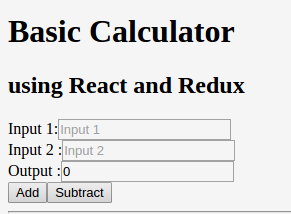
**Testing a React-Redux app using Jest and Enzyme**



And this is how the app looks. There is nothing fancy, but I chose to keep it just basic.

#### Why did I choose Jest as the testing framework?

**Jest** was never considered and I was thinking of using either **Mocha** or **Jasmine**, as these were the two frameworks I was comfortable with as I have used them on projects related to AngularJS earlier.

But on further reading and research I decided to choose **AVA** due to it being simple to setup and ability to run in parallel. And I started using AVA till I stumbled upon this post, and that’s how I finally thought maybe I should try **Jest**

These were the points I loved about **Jest**:

* I wanted some framework which **could be easily setup** and start testing. i.e spend minimum time to setup the framework for my project.
* **Ability to run tests in parallel**. This is where Mocha loses the battle for me. Well for this example running the tests in parallel is trivial. But for a huge app with lot of tests, it will be quick if the tests could run in parallel.
* **Snapshot testing**. This is a really cool feature I like. This helps me in reducing the number of tests I have to write as I just create a snapshot and if anything changes in my component, I will get an error when the snapshot is generated the next time.
* Jest uses **Jasmine for assertion** and since I have used Jasmine earlier, it was easy to write tests for me.
* **Code coverage** is available right out of the box.
* **In-built Manual mocking** (Haven’t tried yet)

When Jest was introduced, it was considerably slow and it was kind of over engineered.

But come 2016, and Facebook did a really good job in revamping Jest and I think in the coming months Jest will gain a lot of popularity. Below is from the Jest blog

### Setting up

The easiest way start working with the example I have provided is to

1. Clone the project into your local folder

git clone https://github.com/Gethyl/ReactReduxTestingUsingJestEnzyme

2. Change to that folder.

cd ./ReactReduxTestingUsingJestEnzyme

3. Install the dependencies and devDependencies by issuing yarn install or npm install

I would suggest you to learn and start using **yarn** over npm. yarn was created to overcome the shortfalls of npm and therefore I would recommend you to start exploring it. Also maybe you can read this article to get convinced :)

#### Understanding the dependencies and devDependencies installed

Take a minute to look at the package.json to see the libraries you just installed from the above step.

"dependencies": {  
 "babel-preset-es2015": "^6.18.0",  
 "babel-preset-react": "^6.16.0",  
 "babel-preset-stage-0": "^6.16.0",  
 "react": "^15.4.1",  
 "react-dom": "^15.4.1",  
 "react-redux": "^5.0.1",  
 "redux": "^3.6.0"  
 },  
 "devDependencies": {  
 "**babel-jest**": "^18.0.0",  
 "babel-loader": "^6.2.10",  
 "babel-plugin-transform-decorators-legacy": "^1.3.4",  
 "**enzyme**": "^2.7.0",  
 "**jest**": "^18.1.0",  
 "**react-addons-test-utils**": "^15.4.1",  
 "**react-test-renderer**": "^15.4.1",  
 "**redux-mock-store**": "^1.2.1",  
 "webpack": "^1.14.0",  
 "webpack-dev-server": "^1.16.2"  
 }

The packages highlighted in bold are the ones we need for testing.

\* **jest** — Unit Testing framework for ReactJs developed by Facebook. Read about it **here**.

\* **babel-jest** — To support ES6 and ES7 for our tests.

\* **enzyme** — JS testing utility developed by Airbnb to make it easier to assert React Components.

I would highly recommend you to use **enzyme** rather than testing directly using React TestUtils.

**\* react-addons-test-utils —**Provides the React TestUtils which we won’t use directly but it is required as a dependency to enzyme.

**\* react-test-renderer —**Used to grab snapshot of DOM tree rendered by React DOM/ React Native components

**\* redux-mock-store** — Used to mock our Redux store.

### Running Tests

If you check the script in package.json I have added ”test”: “jest”

So to run the test all you have to do is

yarn test

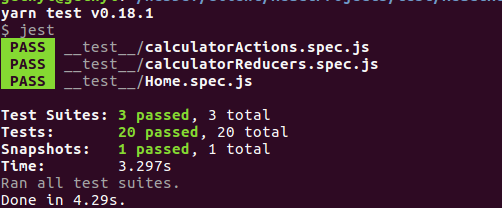
And **Jest** will pick all the test files which are put under the folder \_\_test\_\_.

Place all your test files under the folder \_\_test\_\_ as Jest by default picks all the test from here.

Also a good naming convention is to use <component>.test.js or <component>.spec.js

I have used spec.js as this was the naming I was following when I was writing test in Jasmine in earlier project.

This is how the result will look.



I have created three test suites.

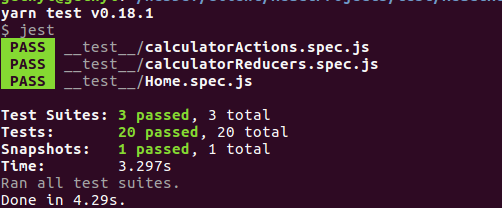
1. *Home.spec.js*— This has tests related to component and connected component.
2. *calculatorActions.spec.js* — This has unit tests for Redux Action Creators.
3. *calculatorReducers.spec.js* — This has unit tests for Redux Reducers.

Great! Now that you have run the test and see the testing passing, let us try to understand the test suites that we have created.

### Understanding our tests and test-suites

Let us go through the below sections to understand the tests better.

I will paste once again the test result I used earlier



Notice snapshots in the above screenshot? This will be explained in S**ection 1.3** below.

There are many unwanted test cases which simply check if a DOM element is present. **Those are actually not needed** and I have just kept them to show you that just in case there is a requirement for you to handle those test. It is possible as I have shown. But **I would advice not to include such scenarios in your tests.**

I will highlight only the important tests for each section that you need to know. Rest you can just read through in the github repo.

#### 1. Components/Connected Components(Home.spec.js)

Now in our case, I would like to test Home.js component. If you see the component, you will notice that it’s actually a smart component/Container or Connected Component as we will refer here.

So we will need to connect the React only part i.e dumb component as well as the connected component separately.

The entire Home.spec.js test can be found here. I will be creating Gists only for the important tests below

If you check Home.js component, we have two exports.

import React from "react"  
import ReactDOM from "react-dom"  
import {connect} from 'react-redux'  
import {addInputs, subtractInputs} from '../actions/calculatorActions'

const mapStateToProps = (state) => ({  
 output:state.output  
})

**export** class Home extends React.Component{  
 render(){  
 let IntegerA,IntegerB,IntegerC,IntegerD;

return(  
 <div className="container">  
 ......   
 ....  
 ..  
 </div>  
 );  
 }  
}

**export** default connect(mapStateToProps)(Home)

The first export will export the *dumb component* ***Home*** and the default export will export the *connected component* ***connect(Home).***

We export both so that if you want to

\* Test just the React component alone, you test the dumb component.

\* Test React-Redux part of the component, then test the connected component.

Don’t use decorator @connect to test.

You should not use decorators to ***connect*** your component as shown below:-

**@connect**(mapStateToProps)  
**export** default class Home extends React.Component{

**1.1 Components**

To test the dumb component, include

import {Home} from ‘../src/js/components/Home’

For those who are not so familiar with ES6, since **Home** is not the default export we import it with {}

and then write our test.

We use ***shallow render*** from *enzyme* for this test, as we just want to go one level deep in this test.

beforeEach(()=>{   
 wrapper = **shallow**(<Home **output={output}**/>)   
})

1. The first test is to check if the component rendered. This is the most important one to be sure that our component is rendering fine.

2. The second test checks if the prop value we pass is equal to the value 10. This is why in the above code snippet, we are passing a *prop* **output**, so that it can be used in this assertion.

Notice in Home.js our output field expects *this.props.output,* therefore we need pass the output as a prop while testing.

<div>Output :  
 <input type="text" placeholder="Output" readOnly ref="output" value={**this.props.output**}></input>  
 </div>

**1.2 Connected Components**

Next, let us look at the connected component.

As I mentioned earlier, we import the default export from Home.js. And we use that to render.

import **ConnectedHome**,{Home} from '../src/js/components/Home'

Also we use **redux-mock-store**, and therefore need to import that as well.

import configureStore from 'redux-mock-store'

Now I will show you two ways to test Connected components. ***I would love to hear from you on what is the right/preferred way to test.***

**(a) Passing the store directly and shallow render**

1. The first test is as mentioned earlier to confirm that the component is rendered.

2. Checks if the *initialState* matches with the value which the *props* get from **mapStateToProps** via the **mockStore**

**(b) Wrapping the connected component in <Provider> and full render**

The first two test cases are same as we did in **Section (a)**, we are trying to assert the same thing in a different way.

1. The first test is as mentioned earlier to confirm that the component is rendered.

2. Checks if the *initialState* matches with the value which the *props* get from **mapStateToProps** via the **mockStore.**

*An important point here if you notice, we assert on* ***Home*** *and not on* ***ConnectedHome.***

3. Checks dispatch in connected component.

**(c) Testing with actual store:-**

This is more of an integration testing. ***Do we need this? I guess not.*** I will not be doing this test for an actual project. But here just to test and make sure I can make a full round from the component to the store I have included this test.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
describe('>>>H O M E --- REACT-REDUX (actual Store + reducers) more of Integration Testing',()=>{  
 const initialState = {output:10}  
 let store,wrapper

beforeEach(()=>{  
 store = createStore(calculatorReducers)  
 wrapper = mount( <Provider store={store}><ConnectedHome /></Provider> )  
 })

it('+++ check Prop matches with initialState', () => {  
 store.dispatch(addInputs(500))  
 expect(wrapper.find(Home).prop('output')).toBe(500)  
 });

});

In (a) and (b) sections earlier, we used mock store, and *therefore we can’t capture change in state*. But here we use the actual store and therefore we can get the change in state, and assert it.

**This is not really required because this is part of React-Redux data flow.**

Just wanted to show that you could assert like this, though its not recommended as this is not an Unit test.

**1.3 Snapshot**

One thing I really liked with Jest is the snapshot testing. When jest captures snapshot for the first time, it will create a folder \_\_snapshots\_\_ under \_\_test\_\_.

First we have to import the following.

import renderer from 'react-test-renderer'

And this is our snapshot assertion:-

// Snapshot for Home React Component  
describe('>>>H O M E --- Snapshot',()=>{  
 it('+++capturing Snapshot of Home', () => {  
 const renderedValue = renderer.create(<Home output={10}/>).toJSON()  
 expect(renderedValue).toMatchSnapshot();  
 });

});

Below is how the snapshot looks for our Home.js

exports[`>>>H O M E --- Snapshot +++capturing Snapshot of Home 1`] = `  
<div  
 className="container">  
 <h2>  
 using React and Redux  
 </h2>  
 <div>  
 Input 1:  
 <input  
 placeholder="Input 1"  
 type="text" />  
 </div>  
 <div>  
 Input 2 :  
 <input  
 placeholder="Input 2"  
 type="text" />  
 </div>  
 <div>  
 Output :  
 <input  
 placeholder="Output"  
 readOnly={true}  
 type="text"  
 value={10} />  
 </div>  
 <div>  
 <button  
 id="add"  
 onClick={[Function]}>  
 Add  
 </button>  
 <button  
 id="subtract"  
 onClick={[Function]}>  
 Subtract  
 </button>  
 </div>  
 <hr />  
</div>  
`;

And say if I change something in Home.js and if I try run the test again, the test will fail at snapshot.



It clearly highlights what has been changed. And asks you to update the snapshot with command

yarn test -- -u OR npm test -- -u

And this is exactly why you don’t have to spend too much time testing a dumb component to see if an element is present or missing or what is the text etc. Because if something changes, the snapshot assertion will fail.

But snapshot doesn’t handle event and props etc, so these should be tested in your component.

Now I hope you realise why I mentioned at the start of the section, that there are many unwanted test. Those tests assert each element in Home.js which was the way we would have to do in other test frameworks, but with snapshot testing, these sort of assertions can be skipped as snapshot handles it in a simpler and better way for us. :-)

#### 2. ActionCreators(calculatorActions.spec.js)

We just try to assert that the actionCreators return the action that we expect.

import {addInputs,subtractInputs} from '../src/js/actions/calculatorActions'

describe('>>>A C T I O N --- Test calculatorActions',()=>{  
 it('+++ actionCreator addInputs', () => {  
 const add = addInputs(50)  
 expect(add).toEqual({type:"ADD\_INPUTS",output:50})  
 });

it('+++ actionCreator subtractInputs', () => {  
 const subtract = subtractInputs(-50)  
 expect(subtract).toEqual({type:"SUBTRACT\_INPUTS",output:-50})  
 });  
});

#### 3. Reducers(calculatorReducers.spec.js)

Just like actionCreator, we assert reducer.

import calculatorReducers from '../src/js/reducers/calculatorReducers'

describe('>>>R E D U C E R --- Test calculatorReducers',()=>{  
 it('+++ reducer for ADD\_INPUT', () => {  
 let state = {output:100}  
 state = calculatorReducers(state,{type:"ADD\_INPUTS",output:500})  
 expect(state).toEqual({output:500})  
 });  
 it('+++ reducer for SUBTRACT\_INPUT', () => {  
 let state = {output:100}  
 state = calculatorReducers(state,{type:"SUBTRACT\_INPUTS",output:50})  
 expect(state).toEqual({output:50})  
 });

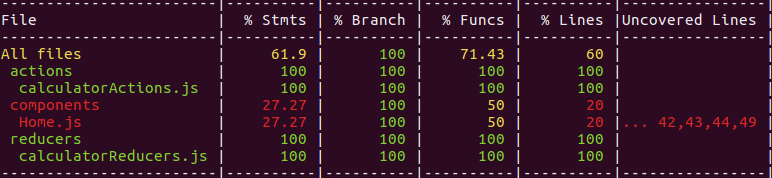
});

### Code Coverage

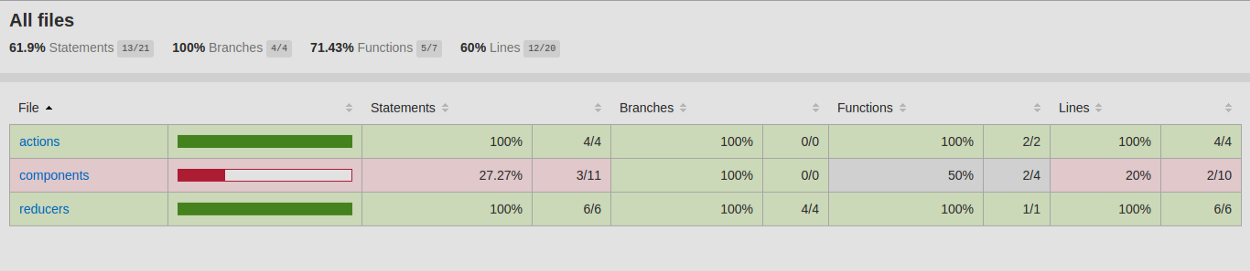
Code coverage comes right out of the box and all you have to do is issue the command as shown

yarn test -- --coverage OR npm test -- --coverage

And this is how the coverage report is created



If you check your project folder, there will be a folder coverage created. Open the index.html from this folder and this is how the report looks in a browser.



Click on the files and see the coverage detail for each line of the code. Pretty nice to have it readily available.

### Conclusion

As mentioned earlier, I would like to hear from you how you prefer to test connected components, and what all you test. It will be a learning for me :)

Hope this post helps you write test cases for React-Redux app. Especially with respect to connected components. It took me some time to figure that out.

Source Code: https://github.com/Gethyl/ReactReduxTestingUsingJestEnzyme