Working with Events (CodeGrade)

(https://github.com/learn-co-curriculum/react-tdd-events-codealong) (https://github.com/learn-co-curriculum/react-tdd-events-codealong/issues/new)

Learning Goals

- Practice test-driven development (TDD)
- Practice finding elements using accessible query methods
- Use the user-event library to simulate events in tests
- Test changes to component state in response to user events

Introduction

In this lesson, you'll get additional practice using test-driven development and writing tests using Jest and React Testing Library. You'll expand on the tools you've already used, and learn how to write tests that simulate a user event and verify that the component is updated as expected. To do this, we'll use the `user-event` library (https://testing-library.com/docs/ecosystem-user-event/), which allows us to simulate user events (clicks, keyboard events, etc.) in our tests.

The `user-event` library is a great tool because it follows React Testing Library's philosophy of writing our tests to mirror how a user would interact with our components in the real world. It also encourages writing tests that don't test implementation details. We'll see when we start writing tests that we never actually check the internals of our component (like whether or not state was updated). All we do is check what the UI looks like before and after triggering an event, and verify that the event caused some part of our UI to change in an expected way.

Getting Started

For this lesson, we're going to build a simple online ordering system for a pizza parlor. We'll keep it simple to start, with just two choices: cheese (the default) or pepperoni. We'll use test-driven development to build an interface that allows customers to select pepperoni (or not) using a checkbox. We also want to display the list of selected toppings on the page so the customer is confident that their selection is recognized.

Fork and clone this repo and run `npm install` . We'll be doing our coding in `App.js` and `__tests__/App.test.js` . Go ahead and run `npm start` , and open a second tab to run `npm test` . At this point, you'll see an error because our test file is empty.

Creating the App

Before we start coding, let's think about what our app should look like. We want a page with some type of heading ("Select Pizza Toppings", say) and a checkbox that will enable the user to add pepperoni to their pizza if they choose. Below that, we'll want to display a list of the selected toppings. That list should initially have "Cheese" as the only topping, and should update to include "Pepperoni" if the user clicks the checkbox. Finally, we should make sure that if users change their mind, they can remove pepperoni by clicking the checkbox a second time.

In general, when writing tests for user events, we want to do the following:

1. Verify that the initial state of the page is what we want
2. Simulate a user event (in this case, clicking the checkbox)
3. Verify that the state of the page updates as expected

Let's update our `App.test.js` file with some comments laying out how we're going to proceed:

```
// __tests__App.test.js

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

// Test the initial state of the page

// Test the effect of clicking the checkbox

// Test the effect of clicking the checkbox a second time
```

Initial State

We want to include two different tests for the initial state:

```
...
// Test the initial state of the page

test("pizza checkbox is initially unchecked", () => {})

test("toppings list initially contains only cheese", () => {})
...
```

To test that the checkbox is initially unchecked, we'll first need to find that component on the page, then check its status. We'll use `getByRole` to find the component. We already have `render` and `screen` imported, so we can find the checkbox and check its initial status as follows:

```
test("pizza checkbox is initially unchecked", () => {
  render(<App />);

  const addPepperoni = screen.getByRole("checkbox", { name: /add pepperoni/i });

  expect(addPepperoni).not.toBeChecked();
});
```

Here we're looking for an element with the role "checkbox" and the name "Add pepperoni" and saving it to the `addPepperoni` variable. We then use the `toBeChecked` **`jest-dom` matcher (https://github.com/testing-library/jest-dom)** chained to the `not` matcher to verify that the box is initially unchecked.

Once you've added the code above and clicked save, you'll see that our first test is failing. Looking at the error, we see that the test can't find our `addPepperoni` element, which isn't surprising given that we haven't coded it yet. This is the "red" phase of the "red/green/refactor" cycle. So the next step is to write the code to get this test passing.

If we check the **MDN ARIA Reference for checkbox (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles/checkbox_role)** to ensure that we're coding our app with accessibility in mind, we see that we should create a native HTML checkbox element (`<input type="checkbox">`) and include both an `aria-checked` attribute and a label. For now, we can simply set the initial checked status of the checkbox to `false` .

The setup below follows the accessibility guidelines and should enable the `getByRole` query we wrote in our tests to find the element:

```
function App() {
  return (
    <div>
      <h1>Select Pizza Toppings</h1>
      <input
        type="checkbox"
        id="pepperoni"
        checked={false}
        aria-checked={false}
      />
      <label htmlFor="pepperoni">Add pepperoni</label>
    </div>
  );
}

export default App;
```

With the code above, you'll see that our test is now passing! (You'll also see a warning — we'll address that in a bit.) If you have the server running, you can also verify in the browser that the checkbox is present and unchecked.

Now that we've got the first "initial state" test passing, let's work on the second one:

```
// Test the initial state of the page
test("toppings list initially contains only cheese", () => {});
```

We'll be creating an unordered list of the pizza ingredients, and we want to check what items it includes. Checking the **MDN - ARIA Role Reference (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques)** again, we see that the `listitem` **role (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles/listitem_role)** will identify `li` elements inside a `ul` or `ol` element. We can use this along with `getAllByRole` to access all the `li` s on our page, and verify that initially there is only one:

```
test("toppings list initially contains only cheese", () => {
  render(<App />);

  expect(screen.getAllByRole("listitem").length).toBe(1);
});
```

However, we also want to verify that the *text* of the `li` matches what we're expecting ("Cheese"). To do this, we'll need to add another query that uses `getByText` :

```
expect(screen.getByText("Cheese")).toBeInTheDocument();
```

Finally, we can explicitly check that "Pepperoni" is *not* included on the page. We can't use `getByText` in this case because the test will throw an error. We'll use `queryByText` instead, which will return `null` if the element isn't found:

```
expect(screen.queryByText("Pepperoni")).not.toBeInTheDocument();
```

Our test should now look like this:

```
test("toppings list initially contains only cheese", () => {
  render(<App />);

  expect(screen.getAllByRole("listitem").length).toBe(1);
  expect(screen.getByText("Cheese")).toBeInTheDocument();
  expect(screen.queryByText("Pepperoni")).not.toBeInTheDocument();
});
```

Save the file and check the tests and you'll see that, as expected, our second test is failing. So the next step is to write the code for our list of toppings. Since "Cheese" is the default topping, we can simply hard code it into the list:

```
...
<label htmlFor="pepperoni">Add pepperoni</label>

<h2>Your Toppings:</h2>
<ul>
  <li>Cheese</li>
</ul>
```

Once you've added this code, the second test should be passing, and the list should now appear in the browser.

Clicking the Checkbox

The behavior we expect when the user clicks the "Add Pepperoni" checkbox is:

1. The checkbox will appear checked in the DOM
2. Pepperoni will be added to the list of toppings

So let's create those two tests:

```
// Test the effect of clicking the checkbox
test("checkboxes become checked when user clicks them", () => {});

test("topping appears in toppings list when checked", () => {});
```

For the first test, we first need to render the app and find the checkbox, just as we did before:

```
test("checkboxes become checked when user clicks them", () => {
  render(<App />);
  const addPepperoni = screen.getByRole("checkbox", { name: /add pepperoni/i });
});
```

We can then use the `user-event` `click` event to simulate the user clicking the checkbox. We'll need to import the `userEvent` object first:

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

Then we can use it to click the checkbox:

```
test("checkboxes become checked when user clicks them", () => {
  render(<App />);

  const addPepperoni = screen.getByRole("checkbox", { name: /add pepperoni/i });
  userEvent.click(addPepperoni);
});
```

And, finally, verify that the checkbox is now checked:

```
expect(addPepperoni).toBeChecked();
```

The full test looks like this:

```
// Test the effect of clicking the checkbox
test("checkbox appears as checked when user clicks it", () => {
  render(<App />);

  const addPepperoni = screen.getByRole("checkbox", { name: /add pepperoni/i });

  userEvent.click(addPepperoni);
```

```
    expect(addPepperoni).toBeChecked();
  });
```

With this, we're ready to start writing the code. To get it working we'll need to:

- Add state to keep track of the status of the checkbox
- Use the state variable to control the checkbox's `checked` and `aria-checked` properties
- Create a callback function to update state when the checkbox is clicked
- Add an `onChange` property to the `input` and assign the callback as the handler

The updated code should look like this:

```
import { useState } from "react";

function App() {
  const [pepperoniIsChecked, setPepperoniIsChecked] = useState(false);

  function togglePepperoni(e) {
    setPepperoniIsChecked(e.target.checked);
  }

  return (
    <div>
      <h1>Select Pizza Toppings</h1>
      <input
        type="checkbox"
        id="pepperoni"
        checked={pepperoniIsChecked}
        aria-checked={pepperoniIsChecked}
        onChange={togglePepperoni}
      />
      <label htmlFor="pepperoni">Add pepperoni</label>
      <h2>Your Toppings:</h2>
      <ul>
        <li>Cheese</li>
      </ul>
    </div>
  );
}

export default App;
```

The third test is now passing, and the warning we've been getting should be gone as well. On to the next one!

```
test("topping appears in toppings list when checked", () => {});
```

For this, we can reuse our second test, with just a couple of changes. Specifically, we now expect to see two list items rather than one, and we expect "Pepperoni" to appear in the list:

```
test("topping appears in toppings list when checked", () => {
  render(<App />);

  const addPepperoni = screen.getByRole("checkbox", { name: /add pepperoni/i });

  userEvent.click(addPepperoni);

  expect(screen.getAllByRole("listitem").length).toBe(2);
  expect(screen.getByText("Cheese")).toBeInTheDocument();
  expect(screen.getByText("Pepperoni")).toBeInTheDocument();
});
```

Then, to add that functionality to our app, we'll use a simple ternary:

```
  <h2>Your Toppings:</h2>
  <ul>
    <li>Cheese</li>
    {pepperoniIsChecked ? <li>Pepperoni</li> : null}
  </ul>
```

Our code is now complete! Just to be sure everything works as intended, however, let's also verify that we can toggle the checkbox on and off and that the page updates to match.

To do this we simply need to add a new test and use two `userEvent.click` events in our test to verify that clicking the checkbox a second time removes the "Pepperoni" item from the page:

```
test("selected topping disappears when checked a second time", () => {
  render(<App />);

  const addPepperoni = screen.getByRole("checkbox", { name: /add pepperoni/i });

  userEvent.click(addPepperoni);

  expect(addPepperoni).toBeChecked();
  expect(screen.getByText("Cheese")).toBeInTheDocument();
  expect(screen.getByText("Pepperoni")).toBeInTheDocument();

  userEvent.click(addPepperoni);

  expect(addPepperoni).not.toBeChecked();
  expect(screen.getByText("Cheese")).toBeInTheDocument();
  expect(screen.queryByText("Pepperoni")).not.toBeInTheDocument();
});
```

The tests are now all passing - we've done it!

Conclusion

In this lesson, we used a test-driven development approach to write a simple pizza ordering app. We started by identifying the expected behavior for our app, including the initial state and what we expect to happen when the user clicks the checkbox. To simulate the user's action, we used the `user-event` (https://testing-library.com/docs/ecosystem-user-event/) library.

So far, we've only scratched the surface of the types of user events that can be tested. We'll learn about more a bit later in the next lesson, where we'll look at how to test forms in React.

Resources

- **user-event (https://testing-library.com/docs/ecosystem-user-event/)**
- **Testing Library - About Queries (https://testing-library.com/docs/queries/about/)**
- **MDN - ARIA Role Reference (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques)**

This tool needs to be loaded in a new browser window

Load Working with Events (CodeGrade) in a new window