Hyperiondev

TASK

React VI: Testing a React App

Visit our website

Introduction

PROPERLY TESTING YOUR CODE

To ensure the quality of your code, it is important to be able to test your application well. In this task, you will consider the types of formal tests that can be conducted. You will also learn to use a testing framework (Jest) to test your React apps.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our expert code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

TESTING

Testing has to do with determining whether the code works as it is expected to. There are various ways of testing code. Some of these include:

- **Manual testing**: Try to execute a piece of code manually and see if it does what you would expect it to do. You have no doubt done loads of manual testing by now. Console.log() statements are often used in manual testing to check whether variables contain the data you expect them to or whether a function is performing as you expect.

- **Documented manual testing**: As you can probably guess, this is manual testing that is documented. This involves having a documented plan for conducting tests.

- **End-to-end testing**: These are automated tests that simulate the user experience.

- **Unit tests**: Instead of testing the functionality of the system as a whole, this type of test focuses on testing one unit (for example, a single function, class, component etc) at a time.

- **Integration testing**: Once you have tested that individual units work separately, integration tests ensure that these units work together as expected.

- **Snapshot testing**: Snapshot tests are used to make sure that your UI doesn't change unexpectedly.

All the tests listed above have to do with *testing the functionality* of your code, i.e. does the code work as expected? There are other things that can, and should, be tested to *improve the quality* of code. These include:

- **Performance testing**: This has to do with testing the stability and responsiveness of code. It has to do with testing not just whether the code works, but how well it works. For example, how fast is it?

- **Usability testing**: Tests the way a user interacts with a system. Again, this type of testing is not to determine whether the code works, but whether users find the system easy and intuitive to use.

- **Security testing**: Involves doing tests that try to determine any potential security flaws within a system.

Once your app is deployed, you can use tools like **PassMarked** to help test the performance and security of your app.

It is considered good practice to create tests early on in the design/development process. Test-driven development (TDD) is an approach to software development where you are encouraged to write tests before you write the actual code. It is ubiquitous in industry to have a continuous integration server run tests on the entire codebase for *every commit pushed*. This gives people viewing the repo an idea of how stable certain branches are, among other things.

To assist with testing, there are testing frameworks that can be used. We will learn more about these frameworks in the next section. Once you know about some of the testing frameworks, we will then go on to learn how to use them to create automated unit tests.

TESTING FRAMEWORKS FOR REACT

The **official React documentation** lists a number of tools that can be used for testing React apps including Jest, Enzyme and React-unit. In this section, we will use **Jest** to test React. Jest is the tool that the creators of React, Facebook, use to test React.

Jest is automatically installed when you create a React app using the Create React App starter kit that we have been using throughout this level of the Bootcamp.

JEST SNAPSHOT TESTING

When testing front-end applications, it is common to use snapshot testing. Snapshot tests are used to determine whether the interface changes unexpectedly or not. The test works by taking a snapshot of what the interface looks like at a given stage. This snapshot is then stored and used in comparisons at later stages of the code review to see whether the interface has changed or not. See how to do snapshots tests with Jest here.

Let's write some tests for custom React links (`<Link/>`). To create a snapshot test for a component called 'Links.React' in a React app created using Create React App:

1. Add react-test-renderer for rendering snapshots. Jest is already installed by default by the Create React App starter kit.
   `npm add --dev react-test-renderer`

2. Create a directory called 'test' within the project structure of our React App. Within this directory, we create various test JavaScript files. E.g. **Links.React.test.js**

3. In the test file (e.g. **Links.React.test.js**), import React, the component you want to test and react-test-renderer. This presupposes the test subject (component) has already been created.

```
import React from 'react';
import Link from '../Link.react';
import renderer from 'react-test-renderer';
```

4. Write the test as shown in the code below. Notice that the test (within `test()`) creates a tree variable which stores the json that represents the DOM tree that is created when the component we are testing is rendered. In the example below, the `.create()` method creates a Link.React component where `page="http://www.facebook.com"` and `Facebook` are passed to the Link.React component as props.

   The first argument ('renders correctly') is simply the name given to the test.

   The code `expect(tree).toMatchSnapshot();` is used to see if the JSON representation of the DOM tree matched the snapshot. We expect it to match the snapshot. If it does, the test passes. If it doesn't, the test fails.

```
test('renders correctly', () => {
  const tree = renderer
    .create(<Link page="http://www.facebook.com">Facebook</Link>)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

   As you see from the code example:
   - Jest uses a `test()` module to create tests.
   - Jest uses an `expect()` object. The expect function is used every time you want to test a value. `expect()` is usually called along with a "matcher" function to assert something about a value. In the example above, we use the `.toMatchSnapshot()` matcher function of the `expect()` object. Other matchers include `.toBe()`, `.toBeEqual()`, `.not.to.be()`, `toBeNull()`. **See here** to see other matchers that you could use in your tests.

5. Run the test. From the command line interface, type `npm test` to run the test. This will automatically run all test files in your code.

JEST UNIT TESTING

The Jest `test()` module and `expect()` object can be used to write any number of unit tests. Consider the example based on Jest's documentation below. In this example, we create a test that tests a function (called `sum`) that adds two numbers together.

1. In the test directory, create a file called **sum.test.js**

2. Add the code shown below to **sum.test.js** and save the file.

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

As you see from the code example:

- Jest uses a `test()` module to create tests.
- Jest uses an `expect()` object. In the example above we use the `.toBe()` matcher function of the expect() object.
- Notice that we don't have to import React or react-test-renderer in this example because we are testing a normal JavaScript function and not a React component.

Compulsory Task 1

Follow these steps:

- Open the app you created in the **previous task**. Navigate to the project folder from your command line interface. Type `npm test` from here without making any changes. You should see something similar to the following output:

```
No tests found related to files changed since last commit.
Press `a` to run all tests, or run Jest with `--watchAll`.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › 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.
```

- Select 'a' to run all tests. You should notice something like the following output:

```
 PASS  src/App.test.js
  √ renders without crashing (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.069s, estimated 1s
Ran all test suites.

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

- Notice that by default when you create a React app using Create React App, a file called App.test.js is created. Examine this test file to see how the App.js file is tested.
- Now create the following tests for this project:
    - A snapshot test for at least one of your components.
    - Implement a `fetch()` function in your project - you are welcome to use a similar `fetch()` function as in your previous tasks.
    - A test to see whether the `fetch()` function that you wrote works correctly. See **here** for help with this.

- Run your tests by typing `npm test` in the command line. Take screenshots to show the results of each of the tests that you have created. Add these screenshots to the project folder of this app in a folder called **screenshots**.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCE

React.js. (2020). Getting Started – React. Retrieved 6 August 2020, from **https://reactjs.org/docs/getting-started.html**