<https://kentcdodds.com/blog/write-tests>

Write tests. Not too many. Mostly integration. - Guillermo Rauch

Why write tests? Gives us enhanced workflows, can make you more productive

Can give you a lot of confidence

Test driven development with cypress - look into this

What does “Not too many” mean?

How much code coverage do you need?

Answer: It depends - not a one size fits all

Graphical user interface, application

Description automatically generated

It is really easy to get good code coverage easily, but after a while the return on your time begins to deteriorate in terms of coverage

There’s some level where it makes sense to focus on coverage versus focusing on shipping features

Start ups tend to be more on the left side of the curve whereas companies where nothing ever breaks will be on the right side

If you are writing a library, 100% code coverage is pretty simple to achieve, especially if library is small

Applications range in coverage - there’s no number

One reason Dodds is nervous about 100% code coverage in applications is that inevitably what will happen is that you will start testing implementation details, writing tests to cover one off hacks that were required to achieve something

You start exposing private functions, doing a bunch of stuff that you shouldn’t do just to get coverage higher

You can tell if you are testing implementation details

* “if your test does something that the consumer of your code doesn’t” - like using a private function, exposing it just for the test - that’s an implementation detail of the module
* “If a refactor breaks your tests” - this is going to make people not like testing

What does “mostly integration” mean?

Static code analysis is a form of testing

eslint like tools are valuable without even running code

TypeScript and other strongly typed eliminate need of writing tests in the case that a parameter that is not of the correct type is passed into a function

Unit tests - simplest

Integration tests take a little more set up and touch more things

Integration tests test business logic, they test things that actually will break

Diagram

Description automatically generated

Diagram

Description automatically generated

Integration tests provide the best balance of cost, confidence and they help solve bigger problems but all types of tests are important

Kent C Dodds “I really don’t like shallow rendering” - “Poke fewer holes in reality” - this means test things more closely to how they are used. This is done most easily by testing higher up in your tree

There’s more benefit to tests that don’t test things in isolation - for instance, no one cares about a snapshot test that just tests a button in isolation

Often I find myself saving time when I put time in to write tests.

The thing you should be thinking about when writing tests is how much confidence they bring you that your project is free of bugs. Static typing and linting tools like [TypeScript](https://www.typescriptlang.org/) and [ESLint](https://eslint.org/) can get you a remarkable amount of confidence,

That said, even a strongly typed language should have tests

The problem is that you get diminishing returns on your tests as the coverage increases much beyond 70% (I made that number up... no science there). Why is that? Well, when you strive for 100% all the time, you find yourself spending time testing things that really don't need to be tested.

*Maintaining tests like this actually really slow you and your team down.*

You *really* want to avoid testing implementation details because it doesn't give you very much confidence that your application is working and it slows you down when refactoring. You should very rarely have to change tests when you refactor code.

Makes a point that projects that are reusable in many different situations makes it easier to achieve higher coverage

So while E2E tests may be slower and more expensive than unit tests, they bring you much more confidence that your application is working as intended.

It doesn't matter if your component **<A />** renders component **<B />** with props **c** and **d** if component **<B />** actually breaks if prop **e** is not supplied.

*it doesn't do you any good if you don't also verify that they work together properly*

I think the biggest thing you can do to write more integration tests is to stop mocking so much stuff. *When you mock something you're removing all confidence in the integration between what you're testing and what's being mocked.*

[*https://kentcdodds.com/blog/why-i-never-use-shallow-rendering*](https://kentcdodds.com/blog/why-i-never-use-shallow-rendering)

**With shallow rendering, I can refactor my component's implementation and my tests break. With shallow rendering, I can break my application and my tests say everything's still working.**

The reason I write tests is to be confident that my application works and there are far better ways to do that than shallow rendering.

So all shallow rendering is doing is taking the result of the given component's **render** method (which will be a React element (read [What is JSX?](https://kentcdodds.com/blog/what-is-jsx))) and giving us a **wrapper** object with some utilities for traversing this JavaScript object. This means it doesn't run lifecycle methods (because we just have the React elements to deal with), it doesn't allow you to actually interact with DOM elements (because nothing's actually rendered), and it doesn't actually attempt to get the react elements that are returned by your custom components (like our **Fade** component).

Text

Description automatically generated

In summary, if your test uses **instance()** or **state()**, know that you're testing things that the user couldn't possibly know about or even care about, which will take your tests further from giving you confidence that things will work when your user uses them.

This is a very common misconception: "To unit test a react component you must use shallow rendering so other components are not rendered." It's true that shallow rendering doesn't render other components (as demonstrated above), what's wrong with this is that it's way too heavy handed because it doesn't render *any* other components. You don't get a choice.

Not only does shallow rendering not render third party components, it doesn't even render in-file components. For example, the **<Fade />** component we have above is an implementation detail of the **<HiddenMessage />** component, but because we're shallow rendering **<Fade />** isn't rendered so changes to that component could break our application but not our test. That's a major issue in my mind and is evidence to me that we're testing implementation details.

I *definitely* want to know if the third party component I'm using breaks my use case. I mean, I'm not going to rewrite their entire test base, but if I can easily test my use case by *not* mocking out their component then why not do that and get the extra confidence?

[I'm in favor of relying more heavily on integration testing](http://kcd.im/write-tests). When you do this, you need to unit test fewer of your simple components and wind up only having to unit test edge cases for components (which can mock all they want). But even in these situations, I still think it leads to more confidence and a more maintainable testbase when you're explicit about which components are being mocked and which are being rendered by doing full mounting and explicit mocks.

[*The more your tests resemble the way your software is used, the more confidence they can give you.*](https://twitter.com/kentcdodds/status/977018512689455106)*— Kent C. Dodds* 👋

I very rarely use snapshot testing with react and I certainly wouldn't use it with shallow. That's a recipe for implementation details. The whole snapshot is nothing but implementation details (it's full of component and prop names that change all the time on refactors). It'll fail any time you touch the component and the git diff for the snapshot will look almost identical to the one for your changes to the component.

This will make people careless about changes to the snapshot updates because they change all the time. So it's basically worthless (almost worse than no tests because it makes you think you're covered when you're not and you won't write proper tests because they're in place).

<https://kentcdodds.com/blog/effective-snapshot-testing>

*They are tests you don't understand, so when they fail, you don't usually understand why or how to fix it. That means you have to do true/false negative analysis & then suffer indirection as you debug how to resolve the issue*

* *They are generated files, and developers tend to be undisciplined about scrutinizing generated files before committing them, if not at first then definitely over time. Most developers, upon seeing a snapshot test fail, will sooner just nuke the snapshot and record a fresh passing one instead of agonizing over what broke it.*

*Good tests encode the developer's intention, they don't only lock in the test's behavior without editorialization of what's important and why. Snapshot tests lack (or at least, fail to encourage) expressing the author's intent as to what the code does (much less why)*

*Because they're more integrated and try to serialize an incomplete system (e.g. one with some kind of side effects: from browser/library/runtime versions to environment to database/API changes), they will tend to have high false-negatives (failing test for which the production code is actually fine and the test just needs to be changed). False negatives quickly erode the team's trust in a test to actually find bugs and instead come to be seen as a chore on a checklist they need to satisfy before they can move on to the next thing.*

*These four things lead to a near total loss in the intended utility of integrated/functional tests: as the code changes make sure nothing is broken. - tied extremely close to implementation details*

*Me! - The reason we write tests is to gain confidence that our application works as expected. Take for instance a calculator application. Refactoring of the code in a calculator application doesn’t change the fact that 1 + 1 should equal 2. If previously working tests break because we have refactored our code (without error), it gets in the way of what really matters => 1 + 1 equals 2. Don’t test implementation details or it has a high chance of getting in the way of what you are writing tests for to begin with.*

*Even snapshot tests with good intention behind them will be overlooked in favor fixing them because of how many badly designed/no intention snapshot tests we have*

*One thing I want to make clear before continuing is that snapshot testing is an assertion, just like the****toBe****in:****expect('foo').toBe('foo')****. I think there's sometimes confusion on this point, so I just wanted to clear that up.*

Text

Description automatically generated

[**jest-glamor-react**](https://kentcdodds.com/blog/effective-snapshot-testing#jest-glamor-react)

If you're using CSS-in-JS, there's a great way to use snapshot testing to reduce some of the difficulty of testing these kinds of changes. If you use a tool like **jest-glamor-react**then you can include the applicable CSS with whatever you rendered. [For example](https://github.com/kentcdodds/jest-glamor-react/blob/84ace7c093d49344268b25a0d5a2518cd20922da/src/__snapshots__/serializer.test.js.snap#L101-L124):

Text

Description automatically generated

This is nice because now if we change logic so some styles aren't applied properly we'll know about it

Avoid huge snapshots -  it's going to suffer major maintenance issues and slow you and your team down

*Remember that tests are all about giving you confidence that you wont ship things that are broken* and you're not going to be able to ensure that very well if you have huge snapshots that nobody will review carefully. I've personally experienced this with a snapshot that's over 640 lines long. Nobody reviews it, the only care anyone puts into it is to nuke it and retake it whenever there's a change (like Justin mentioned).

*I should add that even huge snapshots aren't entirely useless. Because if the snapshot changes unexpectedly it can (and has) inform us that we've made a change with further reaching impacts than anticipated.*

[*https://kentcdodds.com/blog/when-i-follow-tdd*](https://kentcdodds.com/blog/when-i-follow-tdd)

* 🚨 Red: Write a test for the function/module you're going to create before it exists/supports the feature you're adding. This gives you a test that fails (you get a "red" error message).
* ✅ Green: Implement just enough code to get that test passing (you get a "green" success message).
* 🌀 Refactor: Look over the code you have written and refactor it to ensure it's well-written, as easy as possible to read/understand, and well-designed. (The cool thing with this step is that you now have a test in place that will tell you if you break something as you refactor).
* 🔁 Repeat: It's a cycle, after all 😉 Keep going until you've finished implementing everything you need to.

When I've got a bug to fix, I *love* reproducing that bug with a test before fixing it. Doing this gives me a huge amount of confidence that I understand the cause of the bug in the first place and when I get the test to green, I know that I've actually fixed the bug and not just tested around the problem.

if I've got a utility function of sufficient complexity to need isolated unit tests, then that's another great situation that's well suited for TDD. With these kinds of functions, you often have a pretty well-defined set of inputs and outputs based on the requirements you have for the code.

**It's pointless to TDD when you test**[**implementation details**](https://kentcdodds.com/blog/testing-implementation-details)**.**

Part of the point of using TDD is to help you think about the thing you're building from the outside, without thought for the implementation, so when you get to implementing things you don't get lost in the details of the code and can keep the high-level goal in mind.

However, since Testing Library allows you to focus on [the user's](https://kentcdodds.com/blog/avoid-the-test-user) experience, rather than the implementation, you can follow TDD when building UIs that have a well-defined design and user experience.

Building a well-defined UI? Try TDD.

<https://www.youtube.com/watch?v=kCR3JAR7CHE>

React TDD

An interesting approach of iterative TDD - micro red/green/blue cycle where write a little bit in a test to make it fail and then write component code to have the test pass. Key is to have the test name fully explain what you are coding towards

<https://kentcdodds.com/blog/but-really-what-is-a-javascript-test>

A test is code that throws an error when the actual result of something does not match the expected output

The part that says **actual !== expected** is called an "assertion." It's a way to say in code that one thing should be a certain value or pass a certain... eh... test :) It could be an assertion that the **actual** matches a regex, is an array with a certain length, or any number of things. The key is that if our assertion fails, then we throw an error.

One of the most important parts of testing frameworks (or assertion libraries) is how helpful their error messages are

Often when a test fails, the first thing you'll see is the error message. If you can't figure out what the underlying problem is from the error message, then you have to spend a few minutes looking at the code to understand what went wrong

<https://kentcdodds.com/blog/but-really-what-is-a-javascript-mock>

Now so long as our mock can model what the real world version does, we can get back a little confidence that our code is working despite having to mock out what **getWinner** is actually doing.

A screenshot of a computer

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

<https://www.youtube.com/watch?v=Da9wfQ0frGA&list=PLV5CVI1eNcJgNqzNwcs4UKrlJdhfDjshf>

Why unit test at all? It is cheaper and faster to get feedback **ability to fail fast**

<https://kentcdodds.com/blog/confidently-shipping-code>

I realized quickly that spending time to manually verify that everything's working, every single time I fixed a bug or added a new feature, was pretty annoying

Investing time into testing my library ended up saving me a TON of time, and I was able to integrate testing into my workflow.

That was when it really occurred to me that testing had become more than a default workflow for saving time. It was a mechanism for giving me confidence.

So why do I write tests? I write tests because they allow me to accomplish more than I could otherwise. I now have thousands of Kents in the form of automated tests telling me whether changes are breaking use cases. With that venerable army of robots, I'm able to rest easy and get more accomplished.

<https://kentcdodds.com/blog/testing-implementation-details>

I [completely avoided shallow rendering](https://kentcdodds.com/blog/why-i-never-use-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.

Tests which test implementation details:

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

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.

false negative. It means that we got a test failure, but it was because of a broken test, not broken app code.

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.

Text

Description automatically generated

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

"Who is the user of this code."

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.

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.

**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.**

**Implementation detail free and refactor friendly.**

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:

* What part of your untested codebase would be really bad if it broke? (The checkout process)
* 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)
* Look at that code and consider who the "users" are (The developer rendering the checkout form, the end user clicking on the button)
* 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).
* Turn that list of instructions into an automated test.