Fundamentals of Testing in JavaScript

Transcripts for Kent C. Dodds course on egghead.io.

There are 6 transcripts, one transcript for each video.

Description

Do you know what a testing framework does? Do you know how it differs from a testing or assertion library? The best way to use a tool effectively is to understand how it works. And the best way to understand how a tool works is by making it yourself! In this short course, we'll learn how testing frameworks and assertion libraries by building our own simple version.

Throw an Error with a Simple Test in JavaScript

We have a bug in the `sum` function. It's doing subtraction instead of addition. We could easily fix this, but let's go ahead and write an automated test that can make sure that this bug never surfaces again.

An automated test in JavaScript is code that throws an error when things are unexpected. Let's do that. Let's get our `result` from the `sum` of three and seven.

simple.js

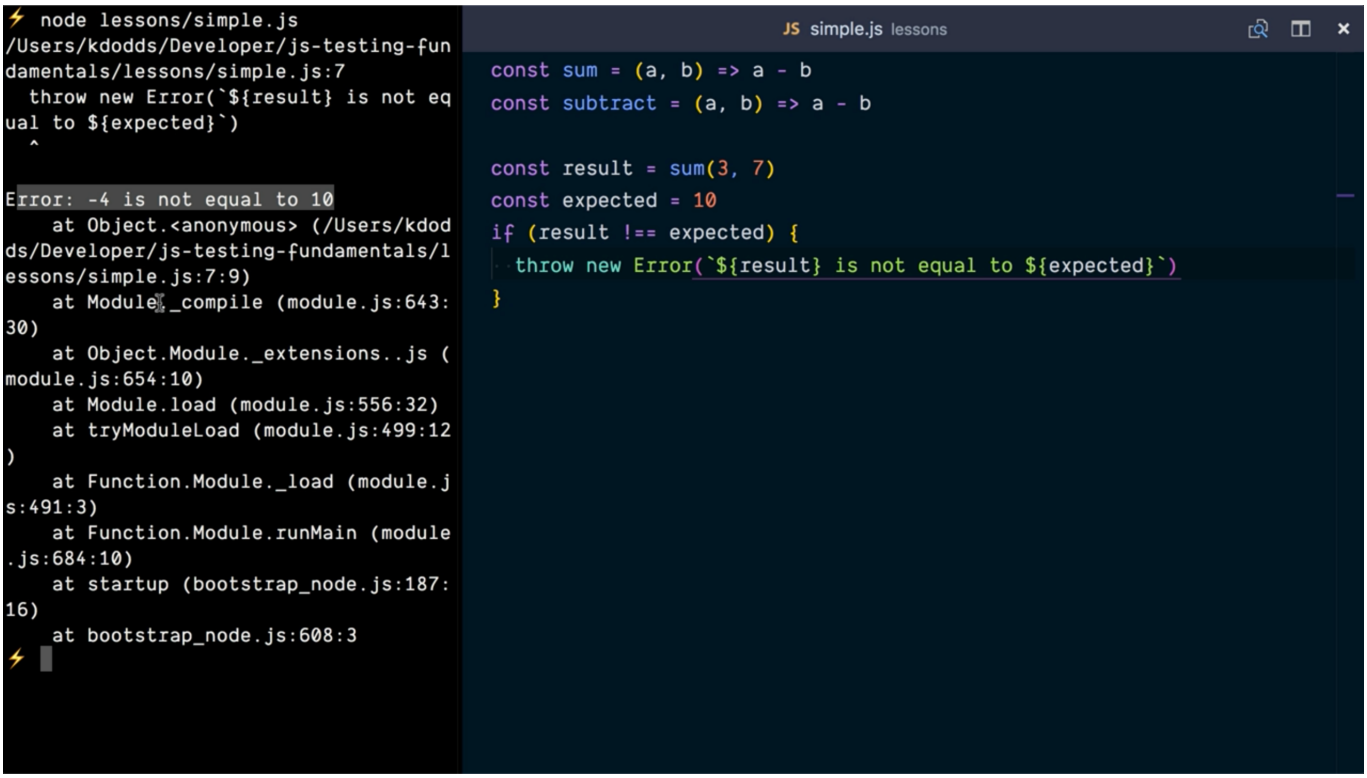
```
const result = sum(3, 7)
```

We will say our `expected` is 10. We can say if the `result` is not equal to the `expected` value, then we can throw a new error that says, "Result is not equal to expected."

```
const expected = 10
if (result !== expected){
    throw new Error(`${result} is not equal to ${expected}`)
}
```

To run this, we can run `node lessons/simple.js`.

We will get our error, "-4 is not equal to 10."



If we replace this with addition and run that again, our script passes without throwing errors. This is the most fundamental form of a test.

```
const sum = (a, b) => a + b
```

Let's go ahead and add another test for `subtract`. I am going to change this from `const` to `let`.

```
const subtract = (a, b) => a - b
```

We will say, `result` is now equal to subtract of seven and three, and our `expected` is now equal to four."

```
result = subtract(7, 3)
expected = 4
```

We will just copy and paste this here.

```
if (result !== expected){
    throw new Error(`${result} is not equal to ${expected}`)
}
```

Save that and run our script again. It passes without error. In review, this is the most fundamental form of a test in JavaScript. It's simply a code that will throw an error when the result is not what we expect.

Let's go ahead and break this again. We'll run that script again. We're getting an error message. The Javascript testing framework is to make that error message as useful as possible so we can quickly identify what the problem is and fix it.

Abstract Test Assertions into a JavaScript Assertion Library

We're pulling `sum` and `subtract` from this math module, and `sum` has a bug.

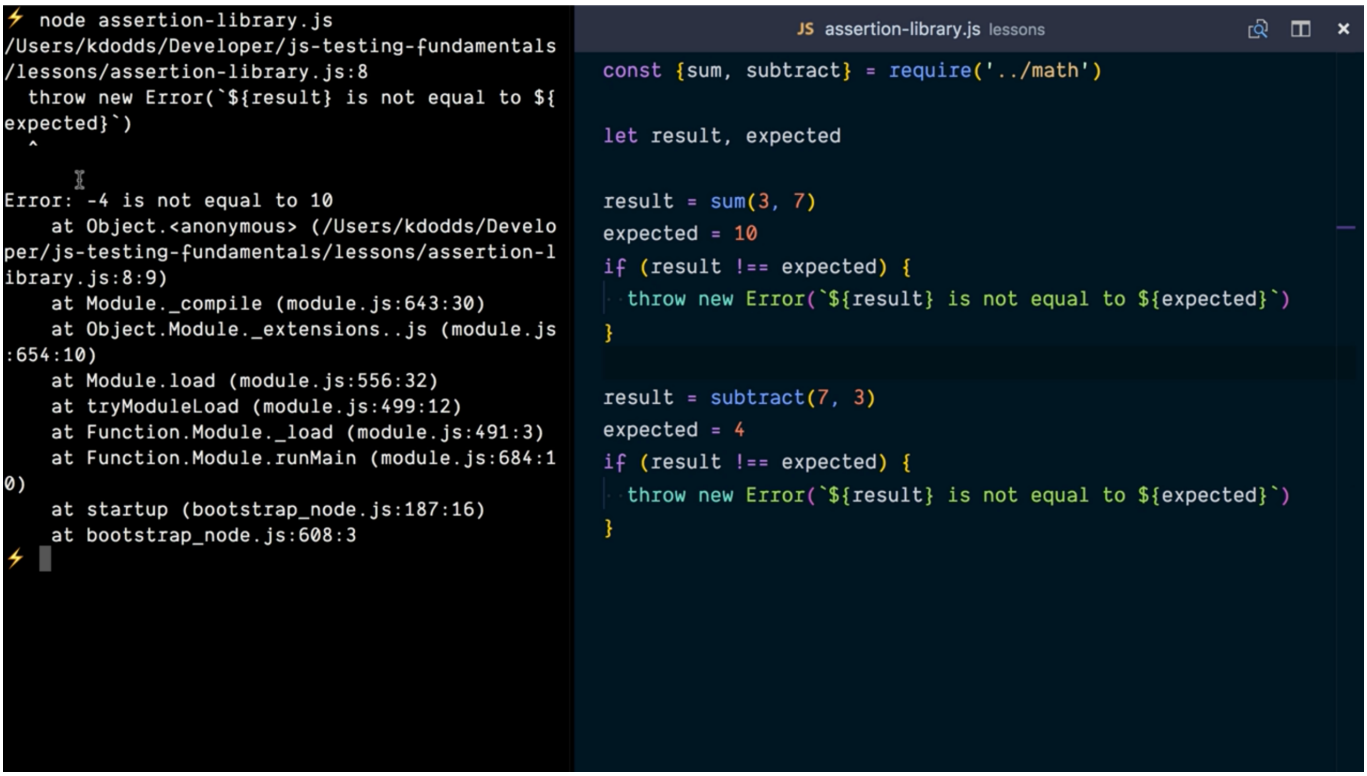
assertion-library.js

```
const {sum, subtract} = require('../math')
```

We can reveal that bug with a simple test.

```
result = sum(3, 7)
expected = 10
if (result !== expected) {
  throw new Error(`${result} is not equal to ${expected}`)
}
```

When we run this file with node, that assertion is throwing an error.



Also, this code is pretty imperative. It'd be nice to write a little abstraction to make it read a little nicer. Let's go ahead and write a simple abstraction to encapsulate this assertion. I'll create a function down here called `expect`. That's going to accept an `actual`.

```
function expect(actual){

}
```

We are going to return an object that has some assertions on it. Our first one here is going to be `toBe`, and that's going to take an `expected` value.

```
function expect(actual) {
  return {
    toBe(expected) {
    }
  }
}
```

We're going to say if the `actual` is not equal to the `expected` value, then we'll `throw a new Error` that says, "Actual is not equal to expected."

```
function expect(actual) {
  return {
    toBe(expected) {
      if (actual !== expected) {
        throw new Error(`${actual} is not equal to ${expected}`)
      }
    }
  }
}
```

We can take this duplicated code, remove that and replace it with:

```
result = sum(3, 7)
expected = 10
expect(result).toBe(expected)
```

If we run that now, we are going to get the same error message. This function is like an assertion library. Fundamentally, it takes an actual value, and then it returns an object that has functions for different assertions that we can make on that actual value.

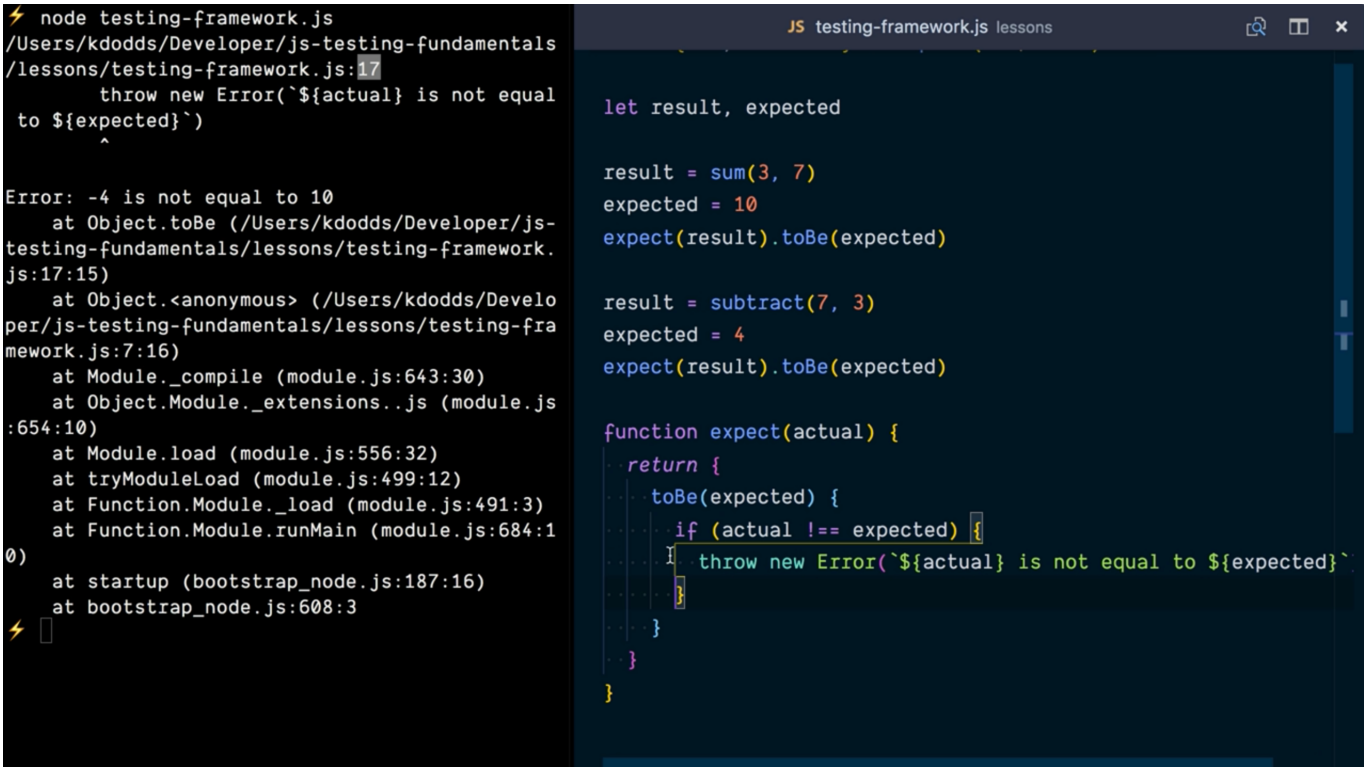
Here, we have `toBe`. We could also have a `twoEqual`, now it'd take an `expected` value, and maybe do a deep equality check.

We could also have a `toBeGreaterThan` or `toBeLessThan`, and then it could take an `expected` value. There're all kinds of assertions that we could add to our little assertion library here to make writing our test a little easier.

Encapsulate and Isolate Tests by building a JavaScript Testing Framework

One of the limitations of the way that this test is written is that as soon as one of these assertions experiences an error, the other tests are not run. It can really help developers identify what the problem is if they can see the results of all of the tests.

In addition to that, because we are throwing our error here, if we look at the stack trace after running our testing file, we see that the error was thrown on line 17 really directly. We know exactly where that's happening.



We have to dig through the stack trace a little bit further to see which one of these is throwing the error. It's not readily apparent whether this `-4 is not equal to 10` is happening because the `sum` is broken or because the `subtract` is broken.

A testing framework's job is to help developers identify what's broken as quickly as possible. It can do that by making more helpful error messages and by running all of the tests. Let's go ahead and make that.

I am going to start with a function called the `test`. It's going to accept a `title` and a `callback`.

testing-framework.js

```
function test(title, callback) {
```

```
}
```

Because this `test` could throw an error, I am going to wrap that in a try-catch. I'll call the `callback`. If that `callback` throws an error, then I'll `log` that error.

```
function test(title, callback) {
  try {
    callback()
  } catch (error) {
    console.error(error)
  }
}
```

I'll also want to `console.error` the `title`. If it doesn't throw an error, then we'll get to this line where I can `console.log` the `title`. Let's add a little `✓` and an `×` to make it more apparent what happened.

```
function test(title, callback) {
  try {
    callback()
    console.log(`✓ ${title}`)
  } catch (error) {
    console.error(`× ${title}`)
    console.error(error)
  }
}
```

Next, let's make a function called `sumTest`. We'll move this code into `sumTest`.

```
function sumTest(){
    result = sum(3, 7)
    expected = 10
    expect(result).toBe(expected)
}
```

We'll use our `test` utility, and we'll title this `sum adds numbers` and pass our `sumTest`.

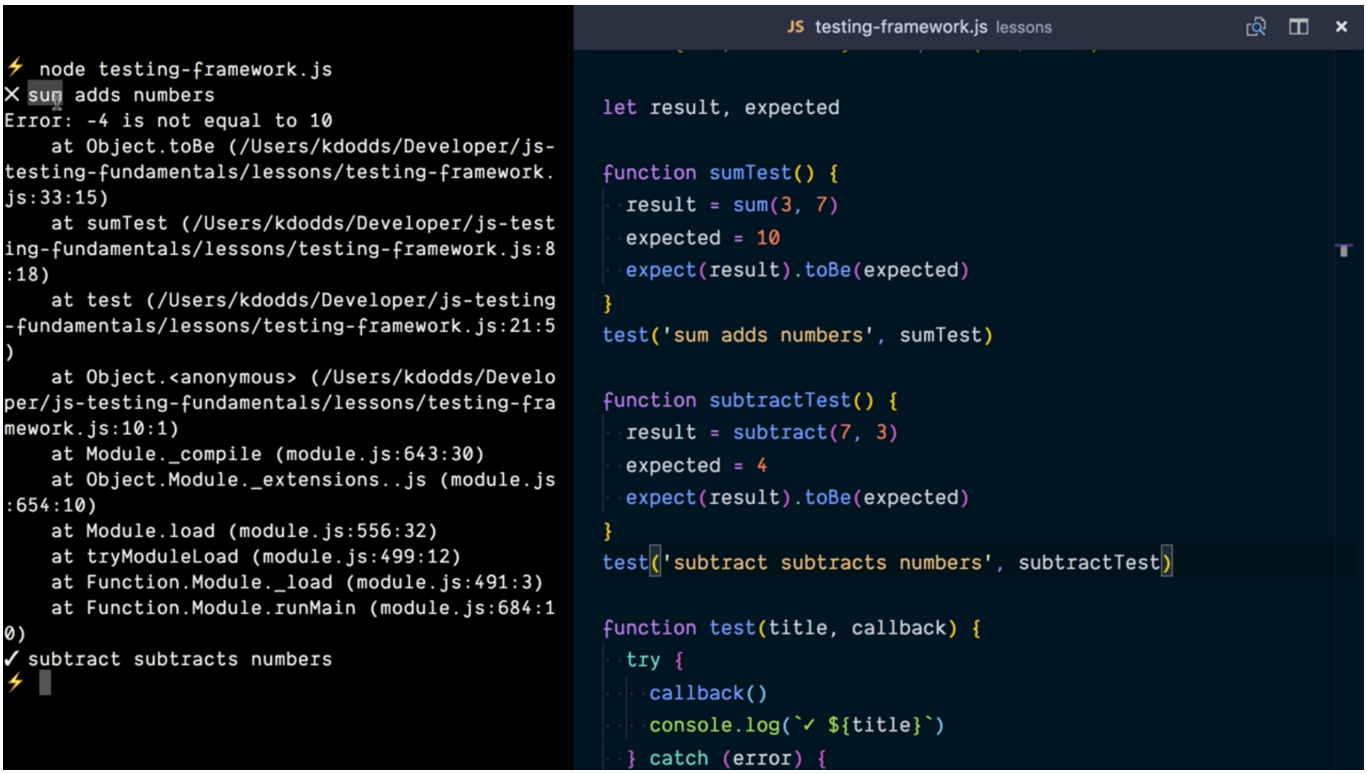
```
test('sum adds numbers', sumTest)
```

We'll do the same thing for our `subtractTest` and move that code up into our subtractTest, then we'll add a `test(subtract subtracts numbers)`.

We'll pass our `subtractTest`.

```
function subtractTest(){
    result = subtract(7,3)
    expected = 4
    expect(result).toBe(expected)
}
test('subtract subtracts numbers', subtractTest)
```

Now, we'll run our test file again. Here, we'll see we had an error with `sum adds numbers` and the passing test with `subtract subtracts numbers`. We know that the problem isn't with `subtract`. It's with `sum`. It's readily apparent.



Let's refractor things a little bit to encapsulate our `test` better. I'll make this `const` and then we can get rid of this let declaration. I am actually going to turn this into an arrow function. We'll move this code into that arrow function. I'll do the same thing for `subtractTest`.

```
test('sum adds numbers', () => {
  const result = sum(3, 7)
  const expected = 10
  expect(result).toBe(expected)
})

test('subtract subtracts numbers', () => {
  const result = subtract(7, 3)
  const expected = 4
  expect(result).toBe(expected)
})
```

In review, our test utility's job is to make it easier for people to quickly identify what's broken so they can fix it quickly. We do that by having a more helpful error message and by running all of the tests in our file.

Support Async Tests with JavaScripts Promises through async await

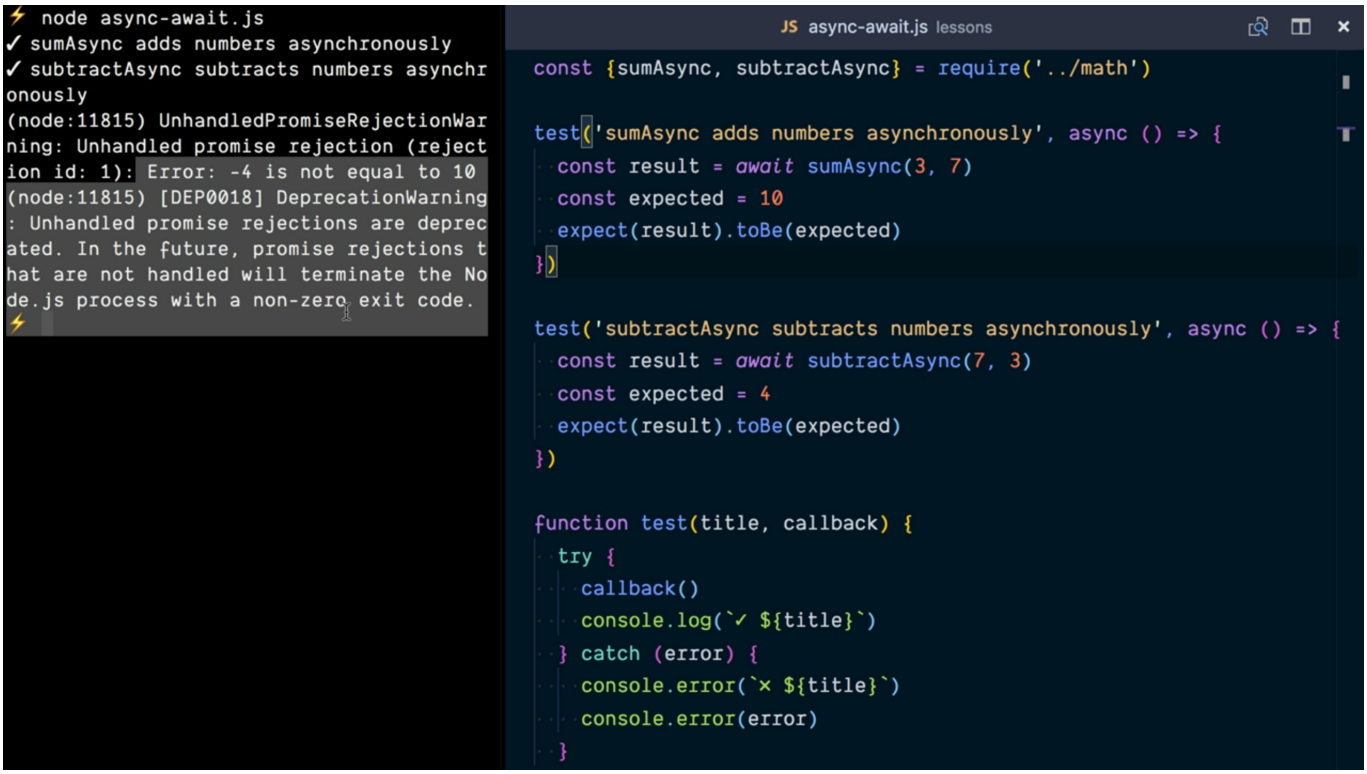
Our testing framework works great for our synchronous test. What if we had some asynchronous functions that we wanted to test? We could make our callback functions `async`, and then use the `await` keyword to wait for that to resolve, then we can make our assertion on the `result` and the `expected`.

async-await.js

```
test('sumAsync adds numbers asynchronously', async () => {
  const result = await sumAsync(3, 7)
  const expected = 10
  expect(result).toBe(expected)
})

test('subtractAsync subtracts numbers asynchronously', async () => {
  const result = await subtractAsync(7, 3)
  const expected = 4
  expect(result).toBe(expected)
})
```

This approach has a little bit of a problem though. If we run our test, we are going to see that they both pass, and then after that, we have an `UnhandledPromiseRejectionWarning`. That is the actual error coming from our `sumAsync` function being broken.

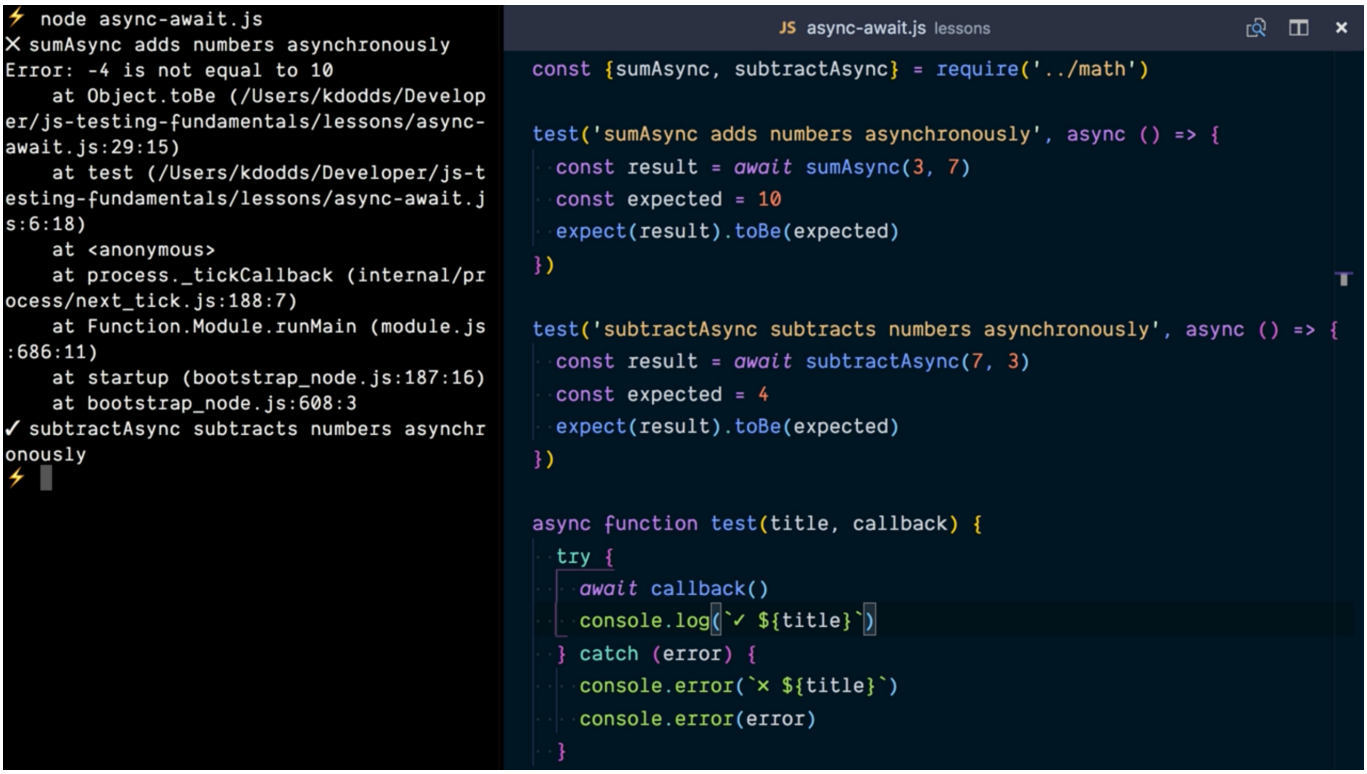


Because this is an async function, this function will return a promise. When this error is thrown, it's going to reject that promise. Here inside of our `test` function, this `callback` is going to return a promise. If we turn this `test` into an `async` function, and then `await` that `callback`, if that promise is rejected, then we'll land in our catch block.

If no error is thrown, then we'll continue on inside the try block. This will work for both our synchronous and our asynchronous tests.

```
async function test(title, callback) {
  try {
    await callback()
    console.log(`✓ ${title}`)
  } catch (error) {
    console.error(`× ${title}`)
    console.error(error)
  }
}
```

With that, we can run our test again, and things happen exactly as we expect.



Provide Testing Helper Functions as Globals in JavaScript

These testing utilities are pretty useful. We want to be able to use them throughout our application in every single one of our test files.

We could put these into a module that we would require an import into every single one of our test files, but many testing frameworks embrace the fact that you're going to be using these in every single one of your test files, and so they just make them available globally.

I am going to cut this out of our testing file. I am going to go to `setup-global.js` file, and I will paste it into here, and then I will say `global.test = test`, and `global.expect = expect`.

setup-globals.js

```
async function test(title, callback) {
  try {
    await callback()
    console.log(`✓ ${title}`)
  } catch (error) {
    console.error(`× ${title}`)
    console.error(error)
  }
}

function expect(actual) {
  return {
    toBe(expected) {
      if (actual !== expected) {
        throw new Error(`${actual} is not equal to ${expected}`)
      }
```

```
    }
  }
}

global.test = test
global.expect = expect
```

I can run

```
node --require ./setup-globals.js lessons/globals.js
```

and then our test file.

We get the same result as we did before. Now, we can use this `setup-globals` in every single one of our test files. With that setup, all of our test files can use the `test` and `expect` global variables.

Verify Custom JavaScript Tests with Jest

The testing framework we've written actually looks remarkably a lot like Jest. Rather than running `node --require`, and then instead of globals and so on and so forth, we could actually just use Jest directly.

If I run `npx jest`,

Terminal

```
npx jest
```

jest will automatically pick up our `jest.test.js` file based off of that convention.

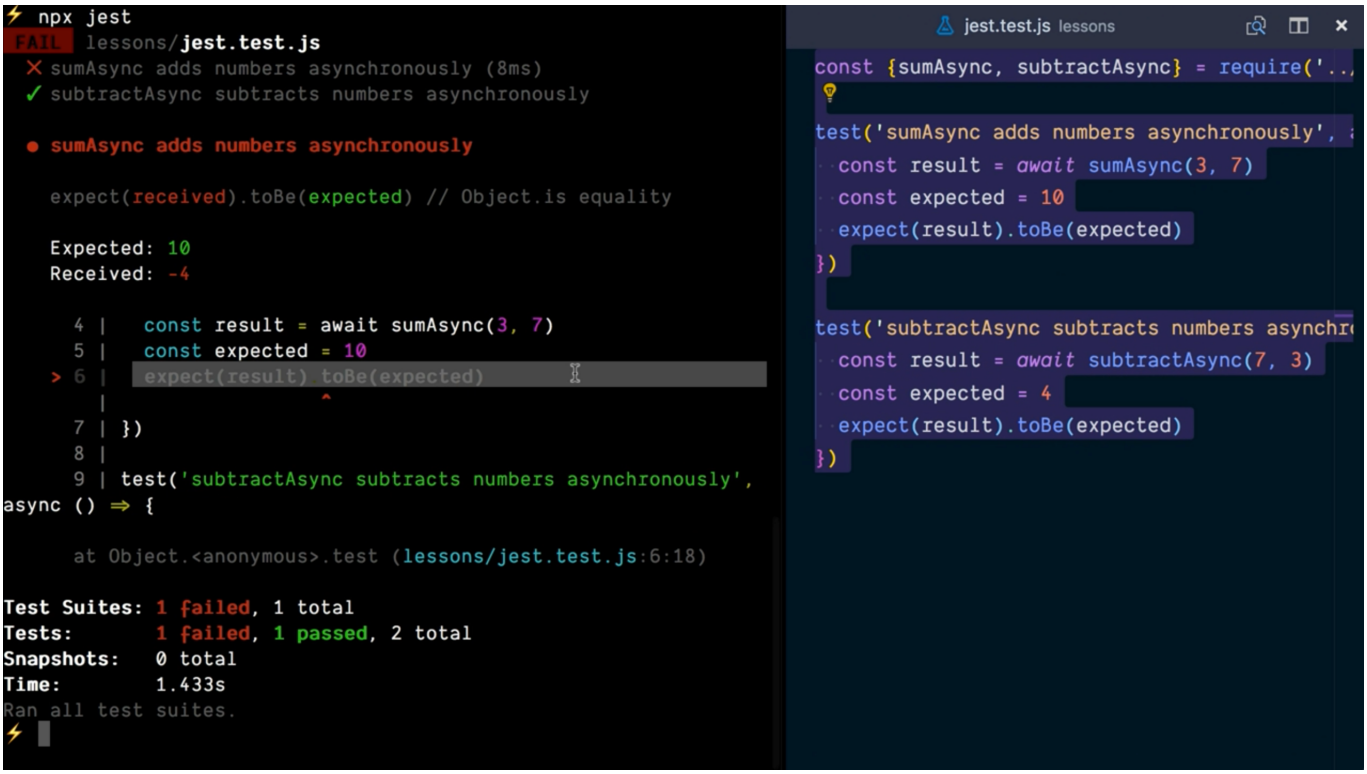
jest.test.js

```
const {sumAsync, subtractAsync} = require('../math')

test('sumAsync adds numbers asynchronously', async () => {
  const result = await sumAsync(3, 7)
  const expected = 10
  expect(result).toBe(expected)
})

test('subtractAsync subtracts numbers asynchronously', async () => {
  const result = await subtractAsync(7, 3)
  const expected = 4
  expect(result).toBe(expected)
})
```

It will show us really helpful error messages, and even a code frame to show us exactly where in our code that error was thrown.



These are some of the things that make jest such an awesome testing framework because the error messages are so clear.