jQuery JavaScript Library

About

QUnit is a powerful, easy-to-use, JavaScript test suite. It's used by the jQuery project to test its code and plugins but is capable of testing any generic JavaScript code (and even capable of testing JavaScript code on the server-side).

QUnit is especially useful for regression testing: Whenever a bug is reported, write a test that asserts the existence of that particular bug. Then fix it and commit both. Every time you work on the code again, run the tests. If the bug comes up again - a regression - you'll spot it immediately and know how to fix it, because you know what code you just changed.

Having good unit test coverage makes safe refactoring easy and cheap. You can run the tests after each small refactoring step and always know what change broke something.

QUnit is similar to other unit testing frameworks like JUnit, but makes use of the features JavaScript provides and helps with testing code in the browser, eg. with it's stop/start facilities for testing asynchronous code.

The code is located at: http://github.com/jquery/qunit

Contact

QUnit is maintained by Jörn Zaefferer and John Resig.

Please post to the QUnit and testing forum for anything related to QUnit or testing in general.

Using QUnit

To use QUnit, you have to include its qunit.js and qunit.css files and provide a basic HTML structure for displaying the test results:

Basic usage examples

```
test("a basic test example", function() {
  ok( true, "this test is fine" );
  var value = "hello";
  equals( "hello", value, "We expect value to be hello" );
});

module("Module A");

test("first test within module", function() {
  ok( true, "all pass" );
});

test("second test within module", function() {
  ok( true, "all pass" );
});

module("Module B");

test("some other test", function() {
  expect(2);
  equals( true, false, "failing test" );
  equals( true, true, "passing test" );
});



<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <script src="http://code.jquery.com/jquery-latest.js"></script>
  <link rel="stylesheet" href="http://code.jquery.com/qunit/git/qunit.css" type="text/
css" media="screen" />
<script type="text/javascript" src="http://code.jquery.com/qunit/git/qunit.js">
</script>

  <script>
  $(document).ready(function(){

test("a basic test example", function() {
  ok( true, "this test is fine" );
  var value = "hello";
  equals( "hello", value, "We expect value to be hello" );
```

```
});

module("Module A");

test("first test within module", function() {
  ok( true, "all pass" );
});

test("second test within module", function() {
  ok( true, "all pass" );
});

module("Module B");

test("some other test", function() {
  expect(2);
  equals( true, false, "failing test" );
  equals( true, true, "passing test" );
});

  });
  </script>

</head>
<body>
  <h1 id="qunit-header">QUnit example</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>
</body>
</html>
```

The #qunit-header element should contain the name of the testsuite, and won't be modified by QUnit. The #qunit-banner element will set to show up as red if a test failed, green if all tests passed. The #qunit-userAgent elements is set to display the navigator.userAgent property. The #qunit-tests element will be used as a container for the test results.

The #qunit-fixture element can be used to provide and manipulate test markup, and will be automatically reset after each test (see QUnit.reset). The element is styled with position:absolute; top:-

10000px; left:-10000; - with these, it won't be obstructing the result, without affecting code the relies on the affected elements to be visible (instead of display:none). Older QUnit versions would only work with the #main element, which is deprecated, but still supported. It just won't get any default styles.

All these are optional. See below for alternatives on processing the test results.

API documentation

Setup:

Name	Type

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.

module(name, lifecycle)

Separate tests into modules.

QUnit.init( )

Initialize the test runner (if the runner has already run it'll be re-initialized, effectively resetting it). This method does not need to be called in the normal use of QUnit.

QUnit.reset( )

Automatically called by QUnit after each test. Can be called by test code, though usually its better to seperate test code with multiple calls to test().

Assertions:

Name	Type

ok(state, message)

A boolean assertion, equivalent to JUnit's assertTrue. Passes if the first argument is truthy.

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(state, message)
Assertion to test if a callback throws an exception when run.

Asynchronous Testing:

URL Parameters

You can customize individual testruns via URL paramters. To start, double click on a test to have QUnit only run that single test.

You can add the name of a module to the URL to run only tests within that module, eg swarm.jquery.org/git/jquery/test/?effects runs only tests for effects. You can combine various filters, eg. ?effects&ajax to run effects and ajax tests.

Another feature is global-variables pollution detection. Add ?noglobals to the URL, and QUnit will detect if a test introduced a new global variable (aka new properties on the window object), making that test fail.

Integration into Browser Automation Tools

To integrate QUnit into browser automation tools, those doing the work of launching various browsers and gathering the results, QUnit provides a simple microformat for its test result.

```
<p id="qunit-testresult" class="result">
  Tests completed in 221 milliseconds.<br/>
  <span class="passed">232</span> tests of
  <span class="total">232</span> passed,
  <span class="failed">0</span> failed.
</p>
```

Additionally, QUnit provides a series of callbacks that can be overwritten to provide updates when various actions occur. The callbacks are:

- `QUnit.log(result, message)` is called whenever an assertion is completed. `result` is a boolean (true for passing, false for failing) and `message` is a string description provided by the assertion.
- `QUnit.testStart(name)` is called whenever a new test batch of assertions starts running. `name` is the string name of the test batch.
- `QUnit.testDone(name, failures, total)` is called whenever a batch of assertions finishes running. `name` is the string name of the test batch. `failures` is the number of test failures that occurred. `total` is the total number of test assertions that occurred.
- `QUnit.moduleStart(name)` is called whenever a new module of tests starts running. `name` is the string name of the module.
- `QUnit.moduleDone(name, failures, total)` is called whenever a module finishes running. `name` is the string name of the module. `failures` is the number of module failures that occurred.

`total` is the total number of module assertions that occurred.

- `QUnit.begin()` is called once before running any tests. (a better would've been QUnit.start, but thats already in use elsewhere and can't be changed.)
- `QUnit.done(failures, total)` is called whenever all the tests have finished running. `failures` is the number of failures that occurred. `total` is the total number of assertions that occurred.

Additionally `QUnit.reset` is called after every test group. You can use override this method to reset the DOM structure to its original state (if you so choose).

Reference Test Suites

jQuery itself has the biggest set of tests using QUnit:

The validation plugin has decent test coverage, too:

Examples from the jQuery UI project:

Further tutorials

If you want to read more on unit testing JavaScript (not specific to QUnit), check out the book Test-Driven JavaScript Development.

Extensions and integrations

More to come.

Original URL:

http://docs.jquery.com/Qunit