Cheatsheets / **Learn React Testing**

codecademy

React Testing Library

What is React Testing Library?

React Testing Library (RTL) is a library for testing React applications. React Testing Library focuses on testing components from the end-user's experience rather than testing the implementation and logic of the underlying React components.

Installing RTL

If you are using create-react-app to initialize your React project, the React Testing Library (RTL) will already be included.

To manually install RTL with `npm`, use the following command:

`npm install @testing-library/react --save-dev`

Though not required, the `--save-dev` flag will add this library as a development dependency rather than a production dependency. Once installed, RTL can be imported into your project.

```
// app.test.js
import {
  render,
  screen,
  waitFor
} from '@testing-library/react';
```

RTL Render

The React Testing Library (RTL) provides a `render()` method for virtually rendering React components in the testing environment. Once rendered in this way, the `screen.debug()` method can be used to view the virtually rendered DOM.

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

const Goodbye = () => {
  return <h1>Bye Everyone</h1>
};

test('should print the Goodbye component',
() => {
  render(<Goodbye/>);
  screen.debug();
});

// Output:
// <body>
//   <div>
```

```
//     <h1>
//       Bye Everyone
//     </h1>
//   </div>
// </body>
```

getByX Queries

The `screen` object from the React Testing Library (RTL) provides methods for querying the rendered elements of the DOM in order to make assertions about their text content, attributes, and more.

The `screen.getByX()` methods (such as `screen.getByRole()` and `screen.getByText()`) return the matching DOM node for a query, or throw an error if no element is found.

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

const Button = () => {
  return <button type="submit">Click Me</button>
};

// The button node can be extracted via its text content with screen.getByText()
test('Extract button node with getByText', () => {
  render(<Button/>);
  const button = screen.getByText('Click Me');
});

// The same button node can also be extracted with screen.getByRole()
test('Extract button node with getByRole', () => {
  render(<Button/>);
  const button = screen.getByRole('button');
});
```

User Event

The `@testing-library/user-event` library is an extension of `@testing-library` that provides tools for simulating user interactions with the DOM. The provided `userEvent` object contains methods that can be used to simulate clicks, typing, and much more.

The `user-event` documentation should be consulted to find the appropriate method for your needs.

```
import { render } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import '@testing-library/jest-dom';

const GreetingForm = () => {
  return(
```

```
    <form>
      <label role="textbox"
htmlFor="greeting">
        Greeting:
      </label>
      <input type="text" id="greeting" />
      <button
type="submit">Submit</button>
    </form>
  );
};

test('should show text content as Hello!',
() => {
  render(<GreetingForm />);
  const textbox =
screen.getByRole('textbox');
  const button =
screen.getByRole('button');

  // Simulate typing 'Hello!'
  userEvent.type(textbox, 'Hello!');
  // Simulate clicking button
  userEvent.click(button);

  // Assert textbox has text content
'Hello!'
  expect(textbox).toHaveValue('Hello!');
});
```

queryByX variant

When using the React Testing Library to determine if an element is NOT present in the rendered DOM, the `screen.queryByX` variants (such as `screen.queryByRole()`) should be used over their `screen.getByX` counterparts.

If the queried element cannot be found, the `screen.getByX` variants will throw an error causing the test to fail whereas the `screen.queryByX` will return `null`. The missing element can then be asserted to be `null`.

```
import { render, screen } from '@testing-
library/react';
import userEvent from '@testing-
library/user-event';
import '@testing-library/jest-dom';

const App = () => {
  // Removes header
  const handleClick = () => {

document.querySelector('h1').remove();
  };
```

```
  return (
    <div>
      <h1>Goodbye!</h1>
      <button onClick={handleClick}>Remove
Header</button>
    </div>
  )
};

test('Should show null', () => {
  // Render App
  render(<App />);
  // Extract button node
  const button =
screen.getByRole('button');
  // Simulate clicking button
  userEvent.click(button);
  // Attempt to extract the header node
  const header =
screen.queryByText('Goodbye!');
  // Assert null as we have removed the
header
  expect(header).toBeNull();
});
```

findByX Variant

When using the React Testing Library to query the rendered DOM for an element that will appear as a result of an asynchronous action, the `screen.findByX` variants (such as `screen.findByRole()`) should be used instead of the the `screen.getByX` and `screen.queryByX` variants.

The `await` keyword must be used when using the asynchronous `screen.findByX` variants and the callback function for the `test()` must be marked as `async`.

```
import { useState, useEffect } from
'react';
import { render, screen } from '@testing-
library/react';
import '@testing-library/jest-dom';

const Header = () => {
  const [text, setText] = useState('Hello
World!');

  // Changes header text after interval of
500ms
  useEffect(() => {
    setTimeout(() => {
      setText('Goodbye!');
    }, 500);
```

```
  });

  return <h1>{text}</h1>;
};

test('should show text content as
Goodbye', async () => {
  // Render App
  render(<Header />);
  // Asynchronously extract header with
new text
  const header = await
screen.findByText('Goodbye!');
  // Assert header to have text 'Goodbye!'
  expect(header).toBeInTheDocument();
});
```

Jest Dom

The `@testing-library/jest-dom` package contains DOM-specific matcher methods for testing front-end applications with Jest. Some common matcher methods include:

- `.toBeInTheDocument()`
- `.toBeVisible()`
- `.toHaveValue()`
- `.toHaveStyle()`

It is common for this library to be used alongside the React Testing Library. The `jest-dom` documentation should be consulted to find the appropriate matcher method for your needs.

```
import {render} from '@testing-
library/react';
import '@testing-library/jest-dom';

const Header = () => {
  return <h1 className='title'>I am a
header</h1>
};

test('should show the button as disabled',
() => {
  // render Button component
  render(<Header />);
  // Extract header
  const header =
screen.getByRole('heading');
  // Use jest-dom assertions
  expect(header).toBeInTheDocument();
  expect(header).toHaveTextContent('I am a
header');
  expect(header).toHaveClass('title');
});
```

waitFor

The `waitFor()` method in RTL is used to wait for asynchronous `expect()` assertions to pass. It is often used in combination with the `.queryByX()` methods to determine if a DOM element disappears asynchronously.

This function accepts two arguments, of which only one is required:

- a required callback function containing asynchronous testing logic
- an optional options object that can be used to configure how the callback is executed

Calling this function requires the use of the `async` keyword.

```
import React, { useEffect } from 'react';
import { waitFor, render, screen } from
'@testing-library/react';
import '@testing-library/jest-dom';
import userEvent from '@testing-
library/user-event';

const Header = () => {
  // Remove the heading after 250ms
  useEffect(() => {
    setTimeout(() => {

document.querySelector('h1').remove()
    }, 250);
  });

  return (
    <div>
      <h1>Hey Everybody</h1>
    </div>
  );
};

test('should remove header display', async
() => {
  // Render Header
  render(<Header/>)
  // Wait for the element to be removed
asynchronously
  await waitFor(() => {
    const heading =
screen.queryByText('Hey Everybody');
    expect(heading).toBeNull()
  })
});
```