



SINGLE PAGE FRONT ENDS USING REACT PT 2.

Unit 3.1 Styling and Testing React Apps

```
function() {  
  //is the element hidden?  
  if (!t.is(':visible')) {  
    //it became hidden  
    t.appeared = false;  
    return;  
  }  
  
  //is the element inside the visible window?  
  var a = w.scrollLeft();  
  var b = w.scrollTop();  
  var o = t.offset();  
  var x = o.left;  
  var y = o.top;  
  
  var ax = settings.accX;  
  var ay = settings.accY;  
  var th = t.height();  
  var wh = w.height();  
  var tw = t.width();  
  var ww = w.width();  
  
  if (y + th + ay >= b &&  
      y <= b + wh + ay &&  
      x + tw + ax >= a &&  
      x <= a + ww + ax) {  
    //trigger the custom event  
    if (!t.appeared) t.trigger('appear', settings.data);  
  } else {  
    //it scrolled out of view  
    t.appeared = false;  
  }  
};  
  
//create a modified fn with some additional logic  
var modifiedFn = function() {  
  //mark the element as visible  
  t.appeared = true;  
  //supposed to happen only once?
```



BY THE END OF THIS LESSON **YOU SHOULD BE ABLE TO...**

- 01** Style JSX elements via inline style assignments or with the help of CSS classes.
- 02** Set inline and class styles, both statically and dynamically or conditionally.
- 03** Build reusable components that allow for style customization.
- 04** Utilize CSS Modules to scope styles to components.
- 05** Understand the core idea behind styled-components, a third-party CSS-in-JS library.





INTRODUCTION

01

So far we've focuses on writing JSX code to render our websites.

02

This imbues the site with structure, but it does nothing for making them look nice.

03

Cascading Style Sheets (CSS) is required to dress up our sites to make them look great on all kinds of devices.

04

Today we'll look at the special considerations for using CSS as part of our React app.





HOW DOES STYLING WORK IN REACT APPS?



TLDR: Same as it does in plain HTML!



Index.css is created when you run create-react-app.



You can put your apps CSS rules right in index.css.



Note, though that index.html doesn't have a reference to index.css.



The reference goes in the index.js file as an import!



```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

index.html X

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8" />
5     <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
6     <meta name="viewport" content="width=device-width, initial-scale=1" />
7     <meta name="theme-color" content="#000000" />
8     <meta
9       name="description"
10      content="Web site created using create-react-app"
11    />
12    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
13    <!--
14      manifest.json provides metadata used when your web app is installed on a
15      user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
16    -->
17    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
18    <!--
19      Notice the use of %PUBLIC_URL% in the tags above.
20      It will be replaced with the URL of the 'public' folder during the build.
21      Only files inside the 'public' folder can be referenced from the HTML.
22
23      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
24      work correctly both with client-side routing and a non-root public URL.
25      Learn how to configure a non-root public URL by running 'npm run build'.
26    -->
27    <title>React App</title>
28  </head>
```





USING INLINE STYLES

```
function TodoItem() {  
  return <li style={{color: 'red', fontSize: '18px'}}>Learn React!</li>;  
};
```

Note the different
syntax from
vanilla HTML!



The above code is TOTALLY LEGAL!



And UTTERLY DISCOURAGED!



We want to separate concerns...so structure & style go in separate files!



It makes our HTML easier to read and understand (because we're focusing on the structure).



It makes our CSS easier to read and understand (because we're focused on all the rules together as a whole).



Also think about how separate CSS makes it easy to apply themes to your app...just a bit of code to change which CSS file we're pointing at and the whole look of the site changes in an instant!



SETTING STYLES VIA CSS CLASSES

```
.goal-item {  
  color: red;  
  font-size: 18px;  
}
```

Using classes
with CSS is
allowed in React!

```
return (  
  <ul>  
    <li className="goal-item">Learn React!</li>  
    <li className="goal-item">>Master React!</li>  
  </ul>  
);
```

Just remember to
use className in
your JSX!



SETTING STYLES

DYNAMICALLY - INLINE



```
function ColoredText() {  
  const [enteredColor, setEnteredColor] = useState('');  
  function updateTextColorHandler(event) {  
    setEnteredColor(event.target.value);  
  };  
  return (  
    <>  
      <input type="text" onChange={updateTextColorHandler}/>  
      <p style={{color: enteredColor}}>This text's color changes dynamically!</p>  
    </>  
  );  
};
```





SETTING STYLES

DYNAMICALLY - GLOBAL



```
function TodoPriority() {  
  const [chosenPriority, setChosenPriority] = useState('low-prio');  
  function choosePriorityHandler(event) {  
    setChosenPriority(event.target.value);  
  };  
  return (  
    <>  
      <p className={chosenPriority}>Chosen Priority: {chosenPriority}</p>  
      <select onChange={choosePriorityHandler}>  
        <option value="low-prio">Low</option>  
        <option value="high-prio">High</option>  
      </select>  
    </>  
  );  
};
```





CONDITIONAL STYLES



```
function TextInput({isValid, isRecommended, inputConfig}) {  
  let cssClass = 'input-default';  
  if (isRecommended) {  
    cssClass = 'input-recommended';  
  }  
  if (!isValid) {  
    cssClass = 'input-invalid';  
  }  
  return <input className={cssClass} {...inputConfig} />  
};
```





COMBINING MULTIPLE DYNAMIC CSS CLASSES



```
function ExplanationText({children, isImportant}) {  
  let cssClasses = 'text-default text-expl';  
  if (isImportant) {  
    cssClasses = 'text-important';  
  }  
  return <p className={cssClasses}>{children}</p>;  
}
```



String concatenation:

```
cssClasses = cssClasses + ' text-important';
```



Using a template literal:

```
cssClasses = `${cssClasses} text-important`;
```



Joining an array:

```
cssClasses = [cssClasses, 'text-important'].join(' ');
```





MERGING MULTIPLE INLINE STYLE OBJECTS



```
function ExplanationText({children, isImportant}) {  
  let defaultStyle = { color: 'black' };  
  if (isImportant) {  
    defaultStyle = { ...defaultStyle, backgroundColor: 'red' };  
  }  
  return <p style={defaultStyle}>{children}</p>;  
}
```

The spread
operator
comes in
handy!

BUILDING COMPONENTS

WITH CUSTOMIZABLE STYLES

You can build a component that comes pre-styled, but then allows the consumer to use `className` to override your styling:

```
function Button({children, config, className}) {  
  let cssClasses = 'btn';  
  if (className) {  
    cssClasses = className  
  }  
  return <button {...config} className={cssClasses}>{children}</button>;  
};
```



CUSTOMIZATION WITH FIXED CONFIGURATION OPTIONS



```
function TextBox({children, mode}) {  
  let cssClasses;  
  if (mode === 'alert') {  
    cssClasses = 'box-alert';  
  } else if (mode === 'info') {  
    cssClasses = 'box-info';  
  }  
  return <p className={cssClasses}>{children}</p>;  
};
```





THE PROBLEM WITH **UNSCOPED STYLES**



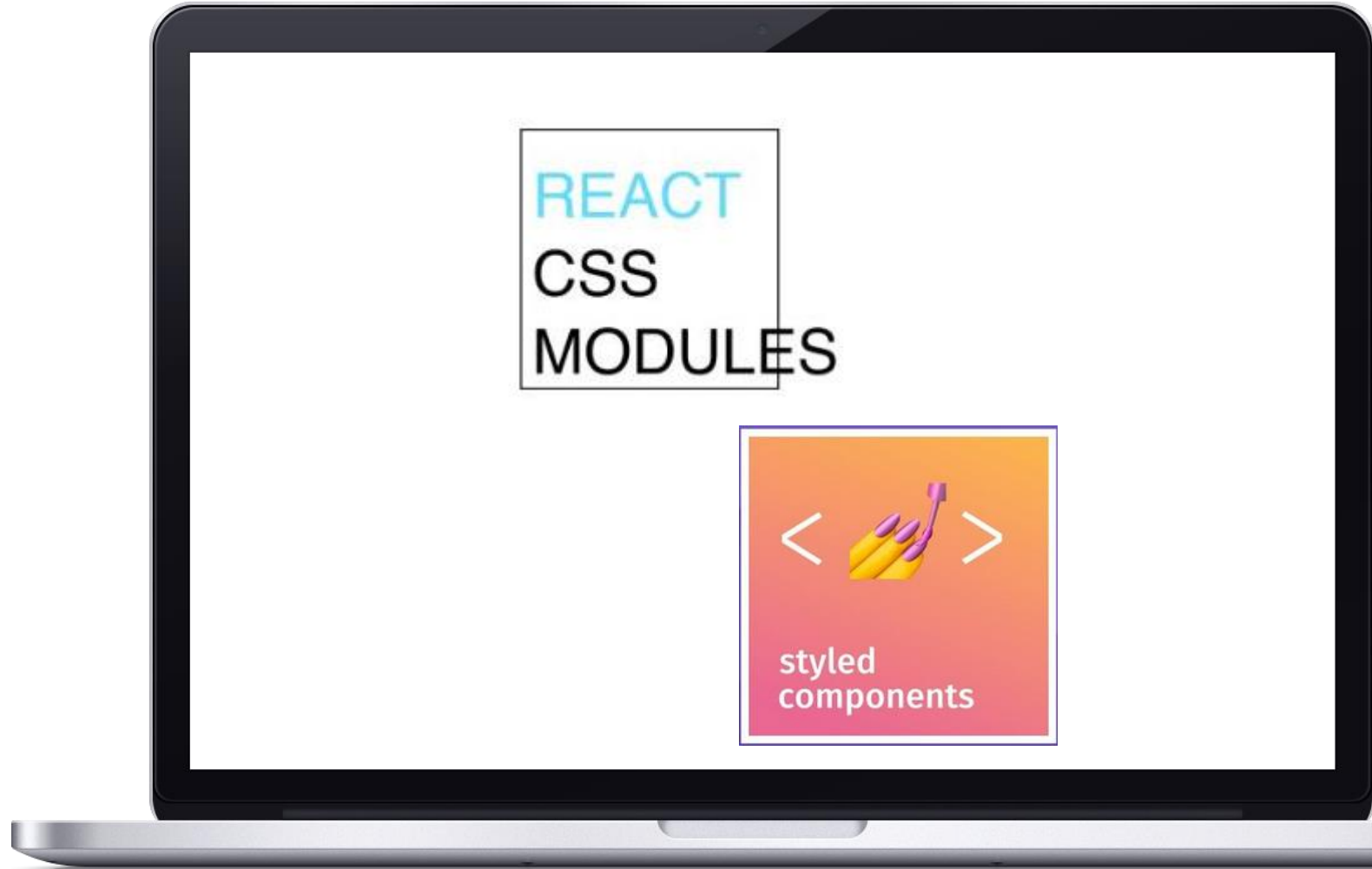
When teams are all working to contribute components to a common app it's easy to write conflicting styles that clash and cause hard to understand UI problems.



There are two solutions to this problem:

- **CSS Modules** - supported out of the box when you use **create-react-app**
- **Styled Components** - a third-party library called **styled-components**.

Let's look at each of these!





SCOPED STYLES WITH **CSS MODULES**



To use CSS Modules you just have to name your CSS files the right way: `<anything>.module.css`



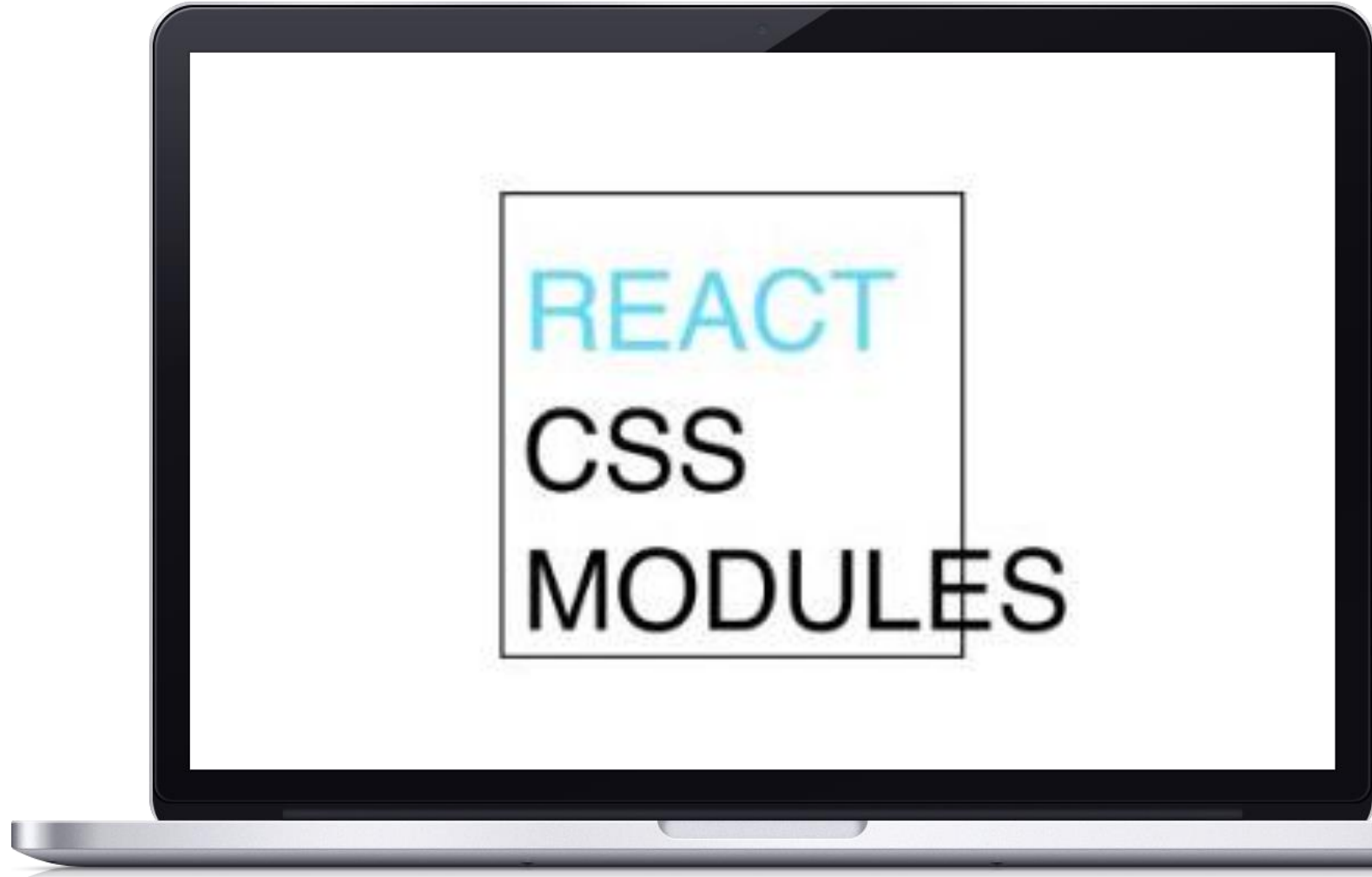
The `<anything>` can be whatever you want. And the `module.css` tips off React that you want to use CSS Modules.



The import is also a little different:
`import classes from './file.module.css';`



ONLY use classes in your `.module.css` files other types of style used in these will be global still!





THE STYLED COMPONENTS LIBRARY



This is a CSS in JS type of library.



You will put CSS information right in your JS code.



npm install styled-components.



```
import styled from 'styled-components';

const Button = styled.button`
  background-color: #370566;
  color: white;
  border: none;
  padding: 1rem;
  border-radius: 4px;
`;

export default Button;
```





OTHER CSS AND STYLING LIBRARIES & FRAMEWORKS



Because ReactJS is just JavaScript you can use any tools you want.



Some libraries will have React specific extensions that allow them to be state aware, or to provide additional hooks, but you can use any that you'd like.



Animate.css, Bootstrap, Tailwind are just a few that work well.





[DAILY ASSIGNMENT]



ASSIGNMENT 6.1:

PROVIDING INPUT VALIDITY FEEDBACK UPON FORM SUBMISSION

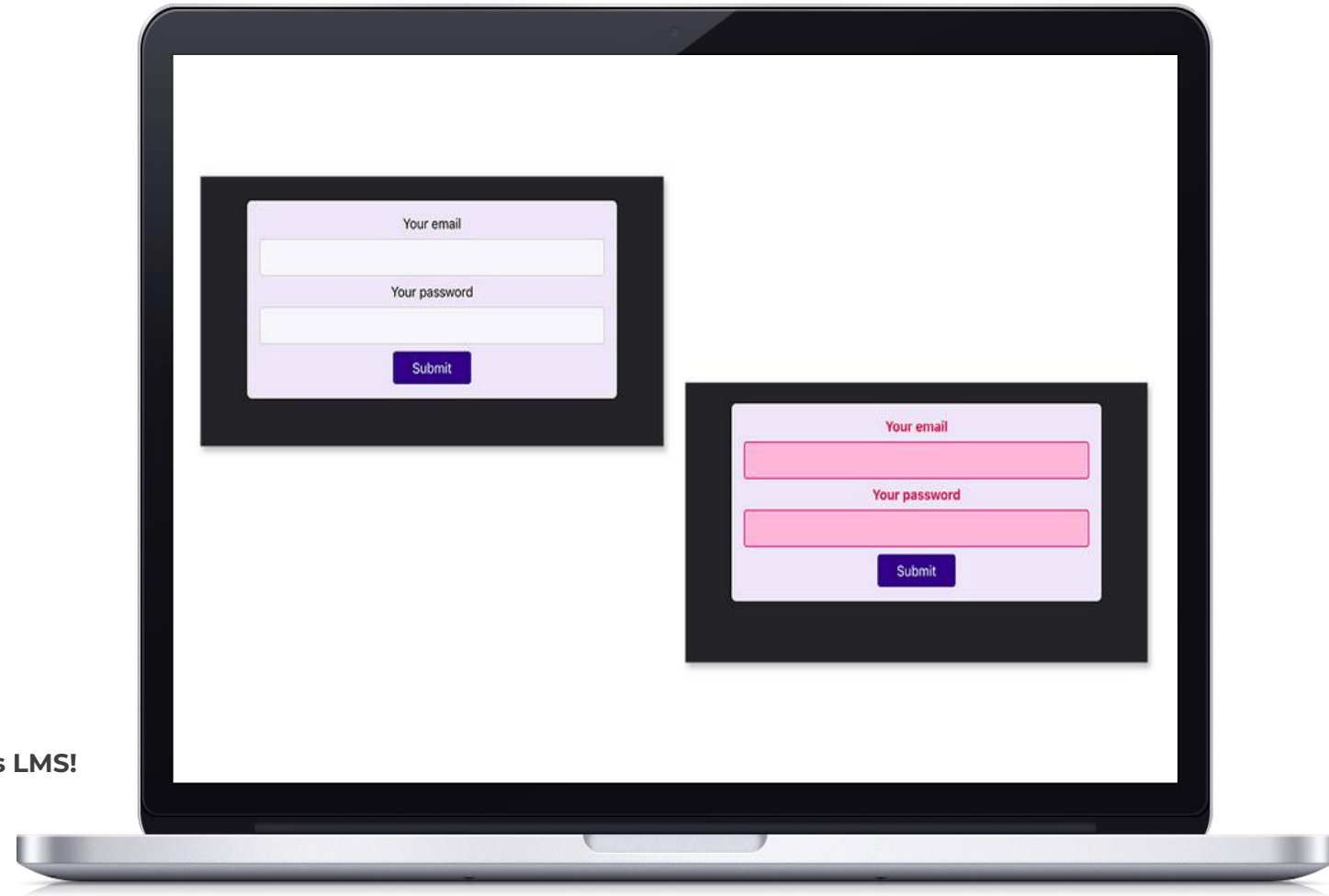


In this assignment, you will build a basic form that allows users to enter an email address and a password. The provided input of each input field is validated, and the validation result is stored (for each individual input field).



The aim of this assignment is to add some general form styling and some conditional styling that becomes active once an invalid form has been submitted. The exact styles are up to you, but for highlighting invalid input fields, the background color of the affected input field must be changed, as well as its border color and the text color of the related label.

Push your work to a Github repo and submit the link via the Emeritus LMS!



See the text book for full details!



ASSIGNMENT 6.2:

USING CSS MODULES FOR STYLE SCOPING

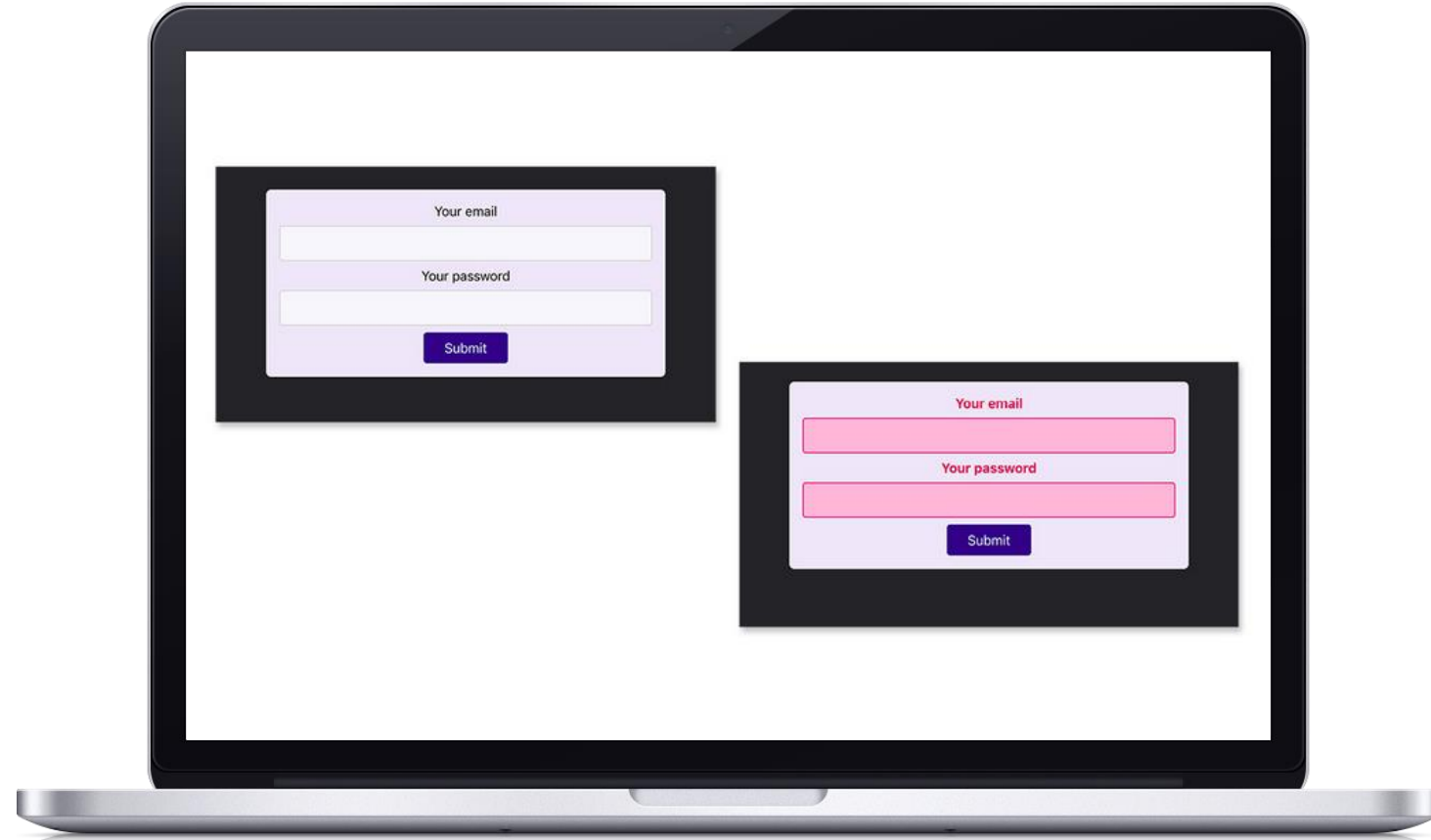


In this assignment, you'll take the final app built in Assignment 6.1 and adjust it to use CSS Modules. The goal is to migrate all component-specific styles into a component-specific CSS file, which uses CSS Modules for style scoping.



The final user interface therefore looks the same as it did in the previous activity. But the styles will be scoped to the Form component so that clashing class names won't interfere with styling.

Push your work to a Github repo and submit the link via the Emeritus LMS!



See the text book for full details!



[**BREAK**]



TESTING REACT APPS



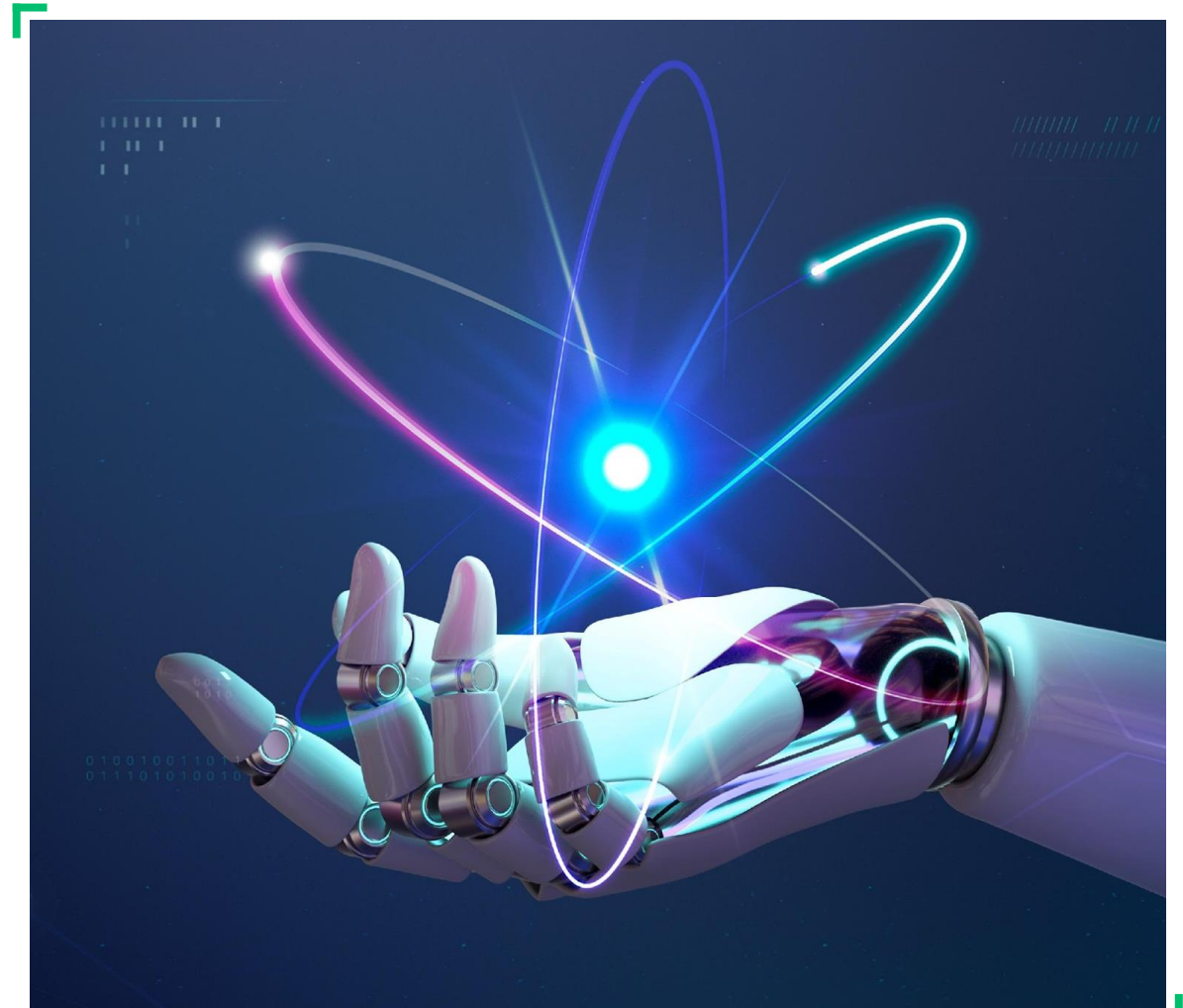
BY THE END OF THIS LESSON
YOU SHOULD BE ABLE TO...

01

Effectively write tests before code using TDD best practices when working with React applications.

02

Understand the basics of mocking fetch calls when working with tests in React.





REACT TESTING LIBRARY



Now that we are creating websites and not just writing simple applications we need additional tooling to help do things like trigger events and simulate the activities of a real user when we do our testing.



React Testing Library is a JavaScript testing utility built specifically to test React components. It simulates user interactions on isolated components and asserts their outputs to ensure the UI is behaving correctly.





WHAT IS REACT TEST LIBRARY?



It works with Jest – it's not a replacement for it.



It IS a replacement for Enzyme.



It lets you inspect the state of the DOM to ensure your rendered output is correct.



You can access specific DOM nodes.



You can inject events and user actions to test their impact.



Ensures that you use best practices!



Comes built in with apps built with create-react-app





GUIDING PRINCIPLES

We try to only expose methods and utilities that encourage you to write tests that closely resemble how your web pages are used. Utilities are included in this project based on the following guiding principles:

01

If it relates to rendering components, then it should deal with DOM nodes rather than component instances, and it should not encourage dealing with component instances.

02

It should generally be useful for testing the application components in the way the user would use it. We are making some trade-offs here because we're using a computer and often a simulated browser environment, but in general, utilities should encourage tests that use the components the way they're intended to be used.

03

Utility implementations and APIs should be simple and flexible.

At the end of the day, what we want is for this library to be pretty light-weight, simple, and understandable.



QUERIES



Help you find elements on the page.



Usually you query for an element first.



Then you fire an event to simulate a user interaction.



Finally you use Jest to make assertions about the element.

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

test('should show login form', () => {
  render(<Login />)
  const input = screen.getByLabelText('Username')
  // Events and assertions...
})
```



TYPES OF QUERIES



Single Elements

- `getBy...` : Returns the matching node for a query, and throw a descriptive error if no elements match or if more than one match is found (use `getAllBy` instead if more than one element is expected).
- `queryBy...` : Returns the matching node for a query, and return `null` if no elements match. This is useful for asserting an element that is not present. Throws an error if more than one match is found (use `queryAllBy` instead if this is OK).
- `findBy...` : Returns a Promise which resolves when an element is found which matches the given query. The promise is rejected if no element is found or if more than one element is found after a default timeout of 1000ms. If you need to find more than one element, use `findAllBy`.



Multiple Elements

- `getAllBy...` : Returns an array of all matching nodes for a query, and throws an error if no elements match.
- `queryAllBy...` : Returns an array of all matching nodes for a query, and return an empty array (`[]`) if no elements match.
- `findAllBy...` : Returns a promise which resolves to an array of elements when any elements are found which match the given query. The promise is rejected if no elements are found after a default timeout of 1000ms.
 - `findBy` methods are a combination of `getBy*` queries and `waitFor`. They accept the `waitFor` options as the last argument (i.e. `await screen.findByText('text', queryOptions, waitForOptions)`)



Type of Query	0 Matches	1 Match	>1 Matches	Retry (Async/Await)
Single Element				
<code>getBy...</code>	Throw error	Return element	Throw error	No
<code>queryBy...</code>	Return <code>null</code>	Return element	Throw error	No
<code>findBy...</code>	Throw error	Return element	Throw error	Yes
Multiple Elements				
<code>getAllBy...</code>	Throw error	Return array	Return array	No
<code>queryAllBy...</code>	Return <code>[]</code>	Return array	Return array	No
<code>findAllBy...</code>	Throw error	Return array	Return array	Yes



FIRING EVENTS



`userEvent` fires off full interactions (an interaction is usually several events).



`fireEvent` just triggers specific events.



Usually you'll want to use `userEvent` unless you are testing some interaction that is undefined with `userEvent` or you need very fine grained control over your event testing.



Here's an example of clicking a button (both `fireEvent` and `userEvent`):

```
// <button>Submit</button>
fireEvent(
  getByText(container, 'Submit'),
  new MouseEvent('click', {
    bubbles: true,
    cancelable: true,
  }),
)
```

```
// inlining
test('trigger some awesome feature when clicking the button', async () => {
  const user = userEvent.setup()
  render(<MyComponent />)

  await user.click(screen.getByRole('button', {name: /click me!/i}))

  // ...assertions...
})
```



ASYNC METHODS

01

Several utilities are provided for dealing with asynchronous code. These can be useful to wait for an element to appear or disappear in response to an event, user action, timeout, or Promise.

02

The async methods return Promises, so be sure to use `await` or `.then` when calling them.





FINDBY EVENTS



`findBy` methods are a combination of `getBy` queries and `waitFor`. They accept the `waitFor` options as the last argument (e.g. `await screen.findByText('text', queryOptions, waitForOptions)`).



`findBy` queries work when you expect an element to appear but the change to the DOM might not happen immediately.

```
const button = screen.getByRole('button', {name: 'Click Me'})
fireEvent.click(button)
await screen.findByText('Clicked once')
fireEvent.click(button)
await screen.findByText('Clicked twice')
```



WAITFOR






When in need to wait for any period of time you can use `waitFor`, to wait for your expectations to pass. Here's a simple example:

```
function waitFor<T>(  
  callback: () => T | Promise<T>,  
  options?: {  
    container?: HTMLElement  
    timeout?: number  
    interval?: number  
    onTimeout?: (error: Error) => Error  
    mutationObserverOptions?: MutationObserverInit  
  },  
) : Promise<T>
```

```
// ...  
// Wait until the callback does not throw an error. In this case, that means  
// it'll wait until the mock function has been called once.  
await waitFor(() => expect(mockAPI).toHaveBeenCalledTimes(1))  
// ...
```


waitForElementToBeRemoved

-  To wait for the removal of element(s) from the DOM you can use `waitForElementToBeRemoved`. The `waitForElementToBeRemoved` function is a small wrapper around the `waitFor` utility.
-  The first argument must be an element, array of elements, or a callback which returns an element or array of elements.
-  Here is an example where the promise resolves because the element is removed:

```
const el = document.querySelector('div.getOuttaHere')

waitForElementToBeRemoved(document.querySelector('div.getOuttaHere')).then(() =>
  console.log('Element no longer in DOM'),
)

el.setAttribute('data-neat', true)
// other mutations are ignored...

el.parentElement.removeChild(el)
// logs 'Element no longer in DOM'
```

```
function waitForElementToBeRemoved<T>(
  callback: (() => T) | T,
  options?: {
    container?: HTMLElement
    timeout?: number
    interval?: number
    onTimeout?: (error: Error) => Error
    mutationObserverOptions?: MutationObserverInit
  },
): Promise<void>
```



WAITING FOR APPEARANCE



If you need to wait for an element to appear, the async wait utilities allow you to wait for an assertion to be satisfied before proceeding. The wait utilities retry until the query passes or times out. The async methods return a Promise, so you must always use `await` or `.then(done)` when calling them.

Using findBy Queries:

```
test('movie title appears', async () => {  
  // element is initially not present...  
  // wait for appearance and return the element  
  const movie = await findByText('the lion king')  
})
```

USING waitFor:

```
test('movie title appears', async () => {  
  // element is initially not present...  
  
  // wait for appearance inside an assertion  
  await waitFor(() => {  
    expect(getByText('the lion king')).toBeInTheDocument()  
  })  
})
```



WAITING FOR DISAPPEARANCE



The `waitForElementToBeRemoved` async helper function uses a callback to query for the element on each DOM mutation and resolves to `true` when the element is removed.

```
test('movie title no longer present in DOM', async () => {  
  // element is removed  
  await waitForElementToBeRemoved(() => queryByText('the mummy'))  
})
```



Using `MutationObserver` is more efficient than polling the DOM at regular intervals with `waitFor`.



The `waitFor` async helper function retries until the wrapped function stops throwing an error. This can be used to assert that an element disappears from the page.

```
test('movie title goes away', async () => {  
  // element is initially present...  
  // note use of queryBy instead of getBy to return null  
  // instead of throwing in the query itself  
  await waitFor(() => {  
    expect(queryByText('i, robot')).not.toBeInTheDocument()  
  })  
})
```



ASSERTING ELEMENTS ARE NOT PRESENT



The standard `getBy` methods throw an error when they can't find an element, so if you want to make an assertion that an element is not present in the DOM, you can use `queryBy` APIs instead:

```
const submitButton = screen.queryByText('submit')
expect(submitButton).toBeNull() // it doesn't exist
```



The `queryAll` APIs version return an array of matching nodes. The length of the array can be useful for assertions after elements are added or removed from the DOM.

```
const submitButtons = screen.queryAllByText('submit')
expect(submitButtons).toHaveLength(0) // expect no elements
```



`not.toBeInTheDocument`

The jest-dom utility library provides the `.toBeInTheDocument()` matcher, which can be used to assert that an element is in the body of the document, or not. This can be more meaningful than asserting a query result is null.

```
import '@testing-library/jest-dom'
// use 'queryBy' to avoid throwing an error with 'getBy'
const submitButton = screen.queryByText('submit')
expect(submitButton).not.toBeInTheDocument()
```



USING FAKE TIMERS



In some cases, when your code uses timers (setTimeout, setInterval, clearTimeout, clearInterval), your tests may become unpredictable, slow and flaky.



To solve these problems, or if you need to rely on specific timestamps in your code, most testing frameworks offer the option to replace the real timers in your tests with fake ones. This should be used sporadically and not on a regular basis since using it contains some overhead.



When using fake timers in your tests, all of the code inside your test uses fake timers.



The common pattern to setup fake timers is usually within the beforeEach, for example:

```
// Fake timers using Jest
beforeEach(() => {
  jest.useFakeTimers()
})
```

Don't forget to switch
back after your test!



```
// Running all pending timers and switching to real timers using Jest
afterEach(() => {
  jest.runOnlyPendingTimers()
  jest.useRealTimers()
})
```

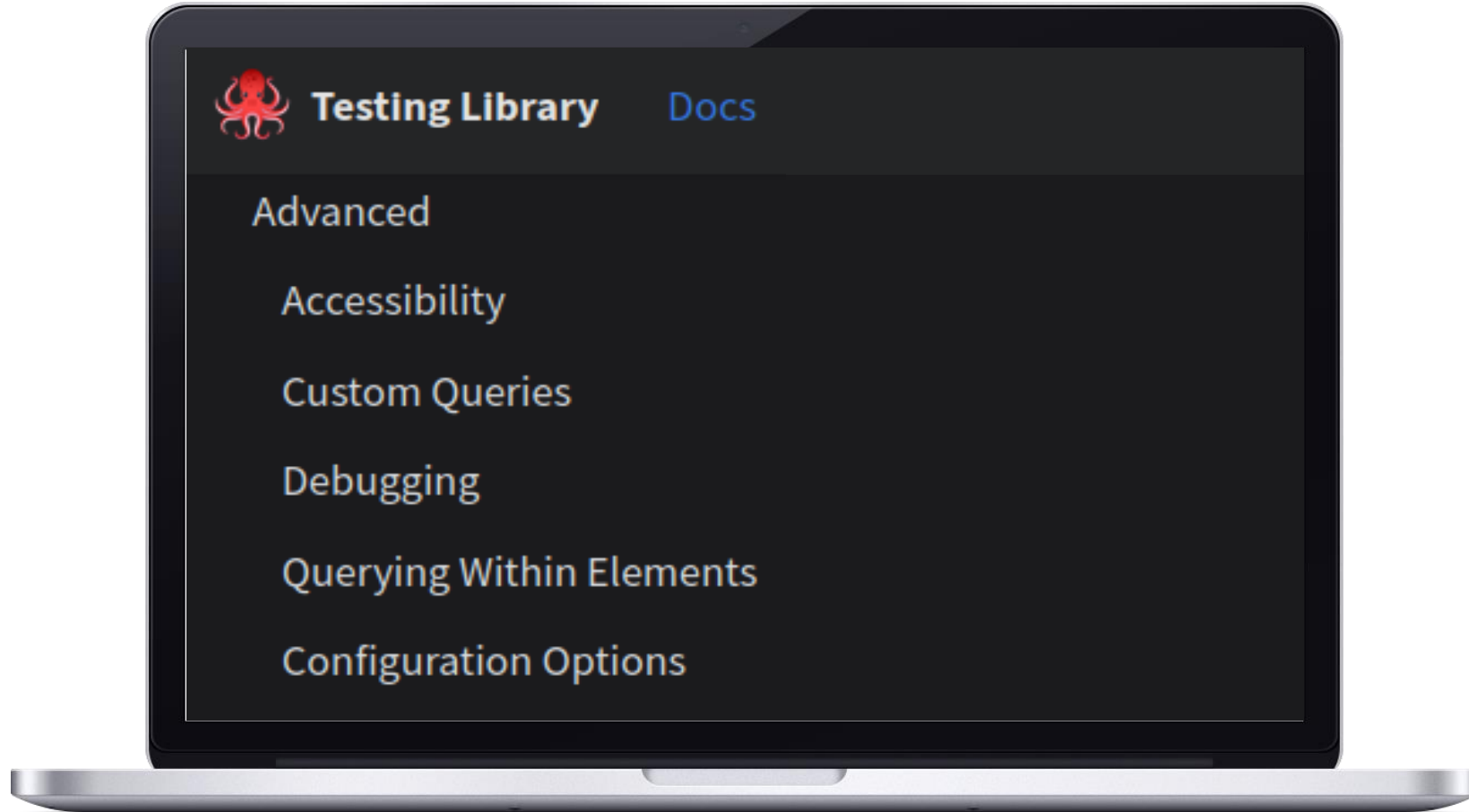


ADVANCED TOPICS



Check out the Testing Library docs for more information about advanced topics:

- [Accessibility](#)
- [Custom Queries](#)
- [Debugging](#)
- [Querying Within Elements](#)
- [Configuration Options](#)



A QUICK WORD ABOUT CALLING AN API...

There is a whole module on this tomorrow, but for today's code demo you need to understand how a call is made to an API:

```
useEffect(() => {  
  async function fetchData() {  
    const result = await fetch('https://jsonplaceholder.typicode.com/todos').then((response) =>  
      response.json()  
    );  
    // now are data is in result and we can parse it as we see fit here.  
  }  
  fetchData();  
}, []);
```

JEST-FETCH-MOCK



Making an HTTP request in your test is a bad idea:

- It's slow
- Unreliable
- And can annoy the API you are calling (excess dev traffic).



Jest-Fetch-Mock implements mocking functionality so our tests don't make real calls over HTTPS.

```
// src/utils/currency.js
async function convert(base, destination) {
  try {
    const result = await fetch(
      `https://api.exchangeratesapi.io/latest?base=${base}`
    );
    const data = await result.json();
    return data.rates[destination];
  } catch (e) {
    return null;
  }
}

export { convert };
```




AUTO MOCKING

STEPS:

01

Add the following to setupTests.js:

```
// src/setupTests.js
import fetchMock from "jest-fetch-mock";

fetchMock.enableMocks();
```

02

Write your test as shown below...

```
// src/utils/currency.test.js
import { convert } from "../currency";

beforeEach(() => {
  fetch.resetMocks();
});

it("finds exchange", async () => {
  fetch.mockResponseOnce(JSON.stringify({ rates: { CAD: 1.42 } }));

  const rate = await convert("USD", "CAD");

  expect(rate).toEqual(1.42);
  expect(fetch).toHaveBeenCalledTimes(1);
});
```



```
// src/utils/currency.test.js
it("returns null when exception", async () => {
  fetch.mockReject(() => Promise.reject("API is down"));

  const rate = await convert("USD", "CAD");

  expect(rate).toEqual(null);
  expect(fetch).toHaveBeenCalledWith(
    "https://api.exchangeratesapi.io/latest?base=USD"
  );
});
```

Above is an example of simulating an API that is down.



MOCK INTERFACES **NOT INTERNALS**



An even better approach is to mock the whole module that has the fetch rather than just the fetch:

```
// src/App.test.js
import React from "react";
import { render } from "@testing-library/react";
import App from "../App";

// Mock the currency module (which contains the convert function)
jest.mock("../utils/currency", () => {
  return {
    convert: jest.fn().mockImplementation(() => {
      return 1.42;
    }),
  };
});

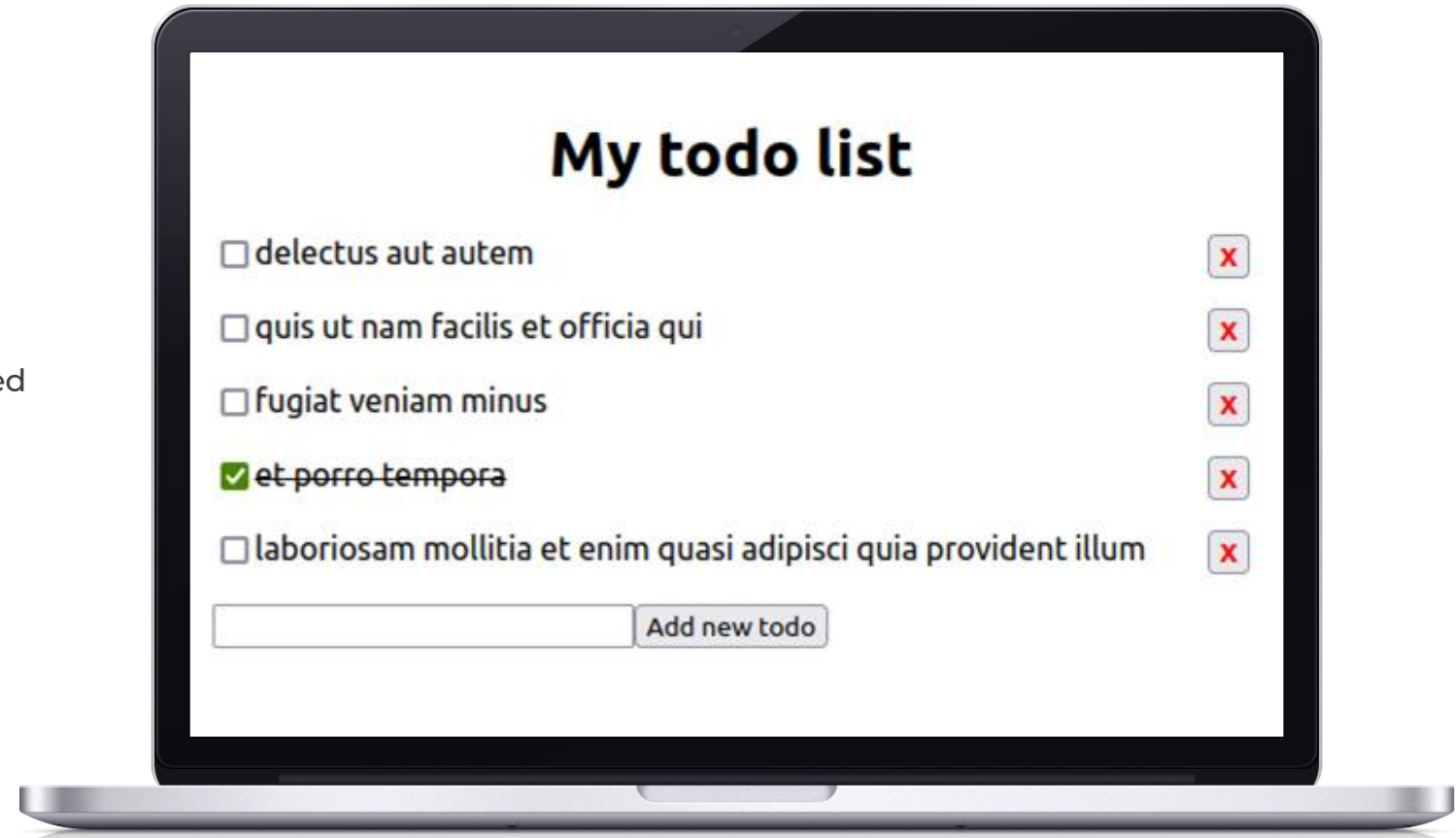
test("renders learn react link", async () => {
  const { findByText } = render(<App />);
  const element = await findByText(/USD to CAD: 1.42/i);
  expect(element).toBeInTheDocument();
});
```



HANDS ON DEMO



This demo shows will show you how to get started with TDD using Jest and React Testing Library.





[DAILY ASSIGNMENT]



WHAT YOU NEED TO DO...

Additional Resources

- [W3 School](#) •
- [MDN](#) •
- [React Testing Library](#) •

Reading

Chapter 7: Portals & Refs
Chapter 8: Handling Side Effects.



ASSIGNMENT

Coding

Extend the Demo that we went over in class. Start with the working code from class (<https://github.com/jeff-lent/demo-tdd>) and extend it for the following use cases

(USE TDD FOR ALL OF THESE!):

1. Improve the styling as you see fit.
2. Add a due date to each TodoItem.
3. If the due date is today or in the past, make the TodoItem red.
4. Sort the items by due date with the closest dates at the top.
5. Add a date finished to each line that is checked off.
6. Style the finish date green if it is on or before the due date or red if it is after the due date.
7. Add an edit button to the task so you can change the text or the due date.

Publish your code to github and submit the link via the Emeritus LMS!



**Don't forget to
submit your
assignment in the
Emeritus LMS!**