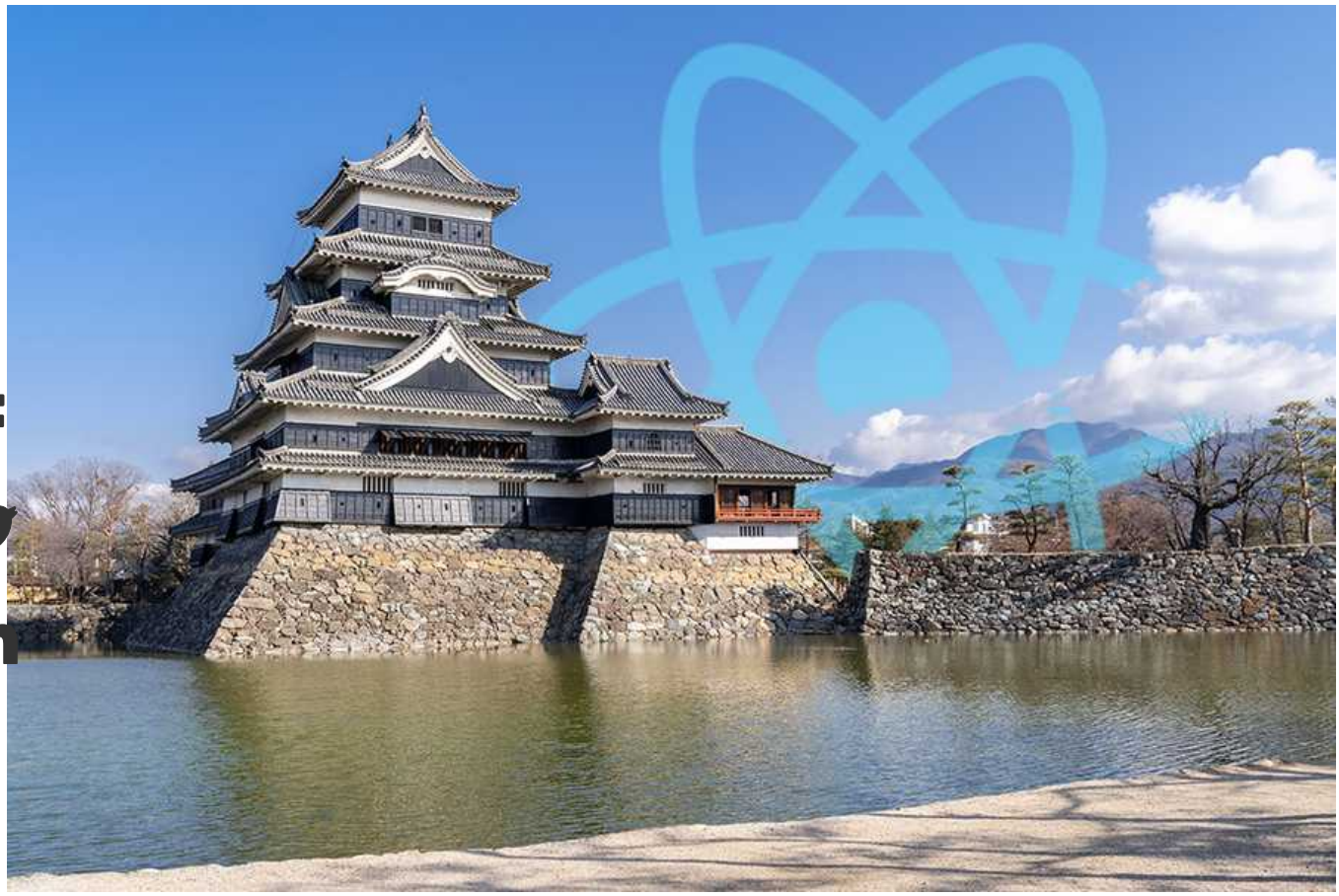
How to test React with Jest

JULY 16, 2019 BY ROBIN WIERUCH - EDIT THIS POST



This tutorial is part 2 of 2 in the series.

Part 1: How to set up React with Webpack and Babel

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, you can simply call Jest from the command line to run all your tests. All your tests are defined as test suites (e.g. `describe`-block) and test cases (e.g. `it`-block or `test`-block).

The Jest setup allows you to add optional configuration, to introduce a setup routine yourself, or to define custom npm scripts to run your Jest tests. In this tutorial, you will learn how to perform all of it. Aside from all the setup, Jest comes with a rich API for test assertions (e.g. true to equal true). The

tutorial will show you how to use these test assertions for your React components and JavaScript functions. Also you will learn about Snapshot Tests to test your React components.

JEST TESTING IN REACT SETUP

Before implementing the test setup and writing the first React component tests, you will need a simple React application which can be tested in the first place. Start with the *src/index.js* file where you can import and render the App component which is not implemented yet:

```
import React from 'react';
import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(<App />, document.getElementById('app'));
```

The App component in your *src/App.js* file will be a React Function Component with React Hooks. It uses axios as third-party library, so make sure to install the node package on the command line with `npm install axios` for your React application.

```
import React from 'react';
import axios from 'axios';

export const dataReducer = (state, action) => {
  if (action.type === 'SET_ERROR') {
    return { ...state, list: [], error: true };
  }

  if (action.type === 'SET_LIST') {
    return { ...state, list: action.list, error: null };
  }

  throw new Error();
};

const initialData = {
  list: [],
  error: null,
};

const App = () => {
  const [counter, setCounter] = React.useState(0);
  const [data, dispatch] = React.useReducer(dataReducer, initialData);

  React.useEffect(() => {
    axios
      .get('http://hn.algolia.com/api/v1/search?query=react')
```

```
      .then(response => {
        dispatch({ type: 'SET_LIST', list: response.data.hits });
      })
      .catch(() => {
        dispatch({ type: 'SET_ERROR' });
      });
  }, []);

  return (
    <div>
      <h1>My Counter</h1>
      <Counter counter={counter} />

      <button type="button" onClick={() => setCounter(counter + 1)}>
        Increment
      </button>

      <button type="button" onClick={() => setCounter(counter - 1)}>
        Decrement
      </button>

      <h2>My Async Data</h2>

      {data.error && <div className="error">Error</div>}

      <ul>
        {data.list.map(item => (
          <li key={item.objectID}>{item.title}</li>
        ))}
      </ul>
    </div>
  );
};

export const Counter = ({ counter }) => (
  <div>
    <p>{counter}</p>
  </div>
);

export default App;
```

The React application is doing two things:

- First, it renders a Counter component which receives props to render a counter property. The counter property is managed as state up in the App component with a useState React Hook. In addition, the counter state can be updated with two buttons by incrementing and decrementing the state.
- Second, the App component fetches data from a third-party API when it's rendered for the first time. Here we are using React's useReducer Hook to manage the data state -- which is either the

actual data or an error. If there is an error, we render an error message. If there is data, we render the data as a list of items in our React component.

Note that we already export our two components and the reducer function from the file to make them testable in our test file(s) later on. This way, every component and the reducer can be tested in isolation -- which makes especially sense for the reducer function to test the state transitions from one to another state. That's what you would call a real unit test: The function is tested with an input and the test asserts an expected output.

In addition, we have a relationship between two React components, because they are parent and child components. That's another scenario which can be tested as integration test. If you would test each component in isolation, you would have unit tests. But by testing them together in their context, for instance rendering the parent component with its child component, you have an integration test for both components.

In order to get our tests up and running, set up Jest by installing it on the command line as development dependencies:

```
npm install --save-dev jest
```

In your *package.json* file, create a new npm script which runs Jest:

```
{
  ...
  "scripts": {
    "start": "webpack serve --config ./webpack.config.js --mode development",
    "test": "jest"
  },
  ...
}
```

In addition, we want to have more configuration in our tests written with Jest. Hence, pass an additional Jest configuration file to your Jest script:

```
{
  ...
  "scripts": {
    "start": "webpack serve --config ./webpack.config.js --mode development",
    "test": "jest --config ./jest.config.json"
  },
  ...
}
```

Next, we can define this optional configuration for Jest in a configuration file. Create it on the command line:

```
touch jest.config.json
```

In this Jest configuration file, add the following test pattern matching to run all the test files which shall be executed by Jest:

```
{
  "testRegex": "((\\.|/*.)(spec))\\.js?$"
}
```

The `testRegex` configuration is a regular expression that can be used to specify the naming of the files where your Jest tests will be located. In this case, the files will have the name `*spec.js`. That's how you can separate them clearly from other files in your *src/* folder. Finally, add a test file next to your App component's file in a new `src/App.spec.js` file. First, create the test file on the command line:

```
touch src/App.spec.js
```

And second, implement your first test case in a test suite in this new file:

```
describe('My Test Suite', () => {
  it('My Test Case', () => {
    expect(true).toEqual(true);
  });
});
```

Now you should be able to run `npm test` to execute your test suites with your test cases. The test should be green (valid, successful) for your previous test case, but if you change the test to something else, let's say `expect(true).toEqual(false);`, it should be red (invalid, failed). Congratulations, you have run your first test with Jest!

Last but not least, add another npm script for watching your Jest tests. By using this command, you can have your tests running continuously in one command line tab, while you start your application in another command line tab. Every time you change source code while developing your application, your tests will run again with this watch script.

```
{
  ...
```

```
  "scripts": {
    "start": "webpack serve --config ./webpack.config.js --mode development",
    "test": "jest --config ./jest.config.json",
    "test:watch": "npm run test -- --watch"
  },
  ...
}
```

Now you can run your Jest tests in watch mode. Doing it this way, you would have one open terminal tab for your Jest tests in watch mode with `npm run test:watch` and one open terminal tab to start your React application with `npm start`. Every time you change a source file, your tests should run again because of the watch mode.

Exercises:

- Read more about getting started with Jest
- Read more about Jest's Globals
- Read more about Jest's Assertions

JEST SNAPSHOT TESTING IN REACT

Jest introduced the so called Snapshot Test. Basically a Snapshot Test creates a snapshot -- which is stored in a separate file -- of your rendered component's output when you run your test. This snapshot is used for diffing it to the next snapshot when you run your test again. If your rendered component's output has changed, the diff of both snapshots will show it and the Snapshot Test will fail. That's not bad at all, because the Snapshot Test should only inform you when the output of your rendered component has changed. In case a Snapshot Test fails, you can either accept the changes or deny them and fix your component's implementation regarding of its rendered output.

By using Jest for Snapshot Tests, you can keep your tests lightweight, without worrying too much about implementation details of the component. Let's see how these work in React. First, install the react-test-renderer utility library commonly used for Jest to render your actual component in your tests:

```
npm install --save-dev react-test-renderer
```

Second, implement your first Snapshot Test with Jest. First, render a component with the new renderer, transform it into JSON, and match the snapshot to the previously stored snapshot:

```
import React from 'react';
```

```
import renderer from 'react-test-renderer';

import { Counter } from './App';

describe('Counter', () => {
  test('snapshot renders', () => {
    const component = renderer.create(<Counter counter={1} />);
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Now run your Jest tests in watch mode again: `npm run test:watch`. Running your tests in watch mode, when having Snapshot Tests in place, gives you the opportunity to run your tests interactively with Jest. For instance, once your watch mode is active, change the div element to a span element in your React component:

```
export const Counter = ({ counter }) => (
  <span>
    <p>{counter}</p>
  </span>
);
```

The command line with the tests running in watch mode should show you a failed Snapshot Test:

```
Counter
    × snapshot renders (21ms)

  ● Counter › snapshot renders

    expect(received).toMatchSnapshot()

    Snapshot name: `Counter snapshot renders 1`

    - Snapshot
    + Received

    - <div>
    + <span>
        <p>
          1
        </p>
    - </div>
    + </span>

Watch Usage: Press w to show more.
```

The previous snapshot doesn't match the new snapshot of the React component anymore. Furthermore, the command line offers you things to do now (optionally you have to hit `w` on your keyboard):

```
Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press u to update failing snapshots.
 › Press i to update failing snapshots interactively.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Pressing `a` or `f` will run all or only your failed tests. If you press `u`, you accept the "failed" test as being valid and the new snapshot of your React component will be stored. If you don't want to accept it as a new snapshot, then fix your test by fixing your component.

```
export const Counter = ({ counter }) => (
  <div>
    <p>{counter}</p>
  </div>
);
```

Afterward, the Snapshot Test should turn green again:

```
 PASS  src/App.spec.js
  Counter
    ✓ snapshot renders (17ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
Time:        4.311s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Anyway, try it yourself by changing the component and either accepting the new snapshot or fixing your React component again. Also add another Snapshot Test for your App component:

```
import React from 'react';
import renderer from 'react-test-renderer';

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

```
describe('App', () => {
  test('snapshot renders', () => {
    const component = renderer.create(<App />);
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});

describe('Counter', () => {
  test('snapshot renders', () => {
    const component = renderer.create(<Counter counter={1} />);
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Most of the time, Snapshot Tests look the same for every React component. You render the component, transform its rendered output to JSON to make it comparable, and match it with the previous snapshot. Having Snapshot Tests in place keeps testing React components more lightweight. Also, Snapshot Tests can be perfectly used to supplement your unit testing and integration tests, because they don't test any implementation logic explicitly.

Note: If you are using Styled Components in React for CSS-in-JS, check out jest-styled-components for testing your CSS style defintions with snapshot tests as well.

Exercises:

- Check your generated *src/**snapshots**/App.spec.js.snap* file
  - Understand why this file exists and how this contributes to diffing snapshots against each other
- Get used to accepting or denying (fixing your component) snapshots
  - Create new React Components and test them with Snapshot Tests
- Read more about Jest Snapshot Testing

JEST UNIT/INTEGRATION TESTING IN REACT

Jest can be used to test your JavaScript logic as integration or unit tests as well. For instance, your App component fetches data and stores the result as state with a reducer function by using a React Hook. This reducer function is exported as standalone JavaScript function which doesn't know anything about React. Thus, there doesn't need to be any rendering for the React component and we can test this reducer function as plain JavaScript function.

```
import React from 'react';
```

```
import renderer from 'react-test-renderer';

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

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

describe('App', () => {
  describe('Reducer', () => {
    it('should set a list', () => {
      const state = { list: [], error: null };
      const newState = dataReducer(state, {
        type: 'SET_LIST',
        list,
      });

      expect(newState).toEqual({ list, error: null });
    });
  });

  ...
});
```

Write two additional tests to cover other parts of your reducer function and edge cases. These two other parts are called the "not so happy"-path, because they don't assume a successful outcome (e.g. data fetching fails). By writing your tests this way, you cover all conditional paths in your application's logic.

```
import React from 'react';
import renderer from 'react-test-renderer';

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

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

describe('App', () => {
  describe('Reducer', () => {
    it('should set a list', () => {
      const state = { list: [], error: null };
      const newState = dataReducer(state, {
        type: 'SET_LIST',
        list,
      });

      expect(newState).toEqual({ list, error: null });
    });

    it('should reset the error if list is set', () => {
      const state = { list: [], error: true };
      const newState = dataReducer(state, {
        type: 'SET_LIST',
        list,
      });
```

```
      expect(newState).toEqual({ list, error: null });
    });

    it('should set the error', () => {
      const state = { list: [], error: null };
      const newState = dataReducer(state, {
        type: 'SET_ERROR',
      });

      expect(newState.error).toBeTruthy();
    });
  });

  ...
});
```

Once you run your tests, you should see the following output on the command line. If a test fails, for instance during watch mode, you will be notified immediately.

```
You should get a similar output:

 PASS  src/App.spec.js
  App
    ✓ snapshot renders (18ms)
    Reducer
      ✓ should set a list (7ms)
      ✓ should reset the error if list is set (1ms)
      ✓ should set the error
  Counter
    ✓ snapshot renders (19ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   2 passed, 2 total
Time:        2.325s
Ran all test suites.

Watch Usage: Press w to show more.
```

You have seen how Jest can also be used to test plain JavaScript functions. It doesn't need to be used for only React. If you have more complex functions in your applications, don't hesitate to extract them as standalone functions which can be exported to make them testable. Then you are always assured that your complex business logic works, because it has been covered by your Jest assertions.

Exercises:

- Explore more Jest Features and how to use them for Snapshot Testing

. . .

Jest gives you (almost) everything you need to test your React components. You can run all your tests from the command line, give it additional configuration, and define test suites and test cases in your test files. Snapshot Tests give you a lightweight way to test your React components by just diffing the rendered output to the previous output. Also you have seen how Jest can be used for testing only JavaScript functions, so it's not strictly bound to React testing.

However, testing the DOM of a React component with Jest is more difficult. That's why there exist other third-party libraries such as React Testing Library or Enzyme to make React component unit testing possible for you. Follow the tutorial series for more testing examples in React.

This tutorial is part 2 of 3 in the series.

Part 1: How to set up React with Webpack and Babel
Part 3: How to test React components with Jest & Enzyme

Continue Reading: How to shallow render Jest Snapshot Tests

Continue Reading: How to Jest Snapshot Test the Difference

Show Comments

Continue Reading: How to test React-Redux connected Components

KEEP READING ABOUT NODE

NODE.JS TESTING WITH MOCHA, CHAI, SINON

This tutorial demonstrates how to setup testing with Mocha, Chai, and Sinon in Node.js. Whereas the previous tutorial has already shown you how to setup your Node.js application, this tutorial sets up...

HOW TO TEST REACT WITH JEST & ENZYME

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...

THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK >

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✔ Join 50.000+ Developers