Testing User Interfaces Lab (CodeGrade)

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

Learning Goals

- Create React component following test-driven development
- Find elements using accessible queries
- Use Jest DOM matchers to write assertions

Introduction

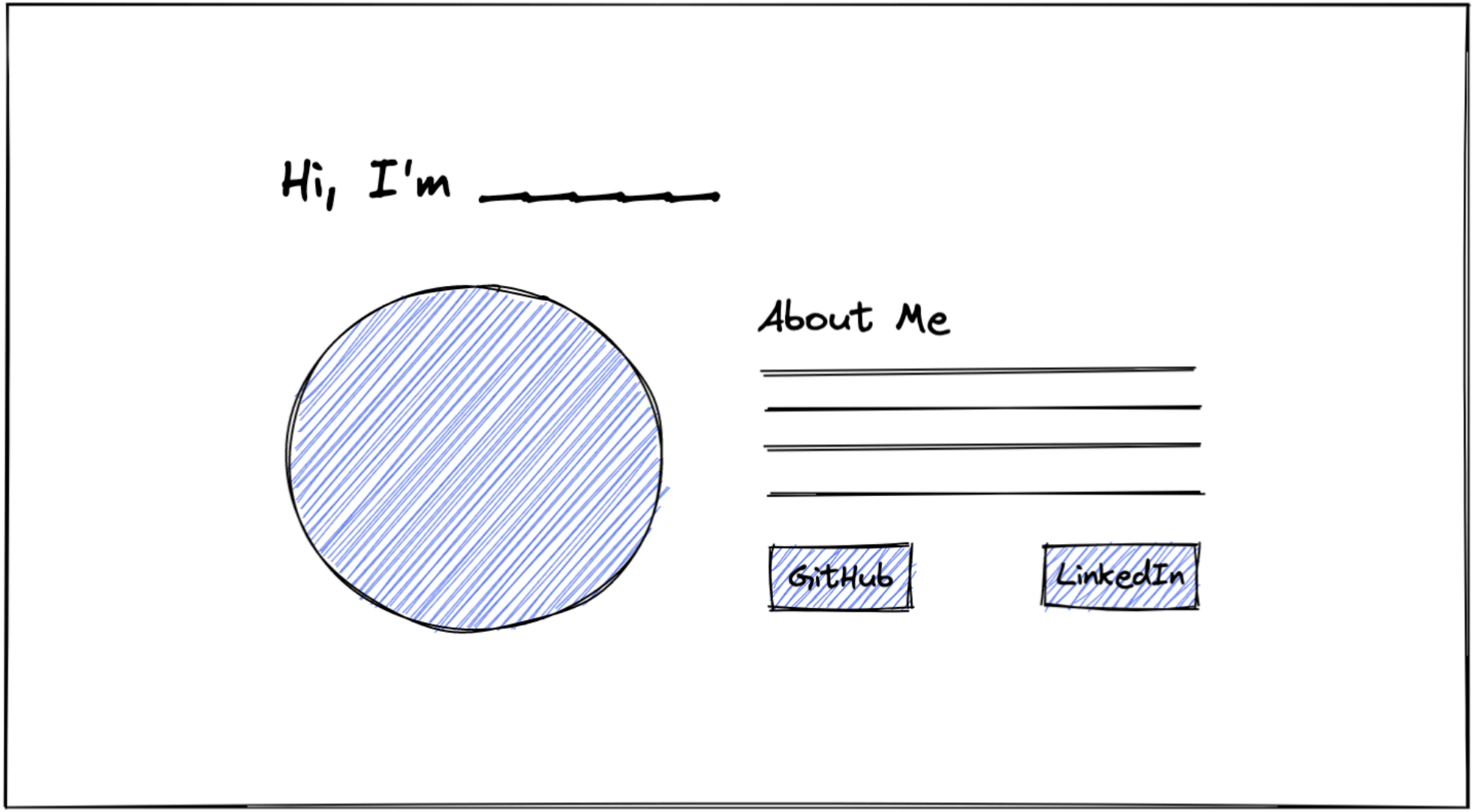
In this lesson, we'll be building a React component following a test-driven development workflow using the concepts we learned in the last lesson.

To get started, fork this lesson and run `npm install`.

Since you'll be responsible for writing the tests, there aren't any tests provided with the starter code. You can check your work against the solution branch once you're finished.

Instructions

In this lesson, we'll be building a React component that represents the home page of a portfolio site for yourself based on this wireframe:



Our component won't have any state or props — all we're concerned about is what it renders to the page. Based on the wireframe, we'll need:

- A top-level heading with the text `Hi, I'm _______`
- An image of yourself with alt text identifying the content of the image
- A second-level heading with the text `About Me`
- A paragraph for your biography

- Two links, one to your GitHub page, and one to your LinkedIn page

We'll work on the first test together, and leave it up to you to write the rest!

> **Note**: don't worry about styling/positioning elements on the page exactly to match the wireframe just yet — our focus isn't on CSS here. You should prioritize writing tests and using your component to render the correct DOM elements.

We'll be writing tests in the `src/__tests__/App.test.js` file, and using the `src/App.js` file for our component. Code along with this example for practice, then it'll be time to write your own tests!

Let's start by our TDD process by writing a test for the top-level heading:

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

test("displays a top-level heading with the text `Hi, I'm _______`", () => {
  // Arrange
  // Act
  // Assert
});
```

In our test, we need a way to do the following:

- Render the `<App>` component
- Find the top-level heading
- Assert that our element is in the document

We can use the `render` method to load our `<App>` component in the test:

```
test("displays a top-level heading with the text `Hi, I'm _______`", () => {
  // Arrange
  render(<App />);
  // Act
```

```
  // Assert
});
```

Next, we need to figure out a way to find the element. We could use the `getByText` method and just check for an element with the right text contents, but a better approach would be to use the `getByRole` method and use an accessible role to check for a `heading` (https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles/heading_role) element:

```
test("displays a top-level heading with the text `Hi, I'm _______`", () => {
  // Arrange
  render(<App />);

  // Act
  const topLevelHeading = screen.getByRole("heading", {
    name: /hi, i'm/i,
    exact: false,
    level: 1,
  });

  // Assert
});
```

Here, we're using a couple additional options with the `byRole` (https://testing-library.com/docs/queries/byrole/) method:

- `name: /hi, i'm/i` : we're using a case-insensitive regular expression to match the text "hi, i'm" in the element
- `exact: false` : partial matches will be included
- `level: 1` : we expect this to be a top-level `<h1>` element (not a `<h2>`, `<h3>`, etc.)

Finally, we can assert that the element we found is in the document.

```
test("displays a top-level heading with the text `Hi, I'm _______`", () => {
  // Arrange
  render(<App />);
```

```
    // Act
    const topLevelHeading = screen.getByRole("heading", {
      name: /hi, i'm/i,
      exact: false,
      level: 1,
    });

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

This last step is a bit redundant, since `getByRole` will throw an error (and thus fail our test) if the element isn't found, but it's useful to use some kind of matcher to complete the story of our test and fully describe the behavior we're testing for.

Now that we've written our test, let's run the test suite and verify that our tests are failing. Great!

Now all that's left is to update our component so that our tests pass:

```
function App() {
  return (
    <div>
      <h1>Hi, I'm (your name)</h1>
    </div>
  );
}
```

Writing Your Tests

Now that you've seen the TDD workflow in action for testing React components, it's time to write your own tests and build out the rest of the features from the wireframe:

- An image of yourself with alt text identifying the content of the image
- A second-level heading with the text `About Me`

- A paragraph for your biography
- Two links, one to your GitHub page, and one to your LinkedIn page

When writing your tests, try as much as possible to use accessible queries such as `getByRole` or `getByAltText`. You should also use the `toHaveAttribute` Jest DOM matcher to check the attributes of some DOM elements, like the `src` attribute of an image or the `href` attribute of a link.

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)**

This tool needs to be loaded in a new browser window

Load Testing User Interfaces Lab (CodeGrade) in a new window