Kent C. Dodds

Back to overview Login

Testing Implementation Details

August 17th, 2020 — 12 min read

繁體中文 简体中文 Português 日本語 Русский

한국어 Español

Add translation

Back when I was using enzyme (like everyone else at the time), I stepped carefully around certain APIs in enzyme. I completely avoided shallow rendering, *never* used APIs like `` `instance()` ``, `` `state()` ``, or `` `find('ComponentName')` ``. And in code reviews of other people's pull requests I explained again and again why it's important to avoid these APIs. The reason is they each allow your test to test implementation details of your components. People often ask me what I mean by "implementation details." I mean, it's hard enough to test as it is! Why do we have to make all these rules to make it harder?

Why is testing implementation details bad?

There are two distinct reasons that it's important to avoid testing implementation details. Tests which test implementation details:

1. Can break when you refactor application code. False negatives
2. May not fail when you break application code. False positives

> To be clear, the test is: "does the software work". If the test passes, then that means the test came back "positive" (found working software). If it does not, that means the test comes back "negative" (did not find working software). The term "False" refers to when the test came back with an incorrect result, meaning the software is actually broken but the test passes (false positive) or the software is actually working but the test fails (false negative).

Let's take a look at each of these in turn, using the following simple accordion component as an example:

```tsx
// accordion.js
import * as React from 'react'
import AccordionContents from './accordion-contents'

class Accordion extends React.Component {
  state = {openIndex: 0}
  setOpenIndex = openIndex => this.setState({openIndex})
  render() {
    const {openIndex} = this.state
    return (
      <div>
        {this.props.items.map((item, index) => (
          <>
            <button onClick={() => this.setOpenIndex(index)}>
              {item.title}
            </button>
            {index === openIndex ? (
              <AccordionContents>{item.contents}</AccordionContents>
            ) : null}
          </>
        ))}
      </div>
    )
  }
}

export default Accordion
```

tsx

If you're wondering why I'm using a dated class component and not modern function component (with hooks) for these examples, keep reading, it's an interesting reveal (which some of those of you experienced with enzyme you might already be expecting).

And here's a test that tests implementation details:

```tsx
// __tests__/accordion.enzyme.js
import * as React from 'react'
// if you're wondering why not shallow,
// then please read https://kcd.im/shallow
import Enzyme, {mount} from 'enzyme'
import EnzymeAdapter from 'enzyme-adapter-react-16'
import Accordion from '../accordion'

// Setup enzyme's react adapter
Enzyme.configure({adapter: new EnzymeAdapter()})

test('setOpenIndex sets the open index state properly', () => {
  const wrapper = mount(<Accordion items={[]} />)
  expect(wrapper.state('openIndex')).toBe(0)
  wrapper.instance().setOpenIndex(1)
  expect(wrapper.state('openIndex')).toBe(1)
})

test('Accordion renders AccordionContents with the item contents', () => {
  const hats = {title: 'Favorite Hats', contents: 'Fedoras are classy'}
  const footware = {
    title: 'Favorite Footware',
    contents: 'Flipflops are the best',
  }
  const wrapper = mount(<Accordion items={[hats, footware]} />)
  expect(wrapper.find('AccordionContents').props().children).toBe(hats.contents)
})
```

tsx

Raise your hand if you've seen (or written) tests like this in your codebase (🙌).

Ok, now let's take a look at how things break down with these tests...

False negatives when refactoring

A surprising number of people find testing distasteful, especially UI testing. Why is this? There are various reasons for it, but one big reason I hear again and again is that people spend way too much time babysitting the tests. "Every time I make a change to the code, the tests break!" This is a real drag on productivity! Let's see how our tests fall prey to this frustrating problem.

Let's say I come in and I'm refactoring this accordion to prepare it to allow for multiple accordion items to be open at once. A refactor doesn't change existing behavior at all, it just changes the implementation. So let's change the implementation in a way that doesn't change the behavior.

Let's say that we're working on adding the ability for multiple accordion elements to be opened at once, so we're changing our internal state from **`openIndex`** to **`openIndexes`**:

```tsx
class Accordion extends React.Component {
-   state = {openIndex: 0}
-   setOpenIndex = openIndex => this.setState({openIndex})
+   state = {openIndexes: [0]}
+   setOpenIndex = openIndex => this.setState({openIndexes: [openIndex]})
    render() {
-     const {openIndex} = this.state
+     const {openIndexes} = this.state
      return (
        <div>
          {this.props.items.map((item, index) => (
            <>
              <button onClick={() => this.setOpenIndex(index)}>
                {item.title}
              </button>
-             {index === openIndex ? (
+             {openIndexes.includes(index) ? (
                <AccordionContents>{item.contents}</AccordionContents>
              ) : null}
            </>
          ))}
        </div>
    )
  }
}
```

Awesome, we do a quick check in the app and everything's still working properly, so when we come to this component later to support opening multiple accordions, it'll be a cinch! Then we run the tests and 💥 kaboom 💥 they're busted. Which one broke? **`setOpenIndex sets the open index state properly`**.

What's the error message?

```
expect(received).toBe(expected)

Expected value to be (using ===):
  0
Received:
  undefined
```

Is that test failure warning us of a real problem? Nope! The component still works fine.

This is what's called a false negative. It means that we got a test failure, but it was because of a broken test, not broken app code. I honestly cannot think of a more annoying test failure situation. Oh well, let's go ahead and fix our test:

```tsx
1   test('setOpenIndex sets the open index state properly', () => {
2     const wrapper = mount(<Accordion items={[]} />)
-     expect(wrapper.state('openIndex')).toEqual(0)
+     expect(wrapper.state('openIndexes')).toEqual([0])
4     wrapper.instance().setOpenIndex(1)
-     expect(wrapper.state('openIndex')).toEqual(1)
+     expect(wrapper.state('openIndexes')).toEqual([1])
6   })
```

The takeaway: Tests which test implementation details can give you a false negative when you refactor your code. This leads to brittle and frustrating tests that seem to break anytime you so much as look at the code.

False positives

Ok, so now let's say your co-worker is working in the Accordion and they see this code:

```tsx
<button onClick={() => this.setOpenIndex(index)}>{item.title}</button>
```

Immediately their premature performance optimization feelings kick in and they say to themselves, "hey! inline arrow functions in `render` are bad for performance, so I'll just clean that up! I think this should work, I'll just change it really quick and run tests."

```tsx
<button onClick={this.setOpenIndex}>{item.title}</button>
```

Cool. Run the tests and... ✅ ✅ awesome! They commit the code without checking it in the browser because tests give confidence right? That commit goes in a completely unrelated PR that changes thousands of lines of code and is understandably missed. The accordion breaks in production and Nancy is unable to get her tickets to see Wicked in Salt Lake next February. Nancy is crying and your team feels horrible.

So what went wrong? Didn't we have a test to verify that the state changes when `setOpenIndex` is called *and* that the accordion contents are displayed appropriately!? Yes you did! But the problem is that there was no test to verify that the button was wired up to `setOpenIndex` correctly.

This is called a false positive. It means that we didn't get a test failure, but we should have! So how do we cover ourselves to make sure this doesn't happen again? We need to add another test to verify clicking the button updates the state correctly. And then I need to add a coverage threshold of 100% code coverage so we don't make this mistake again. Oh, and I should write a dozen or so ESLint plugins to make sure people don't use these APIs that encourage testing implementation details!

... But I'm not going to bother... Ugh, I'm just so tired of all these false positives and negatives, I'd almost rather not write tests at all. DELETE ALL THE TESTS! Wouldn't it be nice if we had a tool that had a wider pit of success? Yes it would! And guess what, we DO have such a tool!

Implementation detail free testing

So we could rewrite all these tests with enzyme, limiting ourselves to APIs that are free of implementation details, but instead, I'm just going to use React Testing Library which will make it very difficult to include implementation details in my tests. Let's check that out now!

```tsx
// __tests__/accordion.rtl.js
import '@testing-library/jest-dom/extend-expect'
import * as React from 'react'
import {render, screen} from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import Accordion from '../accordion'

test('can open accordion items to see the contents', () => {
  const hats = {title: 'Favorite Hats', contents: 'Fedoras are classy'}
  const footware = {
    title: 'Favorite Footware',
    contents: 'Flipflops are the best',
  }
  render(<Accordion items={[hats, footware]} />)

  expect(screen.getByText(hats.contents)).toBeInTheDocument()
  expect(screen.queryByText(footware.contents)).not.toBeInTheDocument()

  userEvent.click(screen.getByText(footware.title))

  expect(screen.getByText(footware.contents)).toBeInTheDocument()
  expect(screen.queryByText(hats.contents)).not.toBeInTheDocument()
})
```

Sweet! A single test that verifies all the behavior really well. And this test passes whether my state is called `openIndex`, `openIndexes`, or `tacosAreTasty` 🌮. Nice! Got rid of that false negative! And if I wire up my click handler incorrectly, this test will fail. Sweet, got rid of that false positive too! And I didn't have to memorize any list of rules. I just use the tool in the idiomatic usage, and I get a test that actually can give me confidence my accordion is working as the user wants it too.

So... What are implementation details then?

Here's the simplest definition I can come up with:

> **Implementation details are things which users of your code will not typically use, see, or even know about.**

So the first question we need an answer to is: "Who is the user of this code." Well, the end user who will be interacting with our component in the browser is definitely a user. They'll be observing and interacting with the rendered buttons and contents. But we also have the developer who will be rendering the accordion with props (in our case, a given list of items). So React components typically have two users: end-users, and developers. End-users and developers are the two "users" that our application code needs to consider.

Great, so what parts of our code do each of these users use, see, and know about? The end user will see/interact with what we render in the `render` method. The developer will see/interact with the props they pass to the component. So our test should typically only see/interact with the props that are passed, and the rendered output.

This is precisely what the React Testing Library test does. We give it our own React element of the Accordion component with our fake props, then we interact with the rendered output by querying the output for the contents that will be displayed to the user (or ensuring that it wont be displayed) and clicking the buttons that are rendered.

Now consider the enzyme test. With enzyme, we access the `state` of `openIndex`. This is not something that either of our users care about directly. They don't know that's what it's called, they don't know whether the open index is stored as a single primitive value, or stored as an array, and frankly they don't care. They also don't know or care about the `setOpenIndex` method specifically. And yet, our test knows about both of these implementation details.

This is what makes our enzyme test prone to false negatives. Because by making our test use the component differently than end-users and developers do, we create a third user our application code needs to consider: the tests! And frankly, the tests are one user that nobody cares about. I don't want my application code to consider the tests. What a complete waste of time. I don't want tests that are written for their own sake. *Automated tests should verify that the application code works for the production users.*

> *"The more your tests resemble the way your software is used, the more confidence they can give you. —me"*

> Read more about this in Avoid the Test User.

So, what about hooks

Well, as it turns out, enzyme still has a lot of trouble with hooks. Turns out when you're testing implementation details, a change in the implementation has a big impact on your tests. This is a big bummer because if you're migrating class components to function components with hooks, then your tests can't help you know that you didn't break anything in the process.

React Testing Library on the other hand? It works either way. Check the codesandbox link at the end to see it in action. I like to call tests you write with React Testing Library:

> **Implementation detail free and refactor friendly.**

Conclusion

So how do you avoid testing implementation details? Using the right tools is a good start. Here's a process for how to know what to test. Following this process helps you have the right mindset when testing and you will naturally avoid implementation details:

1. What part of your untested codebase would be really bad if it broke? (The checkout process)
2. Try to narrow it down to a unit or a few units of code (When clicking the "checkout" button a request with the cart items is sent to /checkout)
3. Look at that code and consider who the "users" are (The developer rendering the checkout form, the end user clicking on the button)
4. Write down a list of instructions for that user to manually test that code to make sure it's not broken. (render the form with some fake data in the cart, click the checkout button, ensure the mocked /checkout API was called with the right data, respond with a fake successful response, make sure the success message is displayed).
5. Turn that list of instructions into an automated test.

I hope that's helpful to you! If you really want to take your testing to the next level, then I definitely recommend you get a Pro license for TestingJavaScript.com 🏆

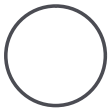
Good luck!

P.S. If you'd like to play around with all this, here's a codesandbox.

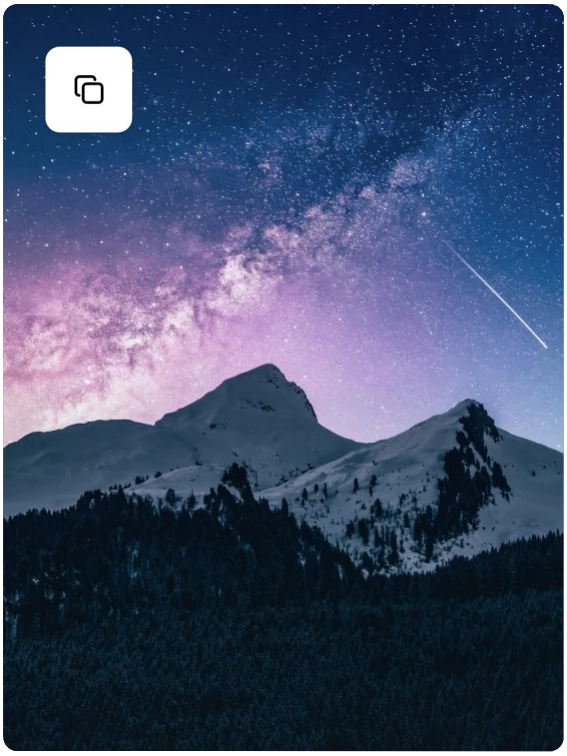
P.S.P.S. As an exercise for you... What happens to that second enzyme test if I change the name of the `AccordionContents` component?

Written by Kent C. Dodds

Kent C. Dodds is a JavaScript software engineer and teacher. Kent's taught hundreds of thousands of people how to make the world a better place with quality software development tools and practices. He lives with his wife and four kids in Utah.

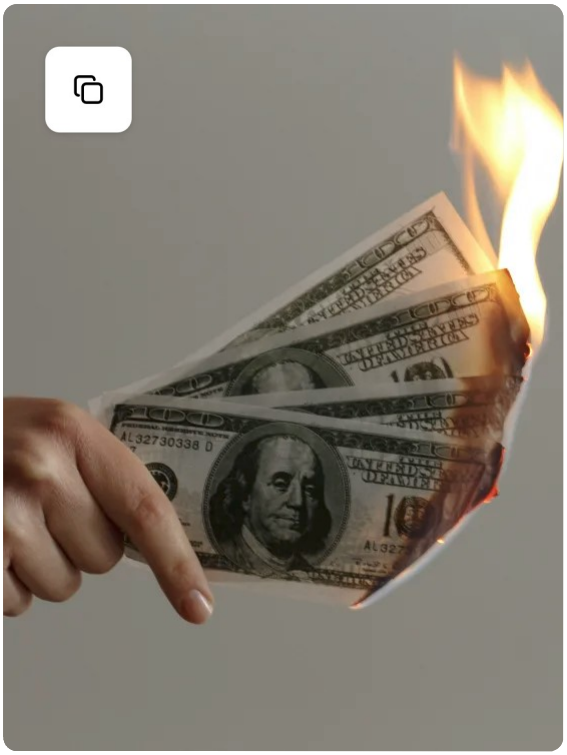
Learn more about Kent 

If you found this article helpful.
You will love these ones as well.



July 15th, 2020 — 14 min read

JavaScript to Know for React



June 4th, 2019 — 14 min read

When to useMemo and useCallback

Kent C. Dodds

Full time educator making our world better

Contact

Email Kent

Call Kent

Office hours

General

My Mission

Privacy policy

Terms of use

Code of conduct

Stay up to date

Subscribe to the newsletter to stay up to date with articles, courses and much more!
Learn more about the newsletter

First name

Email

Sign me up

Sitemap

Home

Blog

Courses

Discord

Chats Podcast

Workshops

Talks

Testimony

Testimonials

About

Credits

Sitemap.xml