

Agenda

- Types of Testing
- What is React Testing Library?
- How to create a test snapshot?
- Testing DOM elements
- Testing events
- Testing asynchronous actions
- Testing React Redux
- Testing React Context
- Testing React Router
- Testing HTTP Request

A Brief Introduction To Testing

Testing is a line-by-line review of how your code will execute. A suite of tests for an application comprises various bit of code to verify whether an application is executing successfully and without error. Testing also comes in handy when updates are made to code. After updating a piece of code, you can run a test to ensure that the update does not break functionality already in the application.

Why Test?

It's good to understand why we doing something before doing it. So, why test, and what is its purpose?

1. The first purpose of testing is to prevent regression. [Regression](#) is the reappearance of a bug that had previously been fixed. It makes a feature stop functioning as intended after a certain event occurs.
2. Testing ensures the functionality of complex components and modular applications.
3. Testing is required for the effective performance of a software application or product.

Testing makes an app more robust and less prone to error. It's a way to verify that your code does what you want it to do and that your app works as intended for your users.

Let's go over the types of testing and what they do.

UNIT TEST

In this type of test, individual units or components of the software are tested. A unit might be an individual function, method, procedure, module, or object. A unit test isolates a section of code and verifies its correctness, in order to validate that each unit of the software's code performs as expected.

In unit testing, individual procedures or functions are tested to guarantee that they are operating properly, and all components are tested individually. For instance, testing a function or whether a statement or loop in a program is functioning properly would fall under the scope of unit testing.

COMPONENT TEST

Component testing verifies the functionality of an individual part of an application. Tests are performed on each component in isolation from other components. Generally, React applications are made up of several components, so component testing deals with testing these components individually.

For example, consider a website that has different web pages with many components. Every component will have its own subcomponents. Testing each module without considering integration with other components is referred to as component testing.

Testing like this in React requires more sophisticated tools. So, we would need Jest and sometimes more sophisticated tools, like Enzyme, which we will discuss briefly later.

SNAPSHOT TEST

A snapshot test makes sure that the user interface (UI) of a web application does not change unexpectedly. It captures the code of a component at a moment in time, so that we can compare the component in one state with any other possible state it might take.

We will learn about snapshot testing in a later section.

Before continuing, let's learn some basics. Some key things are used a lot in this article, and you'll need to understand them.

- **it** or **test** You would pass a function to this method, and the test runner would execute that function as a block of tests.
- **describe** This optional method is for grouping any number of **it** or **test** statements.
- **expect** This is the condition that the test needs to pass. It compares the received parameter to the matcher. It also gives you access to a number of matchers that let you validate different things. You can read more about it [in the documentation](#).

- **mount** This method renders the full DOM, including the child components of the parent component, in which we are running the tests.
- **shallow** This renders only the individual components that we are testing. It does not render child components. This enables us to test components in isolation.

What is the React Testing Library?

The React Testing Library is a very light-weight package created by [Kent C. Dodds](#). It's a replacement for [Enzyme](#) and provides light utility functions on top of **react-dom** and **react-dom/test-utils**.

The React Testing Library is a DOM testing library, which means that instead of dealing with instances of rendered React components, it handles DOM elements and how they behave in front of real users.

It's a great library, it's (relatively) easy to start using, and it encourages good testing practices.

Note – you can also use it without Jest.

So, let's start using it in the next section. By the way, you don't need to install any packages, since **create-react-app** comes with the library and its dependencies.

How to create a test snapshot

As the name suggests, a snapshot allows us to save a given component's snapshot. It helps a lot when you update or do some refactoring, and want to get or compare the changes.

Now, let's take a snapshot of the **App.js** file.

- **App.test.js**

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

afterEach(cleanup);

it("should take a snapshot", () => {
  const { asFragment } = render(<App />);

  expect(asFragment(<App />)).toMatchSnapshot();
});
```

To take a snapshot, we first have to import **render** and **cleanup**. These two methods will be used a lot throughout this article.

render, as you might guess, helps to render a React component. And **cleanup** is passed as a parameter to **afterEach** just to clean up everything after each test to avoid memory leaks.

Next, we can render the App component with **render** and get back **asFragment** as a returned value from the method. And finally, make sure that the fragment of the App component matches the snapshot.

Now, to run the test, open your terminal and navigate to the root of the project and run the following command:

```
npm test
```

As a result, it will create a new folder **__snapshots__** and a file **App.test.js.snap** in the **src** which will look like this:

- **App.test.js.snap**

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`should take a snapshot 1`] = `
<DocumentFragment>
  <div
    class="App"
  >
    <h1>
      Testing Updated
    </h1>
  </div>
```

```
</DocumentFragment>
`;
```

And if you make another change in `App.js`, the test will fail, because the snapshot will no longer match the condition. To make it passes, just press `u` to update it. And you'll have the updated snapshot in `App.test.js.snap`.

Now, let's move on and start testing our elements.

Testing DOM elements

We first have to look at the `TestElements.js` file to test our DOM elements.

- `TestElements.js`

```
import React from "react";

const TestElements = () => {
  const [counter, setCounter] = React.useState(0);

  return (
    <>
      <h1 data-testid='counter'>{counter}</h1>
      <button data-testid='button-up' onClick={() => setCounter(counter + 1)}>
        { " " }
        Up
      </button>
      <button
        disabled
        data-testid='button-down'
        onClick={() => setCounter(counter - 1)}>
        Down
      </button>
    </>
  );
};

export default TestElements;
```

Here, the only thing you have to retain is `data-testid`. It will be used to select these elements from the test file. Now, let's write the unit test:

Test if the counter is equal to 0:

```
import "@testing-library/jest-dom/extend-expect";
import { cleanup, render } from "@testing-library/react";
import React from "react";
import TestElements from "../components/TestElements";

afterEach(cleanup);

it("should equal to 0", () => {
  const { getByTestId } = render(<TestElements />);
  expect(getByTestId("counter")).toHaveTextContent(0);
});
```

```
Test Suites: 1 passed, 1 total
Tests:      1 passed, 1 total
Snapshots:  1 obsolete, 0 total
Time:       4.497s, estimated 14s
Ran all test suites related to changed files.
```

Watch Usage

- > Press **a** to run all tests.
- > Press **f** to run only failed tests.
- > Press **u** to update failing snapshots.
- > 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.

```
█
```

Test if the buttons are enabled or disabled:

```
import "@testing-library/jest-dom/extend-expect";
import { cleanup, render } from "@testing-library/react";
import React from "react";
import TestElements from "../components/TestElements";

afterEach(cleanup);

it("should be enabled", () => {
  const { getByTestId } = render(<TestElements />);
  expect(getByTestId("button-up")).not.toHaveAttribute("disabled");
});

it("should be disabled", () => {
  const { getByTestId } = render(<TestElements />);
  expect(getByTestId("button-down")).toBeDisabled();
});
```

Here, as usual, we use `getByTestId` to select elements and check for the first test if the button has a `disabled` attribute. And for the second, if the button is disabled or not.

```
Test Suites: 1 passed, 1 total
Tests:      2 passed, 2 total
Snapshots:  1 obsolete, 0 total
Time:       5.652s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.█
```

Now, let's learn how to test an event in the next section.

Testing events

Before writing our unit tests, let's first check what the `TestEvents.js` looks like.

- `TestEvents.js`

```
import React from "react";

const TestEvents = () => {
  const [counter, setCounter] = React.useState(0);

  return (
    <>
      <h1 data-testid='counter'>{counter}</h1>
      <button data-testid='button-up' onClick={() => setCounter(counter + 1)}>
        Up
      </button>
      <button data-testid='button-down' onClick={() => setCounter(counter - 1)}>
        Down
      </button>
    </>
  );
};

export default TestEvents;
```



Now, let's write the tests.

Test if the counter increments and decrements correctly when we click on buttons:

```
import "@testing-library/jest-dom/extend-expect";
import { cleanup, fireEvent, render } from "@testing-library/react";
import React from "react";
import TestEvents from "../components/TestEvents";

afterEach(cleanup);

it("increments counter", () => {
  const { getByTestId } = render(<TestEvents />);

  fireEvent.click(getByTestId("button-up"));

  expect(getByTestId("counter")).toHaveTextContent("1");
});

it("decrements counter", () => {
  const { getByTestId } = render(<TestEvents />);

  fireEvent.click(getByTestId("button-down"));

  expect(getByTestId("counter")).toHaveTextContent("-1");
});
```



As you can see, these two tests are very similar except the expected text content.

The first test fires a click event with `fireEvent.click()` to check if the counter increments to 1 when the button is clicked.

And the second one checks if the counter decrements to -1 when the button is clicked.

`fireEvent` has several methods you can use to test events, so feel free to dive into the documentation to learn more.

```
Test Suites: 1 passed, 1 total
Tests:      2 passed, 2 total
Snapshots:  1 obsolete, 0 total
Time:       3.754s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Now that we know how to test events, let's move on and learn how to deal with asynchronous actions in the next section.

Testing asynchronous actions

An asynchronous action is something that can take time to complete. It can be an HTTP request, a timer, and so on.

Now, let's check the `TestAsync.js` file.

- `TestAsync.js`

```
import React from "react";

const TestAsync = () => {
  const [counter, setCounter] = React.useState(0);

  const delayCount = () => {
    setTimeout(() => {
      setCounter(counter + 1);
    }, 500);
  };

  return (
    <>
      <h1 data-testid='counter'>{counter}</h1>
      <button data-testid='button-up' onClick={delayCount}>
        Up
      </button>
      <button data-testid='button-down' onClick={() => setCounter(counter - 1)}>
        Down
      </button>
    </>
  );
};

export default TestAsync;
```

Here, we use `setTimeout()` to delay the incrementing event by 0.5s.

Test if the counter is incremented after 0.5s:

```
import "@testing-library/jest-dom/extend-expect";
import {
  cleanup,
  fireEvent,
  render,
  waitForElement,
} from "@testing-library/react";
import React from "react";
import TestAsync from "../components/TestAsync";

afterEach(cleanup);

it("increments counter after 0.5s", async () => {
  const { getByTestId, getByText } = render(<TestAsync />);
```

```

fireEvent.click(getByTestId("button-up"));

const counter = await waitForElement(() => getByText("1"));

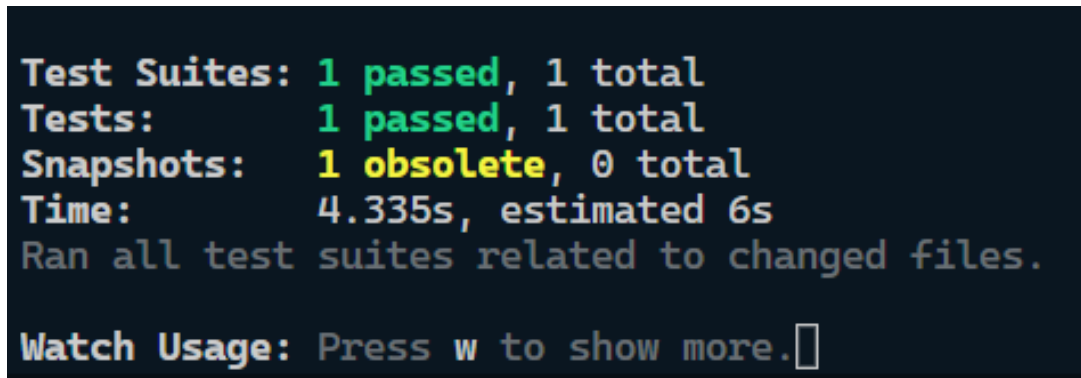
expect(counter).toHaveTextContent("1");
});

```

To test the incrementing event, we first have to use `async/await` to handle the action because, as I said earlier, it takes time to complete.

Next, we use a new helper method `getByText()`. This is similar to `getByTestId()`, except that `getByText()` selects the text content instead of `id` or `data-testid`.

Now, after clicking to the button, we wait for the counter to be incremented with `waitForElement(() => getByText('1'))`. And once the count is incremented to 1, we can now move to the condition and check if the counter is effectively equal to 1.



That being said, let's now move to more complex test cases.

Testing React Redux

In previous modules we have learnt all about react and redux. If you have not visited those we strongly recommend to read those modules to get a better understanding of redux. Otherwise, let's check what the `TestRedux.js` looks like.

- `TestRedux.js`

```

import React from "react";
import { useDispatch, useSelector } from "react-redux";

const TestRedux = () => {
  const state = useSelector(state => state.count);

  const dispatch = useDispatch();

  const increment = () => dispatch({ type: "INCREMENT" });
  const decrement = () => dispatch({ type: "DECREMENT" });

  return (
    <>
      <h1 data-testid='counter'>{state}</h1>
      <button data-testid='button-up' onClick={increment}>
        Up
      </button>
      <button data-testid='button-down' onClick={decrement}>
        Down
      </button>
    </>
  );
};

export default TestRedux;

```

And for the reducer:

- `store/reducer.js`

```
export const initialState = {
  count: 0,
}

export function reducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return {
        count: state.count + 1,
      }
    case 'DECREMENT':
      return {
        count: state.count - 1,
      }
    default:
      return state
  }
}
```



As you can see, there is nothing fancy – it's just a basic Counter Component handled by React Redux.

Now, let's write the unit tests.

Test if the initial state is equal to 0:

```
import "@testing-library/jest-dom/extend-expect";
import { cleanup, render } from "@testing-library/react";
import React from "react";
import { Provider } from "react-redux";
import { createStore } from "redux";
import TestRedux from "../components/TestRedux";
import { reducer } from "../store/reducer";

afterEach(cleanup);

const renderWithRedux = (
  component,
  { initialState, store = createStore(reducer, initialState) } = {},
) => {
  return {
    ...render(<Provider store={store}>{component}</Provider>),
    store,
  };
};

afterEach(cleanup);

it("checks initial state is equal to 0", () => {
  const { getByTestId } = renderWithRedux(<TestRedux />);
  expect(getByTestId("counter")).toHaveTextContent(0);
});
```



There are a couple of things we need to import to test React Redux. And here, we create our own helper function `renderWithRedux()` to render the component since it will be used several times.

`renderWithRedux()` receives as parameters the component to render, the initial state, and the store. If there is no store, it will create a new one, and it doesn't receive an initial state or a store, it returns an empty object.

Next, we use `render()` to render the component and pass the store to the Provider.

That being said, we can now pass the component `TestRedux` to `renderWithRedux()` to test if the counter is equal to `0`.


```
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 obsolete, 0 total
Time:        5.86s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Test if the counter increments and decrements correctly:

```
import "@testing-library/jest-dom/extend-expect";
import { cleanup, fireEvent, render } from "@testing-library/react";
import React from "react";
import { Provider } from "react-redux";
import { createStore } from "redux";
import TestRedux from "../components/TestRedux";
import { reducer } from "../store/reducer";

afterEach(cleanup)

const renderWithRedux = (
  component,
  { initialState, store = createStore(reducer, initialState) } = {},
) => {
  return {
    ...render(<Provider store={store}>{component}</Provider>),
    store,
  };
};

it("increments the counter through redux", () => {
  const { getByTestId } = renderWithRedux(<TestRedux />, {
    initialState: { count: 5 },
  });
  fireEvent.click(getByTestId("button-up"));
  expect(getByTestId("counter")).toHaveTextContent("6");
});

it("decrements the counter through redux", () => {
  const { getByTestId } = renderWithRedux(<TestRedux />, {
    initialState: { count: 100 },
  });
  fireEvent.click(getByTestId("button-down"));
  expect(getByTestId("counter")).toHaveTextContent("99");
});
```

To test the incrementing and decrementing events, we pass an initial state as a second argument to `renderWithRedux()`. Now, we can click on the buttons and test if the expected result matches the condition or not.

```
Test Suites: 1 passed, 1 total
Tests:      2 passed, 2 total
Snapshots:  1 obsolete, 0 total
Time:       4.577s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.[]
```

Now, let's move to the next section and introduce React Context.

Testing React Context

Let's check the `TextContext.js` file.

- `TextContext.js`

```
import React from "react";

export const CounterContext = React.createContext();

const CounterProvider = () => {
  const [counter, setCounter] = React.useState(0);
  const increment = () => setCounter(counter + 1);
  const decrement = () => setCounter(counter - 1);

  return (
    <CounterContext.Provider value={{ counter, increment, decrement }}>
      <Counter />
    </CounterContext.Provider>
  );
};

export const Counter = () => {
  const { counter, increment, decrement } = React.useContext(CounterContext);
  return (
    <>
      <h1 data-testid='counter'>{counter}</h1>
      <button data-testid='button-up' onClick={increment}>
        { " " }
        Up
      </button>
      <button data-testid='button-down' onClick={decrement}>
        Down
      </button>
    </>
  );
};

export default CounterProvider;
```

Now, the counter state is managed through React Context. Let's write the unit test to check if it behaves as expected.

Test if the initial state is equal to 0:

```
import "@testing-library/jest-dom/extend-expect";
import { cleanup, render } from "@testing-library/react";
import React from "react";
import CounterProvider, {
  Counter,
  CounterContext,
} from "../components/TestContext";
```

```

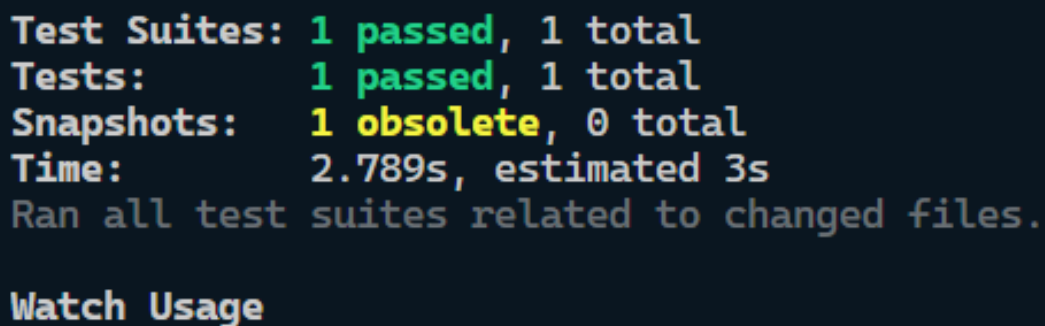
afterEach(cleanup);

const renderWithContext = component => {
  return {
    ...render(
      <CounterProvider value={CounterContext}>{component}</CounterProvider>,
    ),
  };
};

it("checks if initial state is equal to 0", () => {
  const { getByTestId } = renderWithContext(<Counter />);
  expect(getByTestId("counter")).toHaveTextContent(0);
});

```

As in the previous section with React Redux, here we use the same approach, by creating a helper function `renderWithContext()` to render the component. But this time, it receives only the component as a parameter. And to create a new context, we pass `CounterContext` to the Provider.



```

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 obsolete, 0 total
Time:        2.789s, estimated 3s
Ran all test suites related to changed files.

Watch Usage

```

Test if the counter increments and decrements correctly:

```

import "@testing-library/jest-dom/extend-expect";
import { cleanup, fireEvent, render } from "@testing-library/react";
import React from "react";
import CounterProvider, {
  Counter,
  CounterContext,
} from "../components/TestContext";

afterEach(cleanup);

const renderWithContext = component => {
  return {
    ...render(
      <CounterProvider value={CounterContext}>{component}</CounterProvider>,
    ),
  };
};

it("increments the counter", () => {
  const { getByTestId } = renderWithContext(<Counter />);

  fireEvent.click(getByTestId("button-up"));
  expect(getByTestId("counter")).toHaveTextContent("1");
});

it("decrements the counter", () => {
  const { getByTestId } = renderWithContext(<Counter />);

  fireEvent.click(getByTestId("button-down"));

```



```
    expect(getByTestId("counter")).toHaveTextContent("-1");
  });
```

As you can see, here we fire a click event to test if the counter increments correctly to 1 and decrements to -1.

```
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   1 obsolete, 0 total
Time:        3.386s, estimated 12s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

That being said, we can now move to the next section and introduce React Router.

Testing React Router

Let's check the `TestRouter.js` file.

- `TestRouter.js`

```
import React from "react";

import { Link, Route, Switch, useParams } from "react-router-dom";

const About = () => <h1>About page</h1>;

const Home = () => <h1>Home page</h1>;

const Contact = () => {
  const { name } = useParams();

  return <h1 data-testid='contact-name'>{name}</h1>;
};

const TestRouter = () => {
  const name = "John Doe";
  return (
    <>
      <nav data-testid='navbar'>
        <Link data-testid='home-link' to='/'>
          Home
        </Link>
        <Link data-testid='about-link' to='/about'>
          About
        </Link>
        <Link data-testid='contact-link' to={`/${contact}/${name}`}>
          Contact
        </Link>
      </nav>

      <Switch>
        <Route exact path='/' component={Home} />
        <Route path='/about' component={About} />
        <Route path='/about:name' component={Contact} />
      </Switch>
    </>
  );
};

export default TestRouter;
```

Here, we have some components to render when navigating the Home page.

Now, let's write the tests:

```
import "@testing-library/jest-dom/extend-expect";
import { cleanup, render } from "@testing-library/react";
import { createMemoryHistory } from "history";
import React from "react";
import { Router } from "react-router-dom";
import TestRouter from "../components/TestRouter";

afterEach(cleanup);

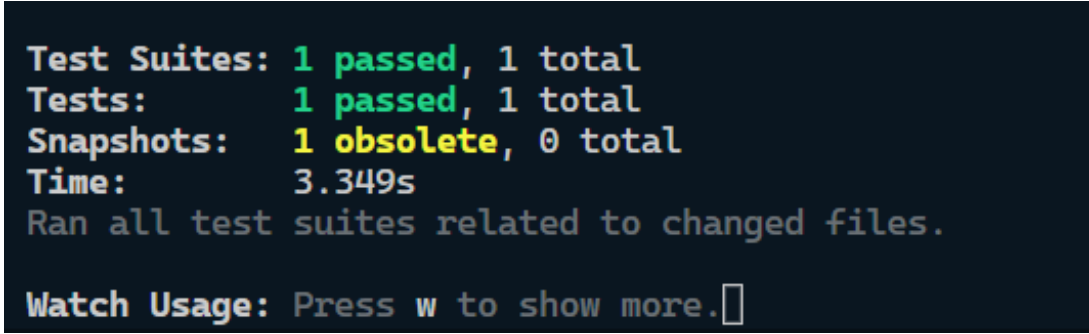
const renderWithRouter = component => {
  const history = createMemoryHistory();
  return {
    ...render(<Router history={history}>{component}</Router>),
  };
};

it("should render the home page", () => {
  const { container, getByTestId } = renderWithRouter(<TestRouter />);
  const navbar = getByTestId("navbar");
  const link = getByTestId("home-link");

  expect(container.innerHTML).toMatch("Home page");
  expect(navbar).toContainElement(link);
});
```

To test React Router, we have to first have a navigation history to start with. Therefore we use `createMemoryHistory()` to well as the name guess to create a navigation history.

Next, we use our helper function `renderWithRouter()` to render the component and pass `history` to the `Router` component. With that, we can now test if the page loaded at the start is the Home page or not. And if the navigation bar is loaded with the expected links.



```
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 obsolete, 0 total
Time:        3.349s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Test if it navigates to other pages with the parameters when we click on links:

```
import "@testing-library/jest-dom/extend-expect";
import { cleanup, render, fireEvent } from "@testing-library/react";
import { createMemoryHistory } from "history";
import React from "react";
import { Router } from "react-router-dom";
import TestRouter from "../components/TestRouter";

afterEach(cleanup);

const renderWithRouter = component => {
  const history = createMemoryHistory();
  return {
    ...render(<Router history={history}>{component}</Router>),
  };
};

it('should navigate to the about page', ()=> {
```

```

const { container, getByTestId } = renderWithRouter(<TestRouter />)

fireEvent.click(getByTestId('about-link'))

expect(container.innerHTML).toMatch('About page')
})

it('should navigate to the contact page with the params', () => {
  const { container, getByTestId } = renderWithRouter(<TestRouter />)

  fireEvent.click(getByTestId('contact-link'))

  expect(container.innerHTML).toMatch('John Doe')
})

```

Now, to check if the navigation works, we have to fire a click event on the navigation links.

For the first test, we check if the content is equal to the text in the About Page, and for the second, we test the routing params and check if it passes correctly.

```

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   1 obsolete, 0 total
Time:        3.896s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.

```

We can now move to the final section and learn how to test an Axios request.

Testing HTTP Request

As usual, let's first see what the `TextAxios.js` file looks like.

- `TextAxios.js`

```

import axios from "axios";
import React from "react";

const TestAxios = ({ url }) => {
  const [data, setData] = React.useState();

  const fetchData = async () => {
    const response = await axios.get(url);
    setData(response.data.greeting);
  };

  return (
    <>
      <button onClick={fetchData} data-testid='fetch-data'>
        Load Data
      </button>
      {data ? (
        <div data-testid='show-data'>{data}</div>
      ) : (
        <h1 data-testid='loading'>Loading...</h1>
      )}
    </>
  );
};

export default TestAxios;

```



As you can see here, we have a simple component that has a button to make a request. And if the data is not available, it will display a loading message. Now, let's write the tests.

Test if the data are fetched and displayed correctly:

```
import "@testing-library/jest-dom/extend-expect";
import {
  cleanup,
  fireEvent,
  render,
  waitForElement,
} from "@testing-library/react";
import axiosMock from "axios";
import React from "react";
import TestAxios from "../components/TestAxios";

afterEach(cleanup);

jest.mock("axios");

it("should display a loading text", () => {
  const { getByTestId } = render(<TestAxios />);

  expect(getByTestId("loading")).toHaveTextContent("Loading...");
});

it("should load and display the data", async () => {
  const url = "/greeting";
  const { getByTestId } = render(<TestAxios url={url} />);

  axiosMock.get.mockResolvedValueOnce({
    data: { greeting: "hello there" },
  });

  fireEvent.click(getByTestId("fetch-data"));

  const greetingData = await waitForElement(() => getByTestId("show-data"));

  expect(axiosMock.get).toHaveBeenCalledTimes(1);
  expect(axiosMock.get).toHaveBeenCalledWith(url);
  expect(greetingData).toHaveTextContent("hello there");
});
```

This test case is a bit different because we have to deal with an HTTP request. And to do that, we have to mock an axios request with the help of `jest.mock('axios')`.

Now, we can use `axiosMock` and apply a `get()` method to it. Finally we will use the Jest function `mockResolvedValueOnce()` to pass the mocked data as a parameter.

With that, now for the second test we can click to the button to fetch the data and use `async/await` to resolve it. And now we have to test 3 things:

1. If the HTTP request has been done correctly
2. If the HTTP request has been done with the `url`
3. If the data fetched matches the expectation.

And for the first test, we just check if the loading message is displayed when we have no data to show.

```
Test Suites: 1 passed, 1 total
Tests:      2 passed, 2 total
Snapshots:  1 obsolete, 0 total
Time:       3.882s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.[]
```

Conclusion

The React Testing Library is a great package for testing React Apps. It gives us access to `jest-dom` matchers we can use to test our components more efficiently and with good practices. Hopefully this module was useful, and it will help you build robust React apps in the future.

Thank You