**Enzyme : Fast and Simple React Testing**

React has quickly climbed its way to being a top framework choice for Javascript single page applications.  
What’s not to like?

* A declarative syntax for UI
* Virtual-DOM for performance
* The possibility of server-side rendering.

There is however one area that could be improved; its built-in testing utilities – and this is where Enzyme steps in as the must have tool for front-end React testing.

This is an example of a test using the native utilities of the framework:

**const** myRenderer = ReactTestUtils.createRenderer();

myRenderer.render(<myComponent/>);

**const** output = renderer.getRenderOutput();

**const** result = scryRenderedDOMComponentsWithTag(output, div);

expect(result[0].props.children).toEqual([

<p>Title</p>

]);

Its verbose, long-winded and not that fun to develop with. The alternative put forward, Enzyme, brings it down to something much more expressive and readable:

**const** wrapper = shallow(<myComponent/>);

expect(wrapper.find('div').html()).to.equal('<p>Title</p>');

**Using the all-powerful find function**

Enzyme uses cheeriojs – a small library that implements a subset of jQuery’s core functionalities and makes manipulating components simple. The find()function, used in the example above, can be applied to HTML, JSX and CSS alike – this is key to Enzyme; It gives you the ability to target DOM elements in a clear and concise manner. Here are a few examples of how it can be applied:

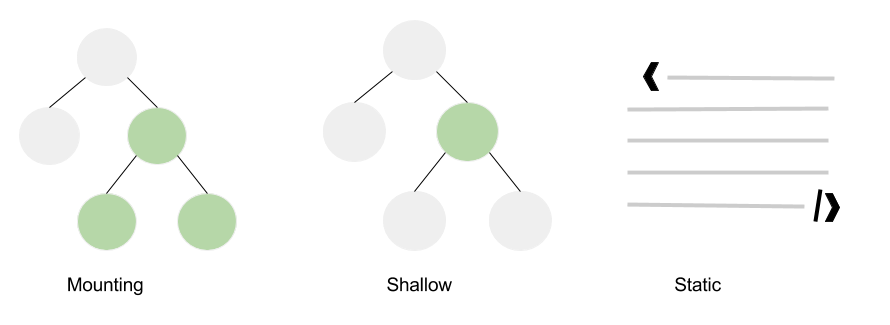
componentToTest.find('div'); *// On HTML tags*

componentToTest.find('.pretty > .red-row'); *// On CSS selectors*

componentToTest.find('div .nice-style'); *// Both !*

componentToTest.find('label[visible=true]'); *// On properties*

**The different rendering modes**

**[[](http://blog.theodo.fr/wp-content/uploads/2017/04/test.png)](http://blog.theodo.fr/wp-content/uploads/2017/04/test.png)**

To understand Enzyme’s key strengths, let’s dive a little into how it simulates components and DOM elements. Although based off react-test-utils, there is enough abstraction that the rendering of a component comes down to 3 functions – shallow, mount and render. Basically ;

* **Shallow rendering** : Is useful to test a component in isolation of every other. In the typical React pattern of smart and dumb components, shallow rendering is usually used to test ‘dumb’ components (stateless components) in terms of their props and the events that can be simulated.
* **Mounting** : Also known as full DOM rendering, it allows you to render a part of the DOM tree and it also gives you access to the lifecycle methods of React components (ComponentWillMount, ComponentWillReceiveProps , etc…)
* **Static rendering** : Is sparsely used but when it is the case, serves as means of testing plain JSX / HTML.

**Prior knowledge**

This article assumes a classic React stack making use of npm scripts, **[webpack](http://www.theodo.fr/blog/2016/07/a-comprehensive-introduction-to-webpack-the-module-bundler/)** as a module bundler along with ES6 syntax and it will detail a simple approach to testing your React application.

You may also want to have a quick look at this article if your application uses Redux ([**link to the article**](http://www.theodo.fr/blog/2016/03/getting-started-with-react-redux-and-immutable-a-test-driven-tutorial-part-1/)), as it is a common library used in React applications and knowing how to test it may be helpful, in complement to what is explored in this article.

Enjoy!

**Setup**

Enzyme is completely agnostic to the test runner and assertion libraries that you use; it works with mocha, AVA, Jest… you choose! In this article we will use, without going into too much detail, the following testing tools – so you can keep using your favourites, for me it’s:

* Jest as the test runner (although it also handles assertions and spies, I still want to use chai and sinon alongside it because of the syntaxic addons with chai-enzyme and sinon-chai).
* Chai as the assertion library.
* Sinon for mocks, stubs and test spies.

For jest the setup is simple, just remember to suffix your test files with .test.js(default configuration):

npm install --save-dev jest

And add the following scripts to your package.json scripts object :

"client:test": "NODE\_ENV=test jest",

"client:test:watch": "NODE\_ENV=test jest --watch"

Along with an object at the root of the package.json with jest as a key that configures the jest testing tool (I’ll just include a few key options):

"jest": {

"rootDir": "./client/src",

"moduleNameMapper": {

"^.+\\.(css|less)$": "<rootDir>/CSSStub.js"

},

"collectCoverage": true,

"coverageDirectory": "<rootDir>/../../coverage",

"verbose": true,

"coveragePathIgnorePatterns": [

"<rootDir>/../../node\_modules/"

]

}

**Important**: The moduleNameMapper options allows you to mock a module for files that match a particular extension. In projects using webpack it is quite typical to load css inline using the webpack css-loader. The problem is Jest doesn’t know how to interpret the css , so instead make a stub that resolves all inline styles to an empty object contained in <rootDir>/CSSStub.js

Also don’t forget to include these libraries of course!

npm install --save-dev enzyme chai-enzyme sinon

**Shallow render and the enzyme API in general**

A shallow rendered and a mounted component, have the same methods exposed but different use cases (as in, you will find the same API in the Enzyme docs for both). As a rule of thumb, shallow render is for unit testing and will probably be used for the majority of your test cases. Mounting would be more for a form of ‘front-end integration testing’ (seeing how a change in one component propagates to other components lower in the DOM tree).

**Testing your component in terms of data**

Let’s use a small snippet of code that renders a rectangle of a certain color, some text and a checkbox. Not an enthralling example, but a useful one in showing how enzyme works.

**import** React, { PureComponent } **from** 'react';

**class** **ColoredRectangleComponent** **extends** **PureComponent** {

render() {

**return** (

<div className={this.props.elementClass}>

{`Square text : ${this.props.text}`}

<input

type="checkbox"

id="checked"

value="active"

checked="checked"

onClick={(event) => { this.props.onCheckboxChange(event); }}

/>

</div>

);

}

}

export default ColoredRectangleComponent;

We want to test three things to begin with; we expect a div, with the correct class and some text. Note that once you have rendered a component for the test, you can easily control the data it handles with setProps() and setState(). You can also access the props and state of a component with props() and state(). This is particularly interesting when testing different outcomes in your component’s display (for instance; hiding part of a component, checking if an error label appears, etc…).

**import** React **from** 'react';

**import** chai, { expect } **from** 'chai';

**import** chaiEnzyme **from** 'chai-enzyme';

**import** { shallow } **from** 'enzyme';

**import** sinon **from** 'sinon';

**import** ColoredRectangleComponent **from** './enzyme';

chai.use(chaiEnzyme());

**const** clickSpy = sinon.spy();

**const** props = {

checked: true,

elementClass: 'red-square',

text: 'Enzyme rocks',

onCheckboxChange: clickSpy,

};

**const** container = shallow(<ColoredRectangleComponent {...props} />);

describe('tests for <ColoredRectangleComponent> container', () => {

it('should render one div', () => {

// You can target DOM, its children(), or an element at() a position

expect(container.find('div').length).to.equal(1);

});

it('should render one div with the correct class applied', () => {

expect(container.find('div').hasClass('red-square')).to.equal(true);

});

it('should contain the text passed as props', () => {

expect(container.text()).to.equal('Square text : Enzyme rocks');

// Here is an alternative making use of html()

expect(container.find('p').html()).to.equal('<p>Square text : Enzyme rocks</p>');

});

[...]

**Testing your component in terms of events**

You are going to want to simulate user interactions with your component. This is where chai-enzyme steps in to provide a variety of assertion addons that will simplify your test syntax. As we are using a checkbox, a quick look at the docs tell us that we are interested by (not.?)to.be.checked().

[...]

it('should render a checked checkbox if prop value is true', () => {

expect(container.find('#checked')).to.be.checked();

});

[...]

If we refer back to our tested component, a function is passed down through props and should be triggered upon clicking the element it is bound to (in this case the input tag). For the moment, event propagation and more complex mouse interactions are actively being developped but most use cases are already covered.

[...]

it('should trigger onCheckboxChange when simulating a click event on checkbox', () => {

container.find('#checked').simulate('click');

expect(clickSpy.calledOnce).to.equal(true);

});

});

**Mounting a component**

There may be instances where you don’t want to fully mount a part of the DOM just to test one nested component inside a shallowRendered component. In this case use dive() – but for every other complex case where several nested components need to be tested together, use mount. Let’s have a look at a parent component that makes use of our ColoredRectangleComponent:

**import** React, { Component } **from** 'react';

**import** \_ **from** 'lodash';

**import** ColoredRectangleComponent **from** './enzyme';

**class** **Parent** **extends** **Component** {

**constructor**(props) {

**super**(props);

**this**.state = {

squareList: [

{

text: 'number 1',

checked: true,

elementClass: 'red',

},

{

text: 'number 2',

checked: false,

elementClass: 'blue',

},

],

};

}

componentDidMount() {}

render() {

**return** (

<div >

{\_.map(this.state.squareList, (square, index) => {

return (

<ColoredRectangleComponent

key={index}

checked={square.checked}

elementClass={square.elementClass}

text={square.text}

onCheckboxChange={() => { return null; }}

/>

);

})}

</div>

);

}

}

export default Parent;

Again we’ll have a look into two simple test cases; checking if the component does mount and whether or not it renders components correctly according to its state. We are expecting 2 ColoredRectangle components with the correct css classes attributed to them.

**import** React **from** 'react';

**import** { expect } **from** 'chai';

**import** { mount } **from** 'enzyme';

**import** sinon **from** 'sinon';

**import** Parent **from** './parent';

**import** ColoredRectangleComponent **from** './enzyme';

describe('tests for <Parent> container', () => {

it('should test that the component mounts', () => {

sinon.spy(Parent.prototype, 'componentDidMount');

**const** container = mount(<Parent />);

expect(Parent.prototype.componentDidMount.calledOnce).to.equal(true);

});

it('should render 2 squares with the correct classes', () => {

const container = mount(<Parent />);

const expectedClassNamesList = ['red', 'blue'];

expect(container.find(ColoredRectangleComponent).length).to.equal(2);

container.find('div').forEach((node, index) => {

expect(node.hasClass(expectedClassNamesList[index])).to.equal(true);

});

});

});