Testing with JavaScript

# Manual Testing

We first need to consider how we have been manually testing our Application so far, serving the application and then interacting with various parts of it in an effort to test different bits of functionality.

There are many downsides to this approach including:

* Time-Consuming
* Prone to Human Error
* Irregular results

We need a mechanism to streamline the process, we need to automate our tests. Think about the time you have spent already reloading pages, clicking buttons and typing data manually!

# Automated Testing

By way of demo we will first run the test that is created for us when we ran *npx create-react-app*.

In your project, run *npm test,* behind the scenes this runs a script outlined in the *package.json*.

This should run the test that was created with create-react-app, your command prompt should show that 1 test has passed.

# How did *npm test* know where to find my test?

It used the name of the file, *App.test.js,* when we ran the script it first looks for a folder called *\_\_tests\_\_,* which we haven’t made yet, and then look for files called *<something>.test.js.* It found a file provided by create-react-app called *App.test.js*.

Locate the *App.test.js* file in your project, this was the test that we just executed. Upon inspection this test is very simple, it is a Smoke Test and merely checks that *App.js* renders something without failing.

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);

});

*it –* This is an alias for the keyword *Test*, basically they mean the same thing and can, and will, be used interchangeably. It indicates that we are creating a test.

*‘renders without crashing’ –* This is how we will refer to the test, the ‘name’ of the test, make this parameter descriptive and unique so that you can easily locate a failing test.

We will do one more thing with this file, currently it is in the *src* folder amidst our other React Components, which is going to quickly fill up. We want to create a new folder here called \_\_*tests*\_\_ this is conventional and is where the tool we are using to run our tests will first look for our tests. It also helps keep or Project structure neat and organised.

# Unit Testing

We will be using Jest and enzyme to perform basic ‘shallow’ unit testing. Firstly, we need to use npm to ensure that we have the correct modules that we need to write and run our tests. Run the following command in the root of your project.

npm install --save enzyme enzyme-adapter-react-16 react-test-renderer enzyme-to-json

We next need to also create a file in our src folder called setupTests.js, it will be responsible for configuring the Enzyme tool we are using give this file the following content.

import Enzyme from 'enzyme'

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

Enzyme.configure({ adapter: new Adapter() })

At a high-level Enzyme allows us to render simple components and check that they contain some of the elements we expect. For instance, if one of my components contains a <p> tag with some specific text in, we can check to see that the <p> element exist with the correct text.

However, Enzyme can do a lot more, take a look at this helpful cheat sheet.

<https://devhints.io/enzyme>

# Smoke Test

Using Enzyme we should write our first test, in it we should first check that we have set up our testing environment correctly, before we start testing our component we will try a simple smoke test to check our imports.

import React from 'react';

import { shallow } from 'enzyme';

import App from '../App';

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

shallow(<App />);

});

This test performs the same purpose as the previous test, except that it makes use of the new enzyme module we have installed.

*shallow –* This new keyword, from the *enzyme* library, allows us to create a ‘shallow’ rendering of a component. This will render that component only, not any child components, therefore it is a ‘shallow’ rendering.

# Content Test

To ‘Unit Test’ our component we can use enzyme to check that the component we are rendering *contains* specific content.

import React from 'react';

import { shallow } from 'enzyme';

import App from '../App';

import RoutingExample from '../Router.js'

it('renders routingExample comp', () => {

const wrapper = shallow(<App />);

const content = <RoutingExample />;

expect(wrapper.contains(content)).toEqual(true);

});

*Wrapper –* the wrapper constant holds the returned value from shallow(<App />), this is of the type ShallowWrapper. Conventionally you will see it called *wrapper* or *wrap,* we can then interact with this constant in various ways.

*Content* – the content constant holds the React element that we expect to be a part of the React element that we are creating a shallow rendering of. It is important to note that the Shallow Rendering can contain many other elements, of which *content* is just one.

*Expect –* This is where we check a condition, when testing we are always checking actual values against expected values.

This test checks that the App component contains the RoutingExample component, however it does not render the RoutingExample component. Therefore, realistically we are only checking the content of 1 unit at a time, the App component in this case.

# Snapshot Testing

## What is it?

With Snapshot testing we create a snapshot of what our component will look like when it is rendered, this is saved as a ‘snap’ file, which is then used by Jest to check for changes. As this method renders child components as well it is a form of integration testing.

However, we cannot interact with these child components, therefore snapshots are really only used to check original state of a rendered component.

Subsequent tests compare the component against the snapshot to see if the way it is rendered has changed. A change can be caused by various reasons:

* A bug, which you will then have to fix.
* A change made to the original component that hasn’t been updated in the snapshot.

Snapshot will check the content of nested Components as well, acting as integration testing.

## Example

Label.js

import React, { Component } from 'react';

import LabelChild from './LabelChild';

class Label extends Component {

constructor(props) {

super(props);

this.state = {

data: 'Initial data'

}

}

    handleClick = () => {

        this.setState({

data: "The button changes the state"

});

    }

render() {

return (

<div>

        <LabelChild title = {this.props.title}/>

<p>

         {this.state.data}

         </p>

         <p>

         {this.props.title}

         </p>

         <button name="clickme" onClick={this.handleClick}></button>

</div>

);

}

}

export default Label;

LabelChild.js

import React, { Component } from 'react';

class LabelChild extends Component {

constructor(props) {

super(props);

this.state = {

data: 'Initial data'

}

}

render() {

return (

<div className='in-LabelChild'>

<p>

         {this.state.data}

         </p>

         <p>

         {this.props.title}

         </p>

</div>

);

}

}

export default LabelChild;

Here we have to pretty simple components, LabelChild which uses its own internal state to render a <p> tag and the props it is passed to render another. We also have Label, which makes use of a LabelChild component which it passes props too and 2 <p> tags which it uses in the same way as LabelChild. Label also has a button, that when clicked calls the handleClick() function which changes the state of the Label component.

Now for the test, notice the name of the file.

Label.test.js

import React from "react";

import { shallow } from "enzyme";

import Label from "../Label";

it("render a label", () => {

const wrapper = shallow(<Label />);

expect(wrapper).toMatchSnapshot();

});

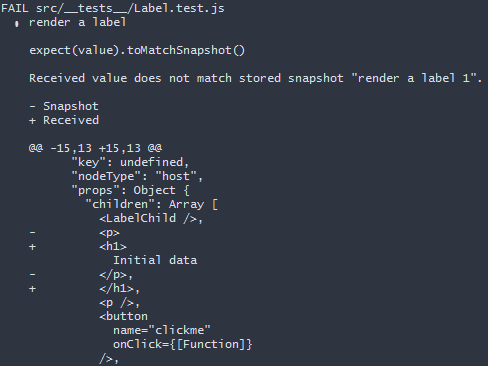
*Expect –* This is where we check a condition, when testing we are always checking actual values against expected values.

*toMatchSnapshot() –* This function will create a ‘snap’, the first time the test is ran it will create the snap and have nothing to compare it too. However as stated above subsequent runs of this test will compare a new ‘snap’ against the original (which you can update). This process means that if *Label* does not render correctly as per the original snap, the test will **fail**.

## Has it worked?

To check that it has been successful we can now **change** the Label.js file so that it renders something different, for instance by changing a <p> tag to a <h1> tag. Remember this will cause our test to render the Label component differently, and then check this rendering against the original snap.

Lets’ try it, change one of the <p> tags in Label.js to be a <h1>, and then run npm test again.



In your command line you will see information similar to this, we see a message informing us that the new value from the test did not match the stored snapshot. It will also return some information that shows us differences between the snapshot and the value it was sent.

If you still had a console open using npm test, you may have noticed that just by saving your changes to Label.js the test was run again, this is one of the benefits of using Jest, it provides this live testing environment and will automatically rerun the test when it detects that a change has been made to the file under test. You can test this out now by keeping your console open and reverting Label.js, save your changes and the test should rerun automatically.

On a final note, as this is integration testing, if we change an element in the LabelChild component the test will fail as the original snapshot rendered that component as well.

# Snapshot Testing with Props

A next logical progression is to test if our system correctly handles the props that we pass into a component we are rendering. To do this we will be using enzyme, we will also be using the wrapper constant you have seen created to expect specific values.

//Testing rendering a Prop

it("render a prop", () => {

const wrapper = shallow(<Label title="A Title" />);

expect(wrapper.instance().props.title).toEqual("A Title");

expect(wrapper).toMatchSnapshot();

});

Here we have used shallow to render a Label component whilst we have passed some props to it (remember that the Label component used props.title as one of the values for a <p> tag).

We are then manipulating our wrapper constant to:

* First retrieve the React Component that it renders
* Then access the props of this component
* Then access the title prop explicitly
* And then finally check that it is equal to the value “A Title”

We cap this test off by creating another snap shot, even though this test can appear in the same file as our previous test, they will use different original snapshots.

# Interacting with a Button

So far we have tested that a component renders correctly, we have also checked that it renders correctly based on the provided props, however we have not checked what happens when certain events are simulated.

For example, in the *Label*.js sample component above there is a button, when this button is clicked it will change the state of the component, which one of the <p> tags relies upon.

We will look at how we write a test in order to check that clicking the button performs the expected functionality.

it("clicks a button", () => {

const wrapper = shallow(<Label title="A Title" />);

expect(wrapper).toMatchSnapshot();

wrapper.find('[name="clickme"]').simulate("click");

expect(wrapper).toMatchSnapshot();

});

We first create a shallow rendering of the Label component, checking with a snapshot that this component has rendered correctly.

*Find –* We pass the name of the component that we want to find into our *find* function, this will find the component for us. You can read more about how *find()* works and what you can pass to it in the Enzyme documentation.

*Simulate –* Using this function and passing in the event we want to simulate, we can effectively ‘send a click’ to the button component, this will trigger the function associated with this event, changing the state in our case. You can read more about how *simulate()* works and what you can pass to it in the Enzyme documentation.

We finish off this test by creating and checking another snapshot, this time we will expect the text in one of the <p> tags to have changed as the state of the Component has changed.

To see if this has worked, modify the function in Label.js so that it changes the state to something different and rerun your test.

# Interacting with a form

We will want to use forms on one form or another quite often in our web applications, however testing them can be a little more tricky.

Below is a sample Component that we will be looking at to demonstrate testing a form.

FormTest.js

export default class FormTest extends Component {

  constructor(props) {

super(props);

this.state = {

first: '',

     last: ''

}

}

  handleSubmit = (event) => {

    event.preventDefault();

    this.setState({

first: this.firstNode.value,

     last: this.lastNode.value

});

  }

render() {

return (

<div>

   <h1>FormTest.js</h1>

<form name ="form" onSubmit={this.handleSubmit}>

   <p>First Name</p>

     <input name="first" ref={node => (this.firstNode = node)}/>

     <p>Last Name</p>

     <input name="last" ref={node => (this.lastNode = node)}/>

     <br />

     <button type="submit">Submit</ button>

     </ form>

     <p name="suppliedFirst">{this.state.first}</p>

     <p name="suppliedLast">{this.state.last}</p>

</div>

)

}

}

This component uses refs in order to get the values from the form for which to change the state, you will also notice that in the *handleSubmit()* function we are calling *event*.*preventDefault*(), this is so that the default functionality of submit is prevented.

FormTest.test.js

import React from "react";

import { mount } from "enzyme";

import FormTest from "../FormTest";

it("submit a form", () => {

const wrapper = mount(<FormTest />);

wrapper.find('[name="first"]').instance().value = "Mark";

wrapper.find('[name="last"]').instance().value = "Wright";

wrapper.find("form").simulate("submit");

expect(wrapper.contains(<p name="suppliedFirst">Mark</p>)).toBeTruthy();

expect(wrapper.contains(<p name="suppliedLast">Wright</p>)).toBeTruthy();

});

*Mount –* The biggest difference between this test and previous ones is the use of *mount* instead of *shallow,* if *shallow* created a simple shallow rendering of a component then *mount* creates a more in depth rendering. This means that *wrapper* can hold state of the component it represents.

*Wrapper.find –* We use this function twice to set the input fields to be a specific value, this is representational of someone typing this text into the fields.

*Expect –* Slightly differently too how we have done it in the past, the 2 final lines of this test check that the *wrapper* constant contains the exact element that we pass to it. We complete the check by calling the *toBeTruthy()* function, this passes the test if exact element in question is found.

Notice how we **have not** used snapshot with this test, we have checked conditions using the *wrapper.contains()* function instead.

# Shallow vs Mount

You should always use *shallow* first to render your components as it is less computationally expensive. However, if you need to either:

* Complex interaction with an element that is rendered
* Interact with a Child Component

Then you should use *mount.*

# Test Coverage

## Running with coverage

A vital concept when we test our system is to see to what extent it has been tested, this can be measured in different ways, but here we will be focusing on test coverage.

Test coverage indicates the percentage of our code base that has been touched by tests, so for instance if I never test a particular component, or any of its parent components, it will be touched by 0 tests and therefore have 0% test coverage.

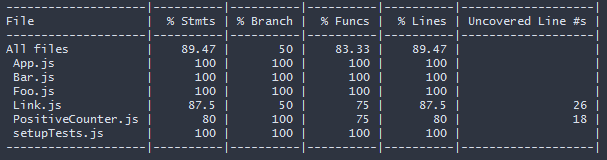
A higher % test coverage indicates a higher degree of testing, which in turn will improve confidence in the system.

To run our tests with coverage enabled and to see a report in our command line we can run:

npm test -- --coverage

Notice the -- in the middle, this sends the next flag, --coverage, as a flag to what the script actually runs.

## How do we interpret our results?



When we run our tests with coverage we will receive a report similar to the one above, each column represents something different and has different importance.

* File
  + The name of the file that the coverage corresponds too.
* % Stmts
  + The % of statements in the program that have been executed by tests.
* % Branch
  + The % of branches in a decision that have been checked, for example a simple if / else statement will have 2 ‘branches’, either true or false. If we run tests to only check one of these branches we will get only 50% coverage here.
* % Funcs
  + The % of functions in our file that have been called by our tests.
* % Lines
  + The % of executable lines in our file that have been executed.
* Uncovered Line #s
  + Ease of life information, this tells us exactly which line/s have not been touched by testing in anyway.

Notice that the first row of the table represents the project in its totality, it gives us a good average representation of how well tested our project is.

## Which % is more important?

Arguments can be made for more than one of these being the most important, however as we are aiming to test the function of the components we will say that *% Funcs* is the most important.

## What % of coverage should we aim for?

Not an easy question to answer, in fact, **‘Testing is context dependant’** this means that depending upon the system we are testing the type and degree to which we test will vary.

However, for this course we should aim for at least **70%** Functional testing in our overall project.

Further Resources

<https://medium.com/@krishankantsinghal/how-to-read-test-coverage-report-generated-using-jest-c2d1cb70da8b>

<https://devhints.io/enzyme>

<https://semaphoreci.com/community/tutorials/snapshot-testing-react-components-with-jest>

<https://hackernoon.com/testing-react-components-with-jest-and-enzyme-41d592c174f>

<https://github.com/sapegin/jest-cheat-sheet>

<https://jestjs.io/docs/en/snapshot-testing>

<https://jestjs.io/docs/en/getting-started.html>

<https://facebook.github.io/create-react-app/docs/running-tests>