**## Getting Started with JavaScript Unit Testing Using Mocha**

Code to accompany the following article: **http://www.sitepoint.com/unit-test-javascript-mocha-chai**

To run the code samples:

- Clone the repo

- Run `npm install`

- Open `testrunner.html` in browser

- Alternatively, uncomment Node specific code and run `mocha` from the command line (this requires Mocha to be installed globally).

**Setting up a Directory Structure**

**Step 1 Install Node JS on localPC**

Download Node js and install on Local PC

**https://nodejs.org/en/**

**https://nodejs.org/en/**

**Step 2**

With Node installed, open up a terminal or command line in your project’s directory.

If you want to test code in the browser, run **npm install mocha chai --save-dev**

If you want to test Node.js code, in addition to the above, run npm install -g mocha

This installs the packages mocha and chai. Mocha is the library that allows us to run tests, and Chai contains some helpful functions that we’ll use to verify our test results.

## Setting up a Test Runner

In order to run our tests in a browser, we need to set up a simple HTML page to be our test runner page. The page loads Mocha, the testing libraries and our actual test files. To run the tests, we’ll simply open the runner in a browser.

If you’re using Node.js, you can skip this step. Node.js unit tests can be run using the command mocha, assuming you’ve followed the recommended directory structure.

Below is the code we’ll use for the test runner. I’ll save this file as testrunner.html.

The important bits in the test runner are:

* We load Mocha’s CSS styles to give our test results nice formatting.
* We create a div with the ID mocha. This is where the test results are inserted.
* We load Mocha and Chai. They are located in subfolders of the node\_modules folder since we installed them via npm.
* By calling mocha.setup, we make Mocha’s testing helpers available.
* Then, we load the code we want to test and the test files. We don’t have anything here just yet.
* Last, we call mocha.run to run the tests. Make sure you call this *after* loading the source and test files.

## The Basic Test Building Blocks

**Now that we can run tests, let’s start writing some**.

We’ll begin by creating a new file test/arrayTest.js. An individual test file such as this one is known as a **test case**. I’m calling it arrayTest.js because for this example, we’ll be testing some basic array functionality.

Every test case file follows the same basic pattern. First, you have a describe block:

describe('Array', function() {

// Further code for tests goes here

});

describe is used to group individual tests. The first parameter should indicate what we’re testing — in this case, since we’re going to test array functions, I’ve passed in the string 'Array'.

Secondly, inside the describe, we’ll have it blocks:

describe('Array', function() {

it('should start empty', function() {

// Test implementation goes here

});

// We can have more its here

});

it is used to create the actual tests. The first parameter to it should provide a human-readable description of the test. For example, we can read the above as “it should start empty”, which is a good description of how arrays should behave. The code to implement the test is then written inside the function passed to it.

All Mocha tests are built from these same building blocks, and they follow this same basic pattern.

* First, we use describe to say what we’re testing – for example, “describe how array should work”.
* Then, we use a number of it functions to create the individual tests – each it should explain one specific behavior, such as “it should start empty” for our array case above.

## Writing the Test Code

Now that we know how to structure the test case, let’s jump into the fun part — implementing the test.

Since we are testing that an array should start empty, we need to create an array and then ensure it’s empty. The implementation for this test is quite simple:

var assert = chai.assert;

describe('Array', function() {

it('should start empty', function() {

var arr = [];

assert.equal(arr.length, 0);

});

});

Note on the first line, we set up the assert variable. This is just so we don’t need to keep typing chai.assert everywhere.

In the it function, we create an array and check its length. Although simple, this is a good example of how tests work.

First, you have something you’re testing — this is called the System Under Test or SUT. Then, if necessary, you do something with the SUT. In this test, we’re not doing anything, since we’re checking the array starts as empty.

The last thing in a test should be the validation — an assertion which checks the result. Here, we are using assert.equalto do this. Most assertion functions take parameters in the same order: First the “actual” value, and then the “expected” value.

* The actual value is the result from your test code, so in this case arr.length
* The expected value is what the result should be. Since an array should begin empty, the expected value in this test is 0

Chai also offers two different styles of writing assertions, but we’re using [assert](http://chaijs.com/api/assert/) to keep things simple for now. When you become more experienced with writing tests, you might want to use the [expect assertions](http://chaijs.com/api/bdd/) instead, as they provide some more flexibility.

## Running the Test

In order to run this test, we need to add it to the test runner file we created earlier.

If you’re using Node.js, you can skip this step, and use the command mocha to run the test. You’ll see the test results in the terminal.

Otherwise, to add this test to the runner, simply add:

<script src="test/arrayTest.js"></script>

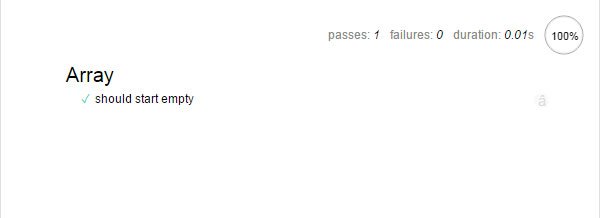
Below:

<!-- load your test files here -->

Once you’ve added the script, you can then load the test runner page in your browser of choice.

## The Test Results

When you run your tests, the test results will look something like this:



Note that what we entered into the describe and it functions show up in the output — the tests are grouped under the description. Note that it’s also possible to nest describe blocks to create further sub-groupings.

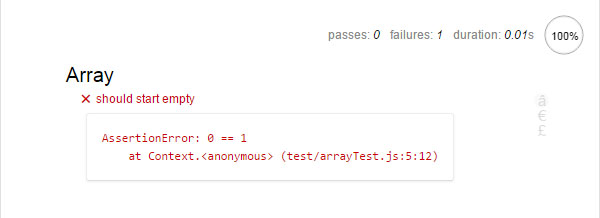
Let’s take a look at what a failing test looks like.

On the line in the test that says:

assert.equal(arr.length, 0);

Replace the number 0 with 1. This makes the test fail, as the array’s length no longer matches the expected value.

If you run the tests again, you’ll see the failing test in red with a description of what went wrong.



One of the benefits of tests is that they help you find bugs quicker, however this error is not very helpful in that respect. We can fix it though.

Most of the assertion functions can also take an optional message parameter. This is the message that is displayed when the assertion fails. It’s a good idea to use this parameter to make the error message easier to understand.

We can add a message to our assertion like so:

assert.equal(arr.length, 1, 'Array length was not 0');

If you re-run tests, the custom message will appear instead of the default.

Let’s switch the assertion back to the way it was — replace 1 with 0, and run the tests again to make sure they pass.

## Putting It Together

So far we’ve looked at fairly simple examples. Let’s put what we’ve learned into practice and see how we would test a more realistic piece of code.

Here’s a function which adds a CSS class to an element. This should go in a new file js/className.js.

function addClass(el, newClass) {

if(el.className.indexOf(newClass) === -1) {

el.className += newClass;

}

}

To make it a bit more interesting, I made it add a new class only when that class doesn’t exist in an element’s classNameproperty — who wants to see <div class="hello hello hello hello"> after all?

In the best case, we would write tests for this function before we write the code. But [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development) is a complex topic, and for now we just want to focus on writing tests.

To get started, let’s recall the basic idea behind unit tests: We give the function certain inputs and then verify the function behaves as expected. So what are the inputs and behaviors for this function?

Given an element and a class name:

* if the element’s className property does not contain the class name, it should be added.
* if the element’s className property does contain the class name, it should not be added.

Let’s translate these cases into two tests. In the test directory, create a new file classNameTest.js and add the following:

describe('addClass', function() {

it('should add class to element');

it('should not add a class which already exists');

});

We changed the wording slightly to the “it should do X” form used with tests. This means that it reads a bit nicer, but is essentially still the same human-readable form we listed above. It’s usually not much more difficult than this to go from idea to test.

But wait, where are the test functions? Well, when we omit the second parameter to it, Mocha marks these tests as pending in the test results. This is a convenient way to set up a number of tests — kind of like a todo list of what you intend to write.

Let’s continue by implementing the first test.

describe('addClass', function() {

it('should add class to element', function() {

var element = { className: '' };

addClass(element, 'test-class');

assert.equal(element.className, 'test-class');

});

it('should not add a class which already exists');

});

In this test, we create an element variable and pass it as a parameter to the addClass function, along with a string test-class (the new class to add). Then, we check the class is included in the value using an assertion.

Again, we went from our initial idea — given an element and a class name, it should be added into the class list — and translated it into code in a fairly straightforward manner.

Although this function is designed to work with DOM elements, we’re using a plain JS object here. Sometimes we can make use of JavaScript’s dynamic nature in this fashion to simplify our tests. If we didn’t do this, we would need to create an actual element and it would complicate our test code. As an additional benefit, since we don’t use DOM, we can also run this test within Node.js if we so wish.

### Running the Tests in the Browser

To run the test in the browser, you’ll need to add className.js and classNameTest.js to the runner:

<!-- load code you want to test here -->

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

<!-- load your test files here -->

<script src="test/classNameTest.js"></script>

You should now see one test pass and another test show up as pending, as is demonstrated by the following CodePen. Note that the code differs slightly from the example in order to make the code work within the CodePen environment.