Intro to React Testing Codealong

(https://github.com/learn-co-curriculum/react-tdd-intro-to-react-testing) (https://github.com/learn-co-curriculum/react-tdd-intro-to-react-testing/issues/new)

Learning Goals

- Identify the testing tools included with Create React App
- Debug React components during testing

Introduction

In this lesson, we'll be creating a new React application using Create React App and exploring the testing tools that come along with it. In doing so, we'll see some of the differences between testing simple functions (as we've been doing so far), and testing user interfaces, as well as some best practices for testing React components.

Create React App

When building React applications, developers have many needs. The React library itself is just one piece of making modern web applications; it's also necessary to have tools like **Babel** (https://babeljs.io/) to compile modern JavaScript and JSX; **webpack** (https://webpack.js.org/) to bundle and minify code for production; **ESLint** (https://eslint.org/) to help with code quality; and more. All of these tools require a fair amount of configuration to use.

In our React lessons, we've used a tool called **Create React App** (https://create-react-app.dev/) to help give us all the above functionality without worrying about needing to configure it ourselves. Create React App *also* has built-in support for testing React components. The tools it uses are:

- **Jest**: the testing framework
- **React Testing Library** (https://testing-library.com/docs/react-testing-library/intro/) : a set of tools for loading React components in a test environment, finding DOM elements, and testing the user interface

You're already familiar with Jest, but what exactly does React Testing Library do for us?

The best way to demonstrate is by example, so let's make a small React application to explore. Run this command to spin up a new React app:

```
$ npx create-react-app react-tdd-codealong --use-npm
```

Then open the new application in your text editor and follow along.

React Testing Library

When you generate a new React application, Create React App provides some starter files. One of these files, `src/App.test.js`, includes an example of a test for the `App` component. Let's check out the test code:

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

test("renders learn react link", () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

In this file, we're using a couple of methods providing by React Testing Library (`@testing-library/react`):

- `render` : This method is used to render a React component inside the testing environment. Our Jest tests don't run in the browser, they run in Node, so one of the challenges of testing React components is that we need to simulate a browser environment within Node. Under the hood, the **JSDOM** (https://github.com/jsdom/jsdom) library simulates a browser environment with browser-specific functionality, like the `document` object, which isn't available in Node. The `render` method then takes our React component and renders it in this simulated browser environment so we can check that it was rendered as expected.
- `screen` : This method provides a way to interact with the simulated browser environment, namely by giving us a number of **query methods** (https://testing-library.com/docs/queries/about#priority) to search the DOM for elements we expect to have been rendered (think of it like a supercharged version of `document.querySelector`).

As you may recall, the `test` method is the same as the `it` method in Jest: it creates a space where we can write our actual test code. Let's break down what's going on in this test:

```
test("renders learn react link", () => {
  // Arrange
  render(<App />);

  // Act
  const linkElement = screen.getByText(/learn react/i);

  // Assert
  expect(linkElement).toBeInTheDocument();
});
```

- **Arrange**: We render the `App` component to the simulated browser environment so we can check what the component displays.
- **Act**: We use a query method (`screen.getByText`) to find a specific element that we expect to be displayed when the component is rendered. The `/learn react/i` syntax is a regular expression that will look for the text "learn react" anywhere on the screen. `i` is a flag for case-insensitive search, so it will match uppercase or lowercase characters.
- **Assert**: We use the `expect` method from Jest to check that the element is actually present in the document using a custom **Jest DOM (https://testing-library.com/docs/ecosystem-jest-dom)** matcher `toBeInTheDocument` , which checks if the element is present in the simulated browser environment. Jest DOM is a custom library of matchers that help write assertions about DOM elements. The code for Jest DOM is imported in the `src/setupTests.js` file, which is where we can do some global setup before all our tests run.

You can run this test now by running `npm test` . Just like our other lessons to this point, this will run Jest in watch mode and we can see the passing test in the terminal.

Check out the `App` component — can you find the DOM element we're testing? Can you write another test for a different DOM element being rendered by the `App` component?

Debugging Tools

One of the challenges of writing tests for user interfaces is that debugging can be more challenging. What we're testing is *what is being displayed to the user*, but when we're running the tests, we're not actually displaying anything. And since the tests are running in Node, we can't use our usual UI debugging tools like the browser's developer tools to see what the DOM looks like.

While we can use `console.log` in our tests, it's not as useful for visualizing DOM elements:

```
test("renders learn react link", () => {
  render(<App />);

  const linkElement = screen.getByText(/learn react/i);

  console.log(linkElement);

  expect(linkElement).toBeInTheDocument();
});
```

This displays info about the actual React DOM element that is returned by our component, but it's not so easy to understand:

```
HTMLAnchorElement {
  '__reactFiber$w022jamqt5l': FiberNode {
    tag: 5,
    key: null,
    elementType: 'a',
    type: 'a',
    stateNode: [Circular *1],
    return: FiberNode {...}
  }
}
```

React Testing Library provides a nice `debug` method to give us a sense of what the DOM looks like when our tests are running. Let's try this out instead:

```
test("renders learn react link", () => {
  render(<App />);

  // add this line:
  screen.debug();

  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Now when we run the tests, we'll get a nice representation of what the DOM looks like for our rendered component printed in the console:

```
<body>
  <div>
    <div class="App">
      <header class="App-header">
        <img alt="logo" class="App-logo" src="logo.svg" />
        <p>
          Edit
          <code> src/App.js </code>
          and save to reload.
        </p>
        <a
          class="App-link"
          href="https://reactjs.org"
          rel="noopener noreferrer"
          target="_blank"
        >
          Learn React
        </a>
      </header>
    </div>
  </div>
</body>
```

We can also use this method to debug a single element:

```
test("renders learn react link", () => {
  render(<App />);

  const linkElement = screen.getByText(/learn react/i);

  // add this line:
  screen.debug(linkElement);

  expect(linkElement).toBeInTheDocument();
});
```

Now we can see what just the `linkElement` looks like:

```
<a
  class="App-link"
  href="https://reactjs.org"
  rel="noopener noreferrer"
  target="_blank"
>
  Learn React
</a>
```

Be sure to keep these debugging tools handy when you're writing tests so you can check if you're testing your elements the right way.

Conclusion

Tools like Create React App and React Testing Library provide developers with a great setup to begin testing React components. These libraries have been refined over the years to take into account many of the best practices of testing user interfaces based on the experiences of many developers. In the coming lessons, we'll learn more about React Testing Library's functionality and philosophy as we work towards testing more advanced React applications.

Resources

- **Create React App: Running Tests (https://create-react-app.dev/docs/running-tests)**
- **React Testing Library (https://testing-library.com/docs/react-testing-library/intro/)**