**A JavaScript Unit Testing framework**

## **What is QUnit?**

QUnit is a powerful, easy-to-use JavaScript unit testing framework.

It's used by the jQuery, jQuery UI and jQuery Mobile projects and is capable of testing any generic JavaScript code, including itself!

To use QUnit, you only need to include two QUnit files on your HTML page.

QUnit consists of qunit.js, the test runner and testing framework, and qunit.css, which styles the test suite page to display test results:

[QUnit](http://docs.jquery.com/QUnit), developed by the jQuery team, is a great framework for unit testing your JavaScript.

**What is Unit Testing?**

In computer programming, unit testing is a software verification and validation method in which a programmer tests if individual units of source code are fit for use.

A unit is the smallest testable part of an application.

In procedural programming a unit may be an individual function or procedure.

## **Getting Started**

**Example:**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>QUnit Example</title>

<link rel="stylesheet" href="//code.jquery.com/qunit/qunit-1.18.0.css">

</head>

<body>

<div id="qunit"></div>

<div id="qunit-fixture"></div>

<script src="//code.jquery.com/qunit/qunit-1.18.0.js"></script>

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

</body>

</html>

**The contents of tests.js:**

QUnit.test( "hello test", function( assert ) {

assert.ok( 1 == "1", "Passed!" );

});

### **Browser Support**

QUnit supports the same browsers as jQuery 1.x.

That's IE6+ and Current - 1 for Chrome, Firefox, Safari and Opera.

## **Download**

QUnit is available from the [jQuery CDN](http://code.jquery.com) hosted by [MaxCDN](http://www.maxcdn.com/)

### **Current Release - v1.18.0**

* [qunit-1.18.0.js](http://code.jquery.com/qunit/qunit-1.18.0.js)
* [qunit-1.18.0.css](http://code.jquery.com/qunit/qunit-1.18.0.css)
* [NPM](https://npmjs.org/package/qunitjs): npm install --save-dev qunitjs
* Bower: bower install --save-dev qunit

**To get latest version, you can use below URL:**

<http://code.jquery.com/qunit/qunit-git.js>

<http://code.jquery.com/qunit/qunit-git.css>

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>QUnit Example</title>

<link rel="stylesheet" href="qunit-1.18.0.css">

</head>

<body>

<div id="qunit"></div>

<div id="qunit-fixture"></div>

<script src="qunit-1.18.0.js"></script>

<script>

function isEven(arg) {

if (arg % 2 == 0) {

return true;

}

else {

return false;

}

}

QUnit.test("hello test", function (assert) {

assert.ok(1 == "1", "Passed!");

});

QUnit.test('test 1', function () {

equal('satya' == 'satya', true, 'test passed');

});

QUnit.test('test isEven method', function () {

equal(isEven(4), true, 'test passed');

});

var app = {};

app.Calculator = {

addition: function (arg1, arg2) {

return arg1 + arg2;

},

substraction: function (arg1, arg2) {

return arg1 - arg2;

},

multiply: function (arg1, arg2) {

return arg1 \* arg2;

}

}

QUnit.test('test app', function () {

equal(app.Calculator.addition(1, 2), 3, 'test passed');

equal(app.Calculator.substraction(1, 2), -1, 'test passed');

equal(app.Calculator.multiply(1, 2), 2, 'test passed');

});

</script>

</body>

</html>

**Sample JavaScript Unit Testing Tools:**

**QUnit**: for JavaScript, jQuery, jQuery plugins

**Jasmine**: Ruby or Ruby on Rails

**Karma** or **Protractor**: for angular

**Buster**.js: JavaScript Test runner built with Node.js

**Test Swarm**

**Sinon**

**Mocha**

**JSUnit**

**More about QUnit.test()**

The test() method comes with two arguments: the name of the test as a string, which is later used to display the test results, and a function.

The function contains the actual testing code, which involves one or more assertions.

The building blocks of unit tests are assertions.

An assertion is a statement that predicts the returning result of your code.

If the prediction is false, the assertion has failed, and you know that something has gone wrong.

To run assertions, you should put them into a test case:

// Let's test this function

function isEven(val) {

    return val % 2 === 0;

}

test('isEven()', function() {

    ok(isEven(0), 'Zero is an even number');

    ok(isEven(2), 'So is two');

    ok(isEven(-4), 'So is negative four');

    ok(!isEven(1), 'One is not an even number');

    ok(!isEven(-7), 'Neither is negative seven');

})

We wrote five assertions, all of which are boolean.

A boolean assertion expects its first parameter to be true. The second parameter is also a message that will be displayed in the result.

## **More Assertions**

ok() is not the only assertion that QUnit provides.

There are other kinds of assertions that are useful when testing your project:

## **Comparison Assertion**

The comparison assertion, equal(), expects its first parameter (which is the actual value) is equal to its second parameter (which is the expected value).

It's similar to ok(), but outputs both actual and expected values, making debugging much easier.

Like ok(), it takes an optional third parameter as a message to be displayed.

**So instead of:**

test('assertions', function() {

    ok( 1 == 1, 'one equals one');

})

**You should write:**

test('assertions', function() {

    equal( 1, 1, 'one equals one');

})

And if the values are not equal:

test('assertions', function() {

    equal( 2, 1, 'one equals one');

})

The comparison assertion uses "==" to compare its parameters, so it doesn't handle array or object comparison:

test('test', function() {

    equal( {}, {}, 'fails, these are different objects');

    equal( {a: 1}, {a: 1} , 'fails');

    equal( [], [], 'fails, there are different arrays');

    equal( [1], [1], 'fails');

})

In order to test this kind of equality, QUnit provides another kind assertion: **identical assertion**.

## **Identical Assertion**

Identical assertion, deepEqual(), expects the same parameters as equal(), but it's a deep recursive comparison assertion that works not only on primitive types, but also arrays and objects

Assertions, in the previous example, will all pass if you change them to identical assertions:

test('test', function() {

deepEqual({}, {}, 'fails, these are different objects');

deepEqual({ a: 1 }, { a: 1 }, 'fails');

deepEqual([], [], 'fails, there are different arrays');

deepEqual([1], [1], 'fails');

})

Notice that deepEqual () uses '===' to do comparison when possible, so it'll come in handy when comparing special values:

test('test', function() {

    equal( 0, false, 'true');

    deepEqual ( 0, false, 'false');

    equal( null, undefined, 'true');

    deepEqual ( null, undefined, 'false');

})

## **Structure Your Assertions**

Putting all assertions in a single test case is a really bad idea, because it's very hard to maintain, and doesn't return a clean result.

What you should do is to structure them, put them into different test cases, each aiming for a single functionality.

You can even organize test cases into different modules by calling the module function:

module('Module A');

test('a test', function() {});

test('an another test', function() {});

module('Module B');

test('a test', function() {});

test('an another test', function() {});

Complete Example:

Test.html

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>QUnit Example</title>

<link rel="stylesheet" href="qunit-1.18.0.css">

<script src="qunit-1.18.0.js"></script>

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

</head>

<body>

<div id="qunit"></div>

<div id="qunit-fixture"></div>

</body>

</html>

Tests.js

function isEven(arg) {

if (arg % 2 == 0) {

return true;

}

else {

return false;

}

}

module('Module1');

QUnit.test("hello test", function (assert) {

assert.ok(1 == "1", "Passed!");

});

QUnit.test('test 1', function () {

equal('satya' == 'satya', true, 'test passed');

});

QUnit.test('test isEven method', function () {

equal(isEven(4), true, 'test passed');

ok(isEven(0), 'Value is Zero');

ok(isEven(3), 'Value is Even Number');

});

var app = {};

app.Calculator = {

addition: function (arg1, arg2) {

return arg1 + arg2;

},

substraction: function (arg1, arg2) {

return arg1 - arg2;

},

multiply: function (arg1, arg2) {

return arg1 \* arg2;

}

}

module('Module2');

QUnit.test('test app', function () {

deepEqual({}, {}, 'fails, these are different objects');

deepEqual({ a: 1 }, { a: 1 }, 'fails');

deepEqual([], [], 'fails, there are different arrays');

deepEqual([1], [1], 'fails');

equal(app.Calculator.addition(1, 2), 3, 'test passed');

equal(app.Calculator.substraction(1, 2), -1, 'test passed');

equal(app.Calculator.multiply(1, 2), 2, 'test passed');

});

**Some of the QUnit assertion methods:**

equal

notEqual

deepEqual

notDeepEqual

ok

notOk

propEqual

notPropEqual

strictEqual

notStrictEqual

## **Asynchronous Test**

In previous examples, all assertions are called synchronously, which means they run one after another.

In the real world, there are also many asynchronous functions, such as ajax calls or functions called by setTimeout() and setInterval().

How can we test these kinds of functions?

QUnit provides a special kind of test case called "asynchronous test," which is dedicated to asynchronous testing:

**Regular Test:**

test('asynchronous test', function() {

    setTimeout(function() {

        ok(true);

    }, 5000)

})

**Here is the correct version:**

QUnit.test('test', function () {

// Pause the test first

stop();

setTimeout(function () {

ok(true);

// After the assertion has been called,

// continue the test

start();

}, 5000)

});

Here, we use stop() to pause the test case, and after the assertion has been called, we use start() to continue.

Calling stop() immediately after calling test() is quite common;

so QUnit provides a shortcut: asyncTest().

You can rewrite the previous example like this:

QUnit.asyncTest('test', function () {

setTimeout(function () {

ok(true);

// After the assertion has been called,

// continue the test

start();

}, 5000)

});

/ A custom function

function ajax(successCallback) {

    $.ajax({

        url: 'server.php',

        success: successCallback

    });

}

test('asynchronous test', function() {

    // Pause the test, and fail it if start() isn't called after one second

    stop(1000);

    ajax(function() {

        // ...asynchronous assertions

        start();

    })

})

You pass a timeout to stop(), which tells QUnit, "if start() isn't called after that timeout, you should fail this test."

You can be sure that the whole testing won't hang and you'll be notified if something goes wrong.

How about multiple asynchronous functions? Where do you put the start()?

**You put it in setTimeout():**

// A custom function

function ajax(successCallback) {

    $.ajax({

        url: 'server.php',

        success: successCallback

    });

}

test('asynchronous test', function() {

    // Pause the test

    stop();

    ajax(function() {

        // ...asynchronous assertions

    })

    ajax(function() {

        // ...asynchronous assertions

    })

    setTimeout(function() {

        start();

    }, 2000);

})

The timeout should be reasonably long enough to allow both callbacks to be called before the test continues.

But what if one of the callback isn't called? How can you know that?

This is where expect() comes in:

// A custom function

function ajax(successCallback) {

    $.ajax({

        url: 'server.php',

        success: successCallback

    });

}

test('asynchronous test', function() {

    // Pause the test

    stop();

    // Tell QUnit that you expect three assertions to run

    expect(3);

    ajax(function() {

        ok(true);

    })

    ajax(function() {

        ok(true);

        ok(true);

    })

    setTimeout(function() {

        start();

    }, 2000);

})

You pass in a number to expect() to tell QUnit that you expect X many assertions to run, if one of the assertion isn't called, the number won't match, and you'll be notified that something went wrong.

There is also a shortcut for expect():

You just pass the number as the second parameter to test() or asyncTest():

// A custom function

function ajax(successCallback) {

    $.ajax({

        url: 'server.php',

        success: successCallback

    });

}

// Tell QUnit that you expect three assertion to run

test('asynchronous test', 3, function() {

    // Pause the test

    stop();

    ajax(function() {

        ok(true);

    })

    ajax(function() {

        ok(true);

        ok(true);

    })

    setTimeout(function() {

        start();

    }, 2000);

})

**QUnit assertions**

#### **ok( truthy [, message ] )**

The most basic one is ok(), which requires just one argument.

If the argument evaluates to true, the assertion passes; otherwise, it fails.

In addition, it accepts a string to display as a message in the test results:

QUnit.test( "ok test", function( assert ) {

assert.ok( true, "true succeeds" );

assert.ok( "non-empty", "non-empty string succeeds" );

assert.ok( false, "false fails" );

assert.ok( 0, "0 fails" );

assert.ok( NaN, "NaN fails" );

assert.ok( "", "empty string fails" );

assert.ok( null, "null fails" );

assert.ok( undefined, "undefined fails" );

});

#### **equal( actual, expected [, message ] )**

The equal assertion uses the simple comparison operator (==) to compare the actual and expected arguments.

When they are equal, the assertion passes; otherwise, it fails.

When it fails, both actual and expected values are displayed in the test result, in addition to a given message:

QUnit.test( "equal test", function( assert ) {

assert.equal( 0, 0, "Zero, Zero; equal succeeds" );

assert.equal( "", 0, "Empty, Zero; equal succeeds" );

assert.equal( "", "", "Empty, Empty; equal succeeds" );

assert.equal( 0, false, "Zero, false; equal succeeds" );

assert.equal( "three", 3, "Three, 3; equal fails" );

assert.equal( null, false, "null, false; equal fails" );

});

Compared to ok(), equal() makes it much easier to debug tests that failed, because it's obvious which value caused the test to fail.

When you need a strict comparison (===), use strictEqual() instead.

#### **deepEqual( actual, expected [, message ] )**

The deepEqual() assertion can be used just like equal() and is a better choice in most cases.

Instead of the simple comparison operator (==), it uses the more accurate comparison operator (===).

That way, undefined doesn't equal null, 0, or the empty string ("").

It also compares the content of objects so that {key: value} is equal to {key: value}, even when comparing two objects with distinct identities.

deepEqual() also handles NaN, dates, regular expressions, arrays, and functions, while equal() would just check the object identity:

QUnit.test( "deepEqual test", function( assert ) {

var obj = { foo: "bar" };

assert.deepEqual( obj, { foo: "bar" }, "Two objects can be the same in value" );

});

## 

## **Synchronous Callbacks**

### **Problem**

Occasionally, circumstances in your code may prevent callback assertions to never be called, causing the test to fail silently.

### **Solution**

QUnit provides a special assertion to define the number of assertions a test contains.

When the test completes without the correct number of assertions, it will fail, no matter what result the other assertions, if any, produced.

Usage is plain and simple;

Just call assert.expect() at the start of a test, with the number of expected assertions as the only argument:

QUnit.test("a test", function (assert) {

assert.expect(2);

function calc(x, operation) {

return operation(x);

}

var result = calc(2, function (x) {

assert.ok(true, "calc() calls operation function");

return x \* x;

});

assert.equal(result, 4, "2 square equals 4");

});

## **Asynchronous Callbacks**

### **Problem**

While assert.expect() is useful to test synchronous callbacks

It can fall short for asynchronous callbacks.

Asynchronous callbacks conflict with the way the test runner queues and executes tests.

When code under test starts a timeout or interval or an AJAX request, the test runner will just continue running the rest of the test, as well as other tests following it, instead of waiting for the result of the asynchronous operation.

### **Solution**

For every asynchronous operation in your QUnit.test() callback, use assert.async(), which returns a "done" function that should be called when the operation has completed.

QUnit.test("asynchronous test: async input focus", function (assert) {

var done = assert.async();

var input = $("#test-input").focus();

setTimeout(function () {

assert.equal(document.activeElement, input[0], "Input was focused");

done();

});

});

## **Testing User Actions**

### **Problem**

Code that relies on actions initiated by the user can't be tested by just calling a function.

Usually an anonymous function is bound to an element's event, e.g., a click, which has to be simulated.

### **Solution**

You can trigger the event using jQuery's trigger() method and test that the expected behavior occurred.

If you don't want the native browser events to be triggered, you can use triggerHandler() to just execute the bound event handlers.

This is useful when testing a click event on a link, where trigger() would cause the browser to change the location, which is hardly desired behavior in a test.

Let's assume we have a simple key logger that we want to test:

function KeyLogger(target) {

this.target = target;

this.log = [];

var that = this;

this.target.off("keydown").on("keydown", function (event) {

that.log.push(event.keyCode);

});

}

QUnit.test("keylogger api behavior", function (assert) {

var doc = $(document),

keys = new KeyLogger(doc);

// Trigger the key event

// We can manually trigger a keypress event to see whether the logger is working:

doc.trigger($.Event("keydown", { keyCode: 9 }));

// Verify expected behavior

assert.deepEqual(keys.log, [9], "correct key was logged");

});