Watch this chapter's introduction video by scanning this QR code with your phone or going to http://reactquickly.co/videos/ch16.

16 Unit testing React with Jest

This chapter covers

- Reasons to use Jest
- Unit testing with Jest
- UI testing with Jest and TestUtils

In modern software engineering, testing is important. It's at least as important as using Agile methods, writing well-documented code, and having enough coffee on hand—sometimes even more so. Proper testing will save you from many hours of debugging later. The code isn't an asset, it's a liability, so your goal is to make it as easy to maintain as possible.

Code is a liability?

Googling the phrase "Code isn't an asset, it's a liability" gives 191 million results, which makes it hard to pinpoint its origins. Although I can't find an author, I can tell you the gist of the idea: when you write software, you're building apps/products/services that are assets, but your code is *not* one of them.

(continued)

Assets are things that generate income. Code does not generate any income by itself. Yes, code enables products, but the code is a tool to make the products (which are assets). The code itself isn't an asset—it's more of a necessary evil to get to the end goal of having a working application.

Thus, code is a liability, because you have to maintain it. More code does *not* automatically translate into more revenue or better product quality; but more code almost always increases complexity and the cost of maintenance. Some of the best ways to minimize the cost of maintaining code are to make it simple, robust, and flexible for future changes and enhancements. And testing—especially automated testing—helps when you're making changes, because you have more assurance that the changes didn't break your app.

Using test-driven/behavior-driven development (TDD/BDD) can make maintenance easier. It can also make your company more competitive by letting you iterate more quickly and make you more productive by giving you the confidence that your code works.

NOTE The source code for the examples in this chapter is at www.manning.com/books/react-quickly and https://github.com/azat-co/react-quickly/tree/master/ch16. You can also find some demos at http://reactquickly.co/demos.

16.1 Types of testing

There are multiple types of testing. Most commonly, they can be separated into three categories: unit, service, and UI testing, as shown in figure 16.1. Here's an overview of each category, from lowest to highest level:

- *Unit testing*—The system tests standalone methods and classes. There are no or few dependencies or interconnected parts. The code for the tested subject should be enough to verify that the method works as it should work. For example, a module that generates random passwords can be tested by invoking a method from a module and comparing the output against a regular-expression pattern. This category also includes tests that may involve a few parts or modules working together to produce one piece of functionality. For example, several components have to work together to provide the functionality for password input with a strength check. They can be tested by supplying the value to one component (input) and monitoring changes in the strength check (sufficient or not). These tests are durable; according to industry best practices, this category should make up roughly 70% of your tests (see figure 16.1) and should definitely outnumber any other types of tests.
- *Service (integration) testing*—Tests typically involve other dependencies and require a separate environment. Integration tests should be roughly 20% of all

your tests. Once you have a solid foundation of unit tests and the assurance of functional tests, you don't want to have too many integration tests, because maintaining them will slow development. Each time there's a UI change, your integration tests need to be updated. This often leads to flaky UI tests and no integration testing at all, which is even worse.

- *UI (acceptance)*—Tests often mimic Agile user stories and/or involve testing the entire system, which obviously has all the dependencies and complexities imaginable. UI tests are more fragile and difficult (expensive) to maintain, and thus they should be only about 10% of your overall tests.

This chapter covers unit testing of React apps with a bit of UI testing of React components, using the mock DOM rendering of React and Jest. You'll also use the standard toolchain of Node, npm, Babel, and Webpack. To begin unit testing, let's investigate Jest.

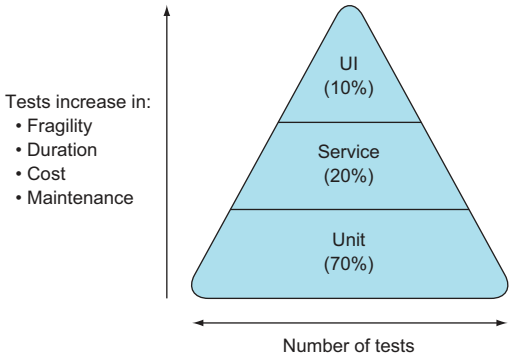

Figure 16.1 Testing pyramid according to software engineering's best practices

16.2 Why Jest (vs. Mocha or others)?

Jest (https://facebook.github.io/jest) is a command-line tool based on Jasmine. It has a Jasmine-like interface. If you've worked with Mocha, you'll find that Jest looks similar to it and is easy to learn. Jest is developed by Facebook and is often used together with React; the API documentation is at https://facebook.github.io/jest/docs/api.html#content.

Jest offers these features:

- Powerful mocking (https://facebook.github.io/jest/docs/mock-functions.html) of JavaScript/Node modules makes it easier to isolate code in order to unit test it.
- Less setup is required to get started than with other test runners, such as Mocha, where you need to import Chai or standalone Expect. Jest also finds tests in the __tests__ folder.
- Tests can be sandboxed and executed in parallel to run them more quickly.[1]
- You can perform static analysis with the support of Facebook's Flow (https://flowtype.org), which is a static type checker for JS.
- Jest provides modularity, configurability, and adaptability (via the support of Jasmine assertions).

[1] Christopher Poher, "JavaScript Unit Testing Performance," *Jest*, March 11, 2016, http://mng.bz/YfXz.

Mocking, static analysis, and Jasmine

The term *mocking* means faking a certain part of a dependency so you can test the current code. *Automocking* means mocking is done for you automatically. In Jest before v15,[2] every imported dependency is automocked, which can be useful if you frequently rely on mocking. Most developers don't need automocking, so in Jest v15+ it's off by default—but automocking can be turned on if necessary.

Static analysis means the code can be analyzed before you run it, which typically involves type checking. Flow is a library that adds type checking to otherwise typeless (more or less) JavaScript.

Jasmine is a feature-rich testing framework that comes with an assertion language. Jest extends and builds on Jasmine under the hood so you don't need to import or configure anything. Thus, you have the best of both worlds: you can tap into the common interface of Jasmine without needing extra dependencies or setup.

There are many opinions about what test framework is better for what job. Most projects use Mocha, which has a lot of features. Jasmine arose from front-end development but is interchangeable with Mocha and Jest. All of them use the same constructs to define test suites and tests:

- `describe`—Test suite
- `it`—Test case
- `before`—Preparation
- `beforeEach`—Preparation for every suite or case
- `after`—Cleanup
- `afterEach`—Cleanup for every suite or case

Without getting into a heated debate in this book about what framework is the best, I encourage you to keep an open mind and explore Jest because of the features I've listed and because it comes from the same community that develops React. This way, you can make a better judgment about which framework to use for your next React project.

Most modern frameworks like Mocha, Jasmine, and Jest are similar for most tasks. Any difference will depend on your preferred style (maybe you like automocking, or maybe you don't) and on the edge cases of your particular project (do you need all the features Mocha provides, or you need something lightweight like the Test Anything Protocol's [TAP, https://testanything.org] `node-tap` [www.node-tap.org]?). Jest is a good place to start, because once you learn how to use Jest with React utilities and methods, you can use other test runners and testing frameworks such as Mocha, Jasmine, and `node-tap`.

[2] See Christoph Pojer, "Jest 15.0: New Defaults for Jest," September 1, 2016, http://mng.bz/p20n.

16.3 Unit testing with Jest

If you've never worked with any of the testing frameworks I've been discussing, don't worry; Jest is straightforward to learn. The main statement is `describe`, which is a test suite that acts as a wrapper for tests; and `it`, which is an individual test called a *test case*. Test cases are nested within the test suite.

Other constructs such as `before`, `after`, and their `Each` brethren `beforeEach` and `afterEach` execute either before or after the test suite or test case. Adding `Each` executes a piece of code many times as compared to just one time.

Writing tests consists of creating test suites, cases, and assertions. *Assertions* are like true or false questions, but in a nice readable format (BDD).

Here's an example, without assertions for now:

Defines the done() callback

```
describe('Noun: method or a class/module name', () => {
  before((done) => {
    // This code will be called just once before all it statements
    done()
  })
  beforeEach((done) => {
    // This code will be called many times before all it statements
    done()
  })
  it('Verb describing the behavior', (done) => {
       // Assertions
        done()
  })
  it('Verb describing the behavior', (done) => {
       // Assertions
    done()
  })
  ...
  after((done) => {
    // This code will be called just once after all it statements
    done()
  })
  afterEach((done) => {
    // This code will be called many times after all it statements
    done()
  })
})
```

Invokes done() when the async test code is finished

You must have at least one `describe` and one `it`, but their number isn't limited. Everything else, such as `before` and `after`, is optional.

You won't be testing any React components yet. Before you can work with React components, you need to learn a little more about Jest by working on a Jest example that doesn't have a UI.

In this section, you'll create and unit-test a module that generates random passwords. Imagine you're working on a sign-up page for a cool new chat app. You need

the ability to generate passwords, right? This module will automatically generate random passwords. To keep things simple, the format will be eight alphanumeric characters. The project (module) structure is as follows:

```
/generate-password
  /__test__
    generate-password.test.js
  /node_modules
    generate-password.js
    package.json
```

You'll use the CommonJS/Node module syntax, which is widely supported in Node (duh) and also in browser development via Browserify and Webpack. Here's the module in the ch16/generate-password.js file.

Listing 16.1 Module for generating passwords

```
module.exports = () => {
  return Math.random().toString(36).slice(-8)
}
```

Uses slice with a negative number to reverse the order (right to left)

Just as a refresher, in this file you export the function via the `module.exports` global. This is Node.js and CommonJS notation. You can use it on the browser with extra tools like Webpack and Browserify (http://browserify.org).

The function uses `Math.random()` to generate a number and convert it to a string. The string length is eight characters, as specified by `slice(-8)`.

To test the module, you can run this eval Node command from the terminal. It imports the module, invokes its function, and prints the result:

```
node -e \"console.log(require('./generate-password.js')())\"
```

You could improve this module by making it work with different numbers of characters, not just eight.

16.3.1 Writing unit tests in Jest

To begin using Jest, you need to create a new project folder and `npm init` it to create package.json. If you don't have npm, this is the best time to install it; follow the instructions in appendix B.

Once you've created the package.json file in a new folder, install Jest:

```
$ npm install jest-cli@19.0.2 --save-dev --save-exact
```

I'm using `jest-cli` version 19.0.2; make sure your version is the same or compatible. `--save-dev` adds the entry to the package.json file. Open the file, and manually change the `test` entry to `jest` as shown next (ch16/jest/package.json). This will add the testing command. Also add the start script.

Listing 16.2 Saving a test CLI command

```
{
  "name": "jest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest",
    "start": "node -e
    ➥ \"console.log(require('./generate-password.js')())\""
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "devDependencies": {
    "jest-cli": "19.0.2"
  }
}
```

Replaces the default test script with jest

Saves the Node eval command to get a random password

Uses 19.0.2 without ^ to ensure the exact version of 19.0.2

Now, create a folder named __tests__. The name is important because Jest will pick up tests from that folder. Then, create your first Jest test in __tests__/generate-password.js.

Typically, you only mock dependencies that you don't need to isolate the library you're currently unit testing. Jest prior to v15 automatically mocks every required file, so you need to use `dontMock()` or `jest.autoMockOff()` to avoid this for the main module you test (generate-password.js). This is one way to do it:

```
jest.dontMock('../generate-password.js')
```

Only for Jest prior to v15

Luckily, for the version of Jest used in this chapter (v19), you *don't need to disable automock*, because it's disabled by default. So, you can skip this `dontMock()` line of code or leave it commented out.

The test file has a single suite (only one `describe`), which expects the value to match the `/^[a-z0-9]{8}$/` regular-expression pattern—only alphanumerics and exactly eight characters—to satisfy your condition for a strong password (ch16/generate-password/__tests__/generate-password.test.js). You don't want your chat users to be hacked by brute force!

Listing 16.3 Test file for the password-generating module

```
describe('method generatePassword', ()=>{
  let password
  generatePassword = require('../generate-password')
  it('returns a generated password of lower-case characters
  ➥ and numbers with the length of 8', (done)=>{
    password = generatePassword()
    expect(password).toMatch(/^[a-z0-9]{8}$/)
    done()
  })
})
```

Uses require, a special Node.js global that imports the module into your script.js file

Invokes done() if you defined an argument needed for asynchronous tests and optional for synchronous (in this case, it's sync)

You can run the test with `$ npm test`. You'll see something like this as the terminal output:

```
Using Jest CLI v13.2.3, jasmine2
 PASS  __tests__/generate-password.test.js (0.031s)
1 test passed (1 total in 1 test suite, run time 1.339s)
```

How many tests passed and how many you have in total

16.3.2 Jest assertions

By default, Jest uses BDD syntax (https://en.wikipedia.org/wiki/Behavior-driven_development) powered by Expect syntax (https://facebook.github.io/jest/docs/api.html). Expect is a popular language that's a replacement for TDD assertions. It has many flavors, and Jest uses a somewhat simplified version (in my opinion). Unlike other frameworks, such as Mocha, where you need to install additional modules for syntax support, in Jest it's automatic.

TDD and BDD

TDD can mean test-driven development or TDD syntax with assertions. Briefly, during test-driven development you write a test, then run it (failing), then make it work (passing), and then make it right (refactor).

You most certainly can perform test-driven development with BDD. The main benefit of BDD style is that it's intended for communicating with *every member of a cross-functional team*, not just software engineers. TDD is more of a techie language. BDD format makes it easier to read tests—ideally the spec title should tell you what you're testing, as in this example:

```
describe('method generatePassword', ()=>{
  ...
  it('returns a generated password of lower-case characters
  ➥ and numbers with the length of 8', ()=>{
    ...
    expect(password).toMatch(/^[a-z0-9]{8}$/)
  })
})
```

Uses a noun to describe the test suite

Uses verbs to describe the behavior for a test case

Uses an expect statement to implement a test case

Here's a list of the main Expect methods that Jest supports (there are many more). You pass the actual values—returned by the program—to `expect()` and use the following methods to compare those values with expected values that are hardcoded in the tests:

- `.not`—Inverses the next comparison in the chain
- `expect(OBJECT).toBe(value)`—Expects the value to be equal with JavaScript's triple equal sign `===` (checks for value and type, not just value)[3]

[3] See "Equality Comparisons and Sameness," Mozilla Developer Network, http://mng.bz/kliO.

- `expect(OBJECT).toEqual(value)`—Expects the value to be deep-equal[4]
- `expect(OBJECT).toBeFalsy()`—Expects the value to be falsy (see the following sidebar)
- `expect(OBJECT).toBeTruthy()`—Expects the value to be truthy
- `expect(OBJECT).toBeNull()`—Expects the value to be `null`
- `expect(OBJECT).toBeUndefined()`—Expects the value to be undefined
- `expect(OBJECT).toBeDefined()`—Expects the value to be defined
- `expect(OBJECT).toMatch(regexp)`—Expects the value to match the regular expression

Truthy and falsy

In JavaScript/Node, a *truthy* value translates to `true` when evaluated as a Boolean in an `if/else` statement. A *falsy* value, on the other hand, evaluates to `else` in an `if/else`.

The official definition is that a value is truthy if it's not falsy, and there are only six falsy values:

- `false`
- `0`
- `""` (empty string)
- `null`
- `undefined`
- `NaN` (not a number)

Everything not listed here is truthy.

To summarize, Jest can be used for unit tests, which should be the most numerous of your tests. They're lower level, and for this reason they're more solid and less brittle, which makes them less costly to maintain.

Thus far, you've created a module and tested its method with Jest. This is a typical unit test. There are no dependencies involved—only the tested module itself. This skill should prepare you to continue with testing React components. Next, let's look at more-complicated UI testing. The following section deals with the React testing utility, which enables you to perform UI testing.

16.4 UI testing React with Jest and TestUtils

Generally speaking, in UI testing (recommended to make up only 10% of your tests), you test entire components, their behavior, and even entire DOM trees. You can test components manually, which is a terrible idea! Humans make mistakes and take a long time to perform tests. Manual UI testing should be minimal or nonexistent.

[4] *Deep equality* compares objects, including all their properties and values, to the last level of nestedness (going deep). There's no standard API for it in JavaScript, but there are implementations like Node's core `assert` module (http://mng.bz/rhoX) and `deep-equal` (www.npmjs.com/package/deep-equal).

What about automated UI testing? You can test automatically using *headless browsers* (https://en.wikipedia.org/wiki/Headless_browser), which are like real browsers but without a GUI. That's how most Angular 1 apps are tested. It's possible to use this process with React, but it isn't easy, it's often slow, and it requires a lot of processing power.

Another automated UI testing approach uses React's virtual DOM, which is accessible via a browser-like testing JavaScript environment implemented by `jsdom` (https://github.com/tmpvar/jsdom). To use React's virtual DOM, you'll need a utility that's closely related to the React Core library but not part of it: TestUtils, which is a React utility to test its components. Simply put, TestUtils allows you to create components and render them into the fake DOM. Then you can poke around, looking at the elements by tags or classes. It's all done from the command line, without the need for browsers (headless or not).

> **NOTE** There are other React add-ons, listed at https://facebook.github.io/react/docs/addons.html. Most of them are no longer in development or are still in the experimental stage, which in practice means the React team may change their interface or stop supporting them. All of them follow the naming convention `react-addons-NAME`. TestUtils is an add-on, and, like other React add-ons, it's installed via npm. (You can't use TestUtils without npm; if you haven't already, you can get npm by following the instructions in appendix A.)

For versions of React before v15.5.4, TestUtils was in an npm package `react-addons-test-utils` (https://facebook.github.io/react/docs/test-utils.html). For example, if you're using React version 15.2.1, you can install `react-addons-test-utils` v15.2.1 with npm using the following command:

```
$ npm install react-addons-test-utils@15.2.1 --save-dev --save-exact
```

And this goes in your test source code (React prior to v15.5.4):

```
const TestUtils = require('react-addons-test-utils')
```

In React v15.5.4, things are somewhat easier, because TestUtils is in ReactDOM (`react-dom` on npm). You don't have to install a separate package for this example, because you're using the newer v15.5.4:

```
const TestUtils = require('react-dom/test-utils')
```

TestUtils has a few primary methods for rendering components; simulating events such as `click`, `mouseOver`, and so on; and finding elements in a rendered component. You'll begin by rendering a component and learn about other methods as you need them.

To illustrate the TestUtils `render()` method, the following listing renders an element into a `div` variable without using a headless (or real, for that matter) browser (ch16/testutils/__tests__/render-props.js).

Listing 16.4 Rendering a React element in Jest

```
describe('HelloWorld', ()=>{
  const TestUtils = require('react-dom/test-utils')
  const React = require('react')

  it('has props', (done)=>{

    class HelloWorld extends React.Component {
      render() {
        return <div>{this.props.children}</div>
      }
    }
    let hello = TestUtils.renderIntoDocument(<HelloWorld>Hello Node!
    ➥ </HelloWorld>)
    expect(hello.props).toBeDefined()
    console.log('my hello props:', hello.props) // my div: Hello Node!

    done()
  })
})
```

And package.json for ch16/testutils example looks like this with Babel, Jest CLI, React, and React DOM:

```
{
  "name": "password",
  "version": "2.0.0",
  "description": "",
  "main": "index.html",
  "scripts": {
    "test": "jest",
    "test-watch": "jest --watch",
    "build-watch": "./node_modules/.bin/webpack -w",
    "build": "./node_modules/.bin/webpack"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-jest": "19.0.0",
    "babel-preset-react": "6.24.1",
    "jest-cli": "19.0.2",
    "react": "15.5.4",
    "react-dom": "15.5.4"
  }
}
```

> **WARNING** `renderIntoDocument()` only works on custom components, not standard DOM components like `<p>`, `<div>`, `<section>`, and so on. So if you see an error like `Error: Invariant Violation: findAllInRenderedTree(...): instance must be a composite component`, make sure you're rendering a custom (your own) component class and not a standard class. See the commit at http://mng.bz/8AOc and the https://github.com/facebook/react/issues/4692 thread on GitHub for more information.

Once you have `hello`, which has the value of the React component tree (includes all child components), you can look inside it with one of the find-element methods. For example, you can get the `<div>` from within the `<HelloWorld/>` element, as shown next (ch16/testutils/__tests__/scry-div.js).

Listing 16.5 Finding a React element's child element `<div>`

```
describe('HelloWorld', ()=>{
  const TestUtils = require('react-dom/test-utils')
  const React = require('react')

  it('has a div', (done)=>{

    class HelloWorld extends React.Component {
      render() {
        return <div>{this.props.children}</div>
      }
    }
    let hello = TestUtils.renderIntoDocument(
      <HelloWorld>Hello Node!</HelloWorld>
    )
    expect(TestUtils.scryRenderedDOMComponentsWithTag(
      hello, 'div'
    ).length).toBe(1)
    console.log('found this many divs: ',
      TestUtils.scryRenderedDOMComponentsWithTag(hello, 'div').length)

    done()
  })
})
...
```

`scryRenderedDOMComponentsWithTag()` allows you to get an array of elements by their tag names (such as `div`). Are there any other ways to get elements? Yes!

16.4.1 Finding elements with TestUtils

In addition to `scryRenderedDOMComponentsWithTag()`, there are a few other ways to get either a list of elements (prefixed with `scry`, plural `Components`) or a single element (prefixed with `find`, singular `Component`). Both use an element class, not a component class, which is a different thing. For example, `btn`, `main`, and so on.

In addition to tag names, you can get elements by type (component class) or by their CSS classes. For example, `HelloWorld` is a type, whereas `div` is a tag name (you used it to pull the list of criteria).

You can mix and match `scry` and `find` with `Class`, `Type`, and `Tag` to get six methods, depending on your needs. Here's what each method returns:

- `scryRenderedDOMComponentsWithTag()`—Many elements; you know their tag name.
- `findRenderedDOMComponentWithTag()`—A single element; you know its unique tag name. That is, no other elements in the component have a similar tag name.
- `scryRenderedDOMComponentsWithClass()`—Many elements; you know their class name.
- `findRenderedDOMComponentWithClass()`—A single element; you know its unique class name.
- `scryRenderedComponentsWithType()`—Many elements; you know their type.
- `findRenderedComponentWithType()`—A single element; you know its type.

As you can see, there's no shortage of methods when it comes to pulling the necessary element(s) from your components. If you need some guidance, I suggest using classes or types (component classes), because they let you target elements more robustly. For instance, suppose you use tag names now because there's just one `<div>`. If you decide to add elements with the same tag names to your code (more than one `<div>`), you'll need to rewrite your test. If you use an HTML class to test a `<div>`, your test will work fine after you add more `<div>` element to the tested component.

The only case when using tag names might be appropriate is when you need to test all the elements with a specific tag name (`scryRenderedDOMComponentsWithTag()`) or your component is so small that there are no other elements with the same tag name (`findRenderedDOMComponentWithTag()`). For example, if you have a stateless component that wraps an anchor tag `<a>` and you add a few HTML classes to it, there will be no additional anchor tags.

16.4.2 UI-testing the password widget

Consider a UI widget that can be used on a sign-up page to automatically generate passwords of a certain strength. As shown in figure 16.2, it has an input field, a Generate button, and a list of criteria.

The following section walks through the entire project. For now, we're focusing on using TestUtils and its interface. Once TestUtils and other dependencies (such as Jest) are installed, you can create the Jest test file to UI-test your widget; let's call it password/__tests__/password.test.js, because you're testing a password component. The structure of this test is as follows:

```
describe('Password', function() {
  it('changes after clicking the Generate button', (done)=>{
    // Importations
    // Perform rendering
    // Perform assertions on content and behavior
    done()
  })
})
```

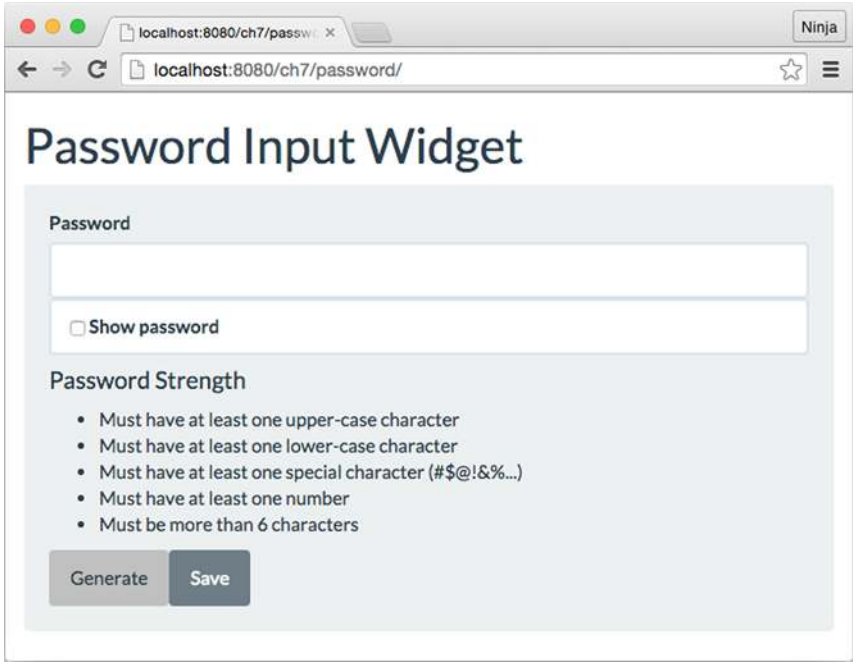

Figure 16.2 Password widget that autogenerates a password according to the given strength criteria

Let's define the dependencies in `describe`. Note that I've created the shortcut `fD` for `ReactDOM.findDOMNode()` because you'll use it a lot:

```
const TestUtils = require('react-dom/test-utils')
    const React = require('react')
    const ReactDOM = require('react-dom')
    const Password = require('../jsx/password.jsx')
    const fD = ReactDOM.findDOMNode
```

To render a component, you need to use `renderIntoDocument()`. For example, this is how you can render a `Password` component and save a reference to the object in the `password` variable. The properties you're passing will be the keys of the rules for the password strength. For example, `upperCase` requires at least one uppercase character:

```
let password = TestUtils.renderIntoDocument(<Password
      upperCase={true}
      lowerCase={true}
      special={true}
      number={true}
      over6={true}
      />
    )
```

This example is in JSX because Jest automatically uses `babel-jest` when you have installed this module (`npm i babel-jest --save-dev`) and sets the Babel configuration to use `"presets": ["react"]`. You cannot use JSX in Jest if you don't want to include `babel-jest`. In this case, call `createElement()`:

```
let password = TestUtils.renderIntoDocument(
    React.createElement(Password, {
      upperCase: true,
      lowerCase: true,
      special: true,
      number: true,
      over6: true
    })
  )
```

Once you've rendered the component with `renderIntoDocument()`, it's straightforward to extract the needed elements—children of `Password`—and execute assertions to see how your widget is working. Think of the extraction calls as your jQuery; you can use tags or classes. At the bare minimum, your test should check for these things:

1. There's a `Password` element with a list of items (`<li>`) that are the strength criteria.
2. The first item in the strength list has specific text.
3. The second item isn't fulfilled (strikethrough).
4. There's a Generate button (class `generate-btn`)—click it!
5. After you click Generate, the second list item become fulfilled (visible).

Clicking Generate fulfills all criteria and makes the password visible (so users can memorize it), but you won't see the test code for that feature in this book. That's your homework for next week.

Let's start with item 1. `TestUtils.scryRenderedDOMComponentsWithTag()` gets all elements from a particular class. In this case, the class is `li` for the `<li>` elements because that's what the criteria list will use: `<ul><li>`. `toBe()`, which works like the triple equal (`===`), can be used to validate the list length as 5:

```
let rules = TestUtils.scryRenderedDOMComponentsWithTag(password, 'li')
    expect(rules.length).toBe(5)
```

For item 2, which checks that the first list item has specific text, you use `toEqual()`. You expect the first item to say that an uppercase character is required. This will be one of the rules for password strength:

```
expect(fD(rules[0]).textContent).toEqual('Must have
  ➥ at least one uppercase character')
```

To check items 3, 4, and 5, you find a button, click it, and compare the values of the second criteria (it must change from text to strikethrough).

toBe() vs. toEqual()

`toBe()` and `toEqual()` aren't the same in Jest. They behave differently. The easiest way to remember is that `toBe()` is `===` (strict equal), whereas `toEqual()` checks that two *objects* have the same value. Thus both assertions will be correct:

```
const copy1 = {
  name: 'React Quickly',
  chapters: 19,
}
const copy2 = {
  name: 'React Quickly',
  chapters: 19,
}

describe('Two copies of my books', () => {
  it('have all the same properties', () => {
    expect(copy1).toEqual(copy2) // correct
  })
  it('are not the same object', () => {
    expect(copy1).not.toBe(copy2) // correct
  })
})
```

But when you're comparing literals such as the number 5 and the string "Must have at least one uppercase character," `toBe()` and `toEqual()` will produce the same results:

```
expect(rules.length).toBe(5) // correct
expect(rules.length).toEqual(5) // correct
expect(fD(rules[0]).textContent).toEqual('Must have
➥ at least one upper-case character') // correct
expect(fD(rules[0]).textContent).toBe('Must have
➥ at least one upper-case character') // correct
```

There's a `TestUtils.findRenderedDOMComponentWithClass()` method that's similar to `TestUtils.scryRenderedDOMComponentsWithTag()` but returns only one element; it'll throw an error if you have more than one element. And to simulate user actions, there's a `TestUtils.Simulate` object that has methods with the names of events in camelCase: for example, `Simulate.click`, `Simulate.keyDown`, and `Simulate.change`.

Let's use `findRenderedDOMComponentWithClass()` to get the button and then use `Simulate.click` to click it. All this is done in the code without a browser:

```
let generateButton =
    ➥ TestUtils.findRenderedDOMComponentWithClass(password, 'generate-btn')
    expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).toBe('#text')
    TestUtils.Simulate.click(fD(generateButton))
    expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).toBe('strike')
```

This test checks that the `<li>` component has a `<strike>` element (to make the text strikethrough) when the button is clicked. The button generates a random password

that satisfies the second (`rules[1]`) criterion (as well as others), which is to have at least one lowercase character. You're finished here; let's move on to the next tests.

You've seen `TestUtils.Simulate` in action. It can trigger not just clicks, but other interactions as well, such as a change of value in an input field or an Enter keystroke (`keyCode` 13):

```
ReactTestUtils.Simulate.change(node)
ReactTestUtils.Simulate.keyDown(node, {
  key: "Enter",
  keyCode: 13,
  which: 13})
```

NOTE You must manually pass data that will be used in the component, such as `key` or `keyCode`, because TestUtils won't autogenerate it. There are methods in TestUtils for every user action supported by React.

For your reference, following is the project manifest file, package.json. It also includes the shallow-rendering library we'll cover next. To run the examples from ch16/password, install dependencies with `npm i` and then execute `npm test`:

```
{
  "name": "password",
  "version": "2.0.0",
  "description": "",
  "main": "index.html",
  "scripts": {
    "test": "jest",
    "test-watch": "jest --watch",
    "build-watch": "./node_modules/.bin/webpack -w",
    "build": "./node_modules/.bin/webpack"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.10.4",
    "babel-jest": "13.2.2",
    "babel-loader": "6.4.1",
    "babel-preset-react": "6.5.0",
    "jest-cli": "19.0.2",
    "react": "15.5.4",
    "react-dom": "15.5.4",
    "react-test-renderer": "15.5.4",
    "webpack": "2.4.1"
  }
}
```

Next, let's look at another way to render React elements.

16.4.3 Shallow rendering

In some cases, you may want to test a single level of rendering: that is, the result of `render()` in a component *without rendering its children* (if any). This simplifies testing because it doesn't require having a DOM—the system creates an element, and you can assert facts about it. First, you must have a package called `react-test-renderer` v15.5.4 (for older versions of React, this class was part of TestUtils, but it's not as of v15.5.4):

```
npm i react-test-renderer -SE
```

To illustrate, here's the same password element being tested with the shallow-rendering approach. This code can go in the same test file ch16/password/__tests__/password.test.js. In this case, you create a renderer and then pass a component to it to get its shallow rendering:

```
const { createRenderer } = require('react-test-renderer/shallow')
const passwordRenderer = createRenderer()
passwordRenderer.render(<Password/>)
let p = passwordRenderer.getRenderOutput()
expect(p.type).toBe('div')
expect(p.props.children.length).toBe(6)
```

Performs shallow rendering

Performs assert on the results of shallow rendering

Now, if you log `p` as in `console.log(p)`, the result contains the children but object `p` isn't a React instance. Look at this result of shallow rendering:

```
{ '$$typeof': Symbol(react.element),
  type: 'div',
  key: null,
  ref: null,
  props:
   { className: 'well form-group col-md-6',
     children: [ [Object], [Object], [Object], [Object],
     [Object], [Object] ] },
  _owner: null,
  _store: {} }
```

Contrast that with the logs of the results of `renderIntoDocument(<Password/>)`, which produces an instance of the `Password` React element with state. Look at this full rendering (not shallow):

```
Password {
  props: {},
  context: {},
  refs: {},
  updater:
   {...
     },
  state: { strength: {}, password: '',
    visible: false, ok: false },
```

You get state, which you don't get with shallow rendering.

```
generate: [Function: bound generate],
checkStrength: [Function: bound checkStrength],
toggleVisibility: [Function: bound toggleVisibility],
_reactInternalInstance:
 { _currentElement:                                    <-- You get an element that looks like the result of shallow rendering.
    { '$$typeof': Symbol(react.element),
      type: [Function: Password],
      key: null,
      ref: null,
      props: {},
      _owner: null,
      _store: {} },
      ...
   }
 }
```

Needless to say, you can't test user behavior and nested elements with shallow rendering. But shallow rendering can be used to test the first level of children in a component as well as the component's type. You can use this feature for custom (composable) component classes.

In the real world, you'd use shallow rendering for highly targeted (almost unit-like) testing of a single component and its rendering. You can use it when there's no need to test children, user behavior, or changing states of a component—in other words, when you only need to test the `render()` function of a single element. As a rule of thumb, start with shallow rendering and then, if that's not enough, continue with regular rendering.

Standard HTML classes can inspect and assert `el.props`, so there's no need for a shallow renderer. For example, this is how you can create an anchor element and test that it has the expected class name and tag name:

```
let el = <a className='btn'/>
expect(el.props.className).toBe('btn')
expect(el.type).toBe('a')
```

16.5 TestUtils wrap-up

You've learned a lot about TestUtils and Jest—enough to begin using them in your projects. That's exactly what you'll be doing in the projects in part 2 of this book: using Jest and TestUtils for behavior-driven development (BDD) of React components. (chapters 18–20). The password widget is in chapter 19, if you want to look at the Webpack setup and all the dependencies used in the real world.

For more information on TestUtils, refer to the official documentation at https://facebook.github.io/react/docs/test-utils.html. Jest is an extensive topic, and full coverage is outside the scope of this book. Feel free to consult the official API documentation to learn more: https://facebook.github.io/jest/docs/api.html#content.

Finally, the Enzyme library (https://github.com/airbnb/enzyme, http://mng.bz/Uy4H) provides a few more features and methods than TestUtils as well as more-compact names for methods. It's developed by AirBnb and requires TestUtils as well as jsdom (which comes with Jest, so you'll need jsdom only if you're not using Jest).

Testing is a beast. It's so frightful that some developers skip it—but not you. You stuck it out to the end. Congratulations! Your code will be better quality, and you'll develop more quickly and live a happier life. You won't have to wake up in the middle of the night to fix a broken server—or at least, not as frequently as someone without tests.

16.6 Quiz

1. Jest tests must be in a folder named which of the following? tests, __test__, or __tests__
2. TestUtils is installed with npm from `react-addons-test-utils`. True or false?
3. What TestUtils method allows you to find a single component by its HTML class?
4. What is the `expect` expression to compare objects (deep comparison)?
5. How do you test the behavior when the user hovers with a mouse? `TestUtils.Simulate.mouseOver(node)`, `TestUtils.Simulate.onMouseOver(node)`, or `TestUtils.Simulate.mouseDown(node)`

16.7 Summary

- To install Jest, use `npm i jest-cli --save-dev`.
- To test a module, turn off automocking for it with `jest.dontMock()`.
- Use `expect.toBe()` and other Expect functions.
- To install TestUtils, use `npm i react-addons-test-utils --save-dev`.
- Use `TestUtils.Simulate.eventName(node)`, where `eventName` is a React event (without the `on` prefix) to test trigger DOM events.
- Use `scry...` methods to fetch multiple elements.
- Use `find...` methods to fetch a single element (you'll get an error if you have more than one element: `Did not find exactly one match (found: 2+)`).

16.8 Quiz answers

1. __tests__. This is the convention Jest follows.
2. True. TestUtils is a separate npm module.
3. `findRenderedDOMComponentWithClass()`
4. `expect(OBJECT).toEqual(value)` compares objects on sameness without comparing that they're the same objects (which is done with `===` or `toBe()`).
5. `TestUtils.Simulate.mouseOver(node)`. The `mouseOver` event is triggered by hovering the cursor.