Testing your app with Jest

Learn how to test your React/Redux app thoroughly, including actions, reducers and components!

WE'LL COVER THE FOLLOWING

Testing apps is something that a lot of developers *should* be doing, but a lot of them don't. It has a bunch of really nice benefits:

1. **You catch bugs before they happen.** The obvious out of the bunch, but nontheless very important. We all make mistakes, and with an automated test suite (an array of tests) we make sure no users has to ever see those!
2. **Tests are executable documentation.** This is in my opinion the biggest benefit. With a function that is well tested you can immediately figure out what it's used for, how you should use it and what to expect it to do! No more tedious documentation writing.
3. **Tests save time.** This might be a bit counterintuitive, since writing tests takes away time you'd spend writing your app. But then you still have to do Q&A, manual testing to make sure you didn't break something else with a change! Automated tests save so much time that would be spent manually doing Q&A and finding bugs!
4. **You will write better code.** Some code is harder to test, some easier. You'll start writing easier testable code, which automatically is better code!

With this in mind, there's no way we could not test our application! Let's get started!

Unit testing

Unit testing is the practice of testing the smallest possible *units* of our code. In JavaScript, those are functions. We run our tests and automatically verify that our functions do the thing we expect them to do. We assert that, given a set of inputs, our functions return the proper values and handle problems.

We'll be using the Jest test framework by facebook. It was written to help test react apps, and is perfect for that purpose! It makes writing tests as easy as speaking - you `describe` a unit of your code and `expect` `it` to do the correct thing.

Thankfully, `create-react-app` comes with it installed by default so we won't need to do any setup! Simply enter `npm run test` into your terminal to run the tests we'll write below.

> Note: **You need version 0.3.0 or higher of `create-react-app` for Jest support.** To check which version you have installed look into the `package.json` of the `weather-app` folder. If it's lower than `0.3.0` follow the upgrade guide to get the testing setup.

Basics

For the sake of this guide, lets pretend we're testing this function. It's situated in the `src/add.js` file:

```
// src/add.js

export function add(x, y) {
  return x + y;
}
```

Jest

Jest is our unit testing framework. Its API, which we write tests with, is speech like and easy to use.

> Note: This is the official documentation of Jest.

We're going to add a second file called `add.test.js` in a subfolder called `src/__tests__/` with our unit tests inside. Running said unit tests requires us to enter `npm run test -- src/__tests__/add.test.js` into the command line.

First, we `import` the function in our `add.test.js` file:

```
// src/__tests__/add.test.js

import { add } from '../add.js';
```

Second, we `describe` our function:

```
describe('add()', function() {

});
```

Third, we tell Jest what `it` (our function) should do:

```
describe('add()', function() {
  it('adds two numbers', function() {

  });

  it('doesnt add the third number', function() {

  });
});
```

Now we have to `expect` our little function to return the same thing every time given the same input. We're going to test that our little function correctly adds two numbers first. We are going to take some chosen inputs, and `expect` the result `toEqual` the corresponding output:

```
// [...]
it('adds two numbers', function() {
  expect(add(2, 3)).toEqual(5);
});
// [...]
```

Lets add the second test, which determines that our function doesn't add the

third number if one is present:

```
// [...]
it('doesnt add the third number', function() {
 expect(add(2, 3, 5)).toEqual(add(2, 3));
});
// [...]
```

> Note: Notice that we call `add` in `toEqual`. I won't tell you why, but just think about what would happen if we rewrote the expect as `expect(add(2, 3, 5)).toEqual(5)` and somebody broke something in the add function. What would this test actually... test?

Should our function work, Jest will show this output when running the tests:

```
PASS  src/__tests__/add.test.js (0.537s)
2 tests passed (2 total in 1 test suite, run time 0.557s)
```

Lets say an unnamed colleague of ours breaks our function:

```
// add.js

export function add(x, y) {
  return x * y;
}
```

Oh no, now our function doesn't add the numbers anymore, it multiplies them! Imagine the consequences to our code that uses the function!

Thankfully, we have unit tests in place. Because we run the unit tests before we deploy our application, we see this output:

```
 FAIL  src/__tests__/add.test.js (0.535s)
● add() › it adds two numbers
  - Expected 6 to equal 5.
        at Object.<anonymous> (__tests__/add.test.js:5:65)
1 test failed, 1 test passed (2 total in 1 test suite, run time 0.564s)
```

This tells us that something is broken in the add function before any users get the code! Congratulations, you just saved time and money!

Redux

The nice thing about Redux is that it makes our data flow entirely consist of “pure” functions. Pure functions are functions that return the same output with the same input everytime – they don’t have any side effects!

Let’s test our actions first!