JavaScript Mocking Fundamentals

Transcripts for Kent C. Dodds (https://egghead.io/instructors/kentcdodds) course on egghead.io (https://egghead.io/courses/javascript-mocking-fundamentals).

Description

When running unit tests, you don't want to actually make network requests or charge real credit cards. That could… get expensive… and also very, very slow. So instead of running your code exactly as it would run in production, you can modify how some of your JavaScript modules and functions work during tests to avoid test unreliability (flakiness) and improve the speed of your tests. This kind of modification can come in the form of stubs, mocks, or generally: "test doubles."

There are some great libraries and abstractions for mocking your JavaScript modules during tests. The Jest testing framework (https://facebook.github.io/jest) has great mocking capabilities built-in for functions

(https://facebook.github.io/jest/docs/en/mock-functions.html) as well as entire modules (https://facebook.github.io/jest/docs/en/manual-mocks.html). To really understand how things are working though, let's implement some of these features ourselves.

Override Object Properties to Mock with Monkey-patching in JavaScript

Here, we have a `thumbWar` module, and its purpose is to take a `player1` and a `player2`,
and run a couple of games of thumb war to get a `winner`, and then `return` the `winner`.

thumb-war.js

```
const utils = require('./utils')

function thumbWar(player1, player2) {
  const numberToWin = 2
  let player1Wins = 0
  let player2Wins = 0
  while (player1Wins < numberToWin &&
player2Wins < numberToWin) {
    const winner = utils.getWinner(player1,
player2)
    if (winner === player1) {
      player1Wins++
    } else if (winner === player2) {
      player2Wins++
    }
  }
  return player1Wins > player2Wins ? player1 :
player2
}

module.exports = thumbWar
```

The way it does this is use this `utils.getWinner`, which is this super advanced algorithm that we've developed to determine who the `winner` is.

Yes, this is a little bit contrived, but our goal is to mock out the `getWinner` function,
so we don't have to run it in our test. So let's see how we could do that. First, let's go ahead and get our `winner`. That's going to be `thumbWar('Kent C. Dodds', 'Ken Wheeler')`.
Then we'll `assert.strictEqual(winner, 'Kent C. Dodds')`, of course!.

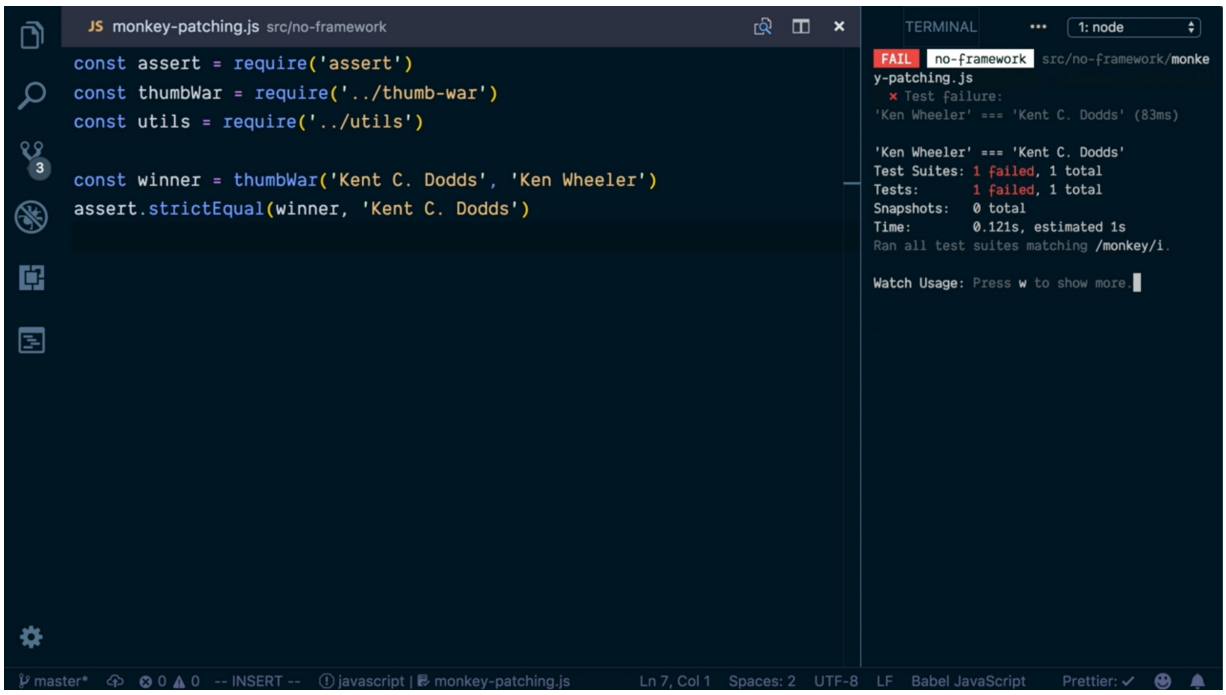
monkey-patching.js

```
const assert  = require('assert')
const thumbWar = require('../thumb-war')
const utils = require('../utils')

const winner = thumbWar('Kent C. Dodds', 'Ken
Wheeler')
assert.strictEqual(winner, 'Kent C. Dodds')
```

If I save this and run, we're going to get a test failure. If I rerun it, then it's going to pass. It's completely random because that `utils` method is a random function. Remember, it's the thing that we want to mock out in this test.

Random Passing Test



We have the `utils` module right here, and we can go ahead and mock out `getWinner` by simply assigning it to a new function that takes a `player1` and a `player2`, and is always going to return

`player1`. If we save that, then every single time we run our test, it's going to pass.

monkey-patching.js

```
utils.getWinner = (p1, p2) => p1

const winner = thumbWar('Kent C. Dodds', 'Ken
Wheeler')
assert.strictEqual(winner, 'Kent C. Dodds')
```

Doing this, we've made our test deterministic, and we can ensure that the thumb war is going to operate normally, considering our mock for `getWinner`. An essential part of mocking is that you clean up after yourself so that you don't impact other tests that may not want to mock the thing that you want, or may want to mock it in a different way.

At the bottom of our test, we need to reassign it to the original value of `getWinner`.
We'll assign it to `originalGetWinner`, and then we'll declare that up here as a variable with `utils.getWinner`. What we're doing here is called monkey patching.

monkey-patching.js

```
const originalGetWinner = utils.getWinner
utils.getWinner = (p1, p2) => p1

const winner = thumbWar('Kent C. Dodds', 'Ken
Wheeler')
assert.strictEqual(winner, 'Kent C. Dodds')

utils.getWinner = originalGetWinner
```

We're taking the `utils` module that the `thumbWar` module is using, and we're overriding the `getWinner` property so that we can make this call deterministic for our test.

thumb-war.js

```
function thumbWar(player1, player2) {
  const numberToWin = 2
  let player1Wins = 0
  let player2Wins = 0
  while (player1Wins < numberToWin &&
player2Wins < numberToWin) {
    const winner = utils.getWinner(player1,
player2) // The getWinner property
    if (winner === player1) {
      player1Wins++
    } else if (winner === player2) {
      player2Wins ++
    }
  }
  return player1Wins > player2Wins ? player1 :
player2
}
```

Then we're cleaning up after ourselves to make sure that other tests that may want to use this module can use it in its unmodified state.

monkey-patching.js

```
const originalGetWinner = utils.getWinner

...

utils.getWinner = originalGetWinner
```

Ensure Functions are Called Correctly with JavaScript Mocks

It would be nice if we could make some more assertions about how `getWinner` is being called to ensure that it's always being called with a `player1` and a `player2`, because we could break the implementation, but our test couldn't catch that.

For example, if we didn't call this with a `player2`, our test continued to pass, but the implementation is definitely wrong.

thumb-war.js

```
function thumbWar(player1, player2) {
  const numberToWin = 2
  let player1Wins = 0
  let player2Wins = 0
  while (player1Wins < numberToWin &&
player2Wins < numberToWin) {
    const winner = utils.getWinner(player1) //
No player2 here
    if (winner === player1) {
      player1Wins++
    } else if (winner === player2) {
      player2Wins ++
    }
  }
  return player1Wins > player2Wins ? player1 :
player2
}
```

I'm going to leave this as it is, and we're going to reveal this bug in our test.
Jest has built into it a function called jest.fn (https://jestjs.io/docs/en/jest-object#jestfnimplementation), which is short for function.

You can provide it an implementation, this is called a mock function, and it keeps track of what arguments get called with it. Now, we can `expect(utils.getWinner).toHaveBeenCalledTimes(2)`. The test is still passing.

Next, let's add `expect(utils.getWinner).toHaveBeenCalledWith('Kent C. Dodds', 'Ken Wheeler')`.

mock-fn.js

```
test('returns winner', () => {
  const originalGetWinner = utils.getWinner
  utils.getWinner = jest.fn((p1, p2) => p1)

  const winner = thumbWar('Kent C. Dodds', 'Ken
Wheeler')
  expect(winner).toBe('Kent C. Dodds)

expect(utils.getWinner).toHaveBeenCalledTimes(2)

expect(utils.getWinner).toHaveBeenCalledWith('Ke
nt C. Dodds', 'Ken Wheeler')
})

// cleanup
utils.getWinner = originalGetWinner
```

Now, we're going to see that error. Let's go ahead and fix that by passing `player2` now.
Now, we are verifying that it's being called properly.

thumb-war.js

```
const winner = utils.getWinner(player1, player2)
// player 2 re-added
```

Because we're calling it two times, we also may want to verify that it's being called with the right things at the right time.
We can also say

`expect(utils.getWinner).toHaveBeenNthCalledWith(1, 'Kent C. Dodds', 'Ken Wheeler')`.
On the second time, it's called in the same way.

mock-fn.js

```
test('returns winner', () => {
  const originalGetWinner = utils.getWinner
  utils.getWinner = jest.fn((p1, p2) => p1)

  const winner = thumbWar('Kent C. Dodds', 'Ken
Wheeler')
  expect(winner).toBe('Kent C. Dodds)

expect(utils.getWinner).toHaveBeenCalledTimes(2)

expect(utils.getWinner).toHaveBeenCalledWith('Ke
nt C. Dodds', 'Ken Wheeler')

expect(utils.getWinner).toHaveBeenNthCalledWith(
1, 'Kent C. Dodds', 'Ken Wheeler')

expect(utils.getWinner).toHaveBeenNthCalledWith(
2, 'Kent C. Dodds', 'Ken Wheeler')
})
```

That gives us a fair amount of control. We could improve this further by inspecting what `utils.getWinner` is.
Let's add a `console.log()` that consoles out `utils.getWinner`.
Here, we can see that it's a function that has a whole bunch of properties on it.

console.log(utils.getWinner)

```
JS mock-fn.js src/__tests__

const utils = require('../utils')

test('returns winner', () => {
  const originalGetWinner = utils.getWinner
  utils.getWinner = jest.fn((p1, p2) => p1)

  const winner = thumbWar('Kent C. Dodds', 'Ken Wheeler')
  expect(winner).toBe('Kent C. Dodds')
  console.log(utils.getWinner)
  expect(utils.getWinner).toHaveBeenCalledTimes(2)
  expect(utils.getWinner).toHaveBeenCalledWith('Kent C. Dodds', 'Ken Wheeler')
  expect(utils.getWinner).toHaveBeenNthCalledWith(
    1,
    'Kent C. Dodds',
    'Ken Wheeler'
  )
  expect(utils.getWinner).toHaveBeenNthCalledWith(
    2,
    'Kent C. Dodds',
    'Ken Wheeler'
  )
```

```
TERMINAL    1: node

PASS  jest  src/__tests__/mock-fn.js
  ✓ returns winner (5ms)

  console.log src/__tests__/mock-fn.js:10
    { [Function: mockConstructor]
      _isMockFunction: true,
      getMockImplementation: [Function],
      mock: [Getter/Setter],
      mockClear: [Function],
      mockReset: [Function],
      mockRestore: [Function],
      mockReturnValueOnce: [Function],
      mockResolvedValueOnce: [Function],
      mockRejectedValueOnce: [Function],
      mockReturnValue: [Function],
      mockResolvedValue: [Function],
      mockRejectedValue: [Function],
      mockImplementationOnce: [Function],
      mockImplementation: [Function],
      mockReturnThis: [Function],
      mockName: [Function],
      getMockName: [Function] }

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.165s, estimated 1s
Ran all test suites matching /__tests__\/mock-fn/i.

Watch Usage: Press w to show more.
```

Here's a `mock` property. Let's take a look at that. The `mock` is an object that has a `calls` property, which is an array that holds all of the arguments that this function is called with.

console output for the mock property

```
{ calls:
    [ [ 'Kent C. Dodds', 'Ken Wheeler' ],
      [ 'Kent C. Dodds', 'Ken Wheeler' ]
    ],
    ...
}
```

We could actually take that,
and do `expect(utils.getWinner.mock.calls).toEqual()`
what we copied.

mock-fn.js

```
expect(utils.getWinner.mock.calls).toEqual([
  [ 'Kent C. Dodds', 'Ken Wheeler' ],
  [ 'Kent C. Dodds', 'Ken Wheeler' ]
])
```

That can cover us for all of these other assertions. Let's go ahead and see how we could implement this ourselves. We'll create our own `fn` function. That's going to take an `impl`, or an implementation. We'll create our own `mockFn` here.

That'll take any number of `args`, and it'll `return` the `impl`, forwarding on the `args`. Then let's `return` that `mockFn`. Now, we could wrap this `mockFn` here with our `fn` function, and everything is still passing like it was before.

mock-fn.js

```
function fn(impl) {
  const mockFn = (...args) => {
    return impl(...args)
  }
  return mockFn
}

const originalGetWinner = utils.getWinner
utils.getWinner = fn((p1, p2) => p1)
```

Next, let's go ahead and add `mockFn.mock` equals this object with that `calls:` that we saw from Jest. Then inside of our `mockFn` body, we could say `mockFn.mock.calls.push(args)`. Now, if we

come down here, we `console.log(utils.getWinner.mock.calls)`, and we'll see exactly the same thing we saw before.

mock-fn.js

```
function fn(impl) {
  const mockFn = (...args) => {
    mockFn.mock.calls.push(args)
    return impl(...args)
  }
  mockFn.mock = {calls: []}
  return mockFn
}

...

assert.strictEqual(winner, 'Kent C. Dodds')
console.log(utils.getWinner.mock.calls)
```

We could `assert.deepStrictEqual(utils.getWinner.mock.calls, ...)` with this array that we copied.
That verifies that it was indeed called twice and that these are the arguments it was called with.

mock-fn.js

```
assert.strictEqual(winner, 'Kent C. Dodds')
assert.deepStrictEqual(utils.getWinner.mock.call
s, [
  [ 'Kent C. Dodds', 'Ken Wheeler' ],
  [ 'Kent C. Dodds', 'Ken Wheeler' ]
])
```

In review, fundamentally, this `fn` function accepts an implementation and returns a function that calls that implementation with all of those arguments.

It also keeps track of all the arguments that it's called with so that we can assert how that function is called, allowing us to catch issues in our integration with the `getWinner` function.

Restore the Original Implementation of a Mocked JavaScript Function with jest.spyOn

Having to keep track of the `originalGetWinner` and restoring it at the end of our test is annoying.
Jest exposes another utility that we can use to simplify this.
We can run `jest.spyOn` and pass `utils` as the object and `'getWinner'` as the method.

spy.js

```
test('returns winner', () => {
  jest.spyOn(utils, 'getWinner')
  const originalGetWinner = utils.getWinner
  utils.getWinner = jest.fn((p1, p2) => p2)

  ...
```

With this, we no longer need to keep track of the `originalGetWinner`.
Instead, we can say, `utils.getWinner.mockRestore()`
The `.spyOn` method will replace the `getWinner` on utils with an empty mock function.

```
test('returns winner', () => {
  jest.spyOn(utils, 'getWinner')
  utils.getWinner = jest.fn((p1, p2) => p2)

  ...

  // cleanup
  utils.getWinner.mockRestore()
```

We have a specific implementation that we want to use for our mock function.
Mock functions have an additional method on them called `mockImplementation`.

Here, we can pass the `mockImplementation` we want to be applied. With this, our tests are still passing.
We can use all the regular assertions from Jest that we like.

```
test('returns winner', () => {
  jest.spyOn(utils, 'getWinner')
  utils.getWinner.mockImplementation((p1, p2) =>
p2)
```

Let's see how we could implement this ourselves. First of all, let's get rid of this `originalGetWinner`.
We'll get rid of that down here below as well. We'll call

`mockRestore`.

Here, we'll call a `.spyOn` function that we'll write. We'll pass in `utils` and `getWinner`.
Then we can say, `utils.getWinner.mockImplementation` and pass our `mockImplementation`.

```
// removed const originalGetWinner =
utils.getWinner
spyOn(utils, 'getWinner')
utils.getWinner.mockImplementation((p1, p2) =>
p2)

...

// cleanup
utils.getWinner.mockRestore()
```

Let's implement this `.spyOn` function that's going to take an `obj` and a `prop`.
Then we'll get `const originalValue = obj[prop]`. We'll then set `obj[prop]` with a mock function.

With this API, we also need to provide a default implementation. We'll make that be an empty arrow function.

```
function fn(impl = () => {}) {...}

function spyOn(obj, prop) {
  const originalValue = obj[prop]
  obj[prop] = fn()
}
```

Then we'll add object at that `obj[prop].mockRestore` equals an arrow
function that simply sets `=> (obj[prop] = originalValue)`.

```
function spyOn(obj, prop) {
  const originalValue = obj[prop]
  obj[prop] = fn()
  obj[prop].mockRestore = () => (obj[prop] =
originalValue)
}
```

Next, let's go ahead and add `mockFn.mockImplementation`, which will be an arrow function that accepts a `newImpl` and assigns `impl` to that `newImpl`. With that, our tests are passing.

```
function fn(impl = () => {}) {
  const mockFn = (...args) => {
    mockFn.mock.calls.push(args)
    return impl(...args)
  }
  mockFn.mock = {calls: []}
  mockFn.mockImplementation = newImpl => (impl =
newImpl)
  return mockFn
}
```

In review, our `spyOn` function takes an object and a prop. It is responsible for tracking the `originalValue`.
Then it provides a `mockRestore` function, which we can use to restore the `originalValue` to that object.

```
function spyOn(obj, prop) {
  const originalValue = obj[prop]
  obj[prop] = fn()
  obj[prop].mockRestore = () => (obj[prop] =
originalValue)
}
```

We also added this `mockImplementation` to our mock function factory `fn()` so we could continue to mock our implementation so that our tests can be deterministic.

Mock a JavaScript module in a test

What we're doing here with the `spyOn` is still a form of monkey patching.
It works because the `thumb-war` module is using `utils.getWinner`, but that only works because we're using common JS.
In an ES module situation, monkey patching doesn't work.

We need to take things a little bit further so that we can mock the entire module,
and Jest allows you to do this with the `jest.mock` API. The first argument to `jest.mock` is the path to the module that you're mocking, and that's relative to our `jest.mock` is being called.

For us, that is this `'../utils'`. The second argument is a module factory function that will `return` the mocked version of the module. Here, we can `return` an object that has `getWinner` and that would be a
`jest.fn()` with our mock implementation.

With that, we can remove both of these. For the cleanup, we want to run `mockReset()`.

inline-module-mock

```
jest.mock('../utils', () => {
  return {
    getWinner: jest.fn((p1, p2) => p1)
  }
})

test('returns winner', () => {
  const winner = thumbWar('Kent C. Dodds', 'Ken
Wheeler')
  expect(winner).toBe('Kent C. Dodds')

expect(utilsMock.getWinner.mock.calls).toEqual([
    ['Kent C. Dodds', 'Ken Wheeler'],
    ['Kent C. Dodds', 'Ken Wheeler']
  ])

  // cleanup
  utils.getWinner.mockReset()
})
```

That will reset our mock function to the initial state clearing out the calls. With that our tests are passing.
Let's go ahead and see what this would look like implementing this on our own.

`jest.mock` works, because Jest is in control of the whole module system.
We can simulate that same kind of control by using the

`require.cache`. Here, before any of our modules are required, let's go ahead and initialize the `require.cache` to have our mock version of the `utils` module.

Let's get a look at what the `require.cache` looks like. If we `console.log(require.cache)`, we're going to see a big object with keys that are paths to modules, and the value is a `Module` object.

Console Output

```
{ '/Users/kdodds/Developer/js-mocking-
fundamentals/node_modules/jest-
worker/build/child.js':
  Module {
    id: '.',
    exports: {},
    parent: null,
    ...
  }
}
```

Let's go ahead and make an entry into the `require.cache`, so that when the `utils` module is required, it gets our `require.cache` version rather than actually requiring the file.

Let's go ahead and get the `utilsPath` with `require.resolve('../utils')`. We'll then say `require.cache` at the `utilsPath` equals an object, and this object needs to resemble a module, so we'll say `id` is `utilsPath`, the `filename` is `utilsPath`, `loaded` is `true`, and `exports` is our mock, so we'll say `getWinner` is a call to our function with `p1`, `p2` always returning `p1`.

inline-module-mock.js

```
const utilsPath = require.resolve('../utils')
require.cache[utilsPath] = {
  id: utilsPath,
  filename: utilsPath,
  loaded: true,
  exports: {
    getWinner: fn((p1, p2) => p1)
  }
}
```

With that, we can change things here a little bit. Let's put `fn()` up here at the top. We'll get rid of this `spyOn` function, we don't need that anymore. We'll get rid of our `getWinner.mockImplementation` call here.

We'll change our cleanup to `delete require.cache` at that `utilsPath`. That way any other modules that want to use the `utils` can do so without having trouble with our module mocking them out.

```
// cleanup
delete require.cache[utilsPath]
```

If I save that, our tests are still passing. This isn't something that you'd typically want to do, but this is similar to how things are working, and Jest has total control over the module system, and I can do special things like this.

For us, we have control over the module system using the `require.cache`.
We can preload the `require.cache` with the mock module that we want to have loaded when `thumb-war` requires the `utils` module.

```
require.cache[utilsPath] = {
  id: utilsPath,
  filename: utilsPath,
  loaded: true,
  exports: {
    getWinner: fn((p1, p2) => p1)
  }
}
```

In Jest, we can put this `jest.mock` call anywhere, and Jest will ensure that our mock is used when the `thumb-war` requires the `utils` module. An interesting fact with the way that it works is, before Jest runs our code, it transforms that to move the `jest.mock` call up to the top of the file to ensure that the mock is in place before any of our modules are loaded.

```
jest.mock('../utils', () => {
  return {
    getWinner: jest.fn((p1, p2) => p1)
  }
})
```

We can move this `jest.mock` call down below our `require` calls. This is especially useful with ES modules, where the imports are always hoisted to the top of the file.

Make a shared JavaScript mock module

Kent C Dodds: 00:00 Often, with modules that you want to *mock* in one file, you'll probably also want to *mock* it in multiple files. Jest allows you to externalize your *mock* by using a `__mocks__` directory.

external-mock-module.js

```
jest.mock('../utils', () => {
  return {
    getWinner: jest.fn((p1, p2) => p1)
  }
})
```

00:11 What you do is create a directory with `__mocks__` and then a file that has the name of the module that you want to *mock*. In our case, that's `utils.js`. Then in that `utils.js`, we place the *mock* that we want to use.

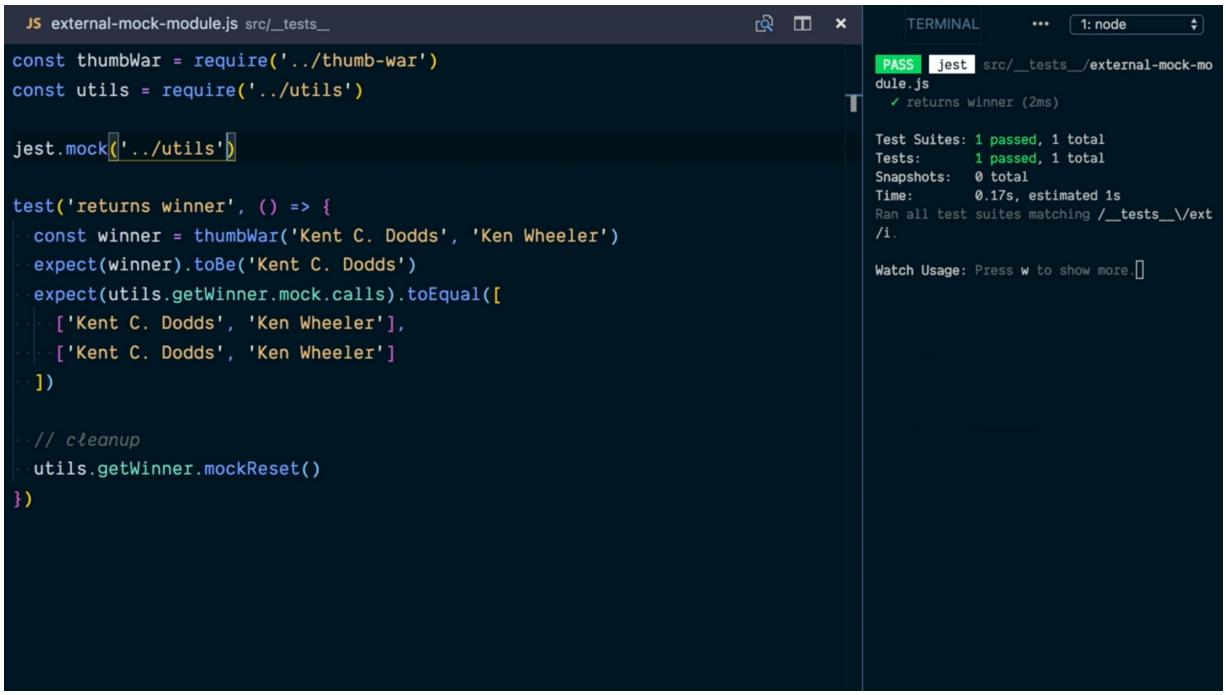
00:27 We'll take this, and we'll `module.exports` the inline *mock* that we had before.

mocks/utils.js

```
module.exports = {
  getWinner: jest.fn((p1, p2) => p1)
}
```

Then we can go back to our test file, and we can remove the second argument from our `jest.mock`.
Jest will automatically pick up the *mock* file that we have created. Our test still works!

Test still works!




```
jest.mock('../utils')
```

00:45 Let's see how we could implement this ourselves. Let's start by creating our own little `__no-framework-mocks__` directory, and we'll have a `utils.js` inside of there. We'll pull out this function that we have here, and we'll `module.exports` the *mock* that we want to have in place.

no-framework-mocks/utils.js

```
function fn(impl = () => {}) {
  const mockFn = (...args) => {
    mockFn.mock.calls.push(args)
    return impl(...args)
  }
  mockFn.mock = {calls: []}
  return mockFn
}

module.exports = {
  getWinner: fn((p1, p2) => p1)
}
```

01:05 Let's go ahead and prime the cache by requiring `__no-framework-mocks__/utils`.
With that now, we can get `const mockUtilsPath = require.resolve('../__no-framework-mocks__/utils')`.
Instead of this `require.cache` object inline here, because we primed the cache with `require.cache[mockUtilsPath]` , we'll have that object, and we can just assign the `require.cache` at the `utilsPath` to be the same thing.

external-mock-module.js

```
require('../__no-framework-mocks__/utils')
const utilsPath = require.resolve('../utils')
const mockUtilsPath = require.resolve('../__no-framework-mocks__/utils')
require.cache[utilsPath] = require.cache[mockUtilsPath]
```

01:36 If we save our file, our test rerun, and everything is working just fine.
Again, this isn't precisely what Jest is doing, because it is in complete control over the module system. So when our code requires the `utils` module, whether that be in our test file or our implementation file, Jest will provide the proper *mock* for it.