# Unit Testing

A robust test suite is a vital constituent of quality software. With a good test suite at his or her disposal, a developer can more confidently refactor or add features to an application. Test suites are an upfront investment that pay dividends over the lifetime of a system.

Testing user interfaces is notoriously difficult. Thankfully, testing React components is not. With the right tools and the right methodology, the interface for your web app can be just as fortified with tests as every other part of the system.

We’ll begin by writing a small test suite without using any testing libraries. After getting a feel for a test suite’s essence, we’ll introduce the Jest testing framework to alleviate a lot of boilerplate and easily allow our tests to be much more expressive.

While using Jest, we’ll see how we can organize our test suite in a behavior-driven style. Once we’re comfortable with the basics, we’ll take a look at how to approach testing React components in particular. We’ll introduce Enzyme, a library for working with React components in a testing environment.

Finally, in the last section of this module, we work with a more complex React component that sits inside of a larger app. We use the concept of a mock to isolate the API-driven component we are testing.

## Writing tests without a framework

 If you’re already familiar with JavaScript testing, you can skip ahead to the next section.

However, you might still find this section to be a useful ref lection on what testing frameworks are doing behind the scenes.

The projects for this module are located inside of the folder testing that was provided with this course’s code download. We’ll start in the basics folder:

$ cd testing/basics

The structure of this project:

$ dir

Modash.js Modash.test.js complete/ package.json

Inside of complete/ you’ll find files corresponding to each iteration of Modash.test.js as well as the completed version of Modash.js.

We’ll be using babel-node to run our test suite from the command-line. babel-node is included in this folder’s package.json.

Go ahead and install the packages in package.json now:

$ npm install

In order to write tests, we need to have a library to test. Let’s write a little utility library that we can test.

### Preparing Modash

We’ll write a small library in Modash.js. Modash will have some methods that might prove useful when working with JavaScript strings. We’ll write the following three methods. Each returns a string:

**truncate(string, length)**

Truncates string if it’s longer than the supplied length. If the string is truncated, it will end with ...:

**const** s = 'All code and no tests makes Jack a precarious boy.';

Modash.truncate(s, 21);

*// => 'All code and no tests...'*

Modash.truncate(s, 100);

*// => 'All code and no tests makes Jack a precarious boy.'*

**capitalize(string)**

Capitalizes the first letter of string and lower cases the rest:

**const** s = 'stability was practically ASSURED.';

Modash.capitalize(s);

*// => 'Stability was practically assured.'*

**camelCase(string)**

Takes a string of words delimited by spaces, dashes, or underscores and returns a camel-cased representation:

**let** s = 'started at';

Modash.camelCase(s);

*// => 'startedAt'*

s = 'started\_at'; Modash.camelCase(s);

*// => 'startedAt'*

We’ll write Modash as an ES6 module.

Open up Modash.js now. We’ll write our library’s three functions then export our interface at the bottom of this file.

First, we’ll write the function for truncate(). There are many ways to do this. Here’s one approach:

**testing/basics/complete/Modash.js**

**function** truncate(string, length) {

**if** (string.length > length) {

**return** string.slice(0, length) + '...';

} **else** {

**return** string;

}

}

Next, here’s the implementation for capitalize():

**testing/basics/complete/Modash.js**

**function** capitalize(string) {

**return** (

string.charAt(0).toUpperCase() + string.slice(1).toLowerCase() );

}

Finally, we’ll write camelCase(). This one’s slightly trickier. Again, there are multiple ways to implement this but here’s the strategy that follows:

1. Use split to get an array of the words in the string. Spaces, dashes, and underscores will be considered delimiters.
2. Create a new array. The first entry of this array will be the lower-cased version of the first word. The rest of the entries will be the capitalized version of each subsequent word.
3. Join that array with join.

That looks like this:

**testing/basics/complete/Modash.js**

**function** camelCase(string) {

**const** words = string.split(/[\s|\-|\_]+/); **return** [

words[0].toLowerCase(),

...words.slice(1).map((w) => capitalize(w)),

].join('');

}

With those three functions defined in Modash.js, we’re ready to export our module.

At the bottom of Modash.js, we first create the object that encapsulates our methods:

**testing/basics/complete/Modash.js**

**const** Modash = { truncate, capitalize, camelCase,

};

And then we export it:

**testing/basics/complete/Modash.js**

**export default** Modash;

We’ll write our testing code for this section inside of the file Modash.test.js. Open up that file in your text editor now.

##### ES6: Import/export with Babel

Our package.json already includes Babel. In addition, we’re including a Babel plug-in,

babel-plugin-transform-es2015

This package will let us use the ES6 import/export syntax. Importantly, we specify it as a Babel plugin

inside the project’s .babelrc:

*// basics/.babelrc*

{

"plugins": ["transform-es2015-modules-commonjs"]

}

With this plugin in place, we can now export a module from one file and import it into another.

However, note that this solution won’t work in the browser. It works locally in the Node runtime, which is fine for the purposes of writing tests for our Modash library. But to support this in the browser requires additional tooling. As we mentioned in the last module, ES6 module support in the browser is one of our primary motivations for using Webpack.

### Writing the first spec

Our test suite will import the library we’re writing tests for, Modash. We’ll call methods on that library and make assertions on how the methods should behave.

At the top of Modash.test.js, let’s first import our library:

**testing/basics/complete/Modash.test-1.js**

**import** Modash from './Modash';

Our first assertion will be for the method truncate. We’re going to assert that when given a string over the supplied length, truncate returns a truncated string.

First, we setup the test:

**testing/basics/complete/Modash.test-1.js**

**const** string = 'there was one catch, and that was CATCH-22';

**const** actual = Modash.truncate(string, 19);

**const** expected = 'there was one catch...';

We’re declaring our sample test string, string. We then set two variables: actual and expected. In test suites, actual is what we call the behavior that was observed. In this case, it’s what Modash.truncate actually returned. expected is the value we are expecting.

Next, we make our test’s assertion. We’ll print a message indicating whether truncate passed or failed:

**testing/basics/complete/Modash.test-1.js**

**if** (actual !== expected) {

console.log(

`[FAIL] Expected \`truncate()\` to return '**${**expected**}**', got '**${**actual**}**'` );

} **else** {

console.log('[PASS] `truncate()`.');

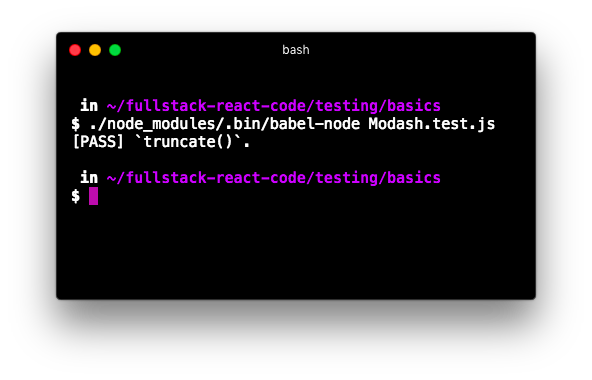
}

###### Try it out

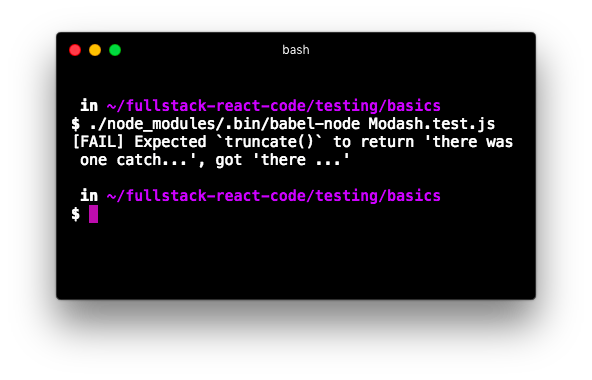
We can run our test suite at this stage in the command line. Save the file Modash.test.js and run the following from the testing/basics folder:

./node\_modules/.bin/babel-node Modash.test.js

Executing this, we see a [PASS] message printed to the console. If you’d like, you can modify the truncate function in Modash.js to observe this test failing:



Test passing



Example of what it looks like when test fails

### The assertEqual() function

Let’s write some tests for the other two methods in Modash.

For all our tests, we’re going to be following a similar pattern. We’re going to have some assertion that checks if actual equals expected. We’ll print a message to the console that indicates whether the function under test passed or failed.

To avoid this code duplication, we’ll write a helper function, assertEqual(). assertEqual() will check equality between both its arguments. The function will then write a console message, indicating if the spec passed or failed.

At the top of Modash.test.js, below the import statement for Modash, declare assertEqual:

**testing/basics/complete/Modash.test-2.js**

import Modash from './Modash';

**function assertEqual(description, actual, expected) {**

**if (actual === expected) {**

**console.log(`[PASS] ${description}`);**

**} else {**

**console.log(`[FAIL] ${description}`);**

**console.log(`\tactual: '${actual}'`);**

**console.log(`\texpected: '${expected}'`); }**

**}**

With assertEqual defined, let’s re-write our first test spec. We’re going to re-use the variables actual, expected, and string throughout the test suite, so we’ll use the let declaration so that we can redefine them:

**testing/basics/complete/Modash.test-2.js**

**let** actual;

**let** expected;

**let** string;

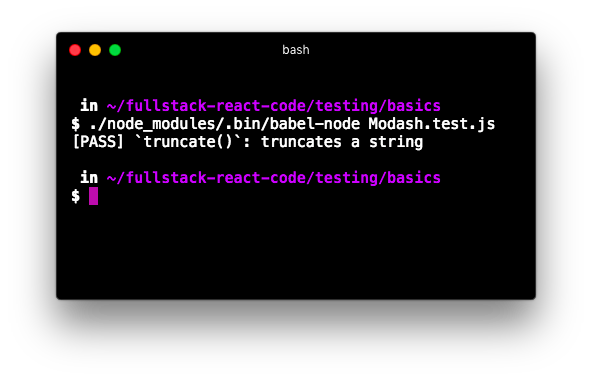
string = 'there was one catch, and that was CATCH-22';

actual = Modash.truncate(string, 19);

expected = 'there was one catch...';

assertEqual('`truncate()`: truncates a string', actual, expected);

If you were to run Modash.test.js now, you’d note that things are working just as before. The console output is just slightly different:



Test passing

With our assert function written, let’s write some more tests.

Let’s write one more assertion for truncate. The function should return a string as-is if it’s less than the supplied length. We’ll use the same string. Write this assertion below the current one:

**testing/basics/complete/Modash.test-2.js**

actual = Modash.truncate(string, string.length);

expected = string;

assertEqual('`truncate()`: no-ops if <= length', actual, expected);

Next, let’s write an assertion for capitalize. We can continue to use the same string:

**testing/basics/complete/Modash.test-2.js**

actual = Modash.capitalize(string);

expected = 'There was one catch, and that was catch-22'; assertEqual('`capitalize()`: capitalizes the string', actual, expected);

Given the example string we’re using, this assertion tests both aspects of capitalize: That it capitalizes the first letter in the string and that it converts the rest to lowercase.

Last, we’ll write our assertions for camelCase. We’ll test this function with two different strings. One will be delimited by spaces and the other by underscores.

The assertion for spaces:

**testing/basics/complete/Modash.test-2.js**

string = 'customer responded at';

actual = Modash.camelCase(string);

expected = 'customerRespondedAt';

assertEqual('`camelCase()`: string with spaces', actual, expected);

And for underscores:

**testing/basics/complete/Modash.test-2.js**

string = 'customer\_responded\_at';

actual = Modash.camelCase(string);

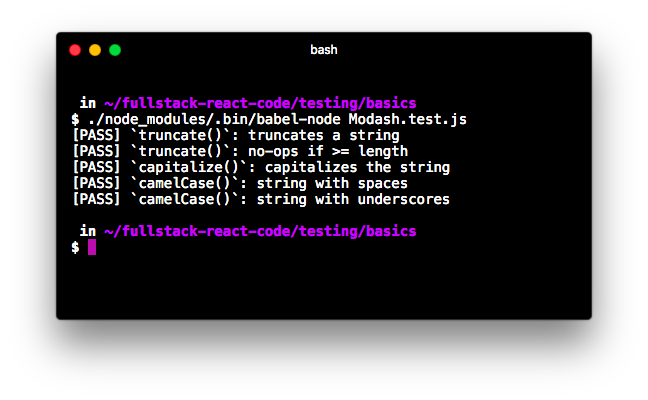
expected = 'customerRespondedAt';

assertEqual('`camelCase()`: string with underscores', actual, expected);

Try it out

Save Modash.test.js. From the console, run the test suite:

./node\_modules/.bin/babel-node Modash.test.js



Tests passing

Feel free to tweak either the expected values for each assertion or break the library and watch the tests fail.

Our miniature assertion framework is clear but limited. It’s hard to imagine how it would be both maintainable and scalable for a more complex app or module. And while assertEqual() works fine for checking the equality of strings, we’ll want to make more complex assertions when working with objects or arrays. For instance, we might want to check if an object contains a particular property or an array a particular element.

## Using Jest

With Jest, like in many other test frameworks, we organize our code into describe blocks and it blocks.

To get a feel for this organization, let’s write our first Jasmine test. Replace the contents of Modash.test.js with the following:

**testing/basics/complete/Modash.test-3.js**

describe('My test suite', () => {

it('`true` should be `true`', () => {

expect(**true**).toBe(**true**);

});

it('`false` should be `false`', () => {

expect(**false**).toBe(**false**);

});

});

Both describe and it take a string and a function. The string is just a human-friendly description, which we’ll see printed to the console in a moment.

As we’ll see throughout this module, describe is used to organize assertions that all pertain to the same feature or context. it blocks are our individual assertions or specs.

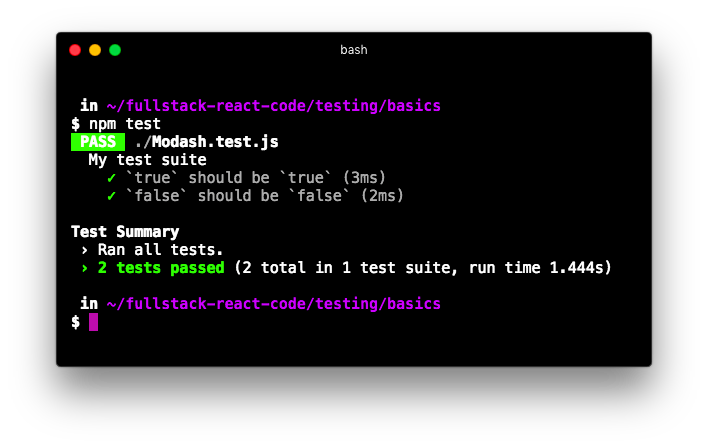
Jest requires that you always have a top-level describe that encapsulates all your code. Here, our top-level describe is titled “My test suite.” The two it blocks nested inside of this describe are our specs. This organization is standard: describe blocks don’t contain assertions, it blocks do.

Throughout the rest of this module, an “assertion” refers to a call to expect(). A “spec” is an it block.

###### Try it out

Inside of package.json, we already have a test script defined. So we can run the following command to run our test suite:

$ npm test



Both tests passing

### The first Jest test for Modash

Let’s replace this test suite with something useful that tests Modash.

Open Modash.test.js again and clear it out. At the top, import the library:

**testing/basics/complete/Modash.test-4.js**

**import** Modash from './Modash';

We’ll title our describe block 'Modash':

describe('Modash', () => {

*// assertions will go here*

});

It’s conventional to title the top-level describe whatever module is currently under test.

Let’s make our first assertion. We’re asserting that truncate() works:

**testing/basics/complete/Modash.test-4.js**

describe('Modash', () => {

it('`truncate()`: truncates a string', () => {

**const** string = 'there was one catch, and that was CATCH-22'; expect(

Modash.truncate(string, 19)

).toEqual('there was one catch...');

});

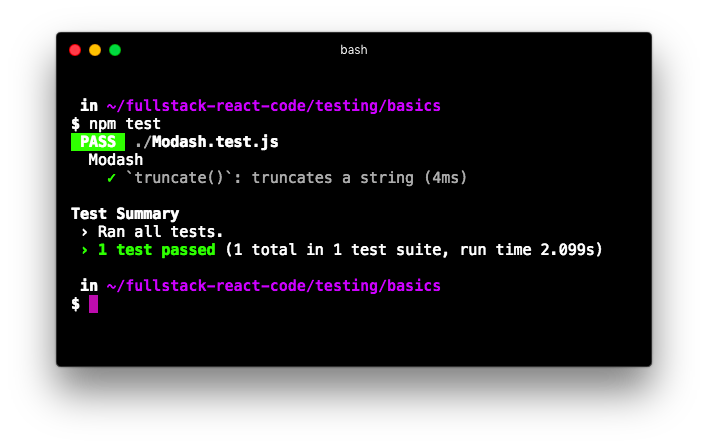
});

We organized our assertion differently, but the logic and end result are the same as before. Note how expect and toEqual provide a human-readable format for expressing what we are testing and how we expect it to behave.

###### Try it out

Save Modash.test.js. Run the single-spec test suite:

$ npm test



Test passing

### The other truncate() spec

We have a second assertion for truncate(). We assert that truncate() returns the same string if it’s below the specified length.

Because both of these assertions correspond to the same method on Modash, it makes sense to wrap them together inside their own describe.

Let’s add the next spec, wrapping both our specs inside of a new describe:

**testing/basics/complete/Modash.test-5.js**

describe('Modash', () => {

describe('`truncate()`', () => {

**const** string = 'there was one catch, and that was CATCH-22';

it('truncates a string', () => { expect(

Modash.truncate(string, 19)

).toEqual('there was one catch...');

});

it('no-ops if <= length', () => { expect(

Modash.truncate(string, string.length)

).toEqual(string);

});

});

});

It’s conventional to group tests using describe blocks like this.

**Note** that we declared the string under test at the top of the truncate() describe block:

**testing/basics/complete/Modash.test-5.js**

describe('Modash', () => {

describe('`truncate()`', () => {

**const** string = 'there was one catch, and that was CATCH-22';

When variables are declared inside describe in this manner, they are in scope for each of the it blocks.

Furthermore, we slightly changed the title of each spec. We were able to drop the truncate(): at the beginning. Because these specs are under the describe block titled 'truncate()', if one of these specs were to fail Jest would present the failure like this:

- Modash > `truncate()` > no-ops **if** less than length

This gives us all the context we need.

### The rest of the specs

We’ll wrap the specs for our other two methods inside their own describe blocks, like this:

describe('Modash', () => {

describe('`truncate()`', () => { *// ... `truncate()` specs*

}); describe('`capitalize()`', () => { *// ... `capitalize()` specs*

});

describe('`camelCase()`', () => { *// ... `camelCase()` specs*

});

});

First, our capitalize() spec:

**testing/basics/complete/Modash.test-6.js**

describe('capitalize()', () => {

it('capitalizes first letter, lowercases rest', () => {

**const** string = 'there was one catch, and that was CATCH-22';

expect(

Modash.capitalize(string)

).toEqual(

'There was one catch, and that was catch-22' );

});

});

Note that the string inside the truncate()describe block is not in scope here, so we declare string at the top of this spec.

Last, our set of camelCase() specs:

**testing/basics/complete/Modash.test-6.js**

describe('camelCase()', () => {

it('camelizes string with spaces', () => {

**const** string = 'customer responded at';

expect(

Modash.camelCase(string)

).toEqual('customerRespondedAt');

});

it('camelizes string with underscores', () => {

**const** string = 'customer\_responded\_at';

expect(

Modash.camelCase(string)

).toEqual('customerRespondedAt');

});

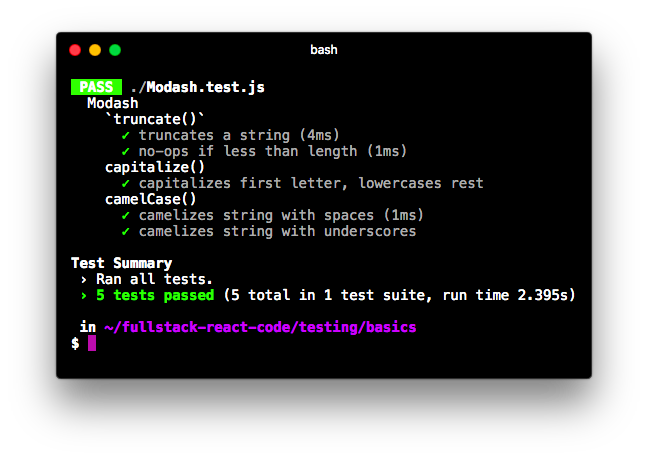
});

###### Try it out

Save Modash.test.js. Fire up Jest from the command-line:

$ npm test

And you’ll see everything pass:



Tests passing

We’ve covered the basics of assertions, organizing code into describe and it blocks, and using the Jest test runner. Let’s see how these pieces come together for testing React applications. Along the way, we’ll dig even deeper into Jest’s assertion library and best practices for behavior-driven test suite organization.

## Testing a basic React component with Enzyme

We’ll get familiar with Enzyme by writing tests for a basic React component.

### Setup

Inside the folder testing/react-basics is an app created with create-react-app. From the testing/basics folder, cd into that directory:

$ cd ../react-basics

And install the packages:

$ npm i

Take a look at the directory:

$ dir

public

node\_modules/ package.json

src/

And src/:

$ dir src/

App.css

App.js

App.test.js

complete

index.css

index.js

semantic-ui

setupTests.js tempPolyfills.js

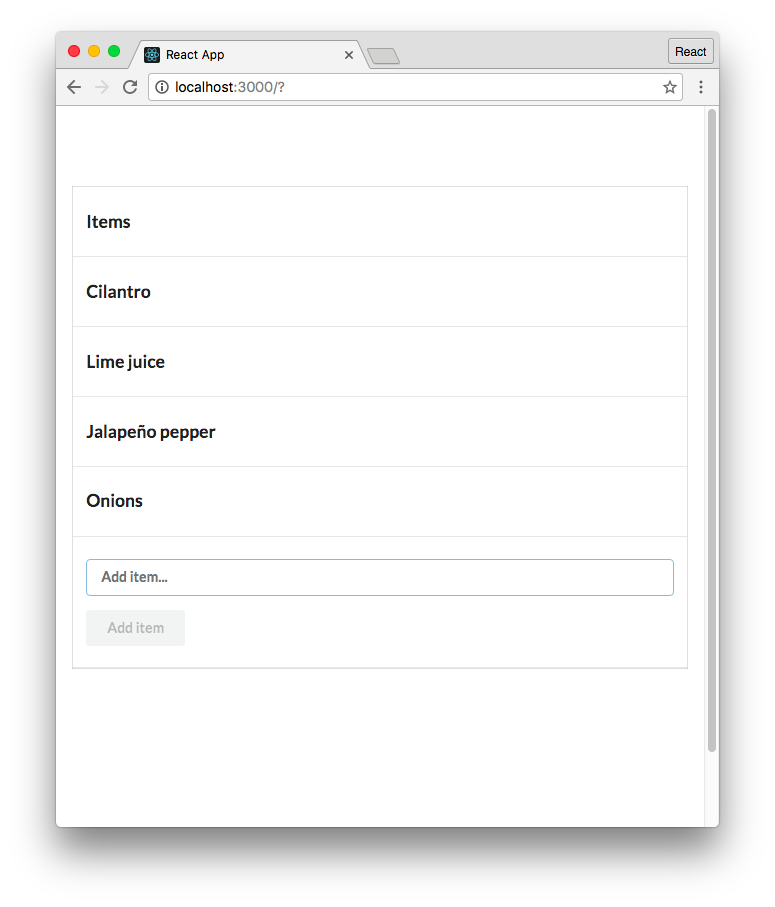
The basic organization of this create-react-app app is the same that we saw in the last module. App.js defines an App component. index.js calls ReactDOM.render(). Semantic UI is included for styling.

### The App component

Before looking at App, let’s see it in the browser. Boot the app:

$ npm start

The app is simple. There is a field coupled with a button that adds items to a list. There is no way to delete items in the list:



The completed list app

Open up App.js. As we see in the initialization of state, App has two state properties:

**testing/react-basics/src/App.js**

**class** App **extends** React.Component {

state = { items: [], item: '',

};

items is the list of items. item is the state property that is tied to our controlled input, which we’ll see in a moment.

Inside of render(), App iterates over this.state.items to render all items in a table:

<tbody>

{ **this**.state.items.map((item, idx) => (

<tr key={idx}

>

<td>{item}</td>

</tr>

))

}

</tbody>

The controlled input is standard. It resides inside of a form:

**testing/react-basics/src/App.js**

<form

className='ui form' onSubmit={**this**.addItem}

>

<div className='field'>

<input

className='prompt' type='text'

placeholder='Add item...' value={**this**.state.item} onChange={**this**.onItemChange}

/>

For the input, onItemChange() sets item in state as expected:

onItemChange = (e) => { **this**.setState({

item: e.target.value,

});

};

For the form, onSubmit calls addItem(). This function adds the new item to state and clears item:

**testing/react-basics/src/App.js**

addItem = (e) => {

e.preventDefault();

**this**.setState({

items: **this**.state.items.concat(

**this**.state.item

),

item: '',

});

};

Finally, the button:

**testing/react-basics/src/App.js**

<button

className='ui button' type='submit' disabled={submitDisabled}

>

Add item

</button>

We set the attribute disabled on the button. This variable (submitDisabled) is defined at the top of render and depends on whether or not the input field is populated:

render() {

**const** submitDisabled = !**this**.state.item;

**return**(

### The first spec for App

In order to write our first spec, we need to have two libraries in place: Jest and Enzyme.

In the last module, we noted that create-react-app sets up a few commands in package.json. One of those was test.

react-scripts already specifies Jest as a dependency. To boot Jest, we just need to run npm test. Like other commands that create-react-app creates for us, test runs a script in react-scripts. This script configures and executes Jest.

To see all of the packages that react-scripts includes, see the file ./node\_modules/react-scripts/package.json.

create-react-app sets up a dummy test for us in App.test.js.

Let’s execute Jest from inside testing/react-basics and see what happens:

$ npm test

Jest runs, emitting a well-formatted report of our test suite’s results:



The sample test run

react-scripts has provided some additional configuration to Jest. One configuration is booting Jest in watch mode. In this mode, Jest does not quit after the test suite finishes. Instead, it watches the whole project for changes. When a change is detected, it re-runs the test suite.

Throughout this module, we’ll continue to instruct you to execute the test suite with npm test. However, you can just keep a console window open with Jest running in watch mode if you’d like.

###### Setting up Enzyme

In order to use Enzyme, we need to:

1. Ensure we install all the required packages
2. Include instructions for Enzyme indicating which React adapter to use
3. Include a patch for React 16

We’ll dig into each of these in turn.

1. Ensure we install all the required packagesreact-scripts does not include enzyme. So we’ve included it in our package.json.

enzyme wraps react-test-renderer. As a result, it depends on that package to be installed too. You’ll see that dependency in the package.json as well.

Further, we need to include an adapter for Enzyme to use. Enzyme has adapters for each version of React. Because we’re using React 16, we include the React 16 adapter in our package.json, enzymeadapter-react-16.

 In the future, if you want to add all these dependencies to your project simply run:

npm i --save-dev enzyme react-test-renderer enzyme-adapter-react-16

1. Include instructions for Enzyme indicating which React adapter to use

Before running our test suite, we need to instruct Enzyme to use the React 16 adapter. Here’s what that instruction looks like:

**import** Enzyme from 'enzyme';

**import** Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: **new** Adapter() });

We could include this snippet at the top of each spec file, but this would quickly become redundant if we were to add more spec files for more components.

Instead, we can create a file called setupTests.js inside the src/ directory. Create React App configures Jest so that this file is loaded automatically before each test suite run.

Taking a look inside src/, we can see this file is already present:

**testing/react-basics/src/setupTests.js**

**import** raf from './tempPolyfills'

**import** Enzyme from 'enzyme';

**import** Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: **new** Adapter() });

This file includes the snippet instructing Enzyme to use the React 16 adapter. But what is the line importing tempPolyfills.js all about?

3. Include a patch for React 16

The underlying architecture of React 16 depends on a browser API called requestAnimationFrame. This is included in the JavaScript environment of modern browsers.

When running React in our test environment, this browser API will not be present. As a result, React will throw an error.

At the time of writing, React 16 was just released. [There is an on-going discussion](https://github.com/facebookincubator/create-react-app/issues/3199)[[1]](#footnote-1) about how to handle this missing API in test environments. Newer versions of Create React App will likely handle this situation automatically.

In the meantime, we’ve included a workaround. Peeking inside tempPolyfills.js, we can see that we’ve defined a dummy requestAnimationFrame API:

**testing/react-basics/src/tempPolyfills.js**

**const** raf = global.requestAnimationFrame = (cb) => {

setTimeout(cb, 0);

};

**export default** raf;

This fallback or polyfill for requestAnimationFrame is sufficient for React to operate as expected in a testing environment.

Again, at the time of writing React 16 is still very new. Hopefully some of this ceremony for setting up Enzyme and React for tests will be reduced in upcoming releases. If you’d like, you can refer to this ticke[t7](https://github.com/facebookincubator/create-react-app/issues/3199)1 to see if the polyfill is still necessary for new projects.

###### Writing the spec

With Enzyme set up, we’re ready to replace the spec in App.test.js with something more useful.

Open up App.test.js and clear out the file. At the top of that file, we first import the React component that is under test:

**testing/react-basics/src/complete/App.test.complete-1.js**

**import** App from './App';

Next, we’ll import React from react and shallow() from enzyme:

**testing/react-basics/src/complete/App.test.complete-1.js**

**import** React from 'react';

**import** { shallow } from 'enzyme';

shallow() is the only function we’ll use from Enzyme, so we explicitly specify it in our import. As you may have guessed, shallow() is the function we’ll use to shallow render components.

We’ll title our describe after the module under test:

describe('App', () => {

*// assertions will go here*

});

Let’s write our first spec. We’ll assert that our table should render with a table header of “Items”:

describe('App', () => {

it('should have the `th` "Items"', () => {

*// our assertion will go here*

});

*// the rest of our assertions will go here*

});

In order to write this assertion, we’ll need to:

* Shallow render the component
* Traverse the virtual DOM, picking out the first th element
* Assert that that element encloses a text value of “Items”

We first shallow render the component:

**testing/react-basics/src/complete/App.test.complete-1.js**

it('should have the `th` "Items"', () => {

**const** wrapper = shallow(

<App />

);

As mentioned earlier, the shallow() function returns what Enzyme calls a “wrapper” object, ShallowWrapper. This wrapper contains the shallow-rendered component. Remember, there is no actual DOM here. Instead, the component is kept inside of the wrapper in its virtual DOM representation.

The wrapper object that Enzyme provides us with has loads of useful methods that we can use to write our assertions. In general, these helper methods help us traverse and select elements on the virtual DOM.

Let’s see how this works in practice. One helper method is contains(). We’ll use it to assert the presence of our table header:

**testing/react-basics/src/complete/App.test.complete-1.js**

it('should have the `th` "Items"', () => {

**const** wrapper = shallow(

<App />

);

expect(

wrapper.contains(<th>Items</th>)

).toBe(**true**);

});

contains() accepts a ReactElement, in this case JSX representing an HTML element. It returns a boolean, indicating whether or not the rendered component contains that HTML.

###### Try it out

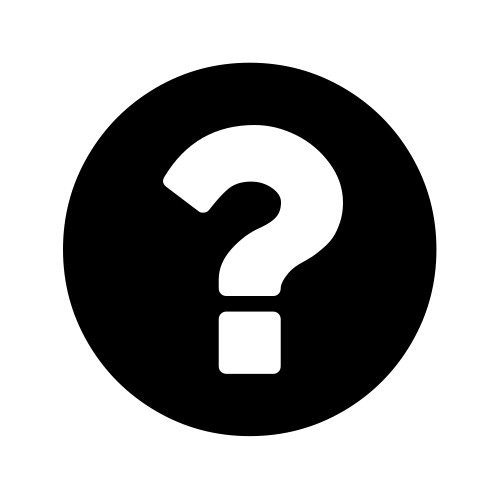
With our first Enzyme spec written, let’s verify everything works. Save App.test.js and run the test command from the console:

$ npm test



Enzyme spec passes

Let’s write some more assertions, exploring the API for Enzyme in the process.

We import React at the top of our test file. Yet, we don’t reference React anywhere in the file. Why do we need it?

You can try removing this import statement and see what happens. You’ll get the following error:

ReferenceError: React is not defined

We can’t readily see the reference to React, but it’s there. We’re using JSX in our test suite. When we specify a th component with <th>Items</th> this compiles to:

React.createElement('th', **null**, 'Items');

### More assertions for App

Next, let’s assert that the component contains a button element, the button that says “Add item.” We might expect we could just do something like this: wrapper.contains(<button>Add Item</button>)

But, contains() matches all the attributes on an element.

Our button inside of render() looks like this:

**testing/react-basics/src/App.js**

<button

className='ui button'

type='submit' disabled={submitDisabled}

>

Add item

</button>

We need to pass contains() a ReactElement that has the exact same set of attributes. But usually this is excessive. For this spec, it’s sufficient to just assert that the button is on the page.

We can use Enzyme’s containsMatchingElement() method. This will check if anything in the component’s output looks like the expected element. We don’t have to match attribute-for-attribute.

Using containsMatchingElement(), let’s assert that the rendered component also includes a button element. Write this spec below the last one:

**testing/react-basics/src/complete/App.test.complete-2.js**

it('should have a `button` element', () => {

**const** wrapper = shallow(

<App />

);

expect(

wrapper.containsMatchingElement(

<button>Add item</button>

)

).toBe(**true**);

});

containsMatchingElement() allows us to write a “looser” spec that’s closer to the assertion we want: that there’s a button on the page. It doesn’t tie our specs to style attributes like className. While the attributes onClick and disabled are important, we’ll write specs later that cover these.

Let’s write another assertion with containsMatchingElement(). We’ll assert that the input field is present as well:

**testing/react-basics/src/complete/App.test.complete-2.js**

it('should have an `input` element', () => {

**const** wrapper = shallow(

<App /> );

expect(

wrapper.containsMatchingElement(

<input />

)

).toBe(**true**);

});

Our specs at this point assert that certain key elements are present in the component’s output after the initial render. As we’ll see shortly, we’re laying the foundation for the rest of our specs. Subsequent specs will assert what happens after we make changes to the component, like populating its input or clicking its button. These fundamental specs assert that the elements we will be interacting with are present on the page to begin with.

In this initial state, there is one more important assertion we should make: that the button on the page is disabled. The button should only be enabled if there is text inside the input.

We actually could modify our previous spec to include this particular attribute, like this:

expect(

wrapper.containsMatchingElement(

<button disabled={**true**}>

Add item

</button> )

).toBe(**true**);

This spec would then be making two assertions: (1) That the button is present and (2) that it is disabled.

This is a perfectly valid approach. However, we like to split these two assertions into two different specs. When you limit the scope of the assertion in a given spec, test failures are much more expressive. If this dual-assertion spec were to fail, it would not be obvious why. Is the button missing? Or is the button not disabled?

This discussion on how to limit assertions per spec touches on the art of unit testing. There are many different strategies and styles for composing unit tests which are highly dependent on the codebase you’re working with. There is usually more than one “right way” to structure a test suite.

Throughout this module, we’ll be exhibiting our particular style. But as you get comfortable with unit testing, feel free to experiment to find a style that works best for you or your codebase. Just be sure to aim to keep your style consistent.

Our three specs so far have asserted that elements are present in the output of our component. This spec is different. We’ll first “find” the component and then make an assertion on its disabled attribute. Let’s take a look at it then break it down:

**testing/react-basics/src/complete/App.test.complete-2.js**

it('`button` should be disabled', () => {

**const** wrapper = shallow(

<App />

);

**const** button = wrapper.find('button').first();

expect(

button.props().disabled

).toBe(**true**);

});

find() is another EnzymeWrapper method. It expects as an argument an Enzyme selector. The selector in this case is a CSS selector, 'button'. A CSS selector is just one supported type of Enzyme selector. We’ll only use CSS selectors in this module, but Enzyme selectors can also refer directly to React components. For more info on Enzyme selectors, see the [Enzyme docs](http://airbnb.io/enzyme/docs/api/selector.html)[[2]](#footnote-2).

find() returns another Enzyme ShallowWrapper. This object contains a list of all matching elements. The object behaves a bit like an array, with methods like length. The object has a method, first(), which we use here to return the first matching element. first() returns another ShallowWrapper object which references the button element.

As you find and select various elements within a shallow rendered component, all of those elements will be Enzyme ShallowWrapper objects. That means you can expect the same API of methods to be available to you, whether you’re working with a shallow-rendered React component or a div tag.

To read the disabled attribute on the button, we use props(). props() returns an object that specifies either the attributes on an HTML element or the props set on a React component.

### Using beforeEach

At this point, our test suite has some repetitious code. We shallow render the component before each assertion. This is ripe for refactor.

We could just shallow render the component at the top of our describe block:

describe('the "App" component', () => {

**const** wrapper = shallow( <App />

);

*// specs here ...*

})

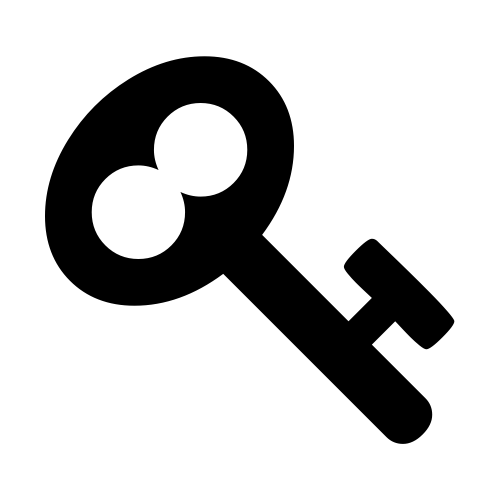
Due to JavaScript’s scoping rules, wrapper would be available inside of each of our it blocks.

But there are problems that can arise with this approach. For instance, what if one of our specs modifies the component? We might change the component’s state or simulate an event. This would cause state to leak between specs. At the start of the next spec, our component’s state would be unpredictable.

It would instead be preferable to re-render the component between each spec, ensuring that each spec is working with the component in a predictable, fresh state.

In all popular JavaScript test frameworks, there’s a function we can use to aid in test setup: beforeEach. beforeEach is a block of code that will run before each **it** block. We can use this function to render our component before each spec.

When writing a test, you’ll often need to perform some setup to get your environment into the proper context to make an assertion. In addition to shallow rendering a component as we do above, we’ll soon write tests that will demand even richer context. By setting up the context inside of a beforeEach, you guarantee that each spec will receive a fresh set of context.

 Setting up fresh context before each spec helps prevent state from leaking between tests.

When writing tests, we strive to keep our individual specs (our it blocks) as terse as possible. We’ll rely on beforeEach to establish context, like the state or props for a component or even events like an element being clicked. Our it blocks, then, will almost always contain exclusively assertions.

Let’s use a beforeEach block to render our component. We can then remove the rendering from each of our assertions:

**testing/react-basics/src/complete/App.test.complete-3.js**

describe('App', () => {

**let** wrapper;

beforeEach(() => {

wrapper = shallow( <App />

);

});

We had to first declare wrapper using a let declaration at the top of the describe block. This is because if we had declared wrapper inside of the beforeEach block like this:

*// ...*

beforeEach(() => {

**const** wrapper = shallow( <App />

);

})

*// ...* wrapper would not have been in scope for our specs. By declaring wrapper at the top of our describe block, we’ve ensured it’s in scope for all of our assertions.

We can now safely remove the declaration of wrapper from each of our assertions:

**testing/react-basics/src/complete/App.test.complete-3.js**

it('should have the `th` "Items"', () => {

expect(

wrapper.contains(<th>Items</th>)

).toBe(**true**);

});

it('should have a `button` element', () => {

expect(

wrapper.containsMatchingElement( <button>Add item</button>

)

).toBe(**true**);

});

it('should have an `input` element', () => {

expect(

wrapper.containsMatchingElement(

<input />

)

).toBe(**true**);

});

it('`button` should be disabled', () => {

**const** button = wrapper.find('button').first(); expect(

button.props().disabled

).toBe(**true**);

});

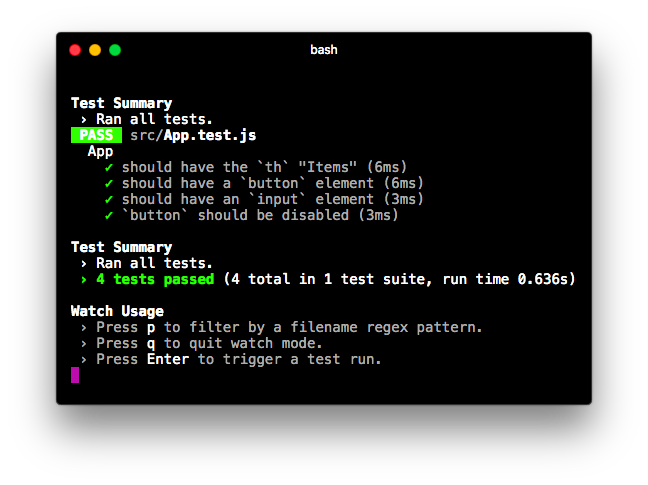
Much better. Our it blocks are no longer setting up context and we’ve removed redundant code.

Try it out

Save App.test.js. Run the test suite:

$ npm test

All four tests pass:



All four passing tests

While limited, these specs set the foundation for our next set of specs. By asserting the presence of certain elements in the initial render as we have so far, we’re asserting what the user will see on the page when the app loads. We asserted that there will be a table header, an input, and a button. We also asserted that the button should be disabled.

For the rest of this module, we’re going to use a behavior-driven style to drive the development of our test suite. With this style, we’ll use beforeEach to set up some context. We’ll simulate interactions with the component much like we were a user navigating the interface. We’ll then write assertions on how the component should have behaved.

After loading the app, the first thing we’d envision a user would do is fill in the input. When the input is filled, they will click the “Add item” button. We would then expect the new item to be in state and on the page.

We’ll step through these behaviors, writing assertions about the component after each user interaction.

### Simulating a change

The first interaction the user can have with our app is filling out the input field for adding a new item. In addition to shallow rendering our component, we want to simulate this behavior before the next set of specs.

While we could perform this setup inside of the it blocks, as we noted before it’s better if we perform as much of our setup as possible inside of beforeEach blocks. Not only does this help us organize our code, this practice makes it easy to have multiple specs that rely on the same setup.

However, we don’t need this particular piece of setup for our other four existing specs. What we should do is declare another describe block inside of our current one. describe blocks are how we “group” specs that all require the same context:

describe('App', () => {

*// ... the assertions we've written so far*

describe('the user populates the input', () => {

beforeEach(() => {

*// ... setup context*

})

*// ... assertions*

});

});

The beforeEach that we write for our inner describe will be run after the beforeEach declared in the outer context. Therefore, the wrapper will already be shallow rendered by the time this beforeEach runs. As expected, this beforeEach will only be run for it blocks inside our inner describe block.

Here’s what our inner describe with our beforeEach setup looks like for the next spec group:

**testing/react-basics/src/complete/App.test.complete-4.js**

describe('the user populates the input', () => {

**const** item = 'Vancouver';

beforeEach(() => {

**const** input = wrapper.find('input').first(); input.simulate('change', {

target: { value: item }

})

});

We first declare item at the top of describe. As we’ll see soon, this will enable us to reference the variable in our specs.

The beforeEach first uses the find() method on EnzymeWrapper to grab the input. Recall that find() returns another EnzymeWrapper object, in this case a list with a single item, our input. We callfirst() to get the EnzymeWrapper object corresponding to the input element.

We then use simulate() on the input. simulate() is how we simulate user interactions on components. The method accepts two arguments:

1. The event to simulate (like 'change' or 'click'). This determines which event handler to use

(like onChange or onClick).

1. The event object (optional).

Here, we’re specifying a 'change' event for the input. We then pass in our desired event object. Note that this event object looks exactly the same as the event object that React passes an onChange handler. Here’s the method onItemChange on App again, which expects an object of this shape:

**testing/react-basics/src/App.js**

onItemChange = (e) => { **this**.setState({

item: e.target.value,

});

};

With this setup written, we can now write specs related to the context where the user has just populated the input field. We’ll write two:

1. That the state property item was updated to match the input field
2. That the button is no longer disabled

Here’s what the describe looks like**, in full:**

**testing/react-basics/src/complete/App.test.complete-4.js**

describe('the user populates the input', () => {

**const** item = 'Vancouver';

beforeEach(() => {

**const** input = wrapper.find('input').first(); input.simulate('change', {

target: { value: item }

})

});

it('should update the state property `item`', () => {

expect(

wrapper.state().item

).toEqual(item); });

it('should enable `button`', () => {

**const** button = wrapper.find('button').first(); expect(

button.props().disabled

).toBe(**false**);

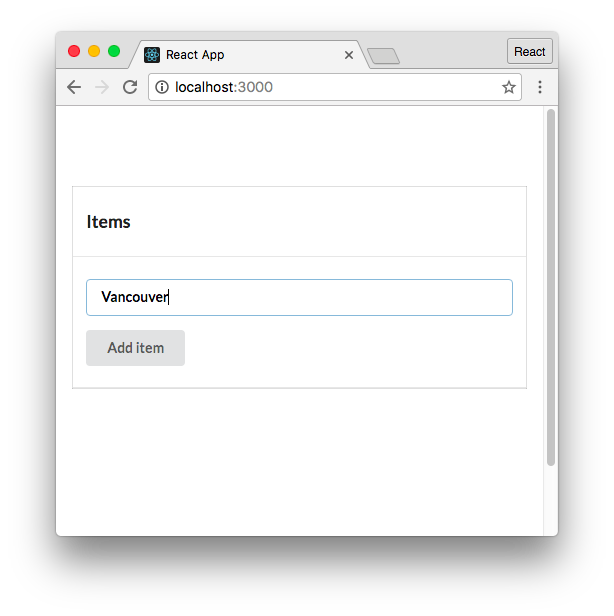
});

});

In the first spec, we used wrapper.state() to grab the state object. Note that it is a function and not a property. Remember, wrapper is an EnzymeWrapper so we’re not interacting with the component directly. We use the state() method which retrieves the state property from the component.

In the second, we used props() again to read the disabled attribute on the button.

Continuing our behavior-driven approach, we now envision our component in the following state:



Imagined state of the component

The user has filled in the input field. There are two actions the user can take from here that we can write specs for:

1. The user clears the input field
2. The user clicks the “Add item” button

### Clearing the input field

When the user clears the input field, we expect the button to become disabled again. We can build on our existing context for the describe “the user populates the input” by nesting our new describe inside of it:

describe('App', () => {

// ... initial state assertions

describe('the user populates the input', () => {

// ... populated field assertions

**describe('and then clears the input', () => { // ... assert the button is disabled again**

**});**

});

});

We’ll use beforeEach to simulate a change event again, this time setting value to a blank string. We’ll write one assertion: that the button is disabled again.

Remember to compose this describe block underneath “the user populates the input.” Our “user clears the input” describe block, in full:

**testing/react-basics/src/complete/App.test.complete-5.js**

it('should enable `button`', () => {

const button = wrapper.find('button').first(); expect(

button.props().disabled

).toBe(false);

});

**describe('and then clears the input', () => {**

**beforeEach(() => {**

**const input = wrapper.find('input').first(); input.simulate('change', {**

**target: { value: '' }**

**})**

**});**

**it('should disable `button`', () => {**

**const button = wrapper.find('button').first(); expect(**

**button.props().disabled**

**).toBe(true);**

**});**

**});**

});

});

Notice how we’re building on existing context, getting deeper into a workflow through our app. We’re three layers deep. The app has rendered, the user filled in the input field, and then the user cleared the input field.

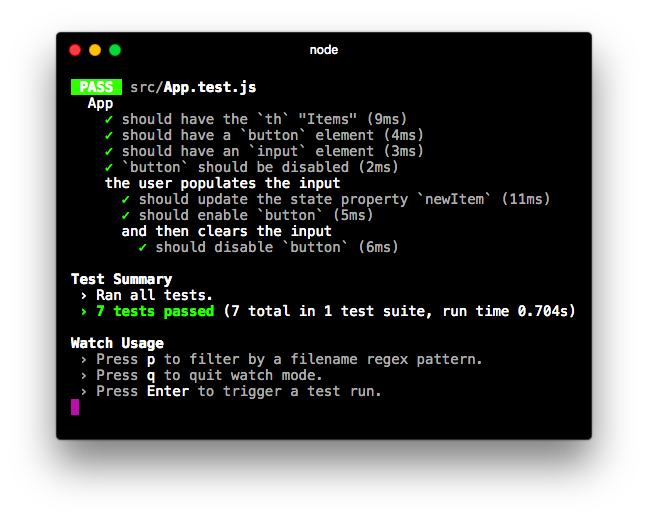
Now’s a good time to verify all our tests pass.

###### Try it out

Save App.test.js. Running the test suite:

$ npm test

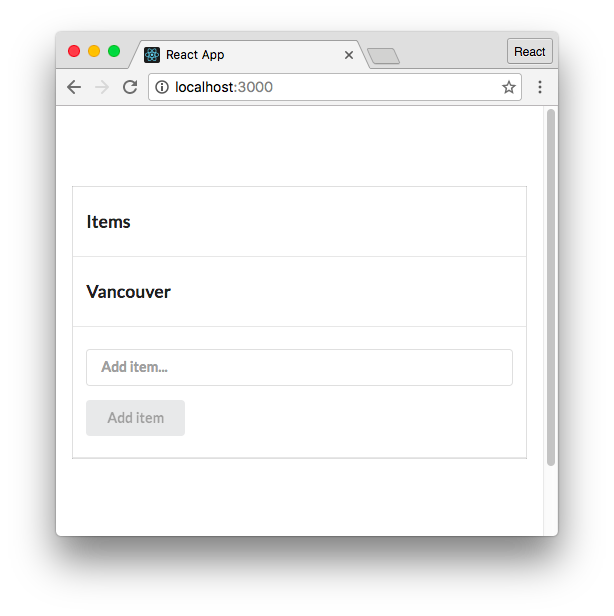
We see everything passes:



Next, we’ll simulate the user submitting the form. This should cause a few changes to our app which we’ll write assertions for.

### Simulating a form submission

After the user has submitted the form, we expect the app to look like this:



We’ll assert that:

1. The new item is in state (items)
2. The new item is inside the rendered table
3. The input field is empty
4. The “Add item” button is disabled

To reach this context, we’ll build on the previous context where the user has populated the input. So we’ll write our describe inside “the user populates the input” as a sibling to “and then clears the input”:

describe('App', () => {

// ... initial state assertions

describe('the user populates the input', () => {

// ... populated field assertions

describe('and then clears the input', () => { // ... assert the button is disabled again

});

**describe('and then submits the form', () => { // ... upcoming assertions**

**});**

});

});

Our beforeEach will simulate a form submission. Recall that addItem expects an object that has a method preventDefault():

**testing/react-basics/src/App.js**

addItem = (e) => {

e.preventDefault();

**this**.setState({

items: **this**.state.items.concat(

**this**.state.item

),

item: '',

});

};

We’ll simulate an event type of submit, passing in an object that has the shape that addItem expects.

We can just set preventDefault to an empty function:

**testing/react-basics/src/complete/App.test.complete-6.js**

describe('and then submits the form', () => { beforeEach(() => {

**const** form = wrapper.find('form').first(); form.simulate('submit', {

preventDefault: () => {},

});

});

With our setup in place, we first assert that the new item is in state:

**testing/react-basics/src/complete/App.test.complete-6.js**

it('should add the item to state', () => { expect(

wrapper.state().items

).toContain(item);

});

Jest comes with a few special matchers for working with arrays. We use the matcher toContain() to assert that the array items contains item.

Remember, wrapper is an EnzymeWrapper object with a React component inside. We retrieve the state with state() (a method).

Next, let’s assert that the item is inside the table. There’s a few ways to do this, here’s one:

**testing/react-basics/src/complete/App.test.complete-6.js**

it('should render the item in the table', () => { expect(

wrapper.containsMatchingElement( <td>{item}</td> )

(

.toBe

)

**true**

)

;

})

;

Next, we’ll assert that the input field has been cleared. We have the option of checking the state property item or checking the actual input field in the virtual DOM. We’ll do the latter as it’s a bit more comprehensive:

**testing/react-basics/src/complete/App.test.complete-6.js**

it('should clear the input field', () => {

**const** input = wrapper.find('input').first(); expect(

input.props().value

).toEqual('');

});

Finally, we’ll assert that the button is again disabled:

**testing/react-basics/src/complete/App.test.complete-6.js**

it('should disable `button`', () => {

**const** button = wrapper.find('button').first(); expect(

button.props().disabled

).toBe(**true**);

});

Our “and then submits the form” describe, **in full:**

**testing/react-basics/src/complete/App.test.complete-6.js**

it('should disable `button`', () => {

const button = wrapper.find('button').first(); expect(

button.props().disabled

).toBe(true);

});

});

**describe('and then submits the form', () => { beforeEach(() => {**

**const form = wrapper.find('form').first(); form.simulate('submit', {**

**preventDefault: () => {},**

**});**

**});**

**it('should add the item to state', () => { expect(**

**wrapper.state().items**

**).toContain(item);**

**});**

**it('should render the item in the table', () => { expect(**

**wrapper.containsMatchingElement(**

**<td>{item}</td>**

**)**

**).toBe(true);**

**});**

**it('should clear the input field', () => {**

**const input = wrapper.find('input').first(); expect(**

**input.props().value**

**).toEqual(''); });**

**it('should disable `button`', () => {**

**const button = wrapper.find('button').first(); expect(**

**button.props().disabled**

**).toBe(true);**

**});**

**});**

});

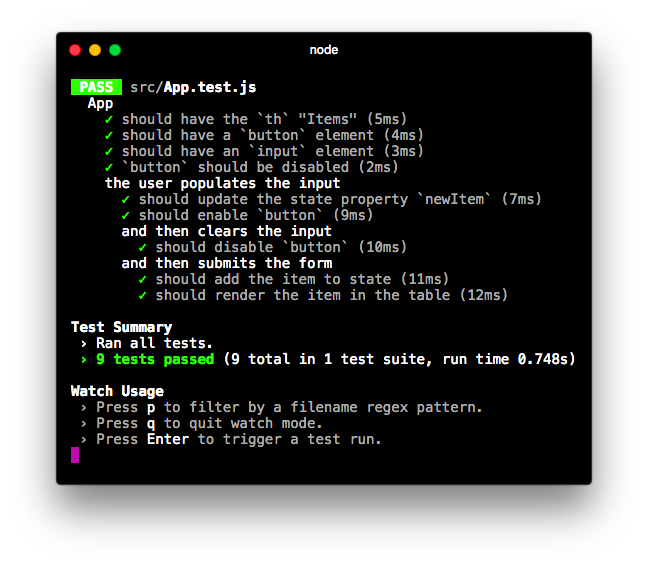
});

###### Try it out

Save App.test.js. Run the test suite:

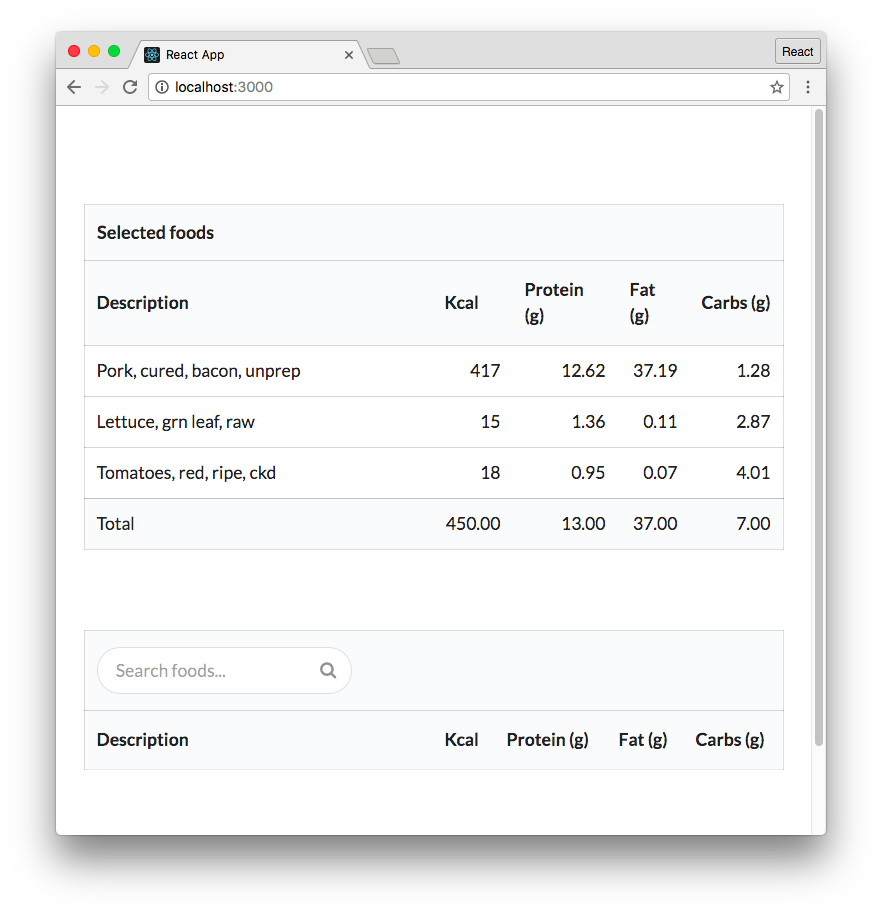
$ npm test

Everything passes:



## OPTIONAL - Writing tests for the food lookup app

In a previous module on Webpack and create-react-app, we set up a Webpack-powered food lookup app:



The food lookup app

We’ll work inside the completed version of this app. It’s in the top-level folder food-lookupcomplete. To get to it from the testing/react-basics folder, run the following command:

$ cd ../../food-lookup-complete

Install the npm packages for both the server and the client if they are not installed already:

$ npm i

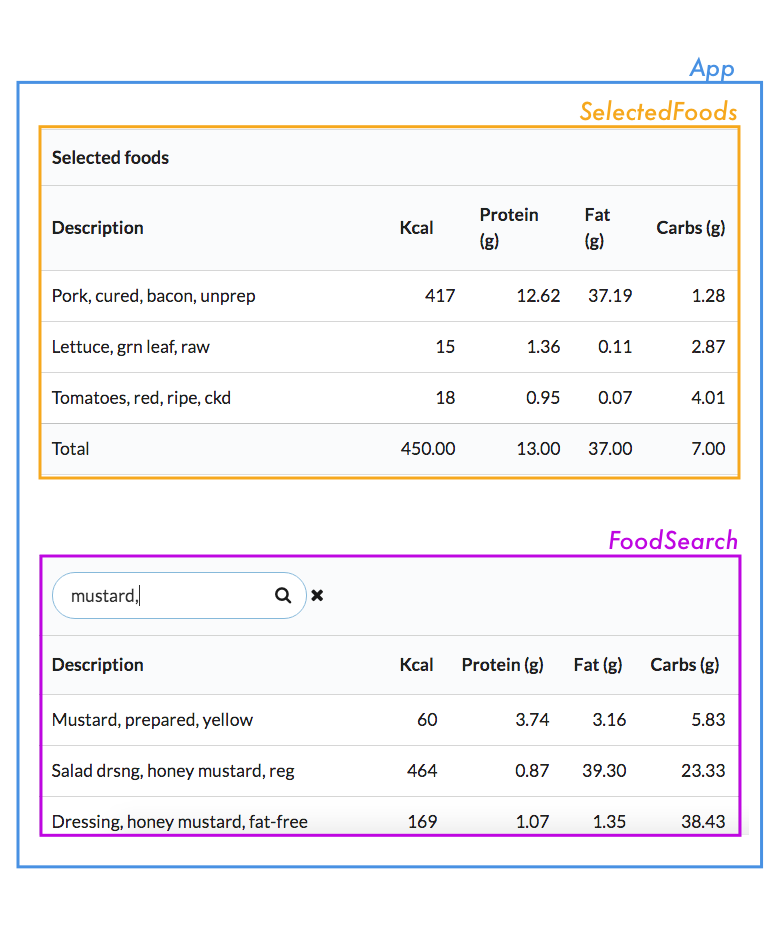
$ cd client

$ npm i $ cd ..

And start up the app to play around with it if you’d like with:

$ npm start

We’ll be writing tests for just the component FoodSearch in this module. We won’t dig into the code for the other components in the app. Instead, it’s sufficient to understand how the app is broken down into components at a high-level:



* App: The parent container for the application.
* SelectedFoods: A table that lists selected foods. Clicking on a food item removes it.
* FoodSearch: Table that provides a live search field. Clicking on a food item in the table adds it to the total (SelectedFoods).

Kill the app if you started it. Change into the client/ directory. We’ll be working solely in this directory for this module:

$ cd client

For this app, instead of having the tests alongside the components in src we’ve placed them inside a dedicated folder, tests. Inside of tests, you’ll see that tests already exist for the other components in our app:

$ dir tests/

App.test.js SelectedFoods.test.js complete/

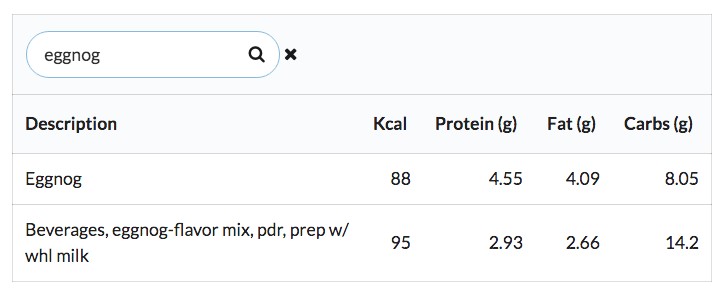
Before writing tests for the component, let’s walk through how FoodSearch works. You can pop open the FoodSearch component and follow along if you’d like (src/FoodSearch.js).

complete/ contains each version of the completed FoodSearch.test.js that we write in this section, for your reference.

### 

### FoodSearch

The FoodSearch component has a search field. As the user types, a table of matching foods is updated below the search field:



The **FoodSearch** component

When the search field is changed, the FoodSearch component makes a request to the app’s API server. If the user has typed in the string truffle, the request to the server looks like this:

GET localhost:3001/api/food?q=truffle

The API server then returns an array of matching food items:

[

{

"description": "Pate truffle flavor",

"kcal": 327,

"fat\_g": 27.12,

"protein\_g": 11.2,

"carbohydrate\_g": 6.3

},

{

"description": "Candies, truffles, prepared-from-recipe",

"kcal": 510,

"fat\_g": 32.14,

"protein\_g": 6.21,

"carbohydrate\_g": 44.88

},

{

"description": "Candies, m m mars 3 musketeers truffle crisp",

"kcal": 538,

"fat\_g": 25.86,

"protein\_g": 6.41,

"carbohydrate\_g": 63.15

}

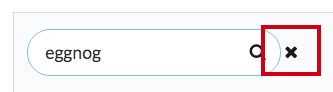
]

FoodSearch populates its table with these items.

FoodSearch has three pieces of state: **foods**

This is an array of all of the foods returned by the server. It defaults to a blank array. **showRemoveIcon**

When the user starts typing into the search field, an X appears next to the field:



The remove icon

This X provides a quick way to clear the search field. When the field is empty, showRemoveIcon should be false. When the field is populated, showRemoveIcon should be true. **searchValue** searchValue is the state that’s tied to the controlled input, the search field.

### Exploring FoodSearch

Armed with the knowledge of how FoodSearch behaves and what state it keeps, let’s explore the actual code. We’ll include the code snippets here, but feel free to follow along by opening **src/FoodSearch.js.**

At the top of the component are import statements:

**food-lookup-complete/client/src/FoodSearch.js**

**import** React from 'react'; **import** Client from './Client';

We also have a constant that defines the maximum number of search results to show on the page. We use this inside of the component:

**food-lookup-complete/client/src/FoodSearch.js**

**const** MATCHING\_ITEM\_LIMIT = 25;

Then we define the component.

state initialization for our three pieces of state:

**food-lookup-complete/client/src/FoodSearch.js**

**class** FoodSearch **extends** React.Component {

state = {

foods: [], showRemoveIcon: **false**, searchValue: '',

};

Let’s step through the interactive elements in the component’s render method along with each element’s handling function.

###### The input search field

The input field at the top of FoodSearch drives the search functionality. The table body updates with search results as the user modifies the input field.

The input element:

**food-lookup-complete/client/src/FoodSearch.js**

<input

className='prompt' type='text' placeholder='Search foods...' value={**this**.state.searchValue} onChange={**this**.onSearchChange} />

className is set for SemanticUI styling purposes. value ties this controlled input to this.state.searchValue.

onSearchChange() accepts an event object. Let’s step through the code. Here’s the first half of the function:

**food-lookup-complete/client/src/FoodSearch.js**

onSearchChange = (e) => {

**const** value = e.target.value;

**this**.setState({

searchValue: value,

});

**if** (value === '') { **this**.setState({

foods: [], showRemoveIcon: **false**,

});

We grab the value off of the event object. We then set searchValue in state to this value, following the pattern for handling the change for a controlled input.

If the value is blank, we set foods to a blank array (clearing the search results table) and set showRemoveIcon to false (hiding the “X” that’s used to clear the search field).

If the value is not blank, we need to:

* Ensure that the showRemoveIcon is set to true
* Make a call to the server with the latest search value to get the list of matching foods

Here’s that code:

**food-lookup-complete/client/src/FoodSearch.js**

} **else** {

**this**.setState({

showRemoveIcon: **true**,

});

Client.search(value, (foods) => { **this**.setState({

foods: foods.slice(0, MATCHING\_ITEM\_LIMIT), });

});

}

};

Client uses the Fetch API under the hood, the same web request interface that we used in Module 3. Client.search() makes the web request to the server and then invokes the callback with the array of matching foods.

If you need a refresher on working with a client library driven by Fetch, see Module 3: Components & Servers.

We then set foods in state to the returned foods, truncating this list so that it’s within the bounds of MATCHING\_ITEM\_LIMIT (which is 25).

**In full,** onSearchChange() looks like this:

**food-lookup-complete/client/src/FoodSearch.js**

onSearchChange = (e) => {

**const** value = e.target.value;

**this**.setState({

searchValue: value,

});

**if** (value === '') { **this**.setState({

foods: [], showRemoveIcon: **false**,

});

} **else** {

**this**.setState({

showRemoveIcon: **true**,

});

Client.search(value, (foods) => { **this**.setState({

foods: foods.slice(0, MATCHING\_ITEM\_LIMIT), });

});

}

};

###### The remove icon

As we’ve seen, the remove icon is the little X that appears next to the search field whenever the field is populated. Clicking this X should clear the search field.

We perform the logic for whether or not to show the remove icon in-line. The icon element has the onClick attribute:

**food-lookup-complete/client/src/FoodSearch.js**

**this**.state.showRemoveIcon ? (

<i className='remove icon' onClick={**this**.onRemoveIconClick}

/>

) : ''

}

The code for onRemoveIconClick():

**food-lookup-complete/client/src/FoodSearch.js**

onRemoveIconClick = () => {

**this**.setState({

foods: [], showRemoveIcon: **false**, searchValue: '',

});

};

We reset everything, including foods.

**props.onFoodClick**

The final bit of interactivity is on each food item. When the user clicks a food item, we add it to the list of selected foods on the interface:

**food-lookup-complete/client/src/FoodSearch.js**

<tbody>

{ **this**.state.foods.map((food, idx) => (

<tr key={idx} onClick={() => **this**.props.onFoodClick(food)}

>

<td>{food.description}</td>

<td className='right aligned'> {food.kcal}

</td>

<td className='right aligned'> {food.protein\_g}

</td>

<td className='right aligned'> {food.fat\_g}

</td>

<td className='right aligned'> {food.carbohydrate\_g}

</td>

</tr>

))

}

</tbody>

Under the hood, when the user clicks a food item we invoke this.props.onFoodClick(). The parent of FoodSearch (App) specifies this prop-function. It expects the full food object.

As we’ll see, for the purpose of writing unit tests for FoodSearch, we don’t need to know anything about what the prop-function onFoodClick() actually does. We just care about what it wants (a full food object).

Shallow rendering helps us achieve this desirable isolation. While this app is relatively small, these isolation benefits are huge for larger teams with larger codebases.

## Writing FoodSearch.test.js

We’re ready to write unit tests for the FoodSearch component.

The file client/src/tests/FoodSearch.test.js contains the scaffold for the test suite. At the top are the import statements:

**food-lookup-complete/client/src/tests/FoodSearch.test.js**

**import** { shallow } from 'enzyme'; **import** React from 'react';

**import** FoodSearch from '../FoodSearch';

Next is the scaffolding for the test suite. Don’t be intimidated, we’ll be filling out each of these describe and beforeEach blocks one-by-one:

**food-lookup-complete/client/src/tests/FoodSearch.test.js**

describe('FoodSearch', () => { *// ... initial state specs*

describe('user populates search field', () => { beforeEach(() => {

*// ... simulate user typing "brocc" in input*

});

*// ... specs*

describe('and API returns results', () => { beforeEach(() => {

*// ... simulate API returning results*

});

*// ... specs*

describe('then user clicks food item', () => {

beforeEach(() => {

*// ... simulate user clicking food item*

});

*// ... specs*

});

describe('then user types more', () => {

beforeEach(() => {

*// ... simulate user typing "x"*

});

describe('and API returns no results', () => { beforeEach(() => {

*// ... simulate API returning no results*

});

*// ... specs*

});

});

});

});

});

As in the previous section, we’ll establish different contexts by using beforeEach to perform setup. Each of our contexts will be contained inside describe.

### In initial state

Our first series of specs will involve the component in its initial state. Our beforeEach will simply shallow render the component. We’ll then write assertions on this initial state:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-1.js**

describe('FoodSearch', () => {

**let** wrapper;

beforeEach(() => { wrapper = shallow( <FoodSearch />

);

});

As in our first round of component tests in the last section, we declare wrapper in the upper scope. In our beforeEach, we use Enzyme’s shallow() to shallow-render the component. Let’s write two assertions:

1. That the remove icon is not in the DOM
2. That the table doesn’t have any entries

For our first test, there are multiple ways we can write it. Here’s one:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-1.js**

it('should not display the remove icon', () => {

expect(

wrapper.find('.remove.icon').length

).toBe(0);

});

We pass a selector to wrapper’s find() method. If you recall, the remove icon has the following className attribute:

**food-lookup-complete/client/src/FoodSearch.js**

<i className='remove icon' onClick={**this**.onRemoveIconClick}

/>

So, we’re selecting it here based on its class. find() returns a ShallowWrapper object. This object is array-like, containing a list of all matches for the specified selector. Just like an array, it has the property length which we assert should be 0.

We could also have used one of the contains methods, like this:

it('should not display the remove icon', () => {

expect(

wrapper.containsAnyMatchingElements( <i className='remove icon' />

)

).toBe(**false**);

});

We’ll primarily be driving our tests with find() for the rest of this module, as we like using the CSS selector syntax. But it’s a matter of preference.

Next, we’ll assert that in this initial state the table does not have any entries.

For this spec, we can just assert that this component did not output any tr elements inside of tbody, like so:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-1.js**

it('should display zero rows', () => {

expect(

wrapper.find('tbody tr').length

).toEqual(0);

});

If we were to run this test suite now, both of our specs would pass.

However, this doesn’t lend us much assurance that our specs are composed correctly. When asserting that an element is not present on the DOM, you expose yourself to the error where you’re simply not selecting the element properly. We’ll address this shortly when we use the exact same selectors to assert that the elements are present.

### A user has typed a value into the search field

Guided by our behavior-driven approach, our next step is to simulate a user interaction. We’ll then write assertions given this new layer of context.

There’s only one interaction the user can have with the FoodSearch component after it loads:

entering a value into the search field. When this happens, there are two further possibilities:

1. The search matches food in the database and the API returns a list of those foods
2. The search does not match any food in the database and the API returns an empty array

This branch happens at the bottom of onSearchChange() when we call Client.search():

**food-lookup-complete/client/src/FoodSearch.js**

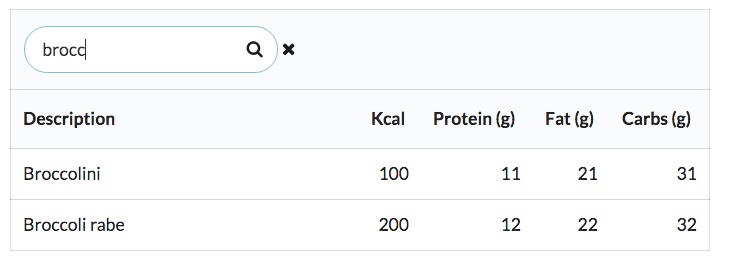
Client.search(value, (foods) => { **this**.setState({

foods: foods.slice(0, MATCHING\_ITEM\_LIMIT), });

});

For our app, the API is queried and results are displayed with every keystroke. Therefore, situation 2 (no results) will almost always happen after situation 1.

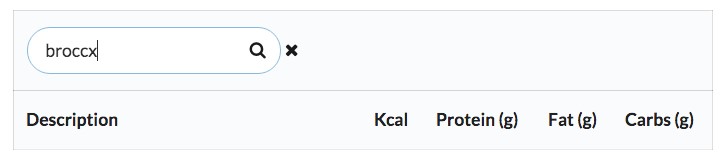
We’ll setup our test context to mirror this state transition. We’ll simulate the user typing 'brocc' into the search field, yielding two results (two kinds of broccoli):



What our component might look like after the user types ‘brocc’

We’ll write assertions against this context.

Next, we’ll build on this context by simulating the user typing an “x” ('broccx'). This will yield no results:



What our component might look like after the user types ‘brocc’ We’ll then write assertions against this context.

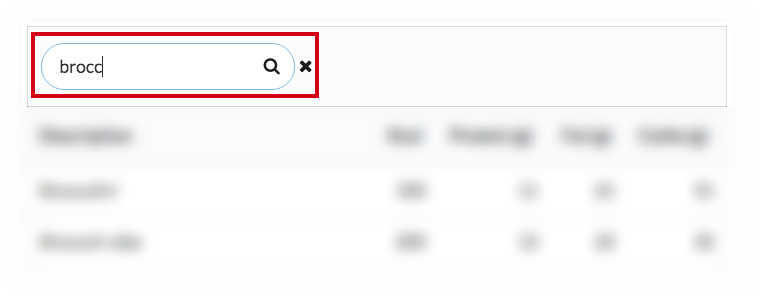
There are exceptions to situation 2 always following situation 1. For instance, this user overestimated the capabilities of our app:



Welp

However, the state transition of foods in state to no foods in state is much more interesting than verifying that foods in state remained blank after the AP I returned empty results.

Regardless of what Client.search() returns, we expect the component to both update searchValue in state and display the remove icon. Those specs will exist up top inside “user populates search field.” We’ll start with these, only writing specs for the search field itself and leaving assertions based on what the API returned for later:



First set of specs focus on the search field

After writing these specs, we’ll see how to establish the context for the API returning results.

We first simulate the user interaction inside a beforeEach. We’ll declare value at the top of the describe so we can reference it later in our tests:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-2.js**

describe('user populates search field', () => {

**const** value = 'brocc';

beforeEach(() => {

**const** input = wrapper.find('input').first(); input.simulate('change', {

target: { value: value },

});

});

Next, we assert that searchValue has been updated in state to match this new value:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-2.js**

it('should update state property `searchValue`', () => {

expect(

wrapper.state().searchValue

).toEqual(value);

});

We assert next that the remove icon is present on the DOM:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-2.js**

it('should display the remove icon', () => {

expect(

wrapper.find('.remove.icon').length

).toBe(1);

});

We use the same selector that we used in our earlier assertion that the remove icon was not present on the DOM. This is important, as it assures us that our earlier assertion is valid and isn’t just using the wrong selector.

Our assertions for “user populates search field” are in place. Before moving on, let’s save and make sure our test suite passes.

Try it out

Save FoodSearch.test.js. From your console:

*# inside client/* $ npm test

Everything should pass:



Tests pass

From here, the next layer of context will be the API returning results.

If we were writing integration tests, we’d take one of two approaches. If we wanted a full end-toend test, we’d have Client.search() make an actual call to the API. Otherwise, we could use a Node library to “fake” the HTTP request. There are plenty of libraries that can intercept JavaScript’s attempt to make an HTTP request. You can supply these libraries with a fake response object to provide to the caller.

However, as we’re writing unit tests, we want to remove any dependency on both the API and the implementation details of Client.search(). We’re exclusively testing the FoodSearch component, a single unit in our application. We only care about how FoodSearch uses Client.search(), nothing deeper.

As such, we want to intercept the call to Client.search() at the surface. We don’t want Client to get involved at all. Instead, we want to assert that Client.search() was invoked with the proper parameter (the value of the search field). And then we want to invoke the callback passed to Client.search() with our own result set.

What we’d like to do is mock the **Client** library.

### Mocking with Jest

When writing unit tests, we’ll often find that the module we’re testing depends on other modules in our application. There are multiple strategies for dealing with this, but they mostly center around the idea of a test double. A test double is a pretend object that “stands in” for a real one.

For instance, we could write a fake version of the Client library for use in our tests. The simplest version would look like this:

**const** Client = {

search: () => {},

};

We could “inject” this fake Client as opposed to the real one into FoodSearch for testing purposes. FoodSearch could call Client.search() anywhere it wanted and it would invoke an empty function as opposed to performing an HTTP request.

We could take it a step further by injecting a fake Client that always returns a certain result. This would prove even more useful, as we’d be able to assert how the state for FoodSearch updates based on the behavior of Client:

**const** Client = {

search: (\_, cb) => {

**const** result = [

{ description: 'Hummus', kcal: '166', protein\_g: '8', fat\_g: '10', carbohydrate\_g: '14',

}, ]; cb(result);

},

};

This test double implements a search() method that immediately invokes the callback passed as the second argument. It invokes the callback with a hard-coded array that has a single food object.

But the implementation details of the test double are irrelevant. What’s important is that this test double is mimicking the API returning the same, one-entry result set every time. With this fake client inserted into the app, we can readily write assertions on how FoodSearch handles this “response”: that there’s now a single entry in the table, that the description of that entry is “Hummus”, etc.

We use \_ as the first argument above to signify that we “don’t care” about this argument. This is purely a stylistic choice.

It would be even better if our test double allowed us to dynamically specify what result to use. That way we wouldn’t need to define a completely different double to test what happens if the API doesn’t return any results. Furthermore, the simple test double above doesn’t care about the search term passed to it. It would be nice to ensure that FoodSearch is invoking Client.search() with the appropriate value (the value of the input field).

Jest ships with a generator for a powerful flavor of test doubles: mocks. We’ll use Jest’s mocks as our test double. The best way to understand mocks is to see them in action.

You generate a Jest mock like this:

**const** myMockFunction = jest.fn();

This mock function can be invoked like any other function. By default, it will not have a return value:

console.log(myMockFunction()); *// undefined*

When you invoke a vanilla mock function nothing appears to happen. However, what’s special about this function is that it will keep track of invocations. Jest’s mock functions have methods you can use to introspect what happened.

For example, you can ask a mock function how many times it was called:

**const** myMock = jest.fn(); console.log(myMock.mock.calls.length);

*// -> 0*

myMock('Paris'); console.log(myMock.mock.calls.length);

*// -> 1*

myMock('Paris', 'Amsterdam'); console.log(myMock.mock.calls.length);

*// -> 2*

All of the introspective methods for a mock are underneath the property mock. By calling myMock.mock.calls, we receive an array of arrays. Each entry in the array corresponds to the arguments of each invocation:

**const** myMock = jest.fn(); console.log(myMock.mock.calls);

*// -> []*

myMock('Paris');

console.log(myMock.mock.calls);

*// -> [ [ 'Paris' ] ]*

myMock('Paris', 'Amsterdam'); console.log(myMock.mock.calls);

*// -> [ [ 'Paris' ], [ 'Paris', 'Amsterdam' ] ]*

This simple feature unlocks tons of power that we’ll soon witness. We could declare our own Client double, using a Jest mock function:

**const** Client = {

search: jest.fn(),

};

But Jest can take care of this for us. Jest has a mock generator for entire modules. By calling this method:

jest.mock('../src/Client')

Jest will look at our Client module. It will notice that it exports an object with a search() method. It will then create a fake object – a test double – that has a search() method that is a mock function. Jest will then ensure that the fake Client is used everywhere in the app as opposed to the real one.

### Mocking Client

Let’s use jest.mock() to mock Client. Using the special properties of mock functions, we’ll then be able to write an assertion that search() was invoked with the proper argument.

At the top of FoodSearch.test.js, below the import statement for FoodSearch, let’s import Client as we’ll be referencing it later in the test suite. In addition, we tell Jest we’d like to mock it:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-3.js**

import FoodSearch from '../FoodSearch'; **import Client from '../Client'; jest.mock('../Client');** describe('FoodSearch', () => {

Now, let’s consider what will happen. In our beforeEach block, when we simulate the change:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-3.js**

beforeEach(() => {

**const** input = wrapper.find('input').first(); input.simulate('change', {

target: { value: value },

});

});

This will trigger the call to Client.search() at the bottom of onSearchChange():

**food-lookup-complete/client/src/FoodSearch.js**

Client.search(value, (foods) => { **this**.setState({

foods: foods.slice(0, MATCHING\_ITEM\_LIMIT), });

});

Except, instead of calling the method on the real Client, it’s calling the method on the mock that Jest has injected. Client.search() is a mock function and has done nothing except log that it was called.

Let’s declare a new spec below “should display the remove icon.” Before writing the assertion, let’s just log a few things out to the console to see what’s happening:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-3.js**

it('should display the remove icon', () => {

expect(

wrapper.find('.remove.icon').length

).toBe(1); });

**it('...todo...', () => {**

**const firstInvocation = Client.search.mock.calls[0]; console.log('First invocation:'); console.log(firstInvocation);**

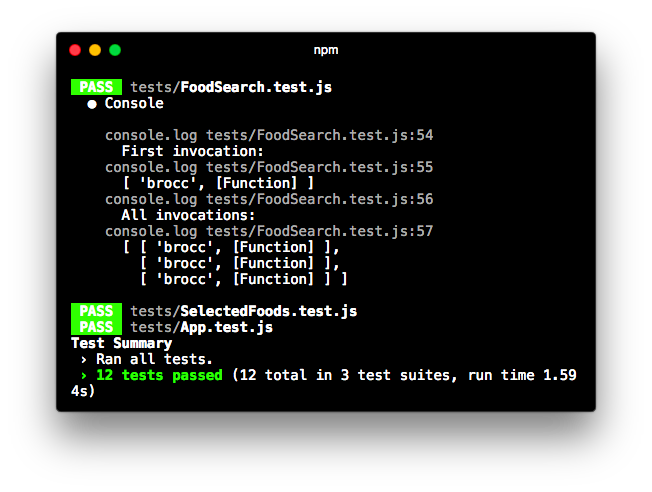
**console.log('All invocations: '); console.log(Client.search.mock.calls);**

**});**

describe('and API returns results', () => {

We read the mock.calls property on the mock function. Each entry in the calls array corresponds to an invocation of the mock function Client.search().

If you were to save FoodSearch.test.js and run the test suite, you’d be able to see the log statements in the console:



Log statements in the test run Picking out the first one:

First invocation:

[ 'brocc', [Function] ]

The mock captured the invocation that occurred in the beforeEach block. The first argument of the invocation is what we’d expect, 'brocc'. And the second argument is our callback function. Importantly, the callback function has yet to be invoked. search() has captured the function but not done anything with it.

If you were to fence the call to Client.search() with console.log() statements, like this:

// Example of "fencing" `Client.search()` **console.log('Before `search()`');**

Client.search(value, (foods) => { **console.log('Inside the callback');** this.setState({

foods: foods.slice(0, MATCHING\_ITEM\_LIMIT), }); });

**console.log('After `search()`');**

You’d see this output in the console when running the test suite:

Before `search()`

After `search()`

The mock function search() is invoked but all it does is capture the arguments. The line that logs “Inside the callback” has not been called.

So, the console output for “First invocation” makes sense. However, check out the console output for “All invocations”:

All invocations:

[ [ 'brocc', [Function] ],

[ 'brocc', [Function] ], [ 'brocc', [Function] ] ]

Reformatting that array:

[

[ 'brocc', [Function] ],

[ 'brocc', [Function] ],

[ 'brocc', [Function] ]

]

We see three invocations in total. Why is this?

We have three it blocks that correspond to our beforeEach that simulates a change. Remember, a beforeEach is run once before each related it. Therefore, our **beforeEach** that simulates a search is executed three times. Which means the mock function Client.search() is invoked three times as well.

While this makes sense, it’s undesirable. State is leaking between specs. We want each it to receive a fresh version of the Client mock.

Jest mock functions have a method for this, mockClear(). We’ll invoke this method after each spec is executed using the antipode of beforeEach, afterEach. This will ensure the mock is in a pristine state before each spec run. We’ll do this inside the top-level describe, below the beforeEach where we shallow-render the component:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-4.js**

describe('FoodSearch', () => {

let wrapper;

beforeEach(() => { wrapper = shallow( <FoodSearch />

); });

**afterEach(() => {**

**Client.search.mockClear();**

**});** it('should not display the remove icon', () => {

We could have used a beforeEach block here as well, but it usually makes sense to perform any “tidying up” in afterEach blocks.

Now, if we run our test suite again:

First invocation: [ 'brocc', [Function] ] All invocations:

[ [ 'brocc', [Function] ] ]

We’ve succeeded in resetting the mock between test runs. There’s only a single invocation logged, the invocation that occurred right before this last it was executed.

With our mock behaving as desired, let’s convert our dummy spec into a real one. We’ll assert that the first argument passed to Client.search() is the same value the user typed into the search field:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-5.js**

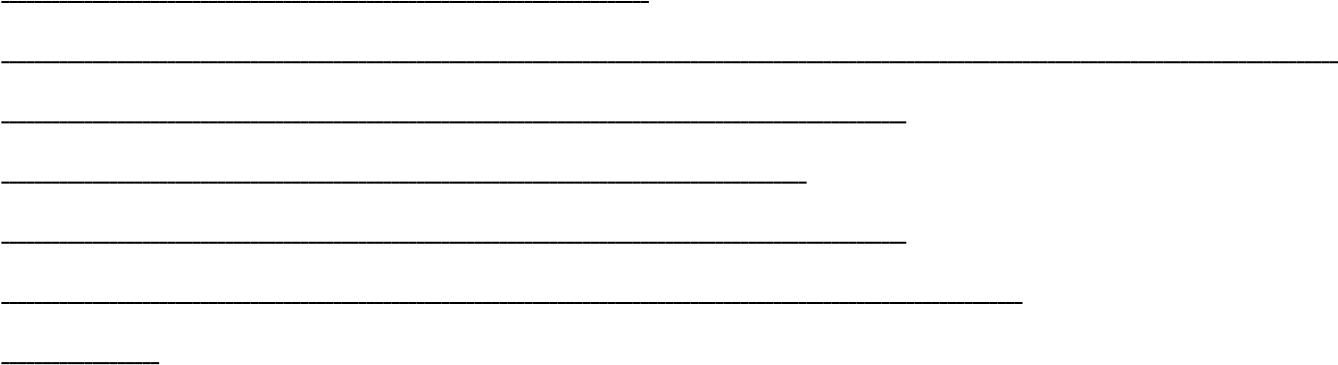
it('should display the remove icon', () => {

expect(

wrapper.find('.remove.icon').length

).toBe(1); });

it('...todo...', () => {

const firstInvocation = Client.search.mock.calls[0]; console.log('First invocation:'); console.log(firstInvocation); console.log('All invocations: '); console.log(Client.search.mock.calls);

});

**it('should call `Client.search() with `value`', () => {**

**const invocationArgs = Client.search.mock.calls[0]; expect(**

**invocationArgs[0]**

**).toEqual(value); });** describe('and API returns results', () => {

We’re asserting that the zeroeth argument of the invocation matches value, in this case brocc.

Try it out

With Client mocked, we can run our test suite assured that FoodSearch is in total isolation. Save FoodSearch.test.js and run the test suite from the console:

$ npm test

The result:



All specs pass

We used a Jest mock function to both capture and introspect the Client.search() invocation. Now, let’s see how we can use it to establish behavior for our next layer of context: when the API returns results.

### The API returns results

As we can see in the pre-existing test scaffolding, we’ll write the specs pertaining to this context inside of their own describe:

describe('FoodSearch', () => { // ...

describe('user populates search field', () => { // ...

**describe('and API returns results', () => { beforeEach(() => {**

**// ... simulate API returning results**

**});**

**// ... specs**

**});**

describe('then user types more', () => { // ...

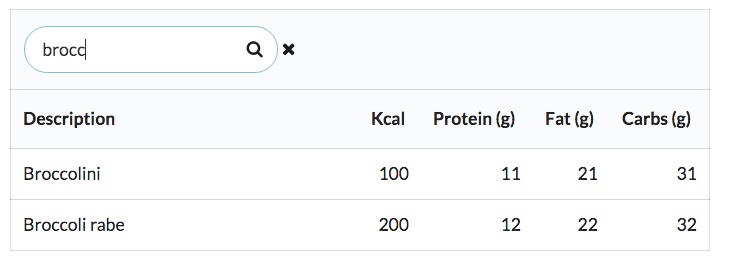
});

});

});

In our beforeEach for this describe, we want to simulate the API returning results. We can do this by manually invoking the callback function passed to Client.search() with whatever we’d like to simulate the API returning.

We’ll fake Client returning two matches. We can picture our component in this state:



Visual representation of desired state for component

Let’s look at the code first then we’ll break it down:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-6.js**

it('should call `Client.search() with `value`', () => {

const invocationArgs = Client.search.mock.calls[0]; expect(

invocationArgs[0]

).toEqual(value); });

describe('and API returns results', () => {

**const foods = [**

**{**

**description: 'Broccolini', kcal: '100',**

**protein\_g: '11',**

**fat\_g: '21',**

**carbohydrate\_g: '31',**

**},**

**{**

**description: 'Broccoli rabe', kcal: '200',**

**protein\_g: '12',**

**fat\_g: '22',**

**carbohydrate\_g: '32',**

**},**

**];**

**beforeEach(() => {**

**const invocationArgs = Client.search.mock.calls[0];**

**const cb = invocationArgs[1];**

**cb(foods);**

**wrapper.update();**

**});**

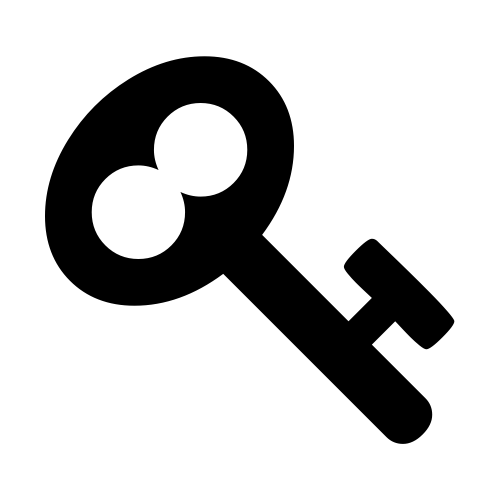
First, we declare an array, foods, which we use as the fake result set returned by Client.search().

Second, in our beforeEach, we grab the second argument that Client.search() was invoked with, in this case our callback function. We then invoke it with our array of food objects. By manually invoking callbacks passed to a mock, we can simulate desired behavior of asynchronous resources.

Last, we call wrapper.update() after invoking the callback. This will cause our component to re-render. When a component is shallow-rendered, the usual re-rendering hooks do not apply. Therefore, when setState() is called within our callback, a re-render is not triggered.

If that’s the case, you might wonder why this is the first time we’ve needed to use wrapper.update(). Enzyme has actually been automatically calling update() after every one of our simulate() calls. simulate() invokes an event handler. Immediately after that event handler returns, Enzyme will call wrapper.update().

Because we’re invoking our callback asynchronously some time after the event handler returns, we need to manually call wrapper.update() to re-render the component.

When a component is shallow-rendered, the usual re-rendering hooks do not apply. If any state changes instigated by a simulate() are made asynchronously, you must call update() to re-render the component.

In this module, we exclusively use Enzyme’s simulate() to manipulate a component. Enzyme also has another method, setState(), which you can use in special circumstances when a simulate() call is not viable. setState() also automatically calls update() after it is invoked.

 Yes, the nutritional info for the broccolis in our test is totally bogus!

With our callback invoked, let’s write our first spec. We’ll assert that the foods property in state matches our array of foods:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-6.js**

it('should set the state property `foods`', () => {

expect(

wrapper.state().foods

).toEqual(foods);

});

Again, we use the state() method when reading state from an EnzymeWrapper.

Next, we’ll assert that the table has two rows:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-6.js**

it('should display two rows', () => { expect(

wrapper.find('tbody tr').length

).toEqual(2);

});

Because this spec uses the same selector as our previous spec “should display zero rows,” this gives us assurance that the previous spec uses the proper selector.

Finally, let’s take it a step further and assert that both of our foods are actually printed in the table. There are many ways to do this. Because the description of each is so unique, we can actually just hunt for each food’s description in the HTML output, like this:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-6.js**

**it('should render the description of first food', () => {**

**expect(**

**wrapper.html()**

**).toContain(foods[0].description);**

**});**

**it('should render the description of second food', () => {**

**expect(**

**wrapper.html()**

**).toContain(foods[1].description);**

**});**

describe('then user clicks food item', () => {

Because we’re hunting for a unique string, there’s no need to use Enzyme’s selector API. Instead, we use Enzyme’s html() to yield a string of the component’s HTML output. We then use Jest’s toContain() matcher, this time on a string as opposed to an array.

html() is also a great method for debugging shallow-rendered components. For instance, seeing the full H TML output of a component can help you determine if an assertion issue is due to an erroneous selector or an erroneous component.

For this set of specs, we’re working with two food items returned from the AP I (and subsequently entered into state).

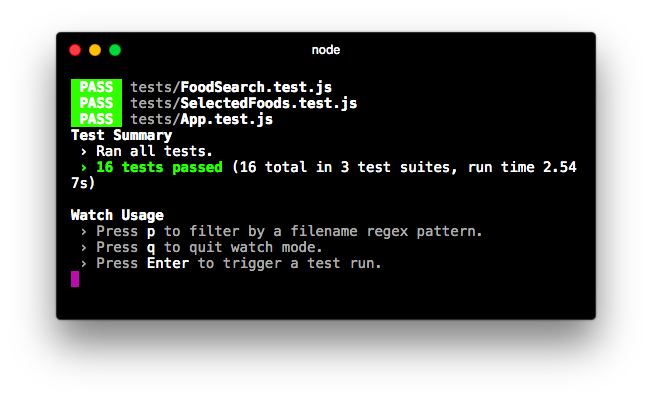
Our assertions would have been robust even if we’d only used one item. However, when writing assertions against arrays some developers prefer that the array has more than one item. This can help catch certain classes of bugs and asserts that the variable under test is the appropriate data structure.

Try it out

Save FoodSearch.test.js. From your console:

$ npm test

Everything passes:



Tests pass

From here, there are a few behaviors the user can take with respect to the FoodSearch component:

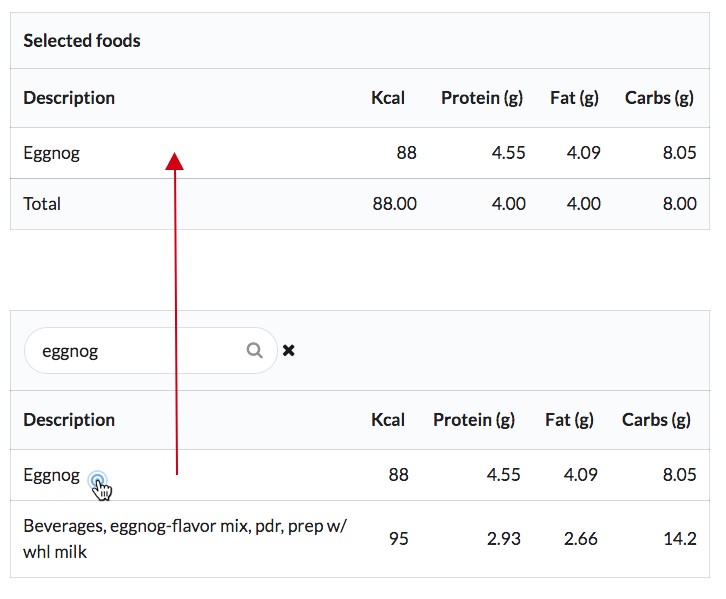
* They can click on a food item to add it to their total
* They can type an additional character, appending to their search string
* They can hit backspace to remove a character or the entire string of text
* They can click on the “X” (remove icon) to clear the search field

We’re going to write specs for the first two behaviors together. The last two behaviors are left as exercises at the end of this module.

We’ll start with simulating the user clicking on a food item.

### The user clicks on a food item

When the user clicks on a food item, that item is added to their list of totals. Those totals are displayed by the SelectedFoods component at the top of the app:



Clicking on an item

As you may recall, each food item is displayed in a tr element that has an onClick handler. That onClick handler is set to a prop-function that App passes to FoodSearch:

**food-lookup-complete/client/src/FoodSearch.js**

<tbody>

{ **this**.state.foods.map((food, idx) => (

<tr

key={idx}

onClick={() => **this**.props.onFoodClick(food)} >

We want to simulate a click and assert that FoodSearch calls this prop-function.

Because we’re unit testing, we don’t want App to be involved. Instead, we can set the prop onFoodClick to a mock function.

At the moment, we’re rendering FoodSearch without setting any props:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-6.js**

beforeEach(() => { wrapper = shallow( <FoodSearch />

);

});

We’ll begin by setting the prop onFoodClick inside our shallow render call to a new mock function:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-7.js**

describe('FoodSearch', () => { let wrapper;

**const onFoodClick = jest.fn();**

beforeEach(() => {

wrapper = shallow(

**<FoodSearch onFoodClick={onFoodClick}**

**/>**

);

});

We declare onFoodClick, a mock function, at the top of our test suite’s scope and pass it as a prop to FoodSearch.

While we’re at it, let’s make sure to clear our new mock between spec runs. This is always good practice:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-7.js**

afterEach(() => {

Client.search.mockClear(); **onFoodClick.mockClear();**

});

Next, we’ll setup the describe ‘then user clicks food item.’ This describe is a child of “and API returns results.”

describe('FoodSearch', () => { // ...

describe('user populates search field', () => { // ...

describe('and API returns results', () => { // ...

**describe('then user clicks food item', () => {**

**beforeEach(() => {**

**// ... simulate the click**

**});**

**// ... specs**

**});**

});

});

});

Our beforeEach block simulates a click on the first food item in the table:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-7.js**

describe('then user clicks food item', () => {

beforeEach(() => {

**const** foodRow = wrapper.find('tbody tr').first(); foodRow.simulate('click');

});

We first use find() to select the first element that matches tbody tr. We then simulate a click on the row. Note that we do not need to pass an event object to simulate().

By using a mock function as our prop onFoodClick, we are able to keep FoodSearch in total isolation. With respect to unit tests for FoodSearch, we don’t care how App implements onFoodClick(). We only care that FoodSearch invokes this function at the right time with the right arguments.

Our spec asserts that onFoodClick was invoked with the first food object in the foods array:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-7.js**

it('should call prop `onFoodClick` with `food`', () => {

**const** food = foods[0]; expect(

onFoodClick.mock.calls[0]

).toEqual([ food ]);

});

**In full,** this describe block:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-7.js**

it('should render the description of second food', () => {

expect(

wrapper.html()

).toContain(foods[1].description);

});

**describe('then user clicks food item', () => {**

**beforeEach(() => {**

**const foodRow = wrapper.find('tbody tr').first(); foodRow.simulate('click');**

**});**

**it('should call prop `onFoodClick` with `food`', () => {**

**const food = foods[0]; expect(**

**onFoodClick.mock.calls[0]**

**).toEqual([ food ]);**

**});**

**});**

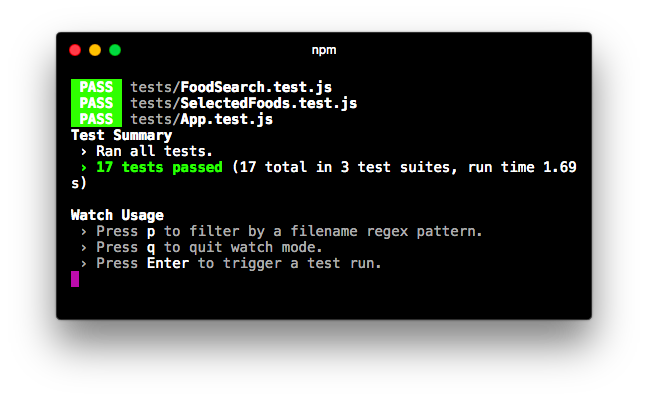
describe('then user types more', () => {

Try it out

Save FoodSearch.test.js and run the suite:

$ npm test

Our new spec passes:



Tests pass

With our spec for the user clicking on a food item completed, let’s return to the context of “and API returns results.” The user has typed 'brocc' and sees two results. The next behavior we want to simulate is the user typing an additional character into the search field. This will cause our (mocked) API to return an empty result set (no results).

### The API returns empty result set

As you can see in the scaffold, our last describe blocks are children to “and API returns results,” siblings to “then user clicks food item”:

describe('FoodSearch', () => { // ...

describe('user populates search field', () => { // ...

describe('and API returns results', () => { // ...

describe('then user clicks food item', () => {

// ...

});

**describe('then user types more', () => {**

**beforeEach(() => {**

**// ... simulate user typing "x"**

**});**

**describe('and API returns no results', () => { beforeEach(() => {**

**// ... simulate API returning no results**

**});**

**// ... specs**

**});**

**});**

});

});

});

We could have combined “then user types more” and “and AP I returns no results” into one describe with one beforeEach. But we like organizing our contextual setup in this manner, both for readability and to leave room for future specs.

After establishing the context in both beforeEach blocks, we’ll write one assertion: that the foods property in state is now a blank array.

If you’re feeling comfortable, try composing these describe blocks yourself and come back to verify your solution.

Our first beforeEach block first simulates the user typing an “x,” meaning the event object now carries the value 'broccx':

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-8.js**

describe('then user types more', () => {

**const** value = 'broccx';

beforeEach(() => {

**const** input = wrapper.find('input').first(); input.simulate('change', {

target: { value: value },

});

});

We won’t write any specs specific to this describe. Our next describe, “and API returns no results,” will simulate Client.search() yielding a blank array:

describe('and API returns no results', () => { beforeEach(() => {

*// ... simulate search returning no results* });

});

Here’s the tricky part: By the time we’ve reached this beforeEach block, we’ve simulated the user changing the input twice. As a result, Client.search() has been invoked twice.

Another way to look at it: If we were to insert a log statement in this beforeEach for Client.search.mock.calls:

describe('and API returns no results', () => { beforeEach(() => {

*// What happens if we log the mock calls here?* console.log(Client.search.mock.calls);

});

});

We would see in the console that it has been invoked twice:

[

[ 'brocc', [Function] ],

[ 'broccx', [Function] ],

]

This is because the beforeEach blocks for “user populates search field” and “then user types more” simulate changing the input which in turn eventually calls Client.search().

We want to invoke the callback function passed to the second invocation. That corresponds to the most recent input field change that the user made. Therefore, we’ll grab the second invocation and invoke the callback passed to it with a blank array:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-8.js**

describe('and API returns no results', () => {

beforeEach(() => {

**const** secondInvocationArgs = Client.search.mock.calls[1];

**const** cb = secondInvocationArgs[1];

cb([]);

wrapper.update();

});

We did not need to call wrapper.update() in the beforeEach block as we’re not making any assertions against the virtual DOM. However, it’s good practice to follow an async state change with an update() call. It will avoid possibly bewildering behavior should you add specs that assert against the DOM in the future.

Finally, we’re ready for our spec. We assert that the state property foods is now an empty array:

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-8.js**

it('should set the state property `foods`', () => {

expect(

wrapper.state().foods

).toEqual([]);

});

The “then user types more” describe**, in full:**

**food-lookup-complete/client/src/tests/complete/FoodSearch.test.complete-8.js**

it('should call prop `onFoodClick` with `food`', () => {

const food = foods[0]; expect(

onFoodClick.mock.calls[0]

).toEqual([ food ]);

});

});

**describe('then user types more', () => {**

**const value = 'broccx';**

**beforeEach(() => {**

**const input = wrapper.find('input').first(); input.simulate('change', {**

**target: { value: value },**

**});**

**});**

**describe('and API returns no results', () => {**

**beforeEach(() => {**

**const secondInvocationArgs = Client.search.mock.calls[1]; const cb = secondInvocationArgs[1];**

**cb([]);**

**wrapper.update();**

**});**

**it('should set the state property `foods`', () => {**

**expect(**

**wrapper.state().foods**

**).toEqual([]);**

**});**

**});**

**});**

});

});

});

Assertions on the component’s output, like that it should not contain any rows, aren’t strictly necessary here. Our assertions against the initial state (like “should display zero rows”) already provide assurance that when foods is empty in state no rows are rendered.



 As you recall, both callback functions look like this:

(foods) => { **this**.setState({

foods: foods.slice(0, MATCHING\_ITEM\_LIMIT), });

};

Because the callback function does not reference any variables inside onSearchChange(), we could technically invoke either callback function and the spec we just wrote would pass. However, this is bad practice and would likely set you up for a puzzling bug in the future.

1. <https://github.com/facebookincubator/create-react-app/issues/3199> [↑](#footnote-ref-1)
2. <http://airbnb.io/enzyme/docs/api/selector.html> [↑](#footnote-ref-2)