React Testing Part 1

JavaScript Course

Introduction

We've learned how to test our vanilla JavaScript applications in a previous section. Testing is indeed a powerful tool that allows us to write maintainable and flexible code. If you've followed along with our lessons so far, we've been using the Jest framework. For React, we'll keep using Jest and add more capabilities to our tests using the React Testing library.

Lesson Overview

This section contains a general overview of topics that you will learn in this lesson.

- Discern what packages will be needed to test a React app.
- Be able to test UI elements.
- Understand snapshot tests.

Setting Up

We'll need to import some packages inside of our testing file:

```
import React from "react";
import { ... } from "@testing-library/react";
import "@testing-library/jest-dom";  // optional
import userEvent from "@testing-library/user-event";
import TestComponent from "path-to-test-component";
```

- `@testing-library/react` will give us access to useful functions like `render` which we'll demonstrate later on.
- `@testing-library/jest-dom` includes some handy custom matchers (assertive functions) like `toBeInTheDocument` and more. (complete list on jest-dom's github). Jest already has a lot of matchers so this package is not compulsory to use.
- `@testing-library/user-event` provides the `userEvent` API that simulates user interactions with the webpage. Alternatively, we could import the `fireEvent` API from `@testing-library/react`.

> *"Note: `fireEvent` is an inferior counterpart to `userEvent` and `userEvent` should always be preferred in practice."*

- No need to import `jest` since it will automatically detect test files (`*.test.js` or `*.test.jsx`).

That's a lot of setup. But good news! If you're initializing your React repositories with `create-react-app`, then all the above packages come preinstalled and the scripts preconfigured in `package.json`.

Our First Query

First, we'll render the component using `render`. The API will return an object and we'll use destructuring syntax to obtain a subset of the methods required. You can read all about what `render` can do in the React Testing Library API docs about render.

```
// App.js

import React from "react";

const App = () => <h1>Our First Test</h1>;

export default App;
```

```
// App.test.js

import React from "react";
import { render, screen } from "@testing-library/react";
import App from "./App";

describe("App component", () => {
```

Lesson contents

```
  it("renders correct heading", () => {
    render(<App />);
    expect(screen.getByRole("heading").textContent).toMatch(/our first te
  });
});

```

Execute `npm test App.test.js` on the terminal and you'll see that test pass. `getByRole` is just one of the dozen query methods that we could've used. Essentially, queries are classified into three types: `getBy`, `queryBy` and `findBy`. Go through the React Testing Library docs page about queries. Pay extra attention to the "Types of Queries" and "Priority" section.

As stated by the React Testing Library docs, `ByRole` methods are the favored methods for querying, especially when paired with the `name` option. For example, we could improve the specificity of the above query like so: `getByRole("heading", { name: "Our First Test" })`. Queries that are done through `ByRole` ensure that our UI is accessible to everyone no matter what mode they use to navigate the webpage (i.e mouse or assistive technologies).

Simulating User Events

There are numerous ways a user can interact with a webpage. Even though live user feedback and interaction is irreplaceable, we can still build some confidence in our components through tests. Here's a button which changes the heading of the App:

```
// App.js

import React, { useState } from "react";

const App = () => {
  const [heading, setHeading] = useState("Magnificent Monkeys");

  const clickHandler = () => {
    setHeading("Radical Rhinos");
  };

  return (
    <>
      <button type="button" onClick={clickHandler}>
        Click Me
      </button>
      <h1>{heading}</h1>
    </>
  );
};

export default App;
```

Let's test if the button works as intended. In this test suite, we'll use a separate utility to query our UI elements. React Testing Library provides the `screen` object which has all the methods for querying. With `screen`, we don't have to worry about keeping `render`'s destructuring up-to-date. Hence, it's better to use `screen` to access queries rather than to destructure `render`.

```
// App.test.js

import React from "react";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import App from "./App";

describe("App component", () => {
  it("renders magnificent monkeys", () => {
    // since screen does not have the container property, we'll destructu
    const { container } = render(<App />);
    expect(container).toMatchSnapshot();
  });

  it("renders radical rhinos after button click", async () => {
    const user = userEvent.setup();

    render(<App />);
    const button = screen.getByRole("button", { name: "Click Me" });

    await user.click(button);

    expect(screen.getByRole("heading").textContent).toMatch(/radical rhi
  });
});
```

The tests speak for themselves. In the first test, we utilize snapshots to check whether all the nodes render as we expect them to. In the second test, we

simulate a click event. Then we check if the heading changed. `toMatch` is one of the various assertions we could have made.

It's also important to note that after every test, React Testing Library unmounts the rendered components. That's why we render for each test. For a lot of tests for a component, the `beforeEach` jest function could prove handy.

Notice that the callback function for the second test is asynchronous. This is because `user.click()` simulates the asynchronous nature of user interaction, which is now supported by the latest version of the testing library's user-event APIs. As of version 14.0.0, the user-event APIs have been updated to be asynchronous. It's worth noting that some examples from other resources or tutorials might still use the synchronous `userEvent.click()` method

```
// This is the old approach of using userEvent.
it("renders radical rhinos after button click", () => {
  render(<App />);
  const button = screen.getByRole("button", { name: "Click Me" });

  userEvent.click(button);

  expect(screen.getByRole("heading").textContent).toMatch(/radical rhinos
});
```

The `setup()` is internally triggered here. This is still supported by React Testing Library to ease the transition from v13 to v14.

What are Snapshots?

Snapshot testing is just comparing our rendered component with an associated snapshot file. For example, the snapshot file which was automatically generated after we ran the *"magnificent monkeys renders"* test was:

```
// App.test.js.snap

// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`magnificent monkeys render 1`] = `
<div>
  <button
    type="button"
  >
    Click Me
  </button>
  <h1>
    Magnificent Monkeys
  </h1>
</div>
`;
```

It's an HTML representation of the `App` component. And it will be compared against the `App` in future snapshot assertions. If the `App` changes even slightly, the test fails.

Snapshot tests are fast and easy to write. One assertion saves us from writing multiple lines of code. For example, with a `toMatchSnapshot`, we're spared of asserting the existence of the button and the heading. They also don't let unexpected changes creep into our code. Read all about what can be achieved with snapshots in the Jest snapshot docs.

Snapshots might seem the best thing that has happened to us while testing thus far. But we are forced to wonder, *what* exactly are we testing? What's being validated? If a snapshot passes, what does it convey about the correctness of the component?

Snapshot tests may cause false positives. Since we cannot ascertain the validity of the component from a snapshot test, a bug might go undetected. Over-reliance on snapshots can make developers more confident about their code than they should be.

The other issue with snapshots is false negatives. Even the most insignificant of changes compel the test to fail. Fixing punctuation? Snapshot will fail. Replacing an HTML tag to a more semantic one? Snapshot will fail. This might cause us to lose our confidence in the test suite altogether. Snapshots aren't inherently bad; they do serve a purpose. But it's beneficial to understand when to snapshot, and when not to snapshot.

Assignment

1. Take a glance at all of the available query methods on the React Testing

Library's cheatsheet page. There's no need to use them all, but it's optimal to employ a specific method for a specific query. If none of the query methods suffice, there's an option to use test ids. Learn about test ids on the React Testing Library's test id docs.

2. Read the userEvent API docs to get a feel of how to achieve user simulation.

3. This article on the Pros and Cons of Jest Snapshot Tests goes in depth regarding the advantages and disadvantages of snapshot testing. And this one, Snapshot Testing: Benefits and Drawbacks, does an excellent job of explaining what is snapshot testing in programming in general.

Knowledge Check

This section contains questions for you to check your understanding of this lesson on your own. If you're having trouble answering a question, click it and review the material it links to.

- What packages are required for React testing?
- What is the significance of the user-event package?
- What does the `render` method do?
- What is the most preferred method for querying?
- How would you test for a click event with `userEvent`?
- What is the advantage of snapshot tests?
- What are the disadvantages of snapshot tests?

Additional Resources

This section contains helpful links to other content. It isn't required, so consider it supplemental.

- This tutorial on Testing React Apps by Academind is a great overview of what you've learned. It goes into testing async code and callbacks which we haven't covered yet. Though you should be able to follow along using your previous knowledge.
- This Intro to React Testing Library video for a hands-on tutorial.