Testing React UI components

Wednesday, August 15, 2018 9:17 AM

Research

Just writing a bunch of random notes here that I encounter while trying to research this for the first time.

Resources:

- https://jestjs.io/docs/en/snapshot-testing - introduction to snapshot testing. Not much here is React-specific.
- https://reactjs.org/docs/test-utils.html - official React support for testing. This is mostly just a reference.
- https://github.com/kentcdodds/react-testing-library

Concepts from test-utils (reference)

- Shallow rendering (reference) - only render "one level deep" so that you don't have to worry about child components
- Simulating events is very easy due to this: https://reactjs.org/docs/test-utils.html#simulate
- react-test-renderer (reference) - this lets you render directly to pure JavaScript objects rather than to a DOM

Concepts from react-testing-library (reference)

- The focus is entirely on testing in the same way that a user would *use* your application, e.g. finding elements by their text and type, etc. Because of this, there will be a DOM when using this library. That means that DOM nodes will have typical functions like "querySelector"
    - An example of how this matters: react-testing-library does provide a "rerender" function (reference) which can specify different props for a component. This would be useful if you *weren't* following this philosophy of testing as the user would; you could simply force the props to be different and ensure that, say, shouldComponentUpdate returns true. Instead, the philosophy suggests that you should simulate a button click that would incur those changes to props and then verify that it worked. Despite that, you may still find uses for "rerender" (e.g. when the button that you want to click is in a completely different tree).
        - It's mostly: if you're unit testing, then test the interaction as the user would (so click buttons). If you're testing a unit and expect some outside state to be set, then feel free to directly change the props.
    - "cleanup" should be called after most tests to ensure that there are no memory leaks.
- jest-dom has some helper functions for checking the text content, CSS classes, etc. on a DOM element
- The "render" function returns many things that can be used specifically for whatever you just rendered, e.g. "getByText" to find an element by its text. One of these is "debug", which will pretty-print the entire tree under the DOM node specified.
- fireEvent is used instead of simulate
- react-testing-library makes use of dom-testing-library (reference). It mostly provides functions like fireEvent, getByLabelText, getByText, and waitForElement.
- To make sure an element is *not* present, you can use "query*" instead of "get*" since "query*" doesn't throw when not found (reference)
- When you don't have an easy way of getting an element, you can specify "data-testid" directly in the dev code. You should try to avoid this where possible.
- The FAQ seems pretty helpful: it covers what you'd need to do for mocking, localization, etc.

Example on react-redux is here. It shows a simple counter component and how to set the initial state for

Redux.

Other learnings

Media queries

Suppose we have a component called TextOrIcon. On desktop, it renders as text, and on mobile, it renders as an icon. Also suppose that this is controlled via a CSS class that has a media query.

My original question was: how would you test to ensure that it's correctly text or an icon?

I think the answer is that you wouldn't, or at least not via this kind of testing. You could verify that the correct CSS class is on the element, but you really be testing CSS if you were to see if it correctly applied "display: none" to the text or image.

Best practices

Import jest-dom/extend-expect from jest.config.json

You're probably going to expect all of the functions from dom-testing-library to be available to all of your tests. These are not available unless you import jest-dom/extend-expect. However, rather than having to import that file in every single test, you can add it in one place (reference).

Assert on the number of matched elements

If you expect exactly 0 elements, then use "query":

```
expect(queryByText('hello world')).toBeUndefined();
```

If you expect exactly 1 element, then use getAll and verify the length:

```
expect(getAllByText('hello world')).toHaveLength(1);
```

Use snapshots when possible

I was originally worried that I would slightly modify the appearance of a component (e.g. something simple like adding a ":" to a name label) and then break all of the tests, but that's kind of what you *want* to have happen, so don't hesitate to use snapshots when testing.

If you know that a component renders a single element and not a Fragment with multiple elements beneath it, then you can make your snapshot a tiny bit easier to read by matching on container.firstChild, that way the outer div is removed. This isn't very important though.

Mocking props

A lot of components that you test will have many props, and it can be a pain to specify them every time. HiDeoo suggested this solution: reference (but parens are missing somewhere). The general idea is to write a function to get the "default" props, then override whatever you want by using the spread operator ("...").

```
const getBasicProps = () => {
  return {
    acceptableDropCssClass: '.draggableHardwareTestEdition',
    conflictSlotHardwareId: null,
    deleteSlot: jest.fn(),
    getItemTemplate: (...args) =>
itemTemplateDepot.getItemTemplate(...args),
    titleText: 'Test Hardware Loadout',
  };
};

test('should render a loadout with some filled slots and some empty
```

```
slots', () => {
  const { container } = render(
    <HardwareLoadout
      {...getBasicProps()}
      titleText={'Override parameters here'}
    />
  );
});
```

Troubleshooting

matcher.test is not a function

Cause #1: you're not using the function that you think you are (AKA "no need to pass the container in")

This happened when I had this really basic code:

```
test('Simple DeltaText test', () => {
  const value = 5;
  const { getByText, container } = render(
    <DeltaText value={value} />
  );

  getByText(container, '5');
});
```

The problem is that I was reading the documentation for dom-testing-library, which *does* take in a Container. However, the getByText that I'm using is returned from react-testing-library's render function (although *only when* you do import 'jest-dom/extend-expect'), and that library already has "Container" implied (reference).

Solution: change to "getByText('5');".

Cause #2: you're not passing in a TextMatch

If you try passing in an argument that is not a string, regex, or function, then you'll get this error:

```
getByText(5); // should be '5'
```

prettyDOM prints color codes

If you ever call prettyDOM and you get something like this:

```
[36m<div [39m
  [33mclass [39m= [32m"undefined" [39m
[36m> [39m
  [36m<div [39m
```

…then it means that your console/output doesn't support the typical terminal color codes. Thankfully, prettyDOM has an "options" object that gets passed to prettyFormat, and one of those options is "**highlight**":

```
import {
  prettyDOM,
} from 'react-testing-library';

prettyDOM(someDomElement, 7000, {highlight: false}); // note: the 7000
is the number of characters to print from the DOM before truncating
```

InteractJS drag events

I have a component that acts as a dropzone for InteractJS, so I wanted to run a test like the following:

1. Make a fake div that has a CSS class on it like "draggableItem"
2. Render my component and tell it to accept "draggableItem"
3. Call fireEvent on the fake div to simulate drag events

I was hoping that InteractJS would trigger ondropactivate, but it didn't even call its own internal functions that may lead to ondropactivate.

I did some investigation, but the conclusion I came to is that this isn't worth worrying about for right now. The investigation follows:

If we put a breakpoint in the client on "collectDrops", we'll be able to see where ondropactivate *should* be called (in that this is where "accept" is processed to see if the CSS class matches). This gives this stack trace:

collectDrops (interact.js:1649)
setActiveDrops (interact.js:1712)
(anonymous) (interact.js:1584)
fire (interact.js:5768)
start (interact.js:929)
onDragMove (makedraggable.js:193)
fireUntilImmediateStopped (interact.js:47)
fire (interact.js:65)
fire (interact.js:5294)
(anonymous) (interact.js:5396)
fire (interact.js:5768)
pointerMove (interact.js:971)
(anonymous) (interact.js:1233)

As you can see, this involves pointerMove. I couldn't figure out why jsdom wasn't doing something similar though. Apparently pointer events weren't simulated in React until 16.4 (which is recent), so it's unlikely that something like jsdom doesn't support it yet. Even if it did, I'm not positive that that's the problem. InteractJS adds its handlers to the document most of the time instead of the element itself.