How React Components Make UI Testing Easy

Testing back-ends is easy. You take your language of choice, pair it with your favourite framework, write some tests, and hit “run.” Your console says *“Yay! It works!”* Your continuous integration service runs your tests on every push, life is great.

Sure, test-driven development (TDD) is weird at first, but a predictable environment, multiple test runners, test tooling baked into frameworks, and continuous integration support, make life easy. Five years ago I thought tests were the solution to every problem I’ve ever had.

Then Backbone got big.

We all switched to [front-end](https://www.toptal.com/front-end) MVC. Our testable backends became glorified database servers. Our most complicated code moved into the browser. And our apps were no longer testable in practice.

That’s because testing front-end code and [UI components](https://www.toptal.com/designers/ui) is kinda hard.

It’s not so bad if all we want is to check that our models behave well. Or, that calling a function will change the right value. All we need to do for unit testing is:

* Write well-formed, isolated modules.
* Use [Jasmine](http://jasmine.github.io/) or [Mocha](http://mochajs.org/) or whatever to run functions.
* Use a test runner, like [Karma](http://karma-runner.github.io/0.12/index.html) or [Chutzpah](http://mmanela.github.io/chutzpah/).

That’s it. Our code is unit tested.

It used to be that *running* front-end tests was the hard part. Every framework had its own ideas and in most cases you ended up with a browser window that you would manually refresh every time you wanted to run the tests. Of course, you would always forget. At least, I know I did.

In 2012, Vojta Jina released the [Karma runner](http://karma-runner.github.io/0.12/index.html) (called Testacular at the time). With Karma, front-end testing becomes a full citizen of the tool chain. Our tests run in a terminal or on a continuous integration server, they re-run themselves when we change a file, and we can even test our code in multiple browsers at the same time.

What more could we wish for? Well, to *actually* test our front-end code.

**Front-End Testing Requires More Than Just Unit Tests**

Unit testing is great: it’s the best way to see if an algorithm does the right thing every time, or to check our input validation logic, or data transformations, or any other isolated operation. Unit testing is perfect for fundamentals.

But front-end code isn’t about manipulating data. It’s about user events and rendering the right views at the right time. Front-ends are about users.

Here’s what we want to be able to do:

* Test user events
* Test the response to those events
* Make sure the right things render at the right time
* Run tests in many browsers
* Re-run tests on file changes
* Work with continuous integration systems like [Travis](https://travis-ci.org/)

In the ten years I’ve been doing this, I hadn’t found a decent way to test user interaction and view rendering until I started poking at React.

**Testing UI Components with React**

[React](http://facebook.github.io/react/) is the easiest way to achieve these goals. In part, because of how it forces us to architect apps using[testable patterns](https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters), in part, because it’s got fantastic test tools.

If you’ve never used React before, you should check out my book [*React+d3.js*](http://swizec.com/reactd3js). It’s geared towards visualisations, but I’m told it’s *“an awesome lightweight intro”* to React.

React forces us to build everything as “components.” You can think of components as widgets, or as chunks of HTML with some logic. They follow many of the best principles of functional programming, except they’re objects.

For instance, given the same set of parameters, a component will always render the same output. No matter how many times it’s rendered, no matter who renders it, no matter where we place the output. Always the same. As a result, we don’t have to perform complex scaffolding to test a React component. It only cares about its properties, no tracking of global variables and config objects required.

We achieve this in large part by avoiding state. You’d call this [referrential transparency](https://en.wikipedia.org/wiki/Referential_transparency_(computer_science)" \t "_blank) in functional programming. I don’t think there’s a special name for this in React, but the official docs recommend avoiding the use of state as much as possible.

When it comes to testing user interactions, React has us covered with events bound to function callbacks. It’s easy to set up test spies and make sure that a click event calls the right function. And because React components render themselves, we can just trigger a click event and check the HTML for changes. This works because a React component cares only about itself. A click *here* doesn’t change things *there*. We will never have to deal with a nest of event handlers, just well-defined function calls.

Oh, and because React is magic, we don’t have to worry about the DOM. React uses the so-called “virtual DOM” to render components into a JavaScript variable. And a reference to the virtual DOM is all we need to test them, really.

It’s pretty sweet.

**React’s TestUtils**

React comes with a suite of built-in TestUtils. There’s even a recommended test runner called Jest, but I don’t like it. I’ll explain why in a bit. First, the TestUtils.

We get them by doing something like require('react/addons').addons.TestUtils. This is our entry point to testing user interactions and checking the output.

The React TestUtils let us render a React component by putting its DOM in a variable, instead of inserting it into a page. For example, to render a component, we’d do something like this:

var component = TestUtils.renderIntoDocument(

<MyComponent />

);

Then we can use TestUtils to check whether all children were rendered. Something like this:

var h1 = TestUtils.findRenderedDOMComponentWithTag(

component, 'h1'

);

findRenderedDOMComponentWithTag will do what it sounds like: go through the children, find the component we’re looking for, and return it. The returned value will behave like a React component.

We can then use getDOMNode() to access the raw DOM element and test its values. To check that an h1 tag in the component says *“A title”*, we’d write this:

expect(h1.getDOMNode().textContent)

.toEqual("A title");

Put together, the full test would look like this:

it("renders an h1", function () {

var component = TestUtils.renderIntoDocument(

<MyComponent />

);

var h1 = TestUtils.findRenderedDOMComponentWithTag(

component, 'h1'

);

expect(h1.getDOMNode().textContent)

.toEqual("A title");

});

The cool part is that TestUtils lets us trigger user events as well. For a click event, we’d write something like this:

var node = component

.findRenderedDOMComponentWithTag('button')

.getDOMNode();

TestUtils.Simulate.click(node);

This simulates a click and triggers any potential listeners, which should be component methods that change either the output, the state, or both. Those listeners can call a function on a parent component if necessary.

All cases are simple to test: The changed state is in component.state, we can access the output with normal DOM functions, and function calls with spies.

**Why Not Jest?**

React’s official documentation recommends using [Jest](https://facebook.github.io/jest/) as a test runner and testing framework. Jest is built on Jasmine and uses the same syntax. On top of everything you get from Jasmine, Jest also mocks everything except the component we are testing. This is fantastic in theory, but I find it annoying. Anything we haven’t implemented yet, or that comes from a different part of the codebase, is just undefined. While this is fine in many cases, it can lead to quietly failing bugs.

I’ve had trouble testing a click event, for instance. No matter what I tried, it just would not call its listener. Then I realized the function got mocked away by Jest and it never told me this.

But Jest’s worst offence, by far, is that it doesn’t have a watch mode to automatically test new changes. We can run it once, get test results, and that’s it. I like to run my tests in the background while I work. Otherwise I forget to run them.

Oh, and Jest doesn’t support running tests in multiple browsers. This is less of an issue than it used to be, but I feel like it’s an important feature for that rare occasion a [heisenbug](https://en.wikipedia.org/wiki/Heisenbug) only happens in a specific version of Chrome..

**An Integrated Example**

Anyway, we’ve seen how a good front-end test should work in theory. Let’s put it to action with a short example.

We’re going to visualise different ways of generating random numbers using a scatterplot component made with React and [d3.js](http://d3js.org/). The code is also on Github [here](https://github.com/Swizec/react-testing-example), and you can see it in action [here](http://swizec.github.io/react-testing-example/).

We’re going to use Karma as a test runner, Mocha as a testing framework, and [Webpack](http://webpack.github.io/" \t "_blank) as a module loader.

Top of Form

**Like what you're reading?**

*Get the latest updates first.*





Bottom of Form

**The Setup**

Our source files will go in a <root>/src directory, and we’ll put tests in a <root>/src/\_\_tests\_\_ directory. The idea is that we can put several directories within src, one for each major component, and each with its own test files. Bundling source code and test files like this makes it easier to reuse components in different projects.

With the directory structure in place, we can install dependencies like this:

$ npm install --save-dev react d3 webpack babel-loader karma karma-cli karma-mocha karma-webpack expect

If anything fails to install, try re-running that part of the installation. NPM sometimes fails in ways that go away on a re-run.

Our package.json file should look like this when we’re done:

// package.json

{

"name": "react-testing-example",

"description": "A sample project to investigate testing options with ReactJS",

"scripts": {

"test": "karma start"

},

// ...

"homepage": "https://github.com/Swizec/react-testing-example",

"devDependencies": {

"babel-core": "^5.2.17",

"babel-loader": "^5.0.0",

"d3": "^3.5.5",

"expect": "^1.6.0",

"jsx-loader": "^0.13.2",

"karma": "^0.12.31",

"karma-chrome-launcher": "^0.1.10",

"karma-cli": "0.0.4",

"karma-mocha": "^0.1.10",

"karma-sourcemap-loader": "^0.3.4",

"karma-webpack": "^1.5.1",

"mocha": "^2.2.4",

"react": "^0.13.3",

"react-hot-loader": "^1.2.7",

"react-tools": "^0.13.3",

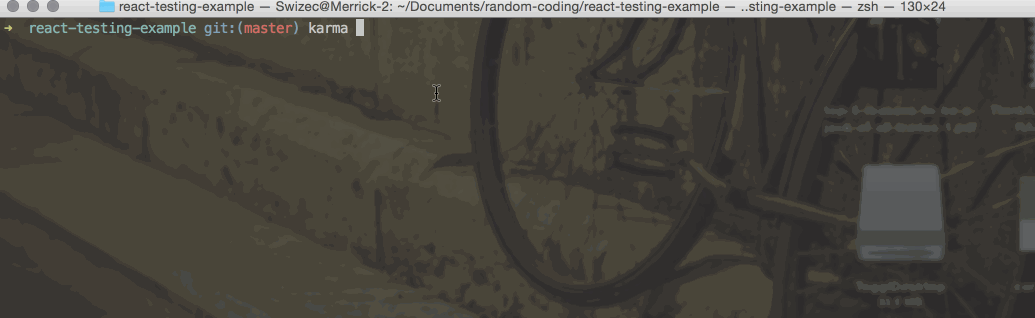
"webpack": "^1.9.4",

"webpack-dev-server": "^1.8.2"

}

}

After some configuration, we’ll be able to run tests with either npm test or karma start.



**The Config**

There’s not much to the configuration. We have to make sure Webpack knows how to find our code, and that Karma knows how to run the tests.

We put two lines of JavaScript in a ./tests.webpack.js file to help Karma and Webpack play together:

// tests.webpack.js

var context = require.context('./src', true, /-test\.jsx?$/);

context.keys().forEach(context);

This tells Webpack to consider anything with a -test suffix to be part of the test suite.

Configuring Karma takes a bit more work:

// karma.conf.js

var webpack = require('webpack');

module.exports = function (config) {

config.set({

browsers: ['Chrome'],

singleRun: true,

frameworks: ['mocha'],

files: [

'tests.webpack.js'

],

preprocessors: {

'tests.webpack.js': ['webpack']

},

reporters: ['dots'],

webpack: {

module: {

loaders: [

{test: /\.jsx?$/, exclude: /node\_modules/, loader: 'babel-loader'}

]

},

watch: true

},

webpackServer: {

noInfo: true

}

});

};

Most of these lines are from a default Karma config. We used browsers to say that tests should run in Chrome,frameworks to specify which testing framework we’re using, and singleRun to make tests run only once by default. You can keep karma running in the background with karma start --no-single-run.

Those three are obvious. The Webpack stuff is more interesting.

Because Webpack handles our code’s dependency tree, we don’t have to specify all our files in the filesarray. We only need tests.webpack.js, which then requires all the necessary files.

We use the webpack setting to tell Webpack what to do. In a normal environment, this part would go in a webpack.config.js file.

We also tell Webpack to use the [babel-loader](https://github.com/babel/babel-loader) for our JavaScripts. This gives us all the fancy new features from [ECMAScript2015](https://github.com/lukehoban/es6features) *and* [React’s JSX](https://facebook.github.io/react/docs/jsx-in-depth.html" \t "_blank).

With the webpackServer configuration, we tell Webpack not to print any debug info. It would only spoil our test output.

**A Component and a Test**

With a running test suite, the rest is simple. We have to make a component that accepts an array of random coordinates and creates an <svg> element with a bunch of points.

Following good TDD practices, we’ll write the test first, then the actual component. Let’s start with a vanilla tests file in src/\_\_tests\_\_/:

// ScatterPlot-test.jsx

var React = require('react/addons'),

TestUtils = React.addons.TestUtils,

expect = require('expect'),

ScatterPlot = require('../ScatterPlot.jsx');

var d3 = require('d3');

describe('ScatterPlot', function () {

var normal = d3.random.normal(1, 1),

mockData = d3.range(5).map(function () {

return {x: normal(), y: normal()};

});

});

First we require React, its TestUtils, d3.js, the expect library, and the code we’re testing. Then we make a new test suite with describe, and create some random data.

For our first test, let’s make sure ScatterPlot renders a title. Our test goes inside the describe block:

// ScatterPlot-test.jsx

it("renders an h1", function () {

var scatterplot = TestUtils.renderIntoDocument(

<ScatterPlot />

);

var h1 = TestUtils.findRenderedDOMComponentWithTag(

scatterplot, 'h1'

);

expect(h1.getDOMNode().textContent).toEqual("This is a random scatterplot");

});

Most tests will follow the same pattern:

1. Render.
2. Find specific node.
3. Check contents.

As we’ve seen earlier, renderIntoDocument renders our component, findRenderedDOMComponentWithTag finds the specific part we’re testing, and getDOMNode gives us raw DOM access.

At first our test will fail. To make it pass, we have to write the component that renders a title tag:

var React = require('react/addons');

var d3 = require('d3');

var ScatterPlot = React.createClass({

render: function () {

return (

<div>

<h1>This is a random scatterplot</h1>

</div>

);

}

});

module.exports = ScatterPlot;

That’s it. The ScatterPlot component renders a <div> with an <h1> tag containing the expected text, and our test will pass. Yes, it’s longer than just HTML, but bear with me.

**Draw the Rest of the Owl**

You can see the rest of our example on GitHub, [here](https://github.com/Swizec/react-testing-example/tree/master/src/__tests__). We’ll [skip describing it step-by-step in this article](http://knowyourmeme.com/memes/how-to-draw-an-owl), but the general process is the same as above. I do want to show you a more interesting test, though. A test that ensures all data points show up on the chart:

// ScatterPlot-test.jsx

it("renders a circle for each datapoint", function () {

var scatterplot = TestUtils.renderIntoDocument(

<ScatterPlot data={mockData} />

);

var circles = TestUtils.scryRenderedDOMComponentsWithTag(

scatterplot, 'circle'

);

expect(circles.length).toEqual(5);

});

Same as before. Render, find nodes, check result. The interesting part here is drawing those DOM nodes. We add some d3.js magic to the ScatterPlot component, like this:

// ScatterPlot.jsx

componentWillMount: function () {

this.yScale = d3.scale.linear();

this.xScale = d3.scale.linear();

this.update\_d3(this.props);

},

componentWillReceiveProps: function (newProps) {

this.update\_d3(newProps);

},

update\_d3: function (props) {

this.yScale

.domain([d3.min(props.data, function (d) { return d.y; }),

d3.max(props.data, function (d) { return d.y; })])

.range([props.point\_r, Number(props.height-props.point\_r)]);

this.xScale

.domain([d3.min(props.data, function (d) { return d.x; }),

d3.max(props.data, function (d) { return d.x; })])

.range([props.point\_r, Number(props.width-props.point\_r)]);

},

...

We use componentWillMount to set up empty d3 scales for the *X* and *Y* domains, and componentWillReceiveProps to make sure they’re updated when something changes. Then update\_d3 makes sure to set the domain and the range for both scales.

We’ll use the two scales to translate between random values in our dataset and positions on the picture. Most random generators return numbers in the [0,1] range, which is too small to see as pixels.

Then we add the points to our component’s render method:

// ScatterPlot.jsx

render: function () {

return (

<div>

<h1>This is a random scatterplot</h1>

<svg width={this.props.width} height={this.props.height}>

{this.props.data.map(function (pos, i) {

var key = "circle-"+i;

return (

<circle key={key}

cx={this.xScale(pos.x)}

cy={this.yScale(pos.y)}

r={this.props.point\_r} />

);

}.bind(this))};

</svg>

</div>

);

}

This code goes through the this.props.data array and adds a <circle> element for each datapoint. Simple.

Despair for your UI tests no more, with React component testing.

If you want to know more about combining React and d3.js to make data visualization components, that’s another great reason to check out my book, *React+d3.js*, [here](http://swizec.com/reactd3js).

**Fin**

That’s all we have to know about writing testable front-end components with React. To see more code using this functionality, check out the example codebase on Github, [here](https://github.com/Swizec/react-testing-example).

We’ve learned that:

1. React forces us to modularize and encapsulate.
2. This makes code easy to test.
3. Unit tests aren’t enough for front-ends.
4. Karma is a great test runner.
5. Jest has potential, but isn’t quite there yet.

If you liked this article, follow me on Twitter, [here](https://twitter.com/Swizec). Thanks for reading, and happy testing!