The case against React snapshot testing

Jun 27 | Ben Jackson

...and what you can do instead!

We've been working with React for a few years now here at ezCater, during which time our unit testing story has been steadily evolving. One of the recent strategies we've tried out is Jest snapshot testing.

Snapshot testing is a popular strategy that we've seen in several React repos. It has a fair amount of positive supporters, as well as some cautious adopters. After spending a little over half a year using it (two repos, almost two dozen devs), we've removed or replaced nearly every existing snapshot test.

Why the change of heart? In short, we've found that **snapshots are more trouble than they're worth, especially across large and/or fluid teams of people**. In almost every case, a more focused and explicit unit test is a much better choice. More on that later.

A quick overview - what is snapshot testing?

A snapshot test is a simple way to evaluate changes in a component's output. When the test is first run, it renders a given component and stores the output in a file that gets committed alongside the code. Each time the test is re-run, the component is rendered and the new output is compared against the stored version. If the two versions differ, a 'diff' is displayed, and the user has the option to update the committed snapshot file with the new output.

What problem is this supposed to solve?

Many React components contain little to no conditional logic, and are used purely for presentational purposes. Just some markup and text, maybe some styles if you're doing the CSS-in-JS thing. An item in a list, a button with a provided click handler, a container that wraps a bunch of child components, etc.

Testing these types of components has always been a bit of a gray area. We want to make sure that nothing blows up, or that some piece of the tree conditionally renders based on the input. Maybe we've written some CSS and want to make sure it gets applied in a `style` attribute. Usually this type of smoke test ends up being some variation of 'pick a random piece of the output and see if it exists'. Unfortunately this approach is at best inconsistent, at worst a false-positive/negative.

With snapshots, the goal is to solve the above problem by just rendering the whole thing and seeing what changes. We can verify that it renders, and we can visually assert that things are where we want them to be. With the right plugins, we can see applied styles and normalize things like environment-based URLs and dynamically generated class names.

That sounds great! So what's the issue?

We've found that in practice, **snapshot tests end up making assertions that aren't clearly represented in the test output**. The assertion tries to focus on a very specific behavior or attribute, but because the test renders the entire component, it regularly leads to tests that fail for unrelated reasons. It's not immediately clear whether something is actually wrong, especially for developers who are unfamiliar with the component.

For example, you see this kind of diff all the time:

```
FAIL src/components/TableList/__tests__/TableList.test.js
  • <TableList /> › shows the item price if a price is present

    expect(value).toMatchSnapshot()

    Received value does not match stored snapshot 1.

    - Snapshot
    + Received

    @@ -142,28 +142,24 @@
           className="emotion-8 emotion-9"
         >
           Chicken Parm. Sandwhich
         </div>
         <div
    -      className="ui basic segment"
    +      className="emotion-10 emotion-9"
         >
    -      <div
    -        className="emotion-10 emotion-9"
    -      >
             <input
               style={
                 Object {
                   "width": "40px",
                 }
               }
    -          type="text"
    -          value={2}
    -        />
    -      </div>
    -      <div
    -        className="emotion-10 emotion-9"
    -      >
    -        $11.99
    -      </div>
    +          type="text"
    +          value={2}
    +        />
    +      $11.99
         </div>
    +    <div
    +      className="emotion-10 emotion-9"
    +    >
    +      $11.99
    +    </div>
       </div>
     </div>

      at Object.<anonymous> (src/components/TableList/__tests__/TableList.test.js:43:23)
          at new Promise (<anonymous>)
      at Promise.resolve.then.el (node_modules/p-map/index.js:46:16)
          at <anonymous>
      at process._tickDomainCallback (internal/process/next_tick.js:228:7)

  <TableList />
    ✕ shows the item price if a price is present (8ms)

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

About ezCater

We're the #1 online – and the only nationwide – marketplace for business catering in the United States. We make it easy to order food online for your office. From routine office lunches to offsite client meetings, from 5 to 2,000 people, we have a solution for you. ezCater connects business people with over 50,000 reliable local caterers and restaurants across the U.S.

ezCater is hiring!

We're always looking for highly skilled full stack and iOS engineers to help execute our technology goals while riding this rocket ship of growth. Our people are terrifically talented straight-shooters who love what they do. And everyone is generous and kind, too — we have a strict no jerk policy.

View ezCater Opportunities

Recent Posts

- Saying goodbye to emotion
- Multi-armed Bandit Experimentation
- Migrate, transform, and backfill data with zero downtime
- You're not in the zone!
- ezCater Code Culture

Posts by Topic

- experiments (6)
- data warehousing (4)
- engineering (4)
- ezcater (4)
- ruby (4)

see all

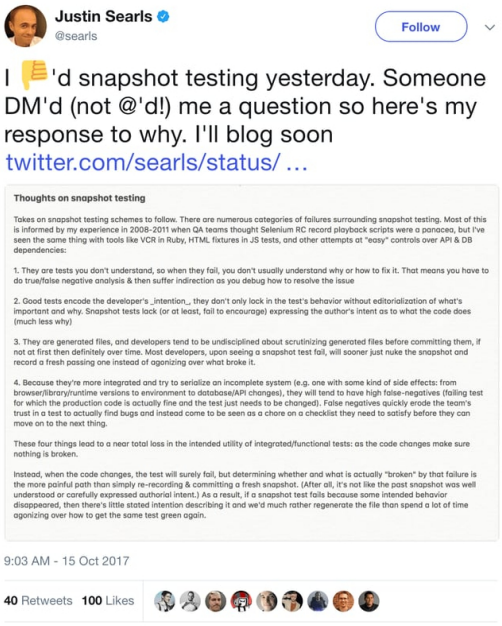
Subscribe to Email Updates

Email*

Subscribe

Did the test *really* fail? It looks like there's a price there, but is it the one that the test is talking about? Now the person running the test has to spend extra time poking around to see what's what. This leads to folks getting annoyed and not paying attention to the snapshot output, and just updating the snapshot to get tests to pass.

The following tweet does a great job outlining some more commonly seen issues with snapshot testing:



After seeing each of them in action, we decided to spend some time coming up with better solutions.

What can I do instead?

The typical fix for an unruly snapshot test is to break the component down into smaller components. While this may temporarily make the test output easier to understand, it doesn't hold up over time; components will continue to grow, and having to constantly split code into smaller bits for the sake of testing is a slow path to madness.

We've found that in most cases, a snapshot test can be replaced with a basic unit test. We also rely on linting and plugins to do a lot of the work for us. Here are some common snapshot test scenarios and some alternate ways of dealing with them.

The `'it renders'` test

Let's start with my favorite of the bunch. I'll admit, I'm guilty of having written some of these gems along the way. We've seen this test (or some similar version) in many repos - render the component, take a snapshot, and voila! More test coverage.

But what is this actually testing? That the code is able to compile? Spinning up a test runner or local server will tell you that. That the JSX is syntactically correct? You can catch that with linting. That React can call `render()` properly? I believe the React codebase has that covered.

These kind of smoke tests can usually be removed entirely in favor of well configured linting tools and a simple build check. At the very least, these tests can be re-written to be more explicit about what the test feedback is telling you:

```
1  it('can render basic markup without error', () => {
2    shallow(<FooComponent />).html();
3  });
```

it_renders.js hosted with ❤ by GitHub — view raw

But overall we've found that these tests have little added value, and just take more of everyone's time to maintain.

The `'it displays the right text'` test

Depending on what this is actually testing, this type of test may be the same as an `'it renders'`. Any snapshot test that checks for static text appearing in a component should probably be removed, as the expectations tend to be ambiguous (e.g. `'it renders the header text'`, `'it displays the right price'`), and having to add them for each component is redundant and unscalable. We use a translation library for all our text strings (`react-i18next`) that will throw an error if a translation is missing, which takes care of pretty much all 'missing text' issues.

If you need to check for some bit of conditionally rendered text, it's best to avoid the snapshot and write a test that asserts exactly what you want to see. This way when the test fails, the reader knows exactly what part of the code to look at. The test becomes a roadmap for the developer, pointing them in the right direction if it fails. Consider the following snapshot test:

```
1  it('displays the correct message if a user has few followers', () => {
2    const followersComponent = shallow(<TweetContainer numFollowers={100} />);
3
4    expect(followersComponent).toMatchSnapshot();
```

```
});
```

followers_message_snap.js hosted with ❤ by GitHub view raw

When this test fails, how is the developer supposed to know what the correct message is? Or where that message is located within the component? The whole thing is ambiguous, and will probably be a pain to debug. Now consider the test written as a unit test:

```
it('displays the correct message if a user has few followers', () => {
  const followersComponent = shallow(
    <TweetContainer numFollowers={100} />
  ).find(FollowersDisplay);

  expect(followersComponent.html()).toContain('Fewer than 500 followers');
});
```

followers_message_unit.js hosted with ❤ by GitHub view raw

The test shows the developer exactly what output is expected so it's clear what broke when the test fails. Also, by using a `find()`, it directs the developer to a specific area to investigate.

The `'it conditionally renders some markup'` test

These tests can be tackled in the same way as the 'conditionally rendered text' tests. We've found that asserting against the presence of a specific component/markup element is much clearer to understand and fix than figuring out which snapshot is displaying the correct output.

The `'it applies some CSS rules'` test

If you are using CSS-in-JS, there are libraries you can use to generate the style attributes that will be applied to a component as part of a snapshot. We use a package called `jest-emotion` to do this.

Checking CSS is probably the strongest case for keeping around snapshot tests; however, this is only because we haven't come up with a better solution yet. It is definitely useful to know if changes you've made have impacted the styles being applied to a component. But because the test snapshot will also contain markup, it will still fail for reasons unrelated to styling.

Also, style attributes written on a page aren't worth much on their own. Sometimes changes in style attributes could have no visible effect on the rendered markup, or could break in certain browsers and not others. In these cases, a snapshot test can lead to false negatives or positives, or just be extra noise to deal with. Ideally style changes would be verified by some sort of visual regression tool, or by a good ol' fashioned "actually look at it in a browser" test. But for now, we've kept a few of these snapshot tests around. We're currently working on a better testing story for this, so stay posted!

Changing our approach to snapshot testing has made our Javascript test suites faster, more concise, less flaky, and easier to understand. Hopefully by adopting some of these ideas, you'll be able to experience the same benefits!

Tags: unit testing, jest, snapshot testing, javascript, react