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.

What is Jest?

JavaScript has a variety of testing libraries that pack a bunch of great features. These libraries help us organize our test suite in a robust, maintainable manner. Many of these libraries accomplish the same domain of tasks but with different approaches.

An example of testing libraries you may have heard of or worked with are Mocha, Jasmine, QUnit, Chai, and Tape.

We like to think of testing libraries as having three major components:

- The test runner. This is what you execute in the command-line. The test runner is responsible for finding your tests, running them, and reporting results back to you in the console.
- A domain-specific language for organizing your tests. As we'll see, these functions help us perform common tasks like orchestrating setup and teardown before and after tests run.
- An assertion library. The assert functions provided by these libraries help us easily make otherwise complex assertions, like checking equality between JavaScript objects or the presence of certain elements in an array.

React developers have the option to use any JavaScript testing framework they'd like for their tests. In this book, we'll focus on one in particular: **Jest**.

Facebook created and maintains Jest. If you've used other JavaScript testing frameworks or even testing frameworks in other programming languages, you'll likely find Jest quite familiar.

For assertions, Jest uses **Jasmine's assertion library**. If you've used Jasmine before, you'll be pleased to know the syntax is exactly the same.

Later in the chapter, we explore what's arguably Jest's biggest difference from other JavaScript testing frameworks: mocking.

Using Jest

Inside of `testing/basics/package.json`, you'll note that Jest is already included.

As of Jest 15, Jest will consider any file that ends with `*.test.js` or `*.spec.js` a test. Because our file is named `Modash.test.js`, we don't have to do anything special to instruct Jest that this is a test file.

We'll rewrite the specs for `Modash` using Jest.

Jest 15

If you've used an older version of Jest before, you might be surprised that our tests do not have to be inside a `__tests__/` folder. Furthermore, later in the chapter, you'll notice that Jest's auto-mocking appears to be turned off.

Jest 15 shipped new defaults for Jest. These changes were motivated by a desire to make Jest easier for new developers to begin using while maintaining Jest's philosophy to require as little configuration as necessary.

You can read about all the changes in this blog post[a]. Relevant to this chapter:

- In addition to looking under `__tests__/` for test files Jest also looks for files matching `*.test.js` or `*.spec.js`
- Auto-mocking is disabled by default

[a] https://facebook.github.io/jest/blog/2016/09/01/jest-15.html

`expect()`

In Jest, we use `expect()` statements to make assertions. As you'll see, the syntax is different than the assert function we wrote before.

Because Jest uses the Jasmine assertion library, these matchers are technically a feature of *Jasmine*, not *Jest*. However, to avoid confusion, throughout this chapter we'll refer to everything that ships with Jest – including the Jasmine assertion library – as Jest.

Here's an example of using the `expect` syntax to assert that `true` is... `true`:

```
expect(true).toBe(true)
```

`toBe` is a **matcher**. Jest ships with a few different matchers. Under the hood, the `toBe` matcher uses the `===` operator to check equality. So these all work as expected:

```
expect(1).toBe(1); // pass
const a = 5;
expect(a).toBe(5); // pass
```

Because it just uses the === operator, `toBe` has its limitations. For instance, while we can use `toBe` to check if an object is the *exact same* object:

```
const a = { espresso: '60ml' };
const b = a;
expect(a).toBe(b); // pass
```

What if we wanted to check if two *different* objects were identical?

```
const a = { espresso: '60ml' };
expect(a).toBe({ espresso: '60ml' }) // fail
```

Jest has another matcher, `toEqual`. `toEqual` is more sophisticated than `toBe`. For our purposes, it will allow us to assert that two objects are identical, even if they aren't the exact same object:

```
const a = { espresso: '60ml' };
expect(a).toEqual({ espresso: '60ml' }) // pass
```

We'll use both `toBe` and `toEqual` in this chapter. We tend to use `toBe` for boolean and numeric assertions and `toEqual` for everything else. We could just use `toEqual` for everything. But we use `toBe` in certain situations as we like how it reads in English. It's a matter of preference. The important part is that you understand the difference between the two.

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 chapter, `describe` is used to organize assertions that all pertain to the same feature or context. `it` blocks are our individual assertions or **specs**.

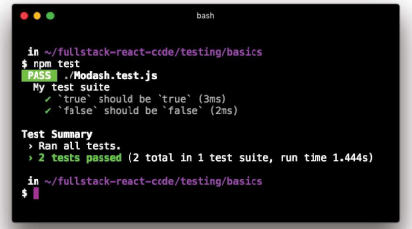
Jest requires that you always have a top-level `describe` that encapsulates all your code. Here, our top-level `describe` is titled "My test suite." The two `it` blocks nested inside of this `describe` are our specs. This organization is standard: `describe` blocks don't contain assertions, `it` blocks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

Throughout the rest of this chapter, 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

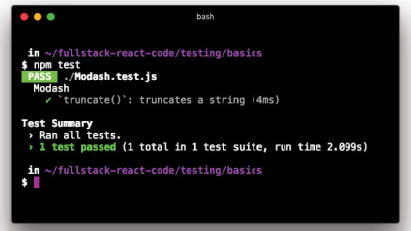
```
describe('Modash', () => {
  it('`truncate()`: truncates a string', () => {
    const string = 'there was one catch, and that was CATCH-22';
    expect(
      Modash.truncate(string, 19)
    ).toEqual('there was one catch...');
  });
});
```

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

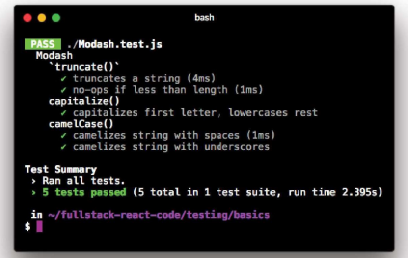
```
  describe('camelCase()', () => {
    it('camelizes string with spaces', () => {
      const string = 'customer responded at';
      expect(
        Modash.camelCase(string)
      ).toEqual('customerRespondedAt');
    });

    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 strategies for React applications

In software testing, there are two primary categories that tests fall into: *integration tests* and *unit tests*.

Integration vs Unit Testing

Integration tests are tests where multiple modules or parts of a software system are tested together. For a React app, we can think of each component as an individual module. Therefore, an integration test would involve testing our app as a whole.

Integration tests might go even further. If our React app was communicating with an API server, integration tests could involve communicating with that server as well. Developers often like to call these types of integration tests **end-to-end** tests.

There are a few ways to drive end-to-end tests. One popular method is to use a driver like Selenium to programatically load your app in a browser and automatically navigate your app's interface. You might have your program click on buttons or fill out forms, asserting what the page looks like after these interactions. Or you might make assertions on the resulting state of the datastore over on the server.

Integration tests are an important component of a comprehensive test suite for a large software system. However, in this book, we'll focus exclusively on **unit testing** for our React applications.

In a unit test, modules of a software system are tested **in isolation**.

For React components, we'll make two kinds of assertions:

1. Given a set of inputs (state & props), assert what a component should output (render).
2. Given a user action, assert how the component behaves. The component might make a state update or call a prop-function passed to it by a parent.

Shallow rendering

When rendered in the browser, our React components are written to the DOM. While we typically see a DOM visually in a browser, we could load a "headless" one into our test suite. We could use the DOM's API to write and read React components as if we were working directly with a browser. But there's an alternative: **shallow rendering**.

Normally, when a React component renders it first produces its virtual DOM representation. This virtual DOM representation is then used to make updates to an actual DOM.

When a component is shallow rendered, it does not write to a DOM. Instead, it maintains its virtual DOM representation. You can then make assertions against this virtual DOM much like you would an actual one.

Furthermore, your component is rendered only one level deep (hence "shallow"). So if the render function of your component contains children, those children won't actually be rendered. Instead, the virtual DOM representation will just contain references to the un-rendered child components.

React provides a library for shallow rendering React components, **`react-test-renderer`**. This library is useful, but is a bit low-level and can be verbose.

Enzyme is a library that wraps `react-test-renderer`, providing lots of handy functionality that is helpful for writing React component tests.

Enzyme

Enzyme was initially developed by Airbnb and is gaining widespread adoption among the React open-source community. In fact, Facebook recommends the utility in its documentation for `react-test-renderer`. Following this trend, we'll be using Enzyme as opposed to `react-test-renderer` throughout this chapter.

Enzyme, through `react-test-renderer`, allows you to shallow render a component. Instead of using `ReactDOM.render()` to render a component to a real DOM, you use Enzyme's `shallow()` to shallow render it:

```
const wrapper = Enzyme.shallow(
  <App />
);
```

As we'll see soon, `shallow()` returns an `EnzymeWrapper` object. Nested inside of this object is our shallow-rendered component in its virtual DOM representation. `EnzymeWrapper` gives us a bunch of useful methods for traversing and writing assertions against the component's virtual DOM.

If you ever want to use `react-test-renderer` directly in the future, you'll find knowing Enzyme helps. Because Enzyme is a lightweight wrapper on top of `react-test-renderer`, the APIs have a lot in common.

There are a two primary advantages to shallow rendering:

It tests components in isolation

This is preferable for unit tests. When we are writing tests for a parent component, we don't have to worry about dependencies on child components. A change made to a child component might break the child component's unit tests but it won't break that of any parents.

It's faster

Another nice benefit is that your tests will be faster. Rendering to, manipulating, and reading from an actual DOM adds overhead. With shallow rendering, you avoid the DOM entirely.

As we'll see, Enzyme has an API for simulating DOM events for shallow rendered components. These allow us to, for example, "click" a component even though no DOM is present.

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
```

We cover create-react-app in detail in the previous chapter, "Using Webpack with create-react-app".

Take a look at the directory:

```
$ ls
index.html
node_modules/
package.json
src/
```

And `src/`:

```
$ ls src/
App.css
App.js
App.test.js
complete/
index.css
index.js
semantic/
```

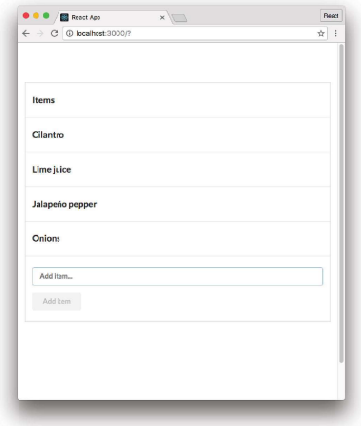
The basic organization of this create-react-app app is the same that we saw in the last chapter. `App.js` defines an `App` component. `index.js` calls `ReactDOM.render()`. Semantic UI is included for styling.

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:

testing/react-basics/src/App.js

```
          <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 more info on controlled inputs, see the section "Uncontrolled vs. Controlled Components" in the *Forms* chapter.

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

testing/react-basics/src/App.js

```
  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:

testing/react-basics/src/App.js

```
  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 chapter, we noted that create-react-app sets up a few commands in `package.json`. One of those was `test`.

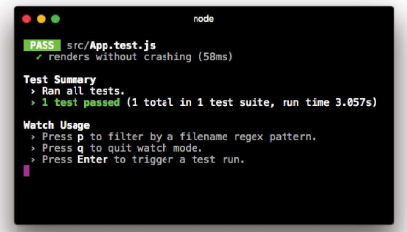
`react-scripts` already specifies Jest as a dependency. To boot Jest, we just need to run `npm test`. Like other commands that create-react-app creates for us, `test` runs a script in `react-scripts`. This script configures and executes Jest.

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 chapter, 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.

`react-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.

Let's 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.

If you need a refresher on the ES6 import syntax, refer to the previous chapter "Using Webpack with create-react-app."

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

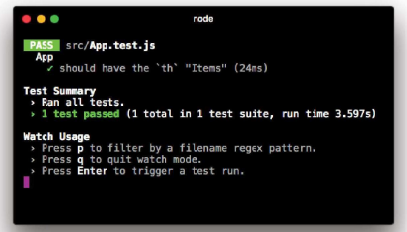
```
it('should have the `th` "Items"', () => {
  const wrapper = shallow(
    <App />
  );
  expect(
    wrapper.contains(<th>Items</th>)
  ).toBe(true);
});
```

`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 chapter, 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 chapter, but Enzyme selectors can also refer directly to React components. For more info on Enzyme selectors, see the Enzyme docs[71].

`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 we set the variable `input` to.

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.

CSS selectors

CSS files use selectors to specify which HTML elements a set of styles refers to. JavaScript applications also use this syntax to select HTML elements on a page. Check out this MDN section[72] for more info on CSS selectors.

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.

[71] http://airbnb.io/enzyme/docs/api/selector.html
[72] https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_started/Selectors

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 "hoisted" it up into 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);
  });
```

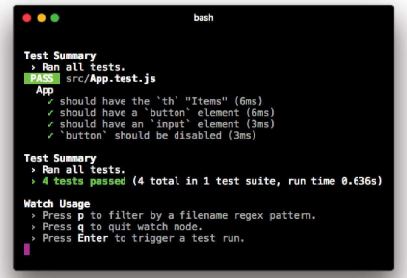
Much better. Our `it` blocks are no longer setting up context and we've removed redundant code.

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 chapter, 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 call `first()` 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`).
2. 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

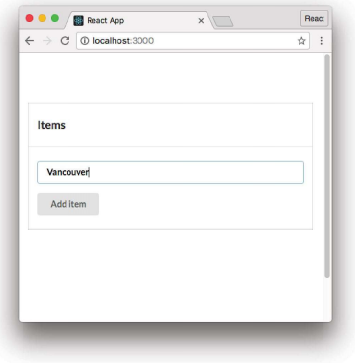
```
  describe('the user populates the input', () => {
    const item = 'Vancouver';

    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.

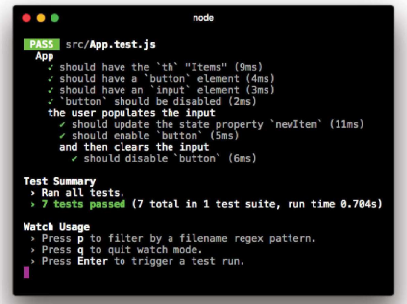
Now's a good time to verify all our tests pass.

Try it out

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

```
$ npm test
```

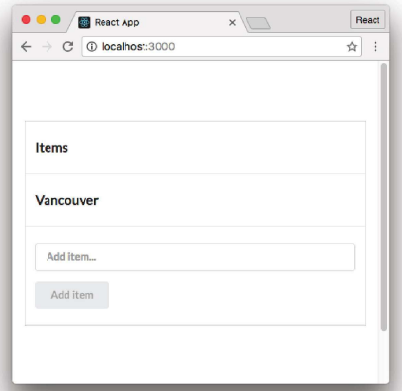
We see everything passes:



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);
      });
```

`contains()` would work for this spec as well, but we tend to use `containsMatchingElement()` more often to keep our tests from being too brittle. For instance, if we added a `class` to each of our `td` elements, a spec using `containsMatchingElement()` would not break.

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);
    });
  });
});
```

It might appear we have another possible refactor we can do. We have a lot of these declarations throughout our test suite:

```
const input = wrapper.find('input').first();
const button = wrapper.find('button').first();
```

You might wonder if it's possible to declare these variables at the top of our test suite's scope, alongside `wrapper`. We could then set them in our top-most `beforeEach`, like this:

```
// Valid refactor?
describe('App', () => {
  let wrapper;
  let input;
  let button;

  beforeEach(() => {
    wrapper = shallow(
      <App />
    );
    const input = wrapper.find('input').first();
    const button = wrapper.find('button').first();
  });
  // ...
});
```

Then, you'd be able to reference `input` and `button` throughout the test suite without re-declaring them.

However, if you were to try this, you'd note some test failures. This is because throughout the test suite, `input` and `button` would reference HTML elements *from the initial render*. When we call a `simulate()` event, like this:

```
  input.simulate('change', {
    target: { value: item }
  });
```

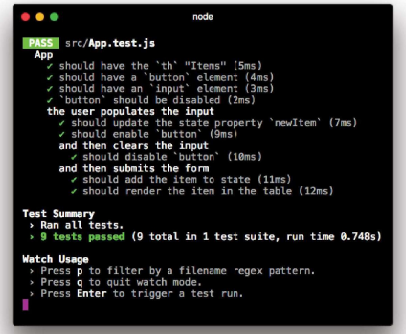
Under the hood, the React component re-renders. This is what we'd expect. Therefore, an entirely new virtual DOM object is created with new `input` and `button` elements inside. We need to perform a `find()` to pick out those elements inside the new virtual DOM object, which we do here.

Try it out

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

```
$ npm test
```

Everything passes:



You might try breaking various parts of `App` and witness the test suite catch these failures.

Our test suite for `App` is sufficiently comprehensive. We saw how we can use a behavioral-driven approach to drive the composition of a test suite. This style encourages completeness. We establish layers of context based on real-world workflows. And with the context established, it's easy to assert the component's desired behavior.

In total, so far we've covered:

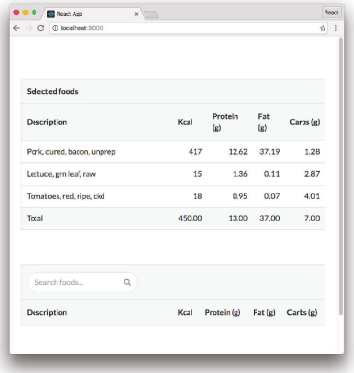
- The basics of assertions
- The Jest testing library (with Jasmine assertions)
- Organizing testing code in a behavioral-driven manner
- Shallow rendering with Enzyme
- `ShallowWrapper` methods for traversing the virtual DOM
- Jest/Jasmine matchers for writing different kinds of assertions (like `toContain()` for arrays)

In the next section, we'll advance our understanding of writing React unit tests with Jest and Enzyme. We'll write specs for a component that exists inside of a larger React app. Specifically, we'll cover:

- What happens when an app has multiple components
- What happens when an app relies on a web request to an API
- Some additional methods for both Jest and Enzyme

Writing tests for the food lookup app

In the previous chapter 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-lookup-complete`. To get to it from the `testing/react-basics` folder, run the following command:

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

It's not necessary that you've completed the chapter on Webpack to proceed. We describe this app's layout and the `FoodSearch` component before writing our specs.

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 chapter. 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** (blue): The parent container for the application.
- **SelectedFoods** (yellow): A table that lists selected foods. Clicking on a food item removes it.
- **FoodSearch** (purple): 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 chapter:

```
$ 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:

```
$ ls tests/
  App.test.js
  SelectedFoods.test.js
  complete/
```

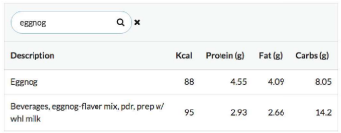
Feel free to peruse the other tests after we've finished writing tests for `FoodSearch`. All the other tests re-use the same concepts that we use for testing `FoodSearch`.

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.

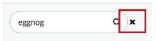
`FoodSearch` has three pieces of state:

`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 Chapter 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 Chapter 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/tests/FoodSearch.test.js` contains the scaffold for the test suite. At the top are the `import` statements:

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

```
import { shallow } from 'enzyme';
import React from 'react';
import FoodSearch from '../src/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/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/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/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 chapter, 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/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.

How comprehensive should my assertions be?

In the last section, we wrote assertions around the presence of key elements in the component's initial output. We asserted that the input field and button were present, setting the stage for interacting with them later.

In this section, we're skipping this class of assertions. We omit them here as they are repetitive. But, in general, you or your team will have to decide how comprehensive you want your test suite to be. There's a balance to strike; your test suite is there to service development of your app. It is possible to go overboard and compose a test suite that ultimately slows you down.

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

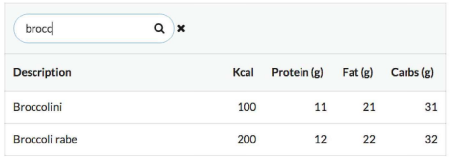
This branch happens at the bottom of `onSearchChange()` when we call `Client.search()`:

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'

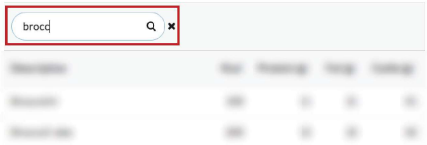
We'll then write assertions against this context.

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 API 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/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/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/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-to-end 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/tests/complete/FoodSearch.test.complete-3.js

```
import FoodSearch from '../src/FoodSearch';
import Client from '../src/Client';

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

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

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

food-lookup-complete/client/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.

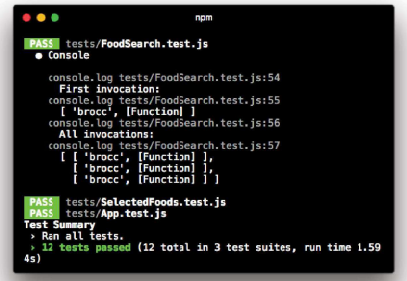
Let's declare a new spec below "should display the remove icon." Before writing the assertion, let's just log a few things out to the console to see what's happening:

food-lookup-complete/client/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/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/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', () => {
```

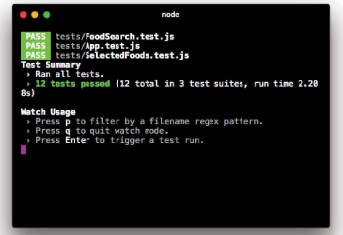
We're asserting that the zeroeth argument of the invocation matches `value`, in this case `brocc`.

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.

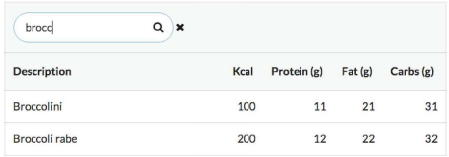
The API returns results

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/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 chapter, 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/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/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/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 HTML 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 API (and subsequently entered into state).

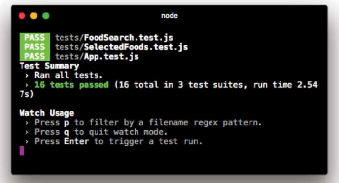
Our assertions would have been robust even if we'd only used one item. However, when writing assertions against arrays some developers prefer that the array has more than one item. This can help catch certain classes of bugs and asserts that the variable under test is the appropriate data structure.

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

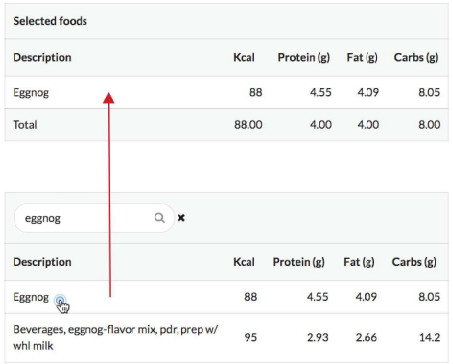
- They can hit backspace to remove a character or the entire string of text
- They can click on the "X" (remove icon) to clear the search field

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 chapter.

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/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/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/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/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/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/tests/complete/FoodSearch.test.complete-7.js

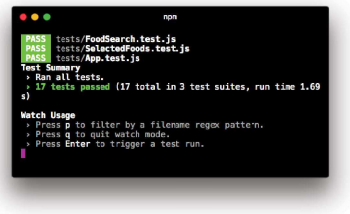
```
      it('should render the description of second food', () => {
        expect(
          wrapper.html()
        ).toContain(foods[1].description);
      });

      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 `'broccʻ` 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 API 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/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/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/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/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.

Further reading

In this chapter, we:

1. Demystified JavaScript testing frameworks, building from the ground up.
2. Introduced Jest, a testing framework for JavaScript, to give us some handy features like `expect` and `beforeEach`.
3. Learned how to organize code in a behavior-driven style.

4. Introduced Enzyme, a library for working with React components in a testing environment.
5. Used the idea of mocks to write assertions for a React component that makes a request to an API.

Armed with this knowledge, you're prepared to isolate React components in a variety of different contexts and effectively write unit tests for them. These unit tests will give you peace of mind as the number and complexity of components in your app grows.

A few resources outside of this chapter will aid you greatly as you compose unit tests:

Jest API reference[73]

These docs will help you discover rich matchers that can both save you time and elevate the expressiveness of your test suite. We used some handy matchers in this chapter, like `toEqual` and `toContain`. Here are a few more examples:

- You can assert that one number is close to another with `toBeCloseTo()`
- You can match a string against a regular expression with `toMatch()`
- You can control time, allowing you to work with `setTimeout` or `setInterval`

create-react-app configures Jest for you. However, if you use Jest outside of create-react-app, you'll also find the reference for Jest configuration useful. You can configure settings like test watching or instruct Jest where to find test files.

Jasmine docs[74]

Jest uses Jasmine assertions, so the docs for Jasmine also apply. You can use that as another reference point to understand matchers.

Furthermore, you can use some additional functionality that's not mentioned in the Jest API reference. For example:

- You can assert that an array contains a specific subset of members with `jasmine.arrayContaining()`
- You can assert that an object contains a specific subset of key/value pairs with `jasmine.objectContaining()`

Enzyme `ShallowWrapper` API docs[75]

We explored a few methods for traversing the virtual DOM (with `find()`) and making assertions on the virtual DOM's contents (like with `contains()`). `ShallowWrapper` has many more methods that you might find useful. Some examples:

[73] http://facebook.github.io/jest/docs/api.html
[74] http://jasmine.github.io/2.5/introduction.html
[75] http://airbnb.io/enzyme/docs/api/shallow.html

- As we saw in this chapter, `ShallowWrapper` is array-like. A `find()` call might match more than one element in a component's output. You can perform operations on this list of matching elements that mirror Array's methods, like `map()`.
- You can grab the actual React component itself with `instance()`. You can use this to unit test particular methods on the component.
- You can set the state of the underlying component with `setState()`. When possible, we like to use `simulate()` or call the component's methods directly to invoke state changes. But when that's not practical, `setState()` is useful.

Earlier, we saw that `find()` accepts an Enzyme selector. The docs contain a reference page for what constitutes a valid Enzyme selector that you'll find helpful.

We have end-to-end tests that we use for the code in this book. We use the tool Nightwatch.js[76] to drive these.

We'll be adding more unit tests to existing projects over the next couple of months. Perusing these test suites will provide you with even more examples to work with.

Chapter Exercises

We mentioned earlier in the chapter that there were two further actions a user can take after `FoodSearch` displays a list of food results:

1. The user can hit backspace to remove a character or the entire string of text
2. The user can click on the "x" (remove icon) to clear the search field

Compose specs for each of these situations.

A possible set of solutions is available in `client/tests/FoodSearch.test.complete.js`.

[76] http://nightwatchjs.org/

Routing

What's in a URL?

A URL is a reference to a web resource. A typical URL looks something like this:

While a combination of the protocol and the hostname direct us to a certain website, it's the **pathname** that references a specific resource on that site. Another way to think about it: the pathname references a specific **location** in our application.

For example, consider a URL for some music website:

```
https://example.com.com/artists/87589/albums/1758221
```

This location refers to a specific album by an artist. The URL contains identifiers for both the artist and album desired:

```
example.com/artists/:artistId/albums/:albumId
```

We can think of the URL as being an **external keeper of state**, in this case the album the user is viewing. By storing pieces of app state up at the level of the browser's location, we can enable users to bookmark the link, refresh the page, and share it with others.

In a traditional web application with minimal JavaScript, the request flow for this page might look like this:

1. Browser makes a request to the server for this page.
2. The server uses the identifiers in the URL to retrieve data about the artist and the album from its database.
3. The server populates a template with this data.
4. The server returns this populated HTML document along with any other assets like CSS and images.
5. The browser renders these assets.