Test Driven development & Qunit

Anup Singh

Points to Discuss

- Unit Testing & Test Driven Development
- Debugging JS
- Writing Testable Code
- Designing own testing framework
- QUnit
- QUnit API
- Mocking AJAX
- Testing Forms
- Code coverage using Blankets
- Automated Testing (a brief introduction)

How do you test your JS?

1. Write your JavaScript code
2. See if it works in your favourite browser
3. Change something + [F5]
4. If it doesn't work repeat #3 until you make it work or you go crazy...
5. In case you made it work, discover few days/weeks later that it doesn't work in another browser

I think I'm going crazy…

Unit Testing

- In computer programming, unit testing is a procedure used to validate that individual modules or units of source code are working properly.
- Unit testing is used for
  - (i) Test Driven Development
  - (ii) Fixing bugs
  - (iii) Regression testing

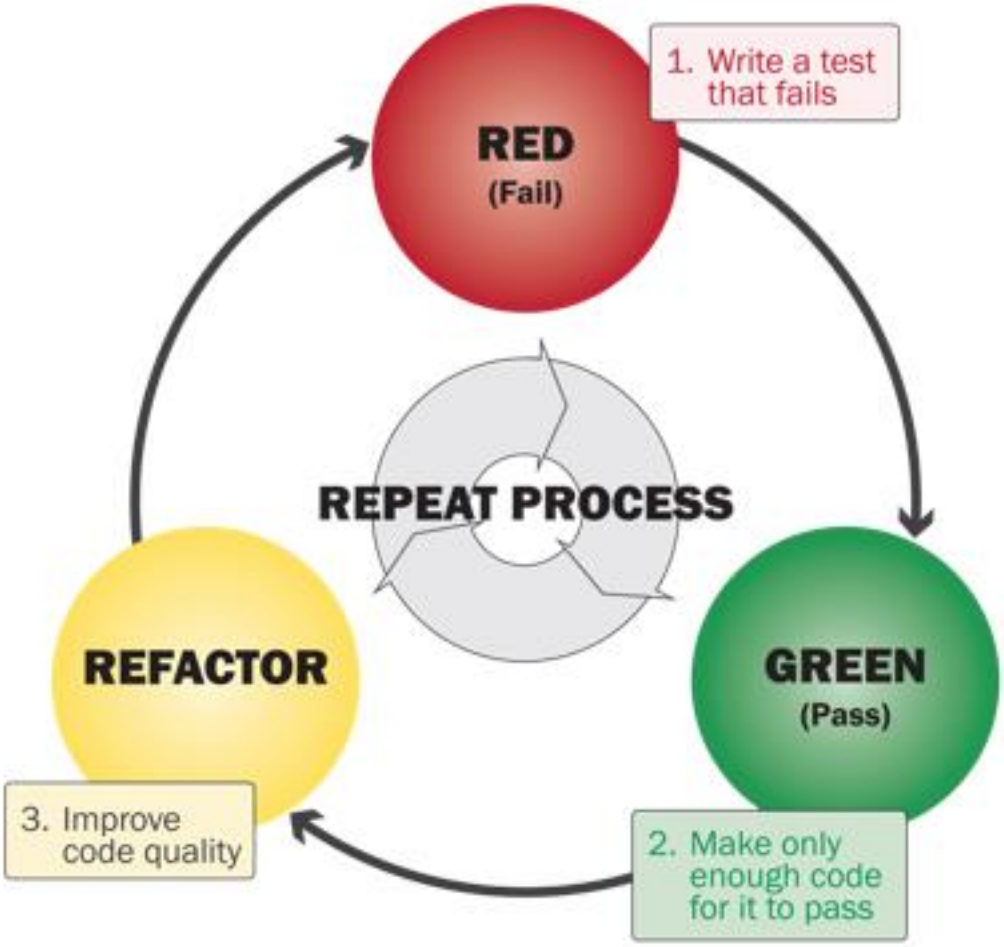
Test Driven Development

- Test-Driven Development (TDD) is a computer programming technique that involves repeatedly first writing a test case and then implementing only the code necessary to pass the test.
- Test-driven development is a method of designing software, not merely a method of testing.

Test Driven Development

- TDD in its simplest form is just this:
  - Write your tests
  - Watch them fail
  - Make them pass
  - Refactor
  - Repeat

The TDD Micro-Cycle



Fixing bugs/Regression Testing

- Fixing bugs
- Regression testing

What do you need?

- A Unit Testing framework
- Development Environment

Testing and debugging - Debugging code

Tools

- **Firebug** - The popular developer extension for Firefox that got the ball rolling. See http://getfirebug.org/.
- **IE Developer Tools** - Included in Internet Explorer 8 and later.
- **Opera Dragonfly** - Included in Opera 9.5 and newer. Also works with mobile versions of Opera.
- **WebKit Developer Tools** - Introduced in Safari 3, dramatically improved as of Safari 4, and now available in Chrome.

Logging - http://patik.com/blog/complete-cross-browser-console-log/

1. alert()
2. Console.log()
3. Common logging method that for all modern browsers

```
function log() {
    try {
        console.log.apply(console, arguments);
    } catch (e) {
        try {
            opera.postError.apply(opera, arguments);
        } catch (e) {
            alert(Array.prototype.join.call(arguments, " "));
        }
    }
}
```

1. Tries to log message using the most common method

2. Catches any failure in logging

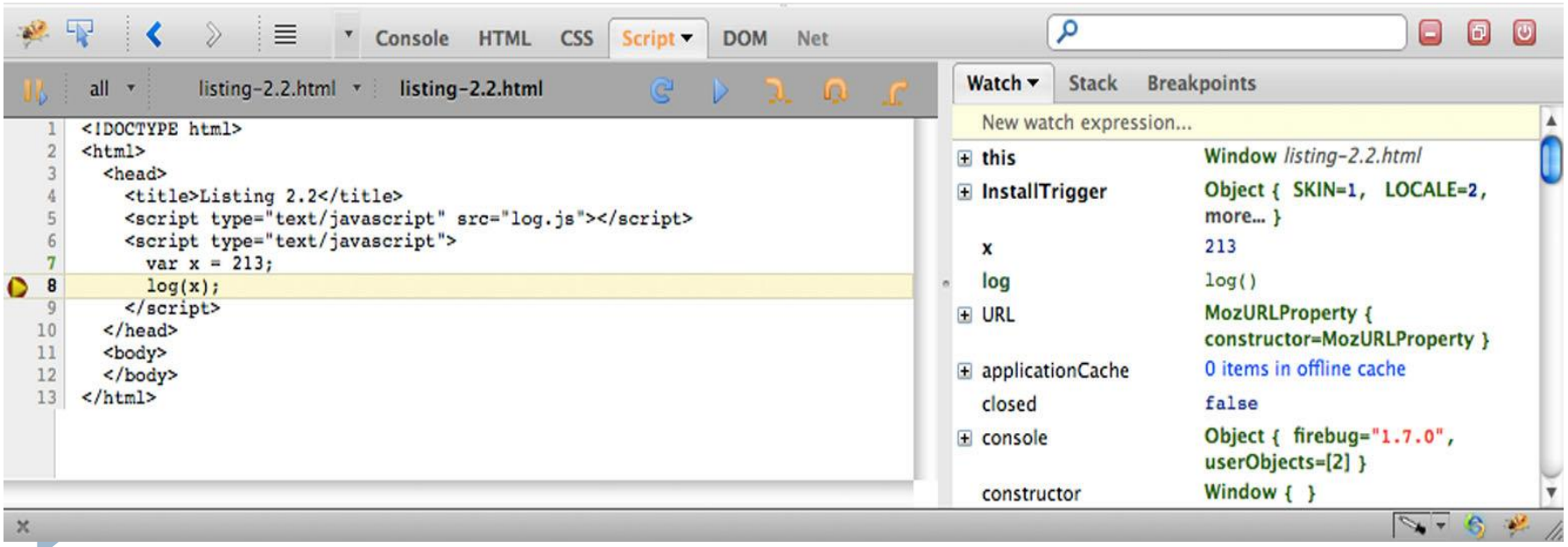
3. Tries to log the Opera way

Uses alert if all else fails

Testing and debugging - Breakpoints

Breakpoints allow us to halt execution at a specific line of code so we can take a gander at the state.

```
<!DOCTYPE html>
<html>
<head>
   <title>Listing 2.2</title>
   <script type="text/javascript" src="log.js"></script>
   <script type="text/javascript">
      var x = 213;
      log(x);
   </script>
</head>
<body>
</body>
</html>
```



Test generation

Good tests make Good code - Emphasis on the word **good**.
It's quite possible to have an extensive test suite that doesn't really help the quality of our code, if the tests are poorly constructed.

Good tests exhibit three important characteristics:

1. Repeatability - Our test results should be highly reproducible. Tests run repeatedly should always produce the exact same results. If test results are nondeterministic, how would we know which results are valid and which are invalid?

2. Simplicity - Our tests should focus on testing one thing. We should strive to remove as much HTML markup, CSS, or JavaScript as we can without disrupting the intent of the test case. The more we remove, the greater the likelihood that the test case will only be influenced by the specific code that we're testing.

3. Independence - Our tests should execute in isolation. We must avoid making the results from one test dependent upon another. Breaking tests down into the smallest possible

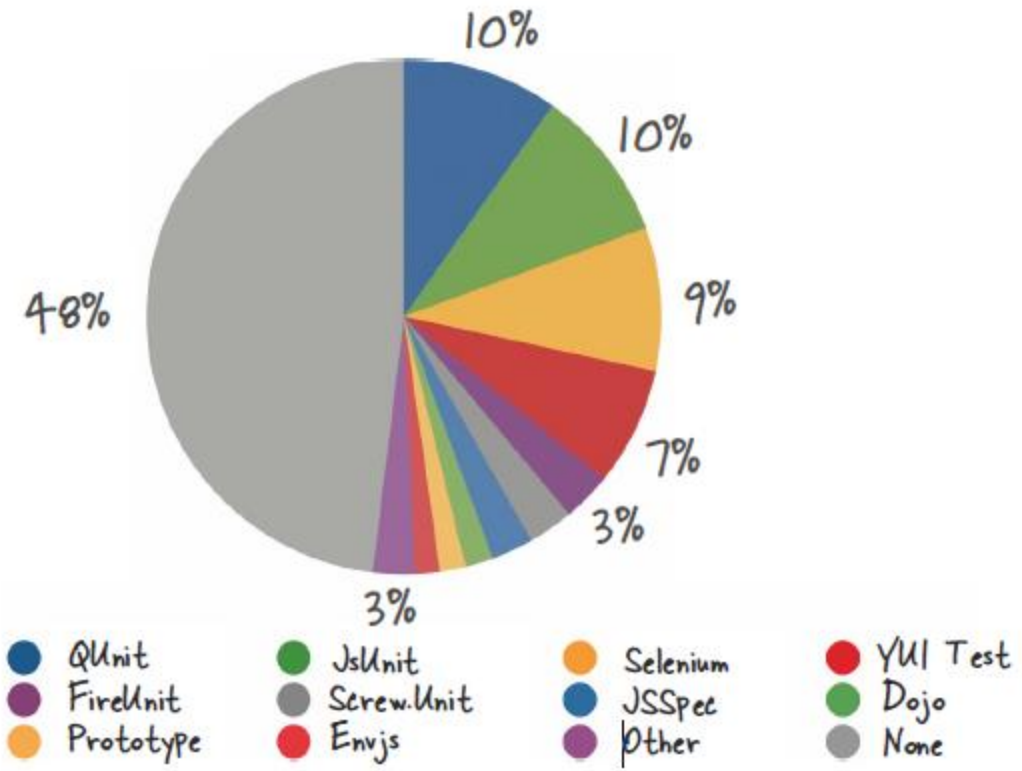
Testing Frameworks

A test suite should serve as a fundamental part of your development workflow, so you should pick a suite that works particularly well for your coding style and your code
base.

JavaScript unit testing framework features

- The ability to simulate browser behaviour (clicks, keypresses, and so on)
- Interactive control of tests (pausing and resuming tests)
- Handling asynchronous test timeouts
- The ability to filter which tests are to be executed

Market Share of Testing frameworks



The fundamentals of a test suite

The fundamentals of a test suite

1. Aggregate all the individual tests into a single unit
2. Run the in Bulk
3. Providing a single resource that can be run easily and repeatedly

How to construct a test suite

Q. Why would I want to build a new test suite, When There are already a number of good-quality suites to choose from?

A. Building your own test suite can serve as a good learning experience, especially when looking at how asynchronous testing works.

The Assertion – (assert.html)

1. The core of a unit-testing framework is its assertion method, usually named assert().

2. This takes a value—an expression whose premise is asserted—and a description that describes the purpose of the assertion. If the value evaluates to true

3. Either the assertion passes or it's considered a failure.

4. The associated message is usually logged with an appropriate pass/fail indicator.

Simple Implementation of JavaScript Assertion

```
<html>
  <head>
    <title>Test Suite</title>
    <script>

      function assert(value, desc) {
        var li = document.createElement("li");
        li.className = value ? "pass" : "fail";
        li.appendChild(document.createTextNode(desc));
        document.getElementById("results").appendChild(li);
      }

      window.onload = function() {
        assert(true, "The test suite is running.");
        assert(false, "Fail!");
      };
    </script>

    <style>
      #results li.pass { color: green; }
      #results li.fail { color: red; }
    </style>
  </head>

  <body>
    <ul id="results"></ul>
  </body>
</html>
```

1. Defines assert() method
2. Executes tests using assertions
3. Defines styles for results
4. Holds test results

More Examples -

- Custom/1_jq_test.html
- Custom/assert.html
- Custom/test_group.html

Test Groups – (test_group.html)

1. Grouping assertions together in a testing context to form test groups.

2. Test group will likely represent a collection of assertions as they relate to a single method in our API or application

3. If any assertion fails, then the entire test group is marked as failing

So what's the first step to sanity?

WRITE TESTABLE CODE

What's wrong with this code?

```
$( function() {
    $("#searchForm").on("submit", function(e) {
        e.preventDefault();

        var query = $("#searchInput").val();

        $.ajax({
            url: "./search",
            data: { "query": query },
            success: function(results) {
                results.forEach(function(item, index) {
                    $("#searchResults").append("<li> " + item + " </li>");
                });
            },
            error: function() {
                $("#messages").append("<p class='error'> ... </p>");
            }
        });
    });
});
```

Anonymous functions, within functions, within functions...

```
$( function() {
    $("#searchForm").on("submit", function(e) {
        e.preventDefault();

        var query = $("#searchInput").val();

        $.ajax({
            url: "./search",
            data: { "query": query },
            success: function(results) {
                results.forEach(function(item, index) {
                    $("#searchResults").append("<li> " + item + " </li>");
                });
            },
            error: function() {
                $("#messages").append("<p class='error'> ... </p>");
            }
        });
    });
});
```

I'll put functions in your functions...

All your DOM elements are belong to JS!

```
$( function() {
    $("#searchForm").on("submit", function(e) {
        e.preventDefault();

        var query = $("#searchInput").val();

        $.ajax({
            url: "./search",
            data: { "query": query },
            success: function(results) {
                results.forEach(function(item, index) {
                    $("#searchResults").append("<li> " + item + " </li>");
                });
            },
            error: function() {
                $("#messages").append("<p class='error'> ... </p>");
            }
        });
    });
});
```

Server URL coupling

```
$( function() {
    $("#searchForm").on("submit", function(e) {
        e.preventDefault();

        var query = $("#searchInput").val();

        $.ajax({
            url: "./search",
            data: { "query": query },
            success: function(results) {
                results.forEach(function(item, index) {
                    $("#searchResults").append("<li> " + item + " </li>");
                });
            },
            error: function() {
                $("#messages").append("<p class='error'> ... </p>");
            }
        });
    });
});
```

Refactoring…

Refactoring…

```
window.jk = (window.jk || {});  // Mind the namespacing

jk.initSearch = function( form, input ) {

};

jk.doSearch = function( value, callback ) {

};

jk.handleResults = function( results, resultsNode ) {

};

jk.handleAjaxError = function( xhr, jqStatus, statusError ) {
};
```

Now that's better...

```
jk.initSearch = function( form, input ) {
        $(form).on( "submit", function submitHandler(e) {
                e.preventDefault();
                jk.doSearch( $(input).val() );
        });
};

$(document).ready( function initPage() {
        jk.initSearch( "#searchForm", "#searchInput" );
});
```

Now that's better...

```
jk.doSearch = function( value, callback ) {
        var xhr = $.ajax({
                url: "/search",
                data: { "query": value },

                success: function successHandler(data) {
                        jk.handleResults(data);
                        callback(data);
                },

                error: function errorHandler(xhr, status) {
                        var msg = jk.handleAjaxError(xhr, status);
                        callback({ "error": msg });
                }
        });

        return xhr;
};
```

Now that's better...

```
jk.handleResults = function( results, resultsNode ) {
        var items = "";

        resultsNode = $( resultsNode || "#searchResults" );

        results.forEach( function resultLoop( item, index ) {
                items += "<li> " + item + " </li>";
        });

        return resultsNode.append(items);
};
```

Now what about testing?

Popular JS Unit-testing frameworks:

- **QUnit**
- **Jasmine**
- **UnitJS**
- **JsUnit *(no longer actively maintained)***
- **Some other – see:**
  http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#JavaScript

What's about QUnit?

- QUnit is a powerful, easy-to-use JavaScript unit testing framework.
- Supports the same browsers as jQuery 1.x. That's IE6+ and Current - 1 for Chrome, Firefox, Safari and Opera.
- Used by the jQuery project to test jQuery, jQuery UI, jQuery Mobile
- Can be used to test any generic JavaScript code, including itself
- Very easy to install – just include JS & CSS file in your HTML

Minimal setup: The tests.htm file

```
1   <!DOCTYPE html>
2   <html>
3   <head>
4       <title>JS Tests run here</title>
5       <link rel="stylesheet" href="qunit.css" />
6   </head>
7   <body>
8       <div id="qunit"></div>
9       <div id="qunit-fixture"></div>
10      <script src="http://code.jquery.com/qunit/qunit-1.14.0.js"></script>
11      <script src="./functions.js"></script>
12      <script src="./tests.js"></script>
13  </body>
14  </html>
```

Getting Started

- Just qunit.js, qunit.css, and a little bit of HTML
- A Quick Demo: http://jsfiddle.net/anups/4cweqhhs/

QUnit API

- Test
- Assert
- Async Control
- Callback

Test : QUnit.test(name, test)

- Adds a test to run.
- Testing the most common, synchronous code

```
Qunit.test("name of the test", function() {
    //write down the assertions
});
```

function: Function to close over assertions

Test : expect()

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

If the number of assertions run does not match the expected count, the test will fail.

```
test(”expected assertions", function() {
    expect( 2 );
    //two assertions are expected
});
```

Asserts

- ok
- Equal
- notEqual
- strictEqual
- notStrictEqual
- deepEqual
- notDeepEqual
- throws

Assert - ok()

ok(state, message)

A boolean check, passes if the first argument is truthy.

```
test("ok”, function() {
    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" );
});
```

Assert - equal()

equal(actual, expected, message)

A comparison assertion that passes if actual == expected.

```
test("equal”, function() {
    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" );
});
```

Assert - notEqual()

notEqual(actual, expected, message)

A comparison assertion that passes if actual != expected.

```
test("notEqual”, function() {
    expect(3);
    var actual = 5 - 4;
    notEqual(actual, 0, "passes because 1 != 0");
    notEqual(actual, false, "passes because 1 != false");
    notEqual(actual, true, "fails because 1 == true");
});
```

Assert - strictEqual()

strictEqual(actual, expected, message)

A comparison assertion that passes if actual === expected.

```
test("notEqual”, function() {
    expect(3);
    var actual = 5 - 4;
    strictEqual(actual, 1, "passes because 1 === 1");
    strictEqual(actual, true, "fails because 1 !== true");
    strictEqual(actual, false, "fails because 1 !== false");
});
```

Assert - notStrictEqual()

notStrictEqual(actual, expected, message)

A comparison assertion that passes if actual !== expected.

```
test("notStrictEqual”, function() {
    expect(3);
    var actual = 5 - 4;
    notStrictEqual(actual, 1, "fails because 1 === 1");
    notStrictEqual(actual, true, "passes because 1 !== true");
    notStrictEqual(actual, false, "passes because 1 !== false");
});
```

Assert - deepEqual ()

deepEqual(actual, expected, message)

Recursive comparison assertion, working on primitives, arrays and objects, using ===.

```
test("deepEqual”, function() {
    expect(3);
    var actual = {a: 1};
    equal( actual, {a: 1}, "fails because objects are different");
    deepEqual(actual, {a: 1}, "passes because objects are equivalent");
    deepEqual(actual, {a: "1"}, "fails because '1' !== 1");

});
```

Assert - notDeepEqual()

notDeepEqual(actual, expected, message)

Recursive comparison assertion. The result of deepEqual, inverted.

```
test("notDeepEqual", function() {
      expect(3);
       var actual = {a: 1};
       notEqual( actual, {a: 1}, "passes because objects are different");     notDeepEqual(actual, {a: 1}, "fails because objects are equivalent"); notDeepEqual(actual, {a: "1"}, "passes because '1' !== 1");
});
```

Assert - throws()

Assertion to test if a callback throws an exception when run and optionally compare the thrown error.

```
test("throws", function() {
    expect(3);
    throws(
    function() { throw new Error("Look me, I'm an error!"); },
    "passes because an error is thrown inside the callback"
    );

    throws(
    function() { x // ReferenceError: x is not defined },
    "passes because an error is thrown inside the callback"
    );

    throws (
    function() { var a = 1; },
    "fails because no error is thrown inside the callback"
    );
});
```

Tests Should be Atomic

- Execution order cannot be guaranteed!
- Each test should be independent from one another.
- QUnit.test() is used to keep test cases atomic.

Async Control : QUnit.asyncTest

For testing asynchronous code, QUnit.asyncTest will automatically stop the test runner and wait for your code to call QUnit.start() to continue.

The following illustrates an asynchronous test that waits 1 second before resuming

```
QUnit.asyncTest( "asynchronous test: one second later!", function(
assert ) {
    expect( 1 );
    setTimeout(function() {
        assert.ok( true, "Passed and ready to resume!" );
        QUnit.start(); }, 1000);
});
```

Async Control : QUnit.stop()

Increase the number of QUnit.start() calls the testrunner should wait for before continuing.

When your async test has multiple exit points, call QUnit.stop() multiple times or use the increment argument.

```
QUnit.test( "a test", function( assert ){
    QUnit.stop();
    setTimeout(function(){
        assert.equals("somedata" , "someExpectedValue" );
        QUnit.start(); }, 150 );
 });
```

Grouping Tests : QUnit.module()

It groups tests together to keep them logically organized and be able to run a specific group of tests on their own.

All tests that occur after a call to QUnit.module() will be grouped into that module. The test names will all be preceded by the module name in the test results.

```
QUnit.module( "group a" );//tests for module a
QUnit.module( "group b" );//test for module b
```

Grouping Tests : QUnit.module()

QUnit.module() can also be used to extract common code from tests within that module.

The QUnit.module() function takes an optional second parameter to define functions to run before and after each test within the module

```
QUnit.module( "module", {
    setup: function( assert )
        {//any setup task},
    teardown: function( assert )
        {//task to be performed after test completion}
    });

    QUnit.test( "test with setup and teardown", function()
    {
        //test cases
    });
```

Callbacks

When integrating QUnit into other tools like CI servers, use these callbacks as an API to read test results.

- QUnit.begin()
- QUnit.done()
- QUnit.moduleStart()
- QUnit.moduleDone()
- QUnit.testStart()
- QUnit.testDone()

Mocking Ajax with the JQuery Mockjax Library

- **MockAjax**

Downlaod it from https://github.com/mobz/mock-ajax

MockAjax is an mock **XMLHttpRequest** implementation designed to allow asynchronous XHR requests to be run inside a synchronous testing framework.

Mockjax benifits

- Mock out Ajax requests, so a server is not required to test server dependent code
- Allow asynchronous requests to run synchronously, allowing tests to run much faster than normal
- Allow you to test multiple simultaneous inflight requests
- Allow tricky edge cases to be tested with ease
  - server timeouts
  - receiving server responses out of order
  - 404's
  - server errors
- Allows tests that use setTimeout to run instantly and reliably
- also supports asynchronous and synchronous ajax without blocking

Mockjax Example - Intercept Request

```
asyncTest('Intercept and proxy (sub-ajax request)', function() {
    $.mockjax({
        url: '/proxy',
        proxy: 'mock-ajax-response/test_proxy.json'
    });
    $.ajax({
        url: '/proxy',
        dataType: 'json',
        success: function(json) {
            ok(json && json.proxy, 'Proxy request succeeded');
        },
        error: function(){
            ok( false, 'Error callback executed');
        },
        complete: function() {
            start();
        }
    });
});
```

More examples - mock-ajax.html

Testing User Action

Testing User Action

- 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
  - Trigger() event using jQuery trigger
  - triggerHandler of jQuery can be used if you don’t wanr the native browser event to be triggered
- If That is not enough
  - https://github.com/bitovi/syn
  - http://tinymce.ephox.com/jsrobot
  - http://dojotoolkit.org/reference-guide/1.8/util/dohrobot.html
  - https://github.com/gtramontina/keyvent.js

Testing User Action

- form.htm
- Form-test.html

Testing Scope

Testing Scope - Variable

```
function outer() {
    var a = 1;
    function inner() {}
    var b = 2;
    if (a == 1) {
      var c = 3;
    }
QUnit.test( "Testing Variable Scope in function", function( assert ){
            assert.ok(typeof c === 'number', "c is in scope");
});
}
QUnit.test( "Testing Variable Scope", function( assert ){
    assert.ok(true, "some descriptive text");
    assert.ok(typeof inner !== 'function', "inner() is not in scope");
    assert.ok(typeof outer === 'function', "outer() is in scope");
    assert.ok(typeof b !== 'number', "b is not in scope");
    assert.ok(typeof a !== 'number', "a is not in scope");
    outer();
});
```

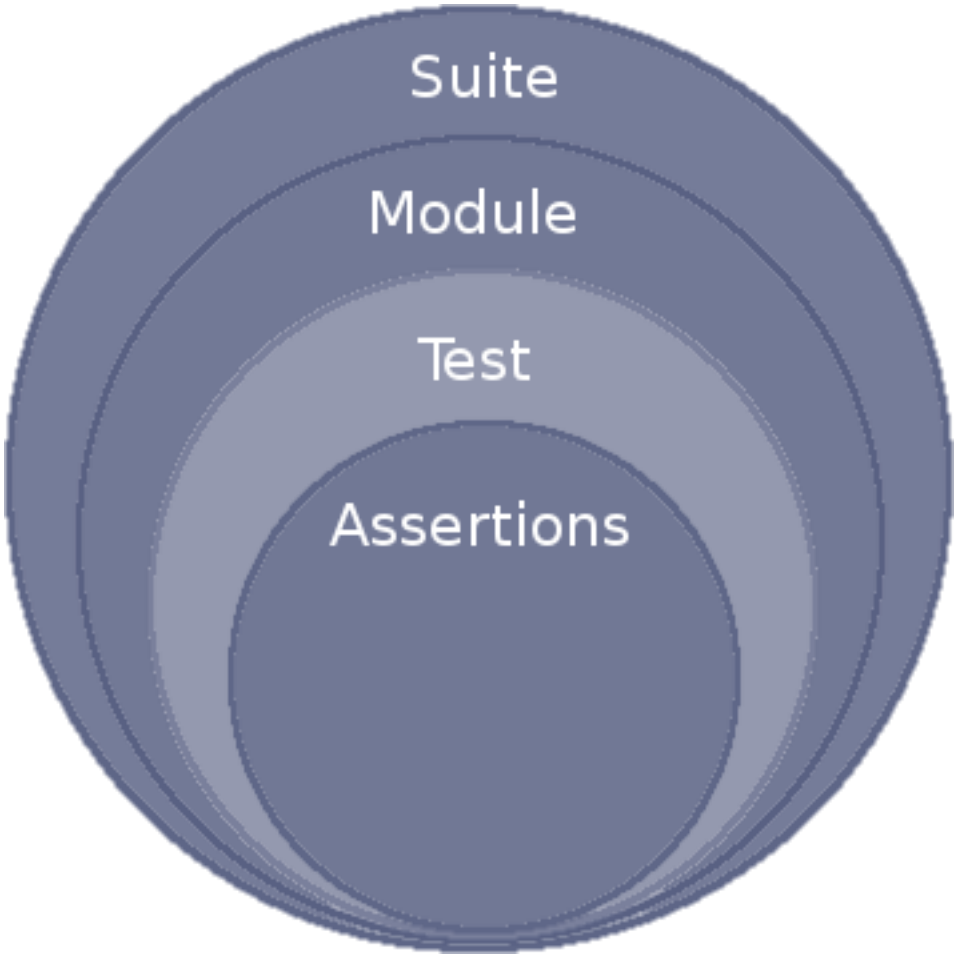
Testing Scope – Function p1

```
function isFunction() {     return true; }
var doSomething = function () {      return true; };
window.testFunction = function () {return true; };

function outer() {
  QUnit.test("Testing Function Scope before declaration", function( assert ){
    assert.ok(typeof inner === "function", "inner() in scope before declaration");
  });

function inner() {}

 QUnit.test("Testing Function Scope after declaration", function( assert ){
         assert.ok(typeof inner === "function", "inner() in scope after declaration");
         assert.ok(window.inner === undefined, "inner() in global scope");
 });
}
```

Testing Scope – Function p2

```
QUnit.test("Testing Function Scope", function( assert ){
   assert.ok(typeof window.isFunction === "function", "isFunction() defined");
   assert.ok(isFunction.name === "isFunction", "isFunction() has a name");
   assert.ok(typeof window.doSomething === "function", "doSomething() defined");
   assert.ok(doSomething.name === "", "doSomething() has no name");
   assert.ok(typeof window.testFunction === "function", "testFunction() defined");
outer();
   assert.ok(window.inner === undefined, "inner() still not in global scope");
   window.doCoding = function writeCode() {
          return true;
   };
   assert.ok(window.doCoding.name === 'writeCode', "wieldSword's real name is
   writeCode");

});
```

QUnit Test - Suite



NodeJS
Integration and Automation

Node :QUnit

1. Install nodejs
2. Install qunit node module

   npm install qunit

testrunner.js

```
var runner = require("../../node/node_modules/qunit");
runner.run({
        code : "/full/path/to/public/js/main.js",
        tests : "/full/path/to/tests/js/tests.js"
});
```

Node command

node tests/js/testrunner.js

Automated Testing

Install Node

Using Node Package Manager install Grunt

Install QUnit module to your project directory(
)

```
└ project
  ├ src   // plugin source, project files, etc
  ├ tests // we'll be working in here mostly
  │ ├ lib
  │ │ ├ jquery-1.x.x.min.js // if you need it (QUnit doesn't)
  │ │ ├ qunit-1.10.0.js
  │ │ └ qunit-1.10.0.css
  │ ├ index.html  // our QUnit test specification
  │ └ tests.js    // your tests (could be split into multiple files)
  ├ Gruntfile.js  // you'll create this, we'll get to it shortly
  ├ package.json  // to specify our project dependencies
```

Automated Testing (contd.)

```
//package.json
{
  "name": "projectName",
  "version": "1.0.0",
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-qunit": ">=0.2.1",
 }
}

//Gruntfile.js
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    taskName: {qunit: {
      all: ['tests/*.html']
    }}
 });
  grunt.loadNpmTasks('grunt-contrib-qunit');
  grunt.registerTask('default', ['qunit']);
 };
```


Thanks