QUNIT FUNDAMENTALS

This document guides:

* What is QUnit?
* How to use it for testing?

# What is QUnit

References:

<http://qunitjs.com/>

<https://github.com/jquery/qunit>

QUnit: JavaScript unit testing framework. Current version is 1.15.0.

# Setting up QUnit

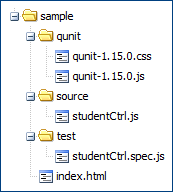
You need two things to use QUnit: qunit.js and qunit.css. You can use them at remote side OR download them to use at your local side.

To download them, use can access to:

<http://code.jquery.com/qunit/qunit-1.15.0.js>

<http://code.jquery.com/qunit/qunit-1.15.0.css>

Sample, I organizate my folders and files as:



qunit folder contains all libraries from qunit. source folder contains actually javascript source. test folder contains testing script files.

# A simple example

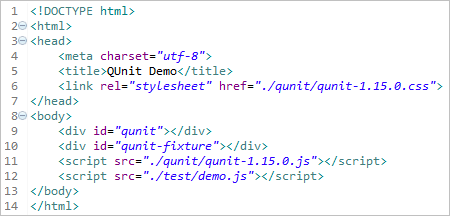
## Step 1, create file contains cases

We need file to contain one/group of case(s). I create new test file at sample/test/demo.js. I use ok function from QUnit to create test case as:



## Step 2, setup index file

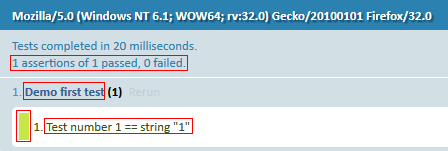
This file is used to include one/some files(s) that contain(s) cases. Modifying index.html file to test demo.js as:



Line 6 and line 11 use resource from QUnit. Line 9 and line 10 use to display the testing result. They are required. Line 12 uses to refer to test files.

## Step 3, use browser to view results

Using browser to open the file and viewing result as:



Description:

* 1 assertions of 1 passed, 0 failed means {passed assertions} assertions of {total of assertions} passed, {failed assertions } failed.
* Demo first test is test case name.
* Test number 1 == string "1" is assertion name.
* Green color says that one assertion is passed.

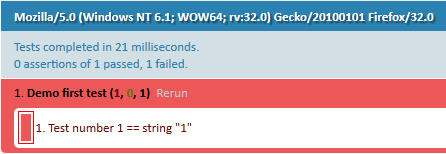
You can click on Rerun link at a specific case to only re-run this case.

## Try to view failed assertion

Trying to change the condition of test file to return false result as:



And refresh browser. You can see as:

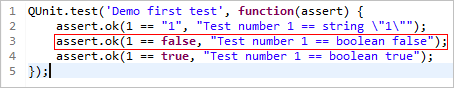


Description:

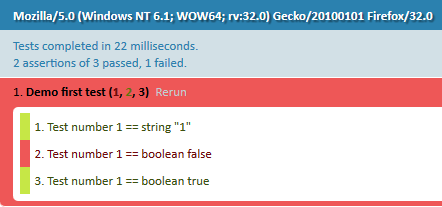
* Demo first test (1, 0, 1) means Demo first test ({failed assertions}, {passed assertions}, {total of assertions}).
* Red color says that one assertion is failed.

## Try to mix passed assertions and failed assertions

I try to create case that contains both passed and failed assertions as:



When refresh browser, you can see as:



# Assert – ok function

Method to check boolean value.

Structure:

ok(state[, message])

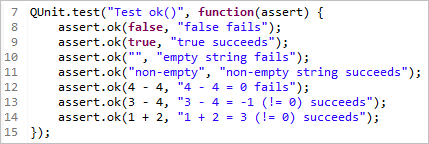
Description for structure:

* state is an expression. Required. Focusing on expression.
* message is a string. Optional.

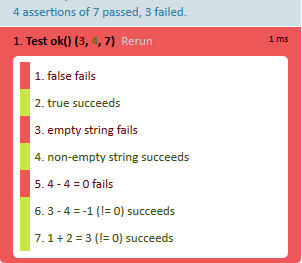
## Using with primitive types

Primitive types consists of: boolean, string, number

Sample as:



The result as:

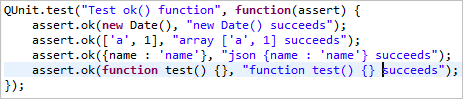


Remarks: false, empty string and zero number mean false.

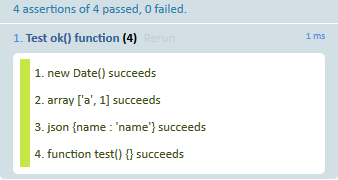
## Using with object

Object consists of: date, array, json, function …

Sample as:



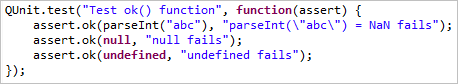
The result as:



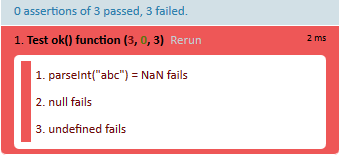
## Using with special types

Special types consist of: NaN, null and undefined.

Sample as:



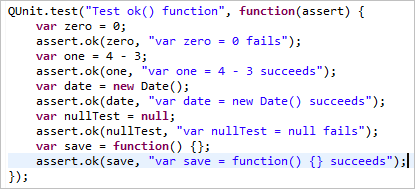
The result as:



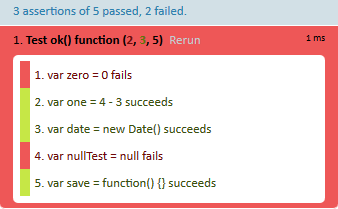
## Using with variables

It depends on the value of variable to return true/false.

Sample as:



The result as:



# Assert – equal function

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.

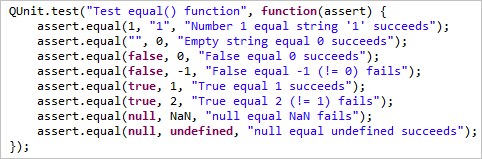
Structure:

equal(actual, expected[, message])

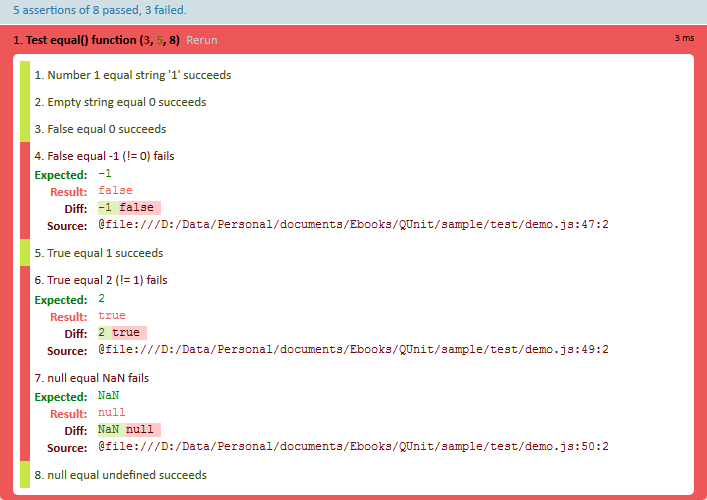
Description:

* actual is an expression. Required.
* expected is an expression. Required.
* message is a string. Optional.

Sample as:



The result as:



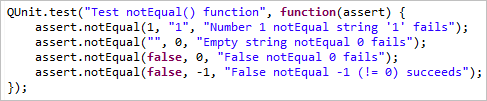
# Assert – notEqual function

The opposite thing of equal function.

Structure:

notEqual(actual, expected[, message])

Sample as:



# Assert – strictEqual function

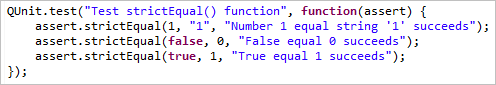
A strict type and value comparison. It uses strict equal (===) for comparing.

Structure:

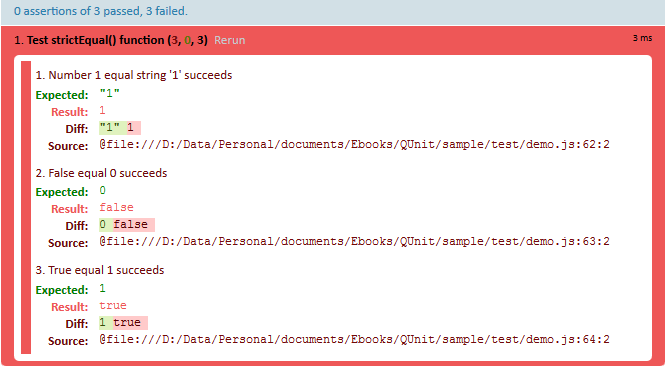
strictEqual(actual, expected[, message])

Referring to Assert – equal function to understand each parameters. They are same.

Sample as:



The result as:



# Assert – notStrictEqual function

The opposite thing of strictEqual function. It has same structure with strictEqual function.

Structure:

notStrictEqual(actual, expected[, message])

Referring to Assert – equal function to understand each parameters.

# Assert – deepEqual function

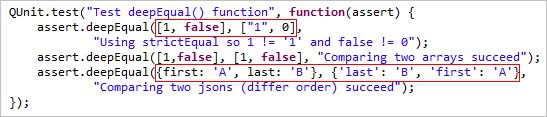
A deep recursive comparison, working on primitive types, arrays, objects, regular expressions, dates and functions. The deepEqual assertion can be used just like equal when comparing the value of objects, such that {key: value} is equal to {key: value}. For non-scalar values, identity will be disregarded by deepEqual.

Structure:

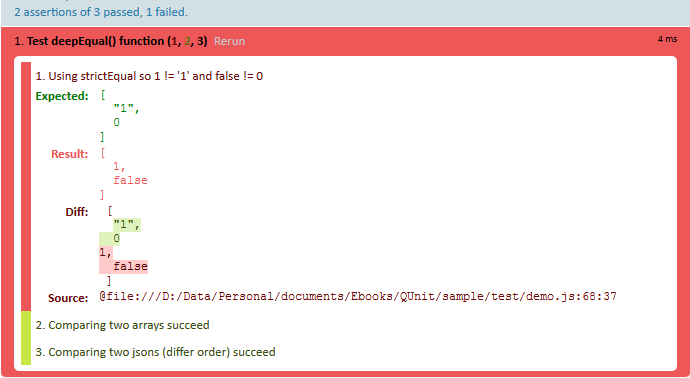
deepEqual(actual, expected[, message])

Referring to Assert – equal function to understand each parameters. They are same.

Sample as:



The result as:



Note: it uses strictEqual to compare each element in an array or json. So, false differs 0, true differs 1, 1 differs ‘1’, null differs undefined … It uses both type and value comparisons.

# Assert – notDeepEqual function

The opposite function of deepEqual function.

Structure:

notDeepEqual(actual, expected[, message])

Referring to Assert – equal function to understand each parameters. They are same.

# Assert – propEqual function

A strict type and value comparison of an object’s own properties.

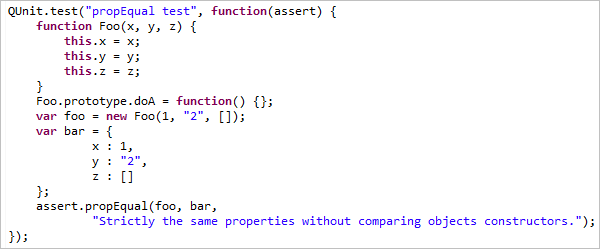
Structure:

propEqual(actual, expected[, message])

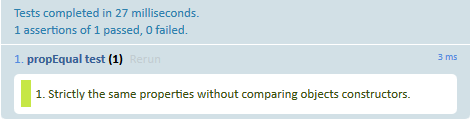
Referring to Assert – equal function to understand each parameters. They are same.

The propEqual assertion provides strictly (===) comparison of Object properties. Unlike deepEqual, this assertion can be used to compare two objects made with different constructors and prototypes.

Sample:



The result as:



# Assert – notPropEqual function

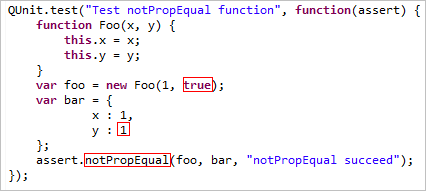
The opposite function of propEqual function.

Structure:

notPropEqual(actual, expected[, message])

Referring to Assert – equal function to understand each parameters. They are same.

Sample:



The result returns true value.

# Assert – expect function

Specify how many assertions are expected to run within a test.

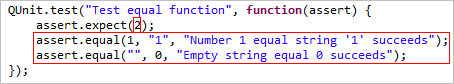
Structure:

expect(amount)

amount is a number.

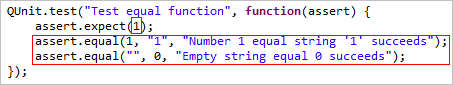
To ensure that an explicit number of assertions are run within any test, use this function to register an expected count. If the number of assertions run does not match the expected count, the test will fail.

Sample:

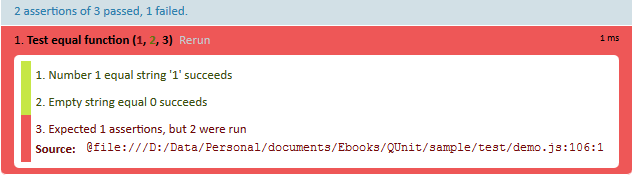


The result is true for expect function.

Sample:



The result as:



# Assert – push function

Report the result of a custom assertion. We can define a custom assertion. If the expectation function invokes push, QUnit will be notified of the result and report it accordingly.

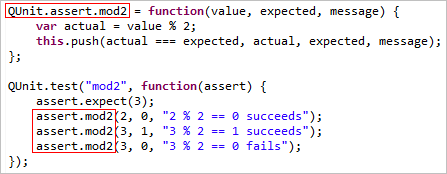
Structure:

push(result, actual, expected, message)

Decription:

* result is a boolean value
* actual, expected and message refer to Assert – equal function for more details.

Sample, define a custom mod2 assertion that tests if the provided numbers are equivalent in modulo 2 as:



The result as:



# Assert – throws function

Test if a callback throws an exception, and optionally compare the thrown error.

Structure:

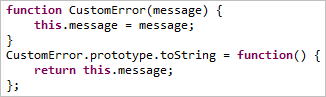
throws(block[, expected][, message])

Description:

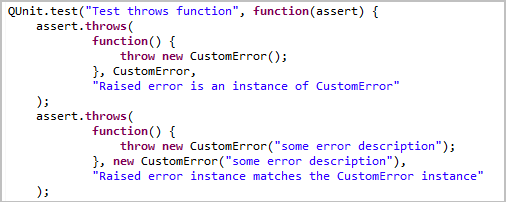
* block is a function
* expected and message refer to Assert – equal for more details

When testing code that is expected to throw an exception based on a specific set of circumstances, use assert.throws to catch the error object for testing and comparison.

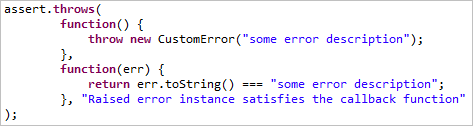
Sample define CustomError as:



And use it for testing as:



OR you can use callback as:



All above cases return true value.

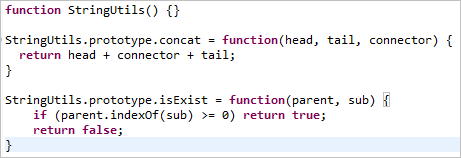
# Test suite with QUnit

From wiki, test suite is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviours.

## Setup source files

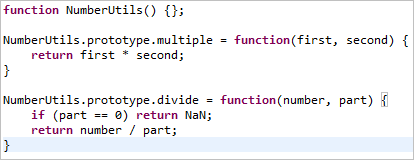
Sample, in the source\utils folder, I have two Javascript files: StringUtils.js and NumberUtils.js.

The content of StringUtils.js as:



We need to write some test cases to test concat and isExist functions.

The content of NumberUtils.js as:

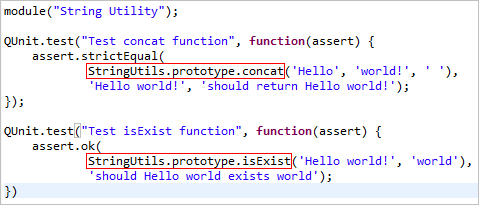


We also write some test cases to test multiple and divide functions.

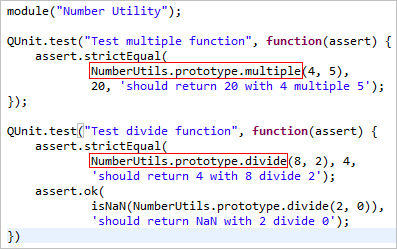
## Setup test files

We place test files at test/utils folder. To test StringUtils.js we will create new file with name StringUtils.spec.js.

The content of StringUtils.spec.js as:



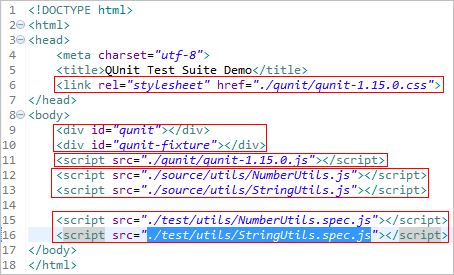
The content of NumberUtils.spec.js as:



Note: 2 divides 0 will return NaN. So we can use isNaN to test the result.

## Setup index file

Test suite should be used to test related things. I setup one html file to do this task as:



Description:

* Line 6 and line 11: use QUnit libraries
* Line 9 and 10: are required
* Line 12 and 13: include all source files need to test
* Line 15 and 16: include all test files

You can use browser to open the file and view the result as:



# Callbacks – QUnit.log function

Register a callback to fire whenever an assertion completes.

Structure:

QUnit.log(callback)

callback provides a single argument with the following properties:

* result: boolean, the result of an assertion, true means passed, false means failed.
* actual and expected: Object, parameter of assertion.
* message: String, parameter of assertion.
* source: String, the associated stacktrace, either from an exception or pointing to the source of the assertion. Depending on browser support for providing stacktraces, so can be undefined.
* module: String, the test module name of the assertion. OR undefined (if have no module).
* name: String, the test block name of the assertion.

Sample:



# Callbacks – QUnit.testStart function

Register a callback to fire whenever a test begins.

Structure:

QUnit.testStart(callback)

callback provides a single argument with the following properties:

* module: String, the test module name of test case OR undefined (if have no module).
* name: String, the name of test case.

Sample:



# Callbacks – QUnit.testDone function

To fire whenever a test ends.

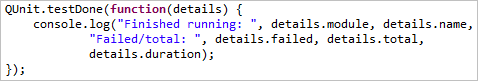
Structure:

QUnit.testDone(callback)

callback provides a single argument with the following properties:

* name: String, the test name
* module: String, the name of current module
* failed: Number, the number of failed assertions
* passed: Number, the number of passed assertions
* total: Number, the total number of assertions
* runtime: Number, the total runtime in millseconds of the test, including setup and teardown

Sample:



# Callbacks – QUnit.moduleStart function

To fire whenever a module begins.

Structure:

QUnit.moduleStart(callback)

callback provides a single argument with the following properties:

* name: String, the name of module to run

Sample:



# Callbacks – QUnit.moduleDone function

To fire whenever a module ends.

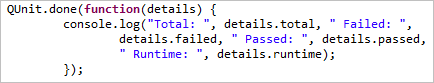
Structure:

QUnit.moduleDone(callback)

callback provides a single argument with the following properties:

* name: String, the module name
* failed: Number, the number of failed assertions
* passed: Number, the number of passed assertions
* total: Number, the total number of assertions

Sample:



# Callbacks – QUnit.begin function

To fire whenever the test suite begins.

Structure:

QUnit.begin(callback)

callback provides a single argument with the following properties:

* totalTests: Number, the number of total tests in the test suite

Sample:



# Callbacks – QUnit.done function

To fire whenever the test suite ends.

Structure:

QUnit.done(callback)

callback provides a single argument with the following properties:

* failed: Number, the number of failed assertions
* passed: Number, the number of passed assertions
* total: Number, the total number of assertions
* runtime: Number, the time in milliseconds it took tests to run from start to finish.

Sample:

