Testing User Interfaces

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

Learning Goals

- Explain best practices for testing user interfaces
- Understand the importance of accessibility in web applications
- Find elements using accessible queries
- Use Jest DOM matchers to write assertions

Introduction

The end goal of all frontend applications is to create a *user interface* (or "UI" for short): a means by which people can interact with the application. Therefore a big element of testing React applications is testing the user interface layer. In this lesson, we'll learn how to test that the elements of our application are displayed correctly to our users, and some best practices for helping ensure our applications are accessible to users who rely on assistive technology to interact with the web.

Best Practices for Testing User Interfaces

Over the years, a number of frameworks for testing React have risen and fallen in terms of popularity. In general, they've all shared a similar set of features: namely, the ability to render a React component in a simulated browser environment during testing, and the ability to query the simulated DOM to check if an element was displayed or not.

Recently, React Testing Library has become the most popular choice for testing React applications. One big reason for that is because helps guide developers to following best practices when testing by following this **guiding principle** (https://testing-library.com/docs/guiding-principles) :

> The more your tests resemble the way your software is used, the more confidence they can give you.

> — React Testing Library

By following this principle when writing tests, and writing your tests to mimic the way your users would experience your application as much as possible, developers can be more confident that their tests are accurately reflecting the functionality you expect your application to have.

In contrast, some earlier popular testing libraries, like **Enzyme (https://enzymejs.github.io/enzyme/)**, don't enforce best practices as strongly. Enzyme gives developers more tools for testing the *implementation details* of a React component (such as "what methods were called" and "how did state change"), which result in tests that are more brittle (it's harder to refactor a component, since you're testing the code rather than the output) and give you less confidence that the application is behaving the way a user would expect it to.

In general, when testing user interfaces, the goal is to *test the application from the perspective of a user*. In our tests then, we shouldn't care about what code is written inside a component, so long as it renders the right elements to the page based on props and state. How would a user interact with the application? By opening it in the browser and checking what content is on the page. Our goal is to take that same approach when writing tests.

Writing Accessible Applications

One important consideration when designing websites is **accessibility (https://www.w3.org/WAI/fundamentals/accessibility-intro/)**: making sure that as many users as possible can interact with your website, including folks with disabilities.

Different users experience your application in different ways. Some see the elements in the browser, others use a screen reader that tells them what's on the page. As developers, we want our apps to be accessible by as many users as possible, so we need to design them with accessibility in mind.

Thankfully, React Testing Library makes accessibility a first-class citizen by providing query methods that help you find elements in accessible ways (the same ways that a screen reader would find the elements).

> **Note**: while React Testing Library can help with accessibility, it isn't a silver bullet for accessibility; there are other considerations beyond what React Testing Library can tell us. For example, things like colors, contrast, and fonts also impact what users can use our applications. Tools like **axe (https://www.deque.com/axe/)**, and accessibility features in **Chrome's DevTools (https://developer.chrome.com/docs/devtools/accessibility/reference/)**, can help check for other accessibility issues as well.

Finding Elements Using Accessible Queries

In the previous lesson, we saw how to use React Testing Library to find an element based on its *text content* using the `getByText` method:

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

While this does give us confidence that there is *some* element with the text "learn react" being rendered by our component, it doesn't give us much more information than that. What if we wanted to validate that the element in question is a button, or a link, or a heading? Those kinds of distinctions are important to make when designing a user interface; an `<a>` element has a different role to play in our application than a `<span>` does.

To ensure that our tests can be written with these considerations in mind, React Testing Library provides a number of different **query methods (https://testing-library.com/docs/queries/about)** that let us find elements in a variety of different ways.

The query method that is preferred is the **byRole (https://testing-library.com/docs/queries/byrole/)** method, because it reflects the experience of both visual/mouse users as well as folks who use screen readers and other assistive technology.

When using the `byRole` method, we can provide:

- an **ARIA role (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques)** for the element we're looking for
- optionally, text content that should be contained within the element

For example, to find this `<a>` element:

```
<a href="https://reactjs.org">Learn React</a>
```

We could use this query:

```
const linkElement = screen.getByRole("link", {
  name: /learn react/i,
});
```

The above query uses the `"link"` role and the `name` option to specify that we're looking for a *link* that displays the text "learn react". **Anchor tags (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a#properties)** have an implicit ARIA role of "link", which is why this query works.

> **Note**: You can also add explicit ARIA roles to any DOM element:
>
> ```
> 
>   Learn React
> 
> ```
>
> The approach above should be avoided in general — it's best to use semantic HTML elements when possible rather than adding explicit roles, since semantic elements (like anchor tags) have additional built-in behavior (like making the browser navigate to a new page).

Writing queries using ARIA roles ensures that our components render content in such a way that not only displays properly in the browser, but is also accessible to users with screen readers and other assistive technology.

Identifying Accessible Roles

One challenge of testing user interfaces using accessible roles is that it can be difficult to remember what elements are associated with what roles. For example, if we know we want to display an `<a>` tag but don't know what ARIA role it has implicitly, we could look up the **documentation on MDN (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a#properties)**, find the "Properties" section, and see that it has an Implicit ARIA role of "link". That's quite a bit of digging though! A better way to reveal this information is to use some built-in debugging tools from React Testing Library.

To view a list of the implicit ARIA roles rendered by a component, we can use the `logRoles` **(https://testing-library.com/docs/dom-testing-library/api-debugging/#logroles)** method. For example, given the following React component:

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
```

```
            Edit <code>src/App.js</code> and save to reload.
          </p>
          <a
            className="App-link"
            href="https://reactjs.org"
            target="_blank"
            rel="noopener noreferrer"
          >
            Learn React
          </a>
        </header>
      </div>
    );
}
```

We can set up a test that uses the `logRoles` method:

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

test("renders learn react link", () => {
  // "container" represents all the DOM elements rendered by your component
  const { container } = render(<App />);

  // prints out the accessible elements in your component along with their roles
  logRoles(container);

  const linkElement = screen.getByRole("link", { name: /learn react/i });

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

Then when we run the tests, we can see the output of all the implicit ARIA roles:

```
banner:

Name "":
<header
  class="App-header"
/>

--------------------------------------------------
img:

Name "logo":
<img
  alt="logo"
  class="App-logo"
  src="logo.svg"
/>

--------------------------------------------------
link:

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

--------------------------------------------------
```

Use this approach any time you're not sure what the accessible role of an element is.

Finding Elements With Other Query Methods

In addition to the `byRole` method, React Testing Library provides several other methods for finding elements, like `byText` (to find elements based on the display text) and `byAltText` (to find images based on their alt text attribute). Check out the Priority section from the documentation for a list, and for guidance on when to use what query:

- **About Queries - Priority (https://testing-library.com/docs/queries/about#priority)**

React Testing Library also has several different types of query available:

- `getBy...` : either returns an element if one is found, or throws an error if not
- `queryBy...` : either returns an element if one is found, or returns `null` if not
- `findBy...` : asynchronously waits for an element to be present on the page (we'll cover this one in more detail later!)

Each of these query types also has an "all" version (`getAllBy` , `queryAllBy` , `findAllBy`) for when you need to find *multiple* elements with the same query. These are useful if you want to verify that a certain number of elements are present (i.e. if you expect a list to have several items).

Using Jest DOM Matchers

In the previous example, we saw how to test if our application renders a link component with the correct text:

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

In this test, we're using the `toBeInTheDocument` Jest matcher provided by Jest DOM to assert that the link element is in the document. Jest DOM also provides other matcher functions to check our DOM elements.

For instance, if we wanted to check that our link element actually links to the correct URL, we can use the `toHaveAttribute` matcher:

```
test("renders learn react link", () => {
  render(<App />);
  const linkElement = screen.getByRole("link", { name: /learn react/i });
  expect(linkElement).toBeInTheDocument();
  expect(linkElement).toHaveAttribute("href", "https://reactjs.org");
});
```

The **Jest DOM documentation** (https://github.com/testing-library/jest-dom#custom-matchers) has a full list of matchers for working with DOM elements (remember, you can also still use regular Jest matchers like `toEqual` too).

Conclusion

To test user interfaces, we should write tests that behave as much like our users as possible. That means we should prioritize testing *what* is rendered by our components, rather than *how* our components work internally. We should also write tests that use ARIA roles when possible so that we have more confidence in our application's accessibility.

Accessibility is a big topic, and we won't be able to cover it in depth in this course, but it's worth exploring more! Here are some resources for learning more:

- **A11Y Project** (https://www.a11yproject.com/checklist/) : free resources for learning about accessibility
- **React Docs on Accessibility** (https://reactjs.org/docs/accessibility.html) : resources and guideline for designing React components with accessibility in mind
- **React Accessibility course on Egghead** (https://egghead.io/courses/develop-accessible-web-apps-with-react) : a free 90-minute course on using tools to help develop accessible React applications

Resources

- **Testing Library - About Queries** (https://testing-library.com/docs/queries/about)
- **Jest DOM - Custom Matchers** (https://github.com/testing-library/jest-dom#custom-matchers)
- **MDN - ARIA Role Reference** (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques)