SAMPLE CHAPTER

React IN ACTION

Mark Tielens Thomas

MANNING

React in Action

by Mark Tielens Thomas

Chapter 9

brief contents

9 Testing React components

This chapter covers

- Testing front-end applications
- Setting up testing for React
- Testing React components
- Setting up test coverage

In the last chapter, you added some significant functionality to your application. It now has routing and user state, and you've broken it up into smaller pieces. You even added some basic authentication so users could log in using their GitHub profile. Your application is starting to look more robust, even if it's probably not going to worry anyone at Facebook or Twitter. You can do lots more with React than you could when we first started. But as we've focused on learning the basics, we've omitted an important part of the development process: *testing*.

I didn't cover testing from the start to spare you the mental overload of learning React and testing fundamentals at the same time. But that doesn't mean it's an unimportant part of either learning or web development. In this chapter, we'll focus on testing because it's a fundamental part of developing high-quality software solutions. Instead of demonstrating tests for every single one of your components, though, we'll go through a representative sample so you'll understand the important principles at work and be able to write your own tests.

By the end of this chapter, you'll understand some of the basic principles of testing web applications. You'll also have set up tests and a test runner, worked with Jest, Enzyme, and the React test renderer, and learned to use and understand test coverage tools. You'll be equipped to start testing your applications, which will add another level of confidence to your React development skills.

How do I get the code for this chapter?

As with every chapter, you can check out the source code for this chapter by going to the GitHub repository at https://github.com/react-in-action/letters-social. If you want to start this chapter with a clean slate and follow along, you can use your existing code from chapters 7 and 8 (if you followed along and built out the examples yourself) or check out the chapter-specific branch (chapter-9).

Remember, each branch corresponds to the code at the end of the chapter (for example, the branch chapter-7 corresponds to the code as it will be at the end of this chapter). You can execute one of the following terminal commands in the directory of your choice to get the code for the current chapter.

If you don't have the repository at all, type the following:

```
git clone git@github.com:react-in-action/letters-social.git
```

If you already have the repository cloned:

```
git checkout chapter-9
```

You may have come here from another chapter, so it's always a good idea to ensure you have all the right dependencies installed:

```
npm install
```

Testing in software development is the process of validating assumptions. For example, say you're building an application (like Medium, Ghost, or WordPress) that lets users write and create blog posts. Users pay a monthly fee and get the hosting and the tools to run their own blog. When creating the front-end of the application, there are several key things it *must* do (among others), including correctly displaying those posts and letting users edit them.

How can you be sure your app is doing what it needs to do? You can try it out yourself and see if it works. Click around, edit things, and use the application in as many ways as you can think of. This manual process works reasonably well and is a first line of defense against bugs and regressions. You should always take care to inspect what you're working on, but you can't test things quickly and or in a perfectly consistent manner.

Also, as your application grows, the number of situations and features you'll need to manually test increases at an incredible rate. I've worked on applications with thousands of tests, but there are many applications where that number would be easily

dwarfed. The React library itself has 4,855 tests at the time of writing. There's no chance someone wanting to test React would be able to validate by hand the assumptions involved in all those tests.

Fortunately, instead of testing everything by hand, you can use software to test software. Computers excel where we fail in at least two important areas: speed and consistency. We can use software to test our code in ways that we never could by hand, even with an army of people trying things out in every possible way. You may be thinking, "My project is small and really straightforward—there's not much that could go wrong." But even as great as your coding skills may be, bugs are inevitable. Your apps will break and work in unpredictable ways when you change things (and sometimes even when you don't).

But instead of despairing about the inevitability of bugs, we can accept that they'll happen and take steps to minimize their impact and frequency. That's where testing comes in. You may have some general idea about what testing is, but to get started we'll need to explore some different types of testing. Bear in mind that the world of testing is huge, and I can't cover even close to everything here. I won't be doing any in-depth coverage of testing as a domain. I also won't be deeply covering several types of testing, including integration testing, regression testing, testing automation, and others. But by the end of the chapter, you should be familiar enough to get started testing React components in a few different ways.

9.1 Types of testing

As I said, testing software is the process of using software to validate your assumptions. Because you're using software to test software, you'll ultimately be using the same primitives you use when building software: Booleans, numbers, strings, functions, objects, and the like. It's important to remember that there's no magic here—just more code.

There are different types of testing, and you'll use a few to test your React application. They encompass different aspects of an application, and when used together and in the right proportions, they should give you a significant degree of confidence in your application. Different types of tests address different parts and scopes of an application. A well-tested app will test the individual units of functionality that make up the basic parts of the app. It will also test the collections of these units of functionality and, at the highest level, the points at which everything comes together (such as the user interface).

Here are a few types of testing:

- *Unit*—Unit tests focus on individual units of functionality. For example, say you have a utility method for fetching new posts from the server. A unit test will focus only on that one function. It doesn't care about anything else. Like components, these tests allow for refactoring and promote modularity.
- *Service*—Service tests focus on bundles of functionality. This part of the "testing spectrum" can include a variety of granularities and focuses. The point, though,

is that you're testing things that aren't at the highest level (see integration tests, next) or the lowest levels of functionality. An example of a service test might be something like a tool that uses several units of functionality but is not itself at the level of an integration test.

- *Integration*—Integration tests focus on an even higher level of testing: the integration of various parts of an application. They test the way that services and lower-level functionality come together. Typically, these tests test an application through its user interface, not through the individual code behind the user interface. These tests may simulate clicks, user input, and other interactions that drive the application.

You may be wondering what these tests will look like in code; we'll get into that shortly, but first we need to talk about how these tests work together in the overall testing approach. If you've done testing before, you may have heard of the *testing pyramid*. This pyramid, illustrated in figure 9.1, generally refers to the proportion of different types of tests you should write. In this chapter, you'll only be writing unit tests for your components.

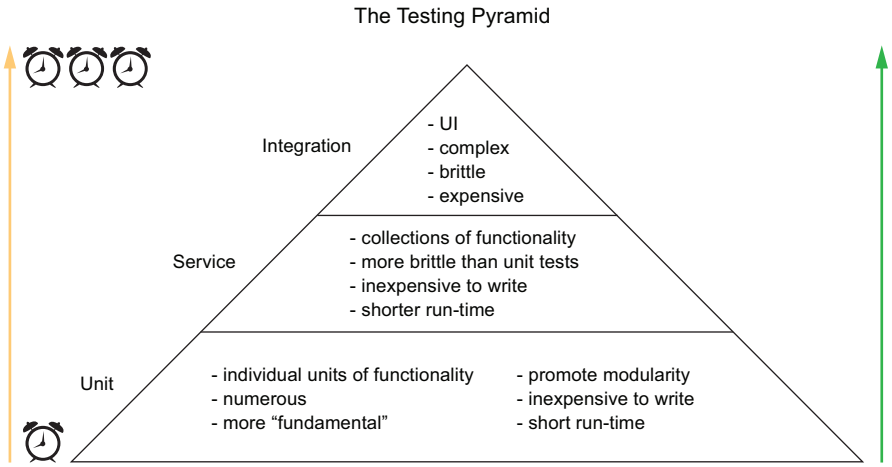


Figure 9.1 The testing pyramid is a way of guiding how many and which types of tests you write as you test your applications. Notice that certain types of tests take longer and are thus more "expensive" in terms of time (and therefore also financial cost).

9.1.1 Why test?

There are some software development paradigms where testing is a "first-class citizen" of the entire development process. That means testing is important, is considered at the beginning and throughout the development process, and usually plays a role in determining when something is considered complete. Granted, the consensus is that testing is a good thing for software development, but there are certain paradigms

where it takes on a central role. For example, you may have heard of *test-driven development* (TDD). When practicing TDD, as its name suggests, the very process of writing software is driven by testing. When working, a developer will usually write a *failing* test (a test that makes assertions that haven't yet been met), write just enough code to get it to pass, refactor any duplication, and then move on to the next feature, repeating the process.

Although you don't have to be a strict practitioner of TDD to write great software, consider some of the benefits before moving on. If you're already wise to the upsides of testing, feel free to move on to the next section where we get started with testing in React. But I want to ask an important question: why do we test at all?

First and foremost, we want to write software that works. There are so many interconnected parts of modern software that it would be foolish to assume that every part of the software stack will reliably work all the time. Things will break, and it's better to assume things will fail than to assume they'll work all the time. We can do our part to minimize the ways that our own software can break by testing our assumptions. Testing forces you to visit (or revisit) your assumptions about your software. You walk it through the different cases it can handle and ensure that it handles them all appropriately.

Secondly, the process of testing your software tends to help you write better code. Going through the process of writing out your tests encourages you to think through what your code does, especially if you do it beforehand (as in TDD). Though it's far less preferable, you can write tests after the fact, too, which is better than having no tests at all. Going through the process of testing will help you better understand the code you write and will validate assumptions you and others make about how things work.

Third, integrating testing into your software development workflow means you can release code more frequently. You may have heard people in the tech industry mention "shipping often" before. That usually means releasing software incrementally and frequently. In the past, companies tended to only release software after an extensive process and only several times a year (or at least relatively infrequently).

Thinking has changed today, and people have realized that incremental iteration leads to generally better results for software: you can get feedback from users and others on it sooner, experiment more easily, and more. The confidence you can have in a well-tested app is a key part of this process. Using *continuous integration* (CI) or *continuous deployment* tools like Circle CI (https://circleci.com), Travis CI (https://travis-ci.org), or others, you can make testing part of the deployment process for your software. The idea is this: *if the tests pass, it gets deployed.* These tools usually run your tests in a pristine environment and, if they pass, send the code off to whatever system runs your application. Figure 9.2 shows the process that the Letters Social app uses to get tested and deployed.

Finally, tests also help you when going back and refactoring your code or moving it around. Say, for example, your requirements change, and you need to move some

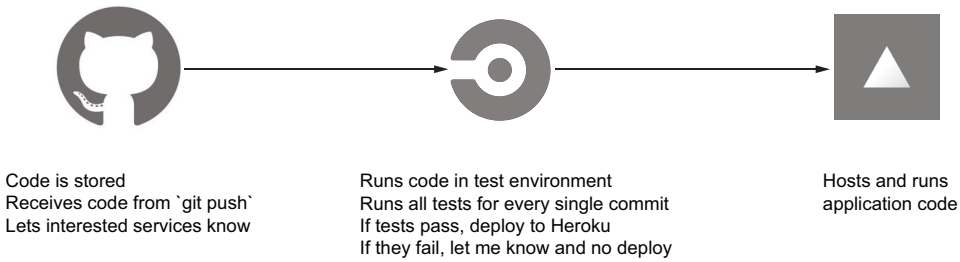


Figure 9.2 The Letters Social deploy pipeline. A CI build process is triggered when I (or anyone who contributes to the repository) *push* code. The CI provider (Circle, in this case) uses Docker containers to run your tests quickly and reliably. If the tests pass, the code will be deployed to whatever service you use to run your code. In our case, that's Now.

components around. If you've kept your components modular and they have good tests, moving them should be easy. Untested code can be moved around, of course, but you have much less of a firm idea of whether it broke other parts of your system than you do when your code is tested.

There's more to be said about the benefits and theory of testing in software, but it's beyond the scope of this book. If you want to learn more, I recommend checking out *The Art of Unit Testing, Second Edition* (Manning Publications, 2013) by Roy Osherove and *Growing Object-Oriented Software: Guided by Tests* by Nat Pryce and Steve Freeman (Addison-Wesley, 2009).

9.2 Testing React components with Jest, Enzyme, and React-test-renderer

Testing software is just more software, made from the same primitives and basic elements that your normal programs are, though people have developed special tools to aid in the testing process. You could try to create the necessary tools to run all your tests, but the open source community has already put an incredible amount of work into a huge number of powerful tools—so you'll use those instead.

You'll need a few types of libraries to test your React applications:

- *Test runner*—You'll need something to run your tests. Most tests could be executed as regular JavaScript files, but you'll want to take advantage of some of the added features of test runners, such as running more than one test at a time and reporting back error or success information in a nicer way.

  For this book, you'll use Jest for most aspects of your testing. Jest is a testing library developed by engineers at Facebook. Some popular alternatives with fewer features built in that you might consider include Mocha (https://mochajs.org) and Jasmine (https://jasmine.github.io). Jest is often used for testing React apps, but adapters are being created for other frameworks, too. The source code includes a setup file (test/setup.js) that invokes the adapter for React.

- *Test doubles*—When writing tests, you want to avoid tying your tests to other fragile or unpredictable parts of your infrastructure as much as possible; other tools you rely on should be *mocked out*—replaced with a "fake" function that behaves in an expected way. Testing this way promotes a focus on the code under test and modularity because your tests aren't tied to the exact structure of your code at a given time. You'll use Jest for mocking and test doubles, but there are other libraries that also do this, such as Sinon (http://sinonjs.org).
- *Assertion libraries*—You can use JavaScript to make assertions about your code (for example, does *X* equal *Y*?), but there are plenty of edge cases that you'll need to account for. Developers have created solutions to make writing assertions about your code easier. Jest comes with assertion methods built in, so you'll rely on those.
- *Environment helpers*—Running tests on code that needs to run in a browser environment places slightly different demands on you. The browser environment is unique and includes things like the DOM, user events, and other normal parts of web applications. These testing tools will help ensure that you can successfully emulate a browser environment. You'll be using Enzyme and the React test renderer to aid in testing your React components. Enzyme makes testing React components easier. It provides a robust API that lets you query for different types of components and HTML elements, set and get props of components, inspect and set component state, and more. The React test renderer does similar things and can also generate snapshots of your components. We won't go into every aspect of Enzyme or the React test renderer APIs, but feel free to explore more at http://airbnb.io/enzyme and www.npmjs.com/package/react-test-renderer.
- *Framework-specific libraries*—There are libraries specifically made for React (or other frameworks) that make writing tests for a particular framework easier. These abstractions are usually developed to aid in the testing of a library or framework and handle setting up anything needed by the framework. In React, almost everything is "just JavaScript," so there's still little "magic" to be seen even in these tools.
- *Coverage tools*—Thanks to the deterministic nature of code, people have figured out ways to determine which parts of your code are "covered" by tests. That's great because you can get a metric that serves as a guideline in determining how well tested your code is. It's no substitute for logic and basic analysis (100% code coverage doesn't mean you can't have bugs), but it can guide how you test your code. You'll use Jest's built-in coverage tool, which utilizes a popular tool called Istanbul (https://github.com/gotwarlost/istanbul).

Next, you'll get started by installing the tools you'll be using for your tests. If you cloned the book repository from GitHub, these tools should already be installed. Make sure to run `npm install` again when changing chapters to make sure you have all the libraries for that chapter.

9.3 Writing your first tests

Once you've installed the tools you'll need, you're ready to start writing some tests. In this section, you're going to set up commands to run your tests and start testing some basic React components. You'll make assertions about your components and look at ways to test rendered output of components.

But before diving in, I should note a few things about Jest and where the code for your tests will run. Jest can be configured to run in different environments depending on the sort of tests you're writing. If you're writing tests for React applications that run in the browser, you'll want to tell Jest that so it can provide the virtual browser environment you need to properly emulate a real browser. Jest uses another library, `jsdom`, to accomplish that. If you're writing tests for node.js applications, you don't want the extra memory and baggage of the `jsdom` environment—you just want to test your server-side code. Jest is configured to run browser-oriented tests by default, so you don't need to override anything.

Exercise 9.1 Reviewing types of testing

There are a few different types of testing. To review, try matching the type with the description of the type of testing.

1. Unit
2. Service
3. Integration

__ Complex, often brittle tests that take a long time to write and run. They test the way different systems work together at a high level. There are often fewer of these types of tests than others.

__ Less complex tests that test the way a particular system works, but without interacting with other systems.

__ Low-level, focused tests that focus on testing small bits of functionality. These should be the most numerous tests in a suite.

9.3.1 Getting started with Jest

To run your tests, as mentioned, you'll use Jest. You can run Jest from the command line, and it will execute your tests, so you're going to add a script to your package.json file so you can run it. The next listing shows how to add the custom script to your package.json. If you cloned the repository from GitHub, this script should already be available.

Listing 9.1 Setting up a custom npm script (package.json)

```
{
   //...
  "scripts": {
```

```
    //...
    "test": "jest --coverage",
    "test:w": "jest -watch --coverage",
  "jest": {
    "testEnvironment": "jsdom",
    "setupFiles": ["raf/polyfill", "./test/setup.js"]
  },
  "repository": {
    "type": "git",
    "url": "git+ssh://git@github.com/react-in-action/letters-social.git"
  },
  "author": "Mark Thomas <hello@ifelse.io>",
// ...
```

Run your tests and output test coverage.

Run the tests in watch mode.

Configure Jest; some testing helpers and stubs are included with the sample code.

Now that you have a command in place to run your tests (`npm test`), try it out. You shouldn't get any helpful info back yet because there are no tests to run (Jest should warn you accordingly in your terminal). You can also run `npm run test:w` to run Jest in watch mode. That's helpful when you don't want to manually run your tests every time. Jest's immersive watch mode makes it especially useful to work with—it will do some work to run only tests that relate to changed files. That's helpful if you have a large test suite and don't want to run every test every time. You can also provide regex patterns or search by text string to run only particular tests.

Tooling matters

Testing libraries and even testing as a whole sometimes get last consideration when it comes to evaluating libraries. That's unfortunate for at least two reasons. First, unusable testing libraries can make it more difficult for teams to buy into testing their code, potentially causing them to forgo it altogether. That, in turn, generally results in code that's harder to maintain, less stable, and more difficult to work with overall.

Another downside is that if you or your team spends a lot of time writing tests, your tools can have a substantial impact on your time. That can quickly translate to money lost by the business because its engineers are taking longer to do the work they need to do. I've seen both results firsthand. If testing wasn't considered a top priority from the beginning, it became more and more difficult over time and was treated as a "one day" kind of thing. The result was code that could be more difficult to change with confidence because assumptions about functionality were no longer backed by tests.

Another reason it pays to treat your testing tools as important is that if you do test your code, a significant time investment will be involved. If you have flaky tests or a testing setup that takes a long time to run, you can end up losing large chunks of time on a daily basis. There's no magic solution to this problem, but treating your testing tools and setup as first-class issues will often help you greatly in the long run.

9.3.2 Testing a stateless functional component

Time to get started writing some tests. First, we'll focus on a relatively straightforward example of testing a component. You're going to test the Content component. It doesn't do much; it just handles rendering a paragraph with content inside of it. The next listing shows the structure of the component.

Listing 9.2 Content component (src/components/post/Content.test.js)

```
import React, { PropTypes } from 'react';

const Content = (props) => {
  const { post } = props;
  return (
    <p className="content">
      {post.content}
    </p>
  );
};

Content.propTypes = {
  post: PropTypes.object,
};
export default Content;
```

Component takes in post props object and uses content property of post to render paragraph element

It assigns content class to paragraph

Inner content of paragraph element is content from post

Component is exported—important because you'll need to import component in your tests

One of the first things you can do when starting to write tests is to think about what assumptions you want to validate. That is, once all the tests pass, they should confirm certain things to you and act as a sort of guarantee. In fact, one of my favorite things about tests is that I rely on them to fail when I'm making changes to a particular feature or part of a system. They back up my assumption that the changes I made represent a change to the application or system. This makes me much more comfortable when writing my code because on the one hand I have a record of how things were supposed to work beforehand, and on the other because I can get a sense of how my changes affect the application as a whole.

Let's look at your component and think about how you might test it. There are a few assumptions you want to validate about this component. For one, it needs to render some content that got passed in as a prop. It also needs to assign a class name to a paragraph element. Aside from that, there's not much to the component that you need to focus on. These things should be enough to get you started writing a test.

You may notice that "React works properly" isn't one of the things you're trying to test here. We also excluded things like "A function can be executed," "The JSX transpiler will work," and some other fundamental assumptions about the technologies you're using. These things are important to test, but the tests you're writing could never adequately or accurately validate these assumption. These other projects are responsible for writing their own tests and ensuring that they work. This underscores the importance of choosing software that's reliable, well-tested, and kept up-to-date. If you have serious doubts about React's reliability, those doubts may be unfounded.

Although not perfect, React is used on some of the most popular web apps in the world, including Facebook.com and Netflix.com, to name two. There are certainly bugs, but it's highly unlikely that you'd encounter them in our straightforward situation.

You know a few things about the component you want to validate, but you could have also gone about this the other way if you were starting from scratch and had written the test first. You may have thought to yourself, "We need a component that displays content, has a certain type, and has a certain class name so our CSS works." You may have then proceeded to write the test that would validate these conditions. You're going about it the other way due to how you've been learning about React, but you can see how starting with a test can make things easy: you start out by having to think through and plan your component. As mentioned, test-driven development (TDD) is a school of thought that makes writing tests first a central part of software development.

Let's see how to test this component. To do that, you'll need to write a test *suite*, which is a group of tests. Individual tests make *assertions* (statements about code that can be true or false) to validate assumptions. For example, a test for your component would *assert* that the right class name is set up. If any of your assertions fail, the test fails. That's how you know something has inadvertently changed or no longer works in your app. Listing 9.3 shows how to set up the skeleton of the test.

Notice that the file for the component ends with *.test.js*. That's a convention that you can choose to follow if you like. Jest will look for files that end in .spec.js or .test.js and run those tests by default. If you choose to follow a different convention, you'll need to explicitly tell Jest about which files you want to run by adding them to the command line invocation (`jest --watch ./my.cool.test.file.js`, for example). You'll follow the .test.js convention for all your tests.

It's also good to note where the test files are placed. Some people choose to place all their tests in a "mirror" directory called test, usually located in the root directory of their project. For every file that gets tested, they'll create a corresponding file in the test directory. That's a fine way to structure things, but you can also locate your test files right next to their source files. You'll go with this method, but either way is perfectly fine.

Listing 9.3 Test skeleton for Content component (src/components/post/Content.test.js)

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import { Content } from './Content';

describe('<Content/>', () => {
    test('should render correctly', () => {

    });
});
```

Import React.

Import related helper methods

Import component to be tested

Jest uses Jasmine-style (https://jasmine.github.io/) methods like describe to group tests.

An actual test—the it function is also provided globally by jest

You may have noticed that there's nothing special about the `describe` functions so far. They're primarily for organization and for ensuring that you can split your tests into the appropriate chunks to test different parts of your code. It may not seem like a huge need for such a small file, but I've worked with test files that are 2,000–3,000 lines long (or more), and I can speak from experience: readable tests help make good tests.

Write clean tests!

Have you ever read test code that hasn't gotten the same treatment as the code that it's testing? I've had this happen to me more than once. It can be confusing or even frustrating to read through test code that isn't clean. Tests are just more code, so they still need to be clean and readable, right? I've already mentioned in this chapter that testing can sometimes take second priority to writing application code. Test code can be treated as a task that has to be done or even a barrier between you and the application code, and so standards are lowered. This tendency can be easy to slip into, but the reality is that poorly written tests can be as bad as poorly written application code. Tests should serve as another form of documentation for your code, and one that still has to be read by developers. Remember that test code should still be clean code.

Jest will look for files to test and then execute these different `describe` and `it` functions, calling the callback functions you've provided to them. But what do you need to put inside them? You need to set up *assertions*. To do that, you need something to assert on. This is where Enzyme comes in; it lets you create a testable version of your component that you can inspect and make assertions about. You'll use Enzyme's *shallow rendering*, which will create a lightweight version of your component that doesn't perform full mounting or insertion into the DOM. You also need to provide some *mock* (fake) data for the component to use. The next listing shows how to add the test version of the component to your test suite. Before you start writing your tests, make sure to run the `npm run test:w` command in your terminal to start the test runner.

Listing 9.4 Shallow rendering (src/components/post/Content.test.js)

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import { Content } from './Content';

describe('<Content/>', () => {
  describe('render methods', () =>  {
    it('should render correctly', () =>  {
      const mockPost = {
        content: 'I am learning to test React components',
      };
```

Create dummy post object that component can use

```
      const wrapper = shallow(<Content post={mockPost} />);
    });
  });
});
```

Perform shallow rendering of component and save returned wrapper for later use

You now have a test component set up that you can make assertions about. To do this, you'll use the built-in `expect()` function from Jest. If you were using a different assertion library, you might use something else. Remember from earlier that these assertion libraries are for making assertions easier. For example, checking whether an object is *deeply equal* (meaning equal in every one of its properties) can be an involved task. When writing your tests, you shouldn't be focusing on implementing tons of new functionality just to write them—you should be focusing on the code under test. Assertion helpers and open source libraries make that easier.

To test the component at hand, you want to make a few assertions we mused about earlier: class name, inner content, and element type. You'll also create a snapshot test using the React test renderer. *Snapshot testing* is a feature of Jest that allows you to test the render output of your components in a unique way. Snapshot testing is closely related to *visual regression testing*, a process where the visual output of an application can be compared and checked for differences.

If a difference in images is found, you know that your test failed and needs adjusting or at least that the output snapshot needs to be updated. Rather than images, Jest will create JSON outputs for tests and store them in specially named directories. These should be added to version control along with all your other code. The following listing shows how to use Jest, Enzyme, and the React test renderer to make those assertions.

Listing 9.5 Making assertions (src/components/post/Content.test.js)

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import Content from '../../../src/components/post/Content';

describe('<Content/>', () => {
    test('should render correctly', () => {
        const mockPost = {
            content: 'I am learning to test React components'
        };
        const wrapper = shallow(<Content post={mockPost} />);
        expect(wrapper.find('p').length).toBe(1);
        expect(wrapper.find('p.content').length).toBe(1);
        expect(wrapper.find('.content').text()).toBe(mockPost.content);
        expect(wrapper.find('p').text()).toBe(mockPost.content);
    });
```

Import enzyme and react-test-renderer.

Import component you want to test

Use Jasmine-style describe function to group tests together

Create mock post

Use Enzyme's shallow method to render component

Create snapshot test using Jest and react-test-renderer

```
    test('snapshot', () => {
        const mockPost = {
            content: 'I am learning to test React components'
        };
        const component = renderer.create(<Content post={mockPost} />);
        const tree = component.toJSON();
        expect(tree).toMatchSnapshot();
    });
});
```

If your test runner is running, you should see a passing result from Jest. The Jest command-line tools have greatly improved since the test runner came out, and you should be able to see important information about your tests in the terminal.

9.3.3 Testing the CreatePost component without Enzyme

Now that you have your first test working, you can move on to testing more complex components. For the most part, testing React components should be straightforward. If you find yourself creating a component that has tons of functionality built into it and subsequently huge tests associated with it, you may want to consider breaking it into several components (although that's not always possible).

The next component you want to test, the CreatePost component, has more functionality than the Content component did, and your tests will need to address this added functionality. Listing 9.6 shows the CreatePost component so you can review it before writing out tests for it. The CreatePost component is used by the Home component to trigger the submission of new posts. It renders out a `textarea` that gets updated when the user types in it and a button that submits the form with data when a user clicks it. When the user clicks, it invokes a callback function passed by a parent component. You can test all these assumptions and make sure that things work as you expect.

Listing 9.6 CreatePost component (src/components/post/Create.js)

```
import PropTypes from 'prop-types';
import React from 'react';
import Filter from 'bad-words';
import classnames from 'classnames';
import DisplayMap from '../map/DisplayMap';
import LocationTypeAhead from '../map/LocationTypeAhead';
class CreatePost extends React.Component {
    static propTypes = {
        onSubmit: PropTypes.func.isRequired
    };
    constructor(props) {
        super(props);
        this.initialState = {
            content: '',
            valid: false,
            showLocationPicker: false,
```

```
            location: {
                lat: 34.1535641,
                lng: -118.1428115,
                name: null
            },
            locationSelected: false
        };
        this.state = this.initialState;
        this.filter = new Filter();
        this.handlePostChange = this.handlePostChange.bind(this);
        this.handleRemoveLocation = this.handleRemoveLocation.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
        this.handleToggleLocation = this.handleToggleLocation.bind(this);
        this.onLocationSelect = this.onLocationSelect.bind(this);
        this.onLocationUpdate = this.onLocationUpdate.bind(this);
        this.renderLocationControls = this.renderLocationControls.bind(this);
    }
    handlePostChange(event) {
        const content = this.filter.clean(event.target.value);
        this.setState(() => {
            return {
                content,
                valid: content.length <= 300
            };
        });
    }
    handleRemoveLocation() {
        this.setState(() => ({
            locationSelected: false,
            location: this.initialState.location
        }));
    }
    handleSubmit(event) {
        event.preventDefault();
        if (!this.state.valid) {
            return;
        }
        const newPost = {
            content: this.state.content
        };
        if (this.state.locationSelected) {
            newPost.location = this.state.location;
        }
        this.props.onSubmit(newPost);
        this.setState(() => ({
            content: '',
            valid: false,
            showLocationPicker: false,
            location: this.defaultLocation,
            locationSelected: false
        }));
    }
    onLocationUpdate(location) {
        this.setState(() => ({ location }));
    }
```

```
onLocationSelect(location) {
    this.setState(() => ({
        location,
        showLocationPicker: false,
        locationSelected: true
    }));
}
handleToggleLocation(event) {
    event.preventDefault();
    this.setState(state => ({ showLocationPicker:
 !state.showLocationPicker }));
}
renderLocationControls() {
    return (
        <div className="controls">
            <button onClick={this.handleSubmit}>Post</button>
            {this.state.location && this.state.locationSelected ? (
                <button onClick={this.handleRemoveLocation}
 className="open location-indicator">
                    <i className="fa-location-arrow fa" />
                    <small>{this.state.location.name}</small>
                </button>
            ) : (
                <button onClick={this.handleToggleLocation}
 className="open">
                    {this.state.showLocationPicker ? 'Cancel' : 'Add
 location'}{' '}
                    <i
                        className={classnames(`fa`, {
                            'fa-map-o': !this.state.showLocationPicker,
                            'fa-times': this.state.showLocationPicker
                        })}
                    />
                </button>
            )}
        </div>
    );
}
render() {
    return (
        <div className="create-post">
            <textarea
                value={this.state.content}
                onChange={this.handlePostChange}
                placeholder="What's on your mind?"
            />
            {this.renderLocationControls()}
            <div
                className="location-picker"
                style={{ display: this.state.showLocationPicker ? 'block'
 : 'none' }}
            >
                {!this.state.locationSelected && (
                    <LocationTypeAhead
                        onLocationSelect={this.onLocationSelect}
```

```
                            onLocationUpdate={this.onLocationUpdate}
                        />
                    )}
                    <DisplayMap
                        displayOnly={false}
                        location={this.state.location}
                        onLocationSelect={this.onLocationSelect}
                        onLocationUpdate={this.onLocationUpdate}
                    />
                </div>
            </div>
        );
    }
}

export default CreatePost;
```

This was a slightly more complicated component than you created in previous chapters. With it you can create posts and add a location to those posts. In my experience, testing larger and more complex components further highlights the importance of clean, readable tests. If you can't read or reason through your test file, how is a future-you or another developer going to?

Listing 9.7 shows a suggested skeleton of tests for the CreatePost component. You don't have enough methods to make it difficult to read through the tests, but if a component had more to it, you might even add nested `describe` blocks to make it easier to reason about. The functions in listing 9.7 will be executed by the test runner (Jest in this case), and within those tests you can make your assertions. Most tests follow this same sort of pattern. You import the code under test, mock out any dependencies to isolate your tests to one unit of functionality (hence *unit tests*), and then a test runner and assertion library will work together to run your tests.

Listing 9.7 Testing the CreatePost component (src/components/post/Create.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';

import CreatePost from '../../../src/components/post/Create';

describe('CreatePost', () => {
    test('snapshot', () => {

    });
    test('handlePostChange', () => {

    });
    test('handleRemoveLocation', () => {

    });
    test('handleSubmit', () => {
```

Using one describe call here, but in larger test files you can have many and even nest them

Create a test for each method in your component, including a snapshot to ensure it renders correctly

```
    });
    test('onLocationUpdate', () => {

    });
    test('handleToggleLocation', () => {

    });
    test('onLocationSelect', () => {

    });
    test('renderLocationControls', () => {

    });
});
```

If you follow a consistent pattern of considering each part of your component that needs to be tested, you'll be more thorough in developing and testing your components. Feel free to follow whatever structure makes the most sense to you—this is just one that has been helpful for me and for teams I've been on. I've also found it helpful to start writing tests by writing out the different `describe` and `test` blocks for a component or module before writing any other tests. I find that I can more easily think through the cases I want to cover (with an error, without an error, with a condition, and so on) if I'm doing that all at once.

What about other types of testing?

You may be wondering about testing such things as user flows, cross-browser testing, and other types of testing I'm not covering here. These other sorts of testing will generally be focused on by an engineer or engineering team dedicated to specialized forms of testing. QA teams and SETs (*software engineers in test*) will generally have a host of specialized tools that allow them to take your application and simulate all the complicated flows that might exist.

These types of testing (*integration testing*) may involve the interaction of one or more disparate systems. If you remember the testing pyramid from figure 9.1, these tests can take a lot of time to write, are hard to maintain, and tend to cost a lot of money. When you think of "testing front-end applications," you may think these sorts of tests are what would be involved. We've seen that this isn't the case (most tests that non-QA engineers write are unit or low-level integration tests). If you're interested in learning more about these sorts of tools, here are a few you could use as a springboard to learn more about higher-level testing:

- *Selenium*—www.seleniumhq.org
- *Puppeteer*—https://github.com/GoogleChrome/puppeteer
- *Protractor*—www.protractortest.org/#/

With this skeleton setup in place, you can begin testing the CreatePost component, starting with the constructor. Remember, the constructor is where initial state gets set

up, class methods get bound, and other setup can occur. To test this part of the CreatePost component, we need to introduce another tool I mentioned earlier: Sinon. You need some test functions that you can give to your component for use that aren't dependent on other modules. With Jest you can create mock functions for your test that help keep your tests focused on the component itself and prevent you from tying all your code together. Remember how I said tests should break when you change your code? That's true, but changing one test also shouldn't break other tests. As with regular code, your tests should be decoupled and only care about the slice of code they're testing.

Jest's mock functions not only help us isolate our code, they help us make more assertions. You can make assertions about how your component used the `mock` function, whether it was called, what arguments it was called with, and more. The following listing shows setting up the snapshot test for your component and mocking some basic props your component needs using Jest.

Listing 9.8 Writing your first test (src/components/post/Create.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';

import CreatePost from '../../../src/components/post/Create';

describe('CreatePost', () => {
    test('snapshot', () => {
        const props = { onSubmit: jest.fn() };
        const component = renderer.create(<CreatePost {...props} />);
        const tree = component.toJSON();
        expect(tree).toMatchSnapshot();
    });
   //...
});
```

Use jest.mock function to tell Jest to use a mock instead of the module when running tests

Create test block within outer describe block you created earlier

Use React test renderer to create your component and pass in props

Call toJSON method to generate a snapshot

Assert that snapshot matches

Create mock props object and use Jest's to create mock function

Now that you have one test under your belt, you can test some other aspects of the component. The component is primarily responsible for allowing users to create posts and attach locations to them, so you need to test those areas of functionality. You'll start by testing post creation. The next listing shows how to test post creator methods in your component.

Listing 9.9 Testing post creation (src/components/post/Create.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';

import CreatePost from '../../../src/components/post/Create';
```

```
describe('CreatePost', () => {
    test('snapshot', () => {
        const props = { onSubmit: jest.fn() };
        const component = renderer.create(<CreatePost {...props} />);
        const tree = component.toJSON();
        expect(tree).toMatchSnapshot();
    });
    test('handlePostChange', () => {
        const props = { onSubmit: jest.fn() };
        const mockEvent = { target: { value: 'value' } };
        CreatePost.prototype.setState = jest.fn(function(updater) {
            this.state = Object.assign(this.state, updater(this.state));
        });

        const component = new CreatePost(props);
        component.handlePostChange(mockEvent);
        expect(component.setState).toHaveBeenCalled();
        expect(component.setState.mock.calls.length).toEqual(1);
        expect(component.state).toEqual({
            valid: true,
            content: mockEvent.target.value,
            location: {
                lat: 34.1535641,
                lng: -118.1428115,
                name: null
            },
            locationSelected: false,
            showLocationPicker: false
        });

    });
    test('handleSubmit', () => {
        const props = { onSubmit: jest.fn() };
        const mockEvent = {
            target: { value: 'value' },
            preventDefault: jest.fn()
        };
        CreatePost.prototype.setState = jest.fn(function(updater) {
            this.state = Object.assign(this.state, updater(this.state));
        });

        const component = new CreatePost(props);
        component.setState(() => ({
            valid: true,
            content: 'cool stuff!'
        }));
        component.state = {
            valid: true,
            content: 'content',
            location: 'place',
            locationSelected: true
        };
        component.handleSubmit(mockEvent);
        expect(component.setState).toHaveBeenCalled();
        expect(props.onSubmit).toHaveBeenCalledWith({
```

Create mock set of props to use

Mock setState so you can make sure your component calls it and that updating post updates state in the right way.

Directly instantiate component and call its methods

Assert that your component invokes the right methods and that method updated state correctly

Create another mock event to simulate what your component will receive from an event

Mock setState again.

Instantiate another component and set state of component to simulate user entering post content

Directly modify component's state (for testing purposes)

Handle post submission with mock event you created and assert that mocks were called

```
            content: 'content',
            location: 'place'
        });
    });
});
```

Finally, you want to test the remainder of the component's functionality. Aside from letting users create posts, the CreatePost component also handles the user picking a location. Other components handle updating the location via callbacks passed as props, but you still need to test the component methods on CreatePost related to this feature.

Remember you implemented a subrender method on CreatePost, which you used to make reading the `render` method's output of CreatePost easier and to reduce clutter. You can test this in a similar way that you've been testing components with Enzyme or the React test renderer. The following listing shows the rest of the tests for the CreatePost component.

Listing 9.10 Testing post creation (src/components/post/Create.test.js)

```
jest.mock('mapbox');
import React from 'react';
import renderer from 'react-test-renderer';

import CreatePost from '../../../src/components/post/Create';

describe('CreatePost', () => {
    test('handleRemoveLocation', () => {
        const props = { onSubmit: jest.fn() };
        CreatePost.prototype.setState = jest.fn(function(updater) {
            this.state = Object.assign(this.state, updater(this.state));
        });
        const component = new CreatePost(props);
        component.handleRemoveLocation();
        expect(component.state.locationSelected).toEqual(false);
    });
    test('onLocationUpdate', () => {
        const props = { onSubmit: jest.fn() };
        CreatePost.prototype.setState = jest.fn(function(updater) {
            this.state = Object.assign(this.state, updater(this.state));
        });
        const component = new CreatePost(props);
        component.onLocationUpdate({
            lat: 1,
            lng: 2,
            name: 'name'
        });
        expect(component.setState).toHaveBeenCalled();
        expect(component.state.location).toEqual({
            lat: 1,
            lng: 2,
            name: 'name'
        });
    });
```

Mock setState

Invoke handleRemoveLocation function

Assert that you updated state in correct manner

Repeat same process for rest of your component methods

```
    test('handleToggleLocation', () => {
        const props = { onSubmit: jest.fn() };
        const mockEvent = {
            preventDefault: jest.fn()
        };
        CreatePost.prototype.setState = jest.fn(function(updater) {
            this.state = Object.assign(this.state, updater(this.state));
        });
        const component = new CreatePost(props);
        component.handleToggleLocation(mockEvent);
        expect(mockEvent.preventDefault).toHaveBeenCalled();
        expect(component.state.showLocationPicker).toEqual(true);
    });
    test('onLocationSelect', () => {
        const props = { onSubmit: jest.fn() };
        CreatePost.prototype.setState = jest.fn(function(updater) {
            this.state = Object.assign(this.state, updater(this.state));
        });
        const component = new CreatePost(props);
        component.onLocationSelect({
            lat: 1,
            lng: 2,
            name: 'name'
        });
    test('onLocationSelect', () => {
        const props = { onSubmit: jest.fn() };
        CreatePost.prototype.setState = jest.fn(function(updater) {
            this.state = Object.assign(this.state, updater(this.state));
        });
        const component = new CreatePost(props);
        component.onLocationSelect({
            lat: 1,
            lng: 2,
            name: 'name'
        });
        expect(component.setState).toHaveBeenCalled();
        expect(component.state.location).toEqual({
            lat: 1,
            lng: 2,
            name: 'name'
        });
    });

    test('renderLocationControls', () => {
        const props = { onSubmit: jest.fn() };
        const component = renderer.create(<CreatePost {...props} />);
        let tree = component.toJSON();
        expect(tree).toMatchSnapshot();
    });
});
```

Repeat same process for rest of your component methods

Create another snapshot test for subrender method you created

9.3.4 Test coverage

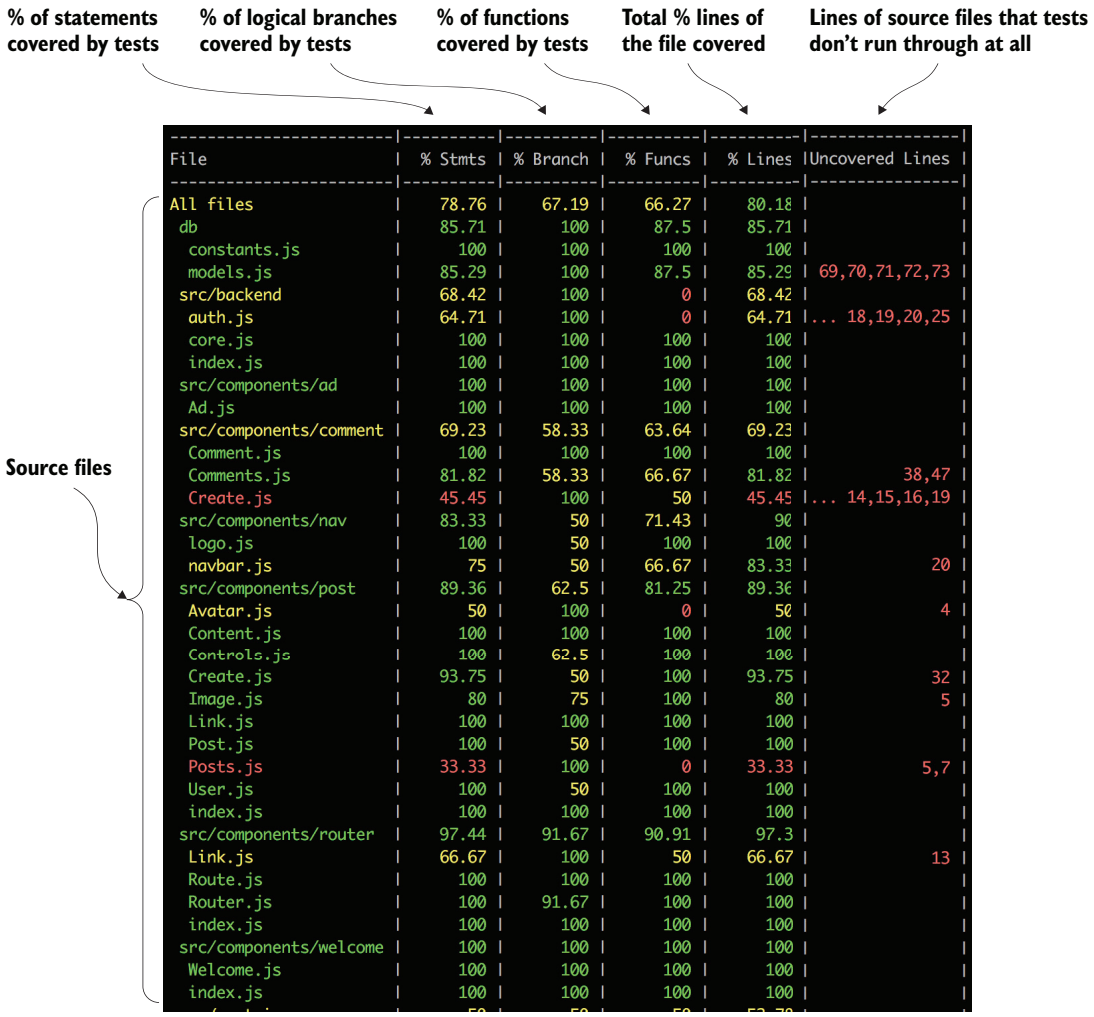
Now that you've gotten your hands dirty testing some components, let's look at test coverage and see what progress you've made. In your terminal, stop the test runner

and execute the command shown in the next listing. This command will turn on the coverage option included in Jest.

Listing 9.11 Enabling test coverage (project root)

```
> npm run test:w
```

Once your test runner finishes executing tests, it should output a colored table that should look something like figure 9.3 (with less coverage). The figure shows the Jest

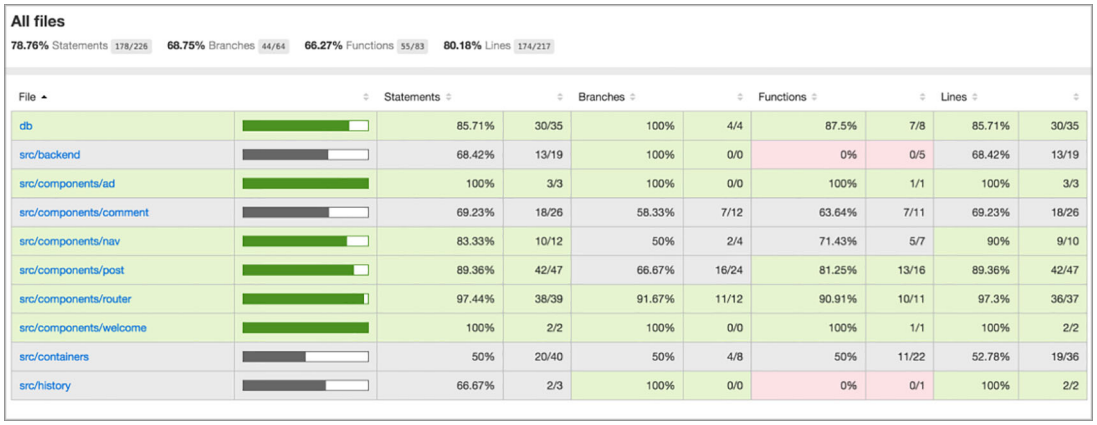


File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	78.76	67.19	66.27	80.18	
db	85.71	100	87.5	85.71	
constants.js	100	100	100	100	
models.js	85.29	100	87.5	85.29	69,70,71,72,73
src/backend	68.42	100	0	68.42	
auth.js	64.71	100	0	64.71	... 18,19,20,25
core.js	100	100	100	100	
index.js	100	100	100	100	
src/components/ad	100	100	100	100	
Ad.js	100	100	100	100	
src/components/comment	69.23	58.33	63.64	69.23	
Comment.js	100	100	100	100	
Comments.js	81.82	58.33	66.67	81.82	38,47
Create.js	45.45	100	50	45.45	... 14,15,16,19
src/components/nav	83.33	50	71.43	90	
logo.js	100	50	100	100	
navbar.js	75	50	66.67	83.33	20
src/components/post	89.36	62.5	81.25	89.36	
Avatar.js	50	100	0	50	4
Content.js	100	100	100	100	
Controls.js	100	62.5	100	100	
Create.js	93.75	50	100	93.75	32
Image.js	80	75	100	80	5
Link.js	100	100	100	100	
Post.js	100	50	100	100	
Posts.js	33.33	100	0	33.33	5,7
User.js	100	50	100	100	
index.js	100	100	100	100	
src/components/router	97.44	91.67	90.91	97.3	
Link.js	66.67	100	50	66.67	13
Route.js	100	100	100	100	
Router.js	100	91.67	100	100	
index.js	100	100	100	100	
src/components/welcome	100	100	100	100	
Welcome.js	100	100	100	100	
index.js	100	100	100	100	

Figure 9.3 Test coverage output from Jest shows coverage stats for the different files in your project. Each column reflects a different aspect of coverage. For each type of coverage, Jest shows a percentage covered. Statements and functions are simply JavaScript statements and functions, whereas branches are logical branches. If your test doesn't address one part of an if statement, that should be reflected in the code coverage both in the uncovered lines column and in the percent-covered stat for branches.

coverage output with annotations about each of the columns. There are different forms of readable code coverage reports (HTML, for example), but the terminal output is most useful during development because it provides immediate feedback.

Istanbul is the tool generating the stats in figure 9.3. If you want to see more detailed coverage information, open the coverage directory that should have been generated by the `jest` command that included the coverage option. In this directory, Istanbul should have created a few files. If you open ./coverage/lcov-report/index.html in a browser, you should see something like figure 9.4.



All files

78.76% Statements 178/226 **68.75%** Branches 44/64 **66.27%** Functions 55/83 **80.18%** Lines 174/217

File	Statements		Branches		Functions		Lines	
db	85.71%	30/35	100%	4/4	87.5%	7/8	85.71%	30/35
src/backend	68.42%	13/19	100%	0/0	0%	0/5	68.42%	13/19
src/components/ad	100%	3/3	100%	0/0	100%	1/1	100%	3/3
src/components/comment	69.23%	18/26	58.33%	7/12	63.64%	7/11	69.23%	18/26
src/components/nav	83.33%	10/12	50%	2/4	71.43%	5/7	90%	9/10
src/components/post	89.36%	42/47	66.67%	16/24	81.25%	13/16	89.36%	42/47
src/components/router	97.44%	38/39	91.67%	11/12	90.91%	10/11	97.3%	36/37
src/components/welcome	100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/containers	50%	20/40	50%	4/8	50%	11/22	52.78%	19/36
src/history	66.67%	2/3	100%	0/0	0%	0/1	100%	2/2

Figure 9.4 Istanbul generates coverage metadata in computer-readable and human-readable formats. The coverage report shown here is useful for more detailed exploration of code coverage. You can even sort by different columns and prioritize files with low coverage. Note that there are columns for statements, branches (if/else statements), functions (which functions were called), and lines (lines of code).

The Istanbul output is useful, but you can also drill down into different files and get more in-depth information about individual files. Each file should display information about how many times different lines were covered and which ones weren't. Most of the time the top-level summary is good enough, but sometimes you may want to inspect individual reports, like the one in figure 9.5. When I'm writing tests, I like to take at least one look at these files once I've covered all my cases to make sure I didn't miss any edge cases or logical branches.

Test coverage is an important and useful tool for software development, but don't treat it as a magical guarantee that your code works. You can get to 100% coverage and still have code that breaks. You can technically also have code that works with 0% code coverage. *Coverage* is about making sure your tests are executing all the different parts of your code—not guaranteeing a lack of errors or things like performance—but it's useful for that and should be treated as an important data point when considering how "complete" your code is. I've been on teams where our definition of success for a particular user story or task included, among other things, code coverage above

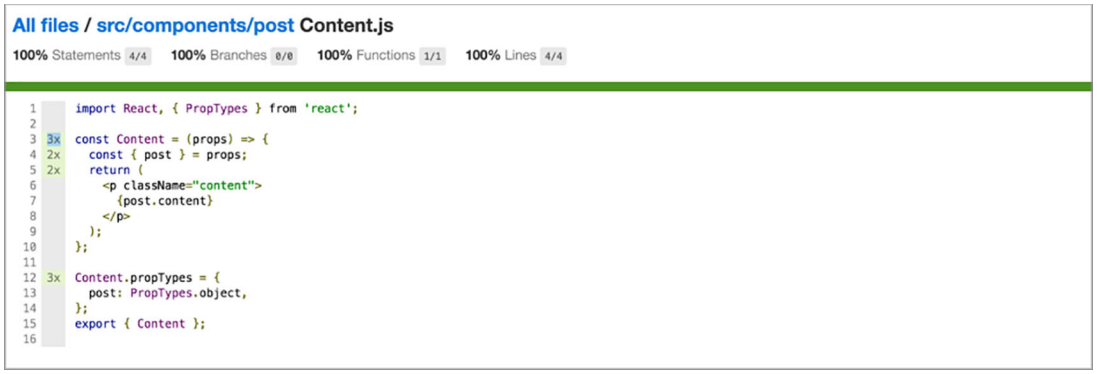


Figure 9.5 Individual file coverage report generated by Istanbul. You can see how many times different lines were or weren't covered and get a sense for exactly which parts of your code were covered.

80% and no decreased coverage overall. Use coverage as a guideline for which parts of your code you have or haven't tested and to check your testing progress.

> **Exercise 9.2 Considering coverage**
> We talked about test coverage in this chapter. Does 100% test coverage mean that your code is perfect? What role should code coverage play in your testing?

9.4 Summary

In this chapter, you learned about some of the principles behind testing and how to test React applications:

- *Testing* is the process of validating assumptions made about software. It helps you better plan your components, prevents breakage in the future, and helps increase confidence in your code. It also plays an important role in a rapid development process.
- Manual testing doesn't scale well because no number of people could ever quickly or adequately test complex software well.
- We use a variety of tools in the software testing process, ranging from tools that run our tests to tools that determine how much of our code is covered by tests.
- Different types of tests should occur in different proportions. *Unit* tests should be the most common and are easy, cheap, and quick to write. *Integration* tests test the interaction of many different parts of the system and can be brittle and take longer to write. They should be less common.
- You can test React components using a variety of tools. Because they're just functions, you could test them strictly as such. But tools like Enzyme make testing React components easier.

- Clean tests, like any clean code, are easy to read and well organized and use appropriate proportions of unit, service, and integration tests. They should provide meaningful assurance that things function in a particular manner and should guarantee that changes to your component can be evaluated.

In the next chapter, we'll look at a more robust implementation of the Letters Social app and explore the Redux architectural pattern. Before moving on, see if you can keep honing your testing skills and get test coverage for the app up above 90%!

JAVASCRIPT/WEB DEVELOPMENT

React IN ACTION

Mark Tielens Thomas

Facebook created React to help deliver amazing user experiences on a website with thousands of components and an incomprehensible amount of traffic. The same powerful tools are available to you too! The key is a clever design for managing state, data flow, and rendering, so your application is easy to think about and runs smoothly. Add an incredibly rich ecosystem of components and libraries, and you've got a recipe for building web apps that will delight both developers and users.

React in Action teaches you to think like a pro about user interfaces and building them with React. This practical book gets you up and running quickly with hands-on examples in every chapter. You'll master core topics like rendering, lifecycle methods, JSX, data flow, forms, routing, integrating with third-party libraries, and testing. And the included application design ideas will help make your apps pop. As you learn to integrate React into full-stack applications, you'll explore state management with Redux and server-side rendering, and even dabble in React Native for mobile UIs.

What's Inside

- React from the ground up
- Implementing a routing system with components
- Server-side rendering in Node.js
- Working with third-party libraries
- Testing React components

Written for developers familiar with HTML, CSS, and JavaScript.

Mark Thomas is an experienced software engineer who works daily with React, JavaScript, and Node.js. He loves clean code, beautiful systems, and good coffee.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/react-in-action

"Read this. Work with React. Never look back."
—Michal Paszkiewicz
Transport for London

"One stop—for concepts as well as for real-world examples and integrations."
—Phaneendra Bommareddy
Openlogix

"A must-have for anyone wanting to create applications using React and Redux!"
—Andrew Courter, Pivotal

"Easy to follow, clearly demonstrates all necessary steps, includes plenty of code examples, and never leaves you in the dark."
—Olivier Ducatteeuw
University of Leuven

ISBN-13: 978-1-61729-385-6
ISBN-10: 1-61729-385-7
54499
9 781617 293856

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]