How to test JavaScript with Mocha — Part 2

Brandon Morelli Follow
Jun 9, 2017 · 5 min read

Yesterday we looked at the basics of Mocha. Today, we'll be integrating Mocha into a project so you can see how it **really** works.

Testing via unsplash.com

Preface

> *Hey! I'm Brandon. I created codeburst.io and I write JavaScript tutorials and articles to help beginners better understand the inner workings of Web Development. If you have any questions about the article, leave a comment and I'll get back to you, or find me on twitter @brandonmorelli. Lastly, when you're ready to really dive into Web Development, Check out the Best Courses for Learning Full Stack Web Development*

What exactly is this tutorial?

This tutorial will give you a small-scale, real-world example of how to use Mocha for testing. By the end of this tutorial you'll have successfully used Mocha to test an existing JS file. Before taking on this tutorial, you should understand what Mocha is, how to group tests, and how to use an assertion library. Please refer back to **my first Mocha tutorial** if you need a refresher.

. . .

Project Setup

You should already have Mocha installed from the previous tutorial. If not, install Mocha globally by running: `$ npm install -g mocha`

Next, we'll create a project directory named `temperature`. In the `temperature` directory we'll create a file named `app.js` and a folder name `test`. Within the `test` folder, create a file named `test.js`. Finally, we'll initialize our project by running `npm init`. During your project initialization, the question that is most important here is '**test command:**'—respond with **'mocha'.** This way we can run mocha by simply typing `npm test`.

When finished, you should have a file structure that looks like this:

```
temperature
|-- app.js
|-- package.json
|-- test
   |-- test.js
```

Your `package.json` file should also contain the following json:

```
"scripts": {
  "test": "mocha"
},
```

Once you have all of the above, we're ready to start!

What are we testing?

We're going to be testing our `app.js` file in this tutorial. To speed the process along, I've already created the contents of our `app.js` for you. Go ahead and copy the code below into your `app.js`

```
cToF = function(celsius) {
  if(!Number.isInteger(celsius)) return undefined;
  return celsius * 9 / 5 + 32;
}

fToC = function(fahrenheit) {
  if(!Number.isInteger(fahrenheit)) return undefined;
```

Our `app.js` is only nine lines of code and consists of two functions:

- `cToF()` —is a function for converting Celsius temperatures to Fahrenheit. It takes one parameter, the temperature in Celsius, and returns the Fahrenheit equivalent.
- `fToC()` —is the reverse function. It converts Fahrenheit to Celsius. Its one parameter is the temperature in Fahrenheit and it returns the Celsius equivalent.
- Both functions initially check to ensure the argument they've been passed is an integer. If it is not an integer (a string, blank, `undefined` , `NaN` , etc.) then `undefined` is returned and function short-circuits.

Setting up our Tests

We're going to test to ensure these functions are working nicely. Here's the structure I see our `test.js` file taking:

1. A testing group named `Temperature Conversion`
2. Within that testing group, two additional testing groups, one named `cToF` and one named `fToC`
3. Each testing group will have three test cases: two numbers, and a non-integer test.

Test Groups

First, we'll set up our two test groups using `describe()`

```
var assert = require('assert');
describe('Temperature Conversion', function() {
  describe('cToF', function() {
    // tests here
  });
  describe('fToC', function() {
    // tests here
  });
});
```

Above, we have our two nested, function specific `describe` blocks within our outer `describe` block for `Temperature Conversion`. Now that we have the basic structure, we can add in our three `it()` test cases to each of our testing groups.

Adding Test Cases

Our three test cases will use the built in assert assertion library. We will be testing equality, so we'll use `assert.equal(actual, expected);`

For our first test, we'll test a number that should remain unchanged. If you didn't know, both Fahrenheit and Celsius are equal at 40 degrees below zero. Lets setup this test to ensure our function handles the math correctly:

```
it('should convert -40 celsius to -40 fahrenheit',
function() {
   assert.equal(-40, cToF(-40));
});
```

Perfect! We're asserting that once we run the number `-40` through our `cToF` function, the resulting value should also equal `-40`.

For our next test, we'll look at the freezing point of water. We want to ensure that 0 degrees Celsius equals 32 degrees Fahrenheit.

```
it('should convert 0 celsius to 32 fahrenheit', function() {
  assert.equal(32, cToF(0));
});
```

Everything is the same as test number one, except we're passing the number `0` into our function and expecting a result to equal `32`.

For our last test, we're going to test a blank string, and make sure the function returns undefined:

```
it('should return undefined if no temperature is input',
function(){
  assert.equal(undefined, cToF(''));
});
```

For this test, we expect the result of `cToF('')` to equal `undefined`.

Awesome, all of our `cToF` tests are complete! Now we need to do the same for the `fToC` function. I wont walk you through writing these tests—you should have that part down by now. Here's what they should look like:

```
it('should convert -40 fahrenheit to -40 celsius',
function() {
  assert.equal(-40, fToC(-40));
});
it('should convert 32 fahrenheit to 0 celsius', function() {
  assert.equal(0, fToC(32));
});
it('should return undefined if no temperature is input',
function(){
  assert.equal(undefined, fToC(''));
});
```

We're done… Right?

Awesome, our tests are setup, now just need to run them with `npm test`

```
0 passing (20ms)
```

Uh oh. So what happened?

Exposing our functions

We never exposed our functions to Mocha. What this means is our `test.js` file has no way to interact with the functions in our `app.js` file. Luckily, there are a number of easy ways to do this.

1. We're going to create an empty object named convert `let convert = {};`

2. Instead of two different functions, we're going to make each function a method on our new `convert` object.

3. At the end of `app.js` we're going to expose our `convert` object using `module.exports`. If you've never used it before, `module.exports` is how we tell JavaScript what object to return as the result of a `require` call. Lets look at the code to see it in action:

```
let convert = {};

convert.cToF = function(celsius) {
  if(!Number.isInteger(celsius)) return undefined;
  return celsius * 9 / 5 + 32;
}

convert.fToC = function(fahrenheit) {
  if(!Number.isInteger(fahrenheit)) return undefined;
```

Now all that's left to do is require our `app.js` file in `test.js` and change the function names used in our tests to include the `convert` object. Here's the final `test.js` code:

```
let convert = require('../app.js')
let assert = require('assert');

describe('Temperature Conversion', function() {
  describe('cToF', function() {
    it('should convert -40 celsius to -40 fahrenheit', func
      assert.equal(-40, convert.cToF(-40));
    });
    it('should convert 0 celsius to 32 fahrenheit', functio
      assert.equal(32, convert.cToF(0));
    });
    it('should return undefined if no temperature is input'
      assert.equal(undefined, convert.cToF(''));
    });
  });
  describe('fToC', function() {
    it('should convert -40 fahrenheit to -40 celsius', func
      assert.equal(-40, convert.fToC(-40));
```

Run our tests

This time when we run `npm test`, everything works as expected!

```
Temperature Conversion
  cToF
    √ should convert -40 celsius to -40 fahrenheit
    √ should convert 0 celsius to 32 fahrenheit
    √ should return undefined if no temperature is input
  fToC
    √ should convert -40 fahrenheit to -40 celsius
    √ should convert 32 fahrenheit to 0 celsius
    √ should return undefined if no temperature is input

6 passing (34ms)
```

You did it.

Good work! You've now test Mocha in a separate file. There's still so much more left to learn—check out the documentation, you should have all the tools you need to understand them now.

I publish a few articles/tutorials each week, please enter your email here if you'd like to be added to my once-weekly email list.

❤ If this post was helpful, please hit the little blue heart! And don't forget to check out my other recent articles:

- What the heck is an Immediately-Invoked Function Expression?
- Three awesome courses for learning Node.js
- And when you're ready to really dive into Web Development, Check out the 5 Best Courses for Learning Full Stack Web Development