Unit Testing Framework

* Enzyme: a JavaScript Testing utility for React that makes it easier to assert, manipulate, and traverse your React Components’ output.
* Jest: Delightful JavaScript testing used by Facebook to test all JavaScript code including React applications.

$Yarn/npm test

Change its contents to look like this:

// src/App.js

import React, { Component } from 'react';

class App extends Component {

constructor() {

super();

this.state = {

count: 0,

}

}

makeIncrementer = amount => () =>

this.setState(prevState => ({

count: prevState.count + amount,

}));

increment = this.makeIncrementer(1);

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button className="increment" onClick={this.increment}>Increment count</button>

</div>

)

}

}

export default App;

Our React app has some initial state count which is set to zero, and a button that, once clicked, increments this count state through the increment function which simply adds 1 to the value of count and updates the application state

If you look at the src/App.test.js, you will see that a test has already been written for us. It tests that the App component can render without crashing.

// src/App.test.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

it('renders without crashing', () => {

const div = document.createElement('div');

ReactDOM.render(<App />, div);

ReactDOM.unmountComponentAtNode(div);

});

Let’s add a dummy test below this one in App.test.js:

// src/App.test.js

...

describe('Addition', () => {

it('knows that 2 and 2 make 4', () => {

expect(2 + 2).toBe(4);

});

});

A describe() function groups related tests together inside one test suite. It takes a name parameter, which should describe the component you’re testing, and a callback function where individual tests are defined with it.

You might see individual tests with test in some projects. To be sure, it and test are one and the same thing. it is only an alias for test.

// src/App.test.js

describe('Addition', () => {

it('knows that 2 and 2 make 4', () => {

expect(2 + 2).toBe(4);

});

// is equaivalent to

test('knows that 2 and 2 make 4', () => {

expect(2 + 2).toBe(4);

});

});

Writing your first test

Before we begin writing our own tests, we need to add a few packages to our application for it to be able to test via Enzyme’s shallow renderer:

----

yarn add enzyme enzyme-adapter-react-16 --dev

Enzyme is built to support different versions of React. In this tutorial, I’m using the latest stable version of React which is 16.4.2, but you might be working with an older version of React, say React 15.x. So you also have to install an Adapter that corresponds to the version of React that you are using.

You also need to create a setupTests.js file within your src folder that tells Jest and Enzyme what Adapters you will be making use of. create-react-app has been configured to run this file automatically before any of our tests, so that Enzyme is set up correctly.

// src/setupTests.js

import { configure } from 'enzyme';

import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });

Now, can can begin writing tests for our application. Jump to src/App.test.js and change its contents to look like this:

// src/App.test.js

import React from 'react';

import { shallow } from 'enzyme';

import App from './App';

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

it('starts with a count of 0', () => {

const wrapper = shallow(<App />);

const text = wrapper.find('p').text();

expect(text).toEqual('Count: 0');

});

});

We’re taking advantage of Enzyme’s shallow rendering to test our app’s initial state. A shallow render is a simulated render of a component tree that does not require a DOM. It renders only one level of components deep, and enables the inspection of the component’s contents as well as the simulation of user interaction.

In the above snippet, the shallow render of our App component is stored in the wrapper variable. We then grab the text inside the p tag within the component’s output and check if the text is the same was what we passed into the toEqual matcher function

Testing user interaction

Let’s go ahead and write a new test that simulates a click on the button and confirms that the count output is incremented by 1.

Add this below the first test:

// src/App.test.js

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

...

it('increments count by 1 when the increment button is clicked', () => {

const wrapper = shallow(<App />);

const incrementBtn = wrapper.find('button.increment');

incrementBtn.simulate('click');

const text = wrapper.find('p').text();

expect(text).toEqual('Count: 1');

});

});

The simulate() function on the button variable can simulate a number of DOM events on an element. Here we are simulating the click event on the button. We’ve also set up an expectation that the count should be equal to now.

If you check the terminal output, you should observe that the test passes as expected.

Let’s go ahead and do something a bit different. We’ll add a test for some functionality that doesn’t exist yet, then go ahead and write the code to make the test pass. This methodology of writing tests before the code is known as Test Driven Development (TDD).

Create another test within the describe() function that looks like this:

// src/App.test.js

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

...

it('decrements count by 1 when the decrement button is clicked', () => {

const wrapper = shallow(<App />);

const decrementBtn = wrapper.find('button.decrement');

decrementBtn.simulate('click');

const text = wrapper.find('p').text();

expect(text).toEqual('Count: -1');

});

});

Now you should have a failing test in the terminal: Method "``simulate``" is only meant to be run on a single node. 0 found instead. If you’re not used to Enzyme, you might find the error message rather cryptic. What it means is that the simulate() method was called on an element that doesn’t exist yet.

Let’s now go into the React component and write the code that will fix this test:

// src/App.js

import React, { Component } from 'react';

class App extends Component {

constructor() {

super();

this.state = {

count: 0,

}

}

makeIncrementer = amount => () =>

this.setState(prevState => ({

count: prevState.count + amount,

}));

increment = this.makeIncrementer(1);

decrement = this.makeIncrementer(-1);

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button className="increment" onClick={this.increment}>Increment count</button>

<button className="decrement" onClick={this.decrement}>Decrement count</button>

</div>

)

}

}

export default App;

We added a decrement button after the increment button in the render() function and a decrement() function that decreases the value of count by 1 and updates the application state. At this point, all three tests should pass successfully.

Testing React components with snapshots

Snapshot testing helps you check that the rendered output of a component is correct at all times. When you run a snapshot test, Jest renders the React component being tested and stores the output in a JSON file.

On further test runs, Jest will check that the output of the component has not deviated from what it saved previously. If you change the a component’s output, Jest will notify you and you can either update the snapshot to the latest version or fix the component so that it matches the snapshot again. This method of testing components helps you avoid accidental changes to your components because Jest will always notify you when a difference is detected.

To use Jest’s snapshot feature, we need an additional package, react-test-renderer, which can be installed through yarn:

yarn add react-test-renderer --dev

Then import it at the top of App.test.js:

// src/App.test.js

import renderer from 'react-test-renderer';

...

Next, create a test below all the previously created ones:

// src/App.test.js

it('matches the snapshot', () => {

const tree = renderer.create(<App />).toJSON();

expect(tree).toMatchSnapshot();

});

The first time this test is run, there is no snapshot for this component so Jest creates it. You can inspect the contents of the snapshots inside the src/\_\_snapshots\_\_ directory.

Open up App.test.js.snap:

// src/\_\_snapshots\_\_/App.test.js.snap

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

exports[`App component matches the snapshot 1`] = `

<div>

<p>

Count:

0

</p>

<button

className="increment"

onClick={[Function]}

>

Increment count

</button>

<button

className="decrement"

onClick={[Function]}

>

Decrement count

</button>

</div>

`;

You can see that the rendered output of the App component is saved in this file. The next time this test is run, Jest will confirm that the outputs are the same. We can demonstrate this concept by changing the rendered output of App slightly.

Let’s assume that we accidentally changed the text of the increment button in App.js to Increment count2. The test should fail.

Since we didn’t intend to make this change, we can simply undo the change and the test will pass again.

Let’s make an actual change to the App component. Change the text of the increment button to Increment and the decrement button to Decrement. Your render function should look like this:

// src/App.js

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button className="increment" onClick={this.increment}>Increment</button>

<button className="decrement" onClick={this.decrement}>Decrement</button>

</div>

)

}

The tests should fail again. But since we actually intended to make this change, we need to update the snapshot to this latest version. We can do so by pressing u in the terminal window where the tests are being run. Everything should be green again!