TASK

React - Testing a React App

Visit our website

Introduction

WELCOME TO THE REACT - TESTING A REACT APP TASKS!

To ensure high-quality code, it is important to test your application thoroughly. In this task, you will consider the types of formal tests that you can conduct. You will also learn to use Jest, a testing framework, to test your React apps.

TESTING

Testing helps to ensure that your code works correctly and that components are bug-free. There are various ways of testing code. Some of these include:

- **Manual testing:** trying to execute a piece of code manually to see if it does what you expect. You have done manual testing by now. `Console.log()` statements are often used in manual testing to check whether variables contain the data that you expect or whether a function is performing as required.
- **Documented manual testing**: ensures that you have a documented plan for conducting tests.
- **End-to-end testing**: automated tests that simulate the user's experience.
- **Unit tests**: instead of testing the system's functionality as a whole, this type of test focuses on testing one unit or a single function, class, component, etc., at a time.
- **Integration testing**: once you are confident that individual units work separately, you can do integration testing to ensure these units work together.
- **Snapshot testing**: ensures that your UI does not change unexpectedly.

All the tests listed above check the functionality of your code. Other aspects of your code should also be tested to improve their quality. These include:

- **Performance testing**: tests the stability and responsiveness of the code. Performance testing checks how well your code is working and how fast it is.
- **Usability testing**: tests the way a user interacts with the system. It does not only determine the functionality of the code but also its simplicity and intuitiveness.

- **Security testing**: This involves tests determining any potential security flaws within the system.

It is considered good practice to create tests early on in the design or 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 the 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.

Testing frameworks assist with testing. We will learn more about these frameworks in the next section. Once you are familiar with some testing frameworks, we will learn how to use them to create automated unit tests.

TESTING FRAMEWORKS FOR REACT

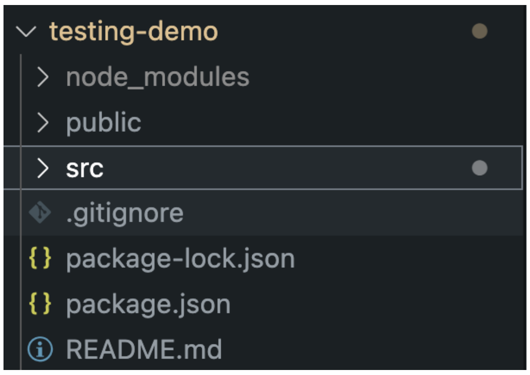
Two tools to test React apps are **Jest** and **React Testing Library**. Jest is the tool that the creators of React, Facebook, use to test React apps. Jest is automatically installed when you create a React app using the Create React App starter kit that we have been using. Next, we will use Jest to test a React app.

REACT PROJECT SETUP FOR TESTING

To create a new React app, we'll use create-react-app, a tool that sets up a new React project with a good default configuration:

```
npx create-react-app testing-demo
cd testing-demo
```

The following directory structure will be created:



```
testing-demo
  node_modules
  public
  src
  .gitignore
  package-lock.json
  package.json
  README.md
```

For this demo, let's create a simple counter app. Populate `src/App.js` with the following code:

```
import React, { useState } from "react";

function App() {
  const [count, setCount] = useState(0);
  const buttonStyle = {
    backgroundColor: "lightblue",
    padding: "10px 20px",
    border: "none",
    borderRadius: "4px",
    cursor: "pointer",
    fontSize: "1rem",
  };

  return (
    <div style={{ textAlign: "center", padding: "50px" }}>
      <h1>Simple Counter App</h1>
      <p>Count: {count}</p>
      <button
        style={{ ...buttonStyle, marginRight: "10px" }}
        onClick={() => setCount(count + 1)}
      >
        Increment
      </button>
      <button style={buttonStyle} onClick={() => setCount(count - 1)}>
        Decrement
      </button>
    </div>
  );
}

export default App;
```

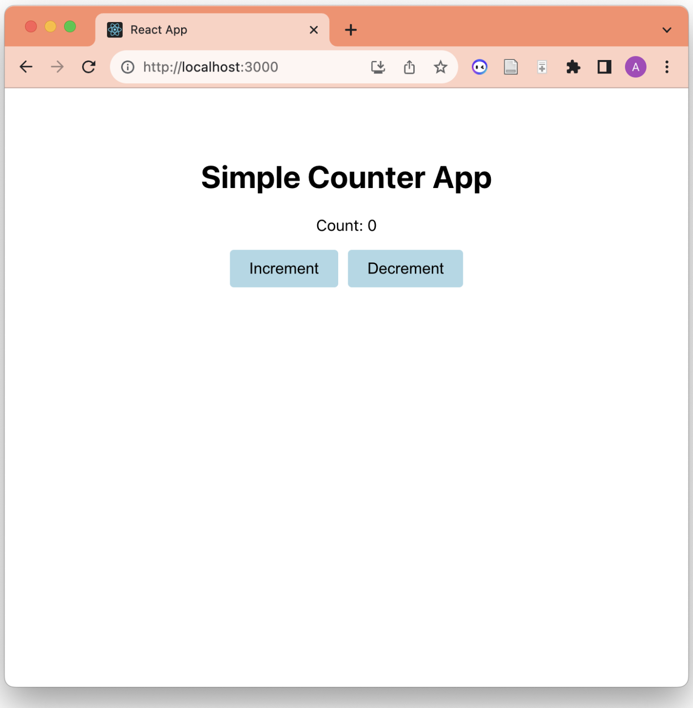
Then, run the server using the following command:

```
npm run start
```

You can now view `testing-demo` in the browser:

```
http://localhost:3000
```

The following will be the output:



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. Perform snapshot tests with Jest using the following steps:

Step 1: Adding Required Libraries

While `create-react-app` already includes Jest, we still need `react-test-renderer` to facilitate rendering snapshots:

```
npm add --dev react-test-renderer
```

Step 2: Setting Up the Test File

In your project, we'll have a test file named `App.test.js`. This is where we'll write our snapshot test for the `App` component.

Step 3: Writing the Test

Within `App.test.js`, import the required libraries and the component:

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

Then, write the snapshot test for the `App` component:

```
test('App renders correctly', () => {
  const tree = renderer.create(<App />).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Here is the breakdown of what is happening:

- We use the `test()` function from Jest to define our test.
- The `renderer.create(<App />).toJSON()` part is where `react-test-renderer` renders the `App` component and converts it into a JSON representation.
- `expect(tree).toMatchSnapshot()` is an assertion. It checks if the JSON representation matches the stored snapshot.

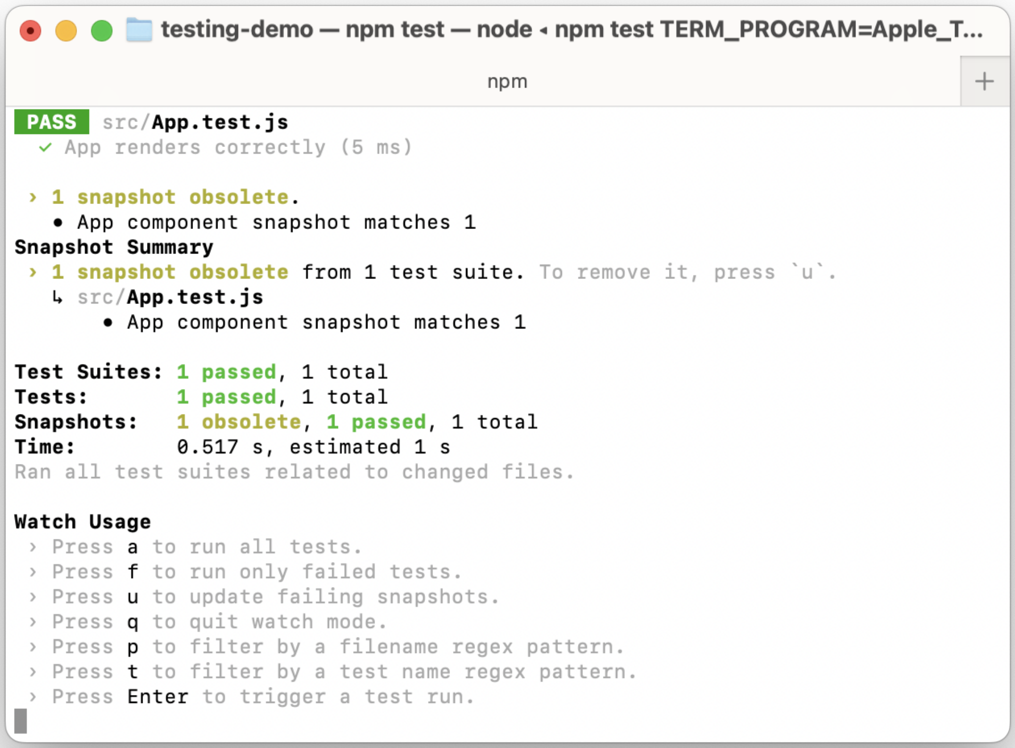
Step 4: Running the Test

To run the test, simply type:

```
npm test
```

This command will execute Jest, which will run all tests in your project.

The following should be the output:


testing-demo — npm test — node ◂ npm test TERM_PROGRAM=Apple_T...
npm

```
PASS  src/App.test.js
  ✓ App renders correctly (5 ms)

 › 1 snapshot obsolete.
   • App component snapshot matches 1
Snapshot Summary
 › 1 snapshot obsolete from 1 test suite. To remove it, press `u`.
   ↳ src/App.test.js
       • App component snapshot matches 1

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 obsolete, 1 passed, 1 total
Time:        0.517 s, estimated 1 s
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.
```

Take note:

The `expect()` function is at the heart of these assertions. It tells Jest what we're about to check and is paired with a "matcher" to define the expected result.

Let's consider this example: in `expect(someValue).toBe(5)`, `.toBe(5)` is the matcher, and it checks if someValue is indeed 5. There are many other matchers available, such as `` `.toBe()` ``, `` `.toEqual()` ``, etc., which provide different ways to compare and evaluate results.

Explore **other matchers** that you could use in your tests. Essentially, `expect()` sets the stage, and the matcher finalises the check.

JEST UNIT TESTING

The Jest `test()` module and `expect()` object can be used to write any number of unit tests. Below, we illustrate how to set up a unit test for our `` `App` `` component to ensure the "`Increment`" functionalities work as expected.

1. In the test directory, create a file called `Counter.test.js`.

2. Add the code shown below to `Counter.test.js` and save the file.

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

test('increments the count', () => {
  const { getByText } = render(<App />);
  const incrementButton = getByText('Increment');
  fireEvent.click(incrementButton);
  expect(getByText('Count: 1')).toBeInTheDocument();
});
```

   As you see from the code example:

   - Jest uses the **`test()`** function to establish test cases.
   - Jest employs the **`expect()`** object for assertions. In the above code, \``.toBeInTheDocument()`\` is a matcher method of the **`expect()`** object, which checks if an element is present in the document.
   - Since we are testing a React component, we import the necessary utilities from \``@testing-library/react`\`. This differs from testing regular JavaScript functions, where no React-specific imports are needed.

3. Run the test:

```
npm test
```

4. The following should be your output:

```
 PASS  src/App.test.js
  √ App renders correctly (15 ms)

 PASS  src/App.test.js
  √ App renders correctly (16 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
 PASS  src/App.test.js
 PASS  src/Counter.test.js

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   1 passed, 1 total
Time:        3.735 s
Ran all test suites.
```

Instructions

Please inspect the supporting code files provided with this task. This project involves testing a React application using unit tests and snapshot testing.

Explore tests in `src` to observe Jest usage, `expect` assertions, and snapshot testing.

Inspect the following test files:

- `API.test.js`: Unit test for API call using `fetch`. This file checks if the response status is `200`.
- `Child1-snapshot.test.js`: Snapshot test for a React component using `react-test-renderer`. It compares UI snapshots.
- `sum.test.js`: Tests a simple ‘sum’ function's correctness.

Compulsory Task

Follow these steps:

Snapshot Test:

- Create a new component named Greetings. This component should simply render a paragraph with the text "`Hello, World`!".
- Create a new test file named `Greetings.test.js` in the same directory as your components.
- Within this file, set up and write a snapshot test for the Greetings component.

Unit Test for "Decrement":

- In `src/Counter.test.js that we created earlier in the task`, write a test to ensure that the "**Decrement**" button decreases the count.

Run your tests by typing `npm test` in the command line. Take screenshots to show the results of each test 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

- Testing Overview (2020). Getting Started – React. Retrieved 10 August 2023, from **https://legacy.reactjs.org/docs/testing.html**
- Testing React Apps (2023) Retrieved 10 August 2023 from **https://jestjs.io/docs/tutorial-react**
- How To Test a React App with Jest and React Testing Library (2023) Retrieved 10 August 2023 **https://www.digitalocean.com/community/tutorials/how-to-test-a-react-app-with-jest-and-react-testing-library**