Selecting Elements By Role

Selecting elements based upon their role is the preferred way of testing elements with React Testing Library. We use role selectors instead of more classic ways of finding elements, like CSS selectors.

ARIA (Accessible Rich Internet Applications) is a set of attributes that can be added to HTML elements to help make web applications more accessible to users with disabilities. These attributes provide additional information about the purpose and behavior of an element, which can be used by assistive technologies such as screen readers to improve the user experience.

Even though these ARIA roles are an additional topic to memorize, we engineers use them because they allow us to write more flexible tests. In many cases it doesn't matter if an element is presenting text in an `h1` element or an `h3` element. By finding elements based on their role, we can make small changes to a component and not break its respective test. Some engineers do not care for this flexibility. If you don't wish to use ARIA roles, you can always fall back to using standard CSS selectors.

Some elements - not all - are 'implicitly' (or automatically) assigned a role. Some of the more commonly-used roles can be found in the `RoleExample` component below.

```
import { render, screen } from '@testing-library/react';

function RoleExample() {
  return (
    <div>
      <a href="/">Link</a>
      <button>Button</button>
      <footer>Contentinfo</footer>
      <h1>Heading</h1>
      <header>Banner</header>
      <img alt="description" /> Img
      <input type="checkbox" /> Checkbox
```

Many elements have roles that are easy to memorize. Here are some of the easier ones to remember:

Element	Role
`a` with `href`	link
`h1` , `h2` , ..., `h6`	heading
`button`	button
`img` with `alt`	img

Other elements can be a little more challenging to remember. For example:

Element	Role
`input` with `type="number"`	spinbutton

Element	Role
`header`	banner
`footer`	contentinfo

+ Code + Text

```
test('can find elements by role', () => {
  render(<RoleExample />);

  const roles = [
    'link',
    'button',
    'contentinfo',
    'heading',
    'banner',
    'img',
    'checkbox',
    'spinbutton',
```

+ Code + Text

Accessible Names

Sometimes multiple elements of the same type will be displayed by a component, and you will need to find a particular instance of that element. You can be more specific by finding elements based upon their role *and* their accessible name.

The accessible name of most elements is the text placed between the JSX tags. For example, the accessible name of `<a href="/">Home</a>` is `Home`.

In the component below, two `button` elements are displayed. The only difference between them is the text they contain. Their accessible names are `Submit` and `Cancel`, respectively.

+ Code + Text

```
function AccessibleName() {
  return (
    <div>
      <button>Submit</button>
      <button>Cancel</button>
    </div>
  );
}
render(<AccessibleName />);
```

+ Code + Text

Selecting By Accessible Name

Elements with a defined acessible name can be selected by passing a filtering object to the `getByRole` method. Example below.

+ Code + Text

```
test('can select by accessible name', () => {
  render(<AccessibleName />);

  const submitButton = screen.getByRole('button', {
    name: /submit/i
  });
  const cancelButton = screen.getByRole('button', {
    name: /cancel/i
  });

  expect(submitButton).toBeInTheDocument();
  expect(cancelButton).toBeInTheDocument();
```

+ Code + Text

Accessible Names for Inputs

Self-closing elements (also known as 'void elements') like `input`, `img`, and `br` cannot contain text. Defining accessible names for them is done differently.

To define an accessible name for `input` elements in particular, you can associate the input with a `label`. The `input` element should have an assigned `id` prop, and the label should have an identical `htmlFor` prop. Once this link has been formed, the `input` can then be selected by using the `label` text as an accessible name.

+ Code + Text

```
function MoreNames() {
  return (
    <div>
      <label htmlFor="email">Email</label>
      <input id="email" />

      <label htmlFor="search">Search</label>
      <input id="search" />
    </div>
  );
}
render(<MoreNames />);
```

+ Code + Text

```
test('shows an email and search input', () => {
  render(<MoreNames />);

  const emailInput = screen.getByRole('textbox', {
    name: /email/i
  });
  const searchInput = screen.getByRole('textbox', {
    name: /search/i
  });

  expect(emailInput).toBeInTheDocument();
  expect(searchInput).toBeInTheDocument();
```

+ Code + Text

Applying a Name to Other Elements

If you're working with a void element (like a `br` or an `img`), or if you're working with an element that doesn't show plain text, you can apply an accessible name by using the `aria-label` attribute.

In the example below, two `button` elements are being displayed, but they do not contain traditional text. Instead, they are displaying `svg` elements, which are used to display icons.

To select these `button` elements, you can apply an `aria-label` attribute to them. This sets their accessible name.

+ Code + Text

```
function IconButtons() {
  return (
    <div>
      <button aria-label="sign in">
        <svg />
      </button>

      <button aria-label="sign out">
        <svg />
      </button>
    </div>
  );
```

+ Code + Text

```
test('find elements based on label', () => {
  render(<IconButtons />);

  const signInButton = screen.getByRole('button', {
    name: /sign in/i
  });
  const signOutButton = screen.getByRole('button', {
    name: /sign out/i
  });

  expect(signInButton).toBeInTheDocument();
  expect(signOutButton).toBeInTheDocument();
```

+ Code + Text