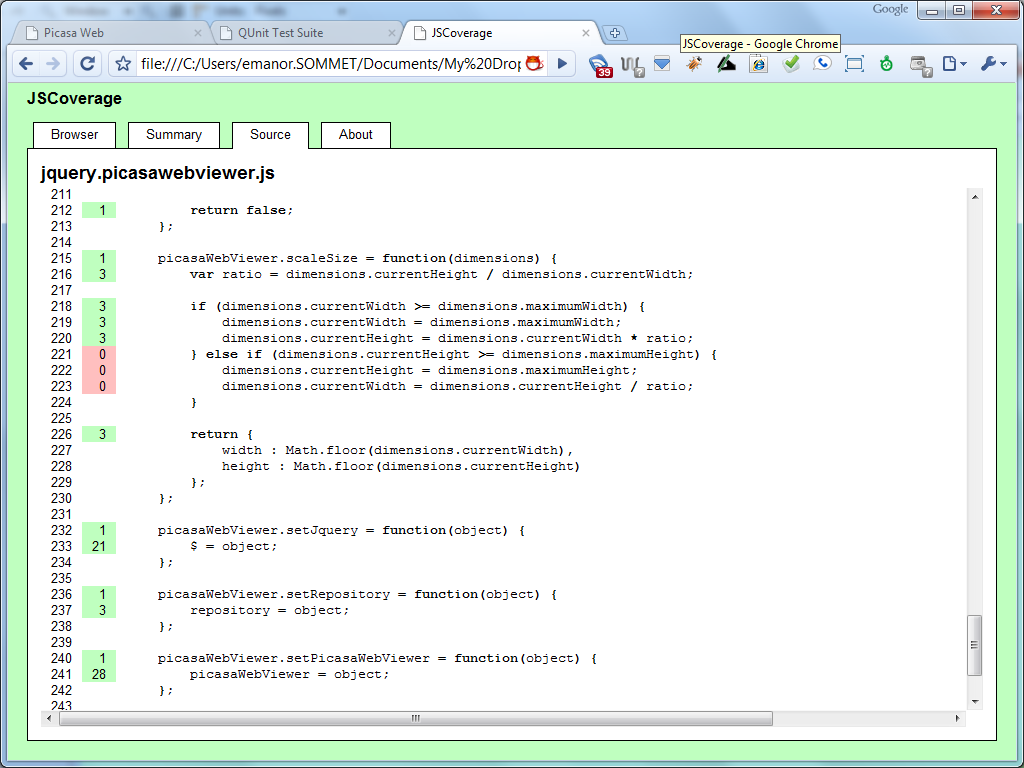


Figure 8 You can check untested lines by clicking them.

Now you can see that code coverage is only 97 percent and that the Missing column shows us which lines weren't tested. By clicking one of the linked numbers, you can view the source code and see which lines were executed, as shown in Figure 9.



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

**Note:** If you decide to write additional unit tests to cover the lines shown in red, you need to rerun the JSCoverage command-line tool to regenerate the Instrumented folder to test code coverage again.

# Conclusion

If you are interested in unit testing your JavaScript code or in JavaScript test-driven development, the tools I’ve demonstrated in this article can help you gain confidence in your code's ability to complete the features you've worked to develop. At the point at which 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 be sure you didn't break anything in the process.