Javascript (Https://Www.Sitepoint.Com/Javascript/) > January 12, 2018 > By Jani Hartikainen (https://www.sitepoint.com/author/jhartikainen/)

Unit Test Your JavaScript Using Mocha and Chai

This article was peer reviewed by Panayiotis «pvgr» Velisarakos (https://github.com/pvgr), Mark Brown (http://www.sitepoint.com/author/mbrown) and Tom Greco (http://www.sitepoint.com/author/tgreco/). Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

Have you ever made some changes to your code, and later found it caused something else to break?

I'm sure most of us have. This is almost inevitable, especially when you have a larger amount of code. One thing depends on another, and then changing it breaks something else as a result.

But what if that didn't happen? What if you had a way of knowing when something breaks as a result of some change? That would be pretty great. You could modify your code without having to worry about breaking anything, you'd have fewer bugs and you'd spend less time debugging.

That's where unit tests shine. They will *automatically* detect any problems in the code for you. Make a change, run your tests and if anything breaks, you'll immediately know what happened, where the problem is *and* what the correct behavior should be. This completely eliminates any guesswork!

In this article, I'll show you how to get started unit testing your JavaScript code. The examples and techniques shown in this article can be applied to both browser-based code and Node.js code.

The code for this tutorial is available from our GitHub repo (https://github.com/sitepoint-editors/mocha-unit-testing).

What Is Unit Testing

When you test your codebase, you take a piece of code — typically a function — and verify it behaves correctly in a specific situation. Unit testing is a structured and automated way of doing this. As a result, the more tests you write, the bigger the benefit you receive. You will also have a greater level of confidence in your codebase as you continue to develop it.

The core idea with unit testing is to test a function's behavior when giving it a certain set of inputs. You call a function with certain parameters, and check you got the correct result.

```
 // Given 1 and 10 as inputs...
var result = Math.max(1, 10);

// ...we should receive 10 as the output
if(result !== 10) {
  throw new Error('Failed');
}
```

In practice, tests can sometimes be more complex. For example, if your function makes an Ajax request, the test needs some more set up, but the same principle of "given certain inputs, we expect a specific outcome" still applies.

Setting up the Tools

For this article, we'll be using Mocha. It's easy to get started with, can be used for both browser-based testing and Node.js testing, and it plays nicely with other testing tools.

The easiest way to install Mocha is through npm (for which we also need to install Node.js (https://nodejs.org/en/)). If you're unsure about how to install either npm or Node on your system, consult our tutorial: A Beginner's Guide to npm — the Node Package Manager (http://www.sitepoint.com/beginners-guide-node-package-manager/)

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 (https://mochajs.org/) is the library that allows us to run tests, and Chai (http://chaijs.com/) contains some helpful functions that we'll use to verify our test results.

Testing on Node.js vs Testing in the Browser

The examples that follow are designed to work if running the tests in a browser. If you want to unit test your Node.js application, follow these steps.

> For Node, you don't need the test runner file.
> To include Chai, add `var chai = require('chai');` at the top of the test file.
> Run the tests using the `mocha` command, instead of opening a browser.

Setting up a Directory Structure

You should put your tests in a separate directory from your main code files. This makes it easier to structure them, for example if you want to add other types of tests in the future (such as integration tests (https://en.wikipedia.org/wiki/Integration_testing) or functional tests (https://en.wikipedia.org/wiki/Functional_testing)).

The most popular practice with JavaScript code is to have a directory called `test/` in your project's root directory. Then, each test file is placed under `test/someModuleTest.js`. Optionally, you can also use directories inside `test/`, but I recommend keeping things simple — you can always change it later if necessary.

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`.

```
 <!DOCTYPE html>
<html>
  <head>
    <title>Mocha Tests</title>
    <link rel="stylesheet" href="node_modules/mocha/mocha.css">
  </head>
  <body>
    <div id="mocha"></div>
    <script src="node_modules/mocha/mocha.js"></script>
    <script src="node_modules/chai/chai.js"></script>
    <script>mocha.setup('bdd')</script>

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

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

    <script>
      mocha.run();
    </script>
  </body>
</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.equal` to 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>
```

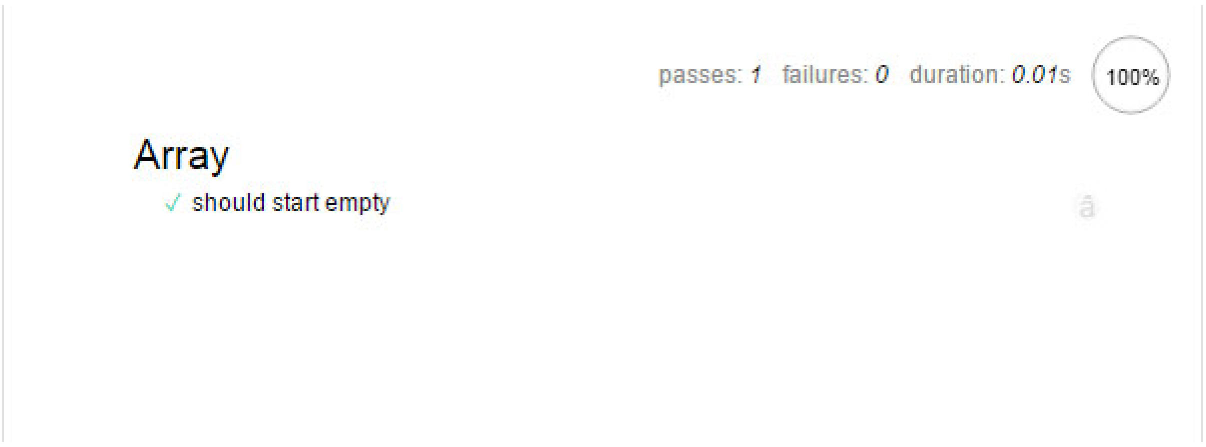
Below:

```
<!-- 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:



passes: 1 failures: 0 duration: 0.01s 100%

Array
✓ should start empty

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.

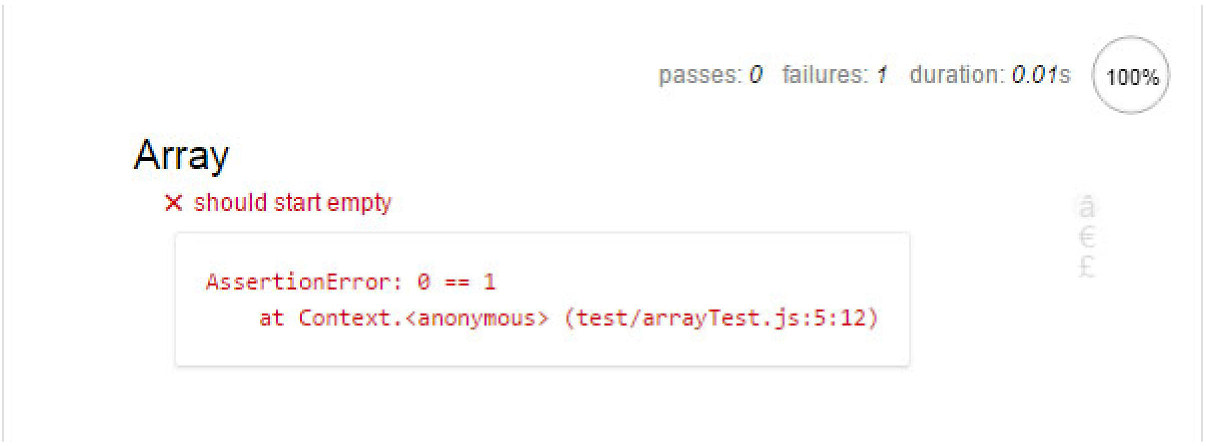
Let's take a look at what a failing test looks like.

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.



```
AssertionError: 0 == 1
    at Context.<anonymous> (test/arrayTest.js:5:12)
```

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 `className` property — 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.

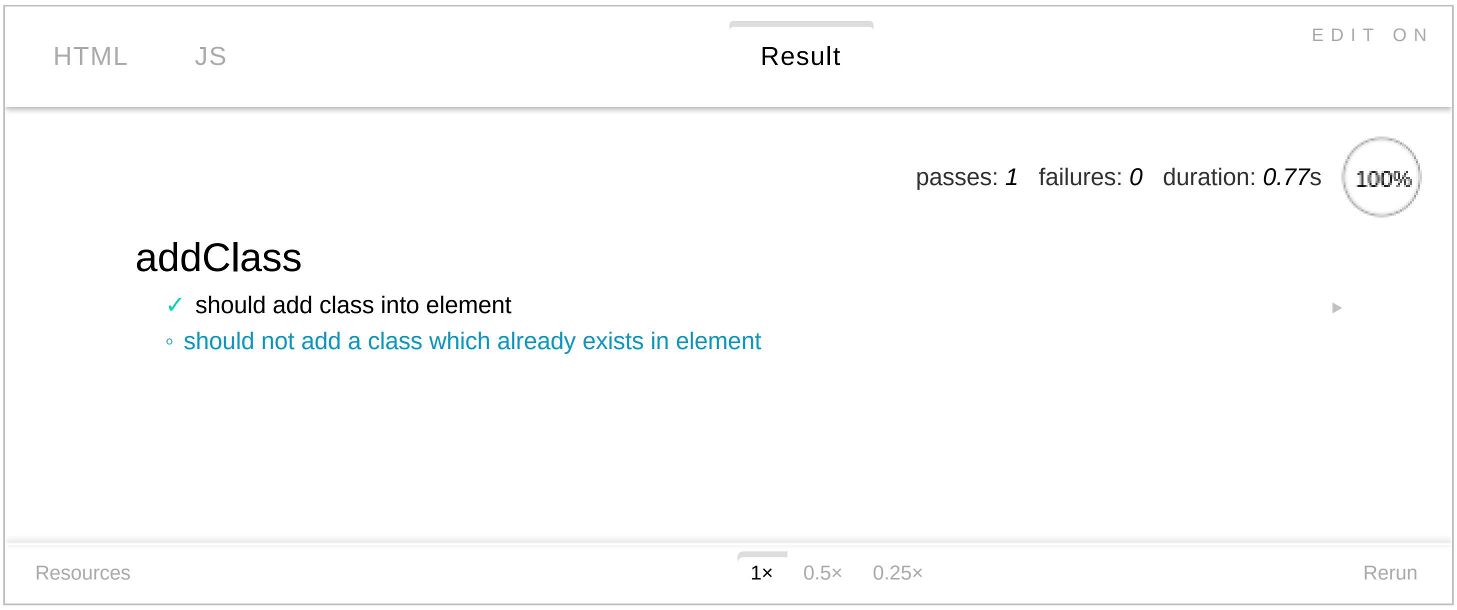
Although this function is designed to work with DOM elements, we're using a plain JS object here. Sometimes we can make use of JavaScript's dynamic nature in this fashion to simplify our tests. If we didn't do this, we would need to create an actual element and it would complicate our test code. As an 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.



passes: 1 failures: 0 duration: 0.77s

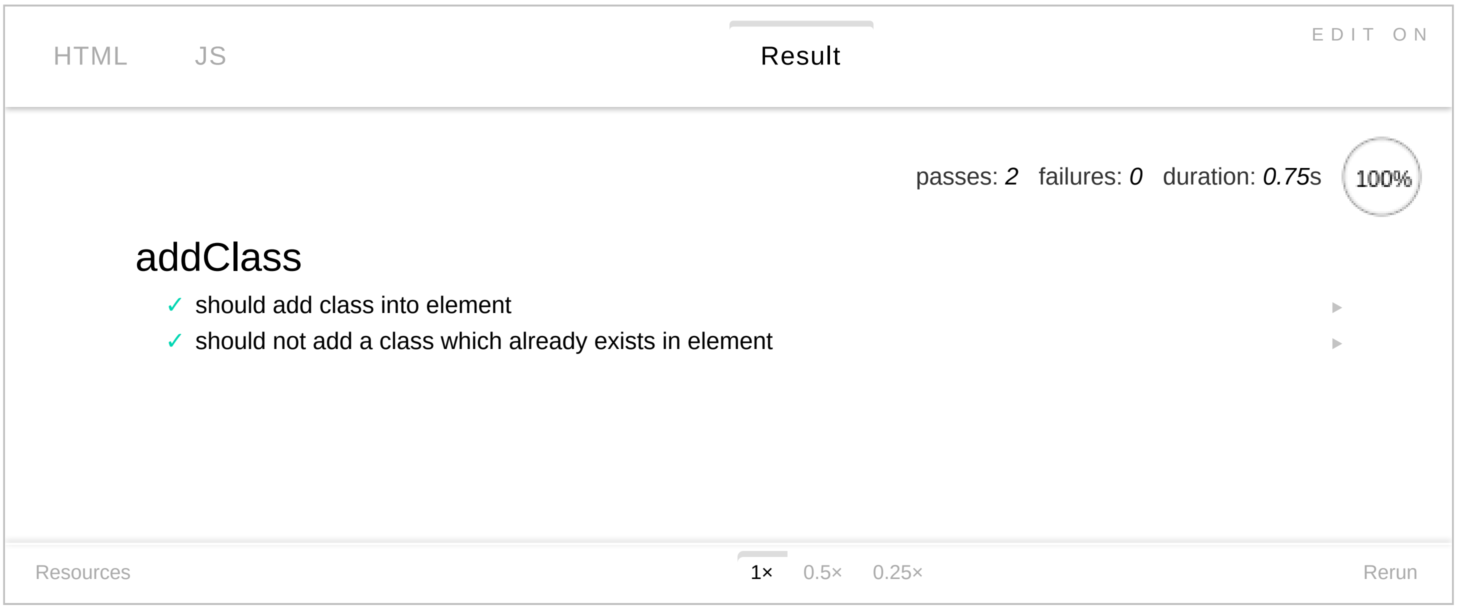
addClass
✓ should add class into element
◦ should not add a class which already exists in element

Next, let's implement the second test…

```
 it('should not add a class which already exists', function() {
  var element = { className: 'exists' };

  addClass(element, 'exists');

  var numClasses = element.className.split(' ').length;
  assert.equal(numClasses, 1);
});
```

It’s a good habit to run your tests often, so let’s check what happens if we run the tests now. As expected, they should pass.

Here’s another CodePen with the second test implemented.



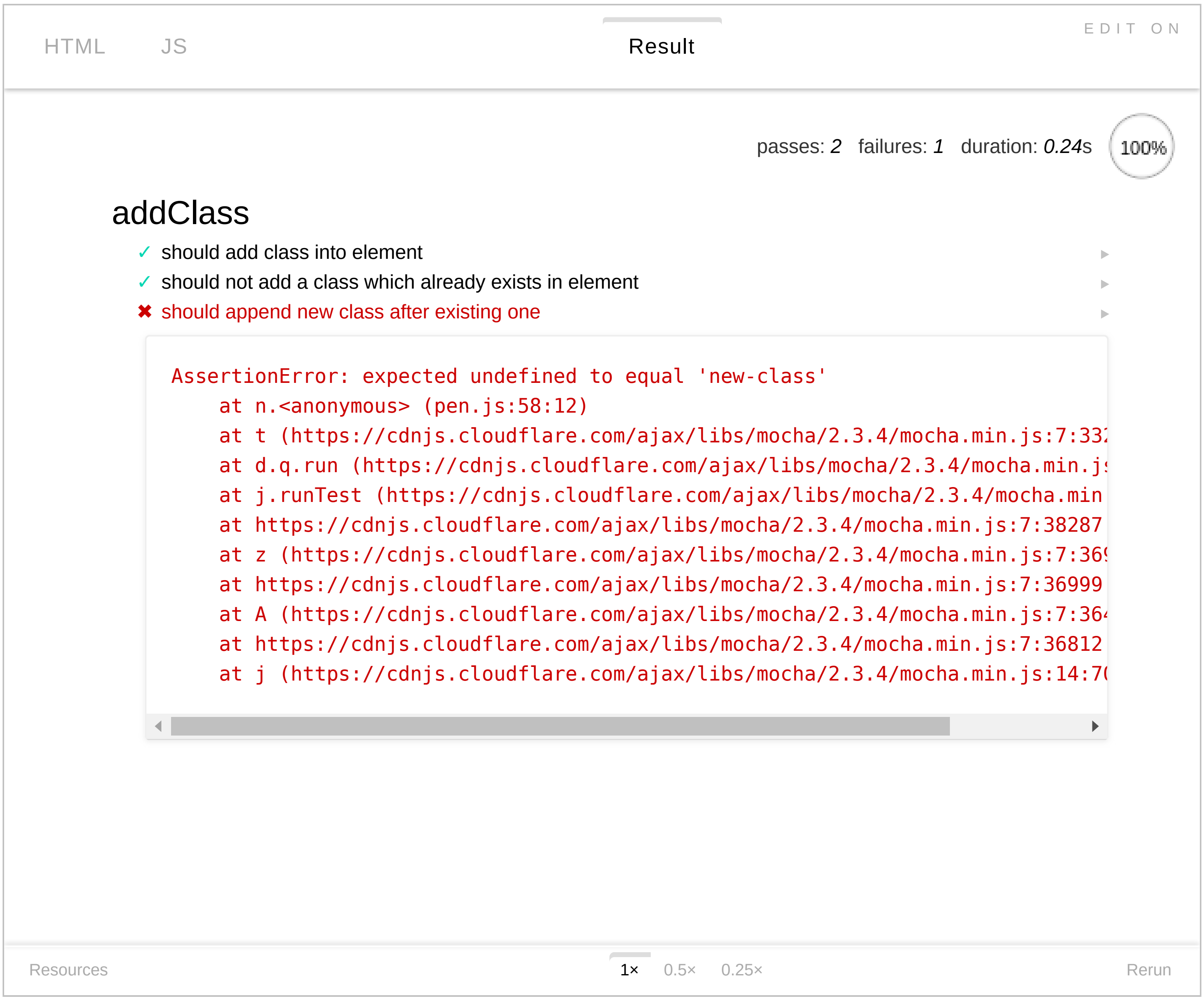
```
passes: 2   failures: 0   duration: 0.75s

addClass
  ✓ should add class into element
  ✓ should not add a class which already exists in element
```

But hang on! I actually tricked you a bit. There is a third behavior for this function which we haven’t considered. There is also a bug in the function — a fairly serious one. It’s only a three line function but did you notice it?

Let’s write one more test for the third behavior which exposes the bug as a bonus.

```
 it('should append new class after existing one', function() {
  var element = { className: 'exists' };

  addClass(element, 'new-class');

  var classes = element.className.split(' ');
  assert.equal(classes[1], 'new-class');
});
```

This time the test fails. You can see it in action in the following CodePen. The problem here is simple: CSS class names in elements should be separated by a space. However, our current implementation of `addClass` doesn’t add a space!



passes: *2* failures: *1* duration: *0.24*s

addClass

✓ should add class into element

✓ should not add a class which already exists in element

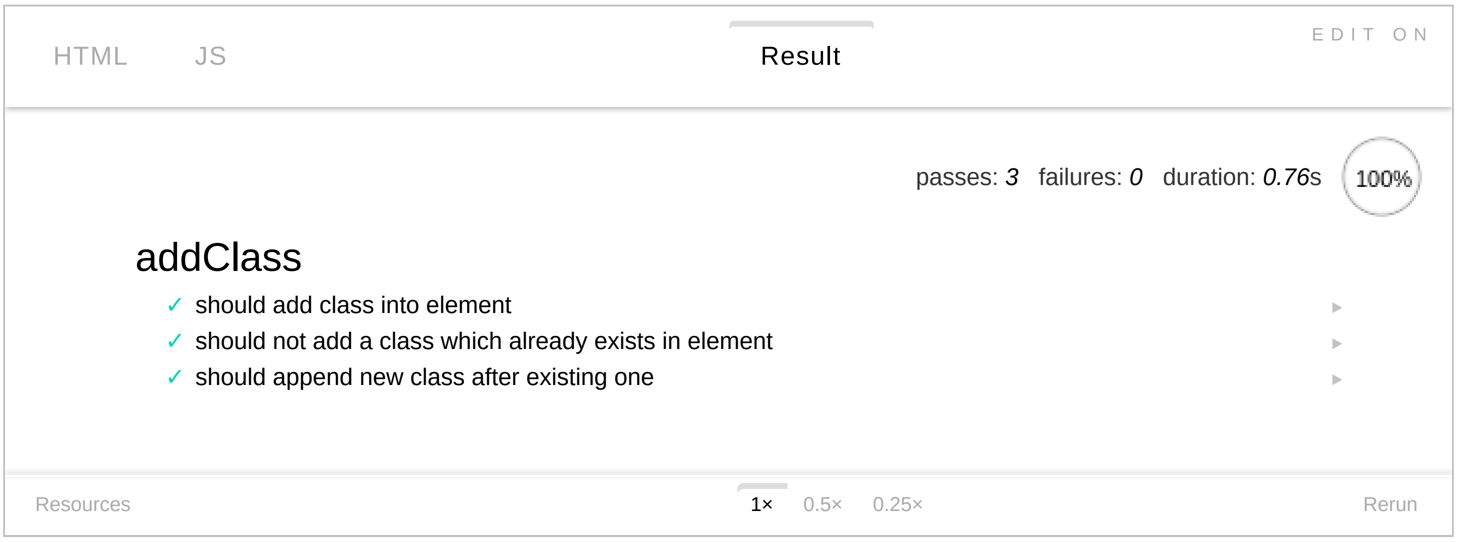
✖ should append new class after existing one

```
AssertionError: expected undefined to equal 'new-class'
    at n.<anonymous> (pen.js:58:12)
    at t (https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min.js:7:332
    at d.q.run (https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min.js
    at j.runTest (https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min
    at https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min.js:7:38287
    at z (https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min.js:7:369
    at https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min.js:7:36999
    at A (https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min.js:7:364
    at https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min.js:7:36812
    at j (https://cdnjs.cloudflare.com/ajax/libs/mocha/2.3.4/mocha.min.js:14:70
```

Let’s fix the function and make the test pass.

```
function addClass(el, newClass) {
  if(el.className.indexOf(newClass) !== -1) {
    return;
  }

  if(el.className !== '') {
    //ensure class names are separated by a space
    newClass = ' ' + newClass;
  }

  el.className += newClass;
}
```

And here’s a final CodePen with the fixed function and passing tests.



passes: 3 failures: 0 duration: 0.76s

addClass

✓ should add class into element
✓ should not add a class which already exists in element
✓ should append new class after existing one

Running the Tests on Node

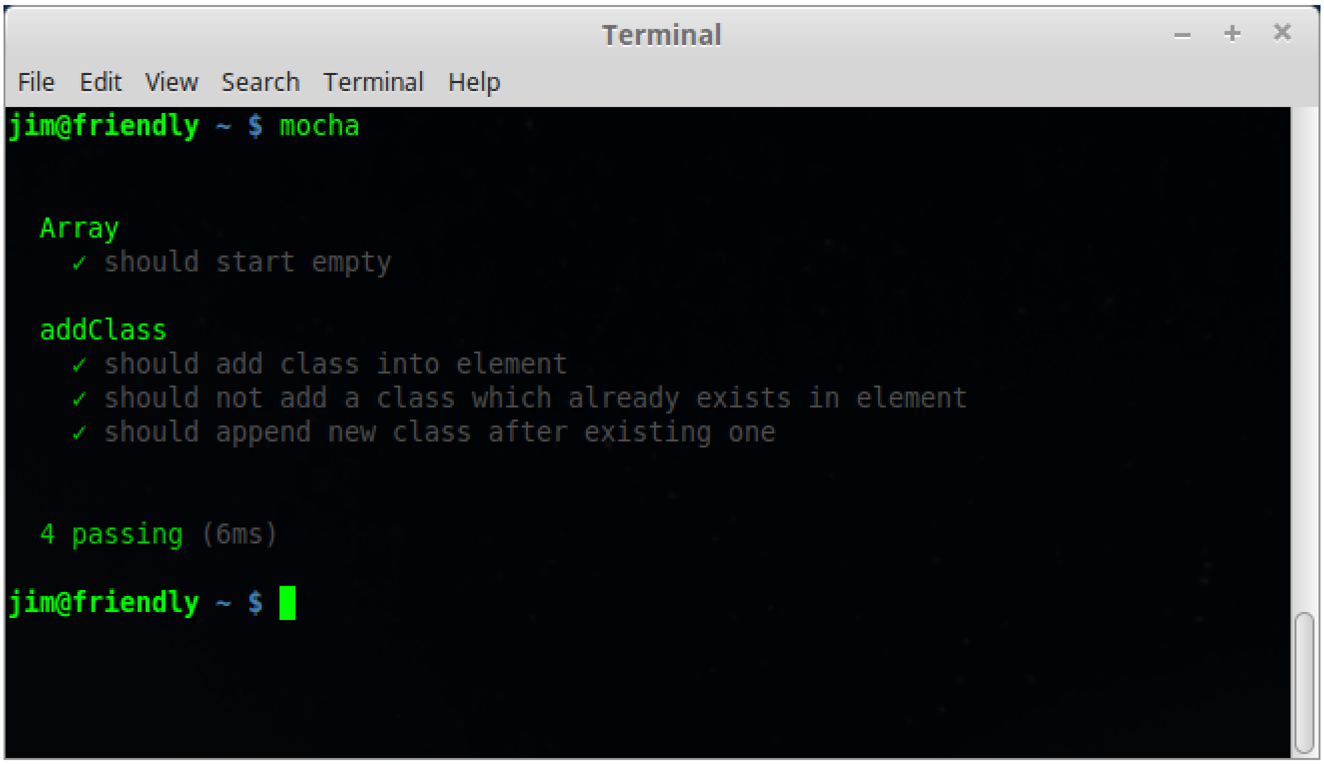
In Node, things are only visible to other things in the same file. As `className.js` and `classNameTest.js` are in different files, we need to find a way to expose one to the other. The standard way to do this is through the use of `module.exports`. If you need a refresher, you can read all about that here: Understanding module.exports and exports in Node.js (http://www.sitepoint.com/understanding-module-exports-exports-node-js/)

The code essentially stays the same, but is structured slightly differently:

```
 // className.js

module.exports = {
  addClass: function(el, newClass) {
    if(el.className.indexOf(newClass) !== -1) {
      return;
    }

    if(el.className !== '') {
      //ensure class names are separated by a space
      newClass = ' ' + newClass;
    }

    el.className += newClass;
  }
}
```

```
// classNameTest.js

var chai = require('chai');
var assert = chai.assert;

var className = require('../js/className.js');
var addClass = className.addClass;

// The rest of the file remains the same

describe('addClass', function() {
  ...
});
```

And as you can see, the tests pass.



```
jim@friendly ~ $ mocha


  Array
    ✓ should start empty

  addClass
    ✓ should add class into element
    ✓ should not add a class which already exists in element
    ✓ should append new class after existing one


  4 passing (6ms)

jim@friendly ~ $
```

What's Next?

As you can see, testing does not have to be complicated or difficult. Just as with other aspects of writing JavaScript apps, you have some basic patterns which repeat. Once you get familiar with those, you can keep using them again and again.

But this is just scratching the surface. There's a lot more to learn about unit testing.

Testing more complex systems
How to deal with Ajax, databases, and other "external" things?
Test-Driven Development

If you want to continue learning this and more, I've created a free JavaScript unit testing quickstart series (http://codeutopia.net/blog/h/subscribe). If you found this article useful, you should definitely check it out here (http://codeutopia.net/blog/h/subscribe).

Alternatively, if video is more your style, you might be interested in SitePoint Premium's course: Test-Driven Development in Node.js (https://www.sitepoint.com/premium/courses/test-driven-development-in-node-js-2932).

(http

Meet the author
Jani Hartikainen (https://www.sitepoint.com/author/jhartikainen/) (https://twitter.com/jhartikainen) G+ (https://plus.google.com/+JaniHartikainen)

Jani has built all kinds of JS apps for more than 15 years. At his blog (http://codeutopia.net/), he helps JavaScript developers learn to eliminate bad code so they can focus on writing awesome apps and solve real problems.

Javascript (Https://Www.Sitepoint.Com/Javascript/) > September 18, 2019 > By Michiel Mulders (https://www.sitepoint.com/author/mmulders/)

How to Build Your First Telegram Chatbot with Node.js

So, this morning you woke up with the idea to develop a way to store and label interesting articles you've read. After playing with the idea, you figure a Telegram chatbot is the most convenient solution for this problem.

In this guide, we'll walk you through everything you need to know to build your first Telegram chatbot using JavaScript and Node.js.

To get started, we have to register our new bot with the so-called Botfather to receive our API access token.

Bot Registration with @BotFather

The first step towards our very own Telegram bot is registering the bot with the BotFather. The BotFather is a bot itself that makes your life much easier. It helps you with registering bots, changing the bot description, adding commands, and providing you with the API token for your bot.

The API token is the most important step, as this allows you to run the code that can perform tasks for the bot.

1. Finding the BotFather