Configure Jest for Testing JavaScript Applications

Transcripts for Kent C. Dodds (https://egghead.io/instructors/kentcdodds) course on egghead.io (https://egghead.io/courses/configure-jest-for-testing-javascript-applications).

Description

Jest is a fully featured testing framework with a developer experience that is second to none. It’s remarkably simple and flexible at the same time. For simple use cases, you often don’t need to configure anything, install and enjoy the built-in coverage and watch mode support.

In a real-world application though, you’ll often have needs specific to your application, especially when testing browser-based applications. You'll need to handle Webpack loaders, dynamic imports, and custom module resolution which Node.js does not support.

In this course we'll go over ways you can optimize your Jest configuration to make testing real-world JavaScript applications a delight. We'll cover what's already been mentioned in addition to Babel support, code coverage, how to make watch mode even more helpful, and how to run test suites with entirely different configurations.

Install and run Jest

To get started with Jest in an existing project, we're going to `npm install` as a dev dependency Jest.

Terminal

```
$ npm install --save-dev jest
```

With Jest installed and saved into our `package.json`, we can now use the Jest binary that's been installed into our node modules in `.bin`. We'll find Jest right there. With that, we can use it in a `test` script. We'll just put Jest here. Let's go ahead and run that `test` script with `npm run test`

```
$ npm run test
```

or `npm test`

```
$ npm test
```

or `npm t`.

```
$ npm t
```

All three of those will run the same thing. First, we're going to get an error, "No test found."

```
$ npm t
> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

No test found

...
npm ERR! Test failed.  See above for more
details.
```

Let's go ahead and make a test, just an example. I'll add a test directory here. In that directory, we'll have an `example.js`. Here, we'll have a simple test that simply says, "It works."

example.js

```
test('works', () => {})
```

If we run `npm t`, it will run our test that says, "It works."

Terminal

```
$ npm t
> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS  src/__tests__/example.js
  ✓ works (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.542s
Ran all test suites.
```

Now we can start writing some tests with Jest. One last thing I'm going to do here is we have a Travis configuration that runs a validate script on CI. I want to run these tests as part of CI. I'm going to update that validate script to include `npm run test`.

```
"validate": "npm run lint && npm run test && npm
run build",
```

When somebody runs the `setup` script, which will run the `validate` script, or when CI runs the `validate` script, it's including our tests automatically.

In review, to make all of this work, we simply need to install Jest and then add a script called `test` to run Jest. Then we need to have a test file in our project.

By default, Jest has a test match that will match any file in a test directory or any file that ends in a `.spec` or `.test.js`. Here we created our test and an `example.js` file in there. When we run our `test` script, it works.

Transpile Modules with Babel in Jest Tests

This test isn't all that useful, so let's go ahead and delete it. We'll make a new one for this utility that we have here. I'm going to add a file next to it called `tests`, and then `utils.js`. Then I'll paste this simple test here. We `import {getFormattedValue}`, and use that in our test.

utils.js

```
import {getFormattedValue} from '../utils'

test('formats the value', () => {

expect(getFormattedValue('1234.0')).toBe('1,234.
0')
})
```

Now, if I run our test, I'm going to get a syntax error, "Unexpected token import."

Terminal

```
$ npm t
...
Unexpected token import
```

The reason this is happening is because ES modules are not supported in Node. Because Jest is running in Node, when it hits this `import` statement, we're getting that syntax error.

This project is using webpack, and it's using the Babel loader, so that we can use this kind of syntax. We have our Babel configured so that it can transpile those modules, except we have that disabled because we want webpack to utilize tree shaking.

We want this to be enabled during our test, but disabled during our production build. What we'll do is something very similar to this `isProd`. We'll make an `isTest`.

.babelrc.js

```
const isTest = String(process.env.NODE_ENV) ===
'test'
```

If the Node environment `isTest`, which is a default that Jest will set for us, then we'll add a ternary here. We'll say `isTest`, then `commonjs`, otherwise `false`.

.babelrc.js

```
presets: [
    ['@babel/preset-env', {modules: isTest ?
'commonjs' : false}],
```

That's saying, "Hey, Babel preset env. When you come across a module, and if we're in the test environment, I want you to transpile that to `commonjs` so that it works in Node. Otherwise, we're probably building with webpack, and so, I don't want you to transpile that." Webpack can take over from there. With this in place, we can now run our tests again, and our tests are passing.

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS  src/shared/__tests__/utils.js
  ✓ formats the value (24ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.913s
```

An important thing to note here is that we didn't actually have to configure Jest at all to make this work.

If we open up our `package.json` again, we'll see that we have Jest installed, and we'll see that we have a test that is specified to run Jest, but we don't have any configuration that is necessary to let Jest run our Babel configuration on our source code and our test code.

Jest will see that we have this `babelrc` file in our project, and it will automatically pick it up, and run all of our source and test files through that configuration. The only thing we needed to make sure of is that our Babel configuration worked for Node.

Configure Jest’s test environment for testing node or browser code

Jest does a lot of awesome things for us by default. One of those awesome things is simulating a browser environment in Node. Here I can add a `console.log(window)`,

utils.js

```
console.log(window)
```

and if I run my test, I'm going to see that huge object logged to the console.

Terminal

```
npm t
```

For these tests, we're not actually using `window` at all. We're not using the browser environment. This all can run in Node or the browser. The way Jest is doing this is it's using a module called JSDOM which will simulate this browser environment. There's a little bit of a performance hit for Jest to set up this JSDOM test environment. If you're just writing code that can run in Node then it's better to tell Jest to run that code in a Node test environment and not set up JSDOM.

utils.js

```
function getFormattedValue(value, language =
'en-US') {
  let formattedValue =
parseFloat(value).toLocaleString(language, {
    useGrouping: true,
    maximumFractionDigits: 6,
  })

  // Add back missing .0 in e.g. 12.0
  const match = value.match(/\.\d*?(0*)$/)

  if (match) {
    formattedValue += /[1-9]/.test(match[0]) ?
match[1] : match[0]
  }
  return formattedValue
}

export {getFormattedValue}
```

We can do this simply by running `npm t` and then passing on the argument `"--env=node"`. That's going to throw an error because now `window` is not defined.

Terminal

```
npm t -- --env-node
...
Reference error: window is not defined
    ...
        7 console.log(window)
    ...
```

Let's go ahead and set up a configuration that will set this for us. I'm going to add a configuration file to the root of the project called `jest.config.js`, and Jest will pick up this configuration file by default.

Here I'm going to say `module.exports` equals this object, and we'll have a property called `testEnvironment`, and that's going to be `'jest-environment-node'`.

jest.config.js

```
    module.exports = {
        testEnvironment: 'jest-environment-
node',
    }
```

Now if I run my test, I'm going to get that error again because `window` is no longer established in this environment we're using Node.

Terminal

```
npm t
...
Reference error: window is not defined
    ...
        7 console.log(window)
    ...
```

If I want to specify that environment then I can say `'jest-environment-jsdom'`.

jest.config.js

```
    module.exports = {
        testEnvironment: 'jest-environment-
jsdom',
    }
```

I'll run my test again, and I get that huge object logged to the console again.

Terminal

```
npm t
```

I'm going to leave it this way because this is a project that's intended for the browser, so we want our Jest environment to be JSDOM. I'm going to make it explicit even though that's the default.

I'm going to go ahead and get back here into `utils.js` and remove that `console.log(window)` because that's kind of annoying. We'll run our test again to make sure that all goes away.

Perfect. In review, this is how we establish our Jest configuration. We make a `jest.config` file here, your `module.export`, your configuration, and then Jest will pick it up automatically when you run the Jest command.

Support importing CSS files with Jest's moduleNameMapper

Let's go ahead and write a test for this React component. I'm going to create a new file called `auto-scaling-text.js` inside of that `__test__` directory, right next to that file I want to test. I'm going to go ahead and paste in a test here.

auto-scaling-text.js

```
import 'react-testing-library/cleanup-after-each'
import React from 'react'
import {render} from 'react-testing-library'
import AutoScalingText from '../auto-scaling-text'

test('renders', () => {
    render(<AutoScalingText />)
})
```

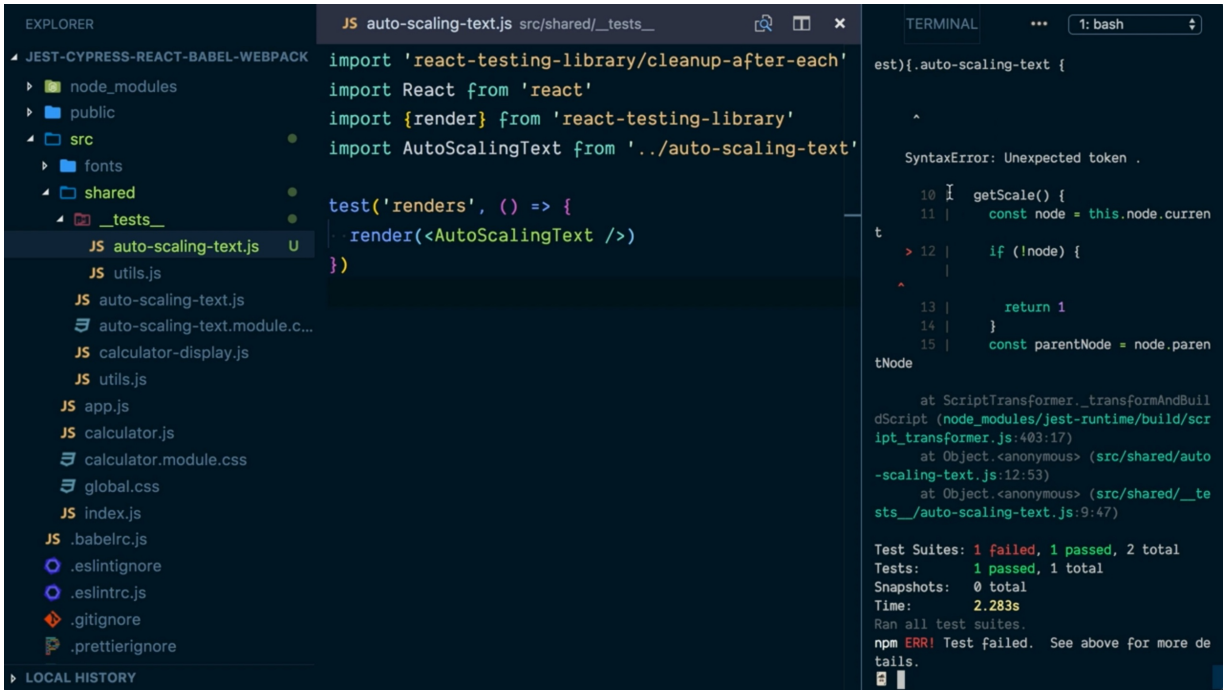
Here I'm going to need the `react-testing-library`, so I'll go ahead and `npm install` as a dev dependency to `react-testing-library`. All this test is doing is making sure that this component will render.

Terminal

```
$ npm install --save-dev react-testing-library
```

With that installed let's go ahead and run our test with `npm t`, and we're going to get a syntax error here.

Syntax Error



This message is a little bit confusing. If you look right here that's where it's pointing -- `auto-scaling-text`. That's coming from this `auto-scaling-text` CSS file. The reason that it's throwing that error is if we look at the `auto-scaling-text` file, we're importing styles from `auto-scaling-text.module.css`.

Jest is running these imports in Node. When it sees this, it's going to treat that file as a Node module. With Webpack, we have a loader and that's how this works.

We have this `css-loader` and this `style-loader`, so that we can `import` the CSS files and use the CSS modules in our React components.

Let's see how we can go ahead and fix this using our Jest configuration. We need to tell Jest that any time it comes across a file that ends in `.css`, we want it to resolve to some other module, or we want to give it some kind of module for it to resolve to instead of trying to `import` that `.css` file itself.

In our `jest.config.js`, we can add a `moduleNameMapper` object here. The key is a string regexp that matches the file we want to map to. Here we'll say it ends in `.css`. We want this to map to a new file that we're going to create in a `test` directory. We'll just call it `style-mock.js`.

jest.config.js

```
module.exports = {
    testEnvironment: 'jest-environment-jsdom',
    moduleNameMapper: {
        '\\.css
:
    }
}
```

A style mock, it doesn't need to be anything special. We'll just say `module.exports` equals this empty object.

style-mock.js

```
module.exports = {}
```

Then back in our configuration, we want this resolve to that module . We'll say `require.resolve('./test/style-mock.js')`.

jest.config.js

```
moduleNameMapper: {
    '\\.css







: require.resolve('./test/style-mock.js'),
  }
```

With that in place, we can run our test again, and our tests are passing. Now we can test files that are using imports of CSS.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/auto-scaling-text.j
s

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        2.334s
Ran all test suites.
```

We can do this with any kind of loader. If it's a `.graphql` import or an `.svg` import, any of those kinds of things we can make our own `moduleNameMapper` and load different kinds of modules based off of the file name. That will cover the custom Webpack loaders that we have defined in our Webpack config.

Support using webpack CSS modules with Jest

This `import` of `styles` from the CSS files actually using CSS modules. We're using `styles.autoScalingText` as the `className`. If we look at our auto-scaling text test file and we get the container from this render, then we'll console log the `container.innerHTML`.

__tests__/auto-scaling-text.js

```
import 'react-testing-library/cleanup-after-
each'
import React from 'react'
import {render} from 'react-testing-library'
import AutoScalingText from '../auto-scaling-
text'

test('renders', () => {
  const {container} = render(<AutoScalingText
/>)
  console.log(container.innerHTML)
})
```

Then, we can run our test. Here, we see this `<div style="transform: scale(1,1);"></div>`, but we don't see the `className` there at all. That's because the `import` of CSS is resolving to this `style-mock.js`, rather than the actual CSS loaded content. In here, when we're saying a `styles.autoScalingText`, that's actually an empty object `.autoScalingText` which will be `undefined`.

This is probably fine for our use cases, but it would be really nice if we could see the `className` here that is associated to the `autoScalingText`. In reality, this `autoScalingText className` is going to be a generated value that's a hash of some sort, because we're using CSS modules. It would be nice to have something here to give us an indication that the `className` is there.

Let's go ahead and do that. We're going to clear this. I'm going to `npm install --save-dev identity-obj-proxy`. With that installed in our dev dependencies, we can now use that in our Jest configuration.

Before resolving a CSS file, let's have it resolve to a file that ends in `.module.css identity-obj-proxy`.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleNameMapper: {
    '\\.module\\.css
: 'identity-obj-proxy',
    '\\.css
: require.resolve('./test/style-mock.js'),
  },
}
```

Now if we run our test again with `npm t`, we're going to see the output includes the property that we're trying to access.
`<div class= "autoScalingText" style="transform: scale(1,1);"></div>`

Again, this isn't what is actually going to be. It's going to be a generated value because this is CSS modules. It is a lot more helpful to have something like that there in our output, both when we're debugging and if we want to take a snapshot of this HTML.

In review, all that we really had to do here is install `identity-obj-proxy`, then use that in our `jest.config.js` with a different `moduleNameMapper`.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleNameMapper: {
    '\\.module\\.css

: 'identity-obj-proxy',
    '\\.css

: require.resolve('./test/style-mock.js'),
  },
}
```

Now, it's important to have these in the right order, because these both will match the `module.css` files, but we want any `module.css` to match the `identity-obj-proxy` instead.

Generate a Serializable Value with Jest Snapshots

Here, I have a list of `superHeros` and I have this function called `getFlyingSuperHeros` that will take all the `superHeros` and `filter` them down to heroes whose powers include `'fly'`. Then, I'm going to `export` that function and I want to write a test for that.

super-heros.js

```
const superHeros = [
    {name: 'Dynaguy', powers: ['disintegration
ray', 'fly']},
    {name: 'Apogee', powers: ['gravity control',
'fly']},
    {name: 'Blazestone', powers: ['control of
fire', 'pyrotechnic discharges']},
    {name: 'Froozone', powers: ['freeze
water']},
    {name: 'Mr. Incredible', powers: ['physical
strength']},
    {name: 'Elastigirl', powers: ['physical
stretch']},
    {name: 'Violet', powers: ['invisibility',
'force fields'],
    {name: 'Dash', powers: ['speed']},
    // {name: 'Jack-Jack', powers:
['shapeshifting', 'fly']}
]

function getFlyingSuperHeros() {
    return superHeros.fliter(hero => {
            return hero.powers.includes('fly')
    })
}

export {getFlyingSuperheros}
```

I got one set up right here, where I'm importing `getFlyingSuperHeros`.

I'm calling that to get a list of `flyingHeros`, and now I want to write an assertion that will expect the `flyingHeroes` to equal an array of `superHeros`. How am I going to get that array of `superHeros`?

tests/super-heros.js

```
import {getFlyingSuperHeros} from '../super-
heros'

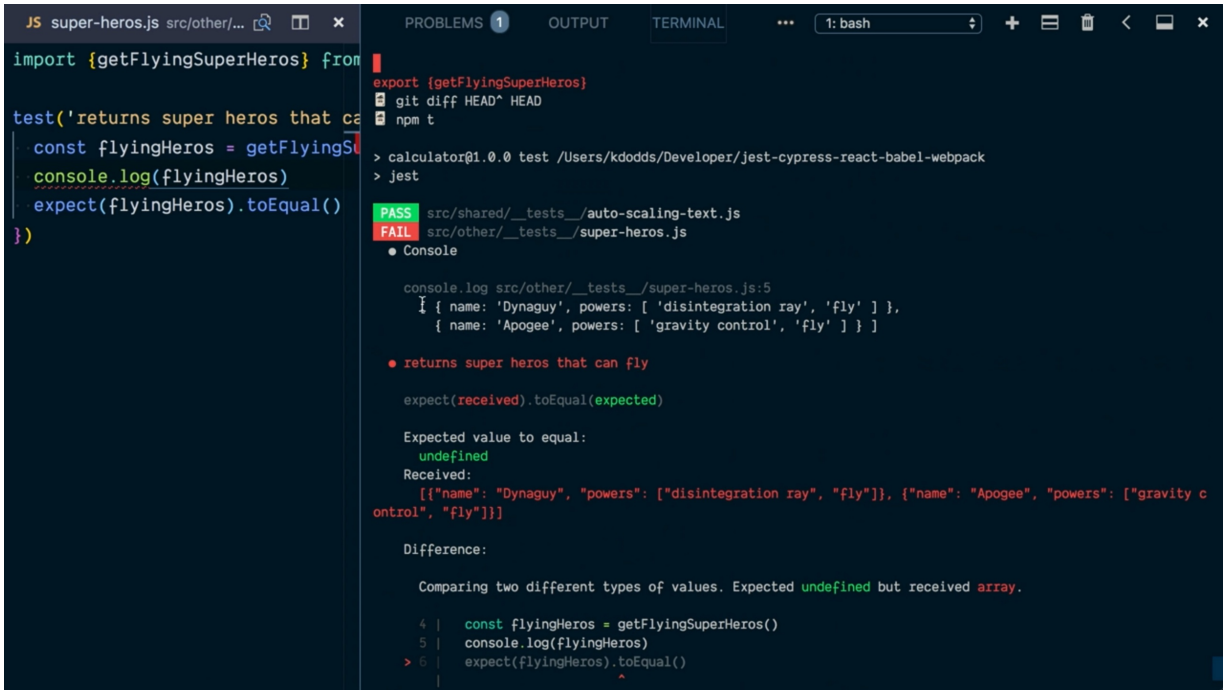
test('returns super heros that can fly', () => {
    const flyingHeros = getFlyingSuperHeros()
    expect(flyingHeros).toEqual()
})
```

I can just go in here and manually look for things, but what I'm going to do is, I'll just `console.log` the `flyingHeros`.

```
const flyingHeros = getFlyingSuperHeros(
console.log(flyingHeros)
expect(flyingHeros).toEqual()
```

I'll pop up in on my test, and I'll run `npm t`.

Error Output



That's going to fail here, but I'm going to get my `console.log` right there, and I'll copy that. I'll paste it right into the `toEqual` there.

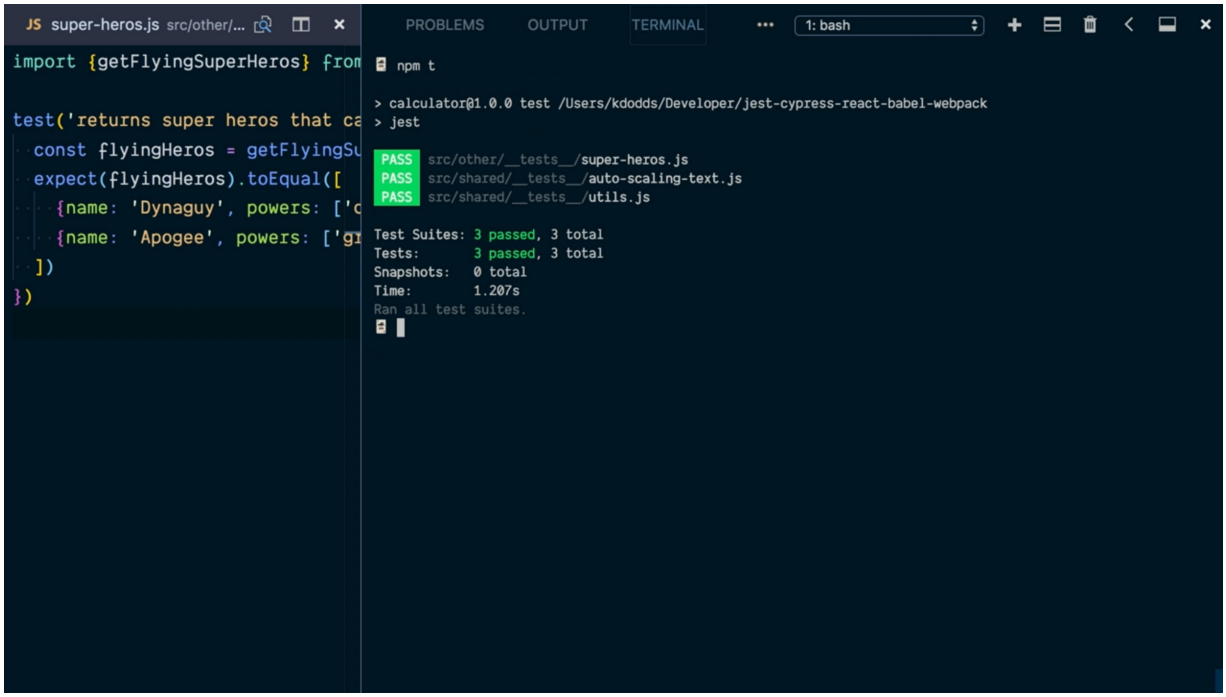
tests/super-heros.js

```
import {getFlyingSuperHeros} from '../super-
heros'

test('returns super heros that can fly', () => {
    const flyingHeros = getFlyingSuperHeros()
    expect(flyingHeros).toEqual([
        {name: 'Dynaguy', powers:
['disintegration ray', 'fly']},
        {name: 'Apogee', powers: ['gravity
control', 'fly']}
    ])
})
```

Now, if I pop up in my test and run `npm t` again, that test is going to pass.

Passing Test



Cool. Now if I run `git diff`, you're going to see exactly what I did here.

I'll go ahead and commit that, and I `git commit -am 'add assertion'`.

Terminal

```
$ git commit -am 'add assertion'
[egghead-2018/config-jest-05 8b68b87] add
assertion
  1 file changed 4 insertions(+)
```

Now, let's say later on down the line, we find out, "Oh hey, Jack Jack does have powers in fact," and those include shape shifting and fly.

super-heros.js

```
{name: 'Jack-Jack, powers: ['shapeshifting',
'fly']}
```

Now, I'll save that. I'm going to run my `test` again. I'm going to see this error output that says, "Hey, here is the object that is in the received value and the expected did not include that object", and so the test fails, because we didn't expect it to include that value, but we actually do want it to include the values.

Terminal

```
$ npm t
...
    Expected value to equal:
        [{name: 'Dynaguy', powers:
['disintegration ray' 'fly']},{name: 'Apogee',
powers: ['gravity control', 'fly']}]
    Recieved:
        [{name: 'Dynaguy', powers:
['disintegration ray', 'fly']},{name: 'Apogee',
powers: ['gravity control', 'fly']}, {name:
'Jack-Jack', powers: ['shapeshifting', 'fly']}]
...
```

Let's go ahead and we'll update our test again. I'm going to `console.log(flyingHeros)`. We'll run our test and come up here, we'll copy that and paste it here.

super-heros

```
expect(flyingHeros).toEqual([
    {name: 'Dynaguy', powers: ['disintegration
ray', 'fly']},
    {name: 'Apogee', powers: ['gravity control',
'fly']},
    {name: 'Jack-Jack, powers: ['shapeshifting',
'fly']}
])
```

We'll run our test again with `npm t`.

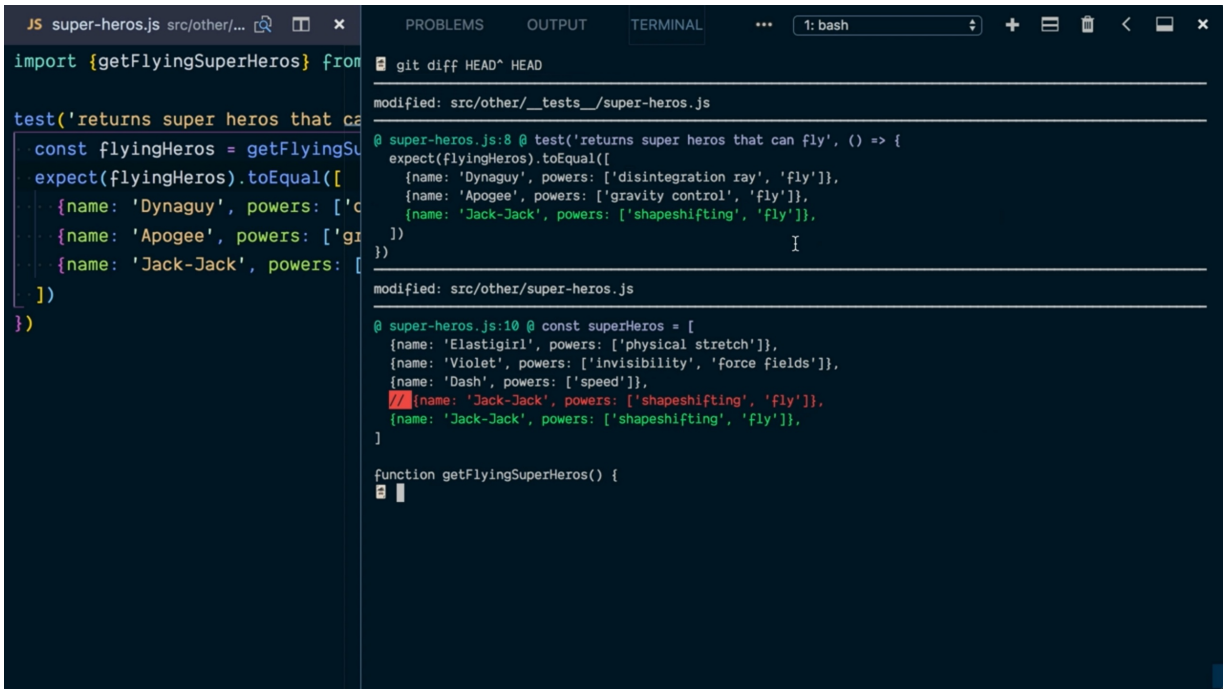
Awesome. Our tests our passing. Let's go ahead and remove that `console.log` now. We'll run `git commit -am 'jack-jack can fly!'`.

```
$ git commit -am 'jack-jack can fly'
```

Now, if I look at the `git diff` of the `HEAD^` previous and `HEAD` current, I'm going to see that is the change I made, and that's the impact of my change, and reviewers can review that. They'll know exactly what's going on here.

```
git diff HEAD^ HEAD
```

git diff



This process of copy paste update the assertion. It's kind of arduous and a little bit more worth than necessary. Let's take things a couple of steps back and see how our snapshots with Jest can make things a little more straightforward.

We're back with our list of `superHeros` that does not include `'Jack-Jack'`, and we haven't need our assertion yet. Instead of console logging, I'm going to do `expect(flyingHeros).toMatchSnapshot`.

super-heros.js

```
import {getFlyingSuperHeros} from '../super-
heros'

test('returns super heros that can fly', () => {
    const flyingHeros = getFlyingSuperHeros()
    expect(flyingHeros).toMatchSnapshot()
})
```

We'll save that and pop up in our terminal, and run our test. We'll see that we have one snapshot written.

Let's take a look at that snapshot. This snapshot assertion actually creates a new file in our file system under a `__snapshots__` directory, and with the same name as the test file with the `.snap`.

This is actually just a regular JavaScript module that add some properties to `exports` and the property value is a string. The string contents is the serialized version of the object that we're snapshotting.

super-heros.js-snap

```
exports[`returns super heros that can fly 1`] =
`
Array [
    Object {
        "name": "Dynaguy",
        "powers": Array [
            "disintegration ray",
            "fly",
        ],
    },
    Object {
        "name": "Apogee",
        "powers": Array [
            "gravity control",
            "fly",
        ],
    },
]
`;
```

In our case, we're taking flying heroes and serializing that into this string. It pretty prints it, so we know exactly what type of objects these are and the values of these properties. So a snapshot is an assertion that lives in two places. It lives here while you write the assertion, and the actual value it lives in the snapshot file. For this reason, you need to make sure that you commit your `__snapshots__` directories to your source control. I'm going to open up my terminal here and I'm going to look at the `git diff`. We'll see that assertion here, but also if we look at the `git status`, we're going to see that were not tracking the snapshot file. I'm going to go ahead and `git add .` everything. Then I'll

`git commit -am 'add assertion'`. Now, let's take a look what happens when we come down and say "Hey, Jack Jack can actually shape shift and he can fly."

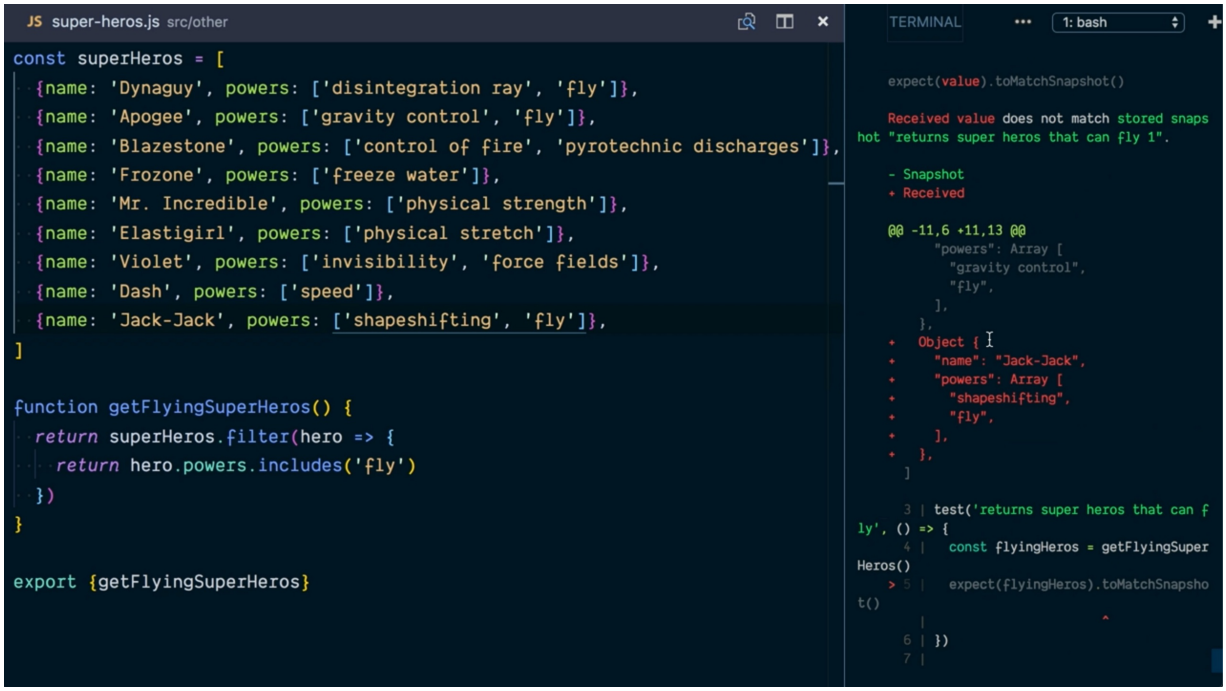
super-heros.js

```
{name: 'Jack-Jack', powers: ['shapeshifting',
'fly']}
```

Let's take a look at what happens to our snapshot when we run our test again with `npm t`.

This error output looks very familiar. It actually looks exactly like it did before when we're doing the actual `toEqual` assertion. It saying that the received value does not match the stored snapshot returns `superHeros` that can fly.

Received Value does not match snapshot



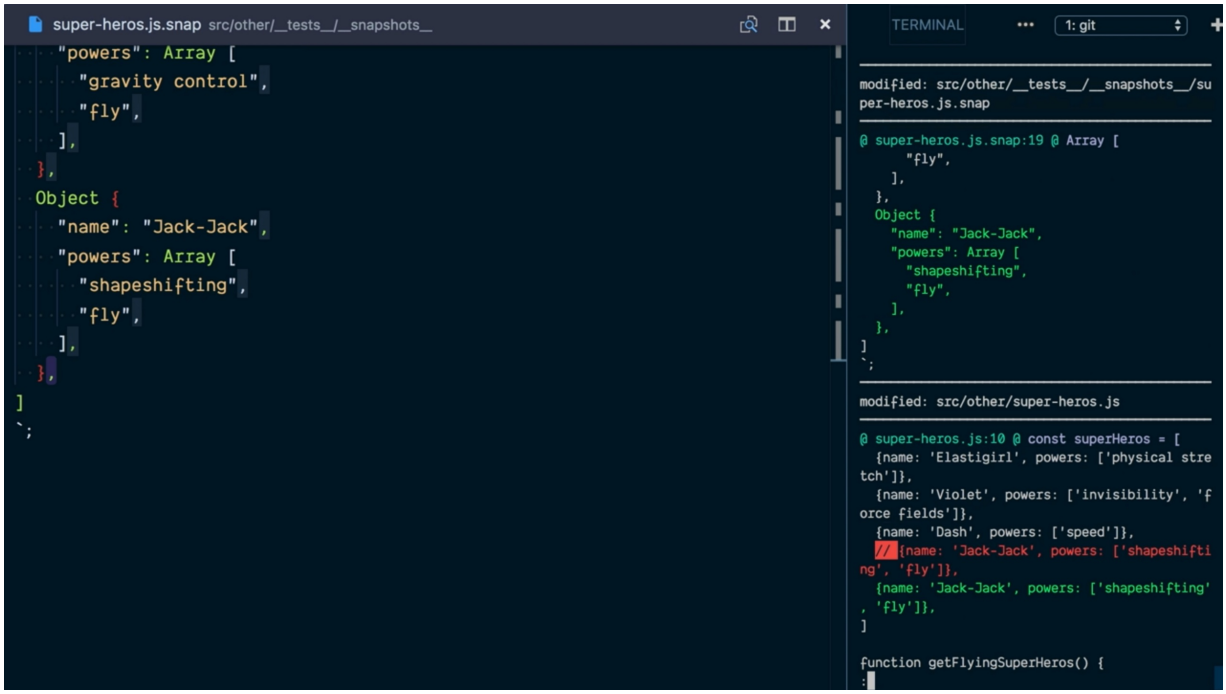
We'll scroll down here and we can look right here the `Snapshot Summary`. "1 snapshot fail from one test suite. Inspect your code changes or run `npm test -- -u` to update them".

What it saying here is that your assertion is now breaking. You need to update your assertion or fix your code. That's the same thing that you'd have, if you had a normal assertion. In our case, this is an intended change. We're going to update our assertion, but instead of doing a `console.log` and a copy paste, I can actually do as it suggesting here to `npm test -- -u` to run Jest with the update flag, and that will update my snapshot for me.

```
$ npm test -- -u
...
PASS src/other/__tests___/super-heros.js
  > 1 snapshot updated.
...
```

Now if I look at that snapshot, I'll see it's adding this `'Jack-Jack'`. Now if I look at the `git diff`, this is what people are going to see when they see my pull request.

git diff



We'll see that the snapshot was updated to add `'Jack-Jack'`, and we'll see in the source code. This list was updated to add `'Jack-Jack'` as well. This change made this impact. That's one of the nice things about `snapshots` is its really easy to see this impact that your changes are making on the rest of the code base.

Let's go and write a test for this `CalculatorDisplay` React component and use a snapshot to verify its structure. I'm going to add a new file on here called `calculator-display.js`. I'll just go ahead and paste a test in here.

We're going to use `react-testing-library`, the `render` method here and we'll pull in the `CalculatorDisplay`, and we'll render `CalculatorDisplay` with a default `value` of `0`. That's the `value` that it will display. Then, our `container` is what we're going to be snapshotting. Let's go ahead and `console.log(container.innerHTML)`.

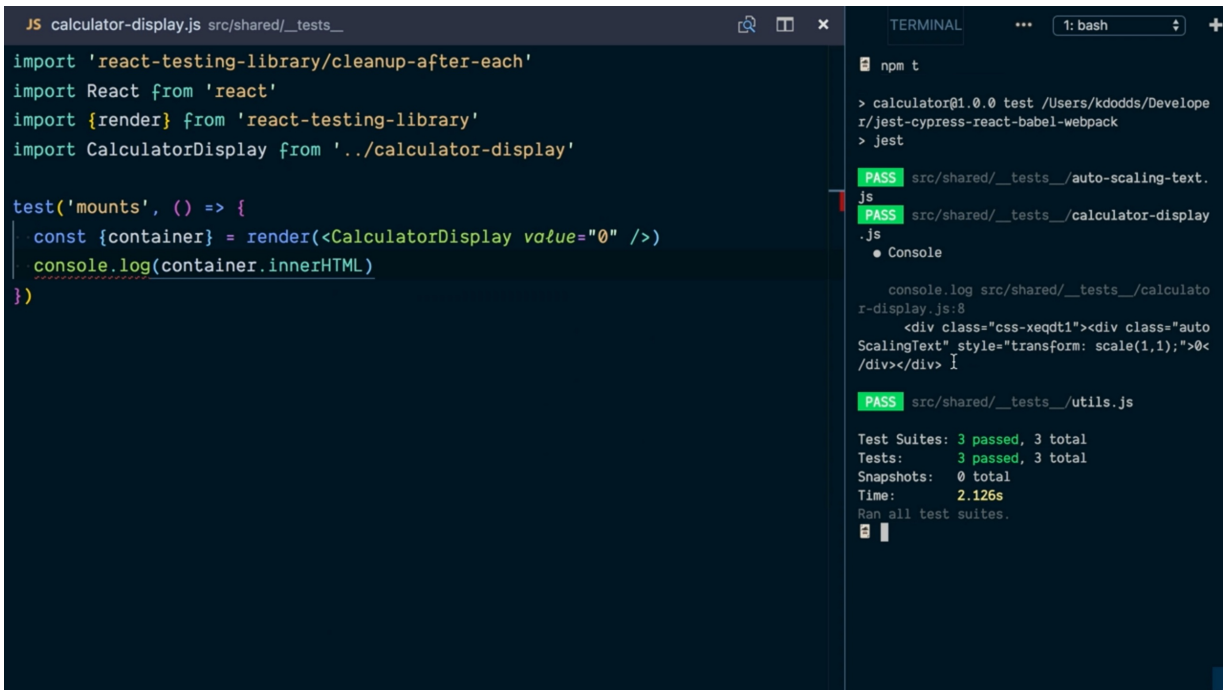
calculator-display.js

```
import 'react-testing-library/cleanup-after-each'
import React from 'react'
import {render} from 'react-testing-library'
import CalculatorDisplay from '../calculator-display'

test('mounts', () => {
  const {container} = render(<CalculatorDisplay value="0" />)
  console.log(container.innerHTML)
})
```

If I pull up my test and I run `npm t`, then we'll see that `console.log` HTML right here.

console.log



We could actually snapshot `container.innerHTML`, but the problem with that is any change to any of these attributes would negate the entire snapshot would be harder to read diffs of the snapshot.

It would be nice, if we could have them formatted. Well Jest has built in the capability of serializing and formatting DOM nodes. Let's go and take advantage of that. We're going to expect the `container` which is a DOM node `toMatchSnapshot`.

calculator-display.js

```
expect(container).toMatchSnapshot()
```

Now, if I run my test again, I'm going to see that one snapshot was written.

```
$ npm t
...
    Snapshot Summary
        > 1 snapshot written from 1 test suite
```

Let's pull that up. Here, it's been serialized and formatted. If we make any changes to the `value` here, if we add another `span` in the JSX, we're going to see those updates right here. Let's go ahead and take a look at that.

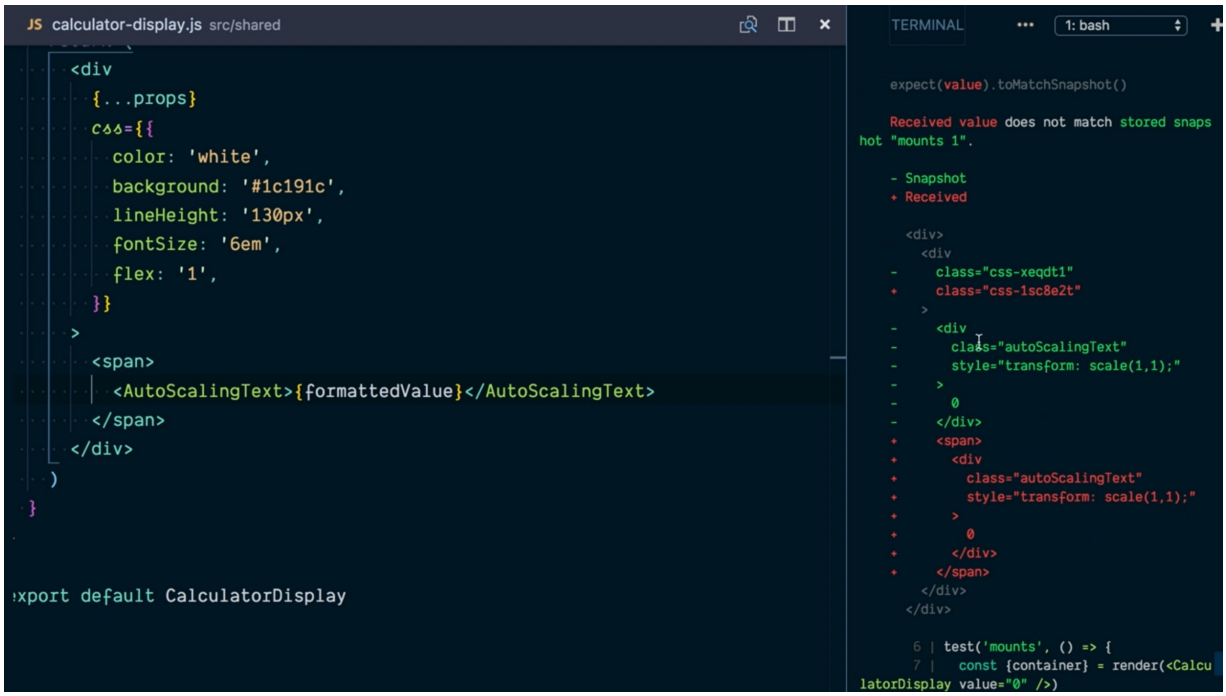
I'm going to add a `span` right here. We'll put the `AutoScalingText` in there.

calculator-display.js

```
<span>
    <AutoScalingText>{formattedValue}
</AutoScalingText>
</span>
```

We'll run our test again, and we'll see a snapshot diff right here saying, "Hey, this is a little bit different. You're going to need to go and take a look at that."

Error Message



Let's take a look at what happens, if I change this `div` to have an `id`. Here, I go down here, add a `id="hi"`,

calculator-display.js

```
<div
    id="hi"
    {...props}
    css={{
        color: 'white',
        background: '#1c191c',
        lineHeight: '130px',
        fontSize: '6em',
        flex: '1'
    }}
>
    <AutoScalingText>{formattedValue}
</AutoScalingText>
</div>
```

and we'll run our test again, and Jest sees that difference and shows us what's going on right here. The last step to this is our snapshot assertion is happening in two places. We need to make sure that we're adding test as well as the snapshot file to our git history, so that as changes are made, people can see those changes in git. `git add .` and we'll look at our `git status`. We have these two new files that we'll commit, `'test calculator-display'`. One last thing that specific to `react-testing-library` is the `container` is actually `div` and it will always be a div.

```
$ git add .
$ git status

$ git commit -am 'test calculator-display'
```

If you only have one child that you're rendering, then it doesn't actually make any sense to snapshot that diff. You could actually snapshot the `firstChild` which will be the root node of the thing that you're rendering. For us that's this `div` here.

calculator-display.js

```
expect(container.firstChild).toMatchSnapshot()
```

That way your snapshot would be a little less nested. If I now run `npm t -- -u` to update my snapshot, I can take a look at my snapshot.

```
npm t -- -u'
```

Now, it's less nested, and only includes the stuff that's specific for my component that I care about.

Test an Emotion Styled UI with Custom Jest Snapshot Serializers

We have this test that is running this snapshot and we have our snapshot here. There is one thing in the snapshot that bothers me and that's this `className`.

If we look at the implementation of our component, we're getting that `className` from the `css` prop, because we're using emotion's babel plugin that will take the `css` prop and turn it into a `className` that's generated with a hash. That's why every single time we make a change to this `css`, we're going to see a change in this `className` as demonstrated here.

If I change this to a `color` of blue and I run my test again, I'm going to get a snapshot failure.

calculator-display.js

```
<div
  {...props}
  css={{
    color: 'blue',
    background: '#1c191c',
    lineHeight: '130px',
    fontSize: '6em',
    flex: '1',
  }}
>
```

It's that `className`, and who knows exactly what the change was that impacted that, it doesn't really make a whole lot of sense.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 FAIL  src/shared/__tests__/calculator-
display.js
  ● mounts

    expect(value).toMatchSnapshot()

    Received value does not match stored
snapshot "mounts 1".

    - Snapshot
    + Received
```

It would be really nice if I could actually remove that generated `className` and actually put in the CSS that's being applied in this case.

One of the really cool things about Jest snapshots is the ability to write custom serializers. Jest automatically includes a serializer that can take a DOM node and serialize it into some HTML that's formatted nicely, but we can make our own serializer that can take any of these class names and replace them with the actual CSS that's going to be generated.

Somebody's already done that for emotion. Let's go ahead and install that with `npm i --save-dev jest-emotion`. With that installed into our dev dependencies in our `package.json`, we can

now use that in our test.

Terminal

```
$ npm install --save-dev jest-emotion
```

Let's go ahead and add an `import {createSerializer} from 'jest-emotion'`. We also need to `import * as emotion from 'emotion'`.

```
import {createSerializer} from 'jest-emotion'
import * as emotion from 'emotion'
```

Then, we'll use the Jest API for adding a snapshot serializer with `expect.addSnapshotSerializer(createSerializer(emotion))`.

tests/calculator-display.js

```
expect.addSnapshotSerializer(createSerializer(em
otion))
```

If I run my test again, I'm going to get a snapshot failure, because that class was replaced with `.emotion-0`. I have my CSS above, which is exactly what I want.

Terminal

```
$ npm t
FAIL  src/shared/__tests__/calculator-display.js
  ● mounts

    expect(value).toMatchSnapshot()

    Received value does not match stored
snapshot "mounts 1".

    - Snapshot
    + Received

    @@ -1,7 +1,7 @@
      .emotion-0 {
```

Now instead of a cryptic `className`, I'm getting a much more helpful `className` and the CSS that would be applied. I'll go ahead and run `npm t -- -u` to update the snapshot, and I can look at that snapshot. It's a lot more helpful to me. As I make changes to the CSS right here I can run my test again with `npm t`, and the snapshot failures will be a lot easier to understand what's going on.

Terminal

```
$ npm t
...
Snapshot Summary
 › 1 snapshot failed from 1 test suite. Inspect
your code changes or run `npmtest -- -u` to
update them.

Test Suites: 1 failed, 2 passed, 3 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:   1 failed, 1 total
Time:        1.371s
...
```

There are bunch of different snapshot serializers available on npm, `jest-emotion` is one of them. Many snapshot serializer packages actually expose the serializer itself rather than requiring you to create it in code, and for snapshot serializers like that, you can actually edit your Jest configuration and add a `snapshotSerializers` property with an array of path to packages.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleNameMapper: {
    '\\.module\\.css
: 'identity-obj-proxy',
    '\\.css
: require.resolve('./test/style-mock.js'),
  },
  snapshotSerializers: []
}
```

For example, there is a package called `jest-serializer-path` that removes absolute paths and normalized paths across all platforms in your Jest snapshots. It can take an absolute path to your project and replace it with project root.

If you were to install that, then we can use that in our `snapshotSerializers` configuration with `jest-serializer-path`. That would apply that snapshot serializer to all of our tests in our project.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleNameMapper: {
    '\\.module\\.css
: 'identity-obj-proxy',
    '\\.css
: require.resolve('./test/style-mock.js'),
  },
  snapshotSerializers: []
}
```

In our case, we're using `jest-emotion`. We're going to import that `jest-emotion`, `createserializer` function, and `emotion` here. We'll call `expect.addSnapshotSerializer(createSerializer(emotion))` .

tests/calculator-display.js

```
import {createSerializer} from 'jest-emotion'
import * as emotion from 'emotion'
import CalculatorDisplay from '../calculator-
display'

expect.addSnapshotSerializer(createSerializer(em
otion))

test('mounts', () => {
  const {container} = render(<CalculatorDisplay
value="0" />)
  expect(container.firstChild).toMatchSnapshot()
})
```

This applies only to this one test. Doing that gives us the CSS output in our snapshot with the cleaner `className` that's much easier to follow, because we're using emotion.

Handle Dynamic Imports using Babel with Jest

Let's go ahead and add a test for this `calculator.js` file. We'll add a folder called `__tests__`, and inside of there, `calculator.js`, right next to the calculator file that we're testing. Then I'm going to just paste in some code to `import {render} from 'react-testing-library'` and that `Calculator` component.

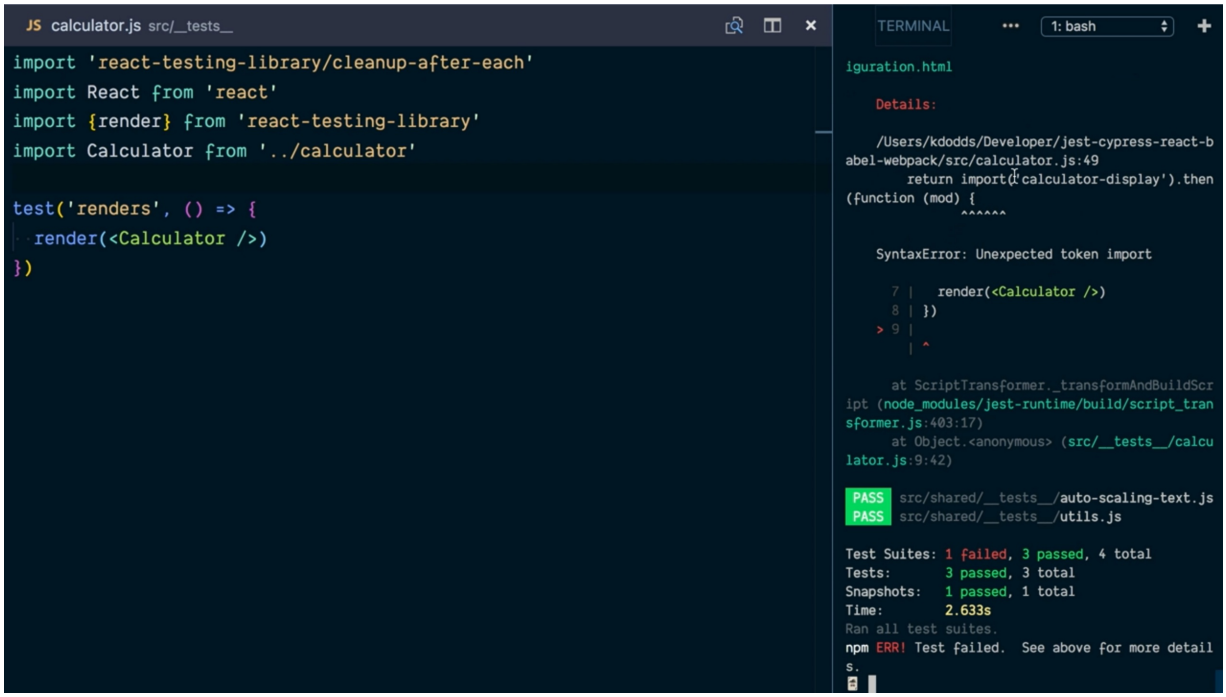
tests/calculator.js

```
import 'react-testing-library/cleanup-after-
each'
import React from 'react'
import {render} from 'react-testing-library'
import Calculator from '../calculator'

test('renders', () => {
  render(<Calculator />)
})
```

Let's go and save this, and we'll run our tests. Now, we're getting a syntax error, unexpected token import, right here.

Unexpected Token Import Error



In our Babel configuration, we're instructing Babel to transpile imports to `commonjs`. We see right here.

.babelrc.js

```
['@babel/preset-env', {modules: isTest ?
'commonjs' : false}],
```

`modules` should be compiled to `commonjs` during the tests. However, that's just for static modules. Here, we're using a dynamic import that Webpack supports natively. That is failing our test because babel won't transpile this to `commonjs`.

src/calculator.js

```
const CalculatorDisplay = loadable({
  loader: () => import('calculator-
display').then(mod => mod.default),
  loading: () => <div style={{height:
120}}>Loading display...</div>,
})
```

We need to install a new package that will transpile this to `commonjs` so that our tests will work. We'll run `npm install --save-dev babel-plugin-dynamic-import-node`.

Now, let's go ahead and update our Babel configuration to use this plugin, but we only want to use it when we're running our tests. We want Webpack to take over in production when we're building our application. Let's add this right here, `isTest ? 'babel-plugin-dynamic-import-node'.`

.babelrc.js

```
  isTest ? 'babel-plugin-dynamic-import-node' :
  null,
```

Otherwise, `null`. With that, if we now run our tests, everything is passing.

In review, the reason that we have this problem is because we have a dynamic import, which is not supported in `Node`. To make this work, we had to install a babel plugin into our `devDependencies` called `babel-plugin-dynamic-import-node`. Then we configure Babel to include that during our test environment when the `process.env.NODE_ENV` is 'test', which `Jest` will set up for us automatically.

.babelrc.js

```
  const isTest = String(process.env.NODE_ENV) ===
  'test'
```

Then everything works just fine.

Setup an afterEach Test Hook for all tests with Jest setupTestFrameworkScriptFile

With our current test situation, we have a commonality between most of our tests. In each one of them, we're importing 'react-testing-library/cleanup-after-each' in our `__tests__/calculator.js`, and `__tests__/auto-scaling-text.js`, and `__tests__/calculator-display.js`.

__tests__ | calculator.js, auto-scaling-text.js, calculator-display.js

```
import 'react-testing-library/cleanup-after-
each'
```

In addition, as our code base grows, we're going to be using more of `emotion`, so we'll probably want to add the `Snapshot Serializer` for all of our tests. It would be really nice if we could avoid this repetition and have that happen in one file that runs before any of the tests run. Let's go ahead and create that file. We'll make a new file here called `setup-tests.js`. Then I'll just take this right here and we'll paste it into here.

setup-tests.js

```
import 'react-testing-library/cleanup-after-
each'
```

I'll also copy out this and put it in here as well.

setup-tests.js

```
import 'react-testing-library/cleanup-after-
each'

import {createSerializer} from 'jest-emotion'
import * as emotion from 'emotion'

expect.addSnapshotSerializer(createSerializer(em
otion))
```

We can get rid of all that from `shared/__tests__/calculator-display.js` and here. We can also get rid it from here `shared/__tests__/auto-scaling-test.js` and here `src/__tests__/calculator-display.js`. Now we don't have that duplication. In each one of these, we could instead `import` all the way up to `test/setup-tests`. We could do that in every single one of the files. That's not a whole lot better. Let's go ahead and configure Jest to `import` these files automatically for us without us having to worry about importing the `setup-tests.js` file. In our Jest configuration, there are actually two ways that you can configure Jest to import a particular file before running the test and that is, `// before Jest is loaded` and `// after Jest is loaded`.

jest.config.js

```
// before Jest is loaded
// after Jest is loaded
```

The before Jest is loaded is called `setupFiles`. That's an array of files that will run before Jest is loaded.

jest.config.js

```
// before Jest is loaded
setupFiles: [],
// after Jest is loaded
```

You can use `setupFiles` for anything that doesn't need Jest environment to be loaded. Things like `expect.addSnapshotSerializer` or `afterEach()` will not be able to go into the `setupFiles`. That's why we're going to be using `setupTestFrameworkScriptFile: ''`. This is a path to a file that we want to have loaded after the Jest framework has been loaded.

jest.config.js

```
setupTestFrameworkScriptFile: ''
```

In our example, `cleanup-after-each` is going to set up an `afterEach()` with Jest and this `expect.addSnapshotSerializer` is also a Jest-specific API. Let's go ahead and add this. We'll get rid of that `setupFiles`. We don't need that.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleNameMapper: {
    '\\.module\\.css
: 'identity-obj-proxy',
    '\\.css
: require.resolve('./test/style-mock.js'),
  },
  // after Jest is loaded
  setupTestFrameworkScriptFile:
}
```

We're going to use `require.resolve('.test/setup/test.js')`.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleNameMapper: {
    '\\.module\\.css
: 'identity-obj-proxy',
    '\\.css
: require.resolve('./test/style-mock.js'),
  },
  setupTestFrameworkScriptFile:
require.resolve('./test/setup-tests.js'),
}
```

With that now, we can run `npm t` to run our tests, and everything will pass just fine because we're running our `setup-tests` before every single one of our tests. Now our tests don't have to have that setup configuration inside of each one of them. It's all happening in our `setup-tests.js` because we have configured Jest with a `setupTestFrameworkFile` property to run the setup file for every one of our tests after Jest has been loaded.

Support custom module resolution with Jest moduleDirectories

The way our `CalculatorDisplay` is implemented, the `calculator-display` is imported. It's asynchronously loaded.

calculator.js

```
const CalculatorDisplay = loadable({
  loader: () => import('calculator-
display').then(mod => mod.default),
  loading: () => <div style={{height:
120}}>Loading display...</div>,
})
```

When we run our test here, in `src/__tests__`, if I pull out the `container`, and I'm going to pull out this `debug` function. We'll `debug(container)`.

src/tests

```
test('renders', () => {
    const {container, debug} =
render(<Calculator />)
    debug(container)
})
```

I pop open the terminal and run my tests.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS src/__tests__/calculator.js

 ...

    <div
        style='height: 120px;'
    >
        Loading display...
    </div>

 ...
```

I'm going to see that the `calculator-display` was not loaded and we're actually rendering the `Loading Display...` instead, which is actually what the user is going to see. Maybe that's what I want.

If I want to be able to skip over that for my testing purposes, then I can actually `import loadable from 'react-loadable'`. I'll turn this into an `async` test.

src/tests

```
import loadable from 'react-loadable'

test('renders', async () => {
    await loadable.preloadAll()
    const {container, debug} =
render(<Calculator />)
    debug(container)
}
```

I will `await loadable.preloadAll()` and then run this again so that I can get my `CalculatorDisplay`.

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 FAIL  src/__tests__/calculator.js
  ● renders

    Cannot find module 'calculator-display' from
'calculator.js'

       9 | // handle with Jest and many people
will want to know :).
      10 | const CalculatorDisplay = loadable({
    > 11 |   loader: () => import('calculator-
display').then(mod => mod.defaul
t),
         |           ^
      12 |   loading: () => <div style={{height:
120}}>Loading display...</div>,
      13 | })
```

```
      14 |

      at Resolver.resolvemodule
(node_modules/jest-resolve/build/index.js:
210:17)
      at src/calculator.js:11:17

 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/calculator-
display.js

Test Suites: 1 failed, 3 passed, 4 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        3.209s
Ran all test suites.
npm ERR! Test failed.  See above for more
details.
```

I'm getting a `cannot find module 'calculator-display'`. The reason for this is because here, in `calculator.js` we're importing `calculator-display` as if it were a node module, but it's not a node module. It actually lives in the `shared` directory as `calculator-display`. The way that it works in the app is we have our webpack configuration set to `resolve modules` to `node_modules`, just like node would in a regular commonJS environment.

```
resolve: {
    modules: ['node_modules',
path.join(__dirname, 'src'), 'shared']
}
```

01:15 We also have this helper here, `path.join(__dirname, 'src')`, to resolve node modules based off of the source so we can import things from that `src` directory, and then also `shared`.

Any of these files inside of our `src` directory, if they're inside of a shared, they can actually be treated as if they were in node modules, which is a really handy thing for a giant project.

However, that poses a problem for us in Jest because Jest doesn't consume this webpack configuration. It doesn't resolve the modules the way webpack is resolving them.

Let's see how we can solve this problem using Jest configuration. I'm going to copy this and go over to my `jest.config.js`. I'll say `moduleDirectories`, and I'll just paste in the exact same thing we had before.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleDirectories: ['node_modules',
path.join(__dirname, 'src'), 'shared'],
  moduleNameMapper: {
    '\\.module\\.css
: 'identity-obj-proxy',
    '\\.css
: require.resolve('./test/style-mock.js'),
  },
  setupTestFrameworkScriptFile:
require.resolve('./test/setup-tests.js'),
}
```

Now, I need to pull in `path`. We'll say `const path = require('path')`.

```
const path = require('path')
```

Now, if I pull up my tests and I run the tests again, everything runs perfectly.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest
 ...
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        3.209s
Ran all test suites.
```

I can scroll up here. I see that my `CalculatorDisplay` was, in fact, rendered.

```
<div
    class="css-xeqdt1"
    >
        <div
            class="autoScalingText"
            style="trasnform: scale(1,1);"
            >
            0
            </div>
        </div>
```

In review, this was pretty simple. I just checkout my `webpack.config.js`, see the `resolve` modules here. I copy that into my `jest.config.js` to have a `modulesDirectory`.

Now, I can require anything from within a `shared` directory anywhere in the structure of my project source, and it will be treated as if it were in `node_module`'s directory, both from my webpack configuration as well as my tests with `Jest`.

Support a test utilities file with Jest moduleDirectories

Someone's made a change to our app that's really cool. It allows us to use this `ThemeProvider` and choose between our `dark` and a `light` theme for our UI.

app.js

```
import * as themes from './themes'

class App extends React.Component {
  state = {theme: 'dark'}
  handleThemeChange = ({target: {value}}) =>
this.setState({theme: value})
  render() {
    return (
      <ThemeProvider theme=
{themes[this.state.theme]}>
        <React.Fragment>

      ...
    )
  }
}
```

Those themes will determine what color are `displayTextColor` and `displayBckgroundColor` should be, so that our UI can change based off of the user's preferences.

themes.js

```
export const dark = {
  displayTextColor: 'white',
  displayBackgroundColor: '#1c191c',
}

export const light = {
  displayTextColor: '#1c191c',
  displayBackgroundColor: 'white',
}
```

This is using the `ThemeProvider` from `emotion-theming`, and that provides those theme values to anything that's using `emotion`.

app.js

```
import {ThemeProvider} from 'emotion-theming'
```

Right now, we're only using that in this `CaclulatorDisplay` component that we're testing with this `calculator-display` file.

tests/calculator-display.js

```
import React from 'react'
import {render} from 'calculator-test-utils'
import CalculatorDisplay from '../calculator-
display'

test('mounts', () => {
  const {container} = render(<CalculatorDisplay
value="0" />)
  expect(container.firstChild).toMatchSnapshot()
})
```

Then, we open up the `CaclulatorDisplay` implementation, and we can see this `DisplayContainer` that's using those theme values to set the `color` and the `background`.

calculator-display.js

```
const DisplayContainer = styled.div(({theme}) =>
({
  color: theme.displayTextColor,
  background: theme.displayBackgroundColor,
  lineHeight: '130px',
  fontSize: '6em',
  flex: '1',
}))
```

In our test, we're using `toMatchSnapshot`, because we've configured our `serializer` for emotion.

tests/calculator-display.js

```
test('mounts', () => {
  const {container} = render(<CalculatorDisplay
value="0" />)
  expect(container.firstChild).toMatchSnapshot()
})
```

setup-tests.js

```
import 'react-testing-library/cleanup-after-
each'

// add jest-emotion serializer
import {createSerializer} from 'jest-emotion'
import * as emotion from 'emotion'

expect.addSnapshotSerializer(createSerializer(em
otion))
```

We should see a `snapshot` here that has our `CSS` right in there.

snapshots/calculator-display.js.snap

```
exports[`mounts 1`] = `
.emotion-0 {
  color: white;
  background: #1c191c;
  line-height: 130px;
  font-size: 6em;
  -webkit-flex: 1;
  -ms-flex: 1;
  flex: 1;
}

...
`
```

With these changes though, let's see how that impacts our output. We'll run `npm t` to run the test.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS  src/__tests__/calculator.js
 FAIL  src/shared/__tests__/calculator-
display.js
    ● mounts

    expect(value).toMatchSnapshot()

    Recieved value does not match stored
snapshot, "mounts 1".

    - Snapshot
    + Recieved

    ßß -1,8 +1,6 ßß
      .emotion-0 {
          color: white;
      -   background: #1c191c;
      ...
      }

Test Suites: 1 failed, 3 passed, 4 total
Tests:       1 failed, 3 passed, 4 total
Snapshots:   1 failed, 1 total
Time:        3.267s
Ran all test suites.
npm ERR! Test failed. See above for more
details.
```

We're going to get a fail snapshot, because now the `color` in `background` are not supplied. If we look at that `CalculatorDisplay` again, we're going to see the `color` in `background` are derived from the theme.

tests/calculator-display.js

```
test('mounts', () => {
  const {container} = render(<CalculatorDisplay
value="0" />)
  expect(container.firstChild).toMatchSnapshot()
})
```

In our application, we're providing the theme using the `ThemeProvider`. In our test, we're not providing the `theme` at all, because we're not rendering this inside of a `ThemeProvider`.

To solve, this is actually fairly simple. We can import `ThemeProvider` from `emotion-theming`, and we can import dark which is the default from back, back, back, back, themes. We can wrap this in a `ThemeProvider` with the theme as dark. We'll take that `calculator-display` and put it inside that `ThemeProvider`.

```
import CalculatorDisplay from '../calculator-
display'
import {dark} from '../../themes'

test('mounts', ( => {
  const {container} = render(
    <ThemeProvider theme={dark}>
      <CalculatorDisplay value="0" />
    </ThemeProvider>
  )
  expect(container.firstChild).toMatchSnapshot
})
```

Now if you run this again with `npm t` to run our test, everything is passing that `snapshot` matches.

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        3.267s
Ran all test suites.
```

This same thing would happen if we're using the Redux Provider, or React Router Provider, or some internationalization provider. Anytime we're rendering a component that needs access to a provider, we have to render that component inside that provider, when we're rendering it for a test.

```
render(
  <ThemeProvider theme={dark}>
      <CalculatorDisplay value="0" />
  </ThemeProvider>
)
```

The thing is that lots of the time, we'll just refactor a component to start using `emotion` and we experience some of these problems.

It would be really nice, if we don't have to even think about the `ThemeProvider` or the React Router Provider to be able to use some of those features that we're going to be using in our application.

It will be nice, if this `render` method could just render all the providers that we're really care about automatically for us. Let's work a way there. I'm going to make a function here called `renderWithProviders`. This is going to accept a `ui` and `options`. I'm just going to copy this and paste it right here. Instead of the `CalculatorDisplay` hard coded here, we can now just `render` the `UI` as a child, and we'll pass `options` as the second argument here. Instead of render, we'll use `renderWithProviders` or get rid of the `ThemeProvider`.

```
function renderWithProviders(ui, options) {
  render(<ThemeProvider theme={dark}>{ui}
</ThemeProvider>, options)
}

test('mounts', ( => {
  const {container} = renderWithProviders(
  expect(container.firstChild).toMatchSnapshot
})
```

Let's go ahead and `return` that value, so that we can get access to the `{container}` and all the other `utilities` that `render` is going to give to us.

```
function renderWithProviders(ui, options) {
  return render(<ThemeProvider theme={dark}>{ui}
</ThemeProvider>, options)
}
```

Now if you run our test again, we should have this continuing to pass.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.431s
Ran all test suites.
```

Now, we have a useful utility that we could use throughout our application. Let's make this accessible throughout our application by copying this over into a new file in our `test` directory called `calculator-test-utils.js`. In here, we'll just paste that. We'll export `renderWithProviders` and we can get rid of the `CalculatorDisplay` and fix this import up.

test/calculator-test-utils.js

```
import React from 'react'
import {render} from 'react-testing-library'
import {ThemeProvider} from 'emotion-theming'
import {dark} from '../src/themes'

function renderWithProviders(ui, options) {
  return render(<ThemeProvider theme={dark}>{ui}
</ThemeProvider>, options)
}

export {renderWithProviders}
```

We can go to our `calculator-display-test` here. We'll import `renderWithProviders` from back couple directories test `calculator-test-utils`. Get rid of this. We don't need that or those anymore.

tests/calculator-display.js

```
import React from 'react'
import CalculatorDisplay from '../calculator-
display'
import {renderWithProviders} from
`../../../__test__/calculator-test-utils`

test('mounts', () => {
  const {container} =
renderWithProviders(<CalculatorDisplay value="0"
/>)
  expect(container.firstChild).toMatchSnapshot()
})
```

Then, we can run our test again, and our tests continue to pass.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest

 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.431s
Ran all test suites.
```

This is great. Now, we can use this throughout our app, but as we store refactoring things and moving things around, these dot dots, `../../../../`, get really, really annoying, really quickly. Let's see if we can make `Jest` understand that we want to treat this like a regular `node module`.

tests/calculator-display.js

```
import {renderWithProviders} from `calculator-
test-utils`
```

I'm going to pop up in `jest.config.js`. We actually did something similar to this with the `module` directories, when we wanted to mimic the `webpack.config.js` configuration for result modules.

We can do something special for our test, so that anything inside the test directory will be accessible, as if it is a node module. Let's go ahead and do that. We're going to into the same thing we did with our source here, except we're going to do it for the test directory.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleDirectories: [
    'node_modules',
    path.join(__dirname, 'src'),
    'shared',
    path.join(__dirname, 'test'),
  ],
  moduleNameMapper: {
    '\\.module\\.css
: 'identity-obj-proxy',
    '\\.css
: require.resolve('./test/style-mock.js'),
  },
  setupTestFrameworkScriptFile:
require.resolve('./test/setup-tests.js'),
}
```

Now, if we go to our test here, when we import the `calculator-test-utils`, it's going to resolve from the test directory. Now, I can pull this out.

tests/calculator-display.js

```
import {renderWithProviders} from `calculator-test-utils`
```

Run `npm t` to run our test again and all our test continue to pass.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-cypress-react-babel-webpack
> jest

 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/calculator-display.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.431s
Ran all test suites.
```

Now, we can use this anywhere in our test and this `renderWithProviders` will be accessible.

tests/calculator-display.js

```
import {renderWithProviders} from `calculator-test-utils`
```

Like I said, that would be really nice, if we don't have to worry about whether or not this `CalculatorDisplay` using anything from the providers.

```
  test('mounts', () => {
  const {container} = render(<CalculatorDisplay value="0" />)
  expect(container.firstChild).toMatchSnapshot()
})
```

If we could just, maybe have a regular `render` from here, `calculator-test-utils`, and this is the one that we use everywhere in our code base.

```
import {render} from `calculator-test-utils`
```

Let's go and do that. We'll just rename this as `render`, and just forget measure we can `export * from react-testing-library`. Now, we can use all of the utilities that `react-testing-library` provides. We're overriding the `render` to be this `renderWithProviders`.

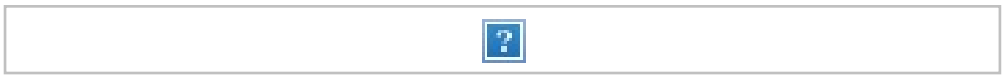
calculator-test-utils.js

```
export * from 'react-testing-library'
export {renderWithProviders as render}
```

Now anywhere in our application, that were using `react-testing-library`, we can actually use `calculator-test-utils`. We can do that here in our calculator test here as well. It will work regard this of whether we add emotion, or Redux, or React Router.

We can always just update the `render` method here and the `calculator-test-utils.js`. That will automatically work for everywhere else that's using the `render` method we're exporting here.

There's one last thing that I want to show you here and that is this red underline.



We're getting a problem with `ESLint` not being able to resolve in this path. It'll be really nice, if we could have `ESLint` to resolve this, so that I could touch typos like this, `'calculator-test-utilss'`

This is working, because I'm using `eslint-plugin-import` to check my imports. What we can do is, we can add a new resolver for `eslint-plugin-import`. `npm install` as a dev dependency, `eslint-import-resolver-jest`.

Terminal

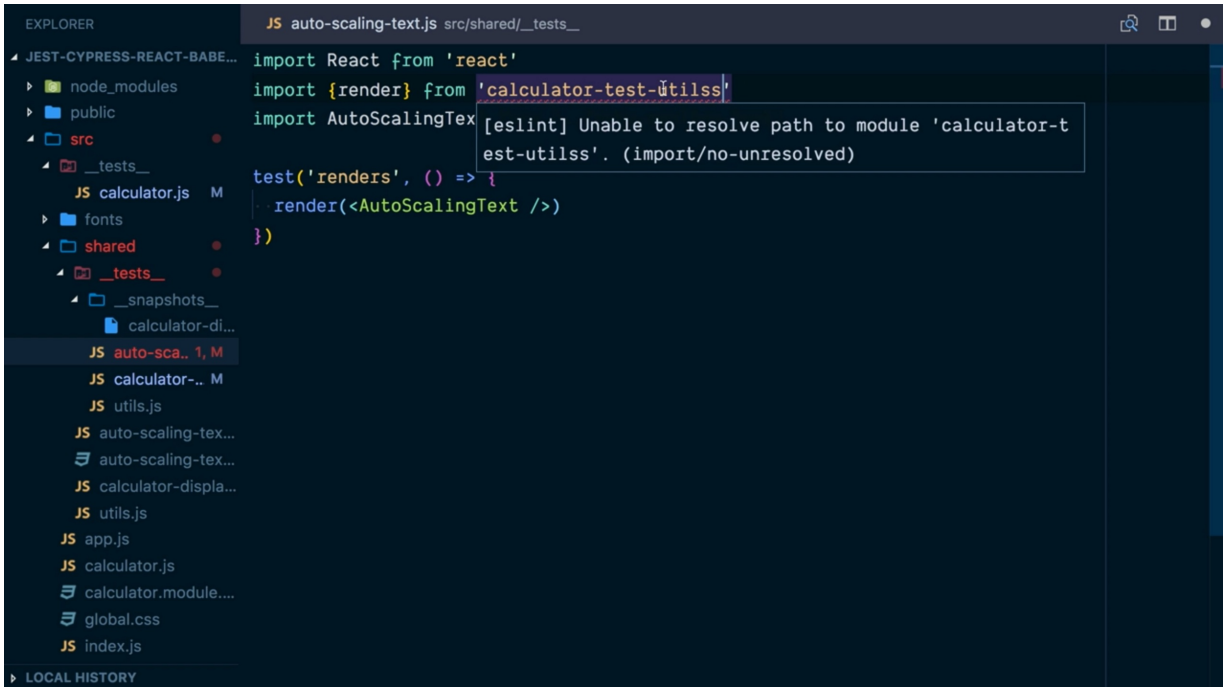
```
$ npm install --save-dev eslint-import-resolver-
jest
```

Now with that installed and saved into our dev dependencies, we can jumped on here to our `ESLint` configuration and add overrides. For files that match `'**/__test__`. Anything in our `tests` directory will have this `settings` be overridden for `import/resolver`. We're going to use the `jest` resolver. We'll specify the `jestConfigFile` to be path that join `__dirname` and `jestconfig.js`.

```
module.exports = {
  extends: [
    'kentcdodds',
    'kentcdodds/import',
    'kentcdodds/webpack',
    'kentcdodds/jest',
    'kentcdodds/react',
  ],
  overrides: [
    {
      files: ['**/__tests__/**'],
      settings: {
        'import/resolver': {
          jest: {
            jestConfigFile: path.join(__dirname,
'./jest.config.js'),
          },
        },
      },
    },
  ],
}
```

We'll require `path` from `path`. With that, `eslint-plugin-import` can use this `Jest` configuration, pull the `module directories`, and use that in its resolution of these packages. For your look at our `calculator-test` here again, we're going to get that this `calculator-test-utils` is correct.

If I typo, then I'm going to get ESLint warning me about that.



In our view, the problem that we're trying to solve here is, we want to render this `CalculatorDisplay` with the `themeProvider` that it requires without having to worry about that `themeProvider`.

We're going to create a `render` method in this `calculator-test-utils`. We can use that and it will provide all the providers that we need for our application. In our `jest.config.js` configuration, we make that possible by using the `moduleDirectories` and providing a path for this `tests` directory.

Any files inside of this `tests` directory can be imported as if there `node_modules` and we can avoid the `../` all over a test base. With this in place now, we can use the `themeprovider`.

We could also use the unstated provider, or the react-router-provider, or the ReduxProvider and have them all wrapped inside of this, so we could say router, and we could say `ReduxProvider` and wrap that as well, just like you would see at the root of your application where you're adding all these providers.

calculator-test-utils.js

```
function renderWithProviders(ui, options) {
  render(
    <Router>
    <ReduxProvider>
      <ThemeProvider theme={dark}>{ui}
</ThemeProvider> options
    </ReduxProvider>
  </Router>,
  )
}
```

08:23 You could also add `options` for this. If you could say `store`, it calls store or `options.store`. You could `import` the default `store` from the application, or somebody could provide their own copy of the store. There are lots of options for you, when you provide this customer `render` method that you can use throughout your application.

```
function renderWithProviders(ui, options) {
  render(
    <Router>
      <ReduxProvider store ={store ||
options.store} >
        <ThemeProvider theme={dark}>{ui}
</ThemeProvider>
      </ReduxProvider>
    </Router>,
    options,
  )
}
```

Step through Code in Jest using the Node.js Debugger and Chrome DevTools

Let's say we're having a little bit of trouble with this `getScale` method, and we want to debug it. One thing you could do is you could `console.log` here to make sure that the code is getting here, and then you could `console.log` maybe `in if statement`.

auto-scaling-text.js

```
class AutoScalingText extends React.Component {
    static propTypes = {
        children: PropTypes.node,
    }
    node = React.createRef()

    getScale(){
        const node = this.node.current
        console.log('here')
        if (!node) {
             console.log('in if statement')
            return 1
        }
    }

    ...
}
```

Then you can open up your tests and run `npm t`, and then `auto-scaling-text` to run just that test.

Terminal

```
$ npm tauto-scaling-text

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest "auto-scaling-text"

 PASS  src/shared/__tests__/auto-scaling-text.js
  ✓ renders (35ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.135s
Ran all test suites matching /auto-scaling-
text/i.
```

We'll see OK, cool. This is a nice feature that Jest provides, as it tell you where these `console.log` happened in the source code. Then we can see the `console.log` here. It is getting inside the `in if statement`.

Let's figure out why that's happening. This sort of debugging with `console.log` is not really super awesome. It'd be really nice if we could have the same kind of experience debugging our tests that we have when debugging our application in the browser.

00:45 We can use Node's built-in `expect` flag and Chrome's integration with that flag to make this a really great experience for debugging our tests. Let's get rid of these `console.log` statements. What I really want is to have a `debugger` statement here, so that my `debugger` can break on that line.

auto-scaling-text.js

```
getScale(){
    const node = this.node.current
    debugger
    if (!node) {
        console.log('in if statement')
        return 1
    }
}
```

If we look at our `package.json` for where we're running these tests, we're going to see that we have a `test` that's pointing to `jest`. For us to be able to use the Node `debugger`, we're going to write a `test:debug` script right here.

package.json

```
{
  "name": "calculator",
  "version": "1.0.0",
  "description": "See how to configure Jest and
Cypress with React, Babel, and Webpack",
  "main": "index.js",
  "scripts": {
    "test": "jest",
    "test:debug": "",
    "dev": "webpack-serve",
    "build": "webpack --mode=production",
    "postbuild": "cp ./public/index.html
./dist/index.html",
    "start": "serve --no-clipboard --listen 8080
dist",
    "lint": "eslint .",
    "format": "prettier \"**/*.js\" --write",
    "validate": "npm run lint && npm run test &&
npm run build",
    "setup": "npm run setup && npm run validate"
  },

  ...

}
```

This, we want to actually use `jest`, but we need to use the `--inspect-brk` flag from Node. What we're going to do is we'll say `node --inspect-brk`. Then we need to point to the `jest` binary, but we can't simply put `jest` here, because that will be interpreted as an argument to `Node`.

```
"scripts": {
    "test": "jest",
    "test:debug": "node --inspect-brk jest",
    "dev": "webpack-serve",
    "build": "webpack --mode=production",
    "postbuild": "cp ./public/index.html
./dist/index.html",
    "start": "serve --no-clipboard --listen 8080
dist",
    "lint": "eslint .",
    "format": "prettier \"**/*.js\" --write",
    "validate": "npm run lint && npm run test &&
npm run build",
    "setup": "npm run setup && npm run validate"
  },
```

What we're going to do is I'll pop open the `node_modules` here. We'll look down here for `jest`, and inside the `bin` directory, there's that Jest file. That's what's actually being run when we run `jest`. I'll say `node_modules/jest/bin/jest.js`.

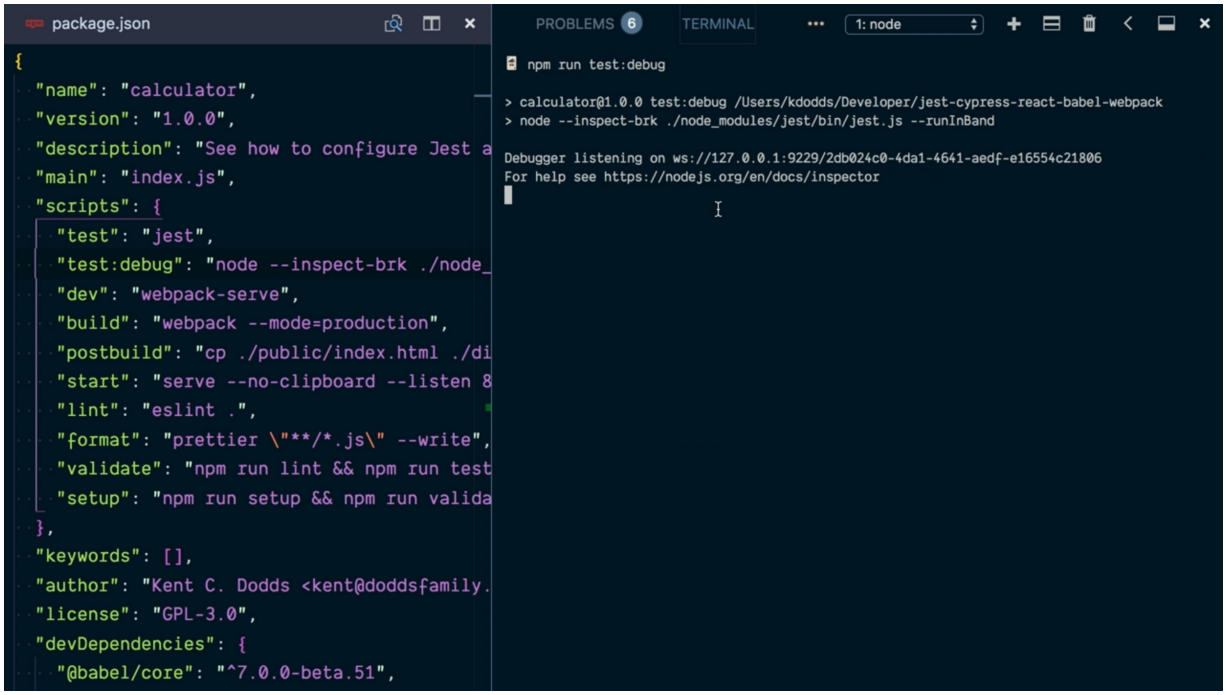
That's the the file I want Node to run, with this `--inspect-brk` flag. Jest, by default, actually runs our tests in parallel, but that doesn't really help us a lot when we want to debug our tests. We want to have them `--runInBand`.

```
"scripts": {
    "test": "jest",
    "test:debug": "node --inspect-brk
./node_modules/jest/bin/jest.js --runInBand",
}
```

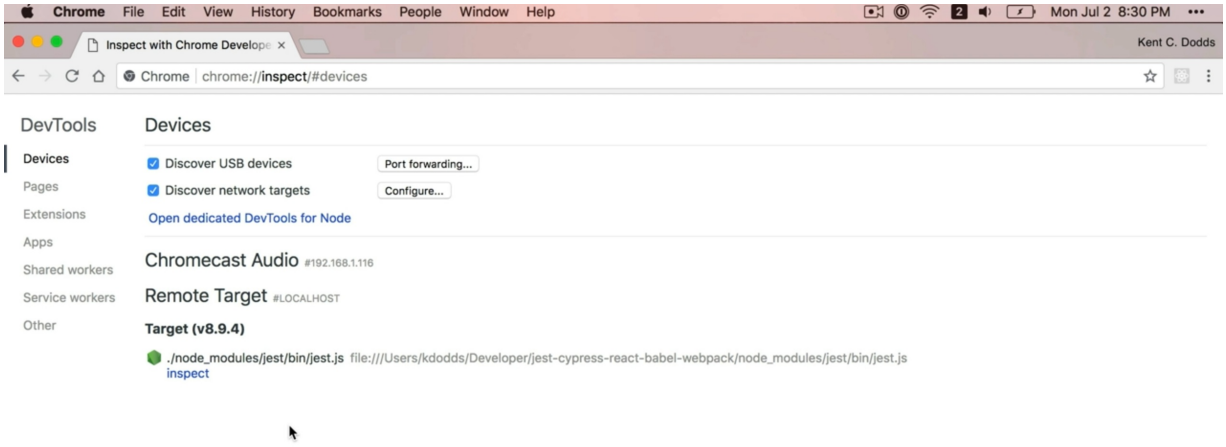
That's a flag that you can pass to the Jest binary. With that set up, now, I can run in my terminal `npm run test debug`, and I get this output, "Debugger listening on this port. For help, you can learn about the inspector."

Terminal

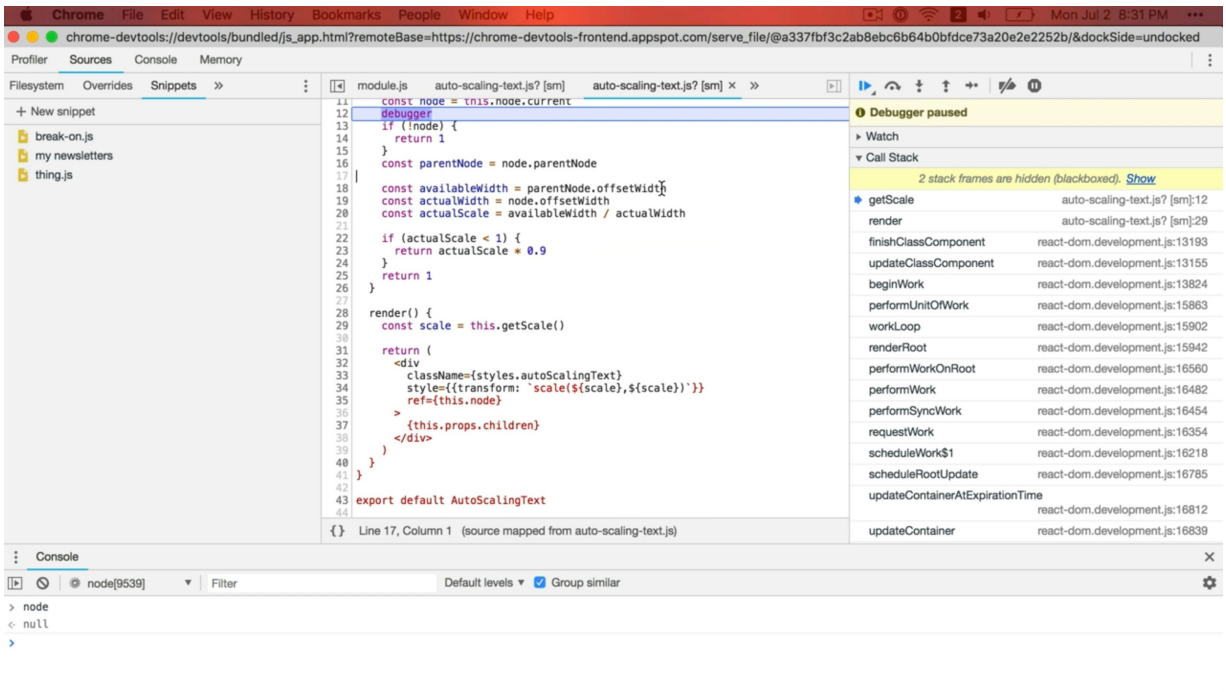
```
$ npm run test debug
```



There are a couple ways to open this up. In the Chrome browser, I can say `chrome://inspect`. That will open up this inspector here. We have this remote `Target`, and we have this as one of the things we can inspect. I'll click on `inspect`, and that pops open a very familiar debugger experience.



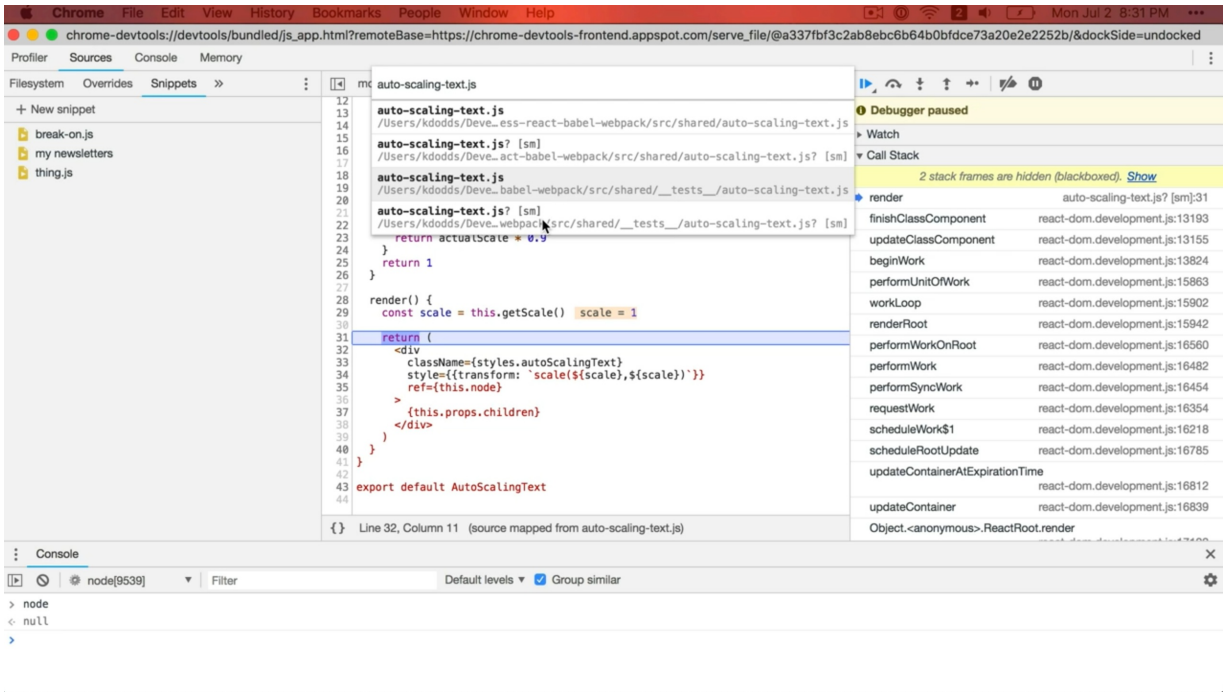
Now, if I hit play through, it will load all of the modules for me, and it breaks on that `debugger` statement. Now, I can `inspect` the world. I can say `node`, and I see, oh, `node` is null. That's because this hasn't actually rendered yet.



If I continue stepping through, I return `1`, and then my `scale` is `1`. That explains what's going on. You can debug through this way. It makes working with your tests a lot easier. You can also

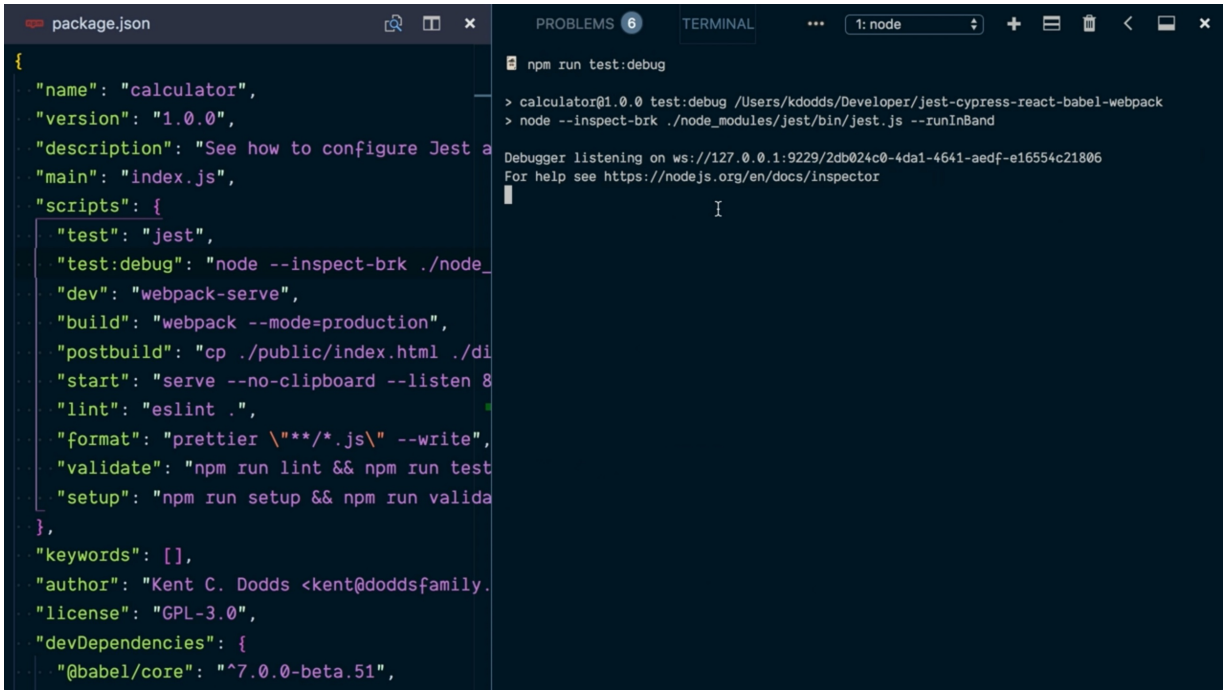
open the test itself, `autoscaling-text.js` inside of my test file.

Jest and Babel do a really great job with source maps. You can click on this `SM`, and that will show you your original code.

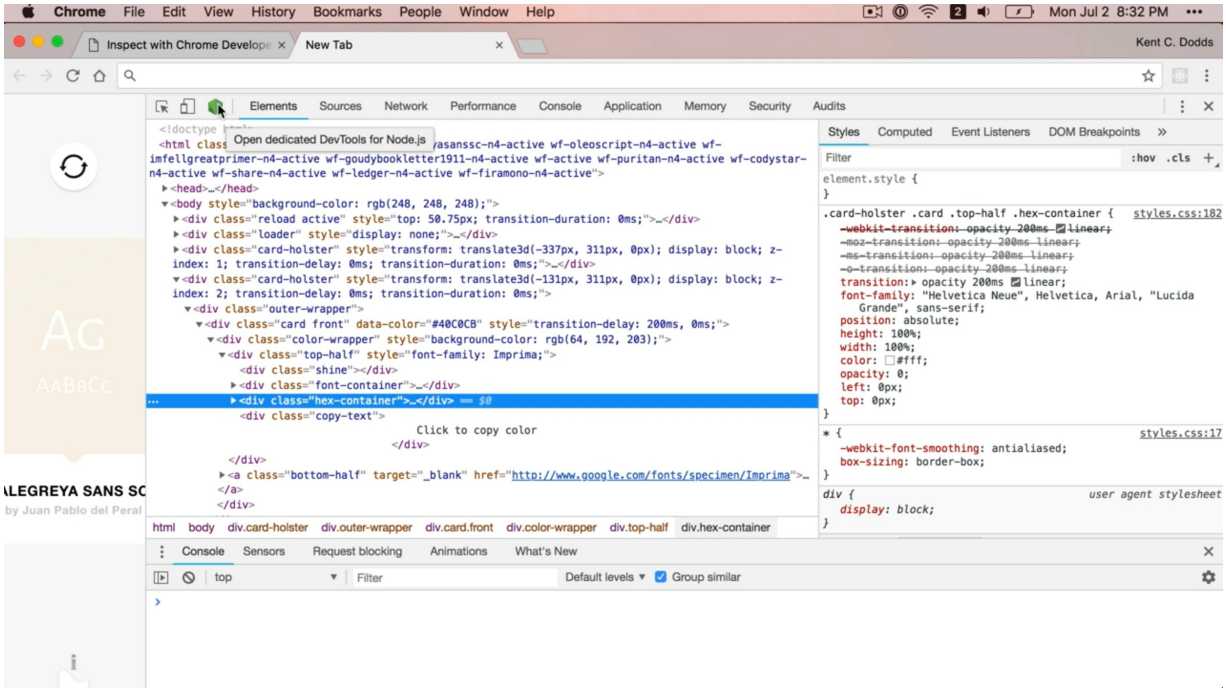


You can see the transpiled code with `auto-scaling-text.js`. This is the version that the browser is actually running. Another way that you can pull this up is, if we cancel this, and then run that again, we'll see that same output again.

```
$ npm run test debug
```



We'll see this in our `chrome://inspect`. If we open up a new tab, and then pop open the developer tools with inspect, then you'll see this nice little `node` icon that you can click on. That will also pop up the dev tools, which you can play through and start debugging.



In review, to make this work, all we needed to do is create this `test debug` script, where we run `node` with the `--inspect-brk` flag, and then point to the `node` file that we want to run. Specifically with `jest`, we run the `runInBand` flag, so that it will work with our debugging experience.

It doesn't try to run all of our tests in parallel, and instead allows us to run each one of these tests within the same process, so we can `debug` just that one process.

Configure Jest to report code coverage on project files

As we start writing test for our application, it might be nice for us to know how much of our application is being run when our tests are run. For example, we're rendering this `Calculator`, and so that's going to cover a bunch of the code that's inside of this `Calculator`, but maybe not all of it.

In addition, it's not going to be covering this `app.js` file that we might want to start adding test for. It would be really nice to be able to see what parts of code are being tested and what parts are not, so we can determine what part of our code base could use a little bit of our extra help with test.

To do with Jest, you can go to your `package.json`. In this test script, we can simply add a `--coverage`.

package.json

```
{
  "name": "calculator",
  "version": "1.0.0",
  "description": "See how to configure Jest and
Cypress with React, Babel, and Webpack",
  "main": "index.js",
  "scripts": {
    "test": "jest --coverage",
    "test:debug": "node --inspect-brk
./node_modules/jest/bin/jest.js --runInBand --
watch",
    ...
  },
  ...
}
```

Now if I run `npm t`, it's going to run all of my tests. It's going to output this report that it tells me a whole bunch of information about the coverage of my code base.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest --coverage

 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/__tests__/calculator.js
---------------------------|----------|---------
```

```
-|----------|----------|------
-------------|
File                         |  % Stmts | % Branch
|  % Funcs |  % Lines | Uncov
ered Line #s |
-----------------------------|----------|---------
-|----------|----------|------
-------------|
All files                    |    26.72 |    14.58
|    26.09 |    25.89 |
 src                         |    15.22 |      2.7
|       15 |    14.61 |
  calculator.js              |    13.33 |      2.7
|       15 |    12.64 |... 94,300,306,312 |
  themes.js                  |      100 |      100
|      100 |      100 |
 src/shared                  |    68.18 |    54.55
|      100 |    66.67 |
  auto-scaling-text.js       |    41.67 |       25
|      100 |    41.67 |... 18,19,21,22,24 |
  calculator-display.js      |      100 |       50
|      100 |      100 |          23 |
  utils.js                   |      100 |       80
|      100 |      100 |          11 |
 test                        |      100 |      100
|      100 |      100 |
  calculator-test-utils.js   |      100 |      100
|      100 |      100 |
  setup-tests.js             |      100 |      100
|      100 |      100 |
-----------------------------|----------|---------
-|----------|----------|-----------------|

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        7.081s
Ran all test suites.
```

In addition, it's also going to create this new folder that's a generated report of this coverage. The first thing that I'm going to is, I'm going to pop up in my `.gitignore` and add coverage to that, so that I don't commit this to source control, because that's the one thing that I really don't want to do is commit this generated code into my source control.

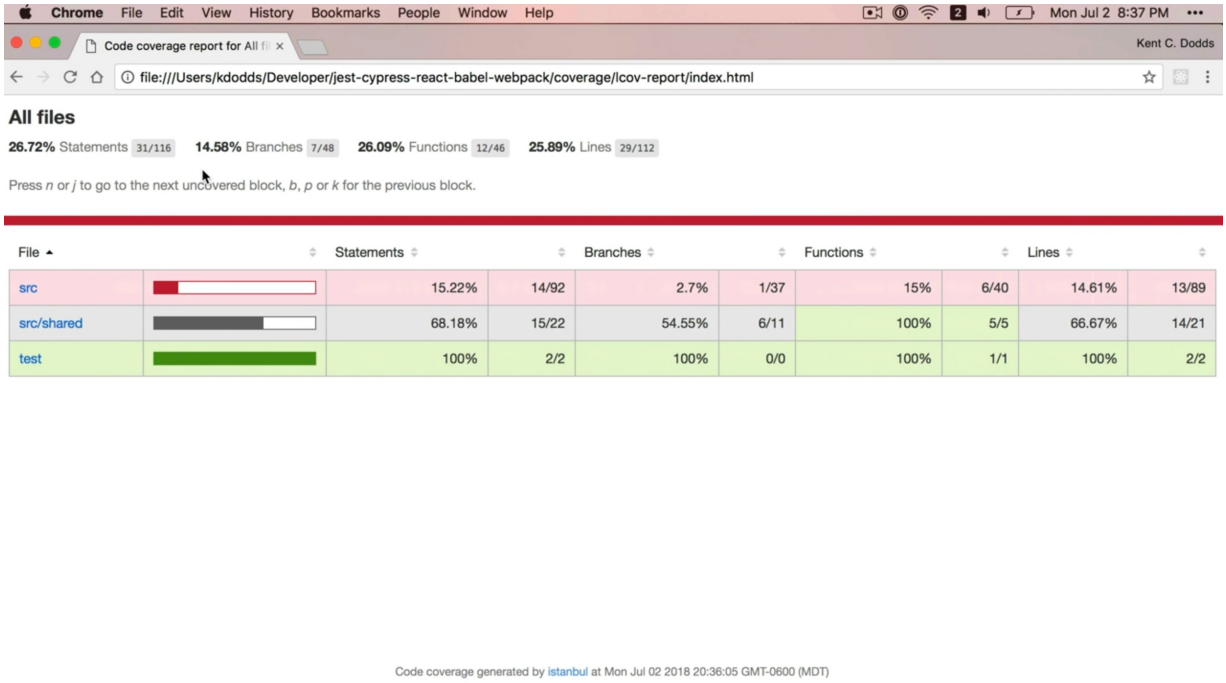
.gitignore

```
node_modules
dist
coverage
```

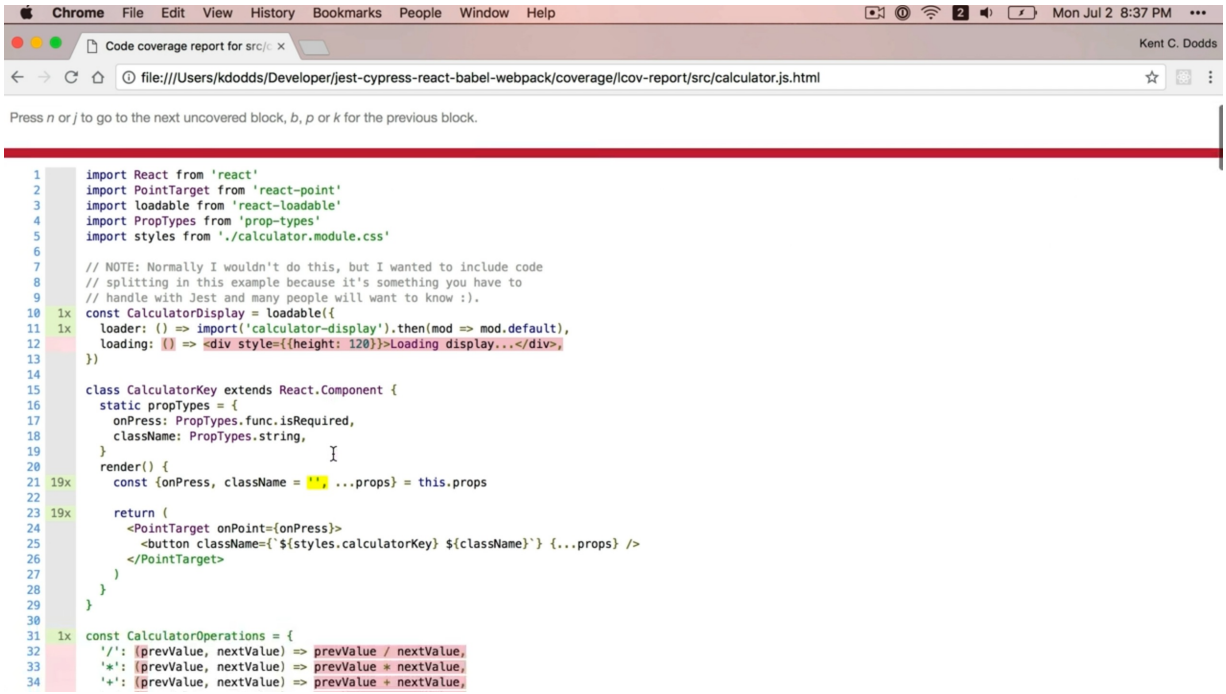
I would get merge conflicts in there on a regular basis. Now, let's go and take a look at this coverage report. We can actually open this `index.html` file in our browser. Here in the terminal, I'll run `open coverage/lcov-report/index.html`.

```
$ open coverage/lcov-report/index.html
```

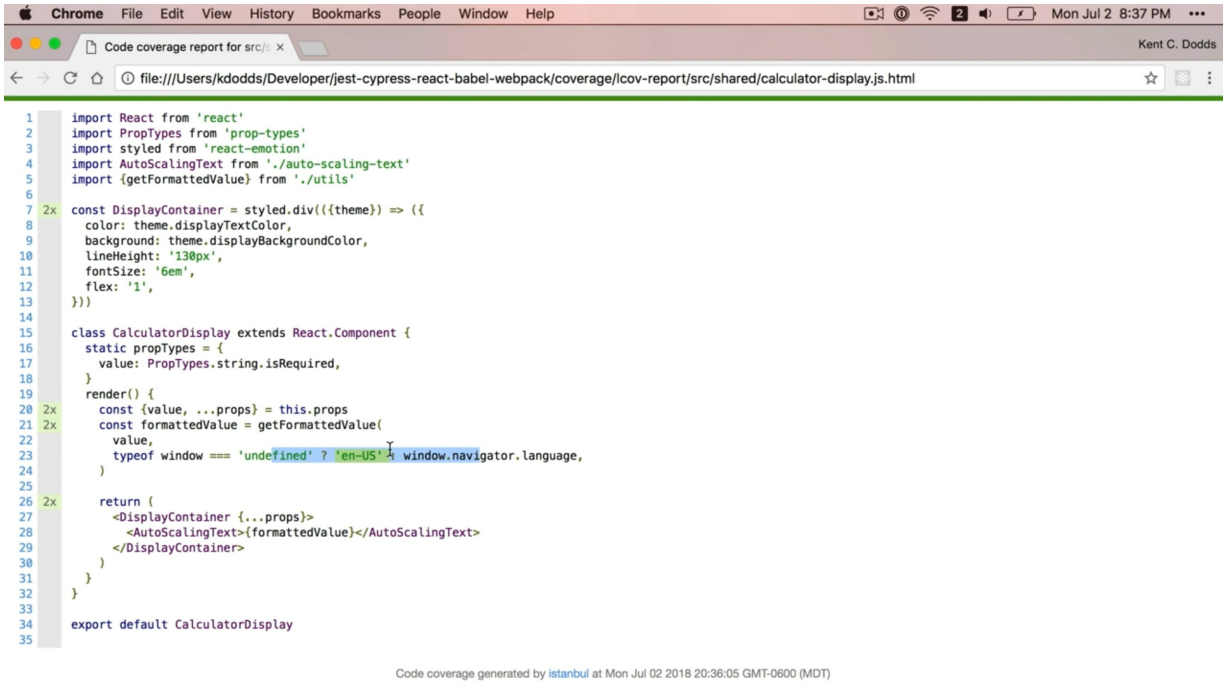
That'll pop right up in my browser where I can look at how we're doing with code coverage.



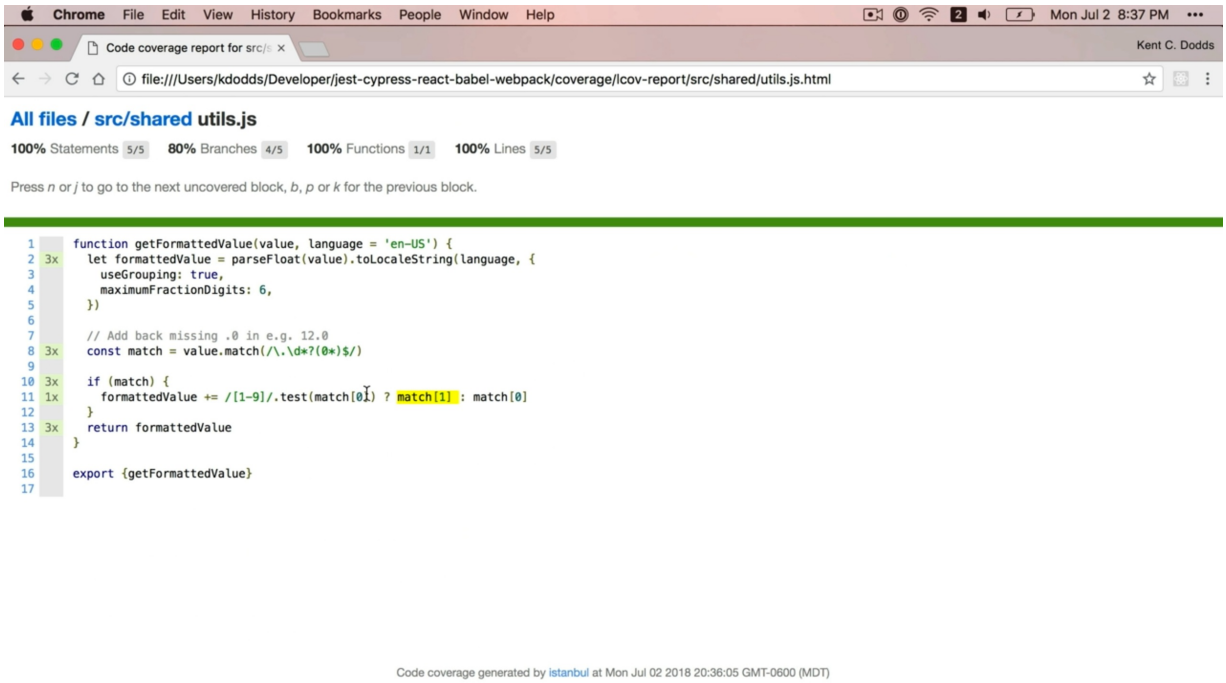
I'll look at this source directory, looks like we're doing just fine. In our `themes`, we're running this three times.



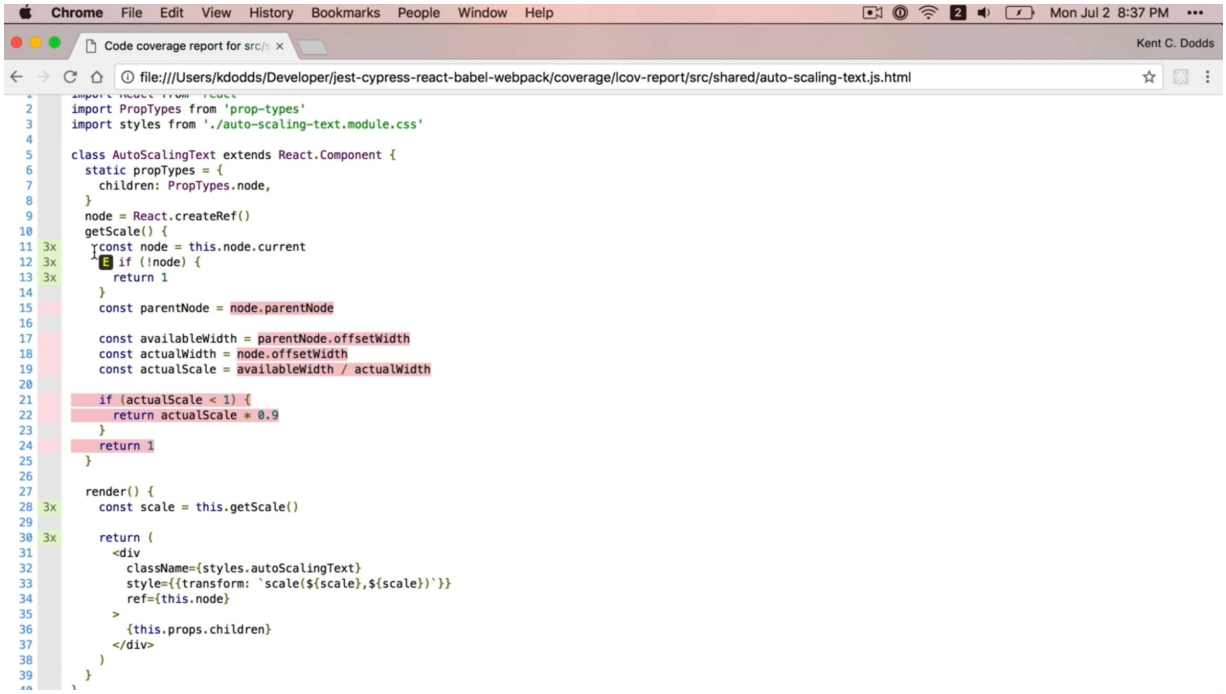
Then, I can look at the calculator file looks like we're not doing a super great job at the calculator. I can also look at this `src` shared, where we're doing great with the `CalculatorDisplay`, maybe missing something right there.



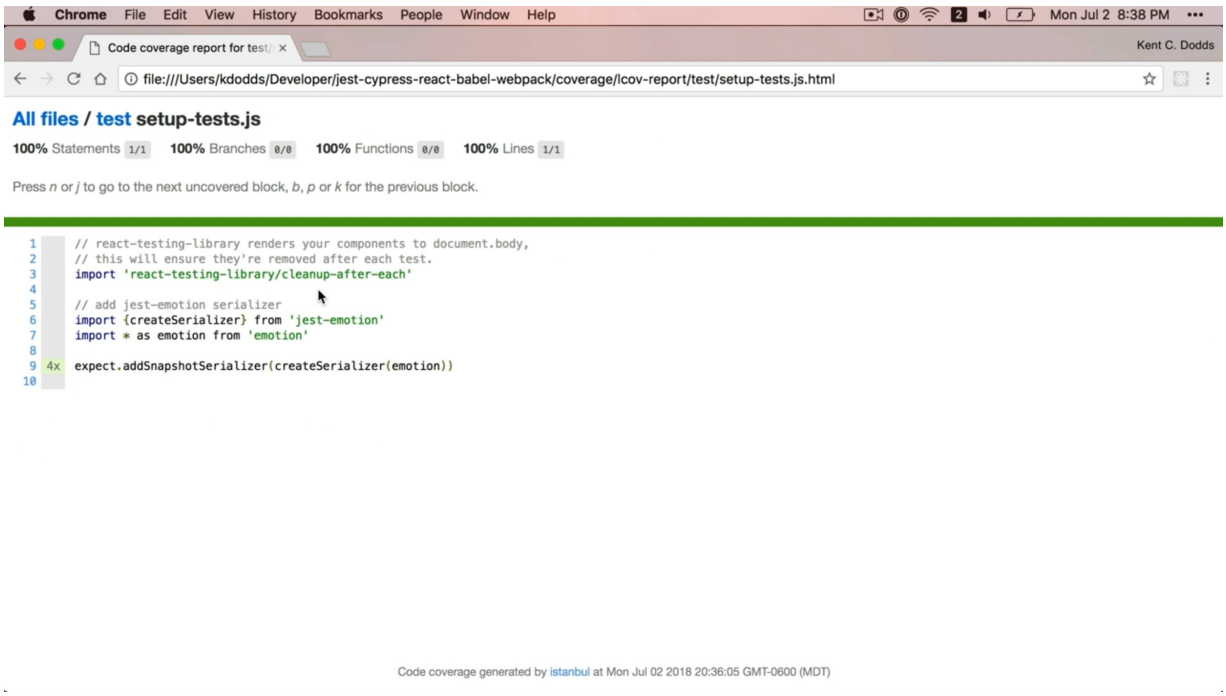
The `utils` were just missing one case right there.



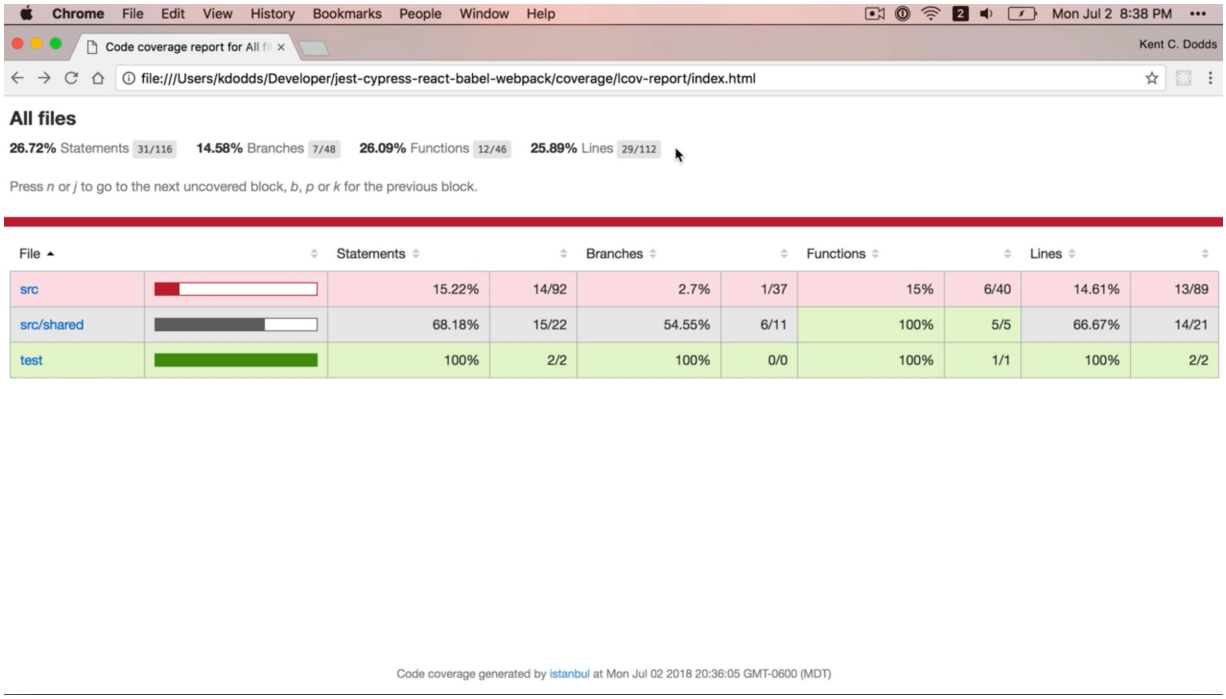
We could use a test or two to get these things up to 100 percent code coverage. `auto-scaling-text` is really missing some stuff. We need to add a couple of more tests here.



One thing that's a little concerning to me though is we have this `test` file, and this has our utilities that we're using and our setup environment in the module that we're using for our test.



Now, these are getting 100 percent code coverage and these aggregated numbers are including those files in our numbers.



That's not something I'm super excited about, because these files are uploading the overall coverage numbers that we have.

We don't get a really good sense of our coverage. Our test files should always be at 100 percent code coverage, otherwise we just remove those utilities. Let's look how we can configure Jest. It doesn't include these test files.

I'm going to go back in here into our `jest.config.js` configuration. I'm going to add a property called `collectCoverageFrom`. This is an array of strings and these strings are globes that should match the files that we want to collect coverage from.

The files we want to collect coverage from all match `**/src/**/*.js`.

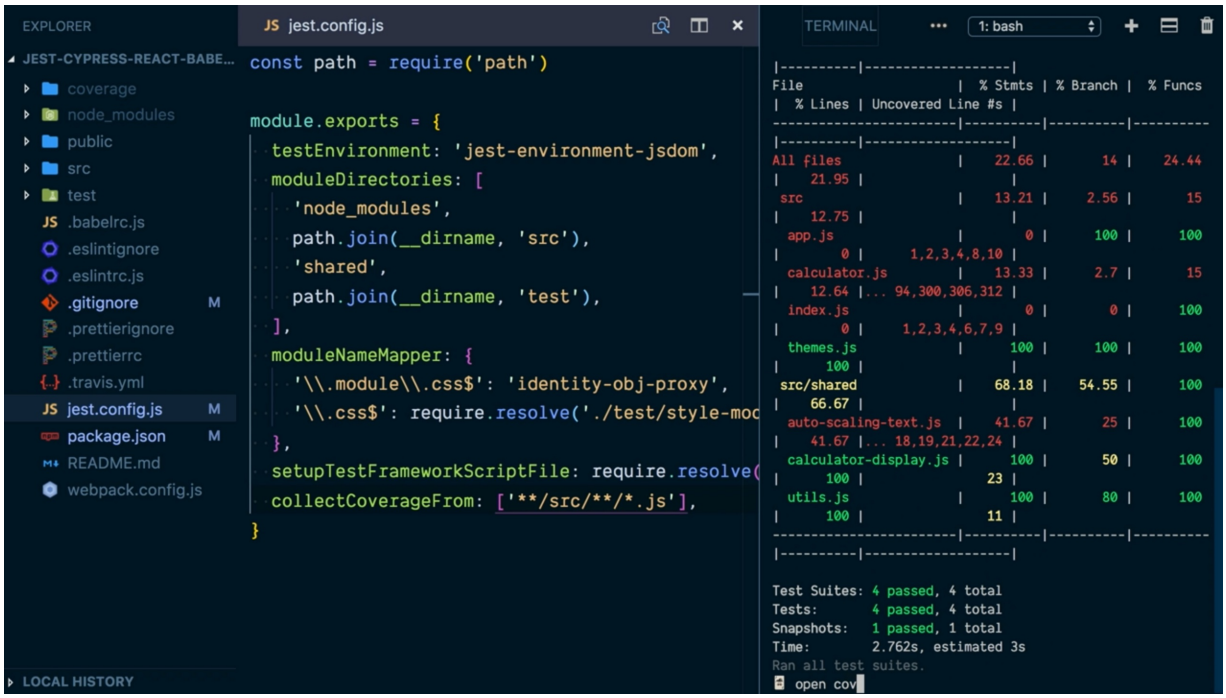
###jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  moduleDirectories: [
    'node_modules',
    path.join(__dirname, 'src'),
    'shared',
    path.join(__dirname, 'test'),
  ],
  moduleNameMapper: {
    '\\.module\\.css
: 'identity-obj-proxy',
    '\\.css
: require.resolve('./test/style-mock.js'),
  },
  setupTestFrameworkScriptFile:
require.resolve('./test/setup-tests.js'),
  collectCoverageFrom: ['**/src/**/*.js'],
}
```

All of JavaScript inside of the `src` directory. By default, this will exclude any of the files that are actually test. We can just leave that and not worry about those.

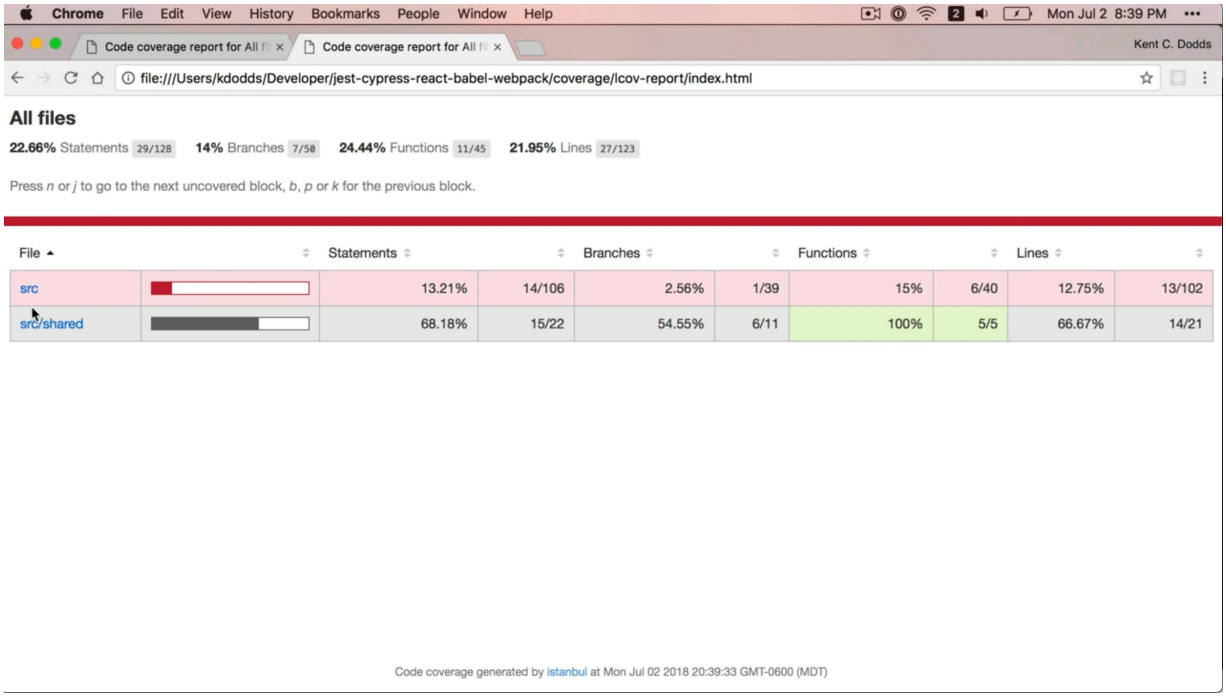
Now if I run `npm t` to run my test again, that's going to run Jest with coverage and it'll split out this report again for me.

Test Report

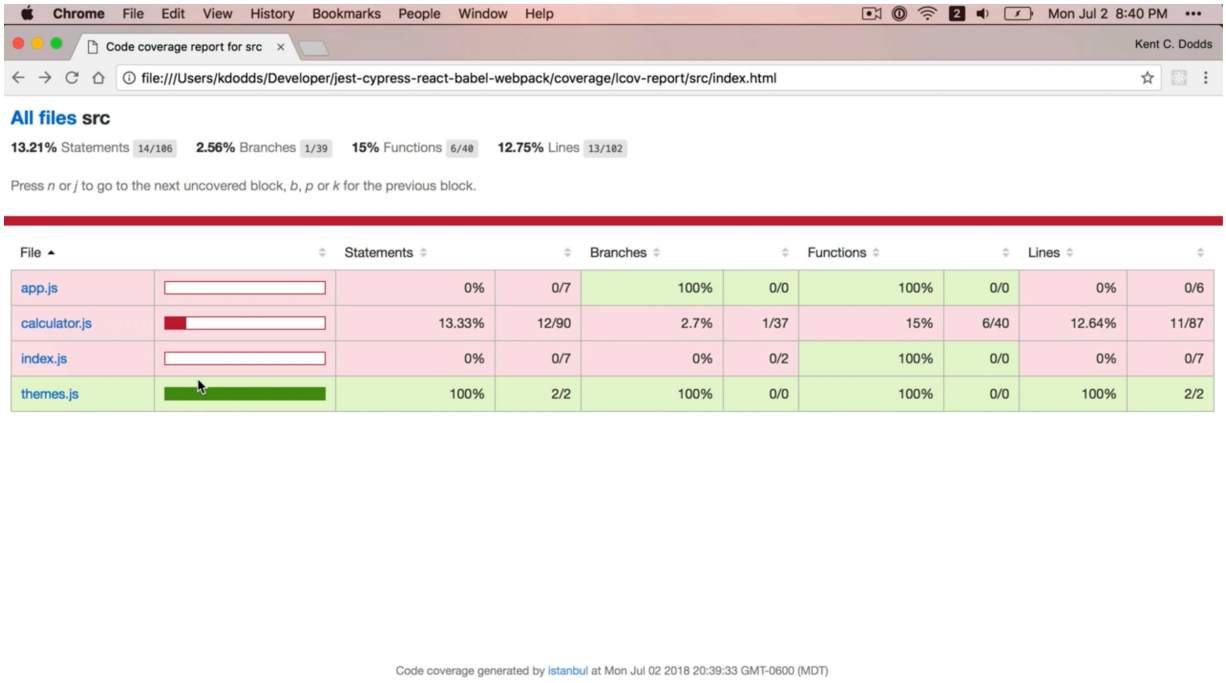


If I open this up again, `coverage/lcov-report/index.html`. I slide over here to see that report now. I'm no longer seeing that test file.

```
$ open coverage/lcov-report/index.html
```



If we open up the old one and compare the new, we're going to see that the coverage the numbers are actually going down, which is because it's giving us a more accurate reporting of our code coverage in our project. In fact, before it wasn't showing us the `app` or the `index` coverage, because those files weren't even being used in our test.



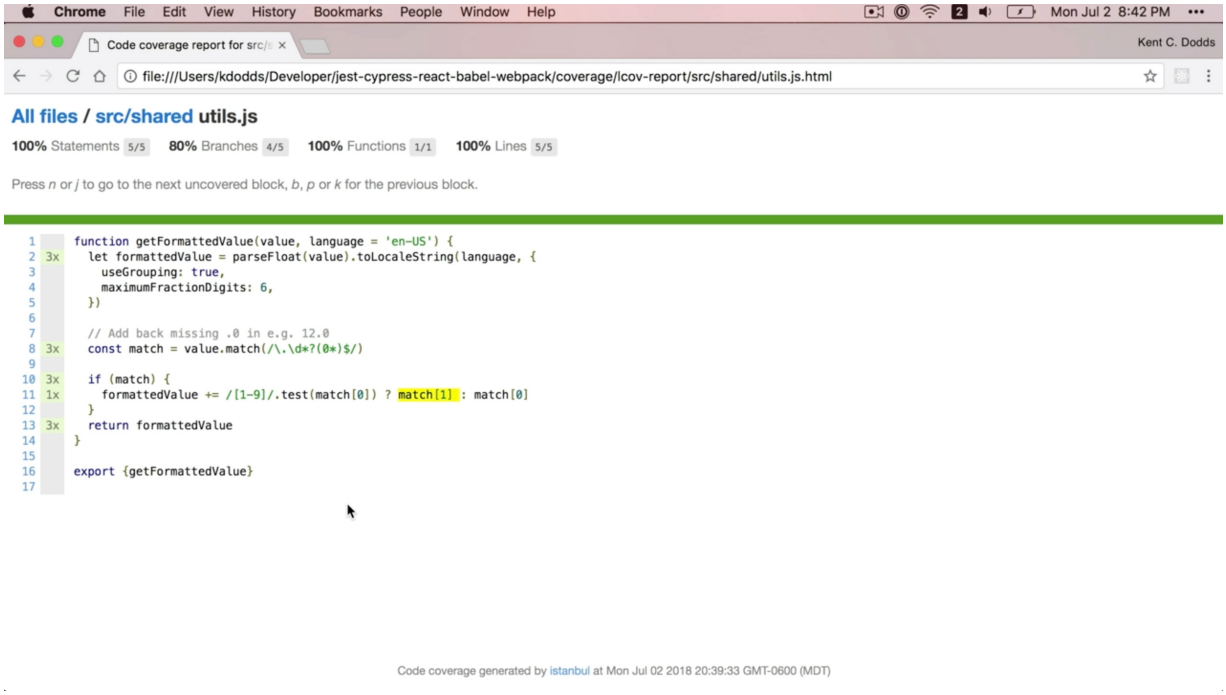
By using the collect coverage from property, it's telling Jest that we need to report coverage on all of those files not just the ones that are included in our tests.

That's the really important part of getting an accurate code `coverage` report for our entire project. In review, to make this work,all we needed to do is, add a `--coverage` flag to our `jest` command.

In our `jest.config.js` configurations, specify `collectCoverageFrom`, so we exclude the `test` directory and include all of the files that are relevant in our `src` directory.

Analyze Jest Code Coverage Reports

You might be wondering how Jest can generate this report. How can it know which one of these statements are being run, which one of these branches is being taken, and which one of these functions and lines are being run during our tests?



`Jest` actually uses a tool under the covers called `Istanbul`, which uses a Babel plugin to instrument this code for coverage. Then when our code is run, it takes that instrumentation and generates a report out of it. Let's take a look at what that Babel plugin does to our code.

This is that `utils.coverage.js` file, instrumented with coverage. There's not a whole lot that's the same, until you get down here to the very bottom. Then we have that `getFormattedValue` function, but even that looks a little bit different.

utils.coverage.js

```
function getFormattedValue(value) {
  var language = arguments.length > 1 &&
arguments[1] !== undefined ? arguments[1] :
(cov_2e1x39rbgn.b[0][0]++, 'en-US');
  cov_2e1x39rbgn.f[0]++;
  var formattedValue = (cov_2e1x39rbgn.s[0]++,
parseFloat(value).toLocaleString(language, {
    useGrouping: true,
    maximumFractionDigits: 6
  })); // Add back missing .0 in e.g. 12.0
```

Here, we have this `cov_2e1x39rbgn` all over the place, and this `.f`, a `.s`, and a `.b`. We have another `.b` over here, and we're using these comma and all of these `++` operations. What's going on here?

```
 var match = (cov_2e1x39rbgn.s[1]++,
value.match(/\.\d*?(0*)$/));
  cov_2e1x39rbgn.s[2]++;

  if (match) {
    cov_2e1x39rbgn.b[1][0]++;
    cov_2e1x39rbgn.s[3]++;
    formattedValue += /[1-9]/.test(match[0]) ?
(cov_2e1x39rbgn.b[2][0]++, match[1]) :
(cov_2e1x39rbgn.b[2][1]++, match[0]);
  } else {
    cov_2e1x39rbgn.b[1][1]++;
  }

  cov_2e1x39rbgn.s[4]++;
  return formattedValue;
}

export { getFormattedValue };
```

Up here at the top of the file, we're creating this variable this `cov_2e1x39rbgn`. Then inside of this function, which is an immediately invoked function expression, we're creating this closure to the store the `path`, a `hash`, and a bunch of other coverage information.

```
var cov_2e1x39rbgn = function () {
  var path = "/Users/kdodds/Developer/jest-
cypress-react-babel-webpack/src/utils.js",
    hash =
"11d29cc5894ffec96585bad5b940d9b13baa3ea3",
    Function = function () {}.constructor,
    global = new Function('return this')(),
    gcv = "__coverage__",
```

This coverage data is particularly of note. It stores the `path` to our file, as well as this `statementMap`, which will have an entry for every single one of our statements in our file.

```
coverageData = {
    path: "/Users/kdodds/Developer/jest-cypress-
react-babel-webpack/src/utils.js",
    statementMap: {
      "0": {
        start: {
          line: 2,
          column: 23
        }
      }
    }
```

It also has a `fnMap`, which as an entry for every function in our file -- in our case, that's only one, the `getFormattedValue` -- and a `branchMap`, which will have an entry for every branch.

```
fnMap: {
      "0": {
        name: "getFormattedValue",
```

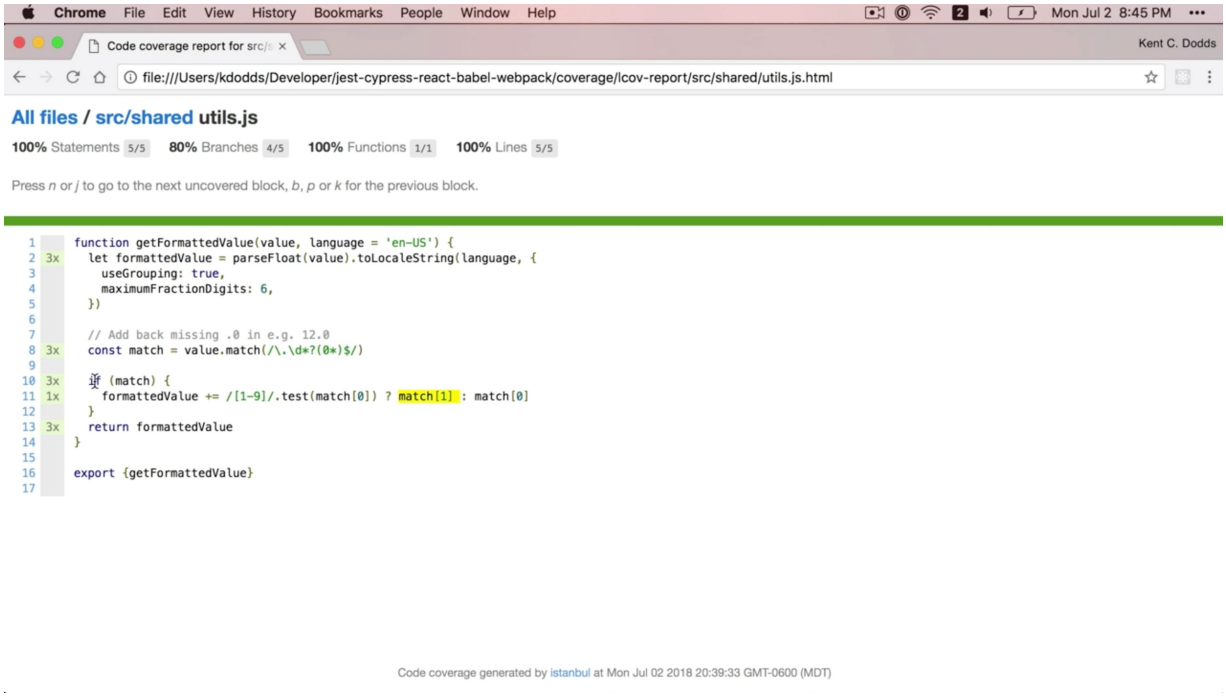
```
        decl: {
          start: {
            line: 1,
            column: 9
          },
          end: {
            line: 1,
            column: 26
          }
        },
        loc: {
          start: {
            line: 1,
            column: 54
          },
          end: {
            line: 14,
            column: 1
          }
        },
        line: 1
      }
    },
    branchMap: {
      "0": {
        loc: {
          start: {
            line: 1,
            column: 34
          },
          end: {
            line: 1,
            column: 52
          }
        }
```

Branches are special, because it also has this `locations`, which is an array of the `start` and `end` of all the `locations` of these branches. We have a `default-arg`, which you can see right here. That's a branch.

```
type: "default-arg",
        locations: [{
          start: {
            line: 1,
            column: 45
          },
          end: {
            line: 1,
            column: 52
          }
        }]
```

We also have this `if` statement, which in this case, has two `locations`.

```
type: "if",
        locations: [{
          start: {
            line: 10,
            column: 2
          }
```



Because we have the `if` here, and then there's an `else` case, which doesn't exist. If this `if` doesn't run, then that `else` case will be taken automatically, and so on, and so forth, including condition expressions, which is this ternary right here.

Output

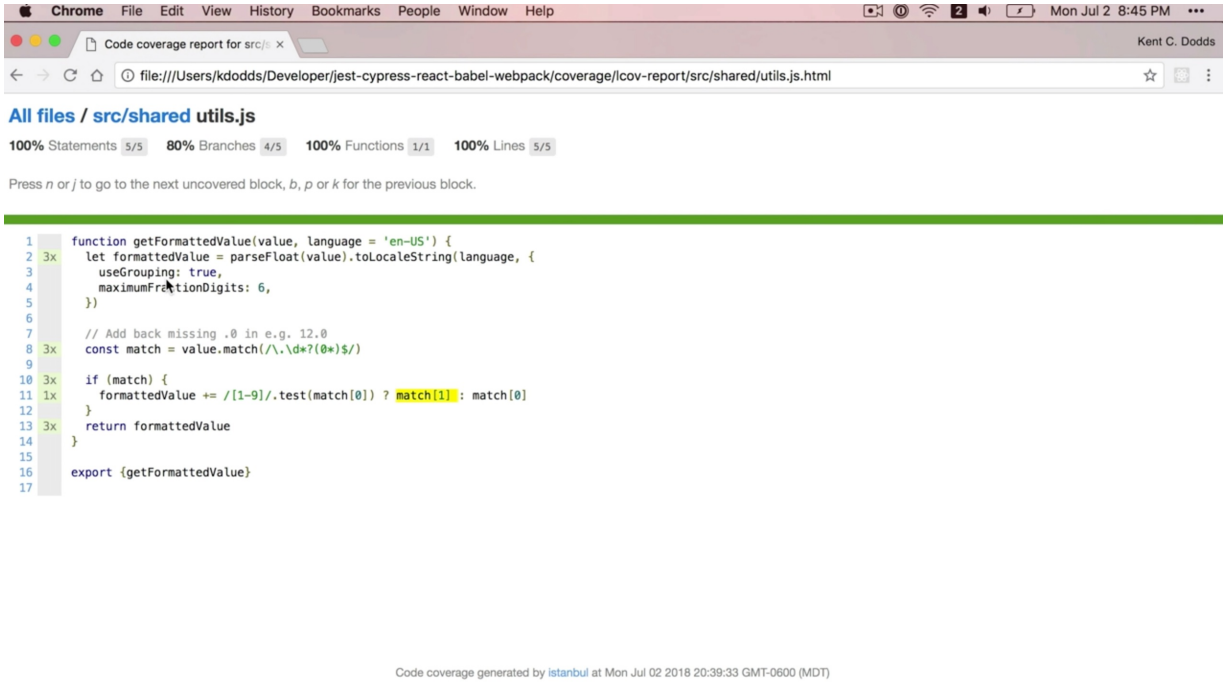
```
formattedValue += /[1-9]/.test(match[0]) ?
match[1] : match[0]
```

With that in place, that coverage object has the property `f` for functions. It will increment that first `function` in the `fmMap` any time this `function` is run.

utils.coverage.js

```
function getFormattedValue(value) {
  var language = arguments.length > 1 &&
arguments[1] !== undefined ? arguments[1] :
(cov_2e1x39rbgn.b[0][0]++, 'en-US');
  cov_2e1x39rbgn.f[0]++;
  var formattedValue = (cov_2e1x39rbgn.s[0]++,
parseFloat(value).toLocaleString(language, {
    useGrouping: true,
    maximumFractionDigits: 6
  })); // Add back missing .0 in e.g. 12.0
```

That's how it knows that this function is run three times, because this `f[0]++` will be incremented every single time this `getFormattedValue` is run.



The same goes for this `if (match)`. It adds an else case for us, so that it can keep track of the else case in this `if` statement. Then in this condition `ternary` expression, it's using a `comma operator` so that it can first increment this branch.

Then the completion value of this comma operator, `(cov_2e1x39rbgn.b[2][0]++, match[1])`, will be exactly what we had in our source, so that our tests work exactly as they would in production, but they can keep track of `coverage`. Understanding this will help you analyze this `coverage` report more accurate.

```
if (match) {
    cov_2e1x39rbgn.b[1][0]++;
    cov_2e1x39rbgn.s[3]++;
    formattedValue += /[1-9]/.test(match[0]) ?
(cov_2e1x39rbgn.b[2][0]++, match[1]) :
(cov_2e1x39rbgn.b[2][1]++, match[0]);
  } else {
    cov_2e1x39rbgn.b[1][1]++;
  }
```

You can understand that here, we are missing some `coverage`, and how we can get that coverage in our `tests`.

Set a code coverage threshold in Jest to maintain code coverage levels

As you start tracking coverage for your project, it's a great idea to set up a `threshold` so you don't dip below the current coverage that you have, and your whole team can work toward improving those coverage numbers over time. Let's go ahead and see how we could enforce this with our `jest.config.js` configuration.

I'm going to add a `coverageThreshold` where it's `global`, and we'll have our `statements`. We'll set this to `100` percent and see what happens. `Branches` is at `100` percent, `lines` is at `100` percent, and `functions` we'll set at `100` percent.

###jest.config.js

```
coverageThreshold: {
  global: {
    statements: 100,
    branches: 100,
    lines: 100,
    functions: 100,
  }
}
```

Now let's run our tests. We're going to see that this actually fails.

Terminal

```
$ npm t
...
Jest: "global" coverage threshold for statements
(100%) not met: 22.66%
Jest: "global" coverage threshold for branches
(100%) not met: 14%
Jest: "global" coverage threshold for lines
(100%) not met: 21.95%
Jest: "global" coverage threshold for functions
(100%) not met: 24.44%

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        5.917s
Ran all test suites.
npm ERR! Test failed.  See above for more
details.
```

We see that our `statements`, `branches`, `lines`, and `functions`, which were set to be at a threshold of 100 percent, that threshold was not met because we're not even close to those numbers.

Let's set this to something that's a little bit more manageable for our project. I like to set things a little bit lower from where they are right now, so we have a little bit of leeway in case something really important needs to get out, and we don't have time to write tests for it.

I'm going to set `statements` to `17` percent, `branches` to `4`, `lines` to `17`, and `functions` to `20`.

###jest.config.js

```
global: {
  statements: 17,
  branches: 4,
  lines: 17,
  functions: 20,
}
```

Now if we run our test again and run against those coverage threshold numbers, we're going to get a passing script.

Terminal

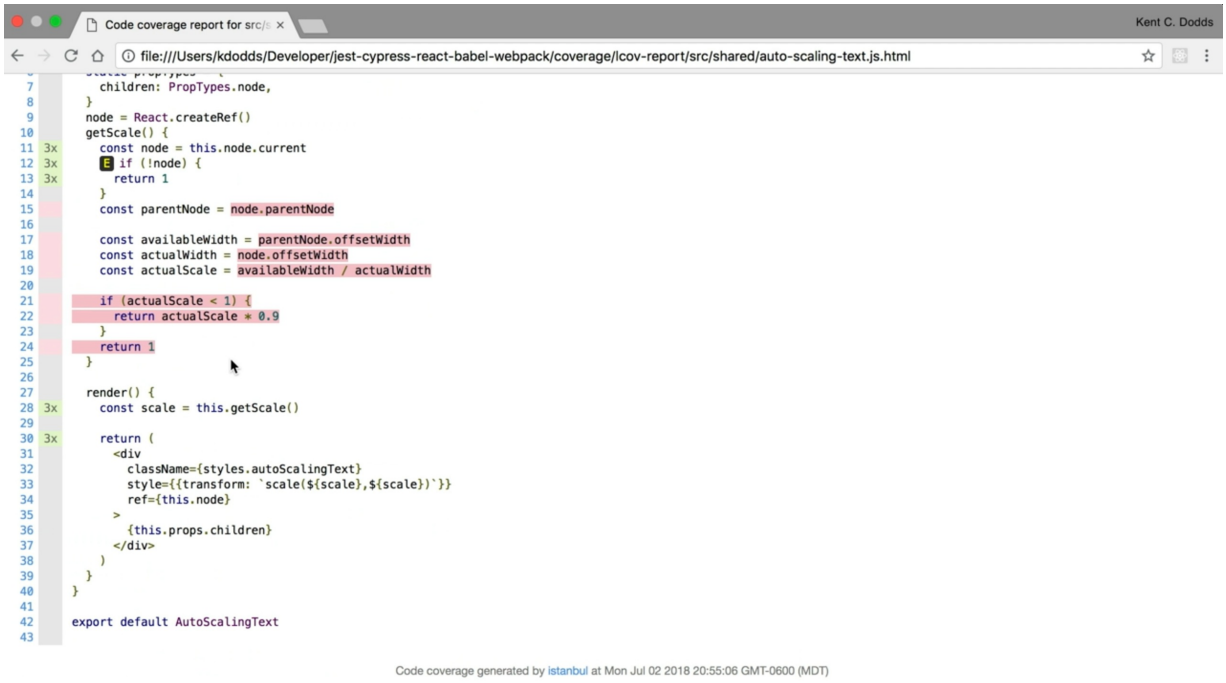
```
$ npm t

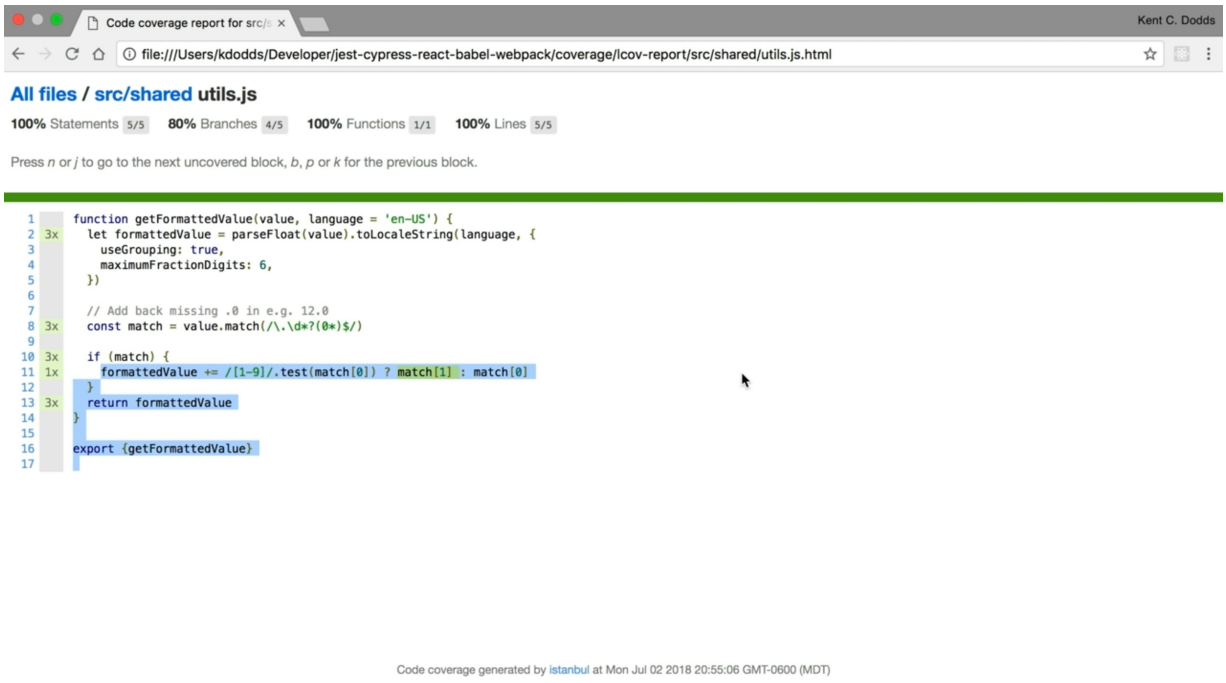
> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest --coverage

 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/__tests__/calculator.js
...
Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        3.225s
Ran all test suites.
```

If somebody were to remove a bunch of tests, then the coverage will be below our `coverageThreshold`, and it will break the test script.

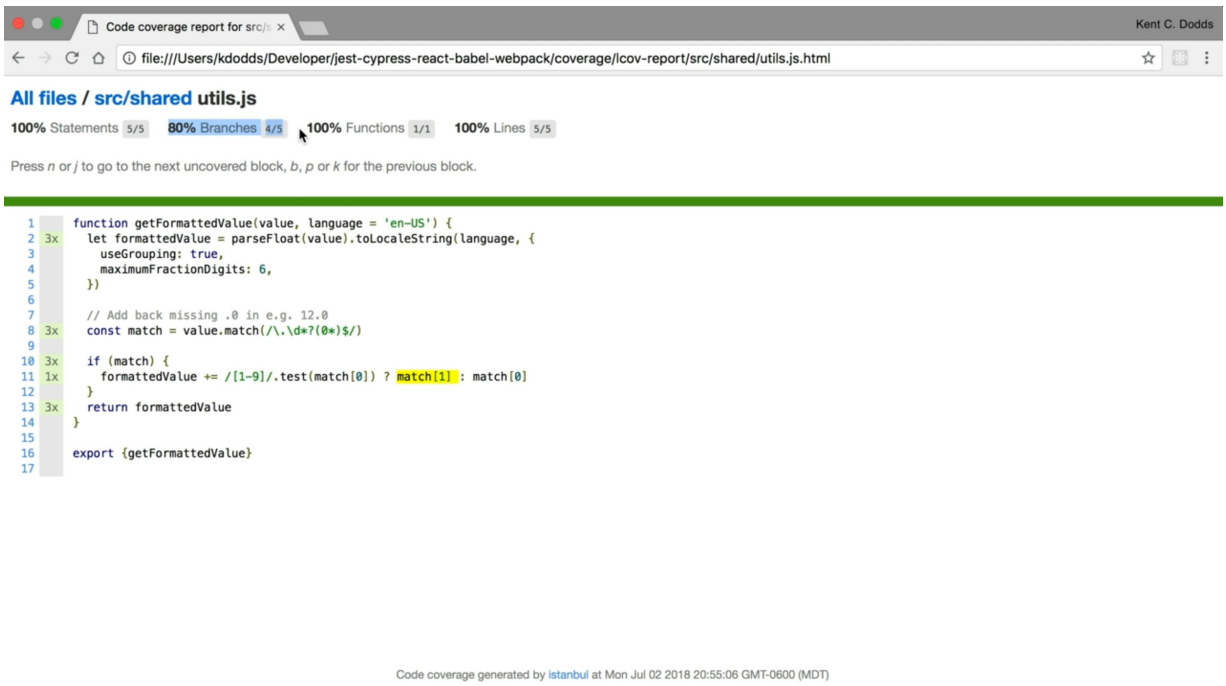
An important thing to remember is that not every line in our program is equal. Making sure that we get coverage for that line right there at the end is not the same as making sure we have coverage for this branch.





There's some code in our code base where it's a lot more important for us to make sure that we maintain our `coverageThreshold` levels than in others. We can do this by specifying a `glob` as a property. Here we have our `global` coverage. Then we can specify some specific files or globs of files that we want to maintain a certain level of `coverage`.

Here I'm going to make a source shared `utils.js` to match our utilities file.



We have 100 percent for everything except for branches.

###jest.config.js

```
coverageThreshold: {
  global: {
    statements: 100,
    branches: 100,
    lines: 100,
    functions: 100,
  },
  './src/shared/utils.js': {
      statements: 100,
      branches: 80,
      functions: 100,
      lines: 100,
    }
}
```

Let's go ahead and set those coverage thresholds to `100` percent for now and see what happens. Set those all to `100`.

```
'./src/shared/utils.js': {
      statements: 100,
      branches: 100,
      functions: 100,
      lines: 100,
    }
```

Now if we run our tests again, we're going to see that we have a failure on source shared `utils`.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest --coverage

 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js
...
Jest: "./src/shared/utils.js" coverage threshold
for branches (100%) not met:80%

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.921s
Ran all test suites.
npm ERR! Test failed.  See above for more
details.
```

That `coverageThreshold` for `branches` is `100` percent. It's not met because it's at `80` right now. We can maintain that `coverage` level for our utils file while still setting this `global coverage` level for the rest of our files in our project.

###jest.config.js

```
'./src/shared/utils.js': {
      statements: 100,
      branches: 80,
      functions: 100,
      lines: 100,
    }
```

Now if I ran `npm t` again, having updated the `branches`, our test script will continue to pass.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest --coverage

 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/shared/__tests__/utils.js
...
Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.82s
Ran all test suites.
```

If somebody makes a change to `utils`, they're going to have to make an update to the tests to ensure that they're covering their changes.

In review, to add a `coverageThreshold` to your `jest.config.js` configuration, you simply add a `coverageThreshold` property to your `jest.config.js`. For the `global` coverage, you can set `global`. You can additionally add specific `files` or `globs` of files to have some specific code `coverage` for those specific files.

Use Jest Watch Mode to speed up development

We have a great test script here that's running `jest` to get our coverage. Anytime we make a change, we can run `npm t` to run that test script and it will run all of our tests. We'll make sure that we haven't broken anything.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> jest --coverage

 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js

...

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        3.315s
Ran all test suites.
```

As great is that is, it would be really nice, if we didn't have to continuously run `npm t` over and over again, every time we make a change and want to verify that we haven't broken anything. This is especially used for when we're trying to develop our software using the test-driven development philosophy with the `red`, `green`, `refactor` cycle. Jest comes built in with the watch mode.

Let's add a script for that. We'll call a `test:watch`. This will simply be `jest -- watch`.

package.json

```
{
  "name": "calculator",
  "version": "1.0.0",
  "description": "See how to configure Jest and
Cypress with React, Babel, and Webpack",
  "main": "index.js",
  "scripts": {
    "test": "jest --coverage",
    "test:watch": "jest --watch",
    "test:debug": "node --inspect-brk
./node_modules/jest/bin/jest.js --runInBand --
watch",
    "dev": "webpack-serve",
    "build": "webpack --mode=production",
    "postbuild": "cp ./public/index.html
./dist/index.html",
    "start": "serve --no-clipboard --listen 8080
dist",
    "lint": "eslint .",
    "format": "prettier \"**/*.js\" --write",
    "validate": "npm run lint && npm run test &&
npm run build",
    "setup": "npm run setup && npm run validate"
  },

  ...
```

We can save that, pop up in our terminal and run `npm run test:watch`.

Terminal

```
$ npm run test:watch

No tests found related to files changed since
last commit.
Press `a` to run all tests, or run Jest with `--
watchAll`.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

This enters Jest interactive watch mode. Here, right at the very top you'll notice it says, "No test found related to files change since last commit."

That's referring to the `git commit`. That means that Jest is actually git intelligent. It can determine what tests are relevant based off of what files you've changed. We can actually run all of the tests, but this one is a little bit more exploration.

I'm going to go ahead and open `auto-scaling-text.js`. I'm going to break something. Instead of returning `1`, I'm going to return `2`.

auto-scaling-text.js

```
if (!node) {
      return 2
```

Then, we'll save that and our test automatically rerun.

Terminal

```
$
 PASS  src/shared/__tests__/auto-scaling-text.js
 FAIL  src/shared/__tests__/calculator-
display.js
  ● mounts

    expect(value).toMatchSnapshot()

    Received value does not match stored
snapshot "mounts 1".

    - Snapshot
    + Received

    ...

      at Object.toMatchSnapshot
(src/shared/__tests__/calculator-
display.js:7:32)

 › 1 snapshot failed.
 PASS  src/__tests__/calculator.js

Snapshot Summary
 › 1 snapshot failed from 1 test suite. Inspect
your code changes or press `u`
 to update them.

Test Suites: 1 failed, 2 passed, 3 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:   1 failed, 1 total
Time:        3.686s
Ran all test suites related to changed files.

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

This time, it actually runs the tests that are impacted by that change -- our `calculator`, our `calculator-display`, and our `auto-scaling-text`.

```
$
PASS  src/shared/__tests__/auto-scaling-text.js
FAIL  src/shared/__tests__/calculator-display.js
```

The only test that's failing here is the `calculator-display`. That's because, it has a snapshot which shows us the scale in the style attribute. If we scroll down to the bottom, we'll see that watch usage right here, press `W` to show more, so I'll do that.

```
$ W

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press u to update failing snapshots.
 › Press i to update failing snapshots
interactively.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Because we have a snapshot failure, we have two additional commands that we could execute.

01:40 We can press U to update the failing snapshots as noted here in the snapshot summary. We can also press I to update failing snapshots interactively. If I press I, it's going to run through each one of our failed snapshots and ask us if we want to update that particular snapshot.

```
$ I
 FAIL  src/shared/__tests__/calculator-
display.js
  × mounts (60ms)

  ● mounts

    expect(value).toMatchSnapshot()

    Received value does not match stored
snapshot "mounts 1".

    - Snapshot
    + Received
...

 › 1 snapshot failed.
Snapshot Summary
 › 1 snapshot failed from 1 test suite. Inspect
your code changes or press `u`
 to update them.

Test Suites: 1 failed, 1 total

Interactive Snapshot Progress
 › 1 snapshot remaining
Watch Usage › Press u to update failing
snapshots for this test.
 › Press s to skip the current test.
 › Press q to quit Interactive Snapshot Mode.
 › Press Enter to trigger a test run.

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

We can press `U` to update the failing snapshot for this test. We can skip this current `test`. We can quit the interactive snapshot mode and trigger another `test` run. I'm going to go ahead and press `S` to skip the current `test`.

```
$ S
Interactive Snapshot Result
 › 1 snapshot reviewed, 1 snapshot skipped

Watch Usage
 › Press r to restart Interactive Snapshot Mode.
 › Press q to quit Interactive Snapshot Mode.

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

Here, we see that `1 snapshot reviewed`, `1 snapshot skipped`. We can press `R` to `restart Interactive Snapshot Mode`, or `press Q` to `quit Interactive Snapshot Mode`. I'm going to `press Q`.

```
$ q
...
› 1 snapshot failed.
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/__tests__/calculator.js

 › 1 snapshot failed from 1 test suite. Inspect
your code changes or run `npmtest -- -u` to
update them.

Test Suites: 1 failed, 2 passed, 3 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:   1 failed, 1 total
Time:        3.999s, estimated 5s
Ran all test suites.
```

That will return as to the mode that we're in before we started interactive snapshot mode. I'll press the `W` key again to see the rest of our watch usage.

```
$ W

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press u to update failing snapshots.
 › Press i to update failing snapshots
interactively.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Here, let's go ahead and try this press F to only run failing tests. We have run three tests here. If we press F, then, that's going to skip those other test and only run the ones that are failing.

```
$ F
FAIL  src/shared/__tests__/calculator-display.js
  × mounts (60ms)

  ● mounts

    expect(value).toMatchSnapshot()

    Received value does not match stored
snapshot "mounts 1".

    - Snapshot
    + Received
...
```

````
This is great when you've uncovered a particular
bag, and you want to just make sure that you're
fixing the `tests` that are failing without
having all the noise at the extra test while
you're writing your `console.log` statement or
using the `debug` mode.

Let's go ahead and press `U` again down here.
Now, we can press `F` to quit the only `failed`
test mode. I'll press `F`, and again, it's going
to run all the tests that were impacted by the
changes that we've made.

```bash
$ f
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/__tests__/calculator.js FAIL
src/shared/__tests__/calculator-display.js
  ● mounts
    expect(value).toMatchSnapshot()
    Received value does not match stored
snapshot "mounts 1".
...
 › 1 snapshot failed.
Snapshot Summary
 › 1 snapshot failed from 1 test suite. Inspect
your code changes or press `u`
 to update them.

Test Suites: 1 failed, 2 passed, 3 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:   1 failed, 1 total
Time:        1.835s
Ran all test suites related to changed files.
````

I'm going to go ahead and fix this broken code. Now, we have no changes in `git`, and no tests are being run.
````

auto-scaling-text.js

```
getScale() {
    const node = this.node.current
    if (!node) {
      return 2
```

Let's go ahead and press W to show more usage here.

```
$ w
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern. › Press t to filter by a test name
regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

We have this A key that we can press to run all the tests. I'll press A, and that will run all the tests in our project.

```
$ a
 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/__tests__/calculator.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 totalSnapshots:   1
passed, 1 total
Time:        1.987s
Ran all test suites.
```

Here, I'll press W, and when we're in the all mode, we have this new key, we can press to press O to run test related to files that have been changed. That's the mode that we start in. If I press O, there are no files that have been changed since last commit, so it doesn't run any test.

```
$ o

No tests found related to files changed since
last commit.
Press `a` to run all tests, or run Jest with `--
watchAll`.
```

Now, I'll press W again.

```
$ w
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern. › Press t to filter by a test name
regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

This time, we're going to look at the `P key to filter by a file name regex pattern`. I'll hit `P` and I'm now in `Pattern Mode Usage`.

```
$ p
Pattern Mode Usage
 › Press Esc to exit pattern mode.
 › Press Enter to filter by a filenames regex
pattern.
```

I'll go ahead and type `auto` and that'll find me just the `auto-scaling-text`, so I can focus on those tests.

```
 $ pattern › auto
  PASS  src/shared/__tests__/auto-scaling-
text.js
  ✓ renders (25ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.283s
Ran all test suites matching /auto/i.
```

I'll press `W` again and I can run `P` again to filter by a different `regex` pattern.

```
$ p
Active Filters: filename /auto/

Pattern Mode Usage
 › Press Esc to exit pattern mode.
 › Press Enter to filter by a filenames regex
pattern.

 pattern ›
```

This time I'll do `calc.*.js`. Now, I'm running both of these tests, because they each have `calc` and a `.js`.

```
$ pattern › calc.*.js
```

Now, I'll press the `P` key again. If I wanted to get out of this `pattern mode` usage, then I can actually press the `escape` key to exit `pattern mode`, but I'll press the `P` key again, and I'll type `util`, and that runs my utils test.

```
$ pattern › util

 PASS  src/shared/__tests__/utils.js
  ✓ formats the value (8ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.124s, estimated 1s
Ran all test suites matching /util/i.
```

Now if I press `W`, we have this new set of commands here called `Active Filters`.

```
$ W
Active Filters: filename /util/
 › Press c to clear filters.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press o to only run tests related to changed
files.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Here, we can press the `C` key to clear all the filters. I'll go ahead and clear all the filters with the `C` key, and I'm returned back to the `--watch` mode.

```
$ c
No tests found related to files changed since
last commit.
Press `a` to run all tests, or run Jest with `--
watchAll`.
```

Next, let's press the `W` key again.

```
$ w
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

We can press the `T` key to filter by a `test name regex pattern`, so it works similar to a `file name regex pattern`. Let's go ahead and take a look at how that is different. I'm going to open up `test utils`.

```
$ t

Pattern Mode Usage
 › Press Esc to exit pattern mode.
 › Press Enter to filter by a tests regex
pattern.

 pattern › test utils
```

Here, I've a test that has the title `formats the value`.

utils.js

```
test('formats the value', () => {

expect(getFormattedValue('1234.0')).toBe('1,234.
0')
})
```

I'm going to press the `T` key and type `formats`. That's going to search through all of my `tests` in my`test suite`, and find if `test` that have a title that match the `regex` formats.

Terminal

```
$ t
pattern › formats
 PASS  src/shared/__tests__/utils.js

Test Suites: 3 skipped, 1 passed, 1 of 4 total
Tests:       3 skipped, 1 passed, 4 total
Snapshots:   0 total
Time:        2.153s
Ran all test suites with tests matching
"formats".

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

That actually applies within a single `test`. I can go ahead and add a `test` that `fails`. Here, we'll throw a new `error`, `Hi`.

utils.js

```
test('fails', () => {
  throw new Error('hi')
})
```

I save that and it actually does not run that test. It's only running the test that match the `regex` pattern that I provided.

```
 PASS  src/shared/__tests__/utils.js

Test Suites: 3 skipped, 1 passed, 1 of 4 total
Tests:       4 skipped, 1 passed, 5 total
Snapshots:   0 total
Time:        2.653s
Ran all test suites with tests matching
"formats".
```

05:19 This allows us to be hyper focused on a particular test that's failing without having the noise of extra test that might be using some code that would be console logging things as we're trying to debug problems. This behave similarly to if I were to add a `.only`.

```
test.only('formats the value', () => {

expect(getFormattedValue('1234.0')).toBe('1,234.
0')
})
```

This test script accepted. I don't have to change any code to do that, I can do all of that inside of the interactive `--watch` mode.

I'll press the `W` key again here.

```
$ w

Active Filters: test name /formats/
 › Press c to clear filters.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press o to only run tests related to changed
files.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

I have `Active Filters`. I can press the `C` key to disable those `Active Filters`.

```
$ c
```

Now, it's going to run all of my tests including this failing test. Let's go ahead and remove this failing test and save. We're back to square one with git, so we're not running any test.

```
$
No tests found related to files changed since
last commit.
Press 'a' to run all tests, or run Jes with '--
watchAll'.
```

Now, let's go ahead and press `W`. I'm going to go back to all mode. We'll press `A` to run all the tests.

```
$ a
 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/__tests__/calculator.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.155s
Ran all test suites.

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

I'll press `W` again,

```
$ w
Watch Usage
 › Press f to run only failed tests.
 › Press o to only run tests related to changed
files.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

and here, I can press `enter` to trigger a test run.

```
$ enter
 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/calculator-
display.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.154s
Ran all test suites.

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

This command is available on pretty much all of the other modes.

Anytime you want to rerun the test for some reason maybe just didn't rerun the test for you automatically, you can `trigger` another `test` run by hitting the `enter` key.

Let's take one last look at the `--watch` usage here. We can press `Q` to quit watch mode. I'll press `Q`, and now, we're no longer in our `--watch` mode and we can execute other commands.

```
$ q
$
```

In review, all that we have to do to make this work is, in our `package.json`, we added a `test:watch`, which runs `jest --watch`. It's actually good idea to add the `--watch` flag to our `debug` script, because it makes it a lot easier for us to `debug` things as we're going.

We can make changes, the `test` will rerun and our `debug` will catch again, and that's `Jest --watch mode`.

Run Jest Watch Mode by default locally with is-ci-cli

The Jest watch mode is really awesome. We pretty much always want to run the Jest watch mode locally. Then we'll run `jest --coverage` in CI with `Travis`. We might also want to run that locally as well.

It would be nice to not have to think about whether we're running `test:watch` or the test script itself. In addition, we can run `npm t` to run our test, but we have to run `npm run test:watch`, because there's no alias for the `test:watch` command.

00:27 It would be really nice if we could just run `npm t`, and that would run watch mode locally, and with coverage on `CI`.

I'm going to `npm install` as a dev dependency `is-ci-cli`.

```
$ npm install --save-dev is-ci-cli
```

With that installed and saved to our `dev dependencies` in our `package.json`, I'm going to add another script here called `test:coverage`.

That's where I'm going to use `jest --coverage`. Then I'll switch my `test-script` to is `CI`. If we are on `CI`, then I want to run `test:coverage`, otherwise I want to run `test:watch`.

package.json

```
{
  "name": "calculator",
  "version": "1.0.0",
  "description": "See how to configure Jest and
Cypress with React, Babel, and Webpack",
  "main": "index.js",
  "scripts": {
    "test": "is-ci \"test:coverage\"
\"test:watch\"",
    "test:coverage": "jest --coverage",
    "test:watch": "jest --watch",
    "test:debug": "node --inspect-brk
./node_modules/jest/bin/jest.js --runInBand --
watch",
    "dev": "webpack-serve",
    "build": "webpack --mode=production",
    "postbuild": "cp ./public/index.html
./dist/index.html",
    "start": "serve --no-clipboard --listen 8080
dist",
    "lint": "eslint .",
    "format": "prettier \"**/*.js\" --write",
    "validate": "npm run lint && npm run test &&
npm run build",
    "setup": "npm run setup && npm run validate"
  },
  ...
}
```

With this, I can run `npm t`, and that will start Jest with watch mode.

```
$ npm t
No tests found related to files changed since
last commit.
Press 'a' to run all tests, or run Jes with '--
watchAll'.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

If I set `CI` equals one, which most continuous integration services do that, then I can run `npm t` again, and that will run `Jest` in `coverage mode`.

```
$ CI=1 npm t
...
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/__tests__/calculator.js

Test Suites: 3 passed, 3 total
Tests:       3 passed, 3 total
Snapshots:   1 passed, 1 total
Time:        2.242s, estimated 3s
Ran all test suites related to changed files.
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Additionally, I can always run `test:coverage` or `test:watch` explicitly if that's what I want to do as well.

```
$ npm run test:watch
...
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/calculator-
display.js
 PASS  src/__tests__/calculator.js
Test Suites: 3 passed, 3 total
Tests:       3 passed, 3 total
Snapshots:   1 passed, 1 total
Time:        1.932s
Ran all test suites related to changed files.

Watch Usage › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex
pattern. › Press t to filter by a test name
regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

In review, all we needed to do for this is add the `is-ci-cli` package to our `package.json`, and then we add our test coverage script. We can use `Jest coverage` there. Then we update our test script to do is `CI`. In the case that we are in `CI`, we'll run `test:coverage`. In the case we're not, we'll run `test:watch`.

Filter which Tests are Run with Typeahead Support in Jest Watch Mode

```
$ npm t
> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> is-ci "test:coverage" "test:watch"
> calculator@1.0.0 test:watch
/Users/samgrinis/jest-cypress-react-babel-
webpack> jest --watch

No tests found related to files changed since
last commit.Press `a` to run all tests, or run
Jest with `--watchAll`.
Watch Usage
› Press a to run all tests.
› Press f to run only failed tests.
› Press q to quit watch mode.
› Press p to filter by a filename regex pattern.
› Press t to filter by a test name regex
pattern.
› Press Enter to trigger a test run.
```

When we're in `Jest watch mode`, it's really awesome that I can hit `P` to filter by a `filename regex pattern`, or `T` to filter by a `test name regex pattern`. Here, I'll hit `P`, and I'll type `auto`.

```
$ p
Pattern Mode Usage
 › Press Esc to exit pattern mode.
 › Press Enter to apply pattern to all
filenames.

 pattern › auto

 Pattern matches 1 file
 › src/shared/__tests__/auto-scaling-text.js
```

Here, I'm going to run the `auto-scaling-text`.

```
$ PASS  src/shared/__tests__/auto-scaling-
text.js
  ✓ renders (36ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.199s
Ran all test suites matching /auto/i.
```

Then, I can type `P` again and run `calculator.js`.

```
$ p
Pattern Mode Usage
 › Press Esc to exit pattern mode.
 › Press Enter to apply pattern to all
filenames.

 pattern › calculator-display.js

 Pattern matches 1 file
 › src/shared/__tests__/calculator-display.js

PASS  src/shared/__tests__/calculator-display.js
✓ mounts (28ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
Time:        0.172s, estimated 1s
Ran all test suites matching /calculator-
display.js/i.
```

See here, I'm running two tests, and really, I just wanted to run the `calculator-display`, so now I have to hit `P calc.*.display`. There we go. Now I'm just running that one.

It would be really nice if I could actually see which tests are going to be running with the given expression that I type in here. Jest doesn't have this capability built in, but likely Jest watch mode is pluggable. There are plugins we can install to enhance the Jest watch mode.

Here, I'm going to `npm install` as a dev dependency `jest-watch-typeahead`.

```
$ npm install --save-dev jest-watch-typeahead
```

With that installed here in my dev dependencies of my `package.json`, I'm going to pull open my `jest.config.js` configuration, and I'm going to configure `Jest` to include these `watch mode` plugins with watch plugins.

This is an array of modules to use as watch plugins. I can say `jest-watch-typeahead/filename` and `jest-watch-typeahead/testname`. There is one plugin for each filter type.

jest.config.js

```
watchPlugins: [
    'jest-watch-typeahead/filename',
    'jest-watch-typeahead/testname',
  ]
```

With those installed and configured, I can now enter in my Jest watch mode.

```
$ npm t

No tests found related to files changed since
last commit.Press `a` to run all tests, or run
Jest with `--watchAll`.
Watch Usage › Press a to run all tests.
 › Press f to run only failed tests. › Press q
to quit watch mode.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press Enter to trigger a test run.
```

If I hit the `P` key, I get this new message here. `Start typing to filter by filename regex pattern`, and I'll start typing `au`. I'm going to see if the pattern matches one file. I can hit enter and I'll run the test for that one file.

```
 $ PASS  src/shared/__tests__/auto-scaling-
text.js
  ✓ renders (25ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.789s, estimated 2s
Ran all test suites matching /au/i.

Active Filters: filename /au/
 › Press c to clear filters.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press o to only run tests related to changed
files.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press Enter to trigger a test run.
```

I'll hit the `P` key again to filter by filename regex pattern and `CAL`, and I'll see it's matching these three files.

```
$ Pattern Mode Usage
 › Press Esc to exit pattern mode.
 › Press Enter to apply pattern to all
filenames.

 pattern › cal

 Pattern matches 3 files
 › src/__tests__/calculator.js
 › src/shared/__tests__/auto-scaling-text.js
 › src/shared/__tests__/calculator-display.js
```

If I just want to match the two, maybe say, I just wanted to match the one, then I can continue my regex until it matches just the files that I want to have run.

In addition, if I really just want to have one file run, then I can actually use the arrow keys up and down to choose a specific file that I want to have run.

I can do the same thing by pressing `T` to filter by a test name regex pattern, but when you're using this feature, I need to make sure that the cache has been primed so that Jest knows which tests are available.

I'm going to run all tests with `A`, and I'll press the `T` key.

I can start typing a regex pattern that matches what I want. That will run just that one test by its test name.

```
$
Pattern Mode Usage
 › Press Esc to exit pattern mode.
 › Press Enter to apply pattern to all
filenames.

 pattern › calculator-displa

 Pattern matches 1 file
 › src/shared/__tests__/calculator-display.js

  PASS  src/shared/__tests__/calculator-
display.js
  ✓ mounts (47ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
Time:        1.309s
Ran all test suites matching /calculator-
displa/i.

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

In review, what I had to do to make this work is install `jest-watch-typeahead` as a dev dependency in my `package.JSON`. In my `Jest` configuration, I add this `watchPlugins` to my Jest configuration that points to each one of the plugins that `jest-watch-typeahead` exposes.

Run tests with a different configuration using Jest's --config flag and testMatch option

Let's say we wanted to start rendering our application on the server side. To accomplish this with any amount of confidence, it would be a good idea to have a couple of tests, at least, that run it without the dom present in our test environment.

Right now in our `jest.config.js`, we have our `testEnvironment` set to `'jest-environment-jsdom'`.

jest.config.js

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
```

It would be a good idea to have a couple of our `tests` run in the `node environment` as well.

To do this, we're going to create a `jest configuration` for the server side and one for the client, then we'll update our scripts to support both server and client. The first thing I'm going to do is I'm actually going to move this file to the test directory and call it `jest-common.js`.

I'm going to add a new file called `jest.client.js`.

Here, we'll `module.exports` an object that spreads the `require` of `./jest-common`. Then we'll have any overrides that we want for the `jest.client` configuration come after this.

jest.client.js

```
module.exports = {
  ...require('./jest-common')
}
```

I'm going to make another file called `jest.server.js` and paste that in there.

jest.server.js

```
module.exports = {
  ...require('./jest-common')
}
```

For the server, we're going to configure a couple of things that will be a little bit different. First of all, the `testEnvironment` will be `'jest-environment-node'`.

```
  testEnvironment: 'jest-environment-node',
```

Then we'll need a different mechanism for matching the server tests than the client tests. We'll leave the client tests as they are inside of these `test` directories, then we'll set the `testMatch` for our server tests to match anything that is in a `server_tests` directory, any JavaScript file in there.

```
  testMatch: ['**/__server_tests__/**/*.js'],
```

In addition, we'll also want to set the `coverageDirectory` to a different directory so it doesn't override the coverage from our client side tests. We'll say `path.join(__dirname, '../coverage/server')`.

```
coverageDirectory: path.join(__dirname,
'../coverage/server'),
```

We'll get `path` from `require('path')`. That'll take care of our server side.

```
const path = require('path')
```

Let's take a look at the client side now. Lots of the things that we have in `jest.common.js` actually just belong in the client.

For example, the `setupFestFrameworkScriptFile` is a client side-specific configuration as well as the `testEnvironment` and our `coverageThreshold`, because our coverage numbers will be different between the client and the server.

I'm going to take this out, cut it, paste it into `jest.client.js`. Then I'll take this out, together with the `testEnvironment`, and paste it into `jest.client.js`.

jest.client.js

```
module.exports = {
  ...require('./jest-common'),
  testEnvironment: 'jest-environment-jsdom',
  setupTestFrameworkScriptFile:
require.resolve('./test/setup-tests.js'),
  coverageThreshold: {
    global: {
      statements: 17,
      branches: 4,
      functions: 20,
      lines: 17,
    },
    './src/shared/utils.js': {
      statements: 100,
      branches: 80,
      functions: 100,
      lines: 100,
    },
  },
}
```

Let's go ahead and create one of these tests for the server. We'll make a new `_server_test_` directory, and we'll just call this `index.js`. That's going to `import React` and `react-dom/server`. We'll also `import loadable` so we can preload all of the `loadable` components, then we'll have the `App`. Then we'll `renderToString` the `App`.

index.js

```
import React from 'react'
import ReactDOMServer from 'react-dom/server'
import loadable from 'react-loadable'
import App from '../app'

test('can render to static markup', async () =>
{
  await loadable.preloadAll()
  ReactDOMServer.renderToString(<App />)
})
```

Not a really robust test, but good enough for our purposes right now. If we run `npx jest --config test/jest.server.js`, it's going to say cannot find the style-mock.js,

Terminal

```
$ npx jest --config test/jest.server.js

Error: Cannot find module '.test/style-mock.js'
...
```

so let's go ahead and fix that in `jest-common.js` to use `./test`.

jest-common.js

```
moduleNameMapper: {
    '\\.module\\.css
```

```
: 'identity-obj-proxy',
    '\\.css
```

```
: require.resolve('./style-mock.js'),
```

Also, we no longer need this to be pointing here. This will need to go up a `directory` to get to our `src` directory.

```
moduleDirectories: [
    'node_modules',
    path.join(__dirname, '../src'),
    'shared',
    __dirname,
  ],
```

With those updates, let's go ahead and run this again. We're going to see `no tests found`.

Terminal

```
$ npx jest --config test/jest.server.js

No tests found
In /Users/samgrinis/jest-cypress-react-babel-
webpack/test
...
```

The reason this is happening is because when we set the jest configuration to this file (jest.server.js), it's going to treat this directory (test) where this file is found as to where it's going to be looking for our tests. It'll look for anything that matches server test under the `_tests_` directory.

That's not what we want, so let's go ahead and add something to our `jest-common` to set the `rootDir` configuration property to be `path.join(__dirname, '..')` and one directory up.

jest-common.js

```
rootDir: path.join(__dirname, '..'),
```

With that, let's try and run this again.

Terminal

```
$ npx jest --config test/jest.server.js
 PASS  src/__server_tests__/index.js
  ✓ can render to static markup (199ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.994s
```

It found our test, and our test is passing. We can do the same thing for our client side tests.

```
$ npx jest --config test/jest.client.js

Error: Cannot find module '.test/setup-tests.js'
...
```

We're going to see it cannot find module test setup test. Let's go ahead and take a look at that. Sure enough, we need to remove that test because we're in that directory right now.

jest.client.js

```
setupTestFrameworkScriptFile:
require.resolve('./setup-tests.js'),
```

Let's go ahead and run the `jest` configuration with the `client` again.

Terminal

```
$ npx jest --config test/jest.client.js
 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/calculator-
display.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        3.247s
Ran all test suites.
```

Our tests are running and passing just fine. With that, let's go ahead and update our `package.json` with a whole bunch of new scripts to deal with this new situation. I'm actually just going to copy and paste these in.

package.json

```
"scripts": {
    "test": "is-ci \"test:coverage\"
\"test:watch:client\"",
    "test:coverage": "npm run
test:coverage:client && npm run
test:coverage:server",
    "test:coverage:client": "jest --config
test/jest.client.js --coverage",
    "test:coverage:server": "jest --config
test/jest.server.js --coverage",
    "test:watch:client": "jest --config
test/jest.client.js --watch",
    "test:watch:server": "jest --config
test/jest.server.js --watch",
    "test:debug:client": "node --inspect-brk
./node_modules/jest/bin/jest.js --config
test/jest.client.js --runInBand --watch",
    "test:debug:server": "node --inspect-brk
./node_modules/jest/bin/jest.js --config
test/jest.server.js --runInBand --watch",
    "dev": "webpack-serve",
    "build": "webpack --mode=production",
    "postbuild": "cp ./public/index.html
./dist/index.html",
    "start": "serve --no-clipboard --listen 8080
dist",
    "lint": "eslint .",
    "format": "prettier \"**/*.js\" --write",
    "validate": "npm run lint && npm run test &&
npm run build",
    "setup": "npm run setup && npm run validate"
  },
```

Now we have a `test` that will just run the `coverage` and run a `watch mode` for the `client`, because we can't do a `watch mode` at the same time for the `client` and `server`, and most of our tests are going to be for the `client`.

```
"test:watch:client": "jest --config
test/jest.client.js --watch",
```

Then we have a `test:coverage;client` and a `server` which are both run when we run the `coverage` command.

```
"test:coverage:client": "jest --config
test/jest.client.js --coverage",
  "test:coverage:server": "jest --config
test/jest.server.js --coverage",
```

We have individual debugs for the client and the server.

```
"test:debug:client": "node --inspect-brk
./node_modules/jest/bin/jest.js --config
test/jest.client.js --runInBand --watch",
  "test:debug:server": "node --inspect-brk
./node_modules/jest/bin/jest.js --config
test/jest.server.js --runInBand --watch",
```

With that, we're all set, and we can set CI to one and npmt and simulate a continuous integration situation. It will run our `client test` first then our `server test` second. We're all set.

Terminal

```
$ npm t

...

 PASS  src/__tests__/calculator.js
 PASS  src/shared/__tests__/auto-scaling-text.js
 PASS  src/shared/__tests__/utils.js
 PASS  src/shared/__tests__/calculator-
display.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        3.247s
Ran all test suites.
```

In review, what we did here was we took out our `jest-config` for our `jest-common` and took all the `common` configuration here. Then we created a `jest.client.js` and a `jest.server.js` configuration that each use the `jest-common`, and configured something specific for each one of these scenarios.

One last thing that we might want to do here is our `jestConfigFile` is no longer at our root directory with `jest.config.js`. Now, we're going to say it's in `test/jest-common.js`. That will help our `esn` to resolve our `imports` of the `calculator-test-utils` properly.

Support Running Multiple Configurations with Jest's Projects Feature

Having all these `test scripts` in my `package.json` is really cramping my style. I'm not super excited about having different `configurations` for the `client` and the `server`. It would be

really nice if I could have a single `script` for my tests that would run all of the tests in my project regardless of the configuration that's being used.

`Jest` actually has built in the capability to do this, to run multiple jest configurations in a single `jest` run. Here, we can run `npx jest`. We'll use the `--projects` feature. The projects we want to run are `jest.client.js` and `jest.server.js`. We can run both of those tests all at once.

Terminal

```
$ npx jest --projects ./test/jest.client.js
./test/jest.server.js
 PASS   server  src/__server_tests__/index.js
 PASS   src/shared/__tests__/utils.js
 PASS   src/shared/__tests__/auto-scaling-
text.js
 PASS   src/shared/__tests__/calculator-
display.js
 PASS   src/__tests__/calculator.js

Test Suites: 5 passed, 5 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        3.927s
Ran all test suites in 2 projects.
```

00:40 We can also use `coverage` to record `code coverage` for all of these files all together.

```
$ npx jest --projects ./test/jest.client.js
./test/jest.server.js --coverage
 PASS   src/shared/__tests__/utils.js
 PASS   src/__server_tests__/index.js
 PASS   src/shared/__tests__/auto-scaling-
text.js
 PASS   src/shared/__tests__/calculator-
display.js
 PASS   src/__tests__/calculator.js

...

Test Suites: 5 passed, 5 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        5.056s
Ran all test suites in 2 projects.
```

That will combine the `code coverage report`.

We can also use `watch mode`.

```
$ npx jest --projects ./test/jest.client.js
./test/jest.server.js --watch
 PASS   src/shared/__tests__/utils.js
 PASS   src/__server_tests__/index.js
 PASS   src/shared/__tests__/auto-scaling-
text.js
 PASS   src/shared/__tests__/calculator-
display.js
 PASS   src/__tests__/calculator.js

...

Test Suites: 5 passed, 5 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        5.056s
Ran all test suites in 2 projects.

Watch Usage › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex
pattern.
 › Press t to filter by a test name regex
pattern.
 › Press Enter to trigger a test run.
```

That will run all of our tests, whether they be in our `client test directories` or our `server test directories`.

This is a really awesome feature. We can actually configure this in a `Jest` configuration. To do this, let's go ahead and add a `jest.config.js` at the root of our project again. Here, we're going to `module.exports` a spread of `...require` as `jest-common`.

jest.config.js

```
module.exports = {
  ...require('./test/jest-common'),
```

Then we need to put our global configuration in this `jest.config`. One of those global configuration options will be the `projects`. This will point to those project configuration files. We'll have `./test/jest.client.js` and `./test/jest.server.js`.

```
projects: ['./test/jest.client.js',
'./test/jest.server.js'],
```

In addition, because `Jest` will combine the coverage report for both the `client` and the `server`, there are some items of configuration for coverage that will be global between the two.

You can learn more about this by running `npx jest --showConfig`, then pointing to our `--config`, which will be `./test/jest.client.js`.

Terminal

```
$ npx jest --showConfig --config
./test/jest.client.js

{
  "configs": [
    {
      ...
    "watchman": true
  },
  "version": "23.6.0"
}
```

That will log out all of the configuration for our `client configuration`. Here, we have our `configs`, which is our project configuration for the client.

02:04 We also have this `globalConfig` option, which is all of the things that will apply globally, including our `coverageThreshold` and our `collectCoverageFrom` property.

```
$ "globalConfig": {
    ...
    "collectCoverage": false,
    "collectCoverageFrom": null,
    "coverageDirectory": "/Users/samgrinis/jest-
cypress-react-babel-webpack/coverage",
    "coverageReporters": [
      "json",
      "text",
      "lcov",
      "clover"
    ],
    "coverageThreshold": null,
    ...
```

Let's go ahead and pull those from our `jest.client.js`. We'll get that `coverageThreshold`, which will now apply to both the `server` and the `client` test runs. We'll paste it right in `jest.config.js`.

jest.config.js

```
  coverageThreshold: {
      global: {
        statements: 17,
        branches: 4,
        functions: 20,
        lines: 17,
      },
      './src/shared/utils.js': {
        statements: 100,
        branches: 80,
        functions: 100,
        lines: 100,
      },
    },
```

We'll go to our `jest-common.js` and pull out the `collectCoverageFrom`, and put that as part of our global configuration as well in `jest.config.js`.

jest.config.js

```
  collectCoverageFrom: ['**/src/**/*.js'],
```

With that, if we run `npx jest`, it will pick up this default configuration, and it will run our `server tests` and our `client tests`.

Terminal

```
$ npx jest
 PASS     src/__server_tests__/index.js
 PASS     src/shared/__tests__/utils.js
 PASS     src/shared/__tests__/auto-scaling-
text.js
 PASS     src/shared/__tests__/calculator-
display.js
 PASS     src/__tests__/calculator.js

Test Suites: 5 passed, 5 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        2.539s, estimated 4s
Ran all test suites in 2 projects.
```

If you run it with `--coverage`, then we can see the coverage report for all of our files.

```
$ npx jest --coverage
 PASS   server  src/__server_tests__/index.js
 PASS   dom  src/shared/__tests__/utils.js
 PASS   dom  src/shared/__tests__/auto-scaling-
text.js
 PASS   dom  src/__tests__/calculator.js
 PASS   dom  src/shared/__tests__/calculator-
display.js
-----------------------|----------|----------|-
---------|----------|---------
----------|
File                   |  % Stmts | % Branch |
% Funcs |  % Lines | Uncovere
d Line #s |
-----------------------|----------|----------|-
---------|----------|---------
```

```
----------|
All files               |    24.39 |       16 |
25.53 |    23.53 |
          |
 src                    |    14.85 |     2.56 |
16.67 |    14.29 |          |
  app.js                |       50 |      100 |
50 |       50 |        8 |
  calculator.js         |    13.33 |      2.7 |
15 |    12.64 |... 94,300,306,312 |
  index.js              |        0 |        0 |
100 |        0 |     1,2,3,4,6,7,9 |
  themes.js             |      100 |      100 |
100 |      100 |          |
 src/shared             |    68.18 |    63.64 |
100 |    66.67 |          |
  auto-scaling-text.js  |    41.67 |       25 |
100 |    41.67 |... 18,19,21,22,24 |
  calculator-display.js |      100 |      100 |
100 |      100 |          |
  utils.js              |      100 |       80 |
100 |      100 |       11 |
----------------------|----------|----------|-
---------|----------|------------------|

Test Suites: 5 passed, 5 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        4.136s
Ran all test suites in 2 projects.
```

This report combines the server runs as well as the client runs, so we get a more accurate number for how our code coverage is in the entire project.

One other thing that we can do to make things a little bit more clear, whether it's a `server` or a `client` test, is we can go into the configuration for each one of these and add a `displayName` property. This is `server`,

jest.server.js

```
displayName: 'server',
```

and we can go into the `client` and have a `displayName` for `DOM`.

jest.client.js

```
displayName: 'dom',
```

Now, if we run `npx jest` again, it will give us the `displayName` property right here in front of each one of the tests.

```
$ npx jest
 PASS   server  src/__server_tests__/index.js
 PASS   dom  src/shared/__tests__/utils.js
 PASS   dom  src/shared/__tests__/auto-scaling-
text.js
 PASS   dom  src/shared/__tests__/calculator-
display.js
 PASS   dom  src/__tests__/calculator.js

Test Suites: 5 passed, 5 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        2.539s, estimated 4s
Ran all test suites in 2 projects.
```

We can more readily see which tests are passing and which are failing. Let's go ahead and we can now update our `scripts` in our `package.json` to be a little bit more friendly. I'll just copy and paste some changes right here. Now, we just have one `test` script. We have a `test:coverage`, a `test:watch`, and our `test:debug`. It all defaults to that jest configuration.

package.json

```
"scripts": {
    "test": "is-ci \"test:coverage\"
\"test:watch\"",
    "test:coverage": "jest --coverage",
    "test:watch": "jest --watch",
    "test:debug": "node --inspect-brk
./node_modules/jest/bin/jest.js --runInBand --
watch",
```

If anybody ever wants to run a single configuration, they can just run `npx jest --config test/jest.client.js`, and it will just run those client tests.

```
$ npx jest --config test/jest.client.js
 PASS   dom  src/shared/__tests__/utils.js
 PASS   dom  src/shared/__tests__/auto-scaling-
text.js
 PASS   dom  src/__tests__/calculator.js
 PASS   dom  src/shared/__tests__/calculator-
display.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.088s, estimated 3s
Ran all test suites.
```

One last thing that we can do here is update our `.eslintrc.js` to go back to our `jest.config.js`.

eslingrc.js

```
jestConfigFile: path.join(__dirname,
'./jest.config.js'),
```

In review, to make all this work so nicely, we create a `jest.config.js` at the root, and the key point here is that our projects specify the configurations we want to use.

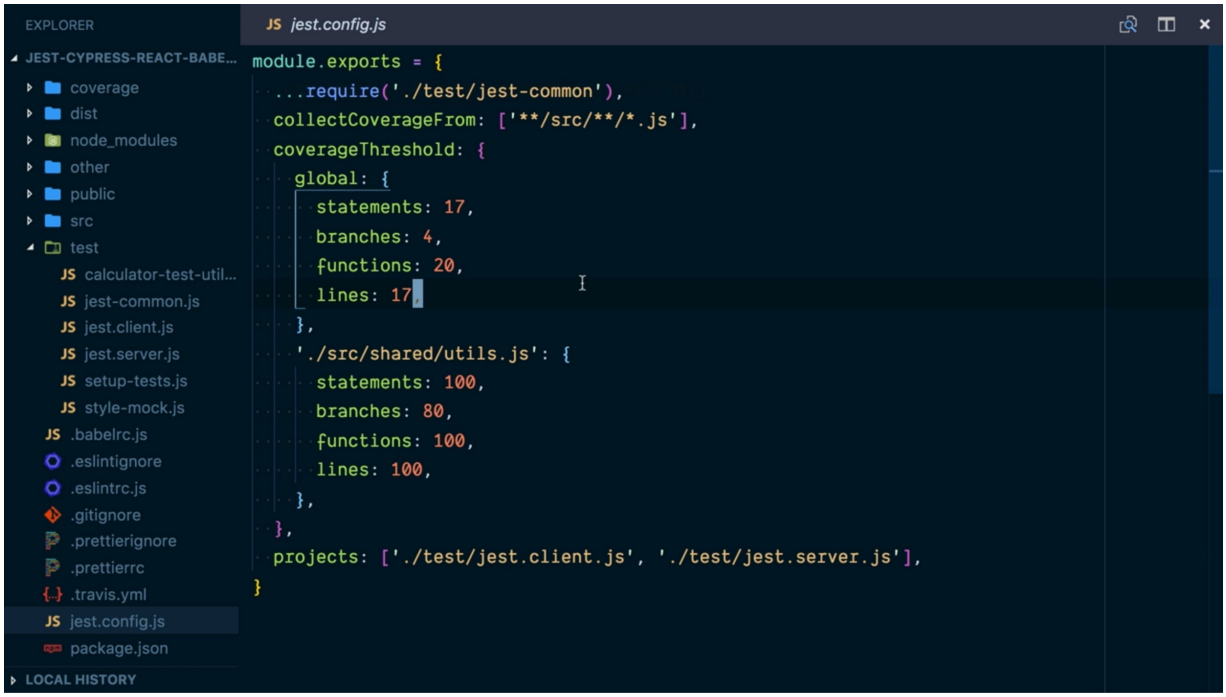
In addition, we're getting the `common` configuration from `jest-common` which is going to include the root directory, `rootDir`, the `moduleDirectories` that we want to include, `moduleNameMappers`, `coveragePathIgnorePatterns`, and our `watchPlugins`.

04:26 In our `client` and our `server`, we just have the `configuration` that's necessary for that particular project. In fact, we no longer need the `coverageDirectoryConfiguration` for the jest server, because those reports are going to be combined between the `client` and the `server`.

We can simplify that a little bit. Then we move some configuration that's global, like `collectCoverageFrom` and `coverageThreshold`, to our global configuration here, and that's Jest projects.

Test specific projects in Jest Watch Mode with jest-watch-select-projects

It's great that we've been able to combine the `Jest` configuration between the `client` and the `server` into a single configuration that we can use to run all of our tests with a single test run.



As our test regrows, it could be really helpful to be able to run each one of these projects individually.

We can do that by running `npx jest --config test/jest.client.js`, and watch if we want to do it in `--watch` mode, but that's a whole lot of stuff. We could add scripts to that, but that would be a little bit of a nightmare.

We can actually add yet another `jest-watch-plugin`, so that we can run a specific project in `--watch` mode. We're going to go ahead and `npm install` as a dev dependency `jest-watch-select-projects`.

```
npm install --save-dev jest-watch-select-projects
```

With that installed and saved into our dev dependencies right here, we can go ahead and pop open our `jest-common` configuration and added to that to our `watchPlugins`. We'll add `jest-watch-select-projects`.

jest.common.js

```
watchPlugins: [
  'jest-watch-typeahead/filename',
  'jest-watch-typeahead/testname',
  'jest-watch-select-projects',
]
```

Now if we `npm t` to start our test in `watch` mode, then we get a new watch action right here.

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> is-ci "test:coverage" "test:watch"

> calculator@1.0.0 test:watch
/Users/samgrinis/jest-cypress-react-babel-
webpack
> jest --watch

No tests found related to files changed since
last commit.
Press 'a' to run all tests, or run Jes with '--
watchAll'.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex
pattern.
 › Press P to select projects (all selected).
 › Press t to filter by a test name regex
pattern.
 › Press Enter to trigger a test run.
```

Press `P` to select projects. Currently, they are all selected. If I hit `shift p`, now I can use the `spacebar` to choose which to select and `return` to submit.

```
$ ? Select projects > - Space to select. Return
to submit
dom
server
```

I can turn off `dom` and even turn off `server` if I want to, using this `up` and `down` arrow keys and the `space bar`, and those that are checked are the ones that are going to run.

When I'm happy with my selection, I can hit `enter`. I'm in `--watch` mode for that particular project. Now, it says that "1/2 projects is selected".

```
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex
pattern.
 › Press P to select projects (1/2 selected).
 › Press t to filter by a test name regex
pattern.
 › Press Enter to trigger a test run.
```

I can press `a` to run all the tests and I'm only running the test for the server.

```
$ a
PASS   server  src/__server_tests__/index.js

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.927s
Ran all test suites in 2 projects.
```

Now if I press `P` again, and I enable `dom` and disable `server`, it's only going to run the test for the `dom`.

```
$ npx jest --projects ./test/jest.client.js
./test/jest.server.js
 PASS   server  src/__server_tests__/index.js
 PASS   src/shared/__tests__/utils.js
 PASS   src/shared/__tests__/auto-scaling-
text.js
 PASS   src/shared/__tests__/calculator-
display.js
 PASS   src/__tests__/calculator.js

Test Suites: 5 passed, 5 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        3.927s
Ran all test suites in 2 projects.
```

I press `P` again and I can enable the `server`, and it will run all of them for me.

```
$ P

 PASS   dom  src/shared/__tests__/utils.js
 PASS   dom  src/shared/__tests__/auto-scaling-
text.js
 PASS   dom  src/__tests__/calculator.js
 PASS   dom  src/shared/__tests__/calculator-
display.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        2.088s, estimated 3s
Ran all test suites.
```

In review, all we needed to do for this is add `just-watch-select-projects` to our `package.JSON` as a dev dependency. In our `jest.config.js`, where we have the `watchPlugins`, we just add `jest-watch-select-projects`.

During our `--watch` mode, we can use the `P` key to select which project we want to have run during our `test` run.

Run ESLint with Jest using jest-runner-eslint

`Jest` is more than just a testing framework. It's an entire platform for running tasks in parallel because it's running all of your tests in parallel. You can see the benefits of this as your project and your test base grows bigger and bigger over time.

In addition to being really efficient, running a lot of tasks in parallel, `Jest` also has an amazing `watch` mode that you can use to help you as you develop your software. One of the really

awesome features of `Jest` is the ability to specify a custom `runner`.

By default, `Jest` has a runner that will run the `Jest` framework tests, but you can actually have runners that run `ESLint`, `Prettier`, or even `Go` and `Python` tests. It's really remarkable what you can do. Let's go ahead and try this out with our `Linting`. I'm going to `npm install` as a `dev` dependency `jest-runner-eslint`.

Terminal

```
$ npm install --save-dev jest-runner-eslint
```

Next, I'm going to create a new configuration for our new runner. We'll go to the `test` directory and add a `jest.lint.js`. This one's going to be a little different from our `client` and our `server` configurations. I am still going to want the `rootDir` from our `jest-common`.

jest.lint.js

```
const {rootDir} = require('./jest-common')
```

Then I'm going to `module exports` this object that specifies that `rootDir`, but then I'm not going to specify a test environment because this isn't actually going to be using the built-in `runner`.

Instead, I'm going to specify my `runner` is that `jest-runner-eslint` which we just installed.

```
module.exports = {
  rootDir,
  runner: 'jest-runner-eslint',
```

I'm also going to add a `displayName` called `lint`. When we see this running with the rest of our test, we can identify what is actually being run.

```
  displayName: 'lint',
```

I'm going to change our `testMatch` to not just be the files that exist inside of our `test` directories, but actually be all the files that are JavaScript files in our project. We're going to use this `rootDir` as part of our pattern. Then we'll do any JavaScript file inside of the root directory of our project.

```
  testMatch: ['<rootDir>/**/*.js'],
```

We want to ignore some of those, so I'm going to add a `testPathIgnorePatterns`.

```
testPathIgnorePatterns: ['/node_modules/',
'/coverage/', '/dist/', '/other/'],
```

That's going to include `node_modules`, which is actually the default for this property. We'll also include `/coverage/`, `/dist/` and `/other/`, some directories that have some `JavaScript` that

we don't actually want to `Lint`.

With that configuration in place, we can run `npx jest --config test/jest.lint.js`, and it will run `Linting` across all the files in our project.

Terminal

```
$ npx jest --config test/jest.lint.js
 PASS   lint  ./webpack.config.js
 PASS   lint  src/shared/utils.js
 PASS   lint  src/shared/calculator-display.js
 PASS   lint  test/jest-common.js
 PASS   lint  src/shared/auto-scaling-text.js
 PASS   lint  ./lint-staged.config.js
 PASS   lint  ./jest.config.js
 PASS   lint  src/app.js
 PASS   lint  test/jest.client.js
 PASS   lint  src/__server_tests__/index.js
 PASS   lint  test/calculator-test-utils.js
 PASS   lint  src/calculator.js
 PASS   lint  test/setup-tests.js
 PASS   lint  test/jest.lint.js
 PASS   lint  src/themes.js
 PASS   lint  src/shared/__tests__/utils.js
 PASS   lint  test/jest.server.js
 PASS   lint  test/style-mock.js
 PASS   lint  src/shared/__tests__/calculator-
display.js
 PASS   lint  src/index.js
 PASS   lint  src/__tests__/calculator.js
 PASS   lint  src/shared/__tests__/auto-scaling-
text.js

Test Suites: 2 skipped, 22 passed, 22 of 24
total
Tests:       2 skipped, 22 passed, 24 total
Snapshots:   0 total
Time:        3.715s
Ran all test suites.
```

The benefits to this include being able to have a really nice `--watch` mode experience that `Jest` provides out of the box, in addition to the ability to `Lint` only the files that are related to the changes that we've made in our project.

Also, we can update our `jest.config.js` here to include our `Linting`. We'll add a `test/jest.lint.js` to our projects configuration,

jest.config.js

```
projects: [
  './test/jest.lint.js',
  './test/jest.client.js',
  './test/jest.server.js',
],
```

and now, when we run our tests, it's going to include our `Linting`.

Terminal

```
$ npm t

> calculator@1.0.0 test /Users/samgrinis/jest-
cypress-react-babel-webpack
> is-ci "test:coverage" "test:watch"


> calculator@1.0.0 test:watch
/Users/samgrinis/jest-cypress-react-babel-
webpack
> jest --watch

PASS   lint  test/jest.lint.js
PASS   lint  ./jest.config.js

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.915s
Ran all test suites.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex
pattern.
 › Press P to select projects (all selected).
 › Press t to filter by a test name regex
pattern.
 › Press Enter to trigger a test run.
```

Here, if I press `A` to run all the tests, that will run `Linting` on all of our files as well as our server and done tests.

```
$ A
 PASS   lint  ./webpack.config.js
 PASS   lint  src/shared/utils.js
 PASS   lint  src/shared/calculator-display.js
 PASS   lint  test/jest-common.js
 PASS   lint  src/shared/auto-scaling-text.js
 PASS   lint  ./lint-staged.config.js
 PASS   lint  ./jest.config.js
 PASS   lint  test/jest.lint.js
 PASS   lint  ./jest.config.js
 PASS   lint  src/app.js
 PASS   lint  test/jest.client.js
 PASS   lint  src/__server_tests__/index.js
 PASS   lint  test/calculator-test-utils.js
 PASS   lint  src/calculator.js
 PASS   lint  test/setup-tests.js
 PASS   lint  test/jest.lint.js
 PASS   lint  src/themes.js
 PASS   lint  src/shared/__tests__/utils.js
 PASS   lint  test/jest.server.js
 PASS   lint  test/style-mock.js
 PASS   lint  src/shared/__tests__/calculator-
display.js
 PASS   lint  src/index.js
 PASS   lint  src/__tests__/calculator.js
 PASS   lint  src/shared/__tests__/auto-scaling-
text.js

Test Suites: 2 skipped, 24 passed, of 26 total
Tests:       2 skipped, 24 passed, of 26 total
Snapshots:   1 passed, 1 total
Time:        3.715s
Ran all test suites in 3 projects

Watch Usage: Press W to show more
```

What's cool about this is now I can go into my `package.json` here and for this `Lint` script, I can now say `jest --config test/jest.lint.js`, and for my validate script I no longer need to run the `Linting` script because that will be run as part of our test script.

package.json

```
"lint": "jest --config test/jest.lint.js",
    "format": "prettier \"**/*.js\" --write",
    "validate": "npm run test && npm run build",
```

In review, what we had to do to make this work is we installed `jest-runner-eslint`. Then we created another custom configuration for Jest that specifies the root directory being the directory of our project, the `displayName` to make it easier to identify which one of the tests is actually `Linting`, and then we specified the runner.

Instead of running tests, it's running our `Linting`. Then we specified a test match so it applies to all of our JavaScript files, not just those that reside in a test directory. Then we added a `testPathIgnorePattern` so that we ignore files that existed, known `node_modules`, `coverage`, `dist`, and `other`.

Then in our `Jest` configuration, we added that to our projects and then we updated our `package.json` for the `Linting` to use `Jest` instead. In our `validate` script, we only need to run the test script now.

At first glance, this may not seem like a huge advantage. It may just seem like another dependency to install, but there are a lot of really awesome benefits that come with `Jest` capabilities

combined with `Linting`. We can explore those in another video.

Run only relevant Jest tests on git commit to avoid breakages

`Jest` has this really cool flank that we can use called `findRelatedTests`. Here, we can specify a specific file, define `tests` that are relevant to that file, and run only those `tests`. Here, I'll say source shared `auto-scaling-text.js`. Here, we have some `linting` of files that are relevant to those.

Terminal

```
$ npx jest --findRelatedTests src/shared/auto-
scaling-text.js
 PASS   lint  src/shared/auto-scaling-text.js
 PASS   lint  src/shared/calculator-display.js
 PASS   lint  src/shared/__tests__/auto-scaling-
text.js
 PASS   lint  src/shared/__tests__/calculator-
display.js
 PASS   server  src/__server_tests__/index.js
 PASS   dom  src/shared/__tests__/auto-scaling-
text.js
 PASS   dom  src/shared/__tests__/calculator-
display.js
 PASS   dom  src/__tests__/calculator.js

Test Suites: 8 passed, 8 total
Tests:       8 passed, 8 total
Snapshots:   1 passed, 1 total
Time:        4.089s
Ran all test suites related to files matching
/src\/shared\/auto-scaling-text.
js/i in 3 projects.
```

The server uses that `file` and a couple of the dom tests use that file. If I were to instead use the `utils`, then this would have `linting` for a couple of `files` that depend on `utils` and pretty much all the files in our project use that one.

```
$ npx jest --findRelatedTests
src/shared/utils.js
 PASS   lint  src/shared/utils.js
 PASS   lint  src/shared/__tests__/utils.js
 PASS   lint  src/shared/calculator-display.js
 PASS   lint  src/shared/__tests__/calculator-
display.js
 PASS   server  src/__server_tests__/index.js
 PASS   dom  src/shared/__tests__/utils.js
 PASS   dom  src/shared/__tests__/calculator-
display.js
 PASS   dom  src/__tests__/calculator.js

Test Suites: 8 passed, 8 total
Tests:       8 passed, 8 total
Snapshots:   1 passed, 1 total
Time:        4.103s
Ran all test suites related to files matching
/src\/shared\/utils.js/i in 3 projects.
```

If I were to now use `index.js` in our `src` directory, then we're going to `lint` that and none of our tests actually use that file. It's only `running` the tests that are relevant for the file that we're running this on.

```
$ npx jest --findRelatedTests src/index.js
 PASS   lint  src/index.js
  ✓ ESLint (1015ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.085s
Ran all test suites related to files matching
/src\/index.js/i.
```

This is really awesome, because we can use this in combination with the package called `lint-staged`, which will allow us to run scripts like this against all of the files that we're about to commit with git.

To do this, we're going to `npm install` as a `dev` dependency, `lint-staged`, and `husky`.

```
$ npm install --save-dev lint-staged husky

> husky@0.14.3 uninstall /Users/samgrinis/jest-
cypress-react-babel-webpack/node_modules/husky
> node ./bin/uninstall.js

husky
uninstalling Git hooks
done

> husky@0.14.3 install /Users/samgrinis/jest-
cypress-react-babel-webpack/node_modules/husky
> node ./bin/install.js

husky
setting up Git hooks
done

...

+ lint-staged@7.3.0
+ husky@0.14.3
updated 2 packages in 11.812s
```

`Husky` will be responsible for installing `git hooks` that we can configure in our `package.json` which is done right here.

We can take a look at one of those with `cat .git/hooks/pre-commit`. Here is the file that husky created for us to run before a commit happens in git.

```
$ cat .git/hooks/pre-commit
#!/bin/sh
#husky 0.14.3
```

```
command_exists () {
  command -v "$1" >/dev/null 2>&1
}

has_hook_script () {
  [ -f package.json ] && cat package.json | grep
-q "\"$1\"[[:space:]]*:"
}

cd "."

# Check if precommit script is defined, skip if
not
has_hook_script precommit || exit 0

load_nvm () {
  # If nvm is not loaded, load it
  command_exists nvm || {
    export NVM_DIR=/Users/samgrinis/.nvm
    [ -s "$1/nvm.sh" ] && . "$1/nvm.sh"
  }

  # If nvm has been loaded correctly, use
project .nvmrc
  command_exists nvm && [ -f .nvmrc ] && nvm use
}

# Add common path where Node can be found
# Brew standard installation path /usr/local/bin
# Node standard installation path /usr/local
export PATH="$PATH:/usr/local/bin:/usr/local"

# nvm path with standard installation
load_nvm /Users/samgrinis/.nvm

# nvm path installed with Brew
load_nvm /usr/local/opt/nvm
```

```
# Check that npm exists
command_exists npm || {
  echo >&2 "husky > can't find npm in PATH,
skipping precommit script in package.json"
  exit 0
}

# Export Git hook params
export GIT_PARAMS="$*"

# Run npm script
echo "husky > npm run -s precommit (node `node -
v`)"
echo

npm run -s precommit || {
  echo
  echo "husky > pre-commit hook failed (add --
no-verify to bypass)"
  exit 1
}
```

With this established, we can now open our `package.json` here and configure a `pre-commit` script, which `husky` will read and run before git `commits` any code. Here, we can use `lint-staged`, which will run some scripts based off of some configuration that we provide.

package.json

```
    "precommit": "lint-staged",
```

We can add a `lint-staged.config.js`. In here, we'll `module.exports` an object. This object will have a `linters` property. Here, we'll say a glob pattern for all JavaScript files that come across `lint-staged`. We want to run `'jest --findRelatedTests'`. With this configuration, any time we commit any code, `husky` will call `lint-staged`, and `lint-staged` will call `jest --findRelatedTests` with every file that ends in `JS` like this.

lint-staged.config.js

```
module.exports = {
  linters: {
    '**/*.js': ['jest --findRelatedTests'],
  },
}
```

`Jest` can run its tests. If it fails, then it will stop our `commit` from going through. Let's see this in action. If we look at our `git status`, we'll see that we've modified a `package.json` and we've added this `lint-staged config`, but there is no test for the `lint-staged` config other than `linting`.

Terminal

```
$ git status
On branch egghead-2018/config-jest-24
Your branch is up to date with 'origin/egghead-
2018/config-jest-24'.

Changes not staged for commit:
  (use "git add <file>..." to update what will
be committed)
  (use "git checkout -- <file>..." to discard
changes in working directory)

        modified:   package.json

Untracked files:
  (use "git add <file>..." to include in what
will be committed)

        lint-staged.config.js

no changes added to commit (use "git add" and/or
"git commit -a")
```

Let's go and take a look at what this does. What I'm going to do is, I'm going to go up here and break a test with a`auto-scaling-text`. We'll return `2` instead of `1`.

auto-scaling-text.js

```
getScale() {
    const node = this.node.current
    if (!node) {
      return 2
    }
```

Then I'm going to pop up in my terminal and we'll look at the `git status`.

```
$ git status

On branch egghead-2018/config-jest-24
Your branch is up to date with 'origin/egghead-
2018/config-jest-24'.

Changes not staged for commit:
  (use "git add <file>..." to update what will
be committed)
  (use "git checkout -- <file>..." to discard
changes in working directory)

        modified:   package.json
        modified:   src/shared/auto-scaling-
text.js

Untracked files:
  (use "git add <file>..." to include in what
will be committed)

        lint-staged.config.js

no changes added to commit (use "git add" and/or
"git commit -a")
```

We'll see that we've made a change to this file, and we've added `lint-staged.config`. I'm going to `git add .` everything and look at the `git status` again. Now, I'm going to `git commit` all the stuff with stuff.

```
$ git add .
➜  jest-cypress-react-babel-webpack git:
(egghead-2018/config-jest-24) ✗ git commit -m
'stuff'
husky > npm run -s precommit (node v10.1.0)

 ❯ Running tasks for **/*.js
   ✖ jest --findRelatedTests
✖ "jest --findRelatedTests" found some errors.
Please fix them and try committing again.

 PASS   lint  src/shared/auto-scaling-text.js
 PASS   lint  src/shared/calculator-display.js
 PASS   lint  src/shared/__tests__/auto-scaling-
text.js
 PASS   lint  src/shared/__tests__/calculator-
display.js
 PASS   server  src/__server_tests__/index.js
 PASS   dom  src/shared/__tests__/auto-scaling-
text.js
 FAIL   dom  src/shared/__tests__/calculator-
display.js
  ● mounts

expect(value).toMatchSnapshot()

Received value does not match stored snapshot
"mounts 1".

- Snapshot
+ Received
```

```
@@ -11,10 +11,10 @@
  <div
    class="emotion-0 emotion-1"
  >
    <div
      class="autoScalingText"
-     style="transform: scale(1,1);"
+     style="transform: scale(2,2);"
    >
      0
    </div>
  </div>

      5 | test('mounts', () => {
      6 |   const {container} =
render(<CalculatorDisplay value="0" />)
    > 7 |
expect(container.firstChild).toMatchSnapshot()
        |                                      ^
      8 | })
      9 |

      at Object.toMatchSnapshot
(src/shared/__tests__/calculator-
display.js:7:32)

 › 1 snapshot failed.
 PASS   dom  src/__tests__/calculator.js

Snapshot Summary
 › 1 snapshot failed from 1 test suite. Inspect
your code changes or re-run jest with `-u` to
update them.

Test Suites: 1 failed, 7 passed, 8 total
Tests:       1 failed, 7 passed, 8 total
Snapshots:   1 failed, 1 total
```

```
Time:        5.944s
Ran all test suites related to files matching
/\/Users\/samgrinis\/jest-cypress-react-babel-
webpack\/src\/shared\/auto-scaling-text.js/i in
3 projects.

husky > pre-commit hook failed (add --no-verify
to bypass)
```

This is going to run `husky`, which is going to run `lint-staged`, which will run `Jest` find related tests for each one of these files. If it finds any related tests, then it's going to run those. If that fails, then we're going to get the failure output from `Jest`.

These are all the tests that ran because the `auto-scaling-text.js` file was changed. We've got our `linting`. We've got our server tests. We've got a couple of dom tests. One of them saved us from committing code that actually would have broken in production.

Let's go ahead and fix that. I'll add a comment here, so that this file will still actually run.

auto-scaling-text.js

```
getScale() {
    const node = this.node.current
    if (!node) {
      // comment
      return 1
    }
```

We'll see that it can pass. I'm going to run `git commit` and stuff again. It will run `Jest` with find related tests. That will pass, and our commit goes through.

```
$ git add .
➜  jest-cypress-react-babel-webpack git:
(egghead-2018/config-jest-24) ✗ git commit -m
'stuff'
husky > npm run -s precommit (node v10.1.0)

 ✔ Running tasks for **/*.js
[egghead-2018/config-jest-24 98af623] stuff
 3 files changed, 15107 insertions(+), 3
deletions(-)
 create mode 100644 package-lock.json
```

In review, to make all of this work, we simply installed two packages, first, `husky`, and second, `lint-staged`. `Husky` is responsible for creating a `git hook` and running a configured script in our `package.json`. `Lint-staged` is responsible for running some scripts with all the files that are ready to be committed.

If any of these scripts that we have listed here fail, then it will prevent the `commit` from going through. By doing things this way, we can avoid breaking things in `CI`, and have a tighter feedback loop as we're developing and committing our code.

This is made possible because `Jest` has the find related test feature. If it didn't, then in a huge code base, you could do the same thing, but it would take you about two or three minutes to wait for all of your tests to run before your commit actually goes through, depending on the size of your code base.

It's really nice that `Jest` can just find the tests that are related to the files that you're changing, so you can get that feedback loop as tight as possible. Then you can run all the tests in continuous integration as you should.