Test React Components with Jest and react-testing-library

Transcripts for Kent C. Dodds (https://egghead.io/instructors/kentcdodds) course on egghead.io (https://egghead.io/courses/test-react-components-with-jest-and-react-testing-library).

Description

If you want to ship your applications with confidence—and of course you do—you need an excellent suite of automated tests to make absolutely sure that when changes reach your users, nothing gets broken. To get this confidence, your tests need to realistically mimic how users actually use your React components. Otherwise, tests could pass when the application is broken in the real world.

In this course, we'll write a series of render methods and run a range of tests to see how we can get the confidence we're looking for, without giving up maintainability or test run-speed.

Render a React component for testing

Kent C Dodds: [00:00] Here we have a basic React component called `FavoriteNumber`. That renders this `<label htmlFor="favorite-number">` and an `<input>` that is `type="number"`.

[00:09] When that number is changed, our `handleChange` callback will be called. That will set our state to indicate that the number has been entered and what that number is.

[00:17] Then we'll calculate that number's validity based off of the `min` and the `max`. If it's valid, then we just won't render anything extra. If it's invalid, then we'll render out `<div>The number is invalid</div>`.

favorite-number.js

```
handleChange = event => {
  this.setState({numberEntered: true, number:
Number(event.target.value)})
}
render() {
  const {number, numberEntered} = this.state
  const {min, max} = this.props
  const isValid = !numberEntered || (number >=
min && number <= max)
  return (
    <div>
      <label htmlFor="favorite-number">Favorite
Number</label>
      <input
        id="favorite-number"
        type="number"
        value={number}
        onChange={this.handleChange}
      />
      {isValid ? null : (
        <div data-testid="error-message">The
number is invalid</div>
      )}
    </div>
  )
}
```

[00:28] Let's go ahead and write a basic test for this in `react-dom.js`. I'll just say `test('renders a number input with a label "Favorite Number"')` I'm going to need to `import {FavoriteNumber} from '../favorite-number'` and then we'll render that. Because we're rendering it using JSX, we're going to need to `import React from 'react'`.

[00:51] We're going to want to use `ReactDOM.render` to render this to a `<div>`. Let's go ahead and import `import ReactDOM from 'react-dom'` and then we'll need to create that `<div>`.

[01:03] We'll make our `const div = document.createElement('div')`.

react-dom.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import {FavoriteNumber} from '../favorite-
number'

test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)
  console.log(div.innerHTML)
})
```

Next, let's go ahead and `console.log(div.innerHTML)` and see what that output is. I'm running my test over here and I see

Console Output

```
<div><label for="favorite-number">Favorite
Number</label>
<input id="favorite-number" type="number"
value="0"></div>
```

[01:19] Cool, so let's go ahead and add a couple of assertions here. We'll `expect(div.querySelector('input').type).toBe('number')` and `expect(div.querySelector('label').textContent).toBe('Favorite Number')`.

react-dom.js

```
test('renders a number input with a label "Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)

expect(div.querySelector('input').type).toBe('number')

expect(div.querySelector('label').textContent).toBe('Favorite Number')
})
```

[01:38] That gets our test passing. Let's just go ahead and make sure that our test can fail. It fails *stupendously*! So we know our assertions are running.

[01:47] This is the most basic React component test. We simply `import React`, and `import ReactDOM`, and the component that we're going to render. We create a `<div>` to render our component to. Then we use that `<div>` to query around the document, so that we can make assertions based off of what is rendered for our component.

Use jest-dom for improved assertions

Kent C Dodds: [00:00] Here we have a simple test for our `FavoriteNumber` component, and we have two assertions to make sure that the `input` and `label` are being rendered correctly.

jest-dom.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import {FavoriteNumber} from '../favorite-
number'

test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)

expect(div.querySelector('input').type).toBe('nu
mber')

expect(div.querySelector('label').textContent).t
oBe('Favorite Number')
})
```

[00:08] If we were to make a mistake here and type-o the "i" out of `expect(div.querySelector('input')`, we're going to get an error that says, *TypeError, cannot read property 'type' of null*.

[00:16] Now, that's not exactly the most helpful error message at all when you have to inspect things to figure out what exactly is wrong. It would be nice if we could get an assertion that could be more helpful when something goes wrong.

[00:26] There's a library called `jest-dom` that we can use to extend `expect` so we can add some assertions that are specific to DOM nodes.

[00:35] Let's go ahead and use this. I have it installed in the project already. I'm going to `import {toHaveAttribute} from 'jest-dom'`. With that, I can add `expect.extend({toHaveAttribute})`. Then I can remove the `.type` and instead say, `.toHaveAttribute('type', 'number')`.

```
import {toHaveAttribute} from 'jest-dom'
...

expect.extend({toHaveAttribute})

test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)

expect(div.querySelector('input')).toHaveAttribu
te('number')

expect(div.querySelector('label').textContent).t
oBe('Favorite Number')
})
```

[00:55] Now if I save this, the error message will be a little bit more helpful. It says, *received value must be an HTMLElement or an SVGElement. Received: null*.

[01:05] That helps me narrow down exactly what I should be looking for because it's expecting to receive a certain type that it didn't actually expect. I can fix my `querySelector` and the

assertion passes.

[01:16] Also, if I were to make a mistake here by calling 'number' as 'numer' instead, I'm going to see a more helpful error message as well, indicating that when we called `element.getAttribute('type')`, we expected it to equal a `"numer"`, but it actually equals a `"number"`.

[01:29] `jest-dom` gives us some really helpful assertions that we can use in our tests when we're dealing with DOM nodes with React. Let's fix that '"numer"` typo here. We can use one of the assertions for this one as well.

[01:40] There's a `{toHaveTextContent}` that we can also import from `jest-dom`. We'll just add that to our extensions here in `expect.extend`. Then we'll remove that `.textContent` and replace it with `toHaveTextContent`, which will improve things with our error messages here, as well.

```
import {toHaveAttribute, toHaveTextContent} from
'jest-dom'
...

expect.extend({toHaveAttribute,
toHaveTextContent})

test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)

expect(div.querySelector('input')).toHaveAttribu
te('number')

expect(div.querySelector('label')).toHaveTextCon
tent('Favorite Number')
})
```

[01:54] Now, we can actually simplify this. It would be really annoying to have to add `expect.extend` in every single test for every assertion that we want to have in our project.

[02:03] What we're going to do is `jest-dom` exposes a module that we can `import` called `jest-dom/extend-expect`, and it will call `expect.extend` for every assertion that it has available automatically for us. We can get rid of that `expect.extend` and our tests are still passing.

```
import 'jest-dom/extend-expect'
...

// REMOVED expect.extend({toHaveAttribute,
toHaveTextContent})

test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)

expect(div.querySelector('input')).toHaveAttribu
te('number')

expect(div.querySelector('label')).toHaveTextCon
tent('Favorite Number')
})
```

[02:21] Now, this is something that you would normally put in your setup files configuration with Jest. Rather than importing this file into every single test of my code base, I'd probably use the setup files configuration with Jest, but we'll go ahead and leave it there for now.

Use dom-testing-library to write more maintainable React tests

Kent C Dodds: [00:00] We have some pretty good assertions here ensuring that the input's type is `"Number"` and the `label` says `"Favorite Number"`, but I could actually break my application and my test wouldn't be able to tell me about that.

dom-testing-library.js

```
test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)

expect(div.querySelector('input')).toHaveAttribu
te('number')

expect(div.querySelector('label')).toHaveTextCon
tent('Favorite Number')
})
```

[00:11] If I were to go down here in the `label` and change the `htmlFor` to maybe a typo missing the t, then I save that and I'll open up my test. My tests are still passing. That's because, the `label` does still have the `"Favorite Number"` and the `input` is still of type `"number"`.

favorite-number.js

```
return (
  <div>
    <label htmlFor="favorie-number">Favorite
Number </label> <!-- typo in htmlFor -->
    <input
      id="favorite-number"
      type="number"
      value={number}
      onChange={this.handleChange}
    />
    {isValid ? null : <div>The number is
invalid</div>}
  </div>
)
```

[00:28] None of my assertions are failing, but my `input` is no longer associated with that `label`. That's an *accessibility problem*. If somebody were to have click on the `label`, I wouldn't focus in the `input` and all the other accessibility features that are associated with having a `label` pointing to a particular `input`.

[00:44] That's an important aspect of our application. It would be nice if we could also make assertions for that as well. In addition, if I were to start adding inputs and other labels inside of the same component, I would have to start doing some interesting things to make sure that I'm querying the right `label` and the right `input` in my test.

[01:01] It would be really nice, if I could actually get the `input` by its `label`. If I could say, "Oh, get me an `input` that has the `label` 'favorite-number'," then that would ensure that I have a `label` that says `favorite-number`, and that it's associated to the `input`, and I can make assertions on the `input`.

[01:16] What we can actually do this with a library called `dom-testing-library`. I'm going to `import {queries} from 'dom-testing-library'`. With that, I'm going to say `const input = queries.getByLabelText`. I'll pass my `div` for where I should be searching for the `label` text.

dom-testing-library.js

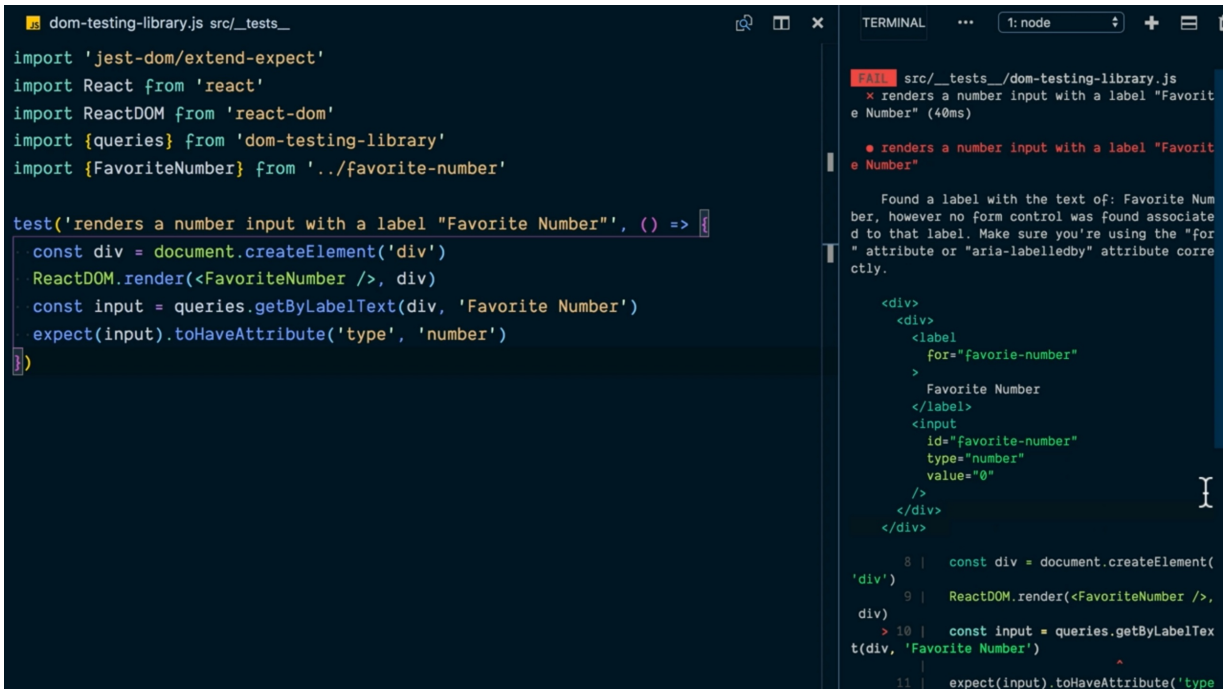
```
import {queries} from 'dom-testing-library'

test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)
  const input = queries.getByLabelText(div,
'Favorite Number')

expect(div.querySelector('input')).toHaveAttribu
te('number')

expect(div.querySelector('label')).toHaveTextCon
tent('Favorite Number')
})
```

[01:34] Then, I'll pass favorite number as my `label` text. Then, I can make my assertion about the `input`, so I could say `expect(input).toHaveAttribute('type', 'number'). I can get rid of this`label`assertion, because I'm basically making that`label`assertion by trying to get an`input` that is labeled "Favorite Number".

```
test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)
  const input = queries.getByLabelText(div,
'Favorite Number')
  expect(input).toHaveAttribute('number')
})
```

[01:55] We'll get rid of that `div.querySelector('label')`. If I open up my test, I can see I'm getting a test failure. Here, it says, *Found a label with the text of: Favorite Number, however no form control was found associated to that label.*

[02:07] It gives some information about how we can ensure that a `label` is associated with an `input`, and here, it's also outputting what the DOM looks like at this point. We cans see that our `for` attribute says `favorie-number` instead of `favorite-number`, and our `input` has the `id` of `favorite-number`.



[02:23] We can see that there is a mismatch. We can go ahead and fix that by adding the t to `favorie-number`. Save our file and our test run and pass. In review, to accomplish this, we imported `queries` from `dom-testing-library`. We use `queries` to `getByLabelText` inside of this `div` and the `label` text of `"Favorite Number"`.

```
test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)
  const input = queries.getByLabelText(div,
'Favorite Number')
  expect(input).toHaveAttribute('number')
})
```

[02:44] That got us the `input` that's associated with the `label` text, and we can make assertions on that `input`. Let's go ahead and make a couple refactors here. First of all, we're mostly concerned about the user being able to interact with our component and the user doesn't actually care about the casing.

[02:59] If somebody were to come in and change this from `Favorite Number` to `Favorite number`, the user wouldn't really care all that much, but our test is going to break.

[03:09] Let's go ahead and make this a little bit more resilient. We'll use a Regex instead. We'll say the case doesn't matter. Then, we can just lower case everything. As the casing changes, our test continue to pass. This is all the user really cares about anyway.

```
test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)
  const input = queries.getByLabelText(div,
/favorite number/i)
  expect(input).toHaveAttribute('number')
})
```

[03:22] Another thing that we can do with `dom-testing-library` is we can use the `getQueriesForElement`. We can say `const {getByLabelText} = getQueriesForElement(div)`. We can just call `getByLabelText` and pass the `label` text that we care about. That query will be scoped down to this `div`. We can get rid of `queries` up here in the `import`.

```
import {getQueriesForElement} from 'dom-testing-
library'

test('renders a number input with a label
"Favorite Number"', () => {
  const div = document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, div)
  const {getByLabelText} =
getQueriesForElement(div)
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('number')
})
```

[03:45] With that, we're ensuring that the `input` has a `label` `favorite number`, and if that relationship between the `input` and its `label` is ever broken, then our test will break as well. Giving us more confidence that our application is working the way that these are expects to do.

Use react-testing-library to render and test React Components

Kent C Dodds: [00:00] There are a couple things in here that would be pretty nice to use across all of our tests of our React components. Let's go ahead and make a simple function called `render` It's going to take our `ui`, so our React elements.

react-testing-library.js

```
function render(ui) {

}
```

[00:12] Then we're going to create our `div` and `render` and get our `getQueriesForElement` inside of this `render` function. I'm going to go ahead and change this.

[00:20] We'll call this `container` just to make that a little bit more specific. We're going to `return` an object called `container`. Then we actually want all of the `queries`. We'll spread the `queries` across here.

react-testing-library.js

```
function render(ui) {
  const container =
document.createElement('div')
  ReactDOM.render(<FavoriteNumber />, container)
  const queries =
getQueriesForElement(container)
  return {
    container,
    ...queries,
  }
}
```

[00:35] Then I can call `render` with my `<FavoriteNumber/>` as my `ui`. We'll also change `<FavoriteNumber/>` in the `ReactDOM.render` to `ui`. What I'm going to get back is an object that I can destructure and get all the stuff that I really care about.

react-testing-library.js

```
function render(ui) {
  const container =
document.createElement('div')
  ReactDOM.render(ui, container) // replaced
<FavoriteNumber/> with ui
  ...
}

test('renders a number input with a label
"Favorite Number"', () => {
  const {getByLabelText} =
render(<FavoriteNumber />)
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('type',
'number')
})
```

[00:49] I can get my `container` if I want. I can `getByText`, or whatever else I want to get from these `queries`, from `dom-testing-library`, that are all pre-bound to this `container`.

[01:02] Now I can use this `render` method for all the tests that are trying to `render` a React component. It makes my tests really nice and slim.

[01:09] This `render` method has already actually been written as an open source library. We can actually `import {render} from "react-testing-library"` I already have that installed in the project.

[01:21] Here, we can get rid of `render` entirely. Get rid of `react-dom`. We don't need that any more,
and `dom-testing-library`. That's all handled by `react-`

`testing-library` with `render`. Then I'll hit save. I pop open my tests. They continue to pass.

react-testing-library.js

```
import 'jest-dom/extend-expect'
import React from 'react'
import {render} from 'react-testing-library'
import {FavoriteNumber} from '../favorite-
number'

test('renders a number input with a label
"Favorite Number"', () => {
  const {getByLabelText} =
render(<FavoriteNumber />)
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('type',
'number')
})
```

Avoid Memory leaks using react-testing-library's cleanup function

Kent C Dodds: [00:00] This `render` method actually does a couple of extra things for us that our original `render` method didn't do. One of those things is, it actually renders our component to `document.body`.

[00:09] This ensures that React's event system will work properly. If I go ahead and `console.log(document.body.outerHTML)` and open up my tests, I'm going to see `body` is rendered and it has a `div` inside. That's our container.

Console Output

```
console.log src/__tests__/react-testing-
library.js:8
<body><div><div><label for="favorite-
number">FavoriteNumber</label><input
id="favorite-number" type="number" value="0"/>
</div></div></body>
```

[00:23] Then it has a `div` from our `FavoriteNumber` element. Because of this, if we have multiple of these tests testing various features of the `FavoriteNumber`, eventually, this `document.body` is going to be filled up with multiple instances of our `FavoriteNumber` component.

[00:38] We need to make sure that we unmount this component. One of the things that the `render` method gives back to us is `unmount`. Here, we can call `unmount`.

react-testing-library.js

```
test('renders a number input with a label
"Favorite Number"', () => {
  const {getByLabelText, unmount} =
render(<FavoriteNumber />)
  console.log(document.body.outerHTML)
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('type',
'number')
  unmount()
  console.log(document.body.outerHTML)
})
```

If I `console.log(document.body.outerHTML)` after the `unmount` has taken place, then we get our initial `console.log()` here, and we get our unmounted component here.

Console Output

```
console.log src/__tests__/react-testing-
library.js:8
<body><div><div><label for="favorite-
number">FavoriteNumber</label><input
id="favorite-number" type="number" value="0"/>
</div></div></body>

console.log src/__tests__/react-testing-
library.js:12
<body><div></div></body>
```

[00:57] Now, that would be really annoying to have to do all over the place, so `react-testing-library` exposes a `cleanup` function that we can use. We no longer need `unmount` here and we can replace this with `cleanup`.

react-testing-library.js

```
import {render, cleanup} from 'react-testing-library'

test('renders a number input with a label "Favorite Number"', () => {
  const {getByLabelText} = render(<FavoriteNumber />)
  console.log(document.body.outerHTML)
  const input = getByLabelText(/favorite number/i)
  expect(input).toHaveAttribute('type', 'number')
  cleanup()
  console.log(document.body.outerHTML)
})
```

[01:08] Now, that also cleans up our `container` as well, which is good, but putting `cleanup` after every one of our tests would also not be super fun, so we can do an `afterEach()`.

[01:16] We'll `cleanup` after each one of our tests. Then we can get rid of this `console.log in`test`. We'll add a`console.log`up in`afterEach` as well. That's working just fine.

[01:26] In addition, if we aren't doing any extra work inside of this `afterEach()`, we can actually just do `afterEach(cleanup)`, just to shorten things up a little bit.

```
afterEach(cleanup)

test('renders a number input with a label
"Favorite Number"', () => {
  const {getByLabelText} =
render(<FavoriteNumber />)
  console.log(document.body.outerHTML)
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('type',
'number')
})
```

[01:33] Also, to make things even easier, we can actually `import 'react-testing-library/cleanup-after-each'`, and we can get rid of the `cleanup` import and the `afterEach()` call right here. That will take care of unmounting and removing our `container` from the DOM after each one of our tests.

```
import 'react-testing-library/cleanup-after-
each'
import {render} from 'react-testing-library'

test('renders a number input with a label
"Favorite Number"', () => {
  const {getByLabelText} =
render(<FavoriteNumber />)
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('type',
'number')
})
```

Debug the DOM state during tests using react-testing-library’s debug function

Kent C Dodds: [00:00] As I am testing my component, it would be really helpful if I could get some insight into what the DOM looks like during any given point of my test, so React's testing library's `render` method exposes a utility called `debug`. This `debug` method can be called at any point in time and it will log out the `document.body` which contains our `container`, which contains the DOM for the component that we're testing.

react-testing-library.js

```
test('renders a number input with a label
"Favorite Number"', () => {
  const {getByLabelText, debug} =
render(<FavoriteNumber />)
  debug()
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('type',
'number')
})
```

Console Output

```
console.log node_modules/react-testing-library-
dist/index.js:57
<body>
  <div>
    <div>
      <label for="favorite-number">
        Favorite Number
      </label>
      <input
        id="favorite-number"
        type="number"
        value="0"
      >
    </div>
  </div>
</body>
```

[00:22] This is all highlighted and looks great in my terminal so I can figure out what's going on. As I interact with the component, I can run `debug` again at any point in my test to see what the DOM looks like at that point in my test.

react-testing-library.js

```
test('renders a number input with a label
"Favorite Number"', () => {
  const {getByLabelText, debug} =
render(<FavoriteNumber />)
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('type',
'number')
  debug()
})
```

[00:33] If my DOM output is really big, then React testing library will actually truncate it. If you want to focus on a particular node, then you can take that node and pass it to `debug`, and it will only log out that particular node.

react-testing-library.js

```
test('renders a number input with a label
"Favorite Number"', () => {
  const {getByLabelText, debug} =
render(<FavoriteNumber />)
  const input = getByLabelText(/favorite
number/i)
  expect(input).toHaveAttribute('type',
'number')
  debug(input)
})
```

Test React Component Event Handlers with fireEvent from react-testing-library

Kent C Dodds: [00:00] Our `FavoriteNumber` component here has some validation logic. If the number `isValid`, then it doesn't render anything here. Otherwise, if that number is invalid, then we're going to render out, `The number is invalid`

favorite-number.js

```
render() {
  const {number, numberEntered} = this.state
  const {min, max} = this.props
  const isValid = !numberEntered || (number >=
min && number <= max)
  return (
    <div>
      <label htmlFor="favorite-number">Favorite
Number</label>
      <input
        id="favorite-number"
        type="number"
        value={number}
        onChange={this.handleChange}
      />
      {isValid ? null : (
        <div data-testid="error-message">The
number is invalid</div>
      )}
    </div>
  )
}
```

[00:10] It determines that validity based on whether that number is between a `min` and a `max`. That defaults to `1` and `9`. Any time the user changes the value of this `input`, then we have this

`handleChange`, which will take that `event.target` and take the `value` property and assign that to the `number` state.

favorite-number.js

```
handleChange = event => {
  this.setState({numberEntered: true, number:
Number(event.target.value)})
}
```

[00:29] If I were a user assigned to manually test this, what I would do is render that component to the page. I would make a change to the `input` and verify that this "is invalid" message shows up. That's exactly what our test is going to do.

[00:42] Let's go ahead. We'll get our `getByLabelText` so that we can get that `input`. We'll assign that to `render(<FavoriteNumber/>)`.

[00:53] We could provide a `min` and a `max` here, but I'll go ahead and rely on the defaults. That's part of the API of our component anyway.

state.js

```
import 'jest-dom/extend-expect'
import 'react-testing-library/cleanup-after-
each'
import React from 'react'
import {render} from 'react-testing-library'
import {FavoriteNumber} from '../favorite-
number'

test('entering an invalid value shows an error
message', () => {
  const {getByLabelText} =
render(<FavoriteNumber />)
})
```

[00:59] Then we'll go ahead and get our `input` from `getByLabelText(/favorite number/i)`. We'll do a regex here because the user doesn't care about the case and neither should our test.

[01:10] Next, we need to fire a change event on this `input`. `react-testin1g-library` exposes a useful utility called `fireEvent`. Then we can use `fireEvent.change` to fire a `change` event on our `input`.

state.js

```
import {render, fireEvent} from 'react-testing-library'

test('entering an invalid value shows an error message', () => {
  const {getByLabelText} = render(<FavoriteNumber />)
  const input = getByLabelText(/favorite number/i)
  fireEvent.change(input)
})
```

[01:24] Our `change` handler takes the `event` and gets the `target.value`, so we need to set the `target.value` to a number that's outside of this min-max range. Let's go ahead. We'll set `target: {value: 10}`, a number outside of the min-max range.

state.js

```
test('entering an invalid value shows an error message', () => {
  const {getByLabelText} = render(<FavoriteNumber />)
  const input = getByLabelText(/favorite number/i)
  fireEvent.change(input, {target: {value: 10}})
})
```

[01:40] Then let's go ahead and take a look at what the DOM looks like. I'm going to pull out `debug`. We'll call `debug` right before we do anything and `debug` right after we do things.

state.js

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, debug} =
render(<FavoriteNumber />)
  debug()
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  debug()
})
```

[01:49] Pop open our terminal. We'll see our `label` and `input` our render here. We get our `label` and `input`. The number is invalid because our number is outside of the range.

Console Output

```
<body>
  <div>
    <div>
      <label for="favorite-number">
        Favorite Number
      </label>
      <input
        id="favorite-number"
        type="number"
        value="10"
      >
      <div>
        The number is invalid
      </div>
    </div>
  </div>
</body>
```

[01:59] In review, to fire a `change` event on an `input`, you can use the `fireEvent` from `react-testing-library`. You fire the `change` event on the `input`. Then you provide anything that you want to have assigned to that event.

state.js

```
fireEvent.change(input, {target: {value: 10}})
```

[02:12] If you provide a `target`, then these values will actually be assigned to the node that you're firing the event on as well. `fireEvent` supports all events that you can imagine.

[02:21] We have `fireEvent` for click and mouse up, down, over, and out, copy, paste... All of the regular events that you're used to working within the DOM.

Assert rendered text with react-testing-library

Kent C Dodds: [00:00] There are a couple of things we can do to make our assertion that this is rendered properly. We'll go ahead and I'll show you a couple of those.

state.js

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, debug} =
render(<FavoriteNumber />)
  debug()
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  debug()
})
```

[00:07] We'll have a `container` that we get from our `render`. We can just say `expect(container).toHaveTextContent()`. We'll just put a regex, `/the number is invalid/i`. That would work. That passes our test.

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, debug} =
render(<FavoriteNumber />)
  debug()
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  debug()
  expect(container).toHaveTextContent(/the
number is invalid/i)
})
```

[00:21] Another thing we could do is we could use the `getByText` here. We could just simply say, `getByText(/the number is invalid/i)`. Actually, if `getByText` can't find a node with that text -- let's change this to `/th number is invalid/i` -- then it's going to throw an error indicating that it's unable to find an element with that text.

[00:39] That basically is an assertion there. If you want to make it look like more of an assertion, then we could `expect(getByText(/the number is invalid/i)).toBeTruthy()`,
or we could also say, `.toBeInTheDocument()`

[00:50] Really, the assertion happens here because that's where an error will be thrown if it can't find an element with that text. That's another way that we could verify that this text is being rendered.

[01:00] One last way I want demonstrate here is being able to select nodes like this `{isValid ? null : <div>The number is invalid</div>}`. Finding an `input` is pretty easy because it

normally should be associated with the `label`. We have `getByLabelText`.

favorite-number.js

```
<div>
  <label for="favorite-number">Favorite
Number</label>
  <input
    id="favorite-number"
    type="number"
    value={number}
    onChange={this.handleChange}
  />
  {isValid ? null : <div>The number is
invalid</div>}
</div>
```

[01:11] Finding a `button` that has the text "submit" should be easy because you can use `getByText` to get the text of the `button`, but finding arbitrary divs with arbitrary messages in there, that might be a little bit more difficult.

[01:22] One utility that `react-testing-library` exposes for just such a case is you can add a `data-testid` attribute to your element and give it any unique identifier for this component. We can say, `"error-message"` for example.

```
<div>
  <label for="favorite-number">Favorite
Number</label>
  <input
    id="favorite-number"
    type="number"
    value={number}
    onChange={this.handleChange}
  />
  {isValid ? null : <div data-testid="error-
message">The number is invalid</div>}
</div>
```

[01:38] With that, we can `getByTestId`. We can say `expect(getByTestId('error-message')).toHaveTextContent(/the number is invalid/i)`. That will also work. Let's get rid of these debugs so we can see our full test output.

state.js

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, debug, getByTestId} =
render(<FavoriteNumber />)
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  expect(getByTestId('error-
message')).toHaveTextContent(
    /the number is invalid/i,
  )
})
```

[01:56] Those are the various ways you can find text that's rendered in your component. Whether you use `toHaveTextContent` on your entire `container` or specifically with a `getByTestId` to target a specific element or if you try to use `getByText`, each of them comes with their own tradeoffs.

[02:12] I'm going to go ahead and leave it with the `getByTestId('error-message')` because I feel like that's a little bit more explicit.

Test prop updates with react-testing-library

Kent C Dodds: [00:00] We've got our `FavoriteNumber` that we're rendering here. We get the `input`. We fire a `change` event on the `input`, and then we can make an assertion that this error message is showing up.

Console Output

```
<body>
  <div>
    <div>
      <label for="favorite-number">
        Favorite Number
      </label>
      <input
        id="favorite-number"
        type="number"
        value="10"
      >
      <div>
        The number is invalid
      </div>
    </div>
  </div>
</body>
```

state.js

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, debug, getByTestId} =
render(<FavoriteNumber />)
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  expect(getByTestId('error-
message')).toHaveTextContent(
    /the number is invalid/i,
  )
  debug()
})
```

[00:09] Now, what if we wanted to change the props, re-render the component to make this value of `10` inside the `min` and `max` range? If we set the `max` prop to `10`, for example, then this `render` method should run again and our `isValid` should now be true in this case. That will make it so that this message error message is no longer rendering.

favorite-number.js

```
static defaultProps = {min: 1, max: 10} // max
changed to 10

...

render() {
  ...
  const isValid = !numberEntered || (number >=
min && number <= max)
  return (
    <div>
      <label htmlFor="favorite-number">Favorite
Number</label>
      <input
        id="favorite-number"
        type="number"
        value={number}
        onChange={this.handleChange}
      />
      {isValid ? null : (
        <div data-testid="error-message">The
number is invalid</div>
      )} <!-- no longer renders  -->
    </div>
  )
}
```

[00:31] To re-render a component with `react-testing-library`, you get another utility here in this object. We are going to de-structure `rerender`. Here with `rerender`, I am going to call `rerender` with our same component, our `FavoriteNumber`.

[00:46] We can pass any different props that we want. When you say `max={10}`, and then I will put a `debug()` after that.

prop-updates.js

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, debug, getByTestId,
rerender} = render(
    <FavoriteNumber />
  )
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  expect(getByTestId('error-
message')).toHaveTextContent(
    /the number is invalid/i,
  )
  debug()
  rerender(<FavoriteNumber max={10} />)
  debug()
})
```

[00:53] Here we have a before where the value is 10, and we see the number is invalid. Then we have the after where the value is still 10, so the error message goes away. Now we can make an assertion that that error message has gone away.

Output Before

```
<body>
  <div>
    <div>
      <label for="favorite-number">
        Favorite Number
      </label>
      <input
        id="favorite-number"
        type="number"
        value="10"
      >
      <div>
        The number is invalid
      </div>
    </div>
  </div>
</body>
```

Output After

```
<body>
  <div>
    <div>
      <label for="favorite-number">
        Favorite Number
      </label>
      <input
        id="favorite-number"
        type="number"
        value="10"
      >
    </div>
  </div>
</body>
```

[01:05] In review, with `react-testing-library`, when you call the `render` function, you get a whole bunch of utilities. One of those is a `rerender` function which you can use to `rerender` that component with different props.

Assert that something is NOT rendered with react-testing-library

Kent C Dodds: [00:00] Now that we've rerendered our `FavoriteNumber` with a different prop here, that value of `10` is now within the min-max range, and that error message is no longer there, we want to make an assertion that the error message no longer appears.

state.js

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, debug, getByTestId,
rerender} = render(
    <FavoriteNumber />
  )
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  expect(getByTestId('error-
message')).toHaveTextContent(
    /the number is invalid/i,
  )
  debug()
  rerender(<FavoriteNumber max={10} />)
  debug()
})
```

[00:14] What we can do is take this that `expect` here, and we basically want to `expect` that `getByTestId('error-message')` returns a `null` value, so it doesn't return any node. We can say `toBeNull()`. Let me get rid of these debugs, and I'll save that.

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, debug, getByTestId,
rerender} = render(
    <FavoriteNumber />
  )
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  expect(getByTestId('error-
message')).toHaveTextContent(
    /the number is invalid/i,
  )
  rerender(<FavoriteNumber max={10} />)
  expect(getByTestId('error-
message')).toBeNull()
})
```

[00:29] We're going to get a failing test. What's happening here, *Unable to find an element by: [data-testid="error-message"]*, and that error is happening right here when it's trying to `getByTestId`.

[00:40] The *get* queries will throw an error when it can't find whatever it's looking for. That applies to `getByLabel`, `getByText`, all of the *get* queries. In our case, we want to find something that we know is not there and make sure that it's not there.

[00:54] In our case, instead of using a *getBy* query, we're going to use a `queryByTestId`. There is an associated query function for all of the get functions. Instead of `getByTestId`, we'll `queryByTestId`.

```
test('entering an invalid value shows an error
message', () => {
  const {getByLabelText, queryByTestId,
getByTestId, rerender} = render(
    <FavoriteNumber />
  )
  const input = getByLabelText(/favorite
number/i)
  fireEvent.change(input, {target: {value: 10}})
  expect(getByTestId('error-
message')).toHaveTextContent(
    /the number is invalid/i,
  )
  rerender(<FavoriteNumber max={10} />)
  expect(queryByTestId('error-
message')).toBeNull()
})
```

[01:07] The only real difference here is that `queryByTestId` will return `null`, whereas `getByTestId` will throw an error if they can find an element that matches the query. With that, I can save, and my test is passing.

[01:20] In review, to assert that something does not exist in your test, you use a *queryBy* function rather than a *getBy* function, because the *getBy* will throw an error if it can't find the element matching the query. The *queryBy* will simply return `null`, and you can make an assertion that it does.

Test accessibility of rendered React Components with jest-axe

Kent C Dodds: [00:00] This `Form` is not accessible. The reason it's not accessible is because this `input` is missing a `label`. Even though it has a `placeholder` here, it needs a `label` so assistive technologies can help out users who are using your application.

a11y.js

```
import 'jest-dom/extend-expect'
import 'react-testing-library/cleanup-after-
each'
import React from 'react'
import {render} from 'react-testing-library'

function Form() {
  return (
    <form>
      <input placeholder="username"
name="username" />
    </form>
  )
}

test('the form is accessible', () => {})
```

[00:14] To start testing for accessibility in our `Form`, we're going to go ahead and `render(<Form/>)` in our `test`. We're going to get the `container` from that function call. Then if we `console.log(container.innerHTML)`, we can see that the HTML renders the `input` with no `label`.

```
test('the form is accessible', () => {
  const {container} = render(<Form />)
  console.log(container.innerHTML)
})
```

[00:30] We can pass this HTML to a tool called `axe-core` which will give us a report of the accessibility violations in that HTML. There's a Jest-specific library called `jest-axe` that we can use to interact with this in a nice way with Jest.

[00:45] I'm going to `import {axe} from 'jest-axe'`. Instead of console.logging that `container.innerHTML`, I'm going to `axe` it. This is an asynchronous operation. It returns a promise. I can `await` that to get my `results`.

```
test('the form is accessible', () => {
  const {container} = render(<Form />)
  const results = await axe(container.innerHTML)
})
```

[01:02] I'll need to turn this test into an `async` test. Now if I `console.log(results)`, I'm going to get a bunch of violations here and a lot of information that I can't really make a whole lot of sense of in my terminal here.

[01:16] I want to make an assertion that will throw a nice, readable error when I have accessibility violations. I could do something like `expect(results.violations).toHaveLength(0)`, but that still wouldn't be super-helpful.

[01:33] `jest-axe` also exposes an expect extension that I can use

to have a much more helpful error message here. I'm going to also `import {toHaveNoViolations}`. I'll then call `expect.extend(toHaveNoViolations)`. Then I can pass my `results` to `toHaveNoViolations` and my error message is a lot more helpful.

```
import {axe, toHaveNoViolations} from 'jest-axe'

expect.extend(toHaveNoViolations)

...

test('the form is accessible', async () => {
  const {container} = render(<Form />)
  const results = await axe(container.innerHTML)
  expect(results).toHaveNoViolations()
})
```

[01:54] It tells me exactly what the node was that is causing that violation. It gives me some helpful information to go look into to find out why I'm experiencing that violation.

Console Output

```
Expected the HTML found at $('input') to have no
violations:

<input placeholder="username" name="username"/>

Received:

"Form elements must have labels (label)"

Try fixing it with this help:
https://dequeuniversity.com/rules/axe/3.1/label?
application=axeAPI
```

[02:05] If I go ahead and fix this by adding a `label` with an `htmlFor="username"`. Inside the `label` I write, `Username`. On my `input`, I give it `id="username"` so that `id` is associated with the `htmlFor`. That associates my `label` to that `input`. If I save this, then I'm going to get a passing test.

a11y.js

```
function Form() {
  return (
    <form>
      <label htmlFor="username">Username</label>
      <input id="username"
placeholder="username" name="username" />
    </form>
  )
}
```

[02:24] One thing I can do to clean this up is along with these two imports -- which I should be putting in a setup file -- I can also `import 'jest-axe/extend-expect'`. Then I don't need to `import` this `toHaveNoViolations` into every file or call `expect.extend`.

```
import 'jest-dom/extend-expect'
import 'react-testing-library/cleanup-after-
each'
import 'jest-axe/extend-expect'
```

[02:40] `axe-core` supports a lot more than reporting just violations of labels and inputs. You might consider re-running axe any time the HTML of your component changes.

Mock HTTP Requests with jest.mock in React Component Tests

Kent C Dodds: [00:00] Here we have a `form` where you can type in your name. You submit this `form` with this `loadGreeting`. It will make a request to this `loadGreeting` API with your name. Then it will `setState` with the `greeting` and it will render out the `greeting` in this `div` with the `data-testid`.

greeting-loader-01-mocking.js

```
import React, {Component} from 'react'
import {loadGreeting} from './api'

class GreetingLoader extends Component {
  inputRef = React.createRef()
  state = {greeting: ''}
  loadGreetingForInput = async e => {
    e.preventDefault()
    const {data} = await
loadGreeting(this.inputRef.current.value)
    this.setState({greeting: data.greeting})
  }
  render() {
    return (
      <form onSubmit=
{this.loadGreetingForInput}>
        <label htmlFor="name">Name</label>
        <input id="name" ref={this.inputRef} />
        <button type="submit">Load
Greeting</button>
        <div data-testid="greeting">
{this.state.greeting}</div>
      </form>
    )
  }
}
```

[00:18] Let's go ahead and write a test for this. We're going to `import React from 'react'`. We'll `import {render} from 'react-testing-library'`. We'll `import {GreetingLoader} from '../greeting-loader-01-mocking'`.

http-jest-mock.js

```
import 'jest-dom/extend-expect'
import 'react-testing-library/cleanup-after-
each'

import React from 'react'
import {render} from 'react-testing-library'
import {GreetingLoader} from '../greeting-
loader-01-mocking'
```

[00:34] We're going to write our `test` that says, `'loads greetings on click'` Then we can `render` the `GreetingLoader`. We need to get the name `input` and the Load Greeting `button`.

[00:49] We'll also need to get this greeting `div` by its `data-testid`. With all of those, we need to `getByLabelText`, `getByText`, and `getByTestId`. We can get our `nameInput`. That's `getByLabelText(/name/i)`.

[01:05] We'll get our `loadButton`. That's `getByText(/load/i)`. We'll set the `nameInput.value` to `'Mary'`.

[01:17] We'll `fireEvent`. Get that `fireEvent` from `react-testing-library`. We'll `.click` on the `loadButton`.

http-jest-mock.js

```
import {render, fireEvent} from 'react-testing-
library'

test('loads greetings on click', () => {
  const {getByLabelText, getByText, getByTestId}
= render(<GreetingLoader />)
  const nameInput = getByLabelText(/name/i)
  const loadButton = getByText(/load/i)
  nameInput.value = 'Mary'
  fireEvent.click(loadButton)
})
```

[01:25] When we click on this `loadButton`, it's going to submit the `form`, which will call this `loadGreetingForInput` function, then `e.preventDefault`. Then we'll wait for the `loadGreeting` to resolve. This is an asynchronous operation. We need to make our test asynchronous.

greeting-loader-01-mocking.js

```
loadGreetingForInput = async e => {
  e.preventDefault()
  const {data} = await
loadGreeting(this.inputRef.current.value)
  this.setState({greeting: data.greeting})
}
```

[01:39] We need to wait for this `greeting` to be loaded, so let's go ahead and import `wait` from `react-testing-library`. Then we will `await` and `wait` for an expectation that `getByTestId('greeting')` will `toHaveTextContent()`.

http-jest-mock.js

```
test('loads greetings on click', () => {
  const {getByLabelText, getByText, getByTestId}
= render(<GreetingLoader />)
  const nameInput = getByLabelText(/name/i)
  const loadButton = getByText(/load/i)
  nameInput.value = 'Mary'
  fireEvent.click(loadButton)
  await wait(() =>
expect(getByTestId('greeting')).toHaveTextConten
t())
})
```

[01:57] The `greeting` now, this `greeting`, right now, we're actually making an HTTP call to get what that `greeting` is going to be. We need to mock this out, so we're going to use Jest's mocking capabilities to mock out the API module.

[02:10] We can get a mock version of `loadGreeting` and have it resolve immediately to some value so we don't make an HTTP call in our unit test. Let's go ahead and use `jest.mock` to mock out `'../api'`.

[02:24] We'll return just the stuff that we need, so just this `loadGreeting`. That's going to be a `jest.fn` so we can keep track of how it's called.

```
jest.mock('../api', () => {
  return {
    loadGreeting: jest.fn()
  }
})
```

[02:34] Our mock implementation is going to take a `subject`. That's going to `Promise.resolve` to the same thing that our server would send back, so `{data: {greeting: }}`.

[02:45] We'll just have it say, `'Hi ${subject}'`. With that, we can expect that our `greeting` div has "Hi Mary". We want to get access to this `loadGreeting` mock function.

[02:58] Let's go ahead. We'll `import {loadGreeting as mockLoadGreeting} from '../api'` and then we can make assertions on that. We'll write `expect(mockLoadGreeting).toHaveBeenCalledTimes(1)` and `expect(mockLoadGreeting).toHaveBeenCalledWith('Mary')`.

```
import {loadGreeting as mockLoadGreeting} from
'../api'

jest.mock('../api', () => {
  return {
    loadGreeting: jest.fn(subject =>
      Promise.resolve({data: {greeting: `Hi
${subject}`}}),
    ),
  }
})

test('loads greetings on click', () => {
  const {getByLabelText, getByText, getByTestId}
= render(<GreetingLoader />)
  const nameInput = getByLabelText(/name/i)
  const loadButton = getByText(/load/i)
  nameInput.value = 'Mary'
  fireEvent.click(loadButton)
  await wait(() =>
expect(getByTestId('greeting')).toHaveTextConten
t())

expect(mockLoadGreeting).toHaveBeenCalledTimes(1
)

expect(mockLoadGreeting).toHaveBeenCalledWith('M
ary')
})
```

[03:23] We open up our test here. We've got a passing test. Let's make sure that this test can fail. Maybe these assertions aren't running or something.

[03:29] I'll just say, `.not` before calling `.toHaveBeenCalledTimes(1)` Great. Our test can fail. Our assertions are running.

[03:35] In review, what we had to do to test our `loadGreeting` component is we mocked the API call that we were going to make so that we could have a fake `loadGreeting` that would immediately resolve to something that the server would send back without actually having to make a server call.

```
jest.mock('../api', () => {
  return {
    loadGreeting: jest.fn(subject =>
      Promise.resolve({data: {greeting: `Hi
${subject}`}}),
    ),
  }
})
```

[03:51] We rendered our `GreetingLoader` getting the `getByLabelText` for our `nameInput`, the getByText for our `loadButton`,
and then `getByTestId` for the `greeting` message. We set the `nameInput` to some value. We set it to "Mary" and then `fireEvent`.click on the button.

[04:06] We waited for our `greeting` to be loaded and then expected that the `mockLoadGreeting` was called appropriately. We could actually move these above here because that function is actually called synchronously.

[04:18] We can make those assertions earlier, which is probably a good idea so we don't have to wait for this
if there's some sort of bug. We save this. Our test is still passing.

```
test('loads greetings on click', () => {
  const {getByLabelText, getByText, getByTestId}
= render(<GreetingLoader />)
  const nameInput = getByLabelText(/name/i)
  const loadButton = getByText(/load/i)
  nameInput.value = 'Mary'
  fireEvent.click(loadButton)

expect(mockLoadGreeting).toHaveBeenCalledTimes(1
)

expect(mockLoadGreeting).toHaveBeenCalledWith('M
ary')
  await wait(() =>
expect(getByTestId('greeting')).toHaveTextConten
t())
})
```

Mock HTTP Requests with Dependency Injection in React Component Tests

Kent C Dodds: [00:00] Our tests here are using the `jest.mock` API, so that we don't have to make HTTP calls in our tests, but there is an alternative to using `jest.mock` that actually works well for other environments as well, like if you're using React's storybook for example.

dependency-injection.js

```
jest.mock('../api', () => {
  return {
    loadGreeting: jest.fn(subject =>
      Promise.resolve({data: {greeting: `Hi
${subject}`}}),
    ),
  }
})
```

[00:13] We're going to refactor this to use a dependency injection model that will work both for our test as well as for a storybook. The first thing we're going to do is I'm going to take this `loadGreeting` mock function.

[00:24] We're going to remove that and I'll get rid of the `jest.mock` entirely. Then we'll get rid of the `import` from `'../api'`, and instead, we'll put a `mockLoadGreeting` right here.

```
// REMOVED import {loadGreeting as
mockLoadGreeting} from '../api'
// REMOVED jest.mock(...)

test('loads greetings on click', () => {
  const mockLoadGreeting = jest.fn(subject =>
    Promise.resolve({data: {greeting: `Hi
${subject}`}})
  )
  const {getByLabelText, getByText, getByTestId}
= render(<GreetingLoader />)
  const nameInput = getByLabelText(/name/i)
  const loadButton = getByText(/load/i)
  nameInput.value = 'Mary'
  fireEvent.click(loadButton)

expect(mockLoadGreeting).toHaveBeenCalledTimes(1
)

expect(mockLoadGreeting).toHaveBeenCalledWith('M
ary')
  await wait(() =>
expect(getByTestId('greeting')).toHaveTextConten
t())
})
```

[00:35] Our `GreetingLoader` is going to accept this as a prop, so we'll say `loadGreeting` is our `mockLoadGreeting`. That's all the changes that are needed for our test.

```
test('loads greetings on click', () => {
  const mockLoadGreeting = jest.fn(subject =>
    Promise.resolve({data: {greeting: `Hi
${subject}`}})
  )
  const {getByLabelText, getByText, getByTestId}
= render(<GreetingLoader loadGreeting=
{mockLoadGreeting} />)
  ...
})
```

[00:45] Now, let's take a look at our implementation in `greeting-loader-02-dependency-injection.js`. Right now, we're getting the `loadGreeting` from this module, but instead, we want to accept it from props. With that, if we run our test, they are passing.

greeting-loader-01-dependency-injection.js

```
import {loadGreeting} from './api'

class GreetingLoader extends Component {
  inputRef = React.createRef()
  state = {greeting: ''}
  loadGreetingForInput = async e => {
    e.preventDefault()
    const {data} = await
this.props.loadGreeting(this.inputRef.current.va
lue)
    this.setState({greeting: data.greeting})
  }
  ...
}
```

[00:56] The problem is now that in our application, if we want to use this `GreetingLoader`, we're going to have to pass the `loadGreeting` function as a prop, and that will be super annoying. Instead, we're going to actually leverage a nice feature from React `static defaultProps = {loadGreeting}`.

[01:15] If the `loadGreeting` prop is not supplied, it will default to the `loadGreeting` from our API just as we had before, but if it is provided, then it will use the one that's being provided.

greeting-loader-01-dependency-injection.js

```
class GreetingLoader extends Component {
  static defaultProps = {loadGreeting}
  inputRef = React.createRef()
  state = {greeting: ''}
  loadGreetingForInput = async e => {
    e.preventDefault()
    const {data} = await
this.props.loadGreeting(this.inputRef.current.va
lue)
    this.setState({greeting: data.greeting})
  }
  ...
}
```

[01:24] This is the format to dependency injection for React and it works really great for both our test environment as well as another environment like *React Storybook*, or in *Code Sandbox* where Jest mocking isn't supported.

[01:35] In review, to make this work, we remove the `jest.mock` call and moved our `mockLoadGreeting` into our `test` function right here. Then we pass that to our `GreetingLoader` as a prop.

http-jest-mock.js

```
test('loads greetings on click', () => {
  const mockLoadGreeting = jest.fn(subject =>
    Promise.resolve({data: {greeting: `Hi
${subject}`}})
  )
  const {getByLabelText, getByText, getByTestId}
= render(
    <GreetingLoader loadGreeting=
{mockLoadGreeting} />
  )
  ...
})
```

[01:46] Then we accepted that. Instead of using the `loadGreeting` from API directly, we use it from this.props, and had a `defaultProp` for the `loadGreeting`.

[01:55] Generally, I favor the `jest.mock` approach because it's more powerful and it doesn't require that you change your implementation, but this is really nice if you have an environment that doesn't support the Jest mocking capabilities.

Mock react-transition-group in React Component Tests with jest.mock

Kent C Dodds: [00:00] Here, we have this `HiddenMessage` component that will show its `children` in a `div` inside of this `Fade` component when you click on this toggle `button`. That `Fade` component is using `CSSTransition` from `react-transition-group`, which is an animation library. It will `Fade` in the `children` after 1,000 milliseconds.

hidden-message.js

```
import {CSSTransition} from 'react-transition-group'

function Fade({children, ...props}) {
  return (
    <CSSTransition {...props} timeout={1000} className="fade">
      {children}
    </CSSTransition>
  )
}

class HiddenMessage extends React.Component {
  state = {show: false}
  toggle = () => {
    this.setState(({show}) => ({show: !show}))
  }
  render() {
    return (
      <div>
        <button onClick={this.toggle}>Toggle</button>
        <Fade in={this.state.show}>
          <div>{this.props.children}</div>
        </Fade>
      </div>
    )
  }
}
```

[00:17] In our test, we don't want to wait 1,000 milliseconds before we can verify that the `children` have been added or removed from the document. We're going to mock out the `CSSTransition` component from `react-transition-group` in our test.

[00:28] Let's go ahead, and we'll `import React from 'react'`, because we're going to need that. We'll also `import {render} from 'react-testing-library'`, and we'll `import {HiddenMessage} from '../hidden-message'`. Then we'll add a test, `'shows hidden message when toggle is clicked'`.

mock-component.js

```
import React from 'react'
import {render} from 'react-testing-library'
import {HiddenMessage} from '../hidden-message'

test('shows hidden message when toggle is
clicked', () => {

})
```

[00:48] Next, our test is going to render that `HiddenMessage`. We need to have some message in here so I'm going to make a variable called `myMessage`. We'll have it say `'hello world'`, and then we'll put `myMessage` as a child to `HiddenMessage`.

[01:01] Then we're going to need to click on that `button`, so we'll `getByText`. That'll equal render that `HiddenMessage`, and we'll get that `toggleButton`. That'll be `getByText(/toggle/i)` in our case there. Then we'll need to fire an event on the `toggleButton`.

[01:17] We'll bring in `fireEvent` from `react-testing-library`, and we'll fire a `click` event on the `toggleButton`. Then we can expect `getByText(myMessage).toBeInTheDocument()`. Cool.

```
test('shows hidden message when toggle is
clicked', () => {
  const myMessage = 'hello world'
  const {getByText} = render(<HiddenMessage>
{myMessage}</HiddenMessage>)
  const toggleButton = getByText(/toggle/i)
  fireEvent.click(toggleButton)

expect(getByText(myMessage)).toBeInTheDocument()
})
```

[01:31] If we save that, and then pop open our test here, we've got a failing test. The reason is, if we come up here, it's going to tell us it cannot find the `getByText(myMessage)`. Let's go ahead and we'll mock out our `CSSTransition` here so that the `Fade` will render the `children` immediately, rather than having to wait for the `CSSTransition`.

[01:53] We'll say `jest.mock('react-transition-group')`, and then we'll return `CSSTransition`. This is going to be a function component that simulates the same API as the component that we have from `react-transition-group`.

[02:07] The way that this works is, we have a `Fade` here. That takes an `in` prop. Then we forward that prop along to `CSSTransition`. `CSSTransition` takes an `in` prop to know whether or not the `children` should be rendered.

hidden-message.js

```
import {CSSTransition} from 'react-transition-group'

function Fade({children, ...props}) {
  return (
    <CSSTransition {...props} timeout={1000} className="fade">
      {children}
    </CSSTransition>
  )
}

class HiddenMessage extends React.Component {
  ...
  render() {
    return (
      <div>
        ...
        <Fade in={this.state.show}>
          <div>{this.props.children}</div>
        </Fade>
      </div>
    )
  }
}
```

[02:20] That's exactly what our mock is going to do. We'll say `props`, and if `props.in`, then we'll render `props.children`, otherwise, we'll render `null`. Now, if we save that, our test is passing.

mock-component.js

```
jest.mock('react-transition-group', () => {
  return {
    CSSTransition: props => (props.in ?
props.children : null),
  }
})
```

[02:32] In addition to this, we're going to take this `getByText` assertion. We'll put it right before the `fireEvent.click`, and we'll assert that it's `.not` in the document. We save that, we're going to get an error, because we're trying to `getByText(myMessage)`, and it can't find that.

```
test('shows hidden message when toggle is
clicked', () => {
  const myMessage = 'hello world'
  const {getByText} = render(<HiddenMessage>
{myMessage}</HiddenMessage>)
  const toggleButton = getByText(/toggle/i)

expect(getByText(myMessage)).not.toBeInTheDocume
nt()
  fireEvent.click(toggleButton)

expect(getByText(myMessage)).toBeInTheDocument()
})
```

[02:46] We're going to use the query version of this API, `queryByText`, and we'll `expect` it not to be in the document at this point. Then we can toggle this thing again, and we'll `expect` it not to be in the document anymore.

```
test('shows hidden message when toggle is
clicked', () => {
  const myMessage = 'hello world'
  const {getByText, queryByText} =
render(<HiddenMessage>{myMessage}
</HiddenMessage>)
  const toggleButton = getByText(/toggle/i)

expect(queryByText(myMessage)).not.toBeInTheDocu
ment()
  fireEvent.click(toggleButton)

expect(getByText(myMessage)).toBeInTheDocument()
  fireEvent.click(toggleButton)

expect(queryByText(myMessage)).not.toBeInTheDocu
ment()
})
```

[02:57] That covers an entire use case for our component. In review, to properly mock a third party component, you need to simulate the same API that it has. We had the `CSSTransition`, we had a good reason to mock it, and so we created a Jest mock for `react-transition-group`.

[03:13] Then we returned our own version of the `CSSTransition` that worked synchronously to render the `children`. Then our test could run synchronously as well.

Test componentDidCatch handler error boundaries with react-testing-library

Kent C Dodds: [00:00] Here, we have a typical `ErrorBoundary` component that we probably put some more at the root of our application to make sure that any errors that happen as the users interacting with your application will be reported to the server, so we can address them later.

[00:12] This `ErrorBoundary` also supports this `button` to `.tryAgain` that basically rerenders this component and that hopes that the error won't happen again. To test this, the first thing that we're going to want to do is mock out this `reportError` API called, because we don't want to make server calls in our tests.

error-boundary.js

```
import {reportError} from './api'

class ErrorBoundary extends React.Component {
  state = {hasError: false}
  componentDidCatch(error, info) {
    this.setState({hasError: true})
    reportError(error, info)
  }
  tryAgain = () => this.setState({hasError:
false})
  render() {
    return this.state.hasError ? (
      <div>
        <div>There was a problem.</div>{' '}
        <button onClick={this.tryAgain}>Try
again?</button>
      </div>
    ) : (
      this.props.children
    )
  }
}
```

[00:28] In our tests, let's go ahead and we'll use `jest.mock`. We'll go back to that `'../api'`. We'll return this `reportError` function as a `jest.fn()` that returns a `Promise` that resolves to something that our server would send `{success: true}`.

src/*tests*/error-boundary.js

```
jest.mock('../api', () => {
  return {
    reportError: jest.fn(() =>
Promise.resolve({success: true}))
  }
})
```

[00:46] Next, let's go ahead and make our test and we'll say `calls reportError and renders that there was a problem`. We're going to want to `render` this. We'll `import {render} from 'react-testing-library'`. We're going to want to `import {ErrorBoundary} from '../error-boundary'`.

[01:06] Then, we'll call `render(<ErrorBoundary></ErrorBoundary>)`. We need to `render` some children that are going to throw an error. We can simulate this kind of error thrown. Let's go ahead and make that.

```
import {render} from 'react-testing-library'
import {ErrorBoundary} from '../error-boundary'

...

test(`calls reportError and renders that there
was a problem`, () => {
  render(<ErrorBoundary></ErrorBoundary>)
})
```

[01:16] We'll make a function called `Bomb`, and that'll take a parameter called `shouldThrow`. We'll say `if (shouldThrow) { throw new Error('') }` and we'll just take a bomb emoji in

there, otherwise we'll `return null`.

[01:33] Great. That's what we're going to `render` here. We'll `render` our `Bomb` and we'll initially not pass the `shouldThrow` props, so that are renders fine.

```
function Bomb({shouldthrow}) {
  if (shouldThrow) {
    throw new Error('   ')
  } else {
    return null
  }
}

test(`calls reportError and renders that there
was a problem`, () => {
  render(<ErrorBoundary><Bomb />
</ErrorBoundary>)
})
```

Next, let's get `rerender`, and we'll `rerender` it.

[01:46] We'll copy all this. Paste in here and we'll say that the `<Bomb shouldThrow={true} />`. We'll go ahead and save that. Looks like I forgot to import React. Let's go ahead and do that for sure.

```
import React from 'react'

...

test(`calls reportError and renders that there
was a problem`, () => {
  const {rerender} = render(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )

  rerender(
    <ErrorBoundary>
      <Bomb shouldThrow={true}/>
    </ErrorBoundary>
  )
})
```

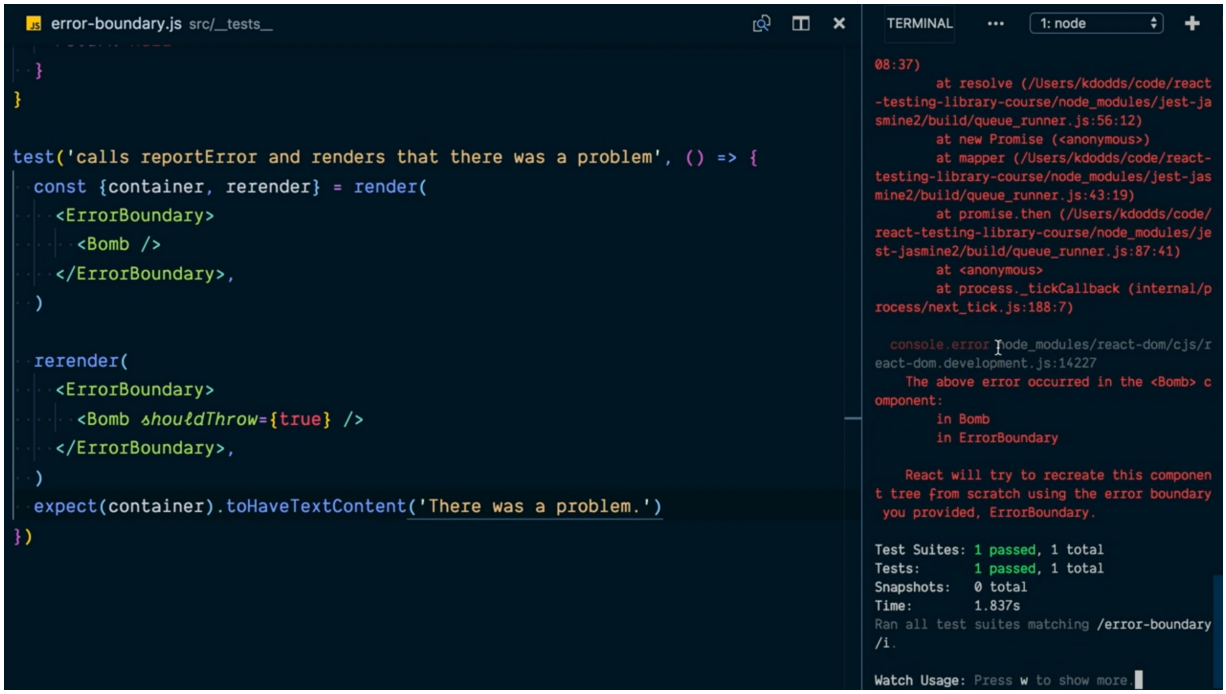
[01:58] `import React from 'react`. Now, we'll save. Let's go ahead and run our test now,`npm run test:watch error-boundary`. We're getting a bunch of console errors. This is good. This is exactly what we're looking for. We want this `rerender` to cause the `ErrorBoundary` to catch an error.

[02:17] If I comment that out, then it shouldn't throw any errors at all. Cool. Now, we can make some assertions on the `container`, so I'll bring in `container`. We'll `expect(container).toHaveTextContent('There was a problem.')`.

```
test(`calls reportError and renders that there
was a problem`, () => {
  const {container, rerender} = render(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )

  rerender(
    <ErrorBoundary>
      <Bomb shouldThrow={true}/>
    </ErrorBoundary>
  )
  expect(container).toHaveTextContent('There was
a problem.')
})
```

[02:31] We save that, and our test is actually passing even though we have a whole bunch of really noisy output here. This is doing exactly what we want.

Noisy Output



[02:43] Let's go ahead and get rid of this noisy output. This is happening from JS DOM's virtual console. When an error is thrown, even if it's caught by an `ErrorBoundary`, React will log to the console, which makes a lot of sense, but in our tests it's really annoying.

[02:58] Let's go ahead and mock out the console. I'll say `beforeEach`. We'll say `jest.spyOn(console, 'error')`. We'll `mockImplementation` to do nothing. We'll cleanup after ourselves with `afterEach` and we'll say `console.error.mockRestore()`.

```
beforeEach(() => {
  jest.spyOn(console,
'error').mockImplementation(() => {})
})

afterEach(() => {
  console.error.mockRestore()
})
```

[03:18] If we save that, now we should be free of all those console warnings, but we want to make sure that the `console.error` is called the right number of times, because now, we don't have any insight into `console.error`.

[03:29] That might actually be an error that is really necessary for us to know about. Let's go ahead and we'll add an assertion here to `expect(console.error).toHaveBeenCalledTimes(2)`.

```
test(`calls reportError and renders that there
was a problem`, () => {
  const {container, rerender} = render(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )

  rerender(
    <ErrorBoundary>
      <Bomb shouldThrow={true}/>
    </ErrorBoundary>
  )
  expect(container).toHaveTextContent('There was
a problem.')
  expect(console.error).toHaveBeenCalledTimes(2)
})
```

[03:42] Let's call once by JS DOM and ones by React. Great. So far, so good. Now, we need to make sure that our `reportError` function is being called properly. Let's go ahead and grab the calls to our `reportError`. We're going to need to `import {reportError as mockReportError} from '../api'`.

[04:06] We're just going to alias it as `mockReportError`, so that it's more clear that this is a mocked version of that function. Then, I'm going to `console.log(mockReportError.mock.calls)` and we'll see with that looks like.

[04:18] It's an array of calls. There should only be one. I'll just grab that first one with `console.log(mockReportError.mock.calls[0])` and we have the `Error:` , and so on and so forth. It's just a lot of stuff here. Let's go ahead and get the `.length` here on this log.

Error

```
error-boundary.js src/__tests__

afterEach(() => {
  console.error.mockRestore()
})

test('calls reportError and renders that there was a problem', () => {
  const {container, rerender} = render(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>,
  )

  rerender(
    <ErrorBoundary>
      <Bomb shouldThrow={true} />
    </ErrorBoundary>,
  )
  console.log(mockReportError.mock.calls[0])
  expect(container).toHaveTextContent('There was a problem.')
  expect(console.error).toHaveBeenCalledTimes(2)
})
```

```
TERMINAL    1: node

RUNS  src/__tests__/error-boundary.js

Test Suites: 0 of 1 total
Tests:       0 total
Snapshots:   0 total
  console.log src/__tests__/error-boundary.j
s:43
    [ Error:
        at Bomb (/Users/kdodds/code/react-te
sting-library-course/src/__tests__/error-bou
ndary.js:17:11)
        at updateFunctionalComponent (/Users
/kdodds/code/react-testing-library-course/no
de_modules/react-dom/cjs/react-dom.developme
nt.js:13126:20)
        at beginWork (/Users/kdodds/code/rea
ct-testing-library-course/node_modules/react
-dom/cjs/react-dom.development.js:13823:14)
        at performUnitOfWork (/Users/kdodds/
code/react-testing-library-course/node_modul
es/react-dom/cjs/react-dom.development.js:15
864:12)
        at workLoop (/Users/kdodds/code/reac
t-testing-library-course/node_modules/react-
dom/cjs/react-dom.development.js:15903:24)
        at renderRoot (/Users/kdodds/code/re
act-testing-library-course/node_modules/reac
t-dom/cjs/react-dom.development.js:15943:7)
        at performWorkOnRoot (/Users/kdodds/
code/react-testing-library-course/node_modul
es/react-dom/cjs/react-dom.development.js:16
561:22)
        at performWork (/Users/kdodds/code/r
```

[04:32] We got two arguments here. Those two arguments that we're passing into `reportError` are the `error` and the `info`. Let's make an assertion for each one of those. We'll `expect(mockReportError).toHaveBeenCalledTimes(1)`, and we'll expect it `toHaveBeenCalledWith(error, info)`.

[04:52] The `error` is going to be `expect.any(Error)`. It's going to be that `error` object that we're throwing in our `Bomb`. The next one is `info` and that's React's specific information, and that object can look something like this, `{componentStack: expect.stringContaining('Bomb')}`.

```
test(`calls reportError and renders that there
was a problem`, () => {
  const {container, rerender} = render(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )

  rerender(
    <ErrorBoundary>
      <Bomb shouldThrow={true}/>
    </ErrorBoundary>
  )

expect(mockReportError).toHaveBeenCalledTimes(1)
  const error = expect.any(Error)
  const info = {componentStack:
expect.stringContaining('Bomb')}

expect(mockReportError).toHaveBeenCalledWith(err
or, info)
  expect(container).toHaveTextContent('There was
a problem.')
  expect(console.error).toHaveBeenCalledTimes(2)
})
```

[05:15] Perfect. All right, we've got a pretty good feel of this `ErrorBoundary` component, but there is one last thing that we don't have covered in `ErrorBoundary` component. We could write

a new test for this, but we've already set up ourselves with some pretty good state in this test.

[05:28] I think I just want to continue on through the lifecycle of this `ErrorBoundary` here. I'm going to go ahead and click on the `tryAgain` button, after I `rerender` this, so that the `Bomb` is set not to explode.

[05:40] Let's go ahead and we're going to clear out our state. We'll say `console.error.mockClear()`. We'll say `mockReportError.mockClear()`. With that we can `rerender` this whole thing. Except this time, we'll say that it should not throw.

```
test(`calls reportError and renders that there
was a problem`, () => {
  ...


expect(mockReportError).toHaveBeenCalledTimes(1)
  const error = expect.any(Error)
  const info = {componentStack:
expect.stringContaining('Bomb')}

expect(mockReportError).toHaveBeenCalledWith(err
or, info)
  expect(container).toHaveTextContent('There was
a problem.')
  expect(console.error).toHaveBeenCalledTimes(2)

  console.error.mockClear()
  mockReportError.mockClear()

  rerender(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )
})
```

[05:59] Then, I'm going to need a `fireEvent`. I can fire a quick event, so say `fireEvent.click()` and we want to get by text. Here, we'll say `getByText` and get `/try again/i`. Then, we basically want to do all the same assertions except we want them to not be the case.

```
import {render, fireEvent} from 'react-testing-library'

...

test(`calls reportError and renders that there was a problem`, () => {
  const {container, rerender, getByText} = render(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )

  ...

  fireEvent.click(getByText(/try again/i))
})
```

[06:23] The `mockReportError` should not have been called, `console.error` should not have been called. We should not see the text "there was a problem". Let's get rid of all of this and that, and we'll say `.not.ToHaveBeenCalled`, and `.not.ToHaveBeenCalled`, and `.not.toHaveTextContent('There was a problem.')`.

```
test(`calls reportError and renders that there
was a problem`, () => {
  ...

  console.error.mockClear()
  mockReportError.mockClear()

  rerender(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )

  fireEvent.click(getByText(/try again/i))

  expect(mockReportError).not.toHaveBeenCalled()
  expect(container).not.toHaveTextContent('There
was a problem.')
  expect(console.error).not.toHaveBeenCalled()
})
```

[06:43] With that, our component is fully tested. This works, because we cleared out the `console.error` and `mockReportError` calls, so that we can make assertions that those things were not called, when we `rerender` our `ErrorBoundary` with the `Bomb` that does not explode.

[06:58] In review what we had to do here was, first, we noted that the `reportError` API was going to be called when there is an error. We mocked out that `reportError` function, so we're not making HTTP calls.

```
jest.mock('../api', () => {
  return {
    reportError: jest.fn(() =>
Promise.resolve({success: true}))
  }
})
```

[07:10] Then we created this `Bomb` component, so that we could `render` this and conditionally throw an error as we're rendering to similar an error doing our component lifecycles.

```
function Bomb({shouldthrow}) {
  if (shouldThrow) {
    throw new Error('   ')
  } else {
    return null
  }
}
```

[07:20] Then, we rendered our `ErrorBoundary` with a `Bomb` that does not explode and `rerender` it with a `Bomb` that does, and verified that the `mockReportError` was called once and that it was called with `error` and an `info`, and that the `container` has the content, "There was a problem." and the `console.error` was called twice, because we're mocking it to avoid a lot of noise in our console.

```
beforeEach(() => {
  jest.spyOn(console,
'error').mockImplementation(() => {})
})
```

```
afterEach(() => {
  console.error.mockRestore()
})

test(`calls reportError and renders that there
was a problem`, () => {
  const {container, getByText rerender} =
render(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )

  rerender(
    <ErrorBoundary>
      <Bomb shouldThrow={true}/>
    </ErrorBoundary>
  )

expect(mockReportError).toHaveBeenCalledTimes(1)
  const error = expect.any(Error)
  const info = {componentStack:
expect.stringContaining('Bomb')}

expect(mockReportError).toHaveBeenCalledWith(err
or, info)
  expect(container).toHaveTextContent('There was
a problem.')
  expect(console.error).toHaveBeenCalledTimes(2)

  ...
})
```

[07:44] We cleared out our `console.error` and `mockReportError`. We've rrendered with the `Bomb` that does not explode and we clicked on that `tryAgain` to reset the state of the `ErrorBoundary`, so it will `rerender` its children.

[07:55] Then, we verify that the `mockReportError` was not called, the `container` does not have "There was a problem.", and `console.error` was not called.

```
test(`calls reportError and renders that there
was a problem`, () => {
  ...

  console.error.mockClear()
  mockReportError.mockClear()

  rerender(
    <ErrorBoundary>
      <Bomb />
    </ErrorBoundary>
  )

  fireEvent.click(getByText(/try again/i))

  expect(mockReportError).not.toHaveBeenCalled()
  expect(container).not.toHaveTextContent('There
was a problem.')
  expect(console.error).not.toHaveBeenCalled()
})
```

Test drive the development of a React Form with react-testing-library

Kent C Dodds: [00:00] I'm going to start with a test that is `'renders a form with title, content, tags, and a submit button'`. Then we're going to get our `getByLabelText` and `getByText`. The `getByLabelText` will help us get our form controls, and the `getByText` will get us our submit button.

[00:18] Then we're going to `render` our `Editor`, and we need to pull in a couple things here. I'm using React, so we need to `import React from 'react'`. Then we'll `import {render} from 'react-testing-library'`. Then we're rendering our `Editor`, so we're going to `import {Editor} from '../post-editor-01-markup'`.

tdd-01-markup.js

```
import React from 'react'
import {render} from 'react-testing-library'
import {Editor} from '../post-editor-01-markup'

test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
})
```

[00:37] Now, I'm going to go ahead and `getByLabelText`. I don't care about the casing here, so I'm going to use a `title` with an ignore-case flag. We're going to get the form control that has a `label` called `title`. We'll do the same thing for `content` and `tags`.

tdd-01-markup.js

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
  getByLabelText(/title/i)
  getByLabelText(/content/i)
  getByLabelText(/tags/i)
  getByText(/submit/i)
})
```

[00:52] Then I'm going to `getByText` to get my `submit` button. Now, if I open up my test here, I can see here that my test is failing. It's because this post-editor markup doesn't have anything that it's exporting. It's completely blank.

[01:06] Let's go ahead, and we'll `import React from 'react'`. We'll create a React component called `Editor`. Then we'll have it `render` a `form`. Then we'll `export {Editor}`.

post-editor-01-markup.js

```
import React from 'react'

class Editor extends React.Component {
  render() {
    return <form />
  }
}

export {Editor}
```

Great. Now, we're onto our next error message, "*Unable to find label with the text of: /title/i*"

[01:23] Let's go ahead and `render` a `label` with a text of title. Let's `<label>Title</label>`. Great. Now, it says it found that `label`, but there was no `form` control associated to that `label`. Let's make a `form` control. In our case, that's an `input`.

[01:38] Now, we need to associate this `label` with this `input`. There are various ways to do that. What we're going to do is an `htmlFor="title-input"`. Then on our `input`, we'll add `id="title-input"`. Then we'll save that, and we've moved onto our next error.

post-editor-01-markup.js

```
class Editor extends React.Component {
  render() {
    return (
      <form>
        <label htmlFor="title-
input">Title</label>
        <input id="title-input"/>
      </form>
    )
  }
}
```

[01:53] *Unable to find a label with the text of: /content/i*. We can go ahead, and we'll just copy this. We'll change title to content. We'll capitalize this inner text. Instead of an `input`, we want this to be a `textarea`. Cool.

post-editor-01-markup.js

```
class Editor extends React.Component {
  render() {
    return (
      <form>
        <label htmlFor="title-
input">Title</label>
        <input id="title-input"/>

        <label htmlFor="content-
input">Content</label>
        <textarea id="content-input"/>
      </form>
    )
  }
}
```

[02:08] Now, we have tags to deal with. Let's go ahead, and we'll copy this. We'll change this to `tags`, make this inner text capital. Now, it's *Unable find an element with the text of: /submit/i*. Let's go ahead, and we'll make a `<button type='submit'>Submit</button>`. That gets our test passing.

post-editor-01-markup.js

```
class Editor extends React.Component {
  render() {
    return (
      <form>
        <label htmlFor="title-
input">Title</label>
        <input id="title-input"/>

        <label htmlFor="content-
input">Content</label>
        <textarea id="content-input"/>

        <label htmlFor="tags-input">Tags</label>
        <input id="tags-input"/>

        <button type='submit'>Submit</button>
      </form>
    )
  }
}
```

[02:28] That gets our test passing. This is the red-green refactor cycle of test-driven development.
First, you write your test for the thing that you're going to be implementing. That makes your test red, because you haven't implemented the thing that you're building yet.

[02:39] Then you go and implement it, going methodically, one step at a time, until your test is passing.
Then there's an important step of refactoring after you've gotten the red to the green. In our case, there's not really anything I care to refactor here.

[02:54] I'm going to leave it as it is, but make sure that you think about the refactor step of the red-green refactor cycle, otherwise you could wind up with some pretty nasty code.

Test drive the submission of a React Form with react-testing-library

Kent C Dodds: [00:00] We have a passing test for our post `Editor`. Now we want to make it so that when the user clicks submit, the submit `button` is disabled. That way, they can't submit multiple posts by accident. Here, in our test, I'm going to get our `submitButton`, so we can click on it.

tdd-02-markup.js

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
  getByLabelText(/title/i)
  getByLabelText(/content/i)
  getByLabelText(/tags/i)
  getByText(/submit/i)
})
```

[00:14] I'll say, `const submitButton = getByText(/submit/i)` Then we need to fire an event, so I'm going to pull in `fireEvent`. We'll come back down here and say `fireEvent.click()` on that `submitButton`. Then we can `expect(submitButton).toBeDisabled()`. That's the red part of our red/green refactor cycle.

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
  getByLabelText(/title/i)
  getByLabelText(/content/i)
  getByLabelText(/tags/i)
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()
})
```

[00:34] Let's go ahead and make this test pass. In `post-editor-02-markup.js`, first, I'm going to add a `disabled` prop here in the `button`, that'll depend on `this.state.isSaving`. Then I'll have that `state = {isSaving: }`. We'll initialize it to `false`. Then we need to have a form submit handler, so I'll add an `onSubmit={this.handleSubmit}` to the `form`.

[00:53] We'll declare that here, `handleSubmit` equals an arrow function that takes an event, and will `preventDefault`. Then we'll say `this.setState({isSaving: true})`. Our test is now passing. As far as the refactor phase goes, I don't really think there's anything in here that I care to refactor, so we'll go ahead and leave it as it is.

post-editor-02-markup.js

```
class Editor extends React.Component {
  state = {isSaving: false}
  handleSubmit = e => {
    e.preventDefault()
    this.setState({isSaving: true})
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label htmlFor="title-
input">Title</label>
        <input id="title-input"/>

        <label htmlFor="content-
input">Content</label>
        <textarea id="content-input"/>

        <label htmlFor="tags-input">Tags</label>
        <input id="tags-input"/>

        <button type='submit' disabled=
{this.state.isSaving}>
          Submit
        </button>
      </form>
    )
  }
}
```

[01:14] In review, what we needed to do here is, we got our `submitButton`, we fired a `click` event by getting the `fireEvent` utility from `react-testing-library`. Then we expected the `submitButton` to be `disabled`. That gave us a broken test. Then

we went into our `Editor`, and we added the necessary `disabled` prop here, and the `state`, and a submit handler, to make that test pass.

tdd-02-markup.js

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
  getByLabelText(/title/i)
  getByLabelText(/content/i)
  getByLabelText(/tags/i)
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()
})
```

Test drive the API call of a React Form with react-testing-library

Kent C Dodds: [00:00] Next, we're going to want to have a special API to submit this form. That's going to come from a `savePost` from our `'../api'`. Then, we'll use the `savePost` function to submit the post data, but we don't actually want to submit that post data in our test.

[00:15] We're going to mock that in our test. To start off in our test, I'm going to use `jest.mock` and we'll direct ourselves to the path of that `'../api'` module. Now, we can mock out that entire `'../api'` module with whatever it is that we want to.

[00:29] It's specifically what we want to mock out is the `savePost` method that we're going to be using. I'll have a `savePost`, and here, we'll simply do `jest.fn()`. We'll have our implementation be an arrow function that returns a `Promise.resolve()`.

tdd-03-api.js

```
import React from 'react'
import {render, fireEvent} from 'react-testing-
library'
import {Editor} from '../post-editor-03-api'

jest.mock('../api', () => {
  return {
    savePost: jest.fn(() => Promise.resolve)
  }
})
```

[00:43] Right now, we don't really care with the resolves too just that it returns a `Promise` that is resolved. Now if we want to get a hold of this function, we need to import it. I'm going to import `savePost`, and we're going to alias it to `mockSavePost` just so that in our test we know that the mock version.

[01:00] We'll get that from the same path to that `'../api'`. Now, we'll go down here and say, `expect(mockSavePost).toHaveBeenCalledTimes(1)` I'll save my file and that gets our test failing.

tdd-03-api.js

```
import {savePost as mockSavePost} from '../api'

...

test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
  getByLabelText(/title/i)
  getByLabelText(/content/i)
  getByLabelText(/tags/i)
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()
  expect(mockSavePost).toHaveBeenCalledTimes(1)
})
```

Let's go ahead and make this test pass by calling `savePost` in this `handleSubmit`.

post-editor-03-api.js

```
class Editor extends React.Component {
  state = {isSaving: false}
  handleSubmit = e => {
    e.preventDefault()
    this.setState({isSaving: true})
    savePost()
  }
  ...
```

[01:19] We'll save that and our test is green. Cool. That's not super useful, because we're not actually saving any of the data that we want to send it to the server. Let's go ahead, get that data, and send it off to the server.

[01:31] In our test, we need to set the value of each one of these fields, so that when the submit `button` is clicked, our submit handler can get those values, and save the post. I'll go ahead and set the `.value` of `/title/i` to `'Test Title'` and the value of `/content/i` to `'Test content'`.

[01:47] The value of `/tags/i` is going to be `'tag1,tag2'`. Then down here, we can `expect(mockSavePost).toHaveBeenCalledWith({})`, and that object can contain `title: 'Test Title'`, `content: 'Test content'`,
and the `tags`. We actually want this to be an array of the comma separated tags, `tags: ['tag1', 'tag2']`.

tdd-03-api.js

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
  getByLabelText(/title/i).value = 'Test Title'
  getByLabelText(/content/i).value = 'Test
content'
  getByLabelText(/tags/i).value = 'tag1,tag2'
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()
  expect(mockSavePost).toHaveBeenCalledTimes(1)
  expect(mockSavePost).toHaveBeenCalledWith({
    title: 'Test Title',
    content: 'Test content',
    tags: ['tag1', 'tag2']
  })
})
```

[02:13] Now, we can save that. We're getting our test failure. Let's go ahead and implement this. I need to get the value, so that I can save this in the post. We have the `title`, `content`, and `tags`. Where am I going to get those values?

post-editor-03-markup.js

```
class Editor extends React.Component {
  state = {isSaving: false}
  handleSubmit = e => {
    e.preventDefault()
    this.setState({isSaving: true})
    savePost({
      title,
      content,
      tags,
    })
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label htmlFor="title-
input">Title</label>
        <input id="title-input" name="title"/>

        <label htmlFor="content-
input">Content</label>
        <textarea id="content-input"
name="content"/>

        <label htmlFor="tags-input">Tags</label>
        <input id="tags-input" name="tags"/>

        <button type='submit' disabled=
{this.state.isSaving}>
          Submit
        </button>
      </form>
    )
  }
}
```

[02:31] We have them in the `form` and we have the `form` elements via `e.target`. Let's go ahead and we'll add a `name` property to these. We have `title`, `content`, and `tags`. Up here, we can get the `title`, `content`, and `tags` from `e.target` that's the form, `.elements`. That's the elements of the `form`.

[02:53] We have access to each one of these elements, because of the `name` attribute, but these are the nodes. We're going to say `title.value` and `content.value`. For `tags`, we're going to take `tags.value`.

[03:07] Then, we'll `split` that by commas, because the `tags` value is a string and we want users to be able to put commas, but we also want to `trim` any void space, if they added a space after the comma, for example, so we'll say,`.map(t => t.trim()`. With that, we'll save, and our test run and we get a passing test.

post-editor-03-api.js

```
class Editor extends React.Component {
  state = {isSaving: false}
  handleSubmit = e => {
    e.preventDefault()
    const {title, content, tags} =
e.target.elements
    this.setState({isSaving: true})
    savePost({
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t =>
t.trim()),
    })
  }
  ...
```

[03:25] Let's go ahead and refactor. Now that, we have our red and our green. Let's enter the refactor phase. Now, one thing that I don't like about this is that were duplicating these strings across both of these values.

tdd-03-api.js

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
  getByLabelText(/title/i).value = 'Test Title'
// duplicate strings
  getByLabelText(/content/i).value = 'Test
content' // duplicate strings
  getByLabelText(/tags/i).value = 'tag1,tag2' //
duplicate strings
  const submitButton = getByText(/submit/i)
  ...

  expect(mockSavePost).toHaveBeenCalledWith({
    title: 'Test Title', // duplicate strings
    content: 'Test content', // duplicate
strings
    tags: ['tag1', 'tag2'] // duplicate strings
  })
})
```

[03:37] I would like to communicate with my test of these values are actually related in one way another. What I'm going to do is I'm going to create a `fakePost` object that's going to have `title: 'Test Title'`, `content: 'Test content'`, and `tags: ['tag1', 'tag2']`.

[03:54] We can set the values to `fakePost.title`, `fakePost.content`, and `fakePost.tags.join(', ')`. In here, we can expect that our `mockSavePost` was called with the `fakePost`.

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const {getByLabelText, getByText} =
render(<Editor />)
  const fakePost = {
    title: 'Test Title',
    content: 'Test content',
    tags: ['tag1', 'tag2']
  }
  getByLabelText(/title/i).value =
fakePost.title
  getByLabelText(/content/i).value =
fakePost.content
  getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()
  expect(mockSavePost).toHaveBeenCalledTimes(1)

expect(mockSavePost).toHaveBeenCalledWith(fakePo
st)
})
```

[04:16] If we save that, our refactor was good. Our test is still green. There is another feature we need to implement here. That is that a post needs to have a user that created the post needs to

have an author. I'm going to add to this assertion that this is going to be all the property's config post as well as an `authorId`.

[04:35] We're going to create a `fakeUser` id here. We'll make our `fakeUser = {id: 'user-1'}`. Then, the `Editor` needs to get that `fakeUser` somehow, so we'll pass it as a prop, so say `user={fakeUser}`. We'll go ahead and save that. Now, we have the failing test, because the `authorId` is not supplied.

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const fakeUser = {id: 'user-1'}
  const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
  ...
  expect(mockSavePost).toHaveBeenCalledWith({
    ...fakePost,
    authorId: fakeUser.id
  })
})
```

[04:55] Let's go ahead and implement this. This actually fairly straightforward. We can add `authorId: this.props.user.id`. We save that. We get our passing test. From here, I don't have any other refactorings that I want to do. I'm pretty happy with this.

post-editor-03-markup.js

```
class Editor extends React.Component {
  state = {isSaving: false}
  handleSubmit = e => {
    e.preventDefault()
    this.setState({isSaving: true})
    savePost({
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t =>
t.trim()),
      authorId: this.props.user.id
    })
  }
  ...
```

[05:11] One last thing before we wrap up here, I'm going to go ahead and add an `afterEach` callback here. I'm going to `mockSavePost.mockClear()`. What this does is that it will clear the state after each one of the tests in this file, so that this `savePost` mock function doesn't hang on to any state from previous tests.

tdd-03-api.js

```
jest.mock('../api', () => {
  return {
    savePost: jest.fn(() => Promise.resolve)
  }
})

afterEach(() => {
  mockSavePost.mockClear()
})
```

[05:30] This keeps our test isolated and we can avoid some confusion in the future, if we add more tests to this file. In review, to make all of this work, we created a mock for our `'../api'` that we're going to be using to save the post using the `jest.mock` API.

[05:44] We use a Jest function that returns a result `Promise` and we also cleanup after ourselves after each one of the tests. Then, we created a `fakeUser` and pass that along to the `Editor`. We also create a `fakePost` and set those values on each one of the associated nodes.

```
test('renders a form with title, content, tags,
and a submit button', () => {
  const fakeUser = {id: 'user-1'}
  const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
  const fakePost = {
    title: 'Test Title',
    content: 'Test content',
    tags: ['tag1', 'tag2']
  }
  getByLabelText(/title/i).value =
fakePost.title
  getByLabelText(/content/i).value =
fakePost.content
  getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()
  expect(mockSavePost).toHaveBeenCalledTimes(1)
  expect(mockSavePost).toHaveBeenCalledWith({
    ...fakePost,
    authorId: fakeUser.id
  })
})
```

[06:00] After the `submitButton` has been clicked, we expected that our `mockSavePost` was called one time and that was called with the properties from the `fakePost` as well as the `authorId`. Then, we went ahead and implemented each one of these features in turn by adding name attributes to each one of our `form` controls.

[06:20] I'm getting those `form` control nodes to get the values for a `savePost` call.

post-editor-03-markup.js

```
class Editor extends React.Component {
  state = {isSaving: false}
  handleSubmit = e => {
    e.preventDefault()
    this.setState({isSaving: true})
    savePost({
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t =>
t.trim()),
      authorId: this.props.user.id
    })
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label htmlFor="title-
input">Title</label>
        <input id="title-input" name="title"/>

        <label htmlFor="content-
input">Content</label>
        <textarea id="content-input"
name="content"/>

        <label htmlFor="tags-input">Tags</label>
        <input id="tags-input" name="tags"/>

        <button type='submit' disabled=
{this.state.isSaving}>
          Submit
```

```
          </button>
        </form>
      )
    }
  }
```

There is one last refactoring I'm going to do here really and that is to take out this object and create a `newPost`.

[06:32] Then, we'll create that `newPost` here. We can double check that our tests are still passing after that little refactor. We're good.

post-editor-03-markup.js

```
class Editor extends React.Component {
  state = {isSaving: false}
  handleSubmit = e => {
    e.preventDefault()
    const newPost = {
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t =>
t.trim()),
      authorId: this.props.user.id
    }
    this.setState({isSaving: true})
    savePost(newPost)
  }
```

Test drive mocking react-router's Redirect component on a form submission

Kent C Dodds: [00:00] Once this post has been successfully saved, I want to redirect the user to the home page. I'm going to use React Router's `Redirect` component to send the user to the home page. Let's go ahead and write our test for this.

post-editor-04-markup.js

```
class Editor extends React.Component {
  state = {isSaving: false}
  handleSubmit = e => {
    e.preventDefault()
    const newPost = {
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t =>
t.trim()),
      authorId: this.props.user.id
    }
    this.setState({isSaving: true})
    savePost(newPost)
  }
```

[00:12] First, I'm going to use `jest.mock` to mock out `react-router`. Then I'll `return` a mock version of `Redirect`. That simply can be a `jest.fn`. It's a function component that renders nothing, just returns `null`. All that I really care about is that I can make assertions on this `Redirect` component.

tdd-04-router-redirect.js

```
jest.mock('react-router', () => {
  return {
    Redirect: jest.fn(() => null)
  }
})
```

[00:34] To do that, I need to `import {Redirect as MockRedirect} from 'react-router'`. With that in place, I can go down here and add the assertion `expect(MockRedirect).toHaveBeenCalledTimes(1)`. I'll save that and my test is failing.

tdd-04-router-redirect.js

```
test('renders a form with title, content, tags,
and a submit button', () => {
  ...
  expect(mockSavePost).toHaveBeenCalledWith({
    ...fakePost,
    authorId: fakeUser.id
  })

  expect(MockRedirect).toHaveBeenCalledTimes(1)
})
```

[00:55] Let's go ahead and make this pass. I'm going to add some state here, `redirect: false`, and then down here in my `render` method, I'll say `if(this.state.redirect) { return <Redirect to="/" /> }`, to send the user home.

post-editor-04-markup.js

```
class Editor extends React.Component {
  state = {isSaving: false, redirect: false}
  ...
  render() {
    if (this.state.redirect) {
      return <Redirect to="/" />
    }
  ...
```

[01:11] I'll `import {Redirect} from 'react-router'` and then when the `savePost` is successful, add a `then` here, we call `this.setState({redirect: true})`. My test is still failing. This is all my implementation needs.

post-editor-04-markup.js

```
import {Redirect} from 'react-router'

class Editor extends React.Component {
  state = {isSaving: false, redirect: false}
  handleSubmit = e => {
    ...
    savePost(newPost).then(() =>
this.setState({redirect: true}))
  }
  render() {
    if (this.state.redirect) {
      return <Redirect to="/" />
    }
  ...
```

[01:28] The problem here is that this `savePost` is asynchronous. This callback happens asynchronously. So our `Redirect` rendering happens after our test is finished. What we need to do is wait for the `savePost` to finish before we make our assertion that `Redirect` has been rendered.

[01:46] `react-testing-library` actually has a utility for this called `wait`. We're going to change our test to be an `async` test, because it's going to be happening asynchronously. We're going to wait for this assertion to pass. We're going `await` to that.

tdd-04-router-redirect.js

```
import {render, fireEvent, wait} from 'react-
testing-library'

...

test('renders a form with title, content, tags,
and a submit button', async () => {
  ...

  await wait(() =>
expect(MockRedirect).toHaveBeenCalledTimes(1))
})
```

[02:00] What `wait` will do is we'll call this callback every 50 milliseconds until the callback no longer throws an error, effectively waiting for `MockRedirect` to have been called once. It times out after four and a half seconds. We'll go ahead and save our test here, and now our test is passing.

[02:16] Let's go ahead and add another assertion. `expect(MockRedirect).toHaveBeenCalledWith({to: '/'}, {})`. We save that, and our test is still passing.

tdd-04-router-redirect.js

```
test('renders a form with title, content, tags,
and a submit button', async () => {
  ...

  await wait(() =>
expect(MockRedirect).toHaveBeenCalledTimes(1))
  expect(MockRedirect).toHaveBeenCalledWith({to:
'/'}, {})
})
```

[02:31] Like I said, `wait` is going to timeout after four and a half seconds. If I make a typo here and expect it to have been called two times, then our test is going to take a little while before it reports that as an error, which is why it's better to limit your `wait` calls to have fewer assertions in them, because if this is working, then my test works quickly.

[02:49] If this is broken, then I get notified of that breakage quickly, but if I were to put both of these inside of my `wait` callback, then I'm going to have to wait four and a half seconds before I am notified of that breakage.

tdd-04-router-redirect.js

```
test('renders a form with title, content, tags,
and a submit button', async () => {
  ...

  await wait(() => {

expect(MockRedirect).toHaveBeenCalledTimes(1))

expect(MockRedirect).toHaveBeenCalledWith({to:
'/'}, {})
  }
})
```

[03:02] It's a good idea to limit what you have in your `wait` callback so you get notified of breakages sooner. Let's fix that up.

tdd-04-router-redirect.js

```
test('renders a form with title, content, tags,
and a submit button', async () => {
  ...

  await wait(() =>
expect(MockRedirect).toHaveBeenCalledTimes(1))
  expect(MockRedirect).toHaveBeenCalledWith({to:
'/'}, {})
})
```

One last thing I want to do is make sure that I clean up after myself. Let's say `MockRedirect.mockClear()`.

tdd-04-router-redirect.js

```
afterEach(() => {
  MockRedirect.mockClear()
  mockSavePost.mockClear()
})
```

[03:17] In review, what we did in our component here is we added a `then` handler to our `savePost` call so that we could update our state for our `redirect`. When we have a `redirect` state, we will render the `Redirect` component from `react-router`, which will redirect our user to the home page.

post-editor-04-markup.js

```
import {Redirect} from 'react-router'

class Editor extends React.Component {
  state = {isSaving: false, redirect: false}
  handleSubmit = e => {
    ...
    savePost(newPost).then(() =>
this.setState({redirect: true}))
  }
  render() {
    if (this.state.redirect) {
      return <Redirect to="/" />
    }
  ...
```

[03:35] In our test, we had to mock out the `react-router`, so we got `MockRedirect`. In our mock, we simply returned an object with a `Redirect`, and we didn't have to mock out the entire `react-router`, just the pieces that we're using.

tdd-04-router-redirect.js

```
jest.mock('react-router', () => {
  return {
    Redirect: jest.fn(() => null)
  }
})
```

[03:48] Our redirect mock is simply a `jest.fn` around a function component that doesn't render anything.
Then we can take that `MockRedirect`, `wait` for our `MockRedirect` to have been called one time while we wait for this `savePost` to resolve, and then assert that `MockRedirect` was called with the props of `{to: '/'}`, and a context of an empty object.

tdd-04-router-redirect.js

```
test('renders a form with title, content, tags,
and a submit button', async () => {
  ...
  expect(mockSavePost).toHaveBeenCalledTimes(1)
  expect(mockSavePost).toHaveBeenCalledWith({
    ...fakePost,
    authorId: fakeUser.id
  })

  await wait(() =>
expect(MockRedirect).toHaveBeenCalledTimes(1))
  expect(MockRedirect).toHaveBeenCalledWith({to:
'/'}, {})
})
```

Test drive assertions with dates in React

Kent C Dodds: [00:00] There's one more thing that our post needs, and that's the date that it was created. Now, we don't have to require the user to enter the date, because we know the date that it was created, and it's right now.

[00:10] Let's go ahead and add some tests to verify that the date was added to our and post and sent to the `savePost` API. The first thing I'm going to do is, in here, I'm going to say `date: new Date().toISOString()`. We get that failure here, because we have a date right there.

tdd-05-dates.js

```
test('renders a form with title, content, tags,
and a submit button', async () => {
  ...
  expect(mockSavePost).toHaveBeenCalledWith({
    ...fakePost,
    date: new Date().toISOString(),
    authorId: fakeUser.id
  })

  ...
})
```

Date in the test

```
post-editor-05-dates.js src
import React from 'react'
import {Redirect} from 'react-router'
import {savePost} from './api'

class Editor extends React.Component {
  state = {isSaving: false, redirect: false}
  handleSubmit = e => {
    e.preventDefault()
    const {title, content, tags} = e.target.elements
    const newPost = {
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t => t.trim()),
      authorId: this.props.user.id,
    }
    this.setState({isSaving: true})
    savePost(newPost).then(() => this.setState({redirect: true}))
  }
  render() {
    if (this.state.redirect) {
      return <Redirect to="/" />
    }
```

```
TERMINAL   1: node
    Expected mock function to have been call
ed with:
      {"authorId": "user-1", "content": "Tes
t content", "date": "2018-09-07T02:14:52.197
Z", "tags": ["tag1", "tag2"], "title": "Test
 Title"}
    as argument 1, but it was called with
      {"authorId": "user-1", "content": "Tes
t content", "tags": ["tag1", "tag2"], "title
": "Test Title"}.

    Difference:

    - Expected
    + Received

      Object {
        "authorId": "user-1",
        "content": "Test content",
    -   "date": "2018-09-07T02:14:52.197Z",
        "tags": Array [
          "tag1",
          "tag2",
        ],
        "title": "Test Title",
      }

      44 |
      45 |   expect(mockSavePost).toHaveBeen
CalledTimes(1)
    > 46 |   expect(mockSavePost).toHaveBeen
CalledWith({
      47 |     ...fakePost,
```

[00:29] Let's go ahead and implement this. We'll say `date: new Date().toISOString()`. Uh-oh, we have a problem here. Those are off by just milliseconds. That's because our tests are pretty fast, but they're not that fast.

Tests are fast but not "that" fast

```
post-editor-05-dates.js src
import React from 'react'
import {Redirect} from 'react-router'
import {savePost} from './api'

class Editor extends React.Component {
  state = {isSaving: false, redirect: false}
  handleSubmit = e => {
    e.preventDefault()
    const {title, content, tags} = e.target.elements
    const newPost = {
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t => t.trim()),
      date: new Date().toISOString(),
      authorId: this.props.user.id,
    }
    this.setState({isSaving: true})
    savePost(newPost).then(() => this.setState({redirect: true}))
  }
  render() {
    if (this.state.redirect) {
      return <Redirect to="/" />
```

```
TERMINAL   1: node
t content", "date": "2018-09-07T02:15:06.521
Z", "tags": ["tag1", "tag2"], "title": "Test
 Title"}
    as argument 1, but it was called with
      {"authorId": "user-1", "content": "Tes
t content", "date": "2018-09-07T02:15:06.507
Z", "tags": ["tag1", "tag2"], "title": "Test
 Title"}.

    Difference:

    - Expected
    + Received

      Object {
        "authorId": "user-1",
        "content": "Test content",
    -   "date": "2018-09-07T02:15:06.521Z",
    +   "date": "2018-09-07T02:15:06.507Z",
        "tags": Array [
          "tag1",
          "tag2",
        ],
        "title": "Test Title",
      }

      44 |
      45 |   expect(mockSavePost).toHaveBeen
CalledTimes(1)
    > 46 |   expect(mockSavePost).toHaveBeen
CalledWith({
      47 |     ...fakePost,
      48 |     date: new Date().toISOString(
```

post-editor-05-dates.js

```
class Editor extends React.Component {
  state = {isSaving: false, redirect: false}
  handleSubmit = e => {
    e.preventDefault()
    const {title, content, tags} =
e.target.elements
    const newPost = {
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t =>
t.trim()),
      date: new Date().toISOString(),
      authorId: this.props.user.id,
    }
    this.setState({isSaving: true})
    savePost(newPost).then(() =>
this.setState({redirect: true}))
  }
```

[00:48] Our source code is fine, but our test is having a little bit of trouble getting the date right. We need to update our assertions so that they can be more accurate with our dates. There are some libraries out there that help you fake out dates in your tests, but there's actually a pretty simple way to verify this behavior without having to do a bunch of weird things with your dates.

[01:07] That's where we're going to do. We don't really care exactly what the date actually is. We just care that it is somewhere around the time that this post was actually created. What I'm going to do is I'm going to create a range, and we're going to say `const preDate = Date.now()`.

[01:24] Then after the user clicks save, we're going to create a `const postDate = Date.now()`.

tdd-05-dates.js

```
...
const preDate = Date.now()

getByLabelText(/title/i).value = fakePost.title
getByLabelText(/content/i).value =
fakePost.content
getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
const submitButton = getByText(/submit/i)

fireEvent.click(submitButton)

expect(submitButton).toBeDisabled()

expect(mockSavePost).toHaveBeenCalledTimes(1)
expect(mockSavePost).toHaveBeenCalledWith({
  ...fakePost,
  date: new Date().toISOString(),
  authorId: fakeUser.id,
})

const postDate = Date.now()
...
```

If the date that the post is created with is between the `preDate` and the `postDate`, then that's good enough for me. Instead of the setting the date to a `new Date()` here, we're going to go ahead and we'll say expect any string.

```
expect(mockSavePost).toHaveBeenCalledWith({
  ...fakePost,
  date: expect.any(String),
  authorId: fakeUser.id,
})
```

[01:45] As long as it's a string, it can make it past that assertion. Now, let's get the date that it actually was called with. We'll say `mockSavePost.mock.calls[0][0]`, the first call, and the first argument `.date`. This is a mock function.

[02:00] It has a `mock` property. `.calls` These are the times that it was called and this is an array of its calls. `[0]` This is the first call. `[0]` This is the first argument of that call, and `.date` this is the date property of that object it was called with.

[02:13] I'm going to call that our `date`, and we're going to take that `ISOString`, and create a `new Date()` object out of that. We can call `getTime()`. That's going to give us a number, and then we can `expect(date).toBeGreaterThanOrEqual(preDate)`.

[02:29] Then we can `expect(date).toBeLessThanOrEqual(postDate)`. It's in between that range.

```
expect(mockSavePost).toHaveBeenCalledWith({
  ...fakePost,
  date: expect.any(String),
  authorId: fakeUser.id,
})

const postDate = Date.now()
const date = new Date(mockSavePost.mock.calls[0]
[0].date).getTime()
expect(date).toBeGreaterThanOrEqual(preDate)
expect(date).toBeLessThanOrEqual(postDate)
```

Now, we can save this, and our test is passing fine. In review, what we did to our implementation is we needed to add the `date` here to the `handleSubmit`. We added it as an `ISOString`, so then it could be saved to the server.

post-editor-05-dates.js

```
handleSubmit = e => {
  e.preventDefault()
  const {title, content, tags} =
e.target.elements
  const newPost = {
    title: title.value,
    content: content.value,
    tags: tags.value.split(',').map(t =>
t.trim()),
    date: new Date().toISOString(),
    authorId: this.props.user.id,
  }
```

[02:47] Then in our test, we created a date range, so before we created that `new Date()` and our after we created the `new Date()`. Then we verified that the `date` our `mockSavePost` was called with is between the `preDate` and the `postDate` range. That's good enough for us to verify that the date was created with the value it's supposed to have.

tdd-05-dates.js

```
...
const preDate = Date.now()

getByLabelText(/title/i).value = fakePost.title
getByLabelText(/content/i).value =
fakePost.content
getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
const submitButton = getByText(/submit/i)

fireEvent.click(submitButton)

expect(submitButton).toBeDisabled()

expect(mockSavePost).toHaveBeenCalledTimes(1)
expect(mockSavePost).toHaveBeenCalledWith({
  ...fakePost,
  date: expect.any(String),
  authorId: fakeUser.id,
})

const postDate = Date.now()
const date = new Date(mockSavePost.mock.calls[0]
[0].date).getTime()
expect(date).toBeGreaterThanOrEqual(preDate)
expect(date).toBeLessThanOrEqual(postDate)
...
```

Use generated data in tests with tests-data-bot to improve test maintainability

Kent C Dodds: [00:00] We have this `fakeUser` and this `fakePost`, and it has properties on there that may or may not be totally relevant to how this thing functions. One thing that I think

is important in testing is that your test communicates the things that are important for this test.

tdd-06-generate-data.js

```
const fakeUser = {id: 'user-1'}
const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
const fakePost = {
  title: 'Test Title',
  content: 'Test content',
  tags: ['tag1', 'tag2'],
}
const preDate = Date.now()
```

[00:14] Whether or not the `title` is called `'Test Title'`, that doesn't matter. That's irrelevant. Same with the `title` and the test `content`. What we're going to do is, we're going to generate these values using a library on NPM, so we can communicate to maintainers of our test that these things actually do not matter.

[00:31] The library is called `test-data-bot`. We're going `import` some things from the `test-data-bot`. We'll `import {build, fake, sequence}`. Then down here, we're going to create a few builders. We'll make a `postBuilder`, and that's going to `build('Post')`.

[00:47] It'll have some fields, `title: fake(f => f.lorem.words())`. It's going to be fake words. This `f` value is actually from another module called `faker`. It has the capability of generating a lot of random things.

tdd-06-generate-data.js

```
import {build, fake, sequence} from 'test-data-bot'
...
const postBuilder = build('Post').fields({
  title: fake(f => f.lorem.words()),
})
```

[01:02] Then we'll create `content: fake(f => f.lorem.paragraphs())`. Then we'll take `title`. This is going to be another fake value. This one is going to be an array. `tags: fake(f => [f.lorem.word(), f.lorem.word(), f.lorem.word()])`.

tdd-06-generate-data.js

```
const postBuilder = build('Post').fields({
  title: fake(f => f.lorem.words()),
  content: fake(f => f.lorem.paragraphs()),
  tags: fake(f => [f.lorem.word(),
f.lorem.word(), f.lorem.word()]),
})
```

[01:24] Then we'll make our `userBuilder`. We'll say `const userBuilder`. We'll `build` a `'User'`. That has `fields`. We only really care about one field, and that's the `id`. This one's going to be a `sequence` and that sequence number,.

tdd-06-generate-data.js

```
const userBuilder = build('User').fields({
  id: sequence(s => `user-${s}`)
})
```

[01:40] We'll say `user-${s}`. Great. Now here, our `fakeUser` can be a `userBuilder`. We can build a user. Our `fakePost` can be a `postBuilder`.

tdd-06-generate-data.js

```
const fakeUser = userBuilder()
const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
const fakePost = postBuilder()
const preDate = Date.now()
```

[01:53] Our test is failing. Now, this is supposed to be a refactor of our test, so our test shouldn't fail. What's happening here is `lorem.paragraphs()` actually returns a string that has two characters for new lines. When that value gets inserted into a text area, one of those is removed.

[02:09] What we're going to do here is, I'll say `.replace(/\r/g, '')`, globally replace all of those with just an empty string.

tdd-06-generate-data.js

```
const postBuilder = build('Post').fields({
  title: fake(f => f.lorem.words()),
  content: fake(f =>
f.lorem.paragraphs().replace(/\r/g, '')),
  tags: fake(f => [f.lorem.word(),
f.lorem.word(), f.lorem.word()]),
})
```

We save our file, and now, our test is passing. Now, if we wanted to get the values there, we can `console.log` our `fakeUser` and our `fakePost`.

[02:29] We can take a look at the output. Our `user` is an object with an `id` of `user-1`, and our `title` has some nonsense in here. If we wanted to override one of these values to be very specific, and suggest in our test that this actually does matter, then we could say `{id: 'foo'}`.

tdd-06-generate-data.js

```
const fakeUser = userBuilder({id: 'foo'})
const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
const fakePost = postBuilder()
const preDate = Date.now()
```

[02:44] Now, the ID is foo. Great. With that, we'll get rid of this id of foo, and get rid of those console.logs.

Passing Test

```
tdd-06-generate-data.js src/__tests__
})

const userBuilder = build('User').fields({
  id: sequence(s => `user-${s}`),
})

test('renders a form with title, content, tags, and a submit button', async () =
  const fakeUser = userBuilder()
  const {getByLabelText, getByText} = render(<Editor user={fakeUser} />)
  const fakePost = postBuilder()
  console.log(fakeUser)
  console.log(fakePost)
  const preDate = Date.now()

  getByLabelText(/title/i).value = fakePost.title
  getByLabelText(/content/i).value = fakePost.content
  getByLabelText(/tags/i).value = fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()
```

```
TERMINAL    1: node

RUNS src/__tests__/tdd-06-generate-data.js

Test Suites: 0 of 1 total
Tests:       0 total
Snapshots:   0 total
Time:        1s, estimated 3s
  console.log src/__tests__/tdd-06-generate-
data.js:43
    { id: 'foo' }

  console.log src/__tests__/tdd-06-generate-
data.js:44
    { title: 'consectetur rem velit',
      content: 'Voluptates qui aut et molest
iae optio. Et tempore quo sint harum laborum
 atque iusto atque earum. Ipsa itaque natus.
 Aut dolorem nihil dicta labore necessitatib
us non velit.\n Eos officia in. Enim adipisc
i non autem eaque officiis nihil neque. Ad u
t impedit unde ex earum sit adipisci. Aut et
 quibusdam aut iusto molestias quia vel occa
PASS src/__tests__/tdd-06-generate-data.js
  ✓ renders a form with title, content, tags
, and a submit button (754ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.067s, estimated 3s
Ran all test suites matching /tdd-06/i.

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

Now, our test is communicating that the `user` is not important. The post data is not important. It just needs to look something like this for our component to work properly.

tdd-06-generate-data.js

```
const postBuilder = build('Post').fields({
  title: fake(f => f.lorem.words()),
  content: fake(f =>
f.lorem.paragraphs().replace(/\r/g, '')),
  tags: fake(f => [f.lorem.word(),
f.lorem.word(), f.lorem.word()]),
})

const userBuilder = build('User').fields({
  id: sequence(s => `user-${s}`)
})
```

Test drive error state with react-testing-library

Kent C Dodds: [00:00] What happens if the `savePost` request fails? The user will be left here with a `disabled <button type="submit">`, just sitting wondering what on Earth is going on. We should probably render something to them. The first thing that we're going to do is add a new test, because this is an entirely new flow.

[00:16] Let's add another `test` here at the end that says `'renders an error message from the server'`. Now we know this is going to be an `async` test, so we'll just make that `async` right off the bet. Then we're going to be doing a lot of the same things that we did up here.

tdd-07-error-state.js

```
test('renders an error message from the server',
async () => {

})
```

[00:30] I'm going to go ahead and copy a bunch of this stuff all the way through the click, and we can copy pretty much all of this stuff. It's just down here with the `MockRedirect` that things are different.

```
test('renders an error message from the server',
async () => {
  const fakeUser = userBuilder()
  const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
  const fakePost = postBuilder()
  const preDate = Date.now()

  getByLabelText(/title/i).value =
fakePost.title
  getByLabelText(/content/i).value =
fakePost.content
  getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()

  expect(mockSavePost).toHaveBeenCalledTimes(1)
  expect(mockSavePost).toHaveBeenCalledWith({
    ...fakePost,
    date: expect.any(String),
    authorId: fakeUser.id,
  })
})
```

[00:41] Do we want to have all these same assertions in both of these tests? I would say no. We have one test here that tests the happy path, and then we have the rest of our tests with assertions that are specific for their use case.

[00:53] I'm going to get rid of these assertions here. I'll get rid of that `expect(submitButton).toBeDisabled()`, I'll get rid of that `preDate`, but we'll go ahead and keep everything else. Once the `click` event has been fired, then I'm going to want to get the `postError` node that we're going to be rendering.

```
test('renders an error message from the server',
async () => {
  const fakeUser = userBuilder()
  const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
  const fakePost = postBuilder()

  getByLabelText(/title/i).value =
fakePost.title
  getByLabelText(/content/i).value =
fakePost.content
  getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)
})
```

[01:07] I'll say `postError` equals, and we're going to need to wait for an element to appear on the page. I'll say `await`, and we're going to use the `waitForElement` function from `react-testing-library`. We're going to `await waitForElement`, and we'll pass our callback that should return an element when it appears.

```
const postError = await waitForElement(() => )
```

[01:24] We're going to go ahead and `getByTestId` and we'll have an element with a testID of `post-error`. Once that's been rendered, we can `expect(postError).toHaveTextContent('test error')`. Now we need to make sure that our `mockSavePost` rejects the promise instead of resolves it.

```
const postError = await waitForElement(() =>
getByTestId('post-error'))
expect(postError).toHaveTextContent('test
error')
```

[01:45] Right now, our `mockSavePost` has a default implementation to `resolve` the promise, so we want this to be rejected, but we can't do that because then that would break our other tests. We'll leave this as it is, and then in here right at the top, we'll say `mockSavePost.mockRejectedValueOnce()` that will override the default implementation just for one time.

```
test('renders an error message from the server',
async () => {
  mockSavePost.mockRejectedValueOnce()
  ...

  const postError = await waitForElement(() =>
getByTestId('post-error'))
  expect(postError).toHaveTextContent('test
error')
})
```

[02:08] Then we can say data error-test error. We'll want this mock to resemble exactly what the server would send back in this API call. With that all established, let's go ahead and run our test and we can get our red test.

```
test('renders an error message from the server',
async () => {
  mockSavePost.mockRejectedValueOnce({data:
{error: 'test error'}})
  ...

  const postError = await waitForElement(() =>
getByTestId('post-error'))
  expect(postError).toHaveTextContent('test
error')
})
```

[02:22] Let's go ahead and make this test pass by changing the implementation slightly. We need to add an error callback, and here's our failure. We're going to get a `response`, and we'll call `this.setState({error: response.data.error})`.

[02:40] Then we'll want to add some state here for `error`, we'll set that to `null` by default. We'll scroll down here and we'll add an `error` right here, so we'll say `this.state.error`. If there is an error, then we're going to render `<div>{this.state.error}</div>`, so the message that came from the server.

[03:00] Remember our test wants to be able to find this node, so we're going to add a `data-testid="post-error"` to the `div`. If there is no error, we'll just render `null`.

post-editor-07-error-state.js

```
class Editor extends React.Component {
  state = {isSaving: false, redirect: false,
error: null} // added error to state
  handleSubmit = e => {
    e.preventDefault()
    const {title, content, tags} =
e.target.elements
    const newPost = {
      title: title.value,
      content: content.value,
      tags: tags.value.split(',').map(t =>
t.trim()),
      date: new Date().toISOString(),
      authorId: this.props.user.id,
    }
    this.setState({isSaving: true})
    savePost(newPost).then(
      () => this.setState({redirect: true}),
      response => this.setState({error:
response.data.error}) // added an error response
    )
  }
  render() {
    if (this.state.redirect) {
      return <Redirect to="/" />
    }
    return (
      <form onSubmit={this.handleSubmit}>
        <label htmlFor="title-
input">Title</label>
        <input id="title-input" name="title" />

        <label htmlFor="content-
input">Content</label>
        <textarea id="content-input"
name="content" />
```

```
          <label htmlFor="tags-input">Tags</label>
          <input id="tags-input" name="tags" />

          <button type="submit" disabled=
{this.state.isSaving}>
            Submit
          </button>
          {this.state.error ? (
            <div data-testid="post-error">
{this.state.error}</div>
           ) : null} {/* render the error */}
        </form>
      )
    }
}
```

That gets our test passing. Now let's go ahead and see if there's anything we'd like to refactor here about our test or our implementation.

[03:20] I think first our implementation looks pretty good, I don't see any reason to refactor this. Our test though, I think I would like to get that content here to illustrate that not only are these things coincidently the same, they actually should be the same.

tdd-07-error-state.js

```
test('renders an error message from the server',
async () => {
  mockSavePost.mockRejectedValueOnce({data:
{error: 'test error'}})
  ...
  expect(postError).toHaveTextContent('test
error')
})
```

[03:34] I'm going to make a variable called `testError`, I'll say `testError`, and the error will be `testError`, and the text content should be the same thing that I get back from my mock rejected value.

```
test('renders an error message from the server',
async () => {
  const testError = 'test error'
  mockSavePost.mockRejectedValueOnce({data:
{error: testError}})
  ...
  expect(postError).toHaveTextContent(testError)
})
```

Now there's one other assertion that I think I want to put in here, and that is if there's an error, then I want the `submitButton` to no longer be disabled.

[03:54] We have an assertion up here that it is disabled `expect(submitButton).toBeDisabled()`, I want to make sure that it's not disabled when the error happens so the user can try again. I'll add `.not.toBeDisabled()` to `expect(submitButton)`. I'll save that, we're going to get another red test.

```
test('renders an error message from the server',
async () => {
  const testError = 'test error'
  mockSavePost.mockRejectedValueOnce({data:
{error: testError}})
  ...
  expect(postError).toHaveTextContent(testError)
  expect(submitButton).not.toBeDisabled()
})
```

Let's make that green by updating the `state` here, `isSaving: false`, and now our tests are all green.

post-editor-07-error-state.js

```
savePost(newPost).then(
  () => this.setState({redirect: true}),
  response => this.setState({error:
response.data.error}) // added an error response
)
```

[04:14] In review here, we created a new test for an error message, we mocked the rejected value one time. This `mockSavePost` will now return a rejected promise with this `{data: {error: testError}}` value for the next time it's called. Then we went ahead and fired the submit event, and then we waited for the `mockSavePost` to reject the promise, and re-render our component to have that `postError`.

post-editor-07-error-state.js

```
test('renders an error message from the server',
async () => {
  const testError = 'test error'
  mockSavePost.mockRejectedValueOnce({data:
{error: testError}})
  const fakeUser = userBuilder()
  const {getByLabelText, getByText, getByTestId}
= render(
    <Editor user={fakeUser} />,
  )
  const fakePost = postBuilder()

  getByLabelText(/title/i).value =
fakePost.title
  getByLabelText(/content/i).value =
fakePost.content
  getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  const postError = await waitForElement(() =>
getByTestId('post-error'))
  expect(postError).toHaveTextContent(testError)
  expect(submitButton).not.toBeDisabled()
})
```

[04:37] We verified that the `postError` was rendering what the server sent back, and we verified that the `submitButton` was no longer disabled. In our implementation, we added an error handler setting `isSaving` to `false`, and our error to whatever

comes back from the server. Then in here, we say `this.state.error`, if there is an error, then we'll render that error, otherwise we'll render `null`.

post-editor-07-error-state.js

```
class Editor extends React.Component {
  state = {isSaving: false, redirect: false,
error: null} // added error to state
  handleSubmit = e => {
    ...
    savePost(newPost).then(
      () => this.setState({redirect: true}),
      response => this.setState({error:
response.data.error}) // added an error response
    )
  }
  render() {
    if (this.state.redirect) {
      return <Redirect to="/" />
    }
    return (
      <form onSubmit={this.handleSubmit}>
        ...
        {this.state.error ? (
          <div data-testid="post-error">
{this.state.error}</div>
         ) : null} {/* render the error */}
      </form>
    )
  }
}
```

Write a custom render function to share code between tests and simplify tests

Kent C Dodds: [00:00] Our component's finished, and it's fully tested, but there's still one last thing I want to refactor about our tests before we move on. That is that I see we have quite a bit of duplicate logic between both of these tests.

[00:12] It would be nice if we could get rid of that and shove it off to the side, so that people who come in to maintain these tests will be able to quickly identify what are the differences between test one and test two?

tdd-08-custom-render.js

```
test('renders a form with title, content, tags,
and a submit button', async () => {
  const fakeUser = userBuilder()
  const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
  const fakePost = postBuilder()
  const preDate = Date.now()

  getByLabelText(/title/i).value =
fakePost.title
  getByLabelText(/content/i).value =
fakePost.content
  getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  expect(submitButton).toBeDisabled()
```

```
  expect(mockSavePost).toHaveBeenCalledTimes(1)
  expect(mockSavePost).toHaveBeenCalledWith({
    ...fakePost,
    date: expect.any(String),
    authorId: fakeUser.id,
  })

  const postDate = Date.now()
  const date = new
Date(mockSavePost.mock.calls[0]
[0].date).getTime()
  expect(date).toBeGreaterThanOrEqual(preDate)
  expect(date).toBeLessThanOrEqual(postDate)

  await wait(() =>
expect(MockRedirect).toHaveBeenCalledTimes(1))

  expect(MockRedirect).toHaveBeenCalledWith({to:
'/'}, {})
})

test('renders an error message from the server',
async () => {
  const testError = 'test error'
  mockSavePost.mockRejectedValueOnce({data:
{error: testError}})
  const fakeUser = userBuilder()
  const {getByLabelText, getByText, getByTestId}
= render(
    <Editor user={fakeUser} />,
  )
  const fakePost = postBuilder()

  getByLabelText(/title/i).value =
fakePost.title
  getByLabelText(/content/i).value =
fakePost.content
  getByLabelText(/tags/i).value =
```

```
fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)

  fireEvent.click(submitButton)

  const postError = await waitForElement(() =>
getByTestId('post-error'))
  expect(postError).toHaveTextContent(testError)
  expect(submitButton).not.toBeDisabled()
})
```

[00:22] We're going to go ahead and take lots of this, and put it into a setup function. I'm going to call it `renderEditor`. It's not going to take any arguments, and we're going to just move a whole bunch of this stuff up here, all of this setup for our editor.

```
function renderEditor() {
  const fakeUser = userBuilder()
  const {getByLabelText, getByText} =
render(<Editor user={fakeUser} />)
  const fakePost = postBuilder()

  getByLabelText(/title/i).value =
fakePost.title
  getByLabelText(/content/i).value =
fakePost.content
  getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
  const submitButton = getByText(/submit/i)
}
```

[00:37] We're going to get rid of the `preDate`, because that's specific to only one of our tests, and then we're going to `return` everything that we would need for both of these tests. First,

instead of destructuring this `render`, I'm going to assign this to `utils`.

[00:49] Then I'll spread `utils` here in the `return`, so we have access to those `utils` in our tests. Then I'll use `utils` for each one of these `getByLabelText`, and `utils` here before `getByText` as well. Then we'll also return the `submitButton`, our `fakeUser`, and our `fakePost`.

```
function renderEditor() {
  const fakeUser = userBuilder()
  const utils = render(<Editor user={fakeUser}
/>)
  const fakePost = postBuilder()

  utils.getByLabelText(/title/i).value =
fakePost.title
  utils.getByLabelText(/content/i).value =
fakePost.content
  utils.getByLabelText(/tags/i).value =
fakePost.tags.join(', ')
  const submitButton =
utils.getByText(/submit/i)
  return {
    ...utils,
    submitButton,
    fakeUser,
    fakePost
  }
}
```

[01:06] With that now, we can remove a whole bunch of this setup, and instead call `renderEditor`. That'll give us back our `submitButton`. We also need the `fakePost` and `fakeUser`. Then

we need to create that `preDate` again.

```
test('renders a form with title, content, tags,
and a submit button', async () => {
  const {submitButton, fakePost, fakeUser} =
renderEditor()
  const preDate = Date.now()

  ...
})
```

[01:21] We can save this, and our test is still passing. Perfect. It was a good refactor. Now, we can do the same thing here. We'll get rid of this duplication. Instead, we'll get the `submitButton` from `renderEditor`. We'll also want the `getByTestId` query.

```
test('renders an error message from the server',
async () => {
  const testError = 'test error'
  mockSavePost.mockRejectedValueOnce({data:
{error: testError}})
  const {submitButton, getByTestId} =
renderEditor()

  fireEvent.click(submitButton)

  const postError = await waitForElement(() =>
getByTestId('post-error'))
  expect(postError).toHaveTextContent(testError)
  expect(submitButton).not.toBeDisabled()
})
```

[01:40] That's a solid refactor. Both of our tests are still green, and the code is a lot more terse, making it easier for people to come in here and say, "Oh, the difference here is that the `mockSavePost` rejects the value, and that a `postError` is rendered."

[01:55] Whereas before, they had to wade through all of this stuff that may or may not be relevant for this specific test. The same goes for this test of the happy path. One other thing I'd like to mention here is that you could have multiple renders.

[02:07] If you have many tests for a specific component, you can have different forms of renders. You could also take arguments here in the `renderEditor`, pass on some parameters, and take those for how a component should be rendered. The sky's the limit for you on how you want to do this.

Test React components that use the react-router Router Provider with createMemoryHistory

Kent C Dodds: [00:00] Here, we have this `Main` component that's rendering a `Link`, a `Switch`, and a `Route`, all from `react-router-dom`. We want to be able to test this `Main` component so that I can click on the `Link` and have that render the `Home` component. I click on this `<Link to="/about">`, it renders the `About` component, and if I go to a `Route` that isn't supported, I go to this `NoMatch` component.

main.js

```
import {Switch, Route, Link} from 'react-router-
dom'

const About = () => (
  <div data-testid="about-screen">You are on the
about page</div>
)
const Home = () => <div data-testid="home-
screen">You are home</div>
const NoMatch = () => <div data-testid="no-
match-screen">No Match</div>

function Main() {
  return (
    <div>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Switch>
        <Route exact path="/" component={Home}
/>
        <Route path="/about" component={About}
/>
        <Route component={NoMatch} />
      </Switch>
    </div>
  )
}
```

[00:20] In `react-router.js`, let's go ahead and `import React from 'react'`. We'll `import {render} from 'react-testing-library'`, and then we'll `import {Main} from '../main'`. I'm going to `test` that `'main renders about and home and I can navigate to those pages'`

react-router.js

```
import 'jest-dom/extend-expect'
import 'react-testing-library/cleanup-after-
each'

import React from 'react'
import {render} from 'react-testing-library'
import {Main} from '../main'

test('main renders about and home and I can
navigate to those pages', () => {

})
```

[00:42] First thing I'm going to do is I'll `render(<Main />)`. I'm going to `getByTestId`, which is how I'm going to know which page I'm looking at. I also want to be able to click on the `Link` components, so I'll `getByText`.

[00:57] I should be rendering the `Home` page right from the start. I'm starting on the path of `/` so it should render my `Home`. I'm going to add an assertion to `expect(getByTestId('home-screen').toBeInTheDocument()`.

[01:12] Then I'll `fireEvent.click(getByText())`. We want to click on this `<Link to="/about">`, so I'll `getByText(/about/i)`. We're going to need that `fireEvent` from `react-testing-library`. Then I'll `expect(getByTestId('about-screen').toBeInTheDocument()`.

react-router.js

```
import 'jest-dom/extend-expect'
import 'react-testing-library/cleanup-after-
each'

import React from 'react'
import {render, fireEvent} from 'react-testing-
library'
import {Main} from '../main'

test('main renders about and home and I can
navigate to those pages', () => {
  const {getByTestId, getByText} = render(<Main
/>)
  expect(getByTestId('home-
screen').toBeInTheDocument()
  fireEvent.click(getByText(/about/i))
  expect(getByTestId('about-
screen').toBeInTheDocument()
})
```

[01:32] I could also verify that the `Home` screen has been removed, so `.not.toBeInTheDocument`. If we're going to do that, then we need to get the `queryByTestId`. For good measure, I'll do the same here, `queryByTestId .not.toBeInTheDocument`.

```
test('main renders about and home and I can
navigate to those pages', () => {
  const {getByTestId, queryByTextId, getByText}
= render(<Main />)
  expect(getByTestId('home-
screen').toBeInTheDocument()
  expect(queryByTestId('about-
screen').not.toBeInTheDocument()
  fireEvent.click(getByText(/about/i))
  expect(queryByTestId('home-
screen').not.toBeInTheDocument()
  expect(getByTestId('about-
screen').toBeInTheDocument()
})
```

[01:51] We've got a pretty good test here. Let's go ahead and open up our test. Wow, we've got a bunch of errors. Here's the problem. When you render a component that uses `Link`, `Switch`, or `Route` from `react-router-dom`, these components are going to be looking for a `Router` in the tree. We don't have a `Router` in the tree.

[02:09] This is a really common problem for components that use `react-router-dom` components. We need to render this within a React `Router` so they have access to the React `Router` context. Let's go ahead and do that.

[02:20] We'll `import {Router} from 'react-router-dom'`. Normally, you're going to be using a `BrowserRouter`, but we're going to use `Router` directly so that we can provide our own `history` object. We `import {createMemoryHistory} from`

'history'. Then we'll create our history from createMemoryHistory. Here, we can specify our initialEntries.

```
import {Router} from 'react-router-dom'
import {createMemoryHistory} from 'history'

test('main renders about and home and I can
navigate to those pages', () => {
  const history =
createMemoryHistory({initialEntries: ['/']})
  const {getByTestId, queryByTextId, getByText}
= render(<Main />)
  expect(getByTestId('home-
screen').toBeInTheDocument()
  expect(queryByTestId('about-
screen').not.toBeInTheDocument()
  fireEvent.click(getByText(/about/i))
  expect(queryByTestId('home-
screen').not.toBeInTheDocument()
  expect(getByTestId('about-
screen').toBeInTheDocument()
})
```

[02:44] I'll say our initialEntries are /, so that we start out on the Home screen. Then we'll take this Main and we'll actually render it in a Router that has the history object history. We'll render Main inside of that Router. We'll save here, pop open our tests. Our tests are working.

```
test('main renders about and home and I can
navigate to those pages', () => {
  const history =
createMemoryHistory({initialEntries: ['/']})
  const {getByTestId, queryByTextId, getByText}
= render(
    <Router history={history}>
      <Main />
    </Router>,
  )
  expect(getByTestId('home-
screen').toBeInTheDocument()
  expect(queryByTestId('about-
screen').not.toBeInTheDocument()
  fireEvent.click(getByText(/about/i))
  expect(queryByTestId('home-
screen').not.toBeInTheDocument()
  expect(getByTestId('about-
screen').toBeInTheDocument()
})
```

[03:02] Let's make sure that they can break. I'll remove this `not` here, and perfect. Our assertions are running.

[03:09] In review, the reason that we had to render the `Main` within the `Router` is because `Main` is using components that rely on a `Router`'s context to be rendered into the tree so we can render that `Router` ourselves.

[03:22] We're using `react-router-dom`'s `Router` component, which is the base component here, so that we can provide our own `history`. That history is a `createMemoryHistory`. It's a

history that lives in memory. It's not actually a browser history. We have some fine-grained control over the `initialEntries` for our history, so we can start out on any page that we like.

[03:40] With that `history`, we render that `Router` with that `history`, then render the `Main`, and all of the components inside of `Main` are going to work. Then we made a couple of assertions for whether the `Home` screen or `About` screen appear on the page.

[03:54] We click on the `<Link to="/about">`, and then we verify that the `Home` screen is no longer rendered, and that the `About` screen is rendered.

Initialize the `history` object with a bad entry to test the react-router no-match route

Kent C Dodds: [00:00] We've proven that we can navigate to the `home-screen` and `about-screen`. What happens if the user lands on a URL that is not supported?

react-router-02.js

```
test('main renders about and home and I can
navigate to those pages', () => {
  const history =
createMemoryHistory({initialEntries: ['/']})
  const {getByTestId, queryByTextId, getByText}
= render(
    <Router history={history}>
      <Main />
    </Router>,
  )
  expect(getByTestId('home-
screen').toBeInTheDocument()
  expect(queryByTestId('about-
screen').not.toBeInTheDocument()
  fireEvent.click(getByText(/about/i))
  expect(queryByTestId('home-
screen').not.toBeInTheDocument()
  expect(getByTestId('about-
screen').toBeInTheDocument()
})
```

[00:07] We have this `NoMatch` component that will render `No match` if the user lands on that page. Let's go ahead and land the user on that page in our test and verify that the `NoMatch` screen is showing up. That way we can catch the scenario if somebody typos the `NoMatch` on our route configuration.

main.js

```
const NoMatch = () => <div data-testid="no-
match-screen">No Match</div>
```

[00:23] To do this, in `react-router-02.js` I am going to add a new `test`. We'll say `'landing on a bad page shows no match component'`. We'll do much of the same stuff we did before, so I am going to pull in all this. We'll paste that in here, except we'll make the `initialEntries '/something-that-does-not-match'`.

react-router-02.js

```
test('landing on a bad page shows no match
component', () => {
  const history =
createMemoryHistory({initialEntries:
['/something-that-does-not-match']})
  const {getByTestId, queryByTextId, getByText}
= render(
    <Router history={history}>
      <Main />
    </Router>,
  )
})
```

[00:43] Then we can simply `expect(getByTestId('no-match-screen').toBeInTheDocument()`. We won't need these queries here anymore. Get rid of those.

[00:56] Let's check out our test. Cool, it's passing.

```
test('landing on a bad page shows no match
component', () => {
  const history =
createMemoryHistory({initialEntries:
['/something-that-does-not-match']})
  const {getByTestId} = render(
    <Router history={history}>
      <Main />
    </Router>,
  )
  expect(getByTestId('no-match-
screen').toBeInTheDocument()
})
```

Let's verify that it's actually running our assertion, so we'll say `.not`.

```
expect(getByTestId('no-match-
screen').not.toBeInTheDocument()
```

[01:03] Our assertion is running because the test can fail. In review, to make this work we just started our history with the `initialEntries: ['/something-that-does-not-match']` and then the `Router` not finding a match for this initial route, rendered our `NoMatch` screen.

Create a custom render function to simplify tests of react-router components

Kent C Dodds: [00:00] Here we have a little bit of duplication between both of these tests. They're both creating a `history` and they're both rendering the `Main` within a `Router`, but I don't want

these tests to have to know that the `Main` needs to be rendered within a `Router` to work, and I definitely don't want to have to `createMemoryHistory` every single time I want to render a component that needs to have a `Router`.

react-router-03.js

```
test('landing on a bad page shows no match
component', () => {
  const history =
createMemoryHistory({initialEntries:
['/something-that-does-not-match']})
  const {getByTestId} = render(
    <Router history={history}>
      <Main />
    </Router>,
  )
  expect(getByTestId('no-match-
screen').toBeInTheDocument()
})
```

[00:18] I'm going to make a function here that will `render` my `ui` inside of a `Router` creating its own `history`, so that way throughout my test base I don't have to worry about whether a component needs to be rendered within the `Router`.

[00:29] Let's go ahead and I'm going to make a new `render` function here, and it's going to take a `ui`, and some `options`, and it's going to return whatever this imported `render` returns. We're going to have to alias this as `rtl-render`, and then we'll `return rtlRender(ui, options)`.

```
import {render as rtl-render, fireEvent} from
'react-testing-library'

function render(ui, options) {
  return rtlRender(ui, options)
}
```

[00:46] So far this doesn't actually make any difference. Now I'm going to go down here and we'll take this `history`, and we're going to allow this route to be configured. We'll say `options.route`, otherwise we'll default to a slash if the option isn't provided.

```
function render(ui, options) {
  const history =
createMemoryHistory({initialEntries:
[options.route || '/']})
  return rtlRender(ui, options)
}
```

[01:00] We'll also make `options` optional, by making that a default of an empty object. Then we're going to take this `Router` and we'll `render` that in our `rtl-render`. Instead of `Main`, we'll `render` whatever `ui` we're given.

```
function render(ui, options = {}) {
  const history =
createMemoryHistory({initialEntries:
[options.route || '/']})
  return rtlRender(<Router history={history}>
      {ui}
    </Router>, options)
}
```

[01:12] Now for this first test, we can remove the `Router` and just `render` `Main`, but for this second test, it's going to be a little bit different because our `Router` needs a special `history` that has some initial entries for something that does not match. I'll cut that, we'll get rid of the `history`, we'll get rid of the `Router` and the `Main`, and here in the `render` function we provided that `options.route`.

[01:33] I'm going to say as my second argument, pass some `options` for route, and that is the route that this is going to be rendered at.

```
test('main renders about and home and I can
navigate to those pages', () => {
  const {getByTestId, queryByTextId, getByText}
= render( <Main /> )
  expect(getByTestId('home-
screen').toBeInTheDocument()
  expect(queryByTestId('about-
screen').not.toBeInTheDocument()
  fireEvent.click(getByText(/about/i))
  expect(queryByTestId('home-
screen').not.toBeInTheDocument()
  expect(getByTestId('about-
screen').toBeInTheDocument()
})

test('landing on a bad page shows no match
component', () => {
  const {getByTestId} = render(
    <Main />,
    { route: '/something-that-does-not-match' }
  )
  expect(getByTestId('no-match-
screen').toBeInTheDocument()
})
```

We'll save that, and open up our test and our tests are still passing. Let's go ahead and clean this up just a little bit, and add a couple of features that might be useful for other people who will be using this `render` method in the future.

[01:52] I'm going to cut `rtlRender`, I'll return an object and I'll spread that value across. I'll also return `history`, that way people can make assertions on the `history` object we created for them if

they need. Then I'm going to destructure the `route` so that we're not passing the `route` onto the `options` for `rtlRender`.

[02:10] We'll destructure this, we'll take all the `options`, then we'll specify the `route` and default that to a slash. Then we can just provide the route and not worry about the *or* here.

```
function render(ui, {route = '/', ...options} =
{}) {
  const history =
createMemoryHistory({initialEntries: [route]})
  return {
    ...rtlRender(<Router history={history}>{ui}
</Router>, options),
    history,
  }
}
```

[02:21] I also want to give people the flexibility to provide their own `history` object as well. I'm going to add `history` to our destructured `options` here, then I can remove that line and we'll save this, open up our terminal, and everything is still passing again.

```
function render(
  ui,
  {
    route = '/',
    history =
createMemoryHistory({initialEntries: [route]}),
    ...options
  } = {},
) {
  return {
    ...rtlRender(<Router history={history}>{ui}
</Router>, options),
    history,
  }
}
```

[02:35] In review, the reason that we did this isn't because we had two tests that had a little bit of boilerplate. The reason that we did this is because we're going to be rendering components inside of our application that use React Router all over the place.

[02:47] It's nice to not have to worry about whether or not we need to `render` that component with the `Router`, especially if in the future we start adding links and routes to different components, we don't want our test to break for those use cases.

[02:58] We'll just `render` everything with the `Router`, and we just use this `render` method instead of the react testing library `render` method for all of our tests. I would recommend that you put this `render` method inside a test utilities module, and make that accessible throughout your codebase so you can `import` that instead of `react-testing-library`.

[03:15] What we did here was we created our own `render` method that renders our `ui` inside of a `Router` with a `history` that we can provide or default to a created one that has a route that you can either provide or will default you to the home route. Then we also return the `history` that we created for you, as well as all the utilities that react testing library will return for you.

Test a redux connected React Component

Kent C Dodds: [00:00] Here we have a simple Redux app file with a `counter.js` file on it that has a `Counter`, that has `increment` and `decrement` using `dispatch` from `react-redux`.

redux-app.js

```
class Counter extends React.Component {
  increment = () => {
    this.props.dispatch({type: 'INCREMENT'})
  }

  decrement = () => {
    this.props.dispatch({type: 'DECREMENT'})
  }
  ...
}
```

This has an `onClick` for `decrement` and `increment`, a minus, and a plus here.

```
class Counter extends React.Component {
  ...
  render() {
    return (
      <div>
        <h2>Counter</h2>
        <div>
          <button onClick={this.decrement}>-
</button>
          <span data-testid="count-value">
{this.props.count}</span>
          <button onClick={this.increment}>+
</button>
        </div>
      </div>
    )
  }
}
```

Then we create a `ConnectedCounter` using the `connect` from `react-redux`.

```
const ConnectedCounter = connect(state =>
({count: state.count}))(Counter)
```

[00:20] Then we have a `reducer`. We have that `initialState` of `count: 0` have a `reducer` that handles `increment` and `decrement`, and we're exporting both of those.

```
const initialState = {count: 0}
function reducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return {
        count: state.count + 1,
      }
    case 'DECREMENT':
      return {
        count: state.count - 1,
      }
    default:
      return state
  }
}

export {ConnectedCounter, reducer}
```

Normally, here is where you'd render your `react-redux` `Provider` and `render` your application inside of there, but we're going to skip that for our tests.

[00:37] I want to be able to test this `Counter` component to make sure that its logic is correct, and I also want to be able to test my Redux `reducer`, but the user doesn't care at all that I'm using Redux under the hood, and neither should my test. We're going to test this `ConnectedCounter` in a way that is ambivalent to Redux.

[00:55] To get started, I'm going to add this test that says it `'can render with redux with defaults'`. Then, I'm going to use the `render` method from `react-testing-library`. I'll `import {render} from 'react-testing-library'`.

redux-01.js

```
import {render} from 'react-testing-library'

test('can render with redux with defaults', ()
=> {
  render
})
```

[01:12] I'm also going to want to render that `{ConnectedCounter} from '../redux-app'`, and we'll `render(<ConnectedCounter />)`. We're going to need to `import React from 'react'`. I'm going to need to select a couple of these elements.

```
import React from 'react'
import {render} from 'react-testing-library'
import {ConnectedCounter} from '../redux-app'

test('can render with redux with defaults', ()
=> {
  render(<ConnectedCounter />)
})
```

[01:27] We've got a `data-testid` on this `'count-value'` here, so we can verify the `'count-value'`.
Then can select by the text for the minus and the plus. Let's get those utilities here.

[01:37] We'll `getByText` and `getByTestId`, and we're going to need a `fireEvent`. We'll `fireEvent.click(getByText('+'))` and we'll `expect(getByTestId('count-`

value')).toHaveTextContent('1').

```
import React from 'react'
import {render, fireEvent} from 'react-testing-
library'
import {ConnectedCounter} from '../redux-app'

test('can render with redux with defaults', ()
=> {
  const {getByText, getByTestId} =
render(<ConnectedCounter />)
  fireEvent.click(getByText('+'))
  expect(getByTestId('count-
value')).toHaveTextContent('1')
})
```

[02:03] Great. My text is actually broken. That's because I'm rendering the `ConnectedCounter` outside of a `Provider` and we need to render within a Redux `Provider` that provides the `store` which will respond to these `dispatch` calls.

[02:17] We need that `store` to use this `reducer` and we're exporting that `reducer`. Let's go ahead and we'll pull in that `reducer`. I'm also going to `import {Provider} from 'react-redux'`, and I'll `import {createStore} from 'redux'`.

```
import React from 'react'
import {createStore} from 'redux'
import {Provider} from 'react-redux'
import {render, fireEvent} from 'react-testing-
library'
import {redux, ConnectedCounter} from '../redux-
app'
```

[02:33] Let's make our `store` with `createStore(reducer)`. Then, we'll render our `ConnectedCounter` inside of a `Provider` which has that `store` provided. With that, our tests are now passing. This not only tests our `ConnectedCounter` component itself and the `increment` method that it has, but it also tests our `reducer` and the `increment` case in our `switch` statement.

redux-01.js

```
test('can render with redux with defaults', ()
=> {
  const store = createStore(reducer)
  const {getByText, getByTestId} = render(
    <Provider store={store}>
      <ConnectedCounter />
    </Provider>
  )
  fireEvent.click(getByText('+'))
  expect(getByTestId('count-
value')).toHaveTextContent('1')
})
```

redux-app.js

```
const ConnectedCounter = connect(state =>
({count: state.count}))(Counter)

const initialState = {count: 0}
function reducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return {
        count: state.count + 1,
      }
    case 'DECREMENT':
      return {
        count: state.count - 1,
      }
    default:
      return state
  }
}
```

[02:56] The really nice thing about this is, well, it's not testing in isolation, it's testing the integration which is a great thing, because now we know that we're connecting this component properly and that the logic in our `reducer` is wired up properly for the logic in our `render` method.

[03:11] We're getting a lot of coverage and all that takes is rendering our `Provider` with the `ConnectedCounter`.

Test a redux connected React Component with initialized state

Kent C Dodds: [00:00] Next let's go ahead and write a test that initializes the `store` with something. I'm going to go ahead and copy this and we'll paste it here and we'll name it `'can render`

`with redux with custom initial state'`.

[00:12] We'll initialize that with `{count: 3}`, and then instead of incrementing we'll decrement and verify that the `count` value goes from `3` to `2`. We'll save this, pull up our test, and our test is passing.

redux-02.js

```
test('can render with redux with custom initial
state', () => {
  const store = createStore(reducer, {count: 3})
  const {getByText, getByTestId} = render(
    <Provider store={store}>
      <ConnectedCounter />
    </Provider>
  )
  fireEvent.click(getByText('-'))
  expect(getByTestId('count-
value')).toHaveTextContent('2')
})
```

[00:25] Now, this example is a little bit contrived. I probably wouldn't actually write a test like this. In fact, I'd probably just move this `fireEvent` up here to the other test and I'd verify that the count value goes from `1` to `0` and that would be enough to test this component.

```
test('can render with redux with defaults', ()
=> {
  const store = createStore(reducer)
  const {getByText, getByTestId} = render(
    <Provider store={store}>
      <ConnectedCounter />
    </Provider>
  )
  fireEvent.click(getByText('+'))
  expect(getByTestId('count-
value')).toHaveTextContent('1')
  fireEvent.click(getByText('-'))
  expect(getByTestId('count-
value')).toHaveTextContent('0')
})
```

[00:39] But the reason that I'm doing this here is to show you that you can initialize the `store` with any state, and that can help you get started with your test really quickly to test a specific edge case. So we'll leave this here.

```
test('can render with redux with custom initial
state', () => {
  const store = createStore(reducer, {count: 3})
  const {getByText, getByTestId} = render(
    <Provider store={store}>
      <ConnectedCounter />
    </Provider>
  )
  fireEvent.click(getByText('-'))
  expect(getByTestId('count-
value')).toHaveTextContent('2')
})
```

Create a custom render function to simplify tests of redux components

Kent C Dodds: [00:00] If you're building an app with Redux, you're probably going to have a lot of components that are connected to Redux. This logic we're doing here to create a new `store` and `render` our connected component with that `store` is probably going to be something you're going to be doing a lot in your tests.

redux-03.js

```
test('can render with redux with custom initial
state', () => {
  const store = createStore(reducer, {count: 3})
  const {getByText, getByTestId} = render(
    <Provider store={store}>
      <ConnectedCounter />
    </Provider>
  )
  fireEvent.click(getByText('-'))
  expect(getByTestId('count-
value')).toHaveTextContent('2')
})
```

[00:14] Let's make a utility `render` function that we'll use to `render` our connected components with the Redux `Provider`. Here, I'm going to make a new `function`. It's going to be called `render`, and it'll take a `ui` and some `options`. Let's alias `render` from `react-testing-library` to `rtlRender`. Then, we'll return `rtlRender` with that `ui` and those `options`.

```
import {render as rtlRender, fireEvent} from
'react-testing-library'

function render(ui, options) {
  return rtlRender(ui, options)
}
```

[00:34] So far, we're doing exactly the same thing as it was doing before, but now I'm going to move this `createStore` logic up into this new `render` function. I'm going to move this `Provider` into the `render` function too. Instead of just rendering the `connectedCounter`, we'll `render` the `ui` that were given. We don't need to `render` the `Provider` here, or here.

```
function render(ui, options) {
  const store = createStore(reducer)
  return rtlRender(<Provider store={store}>
    {ui}
  </Provider>, options)
}

test('can render with redux with defaults', ()
=> {
  const {getByText, getByTestId} =
render(<ConnectedCounter />)
  fireEvent.click(getByText('+'))
  expect(getByTestId('count-
value')).toHaveTextContent('1')
})

test('can render with redux with custom initial
state', () => {
  const {getByText, getByTestId} =
render(<ConnectedCounter />)
  fireEvent.click(getByText('-'))
  expect(getByTestId('count-
value')).toHaveTextContent('2')
})
```

[00:53] We'll rerun our tests and our test is actually broken. That's because we expected to be able to have some `initialState` in this one, that `{count: 3}`.

[01:07] We'll just go ahead and make that an option as `initialState` and in here that is `{count: 3}`.

```
test('can render with redux with custom initial
state', () => {
  const {getByText, getByTestId} =
render(<ConnectedCounter />, {initialState: 3})
  fireEvent.click(getByText('-'))
  expect(getByTestId('count-
value')).toHaveTextContent('2')
})
```

Then, we'll accept `initialState` as an option here. We'll say `options.initialState`. We'll default that `options` to an empty object, so you don't have to pass `options` if you don't need to. We'll save that, and our tests are passing.

```
function render(ui, options = {}) {
  const store = createStore(reducer,
options.initialState)
  return rtlRender(<Provider store={store}>
    {ui}
  </Provider>, options)
}
```

[01:26] Let's go head and clean this up just a little bit. I don't want to pass this `initialState` to the `rtlRender`. I'm going to destructure that `initialState` off. We'll take those `options` and we'll pass `initialState` directly to our `createStore` call here. We'll save, and our tests are still passing.

```
function render(ui, {initialState, ...options} =
{}) {
  const store = createStore(reducer,
initialState)
  return rtlRender(<Provider store={store}>{ui}
</Provider>, options)
}
```

[01:45] We'll make this even more useful by allowing users to provide their own `store` implementation. We'll remove that `createStore` here from our function body. We'll save and now the `render` function is capable of not only allowing people to provide their own `initialState`, but they can also provide their own `store` if they need to with their own reducer or `initialState`.

```
function render(
  ui,
  {initialState, store = createStore(reducer,
initialState), ...options} = {}
) {
  return rtlRender(<Provider store={store}>{ui}
</Provider>, options)
}
```

[02:05] We could combine this with the `render` function that renders any of the providers that our application needs like a ThemeProvider, or a React Router Provider. This `render` method could `render` all of the providers that our application needs and then `render` the `ui` inside of that.

[02:19] Then, we can provide `options` for any of these providers that we need. We could have one `render` method that we use throughout our test space and not have to concern ourselves with updating the providers that we're rendering with components as we refactor them to connect components to Redux, or add a React Router `Link`, or start using a theme from our ThemeProvider.

[02:36] In review, the reason that we're doing this, isn't just a save a couple lines of code in these two tests.
It's because much of our test base is probably going to need to `render` within a Redux `Provider`.

[02:47] We simplify things quite a bit by creating a `render` method that supports rendering connected Redux components and giving `options` for how to customize that Redux `Provider`.

Test a render prop component using a Jest mock function

Kent C Dodds: [00:00] Here, we have a super simple `Toggle` component, but it's special because it is using the render props API, where instead of rendering its `children` or rendering some specific UI, it calls its `children` as a function, expecting and providing the `on` state, and a mechanism for updating the state, this `toggle` function.

toggle.js

```
import React from 'react'

class Toggle extends React.Component {
  state = {on: false}
  toggle = () => this.setState(({on}) => ({on:
!on}))
  render() {
    return this.props.children({on:
this.state.on, toggle: this.toggle})
  }
}

export {Toggle}
```

[00:18] Let's see how we could test this in a way that's simple and comprehensive. I'm going to add a test that it `'renders with on state and toggle function'`. Then we're going to need to `import React from 'react'`, because we'll be rendering the `Toggle` component.

[00:33] We'll `import {render} from 'react-testing-library'`, and we'll `import {Toggle} from '../toggle'`. Let's go ahead, and we'll render that `Toggle`. We need to provide a function as `children` here. I'm going to make a variable called `children`, and that's going to take an `arg`. It's not going to return anything.

[00:52] Then we'll pass `children` here inside `Toggle`, but we need to have access to what `children` is called with. What I'm going to do is I'm going to make a `childrenArg`, and that's going to be an object. Then inside of here, I'm going to say `Object.assign(childrenArg, arg)` and that's the `arg` that it's being passed.

render-props.js

```
import React from 'react'
import {render} from 'react-testing-library'
import {Toggle} from '../toggle

test('renders with on state and toggle
function', () => {
  const childrenArg = {}
  const children = arg =>
Object.assign(childrenArg, arg)
  render(<Toggle>{children}</Toggle>)
})
```

[01:07] Then here, we can `expect(childrenArg).toEqual({on: false, toggle: expect.any(Function)})`. Then we can go ahead and call `childrenArg.toggle()`, and we'll make this assertion again, except `on` should now be `true`.

[01:28] We've verified the logic of this `Toggle` component. Now, if I open up my tests, I have an error here, because I'm returning an object from my `children` function, and `toggle` is returning what my `children` function returns.

[01:40] I need to make sure that `children` returns `null`. There we go. Now, my test is passing.

```
test('renders with on state and toggle
function', () => {
  const childrenArg = {}
  const children = arg => {
    Object.assign(childrenArg, arg)
    return null
  }
  render(<Toggle>{children}</Toggle>)
  expect(childrenArg).toEqual({on: false,
toggle: expect.any(Function)})
  childrenArg.toggle()
  expect(childrenArg).toEqual({on: true, toggle:
expect.any(Function)})
})
```

This component is pretty simple, and a single test is all we really need for this component. If we had a more complex component that required multiple tests, then I would probably make a `setup` function here that does all of this setup for me, and simply returns an argument with `childrenArg`.

```
function setup() {
  const childrenArg = {}
  const children = arg => {
    Object.assign(childrenArg, arg)
    return null
  }
  render(<Toggle>{children}</Toggle>)
  return {
    childrenArg
  }
}
```

[02:03] Then I can get `childrenArg` from calling `setup`. In review, the way that this works is I create a reference to an object that I'm calling `childrenArg`. Then whenever `Toggle` renders, it's going to call this `children` function.

```
function setup() {
  const childrenArg = {}
  const children = arg => {
    Object.assign(childrenArg, arg)
    return null
  }
  render(<Toggle>{children}</Toggle>)
  return {
    childrenArg
  }
}

test('renders with on state and toggle
function', () => {
  const {childrenArg} = setup()
  expect(childrenArg).toEqual({on: false,
toggle: expect.any(Function)})
  childrenArg.toggle()
  expect(childrenArg).toEqual({on: true, toggle:
expect.any(Function)})
})
```

[02:16] Then I'll assign all the properties from the argument that `Toggle` is passing to my `children` function onto that `childrenArg` object. Because I have a reference to that, I can check what the properties of that `childrenArg` are, and verify that those properties are correct.

[02:31] This is the API that my render prop component exposes, so this is what I'm testing. Then I can even make calls to those functions, which should result in a rerender, and then make additional assertions based off of what should have happened when I called that function.

[02:45] It might also be wise to add one or two integration tests, where I use the `Toggle` component in a more typical way, and verify that I can interact with that component in a way that supports one of my use cases for this component in the first place.

Test React portals with react-testing-library

null

Test Unmounting a React Component with react-testing-library

Kent C Dodds: [00:00] Here we have a `Countdown` component that has `state` for the `remainingTime`. As soon as it mounts, it starts counting down from that time. It calculates when the `end` time should be. Then it sets an `interval` to count down the `remainingTime`.

countdown.js

```
class Countdown extends React.Component {
  state = {remainingTime: 10000}
  componentDidMount() {
    const end = Date.now() +
this.state.remainingTime
    this.interval = setInterval(() => {
      const remainingTime = end - Date.now()
      if (remainingTime <= 0) {
        clearInterval(this.interval)
        this.setState({remainingTime: 0})
      } else {
        this.setState({
          remainingTime,
        })
      }
    })
  }
  componentWillUnmount() {
    clearInterval(this.interval)
  }
  render() {
    return this.state.remainingTime
  }
}
```

[00:12] If that `remainingTime <= 0` then it clears the `interval` and sets the `remainingTime` to `0` with `setState({remainingTime: 0})`, otherwise it'll set the `state` to the `remainingTime`. It will count down very quickly from 10 seconds.

[00:24] If the component is unmounted, then it will clear the `interval`. It's doing this so that it can avoid a memory leak. It's very important to clean up all of the work that you have pending

when the component unmounts.

[00:35] Let's go ahead and test this behavior using `unmounting.js`. I'm going to add a test that says, `'does not attempt to set state when unmounted (to prevent memory leaks)'`. With that, I'm going to want to render the `Countdown`.

[00:49] We'll `import React from 'react'`, `import {render} from 'react-testing-library'`, and `import {Countdown} from '../countdown'`. Then we'll `render(<Countdown />)`. What we're going to get back is `unmount`.

unmounting.js

```
import React from 'react'
import {render} from 'react-testing-library'
import {Countdown} from '../countdown'

test('does not attempt to set state when
unmounted (to prevent memory leaks)', () => {
  const {unmount} = render(<Countdown />)
  unmount()
})
```

[01:05] As soon as we're mounted, we'll call `unmount`. Then we want to make an assertion. What will happen if a `setState` call is called on an unmounted component, if this `setInterval` is not cleared, is that React will call `console.error` to indicate that there is a potential memory leak.

[01:22] Let's go ahead and spy on `console.error` with `beforeEach`, `jest.spyOn(console, 'error')`. We'll `mockImplementation` to do nothing so that our console stays

clean during our test. Then we'll add an `afterEach` to `console.error.mockRestore()`.

```
beforeEach(() => {
  jest.spyOn(console,
'error').mockImplementation(() => {})
})

afterEach(() => {
  console.error.mockRestore()
})
```

[01:43] Then we can add our assertion to `expect(console.error).not.toHaveBeenCalled()`. If we open up our test, we're all good, right?

```
test('does not attempt to set state when
unmounted (to prevent memory leaks)', () => {
  const {unmount} = render(<Countdown />)
  unmount()
  expect(console.error).not.toHaveBeenCalled()
})
```

[01:52] Let's go ahead and make sure that this is doing what we think it is by removing this `clearInterval` in our unmount. We'll comment this out. We'll save and our test is still passing...

```
componentWillUnmount() {
  // clearInterval(this.interval)
}
```

[02:00] What's happening is this test finishes. The program exits so quickly that our `setState` call never happens. We need to make sure that our test doesn't exit before our first `setInterval` happens.

[02:13] To do that, we're going to use Jest's built-in mechanisms for faking out timers like `setTimeout` and `setInterval`. We're going to say `jest.useFakeTimers`. Right after our `unmount`, we're going to say `jest.runOnlyPendingTimers()`.

```
jest.useFakeTimers()

beforeEach(() => {
  jest.spyOn(console,
'error').mockImplementation(() => {})
})

afterEach(() => {
  console.error.mockRestore()
})

test('does not attempt to set state when
unmounted (to prevent memory leaks)', () => {
  const {unmount} = render(<Countdown />)
  unmount()
  jest.runOnlyPendingTimers()
  expect(console.error).not.toHaveBeenCalled()
})
```

[02:29] There we get our error. We're asserting that we're not going to call `console.error`, but because our `interval` is now running and `setState` is being called, we're getting a warning that we can't call `setState` or `forceUpdate` on an unmounted component. It indicates a memory leak in your application.

[02:45] Let's go back. We'll restore the `clearInterval`. Our component will unmount. Now our test is passing because we're properly cleaning up after ourselves.

countdown.js

```
componentWillUnmount() {
  clearInterval(this.interval)
}
```

[02:54] In review, to make this work, we needed to spy on the `console.error` so we could have an assertion. Then we used the `unmount` function from our `render` function from `react-testing-library`.

[03:02] We unmounted the component and used Jest to fake out timers for `setInterval` so we could control when those timers run. We ran only the pending timers and then asserted that our `console.error` was not ran.