Testing Components

We'll test components in our Weather app that's built with React and Redux.

WE'LL COVER THE FOLLOWING

- Setup

When testing our components, the one thing we want to verify is that they render the same output as last time the test was ran. Since they are bound to change very often, more specific tests are more of a burden than a help. If the test fails, but we've manually verified the new output is correct we should be able to quickly tell that to our testing framework without much effort.

Exactly for that purpose, Jest recently added support for **component snapshots**. Component snapshots are generated on the first test run and saved in files in your project. They should be checked into version control if you have one, and code reviews should include them.

By having those snapshots after the first test run, we can immediately verify if our component output was changed. If any change happened and we manually verified the new version is correct we can run `jest -u` to update the existing snapshots!

Setup

To render the components without opening a browser we'll have to install the `react-test-renderer`. It allows us to render the component to a JSON object!

```
npm install --save-dev react-test-renderer
```

Let's create a new file and add the basic testing code. We'll be starting with the App component, so `import` that for now:

```
// __tests__/components.test.js
```

```
import React from 'react';
import renderer from 'react-test-renderer';
import App from '../App';

describe('components', function() {
  describe('<App />', function() {

  });
});
```

The thing we want to verify in our `App` component is that it renders without throwing an error and taking a snapshot so we know when the output changes. Let's add an `it` to that effect:

```
// __tests__/components.test.js

import React from 'react';
import renderer from 'react-test-renderer';
import App from '../App';

describe('components', function() {
  describe('<App />', function() {
    it('renders correctly', function() {

    });
  });
});
```

Let's now create a renderer, render our `<App />` component to JSON and expect that to match the snapshot:

```
// __tests__/components.test.js

import React from 'react';
import renderer from 'react-test-renderer';
import App from '../App';

describe('components', function() {
  describe('<App />', function() {
    it('renders correctly', function() {
      var tree = renderer.create(<App />).toJSON();
      expect(tree).toMatchSnapshot();
    });
  });
});
```

Try running this though, and you’ll get this error:

```
- Invariant Violation: Could not find "store" in either the context or props of "Connect(App)". Either wrap the root component in a <Provider>, or explicitly pass "store" as a prop to "Connect(App)".
```

Ugh, what’s this now? Couldn’t find `store`? What?

Remember what we export from the `App.js` file? The `react-redux` `connect`ed component! What we want to test though is the actual component itself, so we’ll have to export that too:

```
// App.js

/* … */

export class App extends React.Component {/* … */}

/* … */

export default connect(mapStateToProps)(App);
```

Awesome! Now we need to change the `import` in our test file to reference that new export and everything should work, right?

```
// __tests__/components.test.js

import { App } from '../App';

/* … */
```

Well, no, but we’re getting a different error now! That’s a good sign!

```
- TypeError: Cannot read property 'getIn' of undefined
```

Remember what `getIn` is used for? ImmutableJS! If you take a look into the component, it expects its `redux` prop to be an ImmutableJS data structure. At the moment, we aren’t passing anything in as a prop so the `getIn` function is undefined.

We can fix that very easily by `import`ing `fromJS` and passing our `<App />` an empty prop of `redux`:

```
import React from 'react';
import renderer from 'react-test-renderer';
import { fromJS } from 'immutable';
import { App } from '../App';

describe('components', function() {
    describe('<App />', function() {
        it('renders correctly', function() {
            var tree = renderer.create(<App redux={fromJS({})} />).toJSON
();
          expect(tree).toMatchSnapshot();
        });
    });
});
```

Awesome, this totally works! When you now run your tests you'll see a new directory inside the `__tests__` directory called `__snapshots__`. It should contain a single file called `components.test.js` that has an export for our `App` component and some HTML as a string.

```
// __tests__/__snapshots__/components.test.js

exports[`components <App /> renders correctly 1`] = `
<div>
  <h1>
    Weather
  </h1>
  <form
    onSubmit={[Function anonymous]}>
    <label>
      I want to know the weather for
      <input
        onChange={[Function anonymous]}
        placeholder="City, Country"
        type="text"
        value={undefined} />
    </label>
  </form>
</div>
`;
```

Now try changing the text in the App component from "I want to know the

weather for” to “I want to know todays weather for” and run `npm run test` again.
This is the output you should see:

```
PASS  src/__tests__/actions.test.js (0.487s)
PASS  src/__tests__/reducer.test.js (0.58s)
FAIL  src/__tests__/components.test.js (1.042s)
● components › <App /> › it renders correctly
 - expected value to match snapshot 1
   - expected + actual

     <div>
       <h1>
         Weather
       </h1>
       <form
         onSubmit={[Function anonymous]}>
         <label>
   -       I want to know the weather for
   +       I want to know todays weather for
           <input
             onChange={[Function anonymous]}
             placeholder="City, Country"
             type="text"
             value={undefined} />
         </label>
       </form>
     </div>

       at Object.<anonymous> (src/__tests__/components.test.js:10:17)

Snapshot Summary
› 1 snapshot test failed in 1 test file. Inspect your code changes or run
with `npm test -- -u` to update them.

snapshot failure, 1 test failed, 19 tests passed (20 total in 3 test suite
s, run time 1.347s)
```

Awesome, Jest caught the changes in the output of our App component and immediately notified us of a potential error! If we wanted to make this the correct text, all we would have to do is run `npm run -- -u` (`-u` stands for “update snapshots”) and Jest would recognize this output as the correct one!

Let’s try to do the same thing for our `Plot`. First, import it and add the testing structure:

```
import Plot from '../Plot.js';

describe('components', function() {
  describe('<App />', function() {/* … */});

  describe('<Plot />', function() {
    it('renders correctly', function() {

    });
  });
});
```

> We don’t need to export the Plot separately here since this isn’t `connect` ed anyway!

Now try adding a first snapshot:

```
import Plot from '../Plot.js';

describe('components', function() {
  describe('<App />', function() {/* … */});

  describe('<Plot />', function() {
    it('renders correctly', function() {
      const tree = renderer.create(<Plot />).toJSON();
      expect(tree).toMatchSnapshot();
    });
  });
});
```

Run `npm run test` and you’ll see this error: `"ReferenceError: Plotly is not defined"`. We use `Plotly.newPlot` in the `drawPlot` method, so at least we know that’s being ran!

We need to pretend to Jest that `Plotly` exists for our component. We do this by adding a new field to the `global` variable which the `Plot` component will try to get `Plotly` from:

```
import Plot from '../Plot.js';
```

```
describe('components', function() {

  describe('<App />', function() {/* … */});

  describe('<Plot />', function() {
    global.Plotly = {
      newPlot: () => {}
    };
    it('renders correctly', function() {
      const tree = renderer.create(<Plot />).toJSON();
      expect(tree).toMatchSnapshot();
    });
  });
});
```

Now that that's “defined” (at least we pretend like it is), let's try running `npm run test` again! Another error, this time saying `"TypeError: Cannot read property 'toJS' of undefined"`?

Wait, didn't we have a similar error before? Exactly, this is an ImmutableJS problem again! Our `Plot` expects two immutable data structures to be passed in as `xData` and `yData`. Soo, let's do that? We have `fromJS` already imported from before, so we just add those as props:

```
import Plot from '../Plot.js';

describe('components', function() {
  describe('<App />', function() {/* … */});

  describe('<Plot />', function() {
    global.Plotly = {
      newPlot: () => {}
    };
    it('renders correctly', function() {
      const tree = renderer.create(<Plot xData={fromJS({})} yData={fromJS
({})} />).toJSON();
      expect(tree).toMatchSnapshot();
    });
  });
});
```

Nothing fancy, let's see what happens now! Ugh, *another* error saying `"ReferenceError: document is not defined"`. How many more errors will we get??

get??

As a short aside, this is what happens when you integrate general JS libraries with React. The nice thing is, React is just JavaScript so contrary to some other frameworks it's possible! That doesn't mean it's easy though, but we're almost through it!

Let's get on with it, since the `react-test-renderer` renders the components in a non-browser context (the command line) we need to tell Jest that the `document` variable we use for `document.getElementById('...').on('...')` isn't undefined for us!

We do this again like `Plotly` above by attaching a new property to `global`:

```
import Plot from '../Plot.js';

describe('components', function() {
  describe('<App />', function() {/* … */});

  describe('<Plot />', function() {
    global.Plotly = {
      newPlot: () => {}
    };
    global.document = {
      getElementById: function() { return {
        on: function() {}
      }}
    };
    it('renders correctly', function() {
      const tree = renderer.create(<Plot xData={fromJS({})} yData={fromJS
({})} />).toJSON();
      expect(tree).toMatchSnapshot();
    });
  });
});
```

Now when you run this, what do you see?!

```
 PASS  src/__tests__/actions.test.js (0.528s)
 PASS  src/__tests__/reducer.test.js (0.623s)
 PASS  src/__tests__/components.test.js (0.947s)

Snapshot Summary
› 1 snapshot written in 1 test file.
› 1 snapshot written in 1 test file.

21 tests passed (21 total in 3 test suites, 2 snapshots, run time 1.242s)
```

Yesss!!! 🎉 We have now successfully tested our entire application, whenever something breaks we now immediately know!