# How to test React-Redux connected Components

AUGUST 30, 2019 BY ROBIN WIERUCH - EDIT THIS POST



*The following implementation is a quick excerpt from one of my daily tasks as a software developer. If I run into a problem and arrive at an example that I find worth sharing, I will put a gist of the code up on this website. It might be useful for someone else stumbling across the same task.*

React components connected to Redux can turn out pretty complex. Thus most people think that testing these complex components can turn out very complex as well. But it shouldn't be complex at all, if you take full control of the Redux store in your integration/unit test for the React component.

Continue Reading: React Redux Tutorial for Beginners

I will use only Jest in this brief testing tutorial for React Redux components. However, it's up to you to extend your testing setup with React Testing Library or Enzyme for rendering and simulating

events. Jest gives you just the barebones to perform this task.

Continue Reading: How to test React components with Jest

Let's say we have a connected React component that receives state from the Redux store. We call this state -- coming in as props to our component -- `myState`. Our component has also a button element which dispatches an action to our Redux store. We call this action `myAction(payload)` whereas `payload can be any parameters passed to that action. In conclusion, our React component is connected both ways to the Redux store: it receives state (e.g. via mapStateToProps) and dispatches an action (e.g. via mapDispatchToProps).

```
Redux State -> React Component -> Redux Action
```

Imagine following scenario: The React component receives a string in `myState` to populate an HTML input field when it renders. A user can change the value in the input field and once a user clicks the button, the changed value is sent as a payload to `myAction`. Now, we can test both ends of the connected React component with two test cases in one test suite:

```
describe('My Connected React-Redux Component', () => {
  it('should render with given state from Redux store', () => {

  });

  it('should dispatch an action on button click', () => {

  });
});
```

In order to get full control over the Redux store, we will use a Redux specific testing library called Redux Mock Store. If you haven't installed it yet, you can do so on the command line:

```
npm install redux-mock-store --save-dev
```

Because of this mocking, we haven't full confidence that our component will work in integration with a non mocked Redux store, however, other tests of yours should ensure that all actions/reducers/sagas within the actual Redux store work as expected. That's why we will mock the Redux store for this case and run only unit tests against our connected react-redux component.

```
Redux State (Mock) -> React Component (Unit Test) -> Redux Action (Mock)
```

Let's see how we can set up the Redux mock store in our unit test:

```
import configureStore from 'redux-mock-store';

const mockStore = configureStore([]);

describe('My Connected React-Redux Component', () => {
  let store;

  beforeEach(() => {
    store = mockStore({
      myState: 'sample text',
    });
  });

  it('should render with given state from Redux store', () => {

  });

  it('should dispatch an action on button click', () => {

  });
});
```

Everything you pass into `mockStore` will be your Redux store's initial state. So make sure you provide everything that's needed by your connected React component to render without any problems. Next, create the React component with a renderer of your choice for your test:

```
import React from 'react';
import { Provider } from 'react-redux';
import renderer from 'react-test-renderer';
import configureStore from 'redux-mock-store';

import MyConnectedComponent from '.';

const mockStore = configureStore([]);

describe('My Connected React-Redux Component', () => {
  let store;
  let component;

  beforeEach(() => {
    store = mockStore({
      myState: 'sample text',
    });

    component = renderer.create(
      <Provider store={store}>
        <MyConnectedComponent />
      </Provider>
    );
```

```
  });

  it('should render with given state from Redux store', () => {

  });

  it('should dispatch an action on button click', () => {

  });
});
```

You can see how the mocked Redux store is used in the wrapping Provider from the actual react-redux library. Thus, the mocked Redux store is provided for your React component for the purpose of this test. For your first unit test, the simplest thing you can do is performing a snapshot test of the rendered component:

```
...

describe('My Connected React-Redux Component', () => {
  let store;
  let component;

  beforeEach(() => {
    store = mockStore({
      myState: 'sample text',
    });

    component = renderer.create(
      <Provider store={store}>
        <MyConnectedComponent />
      </Provider>
    );
  });

  it('should render with given state from Redux store', () => {
    expect(component.toJSON()).toMatchSnapshot();
  });

  it('should dispatch an action on button click', () => {

  });
});
```

Check the snapshot test's output whether everything got rendered as expected with the given state from the mocked Redux store. Of course, you can be more explicit in this test case by not only checking the rendered snapshot, but also by checking explicitly whether certain elements have been rendered with the given state from the Redux store. For instance, you could check whether a given HTML input field receives the state from the Redux store as its initial state.

Now, for your second unit test, you will check with Jest whether a HTML button click will dispatch a specific Redux action. Therefore, you will have to introduce a spy for the Redux store's dispatch function and you will have to simulate a click event on the given button. As said before, it's up to you how to spy a function and how to simulate an event -- in our case, we will use Jest for both cases:

```
...

import MyConnectedComponent from '.';
import { myAction } from './actions'

...

describe('My Connected React-Redux Component', () => {
  let store;
  let component;

  beforeEach(() => {
    store = mockStore({
      myState: 'sample text',
    });

    store.dispatch = jest.fn();

    component = renderer.create(
      <Provider store={store}>
        <MyConnectedComponent />
      </Provider>
    );
  });

  it('should render with given state from Redux store', () => {
    expect(component.toJSON()).toMatchSnapshot();
  });

  it('should dispatch an action on button click', () => {
    renderer.act(() => {
      component.root.findByType('button').props.onClick();
    });

    expect(store.dispatch).toHaveBeenCalledTimes(1);
    expect(store.dispatch).toHaveBeenCalledWith(
      myAction({ payload: 'sample text' })
    );
  });
});
```

With Jest, we simulate a click event on the button and expect the Redux store's dispatch function to have been called one time with the returned values from our desired Redux action.

**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.

. . .

**Bonus:** If you need to simulate other events in between, for instance to fill up a form, you can simply do so:

```
describe('My Connected React-Redux Component', () => {
  ...

  it('should dispatch an action on button click', () => {
    renderer.act(() => {
      component.root.findByType('button').props.onClick();
    });

    renderer.act(() => {
      component.root.findByType('input')
        .props.onChange({ target: { value: 'some other text' } });
    });

    expect(store.dispatch).toHaveBeenCalledTimes(1);
    expect(store.dispatch).toHaveBeenCalledWith(
      myAction({ payload: 'some other text' })
    );
  });
});
```

In this case, for instance, we assume that the input field updates internal state of the component and once a button is clicked, this state, in this case the "some other text" value, gets send as dispatched action to the mocked Redux store.

. . .

Ultimately, that's already it for testing the second part of the connected react-redux component:

- 1) Provide State -> React Component (Unit Test) => Component Renders
- 2) React Component (Unit Test) -> Simulate Event => Dispatch Action Triggers

There are many ways to test connected React components that know about the Redux store. Using a Jest Mock for functions (e.g. Redux dispatch function) and a Redux Store Mock for faking the received state are only one way for unit testing these kind of components. Other approaches try to fully integrate their Redux store into their testing equation or to Jest mock the react-redux connect higher-order component. Anyway, you can add this learned testing method to your tool belt of unit testing best practices for React now.

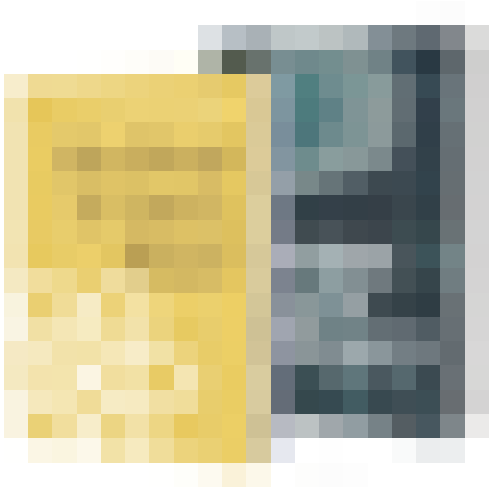
## KEEP READING ABOUT REACT›

### REACT HOOKS MIGRATION

React Hooks were introduced to React to make state and side-effects available in React Function Components. Before it was only possible to have these in React Class Components; but since React's way...

### 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...



### 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
- ✔ Learn Web Development
- ✔ Learn JavaScript
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

SUBSCRIBE ›

View our Privacy Policy.

---

**PORTFOLIO** **ABOUT**

Online Courses
Open Source
Tutorials
About me
What I use
How to work with me
How to support me



Contact Me Privacy & Terms