Testing Forms (CodeGrade)

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

Learning Goals

- Find form elements using accessible query methods
- Use the `user-event` library to simulate form events
- Test changes to component state in response to form events

Introduction

In this lesson, we will continue to learn about using `user-event` to simulate and test user events, this time in the context of web forms. Specifically, we'll look at how to identify accessible form elements and how to simulate and test form events.

To follow along, fork and clone this repo, then run `npm install` and `npm test`. You may also want to run `npm start` in a separate tab so you can see our progress in the browser as well. We'll be doing our coding in `src/App.js` and `src/App.test.js`.

Building on to the Pizza Ordering App

Let's return to our pizza restaurant example. We have the topping-selection functionality set up, but there's a bit more information we need from our customers, including what size pizza they'd like and their contact information. We'll also need to add a button to submit the order. When the order is submitted, we want to display a message so the customer knows their order has been received.

As before, we'll use test-driven development. Specifically, for a given element of the UI, we'll:

- Identify the behavior we want
- Write the tests to check for that behavior

- Write the code to make the tests pass

You'll see in the code files that we've already made a few updates, including wrapping the checkbox element inside a form element. From here, our goals are to:

1. add an input for selecting the size of the pizza; we'll use a dropdown
2. display the selected size and toppings in response to user actions
3. add a text input where the customer will enter their email address
4. add an "Order" button that will submit the form

Select the Pizza Size

We want to give the user the ability to select the size pizza they'd like from a dropdown menu; we can do this by using a `select` element. You learned in the last lesson that, when using an `input` element, it is important to include a label for it for accessibility reasons; the same applies to other form elements. Once we've done that for an element, we can use the **getByLabelText (https://testing-library.com/docs/queries/bylabeltext/)** query to find it, and know that our form and tests follow accessibility guidelines.

We want to set up our dropdown so that it defaults to "Small", so let's write the test for that:

```
// src/__tests__/App.test.js

// ... rest of tests

// Size select element
test("size select element initially displays 'Small'", () => {});
```

Inside our test, we first need to render `App`, then use `getByLabelText` to find the element. Finally, we'll write our assertion. We can use **toHaveDisplayValue (https://github.com/testing-library/jest-dom#tohavedisplayvalue)** to check that the "Small" option is selected:

```
test("size select element initially displays 'Small'", () => {
  render(<App />);

  const selectSize = screen.getByLabelText(/select size/i);
```

```
  expect(selectSize).toHaveDisplayValue("Small");
});
```

Next, we'll write the code to pass this new test. Thinking ahead a little, we eventually want to display the selected size on the page. To do that, we'll need to have access to the size in state, so let's go ahead and make our `select` element a controlled input. We'll start by adding state and creating a callback to update state when the selected size is changed:

```
const [size, setSize] = useState("small");
const selectSize = (e) => setSize(e.target.value);
```

Then we'll add the element inside our `<form>` :

```
<form>
  {/* rest of form...*/}
  <div>
    <h3>Size</h3>
    <label htmlFor="select-size">Select size: </label>
    <select id="select-size" value={size} onChange={selectSize}>
      <option value="small">Small</option>
      <option value="medium">Medium</option>
      <option value="large">Large</option>
    </select>
  </div>
</form>
```

The `value` attribute will display whichever size is currently selected, and the `onChange` handler will update the value in response to the user's selection.

Let's also verify that the display updates when the user selects a different size. We can simulate this user action by calling **userEvent.selectOptions (https://testing-library.com/docs/ecosystem-user-event/#selectoptionselement-values-options)** and passing two arguments: our `selectSize` callback function and the `value` to select.

```
test("select Size dropdown displays the user's selected value", () => {
  render(<App />);

  const selectSize = screen.getByLabelText(/select size/i);

  userEvent.selectOptions(selectSize, "medium");

  expect(selectSize).toHaveDisplayValue("Medium");

  userEvent.selectOptions(selectSize, "large");

  expect(selectSize).toHaveDisplayValue("Large");
});
```

Because we've set the dropdown up as a controlled input, we should be passing this test without having to make any additional changes to the code. To verify that it's working, you can temporarily change one of the expected values, which should cause the test to fail.

Display Selections on the Page

Next, we want to display the user's selections on the page (e.g., "medium cheese"). Because the topping defaults to "cheese" and the size defaults to "small", we expect our initial state to show "small cheese". We're simply displaying something to the screen here, so we'll use `getByText` to find our element:

```
// "Your Selection" text
test("'Your Selection' message initially displays 'small cheese'", () => {
  render(<App />);

  expect(screen.getByText(/small cheese/i)).toBeInTheDocument();
});
```

Then, to get our test passing, we'll add a `p` element above the form:

```
<div>
  <h1>Place an Order</h1>
  <p>
    Your selection: {size} {pepperoniIsChecked ? "pepperoni" : "cheese"}
  </p>
  ...
</div>
```

Finally, we want to verify that the message on the screen updates when the user changes either the size or the toppings:

```
test("selecting options updates the 'Your selection' message", () => {
  render(<App />);

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

  userEvent.click(addPepperoni);

  expect(screen.getByText(/small pepperoni/i)).toBeInTheDocument();

  userEvent.selectOptions(selectSize, "large");

  expect(screen.getByText(/large pepperoni/i)).toBeInTheDocument();
});
```

We have set up the element to access the current values in state, so this test should be passing without making any additional changes to the code.

Enter the Contact Info

Next we'll add a text box to allow our customers to enter their contact info. We want to verify that the text box has "email address" as placeholder text. We can test that the text box appears on the page and has the desired placeholder text with a single test by using

getByPlaceholderText (https://testing-library.com/docs/queries/byplaceholdertext/) :

```
// "Contact Info" text box
test("'Contact Info' text box initially displays a placeholder value of 'email address'", () => {
  render(<App />);

  expect(screen.getByPlaceholderText(/email address/i)).toBeInTheDocument();
});
```

As with our checkbox and dropdown menu, we'll add a label to our text box for accessibility and set it up as a controlled input in the form:

```
const [contactInfo, setContactInfo] = useState("");
const updateContactField = (e) => setContactInfo(e.target.value);

return (
  <div>
    <h1>Place an Order</h1>
    <p>
      Your selection: {size} {pepperoniIsChecked ? "pepperoni" : "cheese"}
    </p>
    <form>
      {/*... rest of form*/}
      <div>
        <h3>Contact Info</h3>
        <label htmlFor="email">Enter your email address: </label>
        <input
          type="text"
          value={contactInfo}
          id="email"
          placeholder="email address"
          onChange={updateContactField}
        />
      </div>
```

```
      </form>
    </div>
  );
```

We have the test passing for the initial state, so now we just need to test what happens when the user enters an email address into the box. This time, we'll search for the element using `getByLabelText`, and we'll use **userEvent.type (https://testing-library.com/docs/ecosystem-user-event/#typeelement-text-options)** to simulate the user typing in a value. `userEvent.type` takes two arguments: the text box element, and the value to be entered.

```
test("the page shows information the user types into the contact form field", () => {
  render(<App />);

  const contact = screen.getByLabelText(/enter your email address/i);

  userEvent.type(contact, "pizzafan@email.com");

  expect(contact).toHaveValue("pizzafan@email.com");
});
```

Once again, because our input is controlled, we're passing this test without making any additional changes to the code.

Submit the Order

Our final task is to add a submit button so customers can submit their order. We'll start by testing that the button exists on the page. We can use `getByRole` to find the button and `toBeInTheDocument` to verify that it's on the page:

```
// Submit Order button
test("form contains a 'Submit Order' button", () => {
  render(<App />);

  expect(
    screen.getByRole("button", { name: /submit order/i })
```

```
  ).toBeInTheDocument();
});
```

Then to get this test passing we'll add our button to the bottom of the form:

```
<form>
  {/*... rest of form*/}
  <button type="submit">Submit Order</button>
</form>
```

When the user submits the order, we want to display a message:

```
test("clicking the Place Order button displays a thank you message", () => {
  render(<App />);

  userEvent.click(screen.getByRole("button", { name: /submit order/i }));

  expect(screen.getByText(/thanks for your order!/i)).toBeInTheDocument();
});
```

To get this passing, we'll keep track of the order submission status in state, and create a `submitOrder` callback function:

```
const [orderIsSubmitted, setOrderIsSubmitted] = useState(false);
const submitOrder = (e) => {
  e.preventDefault();
  setOrderIsSubmitted(true);
};
```

We'll then add an `onSubmit` attribute to our form that will call `submitOrder`, and use a ternary to conditionally render the message if the order has been submitted:

```
<form onSubmit={submitOrder}>{/*...rest of form*/}</form>;
{
  orderIsSubmitted ? <h2>Thanks for your order!</h2> : null;
}
```

Here's what the completed component should look like:

```
function App() {
  const [pepperoniIsChecked, setPepperoniIsChecked] = useState(false);
  const [size, setSize] = useState("small");
  const [contactInfo, setContactInfo] = useState("");
  const [orderIsSubmitted, setOrderIsSubmitted] = useState(false);

  const togglePepperoni = (e) => setPepperoniIsChecked(e.target.checked);

  const selectSize = (e) => setSize(e.target.value);

  const updateContactField = (e) => setContactInfo(e.target.value);

  const submitOrder = (e) => {
    e.preventDefault();
    setOrderIsSubmitted(true);
  };

  return (
    <div>
      <h1>Place an Order</h1>
      <p>
        Your selection: {size} {pepperoniIsChecked ? "pepperoni" : "cheese"}
      </p>
      <form onSubmit={submitOrder}>
        <div>
```

```
        <h3>Toppings</h3>
        <input
          type="checkbox"
          id="pepperoni"
          checked={pepperoniIsChecked}
          aria-checked={pepperoniIsChecked}
          onChange={togglePepperoni}
        />
        <label htmlFor="pepperoni">Add pepperoni</label>
      </div>
      <div>
        <h3>Size</h3>
        <label htmlFor="select-size">Select size: </label>
        <select id="select-size" value={size} onChange={selectSize}>
          <option value="small">Small</option>
          <option value="medium">Medium</option>
          <option value="large">Large</option>
        </select>
      </div>
      <div>
        <h3>Contact Info</h3>
        <label htmlFor="email">Enter your email address: </label>
        <input
          type="text"
          value={contactInfo}
          id="email"
          placeholder="email address"
          onChange={updateContactField}
        />
      </div>
      <button type="submit">Submit Order</button>
    </form>
    {orderIsSubmitted ? <h2>Thanks for your order!</h2> : null}
```

```
      </div>
    );
  }
```

Conclusion

In this lesson, you expanded upon what you already know about Jest to learn how to find and test form elements. As you've seen, there are many ways to accomplish our goals. The resources listed below should help you navigate all the tools and options available for using Jest to test React apps:

Queries:

- **Testing Library: Queries (https://testing-library.com/docs/queries/about)** : Overview of Testing Library queries
- **React Testing Library Cheatsheet (https://testing-library.com/docs/react-testing-library/cheatsheet/#!)** : Additional queries provided by React Testing Library

Matchers:

- **Jest Matchers (https://jestjs.io/docs/using-matchers)** : Overview of Jest matchers
- **jest-dom (https://github.com/testing-library/jest-dom)** : Additional matchers provided by `jest-dom`

User Events:

- **Testing Library: user-event (https://testing-library.com/docs/ecosystem-user-event/)**

Accessibility:

- **Testing Library Queries - Priority (https://testing-library.com/docs/queries/about/#priority)**
- **MDN: ARIA Roles (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques)**
- **W3C: Accessible Forms Tutorial (https://www.w3.org/WAI/tutorials/forms/)**

This tool needs to be loaded in a new browser window

Load Testing Forms (CodeGrade) in a new window