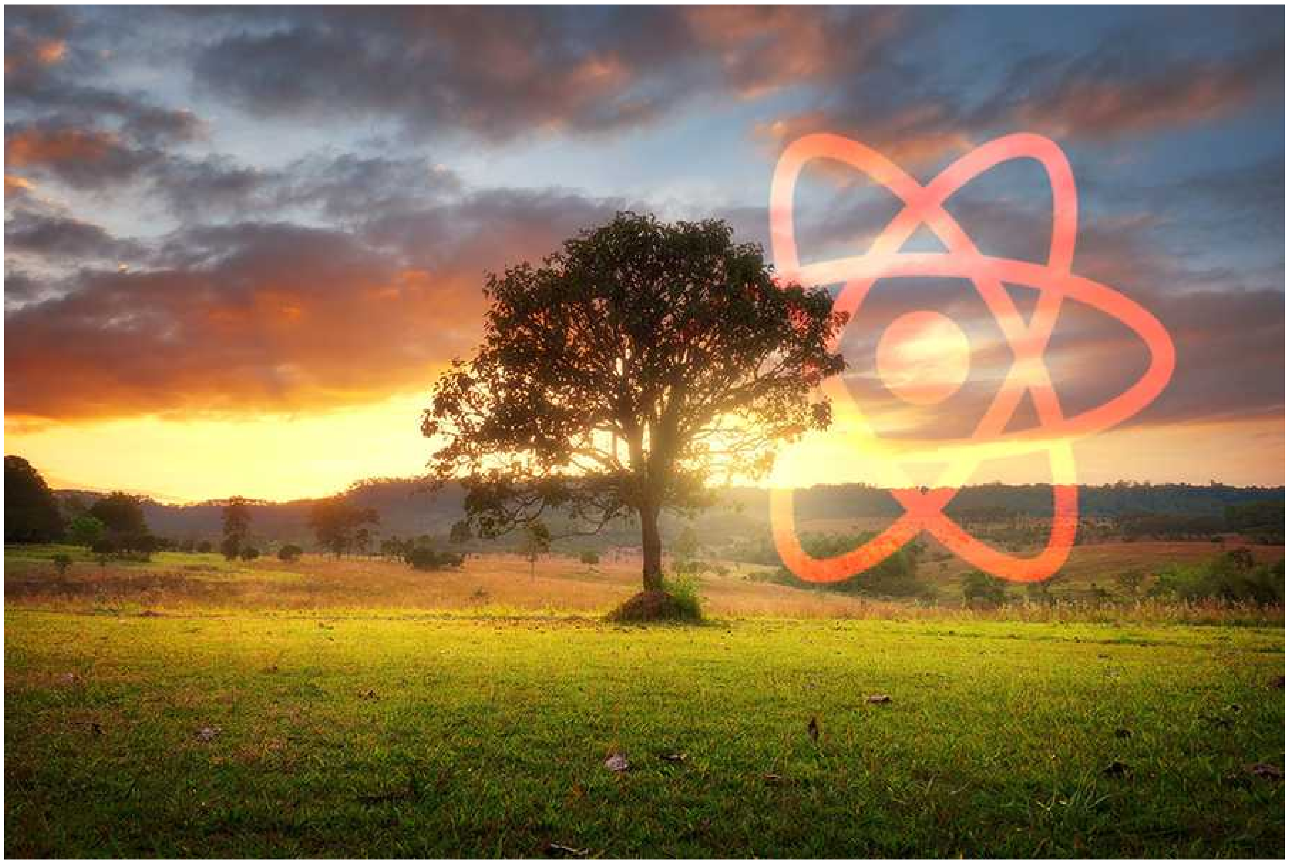
# How to test React with Jest & Enzyme

JULY 17, 2019 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 17k Follow on Facebook



This tutorial is part 3 of 3 in the series.

Part 1: How to set up React with Webpack and Babel
Part 2: How to test React components with Jest

In this React testing tutorial, we will introduce Enzyme in our Jest testing environment. Jest is commonly used as test runner -- to be able to run your test suites and test cases from the command line with optional configuration -- but also to make assertions in your test cases. In contrast, Enzyme can be used within Jest to render components, to access the DOM of these components, and to make assertions based on the DOM. Enzyme adds up perfectly to Jest, because it can cover unit and integration tests, whereas Jest is mainly used for snapshot tests. However, Enzyme is not strictly tied to Jest, it can be used in any other test runner too.

## ENZYME IN JEST SETUP

Enzyme makes it effortless testing React components with integration and unit tests. It is a testing library by Airbnb which got introduced for component tests in React, because it offers different rendering techniques for your React components and selectors to go through your rendered output. The rendered output is taken for the assertions in Jest then.

Let's go through the setup for Enzyme in Jest testing. First, you have to install Enzyme on the command line as development dependency:

```
npm install --save-dev enzyme
```

Enzyme introduces adapters to play well with different React versions. That's why you have to install such an adapter for your test setup too. The version of the adapter depends on your React version:

```
npm install --save-dev enzyme-adapter-react-16
```

In this React testing tutorial, we are using React 16. That's why the Enzyme adapter for React 16 gets installed here. So make sure to check the React version in your application for installing the appropriate adapter. In the next step, we want to set up Enzyme with its adapter in our Jest testing environment. Therefore, Jest offers a so called setup file to make this happen. First, create this Jest setup file on the command line:

```
touch jest.setup.js
```

Second, give it the following setup instructions to make Enzyme play well with React in your Jest testing environment:

```
import React from 'react';

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

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

Now, in your *jest.config.json* file, define this new Jest setup file as dependency:

```
{
  "testRegex": "((\\.|/*.)(spec))\\.js?$",
  "setupFilesAfterEnv": [
    "<rootDir>/jest.setup.js"
  ]
}
```

That's it. You have set up Enzyme in Jest for your React component tests. Next we will dive into your first tests written with Enzyme and Jest.

**Exercises:**

- Read more about getting started with Enzyme

## ENZYME UNIT/INTEGRATION TESTING IN REACT

The Enzyme in Jest setup is up and running. Now you can start to test your React component(s). The following section should show you a couple of basic patterns which you can apply in your React component tests. If you follow these testing patterns, you don't have to make a costly mental decision every time when you test a React component.

You have already exported the Counter component from the *src/App.js* file. So it should be possible to test the following assumption: an instance of the Counter component is rendered when you render the App component. Therefore, add your new test in the *src/App.spec.js* file:

```
import React from 'react';
import renderer from 'react-test-renderer';
import { mount } from 'enzyme';

import App, { Counter, dataReducer } from './App';

const list = ['a', 'b', 'c'];

describe('App', () => {
  describe('Reducer', () => {
    ...
  });
  test('snapshot renders', () => {
    const component = renderer.create(<App />);

    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

  it('renders the inner Counter', () => {
    const wrapper = mount(<App />);
    expect(wrapper.find(Counter).length).toEqual(1);
  });
});

describe('Counter', () => {
  ...
});
```

Basically we just render the App component, use the output to traverse through the DOM by finding the Counter component, and make an equality check that the component's instance is available. As an exercise in between for yourself, try to draw the line between Enzyme and Jest for this test case. What belongs to which testing library?

Whereas Jest is still your test runner -- with its testing setup and optional configuration -- which offers you the surrounding test suites (`describe`-block), test cases (`it`-block and `test-block`), and assertions (`expect`, `toEqual`), Enzyme gives you the new renderer to render your React component (`mount` among others) and an API to traverse the DOM (`find` among others) of it.

*Note: Jest comes with two test case scenarios expressed with `it` and `test`. It's up to you how you use them, but I like to distinguish my snapshot and unit/integration tests with them. While the `test`-block is used for my snapshot tests, the `it`-block is used for integration and unit tests with Enzyme.*

The line between unit and integration test isn't clearly defined. There is lots of room to argue that testing two React components is either a unit or integration test. On the one hand, testing two components in one isolated environment can be called a unit in itself, but also, because two components work together, it could be called an integration between the two as well.

Let's write another test to check the interplay between the two components. In this case, we want to assert whether the child component renders the expected output when we render our parent component. We are using the `mount` function again, because it renders our child components too. In contrast, other rendering functions from Enzyme are only rendering the actual component.

```
import React from 'react';
import renderer from 'react-test-renderer';
import { mount } from 'enzyme';

import App, { Counter, dataReducer } from './App';

const list = ['a', 'b', 'c'];

describe('App', () => {
  describe('Reducer', () => {
    ...
  });

  ...

  it('renders the inner Counter', () => {
    const wrapper = mount(<App />);
    expect(wrapper.find(Counter).length).toEqual(1);
  });

  it('passes all props to Counter', () => {
    const wrapper = mount(<App />);
    const counterWrapper = wrapper.find(Counter);

    expect(counterWrapper.find('p').text()).toEqual('0');
  });
});

describe('Counter', () => {
  ...
});
```

Again, you are rendering your React component with Enzyme, traverse through your component by instance (e.g. `Counter`) and HTML elements (e.g. p), and make an equality check on the rendered inner text of the HTML element. Since no one clicked the buttons yet, the output should resemble the initial given state from the App component.

The last tests have shown you how to access the DOM of the rendered output via Enzyme and how to make assertions on the rendered output via Jest. Let's take this one step further by testing interactions on our HTML elements. For instance, our two button elements can be used to increment and to decrement the counter state in the App component. Let's simulate click events with Enzyme and check the rendered output in our child component afterward:

```
import React from 'react';
import renderer from 'react-test-renderer';
import { mount } from 'enzyme';
import { mount } from 'enzyme';

import App, { Counter, dataReducer } from './App';

const list = ['a', 'b', 'c'];

describe('App', () => {
  describe('Reducer', () => {
    ...
  });

  ...

  it('passes all props to Counter', () => {
    const wrapper = mount(<App />);
    const counterWrapper = wrapper.find(Counter);

    expect(counterWrapper.find('p').text()).toEqual('0');
  });

  it('increments the counter', () => {
    const wrapper = mount(<App />);

    wrapper
      .find('button')
      .at(0)
      .simulate('click');

    const counterWrapper = wrapper.find(Counter);
    expect(counterWrapper.find('p').text()).toBe('1');
  });

  it('decrements the counter', () => {
    const wrapper = mount(<App />);

    wrapper
      .find('button')
      .at(1)
      .simulate('click');

    const counterWrapper = wrapper.find(Counter);
    expect(counterWrapper.find('p').text()).toBe('-1');
  });
});

describe('Counter', () => {
  ...
});
```

After simulating our click events with Enzyme, we are able to traverse the DOM of the rendered output again to check whether the rendered output has changed. In general that's a good testing practice, because we test how a user interacts with the React components and what's rendered after the interaction took place.

**Exercises:**

- Read more about Enzyme's Rendering Techniques and Selectors

## ENZYME ASYNC TESTING IN REACT

What about testing data fetching in our React component? Fortunately, we can test this behavior with a combination of Jest and Enzyme as well. While Jest takes over for the data fetching, Enzyme makes sure to update our React component accordingly.

How would you implement a fake data fetching request? In JavaScript, promises are used for asynchronous logic. Let's define a promise which will return a result with a delay.

```
const promise = new Promise((resolve, reject) =>
  setTimeout(
    () =>
      resolve({
        data: {
          hits: [
            { objectID: '1', title: 'a' },
            { objectID: '2', title: 'b' },
          ],
        },
      }),
    100
  )
);
```

Once we resolve the promise, we should have the result at our disposal eventually. Now let's take this one step further by using this promise in our new asynchronous test. The basic assumption is that we render our React component, make assertions before the promise resolves, resolve the promise, and make assertions afterward.

```
import React from 'react';
import renderer from 'react-test-renderer';
import { mount } from 'enzyme';

import App, { Counter, dataReducer } from './App';

const list = ['a', 'b', 'c'];
```

```
describe('App', () => {
  describe('Reducer', () => {

    ...
  });

  ...

  it('fetches async data', () => {
    const promise = new Promise((resolve, reject) =>
      setTimeout(
        () =>
          resolve({
            data: {
              hits: [
                { objectID: '1', title: 'a' },
                { objectID: '2', title: 'b' },
              ],
            },
          }),
        100
      )
    );

    const wrapper = mount(<App />);

    expect(wrapper.find('li').length).toEqual(0);

    promise.then(() => {
      expect(wrapper.find('li').length).toEqual(2);
    });
  });
});
```

Next we need to tell our data fetching library, which is used in our App component, to return the desired promise for our test case. This process is called *mocking* when testing implementation logic, because we mimic a different return result from a function. If we wouldn't do it, our data fetching library would make a request to the actual remote API that is used in our App component. But since we want to have control over the returned result, we mock the promise with its result:

```
import React from 'react';
import renderer from 'react-test-renderer';
import { mount } from 'enzyme';
import axios from 'axios';

import App, { Counter, dataReducer } from './App';

const list = ['a', 'b', 'c'];
```

```
describe('App', () => {
  describe('Reducer', () => {
    ...
  });

  ...

  it('fetches async data', () => {
    const promise = new Promise(...);

    axios.get = jest.fn(() => promise);

    const wrapper = mount(<App />);

    expect(wrapper.find('li').length).toEqual(0);

    promise.then(() => {
      expect(wrapper.find('li').length).toEqual(2);

      axios.get.mockClear();
    });
  });
});

describe('Counter', () => {
  ...
});
```

**Important:** Always make sure to clean up your mocks in testing, otherwise another test may run into a mocked function. You can clear mocks in Jest individually, like the previous code snippets has shown it, but also globally by setting the `clearMocks` flag to true in your *jest.config.json* file. This will clear all mocks after every test without leaving any zombie mocks around.

In a perfect world this would already work, but we are not there yet. We need to tell our React component to render again. Fortunately, Enzyme comes with a re-rendering API. In addition, we need to wait for all asynchronous events to be executed before updating our React component and making test assertions. That's where the built-in JavaScript function setImmediate comes in, because it's callback function gets executed in the next iteration of the event loop.

```
import React from 'react';
import renderer from 'react-test-renderer';
import { mount } from 'enzyme';
import axios from 'axios';

import App, { Counter, dataReducer } from './App';
```

```
const list = ['a', 'b', 'c'];

describe('App', () => {
  describe('Reducer', () => {
    ...
  });

  ...

  it('fetches async data', () => {
    const promise = new Promise(...);

    axios.get = jest.fn(() => promise);

    const wrapper = mount(<App />);

    expect(wrapper.find('li').length).toEqual(0);

    promise.then(() => {
      setImmediate(() => {
        wrapper.update();
        expect(wrapper.find('li').length).toEqual(2);

        axios.get.mockClear();
      });
    });
  });
});

describe('Counter', () => {
  ...
});
```

We are almost done. One piece is missing: We need to tell our Jest test runner that we are testing asynchronous logic in our test case. Otherwise, the test will run synchronously and wouldn't wait for the promise to be resolved. Hence, a test case's callback function comes with the handy done callback function that can be used to signalize Jest about a finished test explicitly.

```
import React from 'react';
import renderer from 'react-test-renderer';
import { mount } from 'enzyme';
import axios from 'axios';

import App, { Counter, dataReducer } from './App';

const list = ['a', 'b', 'c'];

describe('App', () => {
  describe('Reducer', () => {
    ...
  });

  ...

  it('fetches async data', done => {
    const promise = new Promise(...);

    axios.get = jest.fn(() => promise);

    const wrapper = mount(<App />);

    expect(wrapper.find('li').length).toEqual(0);

    promise.then(() => {
      setImmediate(() => {
        wrapper.update();
        expect(wrapper.find('li').length).toEqual(2);

        axios.get.mockClear();

        done();
      });
    });
  });
});

describe('Counter', () => {
  ...
});
```

That's it! You have tested asynchronous logic with Jest and Enzyme for a React component with data fetching. There are a few things to take care of, but once you ran through this set up once, you should be able to replicate it for other asynchronous test cases.

Next, we are going to test the "not so happy"-path by testing our error handling in case of a failing data fetching:

```
import React from 'react';
import renderer from 'react-test-renderer';
import { mount } from 'enzyme';
import axios from 'axios';

import App, { Counter, dataReducer } from './App';

const list = ['a', 'b', 'c'];

describe('App', () => {
  describe('Reducer', () => {
    ...
  });
```

```
    });

    ...

    it('fetches async data but fails', done => {
      const promise = new Promise((resolve, reject) =>
        setTimeout(() => reject(new Error('Whoops!')), 100)
      );

      axios.get = jest.fn(() => promise);

      const wrapper = mount(<App />);

      promise.catch(() => {
        setImmediate(() => {
          wrapper.update();

          expect(wrapper.find('li').length).toEqual(0);
          expect(wrapper.find('.error').length).toEqual(1);

          axios.get.mockClear();
          done();
        });
      });
    });
  });

  describe('Counter', () => {
    ...
  });
```

As you can see, the testing pattern is almost identical. We have to mock our result for the data fetching with a promise, render the component, make assertions, wait for the promise, wait for the event loop and the component update, and make more assertions after the asynchronous logic happened. Also we signalise Jest again that our test case has finished.

What's different is that we mock a promise with an error. This way, we can test the error handling of our React component. Also our test assumptions are different here, because instead of expecting a rendered list of items, we expect to find a HTML element with an error CSS class.

**Exercises:**

- Read more about Jest's Mocking API

. . .

The testing tutorial has shown you how Jest and Enzyme can be used perfectly together to snapshot/unit/integration test your React components. You can traverse the DOM of rendered components, mock away and wait for asynchronous logic to happen, and

simulate events on HTML elements to mimic the user's behavior. You can find all the tests written for this tutorial in this GitHub repository.

Continue Reading: How to test Axios in Jest by Example

Continue Reading: End to End Testing React with Cypress

Continue Reading: Test Coverage in JavaScript

Show Comments

## KEEP READING ABOUT TOOLING

### HOW TO TEST REACT WITH JEST

Jest got introduced by Facebook for testing JavaScript and especially React applications. It's one of the most popular ways to test React components nowadays. Since it comes with its own test runner...

### HOW TO USE REACT TESTING LIBRARY TUTORIAL

React Testing Library (RTL) by Kent C. Dodds got released as alternative to Airbnb's Enzyme . While Enzyme gives React developers utilities to test internals of React components, React Testing...

## TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✓ Join 50.000+ Developers

✓ Learn Web Development with JavaScript

✓ Tips and Tricks

✓ Access Tutorials, eBooks and Courses

✓ Personal Development as a Software Engineer

Your email address SUBSCRIBE

View our Privacy Policy.

**PORTFOLIO**

Online Courses

Open Source

Tutorials

**ABOUT**

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms