BLOG

Testing apps with Jest and React Testing Library

March 1, 2021 · 7 min read

Testing is an essential practice in software engineering. It helps build robust a high-quality software and boosts the team's confidence in the code, making the application more flexible and prone to fewer errors when introducing or modifying features.

Highly efficient teams make testing a core practice in their everyday routines, and no feature is released before automated tests are in place. Some developers even write the tests before writing the features, following a process called TDD (Test Driven Development).

In this article, we'll test React applications with Jest and React Testing Library, a popular combination of a JavaScript testing framework and a React utility for testing components. It's also the official recommendation given on React's documentation.

What is testing?

Testing is the process of automating assertions between the results the code produces and what we expect the results to be.

When testing React applications, our assertions are defined by how the application renders and responds to user interactions.

Introduction to Jest and React Testing Library

What is Jest?

Jest is a JavaScript testing framework that allows developers to run tests on JavaScript and TypeScript code and integrates well with React.

It's a framework designed with simplicity in mind and offers a powerful and elegant API to build isolated tests, snapshot comparison, mocking, test coverage, and much more.

React Testing Library

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

Setting up your testing environment

Let's begin by installing the required libraries and setting up the project. The easiest way to get a React application up and running is by using Create React App, which comes with Jest already pre-installed.

I recommend that you follow the steps along with the tutorial by running the commands and writing the code yourself. However, if you prefer to follow the code on the final project result, you can clone the tutorial project from GitHub.

First, create a React app:

Now, install React Testing Library:

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

Finally, install additional libraries:

```
npm install axios
```

Building a React application for testing

Next, we build a minimal application that will display users from an API. Because we are only focusing on the frontend, we will use JSONPlaceHolder User API. This app is built exclusively for building tests.

Replace the contents of the file *App.js* with the following:

```
import { useEffect, useState } from 'react';
import axios from 'axios';
import { formatUserName } from './utils';
import './App.css';

function App() {
 const [users, setUsers] = useState([]);

 // Load the data from the server
 useEffect(() => {
   let mounted = true;

   const getUsers = async () => {
     const response = await
axios.get('https://jsonplaceholder.typicode.com/users');
     if (mounted) {
       setUsers(response.data);
     }
   };
```

Next, create a file called *utils.js* in *src* folder and write the following function:

```
export function formatUserName(username) {
 return '@' + username;
}
```

You can now run the app with this command:

```
npm start
```

After, you should see this screen.

Building a unit test

Unit tests test individual units or components of software in isolation. A unit could be a function, routine, method, module, or object, and the test objective is to determine if the unit outputs the expected results for a given input.

A test module comprises a series of methods provided by Jest to describe the tests' structure. We can use methods like **describe** or **test** as follows:

```
describe('my function or component', () => {
 test('does the following', () => {
   // Magic happens here
 });
});
```

The describe-block is the test suite, and the test-block (or simply test) is the test case. A test suite can have multiple test cases, and a test case doesn't have to be in a test suite, although it's common practice to do so.

Inside a test case, we write assertions (e.g., expect in Jest) which validate successful (green) or erroneous (red). Each test case can have multiple assertions.

Here is a somewhat trivial example of assertions that turns out successfully:

```
describe('true is truthy and false is falsy', () => {
 test('true is truthy', () => {
   expect(true).toBe(true);
 });

 test('false is falsy', () => {
```

Next, let's write our first test case targeting the function *formatUserName* from the *utils* module.

We need to create a new file: *utils.test.js*. Note that all test files use the pattern *{file}.test.js* where *{file}* is the name of the module file to test.

Our function in question takes a string as input and outputs the same string adding an `@` at its beginning. Our test function can assert that given a string, for example, "jc", the function will output "@jc".

Here is the code for the test file:

```
import { formatUserName } from "./utils";

describe('utils', () => {
 test('formatUserName adds @ at the beginning of the username', () => {
   expect(formatUserName('jc')).toBe('@jc');
 });
});
```

We usefully describe what the module is and what the test case is for, so that if they fail, we get a clear idea of what could have gone wrong.

Now that our first test is ready, we can run it and see what outputs. CRA makes it easy for us to run all tests by using a simple npm command.

```
npm run test
```

For now, let's focus on running a single test by using:

We do this because we have other tests already created by CRA that we need to ignore for now.

If everything went well, you should see a similar output to:

Notice that one test skipped (we wanted it that way) and that one test passed successfully.

But what would happen if something went wrong? Let's add a new test to the *utils* test suite to find out.

```
test('formatUserName does not add @ when it is already provided', () => {
   expect(formatUserName('@jc')).toBe('@jc');
 });
```

Now the situation is different; if the username already contains an `@` symbol at the beginning of the string, we expect that the function returns the username as provided, without adding a second symbol.

Let's run it:

As predicted, the test failed, and we receive information specifically about which *expect* call failed, the expected value, and the actual outcome. Because we've detected an issue with our original function, we can fix it.

```
export function formatUserName(username) {
 return username.startsWith('@') ? username : '@' + username;
}
```

We have made great progress so far. We wrote two test cases for our app, detected a bug thanks to writing those test cases, and we were able to fix it before releasing it.

Testing components with Jest

Testing components is not much different than testing functions. The idea and concepts are the same. The difference is in how we write the assertions.

We will test our *App* component by building a few test cases, and on each test case, we will introduce different things we can do to validate React components.

Our first test will be elemental. It will only validate the component renders.

Jump over to the file *App.test.js* (autogenerated by CRA) and replace its contents with:

```
import { render } from '@testing-library/react';
import App from './App';

describe('App component', () => {
 test('it renders', () => {
   render(<App />);
 });
})
```

Similar to before, we have a describe-block and a test-block, but this time, we use the *render* function mount and render an individual component in isolation. This test will only fail if there's a compilation error or an error in the function component that impedes its rendering. Though valid, it is not a complete test because it is not performing any assertions.

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

describe('App component', () => {
 test('it renders', () => {
   render(<App />);


   expect(screen.getByText('Users:')).toBeInTheDocument();
 });
})
```

Our new test is better. It validates that the component can render, but it also searches for an element present in the DOM with the text *"Users:"*, which is, in our case, it is, and, thus, the test passed successfully.

The object screen is essential in React Testing Library, as it provides helper methods to interact with the components and its elements.

Waiting for asynchronous operations

Next, we want to validate that the user list renders with items after the API completes. For that, we can write a test case as follows:

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

describe('App component', () => {
 test('it displays a list of users', async () => {
   render(<App />);


   expect(screen.getByTestId('user-list')).toBeInTheDocument();
 });
});
```

However, when we run the tests, it fails with the following message:

The reason it fails is simple: the async operation (fetch) is still pending when we evaluate the screen, so the “Loading users...” message is shown instead of the user list.

The solution is to wait:

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

describe('App component', () => {
 test('it displays a list of users', async () => {
   render(<App />);


   const userList = await waitFor(() => screen.getByTestId('user-list'));
   expect(userList).toBeInTheDocument();
 });
});
```

Mocking with React and Jest

Our next step is to validate how the component will react to the data gathered from the API. But how can we test the data if we are not sure what the API's response would be? The solution for this problem is mocking.

The purpose of mocking is to isolate the code being tested from external dependencies such as API calls. This is achieved by replacing dependencies with controlled objects that simulate those dependencies.

Mocking is a three-step process:

1. Import the dependencies

```
import axios from 'axios';
```

2. Mock the dependency

```
jest.mock('axios');
```

3. Fake the function outputs

```
axios.get.mockResolvedValue({ data: fakeUsers });
```

Let's see them now in action:

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

jest.mock('axios');

const fakeUsers = [{
  "id": 1,
  "name": "Test User 1",
  "username": "testuser1",
 }, {
  "id": 2,
  "name": "Test User 2",
  "username": "testuser2",
 }];

describe('App component', () => {

 test('it displays a row for each user', async () => {
   axios.get.mockResolvedValue({ data: fakeUsers });
```

One last note. Because we mock *axios*, each test case that uses the library will be returning *undefined* unless a mocked value is passed. So, to recap our full component test:

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

jest.mock('axios');

const fakeUsers = [{
  "id": 1,
  "name": "Test User 1",
  "username": "testuser1",
 }, {
  "id": 2,
  "name": "Test User 2",
  "username": "testuser2",
 }];

describe('App component', () => {
 test('it renders', async () => {
   axios.get.mockResolvedValue({ data: fakeUsers });
   render(<App />);
```

Let's run all tests and see the results:

Conclusion

Testing your React application is the key to producing high-quality apps, and, thanks to React, Jest, and the React Testing Library, it's easier than ever to test our components and applications.

All the code for the application and tests are available at GitHub.

Thanks for reading!

Debugging React applications can be difficult, especially when users experience issues that are hard to reproduce. If you're interested in monitoring and tracking Redux state, automatically surfacing JavaScript errors, and tracking slow network requests and component load time, try LogRocket.

LogRocket is like a DVR for web apps, recording literally everything that happens on your React app. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred. LogRocket also monitors your app's performance, reporting with metrics like client CPU load, client memory usage, and more.

The LogRocket Redux middleware package adds an extra layer of visibility into your user sessions. LogRocket logs all actions and state from your Redux stores.

Modernize how you debug your React apps — start monitoring for free.

Juan Cruz Martinez Follow

I'm an entrepreneur, developer, author, speaker, YouTuber, and doer of things.

#jest #react