# **Basic**

<https://medium.com/@rossbulat/testing-in-react-with-jest-and-enzyme-an-introduction-99ce047dfcf8>

## **An introduction to testing methods in React and the tools needed for the job**

### **Start integrating Tests within your React Apps**

As React apps grow in complexity there becomes a higher likelihood that code will break. Whether abstracting, adding features or refactoring, there always lies the possibility of bugs being introduced — and this is multiplied as more developers start contributing to the project. The solution we need here are automated tests that can be run in real-time during development.

This article will introduce the tools needed to do this, what types of tests we can carry out in React, and why we would want to do so. Let’s start the journey of bug free React apps by introducing Jest — the most widely adopted Javascript testing package that Facebook have opted to use as their primary means of testing React apps.

### Introducing Jest

The first tool we will visit is [Jest](https://jestjs.io/). Jest is a node-based test runner allowing fast parallel running of tests in a node environment. It works out of the box within any [Create React App](https://github.com/facebook/create-react-app) project. Running npm test in a Terminal window within your app directory will initialise Jest and start testing in watch mode — meaning changes to files will re-run the tests associated with those files immediately as you are developing.

#### But, what about Mocha?

You may have read about or even used another Javascript testing framework called [Mocha](https://mochajs.org/); an alternative capable and feature rich package for asynchronous Javascript testing. Even though this article is Jest focused primarily, it is worth noting that Mocha is another suitable testing framework that also has strong adoption amongst the Javascript community.

At the time of writing [Jest on NPM](https://www.npmjs.com/package/jest) is boasting 3,695,443 weekly downloads, showing a lot more momentum in adoption than the [Mocha package](https://www.npmjs.com/package/mocha), currently on 2,247,393 weekly downloads. Both packages have been around a while — Jest on 147 releases and Mocha on 146 releases — and their histories have had highs and lows, prompting many online articles to route for one package or the other. However, it is clear in 2019 that Jest is winning the battle as the dominant React testing suite.

#### Not using Create React App?

If you are not using Create React App, you can install Jest with npm or yarn:

yarn add --dev jest  
#or  
npm install --save-dev jest

In order to run jest with npm test, replace or append atest script to your scripts block within your package.json file:

...  
scripts: {  
 "test": "jest",  
 ...  
}

Within a Create React App environment this will look slightly different, as react-scripts is called instead of jest: (Nothing needs to be changed here)

...  
scripts: {  
 "test": "react-scripts test",  
 ...  
}

To verify things are working, run npm test now in your Terminal at your project root directory. You will now be in watch mode, and should see a prompt displaying something similar to “No tests found related to files changed since last commit”, along with some shortcuts to run tests in various ways.

So what is the best way to introduce tests to your React project? Let’s visit this first before diving into some more useful tools.

### Start Using Jest

Jest discovers test files within your project via their filenames, of which can be located at any depth of your project. There are 3 naming conventions we can adopt in order for Jest to pick up our tests:

* **Any file with a .test.js suffix or a .spec.js suffix.** This may be preferable when a component (e.g. App.js) can be accompanied by an App.test.js file in the same directory, where they will reside next to each other. This optimises discoverability and keeps import statements to a minimum.
* **Any .js file within \_\_tests\_\_ folders throughout your project.** If you have multiple test files to test a particular component or component directory, a \_\_tests\_\_ folder allows a more coherent structure whereby your tests and components are not mixed. This may be preferable in larger projects.

Which method you adopt is your call and will depend on your project.

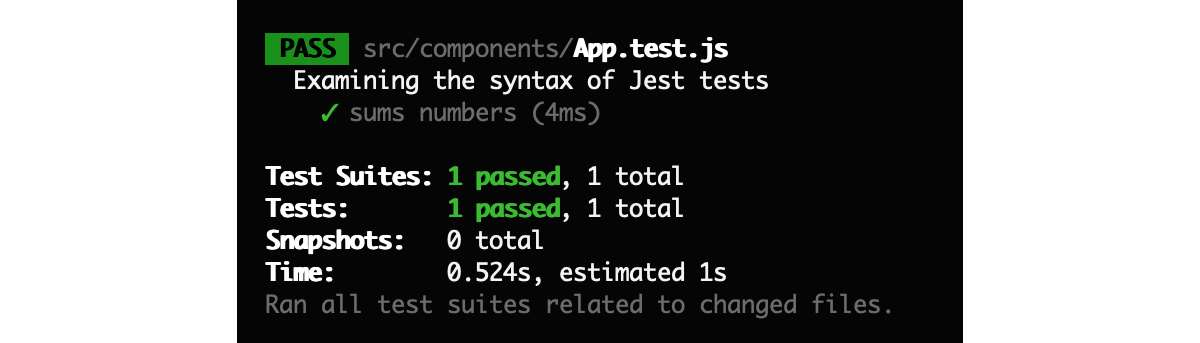
Now let’s write a simple test to examine Jest syntax. Let’s add an App.test.js file within the same directory as App.js. This test will have no relation to App.js just yet, but instead will introduce some key methods of Jest; describe() and it(), as well as the expect() methods:

**App.test.js:**

**describe**('Examining the syntax of Jest tests', () => {  
   
 **it**('sums numbers', () => {  
 **expect**(1 + 2).toEqual(3);  
 **expect**(2 + 2).toEqual(4);  
 });  
});

Upon first glance you may find this test quite obvious, and that is a good thing; readability is an intentional design choice to keep tests simple and easy to understand.

Now, upon saving this test and provided you have Jest running in the Terminal, you should receive an output similar to the following:



Let’s break down the above example to understand this syntax:

* **describe()**: An optional method to wrap a group of tests with. describe() allows us to write some text that explains the nature of the group of tests conducted within it. As you can see in the Terminal, the describe() text acts as a header before the test results are shown.
* **it()**: Similar in nature to describe(), it() allows us to write some text describing what a test should successfully achieve. You may see that the test() method is used instead of it() throughout the [Jest documentation](https://jestjs.io/docs/en/getting-started), and vice-versa in the [Create React App](https://facebook.github.io/create-react-app/docs/running-tests#writing-tests) documentation. Both are valid methods.
* **expect() and .toEqual()**: Here we carry out the test itself. The expect() method carries a result of a function, and toEqual(), in this case, carries a value that expect() should match.

**expect() makes assertions: what is an assertion?**

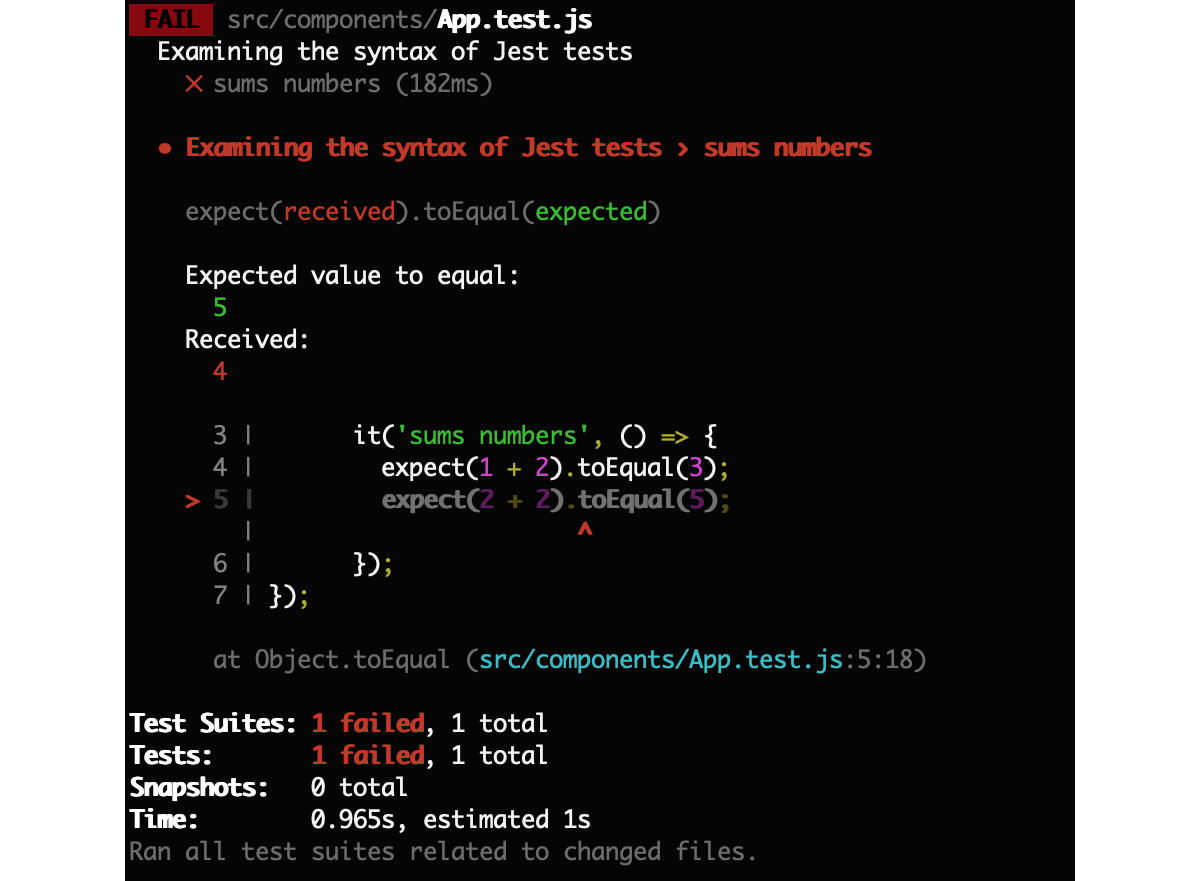
expect() is a global Jest function for making **assertions**. An assertion takes a boolean valued function and always expects that return value to be true — hence the name expect. In the event that false is returned, the test fails and execution stops within the corresponding it() or test() block.

**toEqual() is a matcher: what is a matcher?**

toEqual() is called a **matcher**. A matcher is a function whose resulting value must be true relative to what it is testing from expect(). Jest have documented the full list of matchers [here](https://jestjs.io/docs/en/expect.html#content), ranging from toContain, toBeFalsy, toMatch(regexpOrString) and toThrow(error). It’s worthwhile familiarising yourself with what is available to optimise your test syntax.

We get a full breakdown of each test result in the Terminal. Green ticks highlight a success whereas red crosses highlight a failure.

To highlight a failure (hopefully a rare occasion!) let’s change the last test to expect(2 + 2).toEqual(5):



Our terminal now displays some handy output, including exactly where our test failed within the script itself, line number and code.

### Why test two or more sums instead of just one?

Our first example above tested a sum function twice, the first time with 1 + 2 and then with 2 + 2. This is important; and to demonstrate why, let’s now introduce a sum() function instead of hard-coding our sums:

**./math.js:**

export const sum = (x, y) => x + y;

**./App.test.js:**

import {sum} from './math';

**describe**('Examining the syntax of Jest tests', () => {  
   
 **it**('sums numbers', () => {  
 expect(sum(1, 2)).toEqual(3);  
 expect(sum(2, 2)).toEqual(4);  
 });  
});

Making these changes will again yield successful tests. However, what if we change sum() to the following:

export const sum = (x, y) => 3;

What we have done here is simply return a hard-coded value, 3, from our sum function. Now, if we simply left our test suite with a singular assertion of a 1 + 2 sum, the tests would have passed even though the sum function itself does not function as we wish — it always returns 3!

Why would this happen? Mistakes are made during development. I may wish to test a fetch request handler with hard-coded values, and later on forget to switch back to my dynamic call. I could even be coding a multitude of database calls and forget to plug in some queries from my placeholders. In any case, this demonstrates that it is always useful to carry out multiple assertions for each function or component within your it() or test()block.

Up until this point we have been testing vanilla Javascript functions. Now let’s explore how we can test React components. To do so we will turn to another package to use in conjunction with Jest: [Enzyme](https://airbnb.io/enzyme/).

### Testing React Components with Enzyme

Enzyme is a testing utility package developed by Airbnb that makes testing React components easier. React do provide their own [testing suite](https://reactjs.org/docs/test-utils.html), but these tools can be verbose and granular, whereas Enzyme provides more streamlined methods to carry out tests.

Unlike Jest, Enzyme is not included with Create React App. Install Enzyme with the following:

yarn add enzyme enzyme-adapter-react-16 react-test-renderer  
#or  
npm install --save enzyme enzyme-adapter-react-16 react-test-renderer

The package you install is dependant on the version of React you are using. React 16 is used here, but Enzyme [provide](https://airbnb.io/enzyme/#installation) other packages for previous versions also.

A src/setupTests.js file also needs to be present in order to use Enzyme, which essentially configures Enzyme to run with React using the adaptor we just installed. Make sure this file is included before using Enzyme:

**src/setupTest.js:**

import { configure } from 'enzyme';  
import Adapter from 'enzyme-adapter-react-16';

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

We can go ahead and begin using Enzyme within our tests. For example, if we wish to test our <App /> component, we can extend our App.test.js file by adding the following:

import React from 'react';  
**import { shallow } from 'enzyme';**import App from './App';

describe('First React component test with Enzyme', () => {  
 it('renders without crashing', () => {  
 **shallow(<App />);**  
 });  
});

Here we have used the shallow() assertion method. shallow() will test the component we are providing, and ignores any child components that may be present in the component tree thereafter; if we had a <Header /> and <Footer /> component within <App /> for example, they would be ignored in this test.

shallow() is a type of unit test for React. A unit test is a test specifically designed to only test one function. When we are testing functions that rely on other functions, we refer to such tests as integration tests. Likewise, if we are testing an entire React component tree instead, this would be an integration test.

In Enzyme we can do such a test with mount(); a full rendering test that takes the entire component tree and lifecycle methods into consideration.

#### How does Jest test within a DOM environment?

We know that Jest runs in a node environment, but React components need to be tested within a DOM, so how are they tested? By using a package named [jsdom](https://github.com/jsdom/jsdom" \t "_blank), a Javascript implementation of the WHATWG [DOM](https://dom.spec.whatwg.org/) and [HTML](https://html.spec.whatwg.org/multipage/" \t "_blank)Standards.

jsdom is enabled by default within our Create React App testing environment, and as such, globals including window and document are provided allowing us to mimic browser behaviour when running our tests. Although jsdom allows us to perform unit and integration tests for our components, it is not recommended to perform end-to-end browser tests: Run an actual browser driver to see real world results.

You can disable jsdom by passing the --env=node flag with your test command within package.json, but this will unlikely be suitable when integration testing React components as there will very likely be some element of markup being rendered.

Create React App documentation highlights the cases of when you can disable jsdom [here](https://facebook.github.io/create-react-app/docs/running-tests#disabling-jsdom). Disabling the package will simply make your tests run faster.

### Testing Terminology

At this point it is worth going over the main terminology we encounter with React testing — let’s go over some commonly used terms:

* **Unit test:** Testing one isolated function, or one React component. Enzyme’s shallow() is a unit test.
* **Integration test:** Testing a multitude of functions working together, or an entire React component including children components. Enzyme’s mount() is an integration test.
* **Mock function:** Redefining a function specifically for a test to generate a result. E.g. returning hard-coded data instead of relying on fetch requests or database calls. This strategy could prevent the hard-coded sum issue we were discussing earlier!Mock functions can be defined in jest with jest.fn(() => { //function here });.

Testing terminology:

* **Smoke test:** Verifies that a component renders without throwing. Utilising Enzyme’s shallow() is indeed a smoke test, as is mount().
* **Shallow rendering:** rendering a component with no children. Enzyme’s shallow() is a form of shallow rendering, whereas mount() is not.
* **Full rendering:** Rendering a component and all of its children. mount()is a full rendering method whereas shallow() is not.
* **Static rendering:** Rendering components to static HTML files and then analysing the results. Enzyme’s render() method is used for static rendering.

### What Next?

We will wrap things up here to save this article from becoming too long. The Jest and Enzyme packages offer a lot of functionality, much more than we can fit within this introductory article.

If you would like to continue reading, my next article focuses on test driven development (TDD), as well as coverage of the most widely tools Jest and Enzyme have to offer, including snapshotting, mocking and simulating events:

# Advance - Test Driven Development in React with Jest and Enzyme

<https://medium.com/@rossbulat/test-driven-development-in-react-with-jest-and-enzyme-2a6cf2cc3071>

## Exploring snapshots, mock functions simulation testing and more

### Introduction: How do you Develop?

This talk covers a range of testing tools that Enzyme and Jest offer for React, ranging from **snapshot testing**, **mock functions**, **simulating events**, and how to test component functions, props and state. We will also explore other aspects of testing including abstract-ability and logic isolation within your React projects, and finally wrap up with a testing checklist to adhere to when designing your own tests.

[Enzyme](https://airbnb.io/enzyme/) and [Jest](https://jestjs.io/) are critical tools that can ensure a high level of stability to your code from your first deployment, and can be used whether you are adopting a Test Driven Development or Behaviour Driven Development strategy. Before jumping into the tools let’s review the differences between these two development approaches in an abstract way.

#### Behavioural Driven Development

How much time do you spend testing your apps? Is it done **after a milestone**or **at** **intervals as you are developing components**? The former approach gives precedence to the behaviour of the app, prioritising business logic completion over component stability. Your testing suites will be coded once the behavioural implementation is complete, the results of such tests determining the best route to refactoring your code or fixing any holes along the way.

What this approach is doing in essence is integration testing your components before writing your unit tests — or any formal tests at all using a sophisticated framework like Jest. This indeed yields a faster turnaround, and is quite widely adopted. Why is this the case? It could be because business meetings rarely discuss apps from a test driven angle — the focus is always on the end result or the intended behaviour set out to be achieved. It is arguably natural to lean towards BDD given these factors; meeting notes will be totally behavioural-based, with a tight schedule to get the job done.

Now, this may be great for rapid prototyping or creating a proof of concept — but for apps with a sizeable codebase, complex integrations with a range of services, or simply a large team concurrently contributing (a blockchain being a great example for all of these cases), a test-first approach to developing makes a lot of sense, in the form of Test Driven Development — and this method also happens to be naturally suited for React development.

#### Test Driven Development

Whether you are developing a front end user experience or an NPM package that thousands of developers and millions of end-users rely on, stability and reliability will be among your highest priorities for your codebase. What TDD does is promote clean and functional code, prompting the developer to consider things like how the component will be integrated, how reusable and how refactor-able it is.

TDD promotes developing with intent rather than procedure. By doing this, lots of complexity of stitching an entire experience together is removed. React components already do a great job at compartmentalising logic, making them suited to TDD whereby unit tests and integration testing can be carried out on a per-component basis.

Ultimately, TDD aims to solve the problem of delivering buggy software to the end consumer by **treating the testing procedure as part of the development process**, rather than a job to do at the end of a sprint. Minimising bugs is of course in the interest of an entire organisation, and not just the dev team. In other words, testing should be something everyone cares about, not only developers.

This is all very abstract, so let’s delve into React specific TDD to get a feel of what can be done here.

This article builds on some basic knowledge of Enzyme and Jest. If you would like to familiarise yourself with Jest and Enzyme at this stage, please refer to my introductory article on the packages:

[**Testing in React with Jest and Enzyme: An Introduction**  
An introduction to testing methods in React and the tools needed for the jobmedium.com](https://medium.com/@rossbulat/testing-in-react-with-jest-and-enzyme-an-introduction-99ce047dfcf8)

### Structuring React Components for Testing

React developers accustomed to developing components will know that component logic can build up very fast as state changes, API calls and other logic like mappings are introduced. What we want to do to this logic is **ensure that it works under all the circumstances it will be used with**. To do this, we want to achieve 2 things:

#### **1. Test the logic in an isolated manor**

Abstracting code becomes a primary concern in TDD, which also indirectly promotes principles such as DRY (Don’t Repeat Yourself) for maximum component reusability. This is easily achieved in React with the flexibility of components.

Consider an example project, a navigation app with a <SideBar /> and <TopBar /> component — both of which use an <Item /> component for navigation links. By separating each component (and their dedicated styling), we are isolating the logic as well as the tests associated with each component.

Remember, TDD with component isolation in mind would limit a component’s purpose to do just one thing, and reject the notion of it doing anything else but that thing.

Check out this hypothetical navigation app project structure:

**// @rossbulat/nav package**  
src/  
 Item/  
 index.js  
 index.scss  
 SideBar/  
 index.js  
 index.scss  
 TopBar/  
 index.js  
 index.scss  
tests/  
 Item.test.js  
 nav.test.js  
 SideBar.test.js  
 TopBar.test.js

**// @rossbulat/app**src/  
 App.js  
 api.js  
 Home/  
 index.js  
 index.scss

tests/  
 app.test.js  
 api.test.js

**// src/Home/index.js**import { SideBar, TopBar } from '@rossbulat/nav';

For ultimate abstract-ability, the navigation components actually come as a package, @rossbulat/nav, allowing me to include them in more projects to serve navigation needs. This warrants the usage of abstract component names like <Item /> as well as every component to be listed one level down under src/. With these components totally isolated, and with the testing suite embedded within the package, our navigation is much easier to manage.

Separating both <SideBar /> and <TopBar /> ensures that I can test each component independently. Not only this, each navigation link consists of an <Item /> component; this component may include props like title, link, icon, and more — all of which can be tested in an isolated manor.

As well as my separate component tests, I also have a nav.test.js file that can include my integration tests, or all 3 components working together, to make sure they adhere to an expected snapshot — perhaps checking how many <li> tags are nested within <SideBar /> markup to coincide with how many <Item /> components are present. We will visit snapshots further down.

My API is also completely separate from other components within my main app project; the api.test.js file is specifically for API calls, which will no doubt include various mock functions to serve fake API calls. Again, we will explore mocks further down this talk.

#### **2. Test the code within a range of scenarios**

This can only be achieved if the prior condition of abstracting logic is met, whereby an extensible range of props can be passed into the component, or in the case of integration testing, an extensible number of other components that work together.

With the power of isolation in mind, let’s next explore some tangible code examples of testing React components.

Perhaps the simplest of ways to test a React component is to compare the result of a render function with a snapshot file**.**Jest provide us with tools necessary for doing exactly this.

### Using Snapshots with Jest

Using Jest’s terminology, [snapshot testing](https://jestjs.io/docs/en/snapshot-testing.html) is a useful feature to make sure that your markup does not unexpectedly change, and equally as true, makes sure that render() outputs what you intended it to.

Testing snapshots against components can be done with Enzyme methods (which we will see below) but can also be used with the [react-test-renderer](https://www.npmjs.com/package/react-test-renderer)package (extremely popular with over 2 million weekly downloads at publishing time). Add it to your project via NPM or yarn:

yarn add react-test-renderer

A snapshot file is automatically generated when the toMatchSnapshot()method is called within your tests, often in conjunction with expect() in the following form:

expect(<component>).toMatchSnapshot();

The contents of a snapshot file mostly consists of markup that represents the expected output of your React component. [Here](https://github.com/facebook/jest/blob/master/examples/snapshot/__tests__/__snapshots__/link.react.test.js.snap) is an example .snap file that Facebook have provided, that tests different link states.

Even though we would not manually edit a snapshot file under normal circumstances, they are easy to read by design for the purpose of code reviewing, and are designed to be added to source control — commit your snapshot files with your code updates.

Let’s visit a practical example of a snapshot test — I will test whether my <Item /> component from earlier renders correctly. Note that running the test for the first time will generate the snapshot file:

import React from 'react'  
import { Item } from '@rossbulat/nav'  
import renderer from 'react-test-renderer'  
import icon from './img/ross.png'

describe('testing navigation', () => {  
 it('renders correctly', () => {  
 const item = renderer  
 .create(  
 <Item   
 link="https://rossbulat.co"   
 text="My Homepage"   
 icon={icon} />  
 ).toJSON();

**expect(item).toMatchSnapshot();**  
 });  
});

This next example using Enzymes shallow() method, in this case testing whether my <SideBar /> component will render correctly against a snapshot:

...  
it('sidebar should render correctly', () => {  
 const sidebar = shallow(<SideBar />);  
 expect(sidebar).toMatchSnapshot();  
});

I could also test whether <SideBar /> renders correctly given an array of links:

it('sidebar should render links correctly', () => {  
 const links =   
 [{link: 'https://rossbulat.co', text: 'My Homepage'},  
 {link: 'https://medium.com/@rossbulat', text: 'Medium'}];

const component = shallow(<MyComponent links={links} />);  
 expect(component).toMatchSnapshot();  
});

This test will generate another snapshot. Concretely, each of my tests will generate a different snapshot file, even if the same component is being used.

If you are running your tests in watch mode, with yarn test --watch, followed by pressing i for interactive snapshot mode, failed snapshots will pop up as they happen and the exact failure will be documented, allowing you to troubleshoot inconsistencies.

Snapshot files are generated in a \_\_snapshots\_\_ folder under your test directories, all files of which will be named **<test\_name>**.snap.

**Note:** The package used to convert Javascript objects into string representations within these .snap files is [pretty-format](https://github.com/facebook/jest/tree/master/packages/pretty-format). The package is just another NPM package, and can be used in your own projects where you wish to debug or document Javascript implementations.

If after you have updated your UI and wish to update your snapshots, we have a handy command to do just that. Run the following in your project directory to update all your snapshots:

jest --updateSnapshot

You will want to update your snapshots as you are enhancing your components. Make sure your initial snapshot outputs are as expected before running subsequent tests against them.

For the full documentation and feature set of snapshots in Jest, [visit this page](https://jestjs.io/docs/en/snapshot-testing.html).

### Simulating Events

Simulating events such as clicks and inputting text can be done within our tests using Enzyme’s simulate() method. For example, let’s say I wanted to test the result of clicking a button inside a component. This can be done like so:

it('should update form submitted state with button click', () => {  
 const component = mount(<RegistrationForm />);  
 component  
 .find('button#submit\_form')  
 .simulate('click');  
   
 expect(component.state('form\_submitted')).toEqual(true);  
 component.unmount();  
});

Here we are testing a component, <RegistrationForm />, and the resulting state after clicking the button with id #submit\_form. As demonstrated, component state can be accessed component.state('**<key>**'), and then compared with a matcher with expect().

In reality, we would probably need a form to be filled before submitting a registration form, at least with an email address or phone number. Can we easily test input values? Yes we can, like so:

component  
 .find('#name')  
 .simulate('change', { target: { value: 'Ross' } });

Similarly, we can also toggle checkboxes with simulate():

component  
 .find('#agreetoterms')  
 .simulate('change', {target: {checked: true}});

And even simulate a key press via a keycode (refer to keycodes [here](http://keycode.info/)):

component.find('#input').simulate('keydown', { keyCode: 70 });

You may also wish to call component methods before testing state, snapshots or anything else. We can access component methods by its .instance()property:

const component = shallow(<MyComponent />);  
const result = component.instance().callMethod();

### Using Mock Functions

As the name suggests, [Mock functions](https://jestjs.io/docs/en/mock-functions.html) allow us to re-implement a function, stripping away logic for the purpose of capturing calls to the function, testing parameters, and testing return values.

The simplest way to use mock functions is to simply define an empty function and put it in place of an actual function within your tests. The benefit of doing this is tracking whether the function was actually called. (How many times have you been debugging and noticed that a particular function was not actually called, leading to the error?).

We can define such a function by using jest.fn(). This is how we would replace an onClick handler with an empty mock function, and test whether it has been called:

**//define empty mock function**const fnClick = jest.fn();

describe('click events', () => {  
 it('button click should show menu', () => {

**//replace actual function with mock function**  
 const component = shallow(<MyButton **onClick={fnClick}** />);  
   
  **//simulate a click**  
 component  
 .find('button#btn\_open\_menu')  
 .simulate('click');  
   
 **//check if function was called**  
 expect(fnClick).toHaveBeenCalled();  
 });  
});

The matcher .toHaveBeenCalled() is only one of a few we can test with mocks. Here are a few more:

**// Test how many times the function is called**expect(fnClick.mock.calls.length).toBe(3);

**// Test the values passed as arguments**  
// The second argument of the third call to the function was 'yes'  
expect(mockCallback.mock.calls[2][1]).toBe('yes');

**// Test return values**  
// The return value of the second call to the function was true  
expect(mockCallback.mock.results[1].value).toBe(true);

Here we have tapped into the .mock property of a mock function, giving us access to information about how the function has been called, and what has been returned. Read more about it [here](https://jestjs.io/docs/en/mock-functions.html#mock-property).

#### Injecting Mock Functions

Another cool thing we can do with mocks is inject them in a test whenever we wish to retrieve the return value, with console.log():

const myMock = jest.fn();  
console.log(myMock('return this string'));

// > return this string

We can also direct a mock function to return specific values before calling it within the test. This is handy to simulate a range of return values as proof your app can handle them:

// return `true` for first call,   
// return `false` for the second call   
// return a string 'hello mock' for the third call

myMock.mockReturnValueOnce(**true**)  
 .mockReturnValueOnce(**false**)  
 .mockReturnValueOnce('hello mock');

//call the mock three times to witness the results:  
console.log(myMock(), myMock(), myMock());

This is useful for retrieving data and simulating form submissions where there is arbitrary data being supplied. Test whether your app can handle an undefined value for example, along with a range of string lengths for input fields, as well as unexpected values, e.g. An object where a boolean value is expected.

Imagine a scenario where a JSON string {result: true} is returned instead of true. The JSON string itself is s truthy value, with a value of {result: false} also evaluating as truthy. Testing return values with mocks will ensure these scenarios do not happen.

Mock functions can be used in conjunction with everything we have explored so far, including snapshots and simulating events. You will find your tests becoming more sophisticated as more scenarios are covered. You are of course free to actually implement mock functions:

const myMock = jest.fn(() => {  
 ...  
});

The general rule of doing so is to not introduce arbitrary logic to the original function. What if one implementation is not enough — what if we would like to test multiple implementations of the same function? This can be done with mockImplementationOnce(() => {...}), taking the implementation itself as an argument:

**//define implementations**

const myMockFn = jest.fn(() => 'default return value')   
 .mockImplementationOnce(() => {  
 ...  
 return 'first implementation';  
 })  
 .mockImplementationOnce(() => {  
 ...  
 return 'second implementation';  
 });

**//call function**  
console.log(myMockFn(), myMockFn());

**//any calls hereafter will refer to the default implementation.**

Check out more on mock implementations [here](https://jestjs.io/docs/en/mock-functions.html#mock-implementations).

Modules can also be mocked using jest.mock(). Think database calls or API requests that are slow and fragile — things could break quite easily making for an unreliable test. Jest provide a [simple example](https://jestjs.io/docs/en/mock-functions.html#mocking-modules) of mocking the axios module to overwrite API calls, but this is the general gist of how to mock a module function:

**// import the module to mock**import axios from 'axios';

**// wrap the module in jest.mock()**jest.mock('axios');

test('should fetch users', () => {  
 const users = [{first\_name: 'Ross'}];  
 const resp = {data: users};

**// append .mockResolvedValue(<return value>) to the module method**  
 axios.get.mockResolvedValue(resp);

**// carry out your test**  
 return expect(resp.data).toEqual(users));  
});

This is quite an elegant solution to bypassing module methods; by overwriting the functionality with Jest — .mockResolvedValue(<return value>) allows us to do exactly this.

#### Debugging components

It is worth mentioning here that if you wish to log a component’s render()output in HTML like fashion, use the debug() method of a wrapper:

const component = shallow(<MyComponent />);  
console.log(component.debug());

Debugging components within your tests can quickly pinpoint why a test is failing.

### Summary

In conjunction with my [original introduction on testing](https://medium.com/@rossbulat/testing-in-react-with-jest-and-enzyme-an-introduction-99ce047dfcf8), we have covered enough in this talk to get into serious React testing suites, with a range of methods that when used together can provide a comprehensive means of testing components.

Have we touched everything? No — but these powerful tools will kickstart your testing suite with Jest and Enzyme:

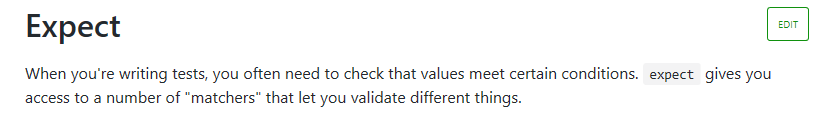
* **Snapshots:** allowing you to compare render() output of components to an expected result.
* **Simulations:** carry out tests that involve the events you would expect in the browser — click events, form inputs and function calls.
* **Mock functions:** Test arguments and return values with empty mock functions, or carry out a range of implementations to test the expected result. Also, overwrite fragile module methods that rely on external data sources, or involve heavy processing that would slow down your tests.
* **Abstract-ability:** keep logic isolated and tied into specific components, promoting re-usability, DRY and wide-ranging integration testing.

#### How will you develop your tests?

The tools we have discussed here are used with BDD and TDD. I personally think that TDD can be adopted for the majority of application projects, but is particularly suited for long term, protocol based projects, API services, critical package and infrastructure projects, as well as cutting-edge research projects. In agile development, perhaps opting for a particular method depending on your sprint goals will make sense for your team.

# Jest – expect

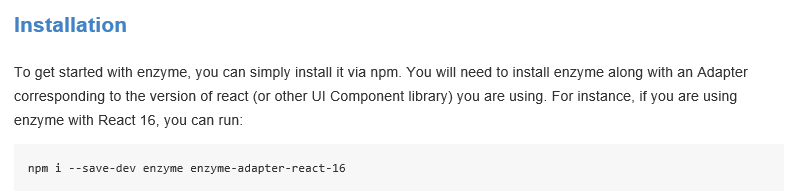
<https://jestjs.io/docs/en/expect.html#content>

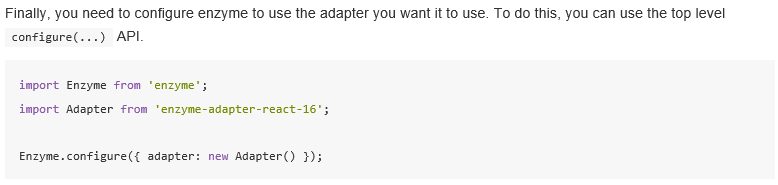


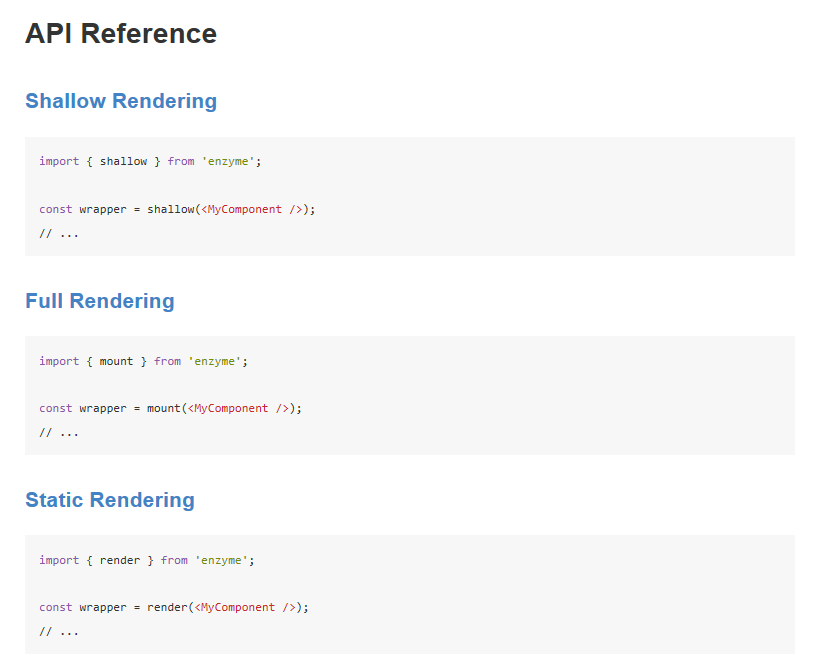


# Enzyme

Enzyme is a JavaScript Testing utility for React that makes it easier to test your React Components' output. You can also manipulate, traverse, and in some ways simulate runtime given the output.







## Shallow Rendering API

Shallow rendering is useful to constrain yourself to testing a component as a unit, and to ensure that your tests aren't indirectly asserting on behavior of child components.

As of Enzyme v3, the shallow API does call React lifecycle methods such as componentDidMount and componentDidUpdate.



## **Full Rendering API (mount(...))**

Full DOM rendering is ideal for use cases where you have components that may interact with DOM APIs or need to test components that are wrapped in higher order components.

Full DOM rendering requires that a full DOM API be available at the global scope. This means that it must be run in an environment that at least “looks like” a browser environment. If you do not want to run your tests inside of a browser, the recommended approach to using mount is to depend on a library called [jsdom](https://github.com/tmpvar/jsdom) which is essentially a headless browser implemented completely in JS.

**Note**: unlike shallow or static rendering, full rendering actually mounts the component in the DOM, which means that tests can affect each other if they are all using the same DOM. Keep that in mind while writing your tests and, if necessary, use [.unmount()](https://airbnb.io/enzyme/docs/api/ReactWrapper/unmount.html) or something similar as cleanup.



## Static Rendering API

Use enzyme's render function to generate HTML from your React tree, and analyze the resulting HTML structure.

render returns a wrapper very similar to the other renderers in enzyme, [mount](https://airbnb.io/enzyme/docs/api/mount.html) and [shallow](https://airbnb.io/enzyme/docs/api/shallow.html); however, render uses a third party HTML parsing and traversal library [Cheerio](http://cheeriojs.github.io/cheerio/). We believe that Cheerio handles parsing and traversing HTML extremely well, and duplicating this functionality ourselves would be a disservice.

For the purposes of this documentation, we will refer to Cheerio's constructor as CheerioWrapper, which is to say that it is analogous to our ReactWrapper and ShallowWrapper constructors.



# Snapshot Testing

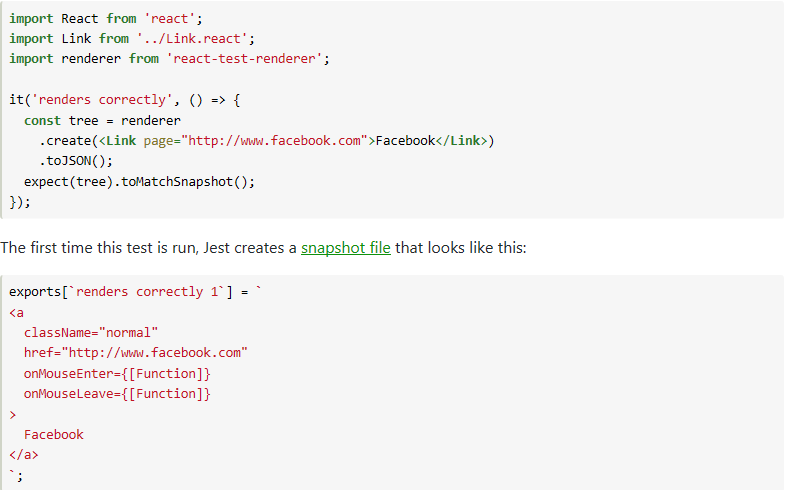
https://jestjs.io/docs/en/snapshot-testing

Snapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly.

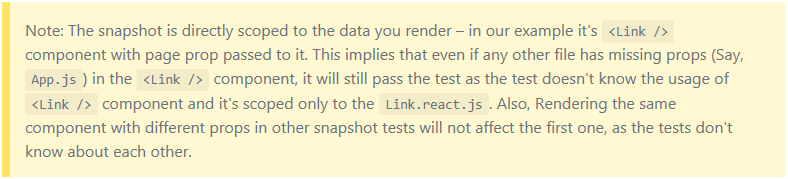
A typical snapshot test case for a mobile app renders a UI component, takes a snapshot, then compares it to a reference snapshot file stored alongside the test. The test will fail if the two snapshots do not match: either the change is unexpected, or the reference snapshot needs to be updated to the new version of the UI component.

## Snapshot Testing with Jest

A similar approach can be taken when it comes to testing your React components. Instead of rendering the graphical UI, which would require building the entire app, you can use a test renderer to quickly generate a serializable value for your React tree.



The snapshot artifact should be committed alongside code changes, and reviewed as part of your code review process. Jest uses [pretty-format](https://github.com/facebook/jest/tree/master/packages/pretty-format) to make snapshots human-readable during code review. On subsequent test runs Jest will simply compare the rendered output with the previous snapshot. If they match, the test will pass. If they don't match, either the test runner found a bug in your code (in this case, it's <Link> component) that should be fixed, or the implementation has changed and the snapshot needs to be updated.



## **Updating Snapshot**

