Running Tests in React

- Due No Due Date
- Points 1
- Submitting a website url

(https://github.com/learn-co-curriculum/react-hooks-running-tests) (https://github.com/learn-co-curriculum/react-hooks-running-tests/issues/new)

Learning Goals

- Use Jest to run tests in React applications
- Read test files and identify the purpose of test code

Introduction

In this lab, we'll discuss how the tests are set up for the labs in a typical React application, and give some tips for running tests.

Fork and clone this lesson by navigating to the GitHub repo with the "OctoCat" icon above so you can code along!

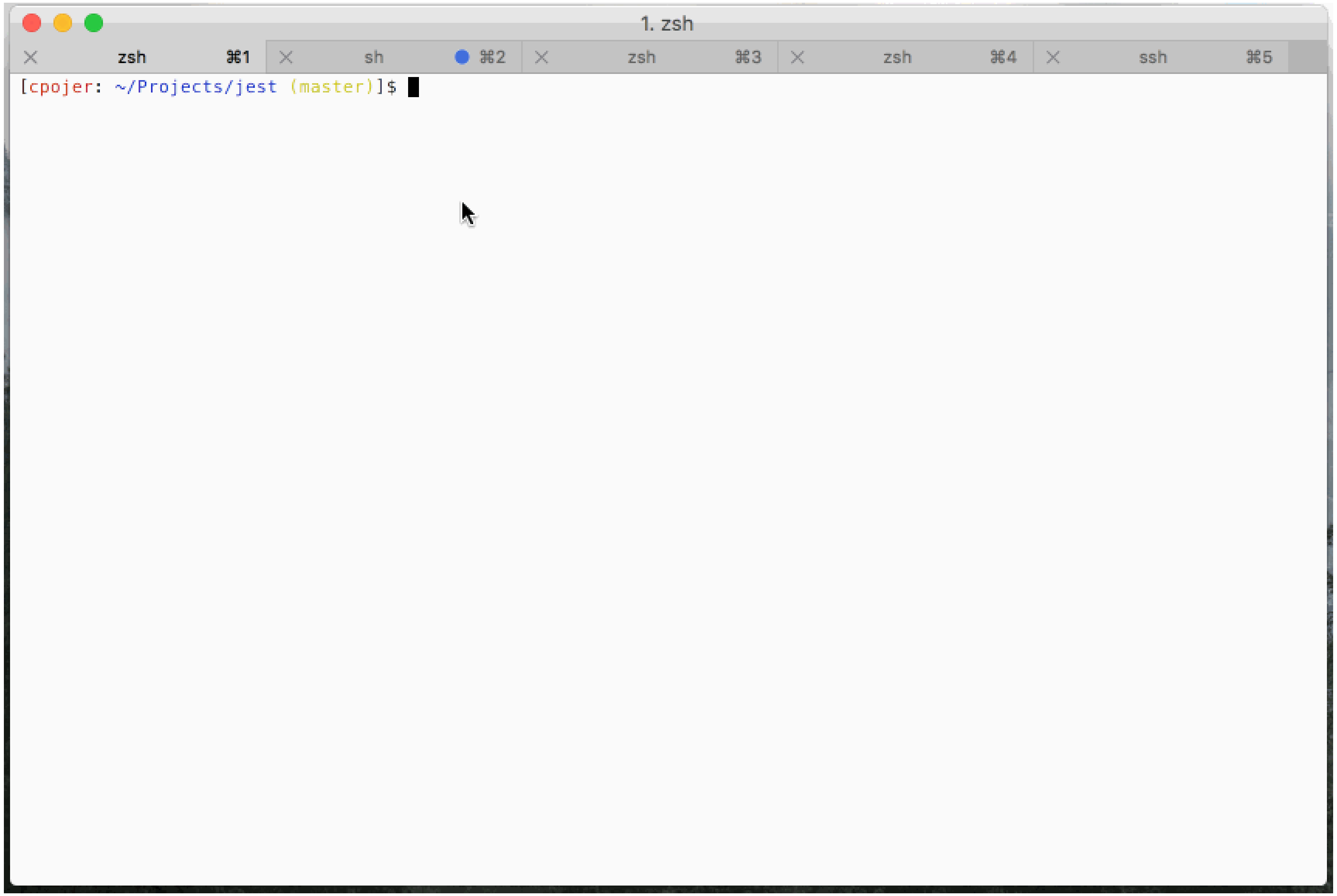
Running Jest Tests

There are a number of JavaScript test frameworks out there. For testing vanilla JavaScript applications, one popular choice is Mocha, which we use for the tests in our JavaScript labs.

Jest (https://jestjs.io/) is another popular choice for JavaScript developers, and in particular the React community. Jest, like React, was developed by Facebook and is an open source project. You can read the **Jest docs** (https://jestjs.io/docs/getting-started) if you're curious to learn more.

Jest comes preinstalled when you generate a React project using `create-react-app`, so all you have to do to run tests in React labs is run `npm test`, which will execute the test script found in the `package.json` file.

Running `npm test` should produce output like this in your terminal:



```
[cpojer: ~/Projects/jest (master)]$
```

This command will run all tests in the `src` directory by looking for files that have `.test.js` in the file name (you'll typically find them in the `__tests__` directory for our labs).

The tests are set to run in "watch mode" by default, so after running `npm test`, any changes you make to your components will cause the tests to run again. That means you can keep the tests running as you work!

Reading Test Files

> **Note**: some of the explanations in this section depend on you understanding a thing or two about React, so don't worry if this doesn't all make sense just yet! You can refer back to this lesson when you're curious about what the tests in our lessons are looking for.

In our React lessons, the tests use **Jest (https://jestjs.io/)** and **React Testing Library (https://testing-library.com/docs/react-testing-library/intro)** to test your React code. Tests will be written in the folder `src/__tests__` , and typically there will be one test file for each component. A typical test file looks something like this:

```
// import libraries needed for testing
import "@testing-library/jest-dom";
import { render, screen } from "@testing-library/react";
import React from "react";

// import the component you wrote
import Article from "../components/Article";

// test the component
test("displays the text 'please pass this test'", () => {
  render(<Article />);

  expect(screen.queryByText("please pass this test")).toBeInTheDocument();
});
```

You can ignore most of the setup code at the top: this is just giving us access to code that is needed to test your React components, and to access the code in your component file itself. Let's focus on the test itself:

```
test("displays the text 'please pass this test'", () => {
  render(<Article />);
  expect(screen.queryByText("please pass this test")).toBeInTheDocument();
});
```

This example is doing the following:

- `render` : This method is used to render a React component inside the testing environment. Our Jest tests don't run in the browser, they run in Node, so one of the challenges of testing React components is that we need to simulate a browser environment within Node. Under the hood, the **JSDOM** (https://github.com/jsdom/jsdom) library simulates a browser environment with browser-specific functionality, like the `document` object, which isn't available in Node. The `render` method then takes our React component and renders it in this simulated browser environment so we can check that it was rendered as expected.
- `expect` : This is a **Jest** (https://jestjs.io/) method, to which we pass some expected value for testing. In this case, we're using a custom **Jest DOM** (https://testing-library.com/docs/ecosystem-jest-dom) matcher `toBeInTheDocument` , which checks if the element is present in the simulated browser environment after our component is rendered.
- `screen` : This method provides a way to interact with the simulated browser environment, namely by giving us a number of **query methods** (https://testing-library.com/docs/queries/about#priority) to search the DOM for elements we expect to have been rendered (think of it like a supercharged version of `document.querySelector`). `screen.queryByText()` searches the virtual DOM for some element that has the text content `"please pass this test"` . If no element is found, it will return `null` .

So, all together, the test is:

- rendering the `<Article>` component in a virtual environment
- looking for an element that has the text `"please pass this test"`
- if the element is found in the document, the test passes; if not, the test fails

Code Along

There are a couple of tests defined for this lab so you can get some practice.

To get started, run `npm install` (if you haven't already), then run `npm test` .

> **Pro tip**: you can use the shorthand `npm i` and `npm t` to run the install and test scripts as well!

You should see something like this in the output:

```
 FAIL  src/__tests__/Header.test.js
  ● displays the text 'hello from the Header!'

    expect(received).toBeInTheDocument()

    received value must be an HTMLElement or an SVGElement.
    Received has value: null

       8 |   render(<Header />);
       9 |
    > 10 |   expect(screen.queryByText("hello from the Header!")).toBeInTheDocument();
         |                                                        ^
      11 | });
      12 |

      at __EXTERNAL_MATCHER_TRAP__ (node_modules/expect/build/index.js:342:30)
      at Object.<anonymous> (src/__tests__/Header.test.js:10:56)

 FAIL  src/__tests__/Article.test.js
  ● displays the text 'please pass this test'

    expect(received).toBeInTheDocument()

    received value must be an HTMLElement or an SVGElement.
    Received has value: null

       8 |   render(<Article />);
       9 |
    > 10 |   expect(screen.queryByText("please pass this test")).toBeInTheDocument();
         |                                                        ^
      11 | });
      12 |
      at __EXTERNAL_MATCHER_TRAP__ (node_modules/expect/build/index.js:342:30)
      at Object.<anonymous> (src/__tests__/Article.test.js:10:56)

Test Suites: 2 failed, 2 total
Tests:       2 failed, 2 total
Snapshots:   0 total
Time:        3.486 s
Ran all test suites.

Watch Usage
 › Press f to run only failed tests.
 › Press o to only run tests related to changed files.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

Let's focus on the `Header.test.js` file first. To tell Jest to only run tests on this one file, press the **p** key in your terminal (this will let you filter out tests by their filename). In the next screen, type in `Header` :

```
Pattern Mode Usage
 › Press Esc to exit pattern mode.
 › Press Enter to filter by a filenames regex pattern.

 pattern › Header

 Pattern matches 1 file
 › src/__tests__/Header.test.js
```

Then, press the Enter key to run tests in the `Header.test.js` file only.

See if you can get this test passing by updating the code in `src` > `components` > `Header.js` as follows:

```
import React from "react";

function Header(props) {
  return <h1>hello from the Header!</h1>;
}

export default Header;
```

Next, press the **a** key in your terminal to tell Jest to run **all** tests. Try getting the tests for the `Article` component to pass too.

When you've finished, you can hit the **q** key to exit Jest.

Debugging Tools

One of the challenges of writing tests for user interfaces is that debugging can be more challenging. What we're testing is *what is being displayed to the user*, but when we're running the tests, we're not actually displaying anything. And since the tests are running in Node, we can't use our usual UI debugging tools like the browser's developer tools to see what the DOM looks like.

React Testing Library provides a nice `debug` method to give us a sense of what the DOM looks like when our tests are running. Let's try this out instead:

```
// src/__tests__/Article.test.js
test("displays the text 'please pass this test'", () => {
  render(<Article />);

  // add this line
  screen.debug();

  expect(screen.queryByText("please pass this test")).toBeInTheDocument();
});
```

Now when we run the tests, we'll get a nice representation of what the DOM looks like for our rendered component printed in the console:

```
<body>
  <div>
    <div>please pass this test</div>
  </div>
</body>
```

We can also use this method to debug a single element:

```
test("displays the text 'please pass this test'", () => {
  render(<Article />);

  const element = screen.queryByText("please pass this test");

  screen.debug(element);

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

Now we can see what just the `element` found by `screen.queryByText("please pass this test")` looks like:

```
<div>please pass this test</div>
```

Be sure to keep these debugging tools handy when you're writing tests so you can check if you're testing your elements the right way.

Conclusion

Being able to run the tests and understand the output is an important skill to develop as a React developer. Make sure to take some time reading through the error messages when your tests aren't passing. It's also helpful to run the actual application in the browser to understand what your components are doing. At the end of the day, the job of the tests is to guide you in the right direction in your code, but what's most important is how your application actually works in the browser, so make sure to check that as well when you're writing your code!

If you're stuck wondering what a particular test is asking for, check out the resources links below (particularly for React Testing Library) to help get a better sense of how the test is written.

Resources

- **Jest (https://jestjs.io/)**
- **React Testing Library (https://testing-library.com/docs/react-testing-library/intro)**
- **Jest DOM (https://testing-library.com/docs/ecosystem-jest-dom)**