Chapter 10. About Testing and Debugging

React, thanks to components, makes it easy to test our applications. There are many different tools that we can use to create tests with React and we'll cover the most popular ones to understand the benefits they provide.

Jest is an all-in-one testing framework solution, maintained by *Christopher Pojer* from Facebook and contributors within the community, and aims to give you the best developer experience; but you can decide to create a custom setup with **Mocha** as well. We will look at both ways of building the best test environment.

You'll learn the difference between **Shallow rendering** and full DOM rendering with both **TestUtils** and **Enzyme**, how **Snapshot** Testing works, and how to collect some useful code coverage information.

Once the tools and their functionalities are well understood, we will cover a component from the Redux repository with tests and we'll look at some common testing solutions that can be applied in complex scenarios.

At the end of the chapter, you'll be able to create a test environment from scratch and write tests for your application's components.

In this chapter we will look at:

- Why it is important to test our applications, and how this help developers move faster
- How to set up a Jest environment to test components using the TestUtils
- How to set up the same environment with Mocha
- What Enzyme is and why it is a must-have for testing React applications
- How to test a real-world component
- The Jest snapshots and the Istanbul code coverage tools
- Common solutions for testing Higher-Order components and complex pages with multiple nested children
- The React developer tools and some error handling techniques

The benefits of testing

Testing web UIs has always been a hard job. From unit to end-to-end tests, the fact that the interfaces depend on browsers, user interactions, and many other variables makes it difficult to implement an effective testing strategy.

If you've ever tried to write end-to-end tests for the Web, you'll know how complex it is to get consistent results and how the results are often affected by false negatives due to different factors, such as the network. Other than that, user interfaces are frequently updated to improve experience, maximize conversions, or simply add new features.

If tests are hard to write and maintain, developers are less prone to cover their applications. On the other hand, tests are pretty important because they make developers more confident with their code, which is reflected in speed and quality. If a piece of code is well-tested (and tests are well-written) developers can be sure that it works and is ready to ship. Similarly, thanks to tests, it becomes easier to refactor the code because tests guarantee that the functionalities do not change during the rewrite.

Developers tend to focus on the feature they are currently implementing and sometimes it is hard to know if other parts of the application are affected by those changes. Tests help to avoid regressions because they can tell if the new code breaks the old tests. Greater confidence in writing new features leads to faster releases.

Testing the main functionalities of an application makes the codebase more solid and, whenever a new bug is found, it can be reproduced, fixed, and covered by tests so that it does not happen again in the future.

Luckily, React (and the component era), makes testing user interfaces easy and efficient. Testing components, or trees of components, is a less arduous job because every single part of the application has its responsibilities and boundaries.

If components are built in the right way, if they are pure and aim for composability and reusability, they can be tested as simple functions.

Another great power that modern tools bring to us the ability to run tests using Node and the console. Spinning up a browser for every single test makes tests slower and less predictable, degrading the developer experience; running the tests using the console instead is faster.

Testing components only in the console can sometimes give unexpected behaviors when they are rendered in a real browser but, in my experience, this is rare.

When we test React components we want to make sure that they work properly and that, given different sets of props, their output is always correct.

We may also want to cover all the various states that a component can have. The state might change by clicking a button so we write tests to check if all the event handlers are doing what they are supposed to do.

When all the functionalities of the component are covered but we want to do more, we can write tests to verify its behavior on **Edge cases**. Edge cases are states that the component can assume when, for example, all the props are `null` or there is an error. Once the tests are written, we can be pretty confident that the component behaves as expected.

Testing a single component is great, but it does not guarantee that multiple individually tested components will still work once they are put together. As we will see later, with React we can mount a tree of components and test the integration between them.

There are different techniques that we can use to write tests, and one of the most popular ones is **Test Driven Development (TDD)**. Applying TDD means writing the tests first and then writing the code to pass the tests.

Following this pattern helps us to write better code because we are forced to think more about the design before implementing the functionalities, which usually leads to higher quality.

Painless JavaScript testing with Jest

The most important way to learn how to test React components in the right way is by writing some code, and that is what we are going to do in this section.

The React documentation says that at Facebook they use Jest to test their components. However, React does not force you to use a particular test framework and you can use your favorite one without any problems.

In the next section, you will learn how it's possible to test components using Mocha.

To see Jest in action we are going to create a project from scratch, installing all the dependencies and writing a component with some tests. It'll be fun!

The first thing to do is to move into a new folder and run:

```
npm init
```

Once the `package.json` is created we can start installing the dependencies, with the first one being the `jest` package itself:

```
npm install --save-dev jest
```

To tell npm that we want to use the jest command to run the tests, we have to add the following scripts to the package.json:

```
"scripts": {
  "test": "jest"
},
```

In order to write components and tests using ES2015 and JSX, we have to install all Babel-related packages so that Jest can use them to transpile and understand the code.

The second set of dependencies is:

```
npm install --save-dev babel-jest babel-preset-es2015 babel-preset-
    react
```

As you may know, we now have to create a .babelrc file, which is used by Babel to know the presets and the plugins that we would like to use inside the project.

.babelrc looks like the following:

```
{
  "presets": ["es2015", "react"]
}
```

Now it is time to install React and ReactDOM, which we need to create and render components:

```
npm install --save react react-dom
```

The setup is ready and we can run Jest against ES2015 code and render our components into the DOM, but there is one more thing to do.

As we said, we want to be able to run the tests with Node and the console. To do that we cannot render the components using ReactDOM, because it needs the browser's DOM.

The Facebook team have created a tool, called TestUtils, which makes it easy to test React components using any testing framework.

Let's first install it and then we will see what functionalities it provides:

```
npm i --save-dev react-addons-test-utils
```

Now we have everything we need to test our components. The TestUtils library has functions to shallow render components or render them into a detached DOM outside the browser. It gives us some utility functions to reference the nodes rendered in the detached DOM so that we can make assertions and expectations on their values.

With TestUtils it is possible to simulate browser events and check whether the event handlers are set up correctly.

Let's start from the beginning by creating a component to test.

The component is a Button that will render a button element using the text prop and an event handler for the click event. To start with, we'll create only the skeleton, and we will write the implementation following the tests, using the TDD approach.

We need to implement a class for this component because the TestUtils have some limitations with stateless functional ones.

Let's create a button.js file and import React:

```
import React from 'react'
```

Now we can define the button as follows:

```
class Button extends React.Component
```

With an empty render function that returns a div just to make it work:

```
render() {
  return <div />
}
```

Last but not least we export the button:

```
export default Button
```

The component is now ready to be tested so let's start writing the first test, in a file called button.spec.js.

Jest looks into the source folder to find files that end with .spec, .test, or the ones inside a __tests__ folder; but you can change this configuration as you wish to fit the needs of your project.

Inside the button.spec.js we first import the dependencies:

```
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import Button from './button'
```

The first one is React, needed to write JSX, the second one is the TestUtils and we will see later how to use it. The last one is the previously created Button component.

The very first test that we can write to be sure that everything is working is the following one (I always start in this way):

```
test('works', () => {
  expect(true).toBe(true)
})
```

The test function accepts two parameters, with the first one being the description of the test and the second one a function that contains the actual test implementation. There is also a function called expect that can be used to make expectations on an object and it can be chained with other functions such as toBe, to check if the object passed to expect is the same as the object passed to toBe.

If we now open the terminal and run:

```
npm test
```

You should see the following output:

```
PASS  ./button.spec.js
✓ works (3ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.48s
Ran all test suites.
```

If your console output says PASS you are now ready to write real tests.

As we said, we want to test that the component renders correctly given some props, and that the event handlers do their job.

There are generally two ways of testing React components, by:

- Shallow rendering
- Mounting the components into a detached DOM

The first one is the most simple and easy to understands, we'll start with that. Shallow rendering allows you to render your component *one level deep* and it returns the result of the rendering so that you can run some expectations against it.

Rendering only one level means that we can test our component in isolation and, even if it has some complex children, they do not get rendered and they do not affect the results of the test if they should change or fail.

The first basic test that we can write is checking if the given text is rendered as a child of the button element.

The test starts like this:

```
test('renders with text', () => {
```

As a first operation we define the text variable we pass as a prop and that we will use to check if the right value is rendered:

```
const text = 'text'
```

Now it is time for Shallow rendering, which is implemented in these three lines:

```
const renderer = TestUtils.createRenderer()
renderer.render(<Button text={text} />)
const button = renderer.getRenderOutput()
```

The first one creates the `renderer`, the second one renders the `button` passing the text variable, and finally we get the output of the rendering.

The rendering output is similar to the following object:

```
{ '$$typeof': Symbol(react.element),
      type: 'button',
      key: null,
      ref: null,
      props: { onClick: undefined, children: 'text' },
      _owner: null,
      _store: {} }
```

You may recognize this as a React Element with the `type` property and the props. The second prop is the `children` element, which we want to make sure contains the right value.

Now that we know how the output looks, we can easily write our expectations:

```
expect(button.type).toBe('button')
expect(button.props.children).toBe(text)
```

Here we are declaring that we expect the `button` component to return an element of type `button` with the given text as children.

The last thing to do is to close the test block:

```
})
```

If you now switch to the console and type:

```
npm test
```

You should see something like this:

```
 FAIL  ./button.spec.js
● renders with text
    expect(received).toBe(expected)
    Expected value to be (using ===):
      "button"
    Received:
      "div"
```

The test fails, and this is something we expect when we run the tests for the first time doing TDD because the component has not been implemented yet. It just returns a `div` element. It's time to go back to the `button` component and make the tests pass. If we change the `render` method to something like:

```
render() {
  return (
    <button>
      {this.props.text}
    </button>
  )
}
```

And we run `npm test` again, we should see a green check mark in our console:

```
PASS  ./button.spec.js
✓ renders with text (9ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.629s
Ran all test suites.
```

Congratulations! Your first test on a component written using TDD passed.

Let's now see how we can write a test to check that an `onClick` handler passed to the component gets called when the button is clicked.

Before starting to write the code we should introduce two new concepts: mocks and detached DOM.

The first makes it easy to test the behavior of the functions inside a test. What we want to do here is pass a function to the button using the `onClick` property and verify that the function gets called when the user clicks it.

To do that, we have to create a special function called **mock** (or *spy*, depending on the framework) that behaves like a real function but has some special properties. For example, we can check if it has been called and, if so, how many times and with which parameters.

Jest is an all-in-one testing framework and it gives us all the tools we need to write a proper test. To create a mock function with Jest we can use: `jest.fn()`.

The second thing that we have to learn before writing the next test is that we cannot, using the `TestUtils`, simulate a DOM event using Shallow rendering.

The reason is that, to simulate an event with the `TestUtils`, we need to operate on a real component and not a simple React element.

So, in order to test a browser event, we have to `render` our component into a detached DOM. Rendering a component into a real DOM would require a browser but Jest comes with a special DOM in which we can render our components using the console.

The syntax for rendering a component into the DOM instead of using Shallow rendering is a little bit different; let's see what the test looks like.

We first create a new test block:

```
test('fires the onClick callback', () => {
```

With the title saying that we are going to check for the `onClick` callback to work properly.

Then we create the `onClick` mock, using the `jest` function:

```
const onClick = jest.fn()
```

Here comes the part where we fully render the component into the DOM:

```
const tree = TestUtils.renderIntoDocument(
  <Button onClick={onClick} />
)
```

If we log the tree, we can see that we get back the real component instance and not just the React Element.

For the same reason, we cannot simply inspect the object returned from the `renderIntoDocument` function; instead, we have to use a function from the `TestUtils` library to get the button element that we are looking for:

```
const button = TestUtils.findRenderedDOMComponentWithTag(
  tree,
  'button'
)
```

As the function name suggests, it searches for a component with the given tag name inside the tree.

Now we can use another function from the `TestUtils` to simulate an event:

```
TestUtils.Simulate.click(button)
```

The `Simulate` object accepts a function with the name of the event and the target component as a parameter.

Finally, we write the expectation:

```
expect(onClick).toBeCalled()
```

Here, we have simply said that we expect the `mock` function to be called.

If we run the tests with `npm test` they will fail because the `onClick` functionality hasn't been implemented yet.

```
 FAIL  ./button.spec.js
● fires the onClick callback
    expect(jest.fn()).toBeCalled()

    Expected mock function to have been called.
```

This is how the TDD process works. Let's now move back to `button.js` and implement the event handler, modifying the `render` as follows:

```
render() {
  return (
    <button onClick={this.props.onClick}>
      {this.props.text}
    </button>
  )
}
```

If we now run `npm` test again the tests are now green:

```
PASS  ./button.spec.js
✓ renders with text (10ms)
✓ fires the onClick callback (17ms)
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.401s, estimated 2s
Ran all test suites.
```

Our component is fully tested and correctly implemented.

Mocha is a flexible testing framework

In this section, we will see how it's possible to achieve the same results with Mocha in order to make it clear that, with React, you can use any testing framework. Also, it is good to learn the main differences between Jest, which is an integrated test framework and tries to automate all operations to provide a smooth developer experience, and Mocha, which does not make any assumptions on the tools you need. With Mocha, it is up to you to install all the different packages you need to test React in the right way.

Let's create a new folder and initialize a new `npm` package with:

```
npm init
```

The first thing to install is the `mocha` package:

```
npm install --save-dev mocha
```

As with Jest, to be able to write ES2015 code and JSX we have to ask Babel for some help: To make it work with Mocha, we have to install the following packages:

```
npm install --save-dev babel-register babel-preset-es2015 babel-
    preset-react
```

Now that Mocha and Babel have been installed we can set up the test script as follows:

```
"scripts": {
  "test": "mocha --compilers js:babel-register"
},
```

We are telling `npm` to run `mocha` as a test task with the compiler flag set in such a way that the JavaScript files are processed using the Babel register.

The next step is installing React and ReactDOM:

```
npm install --save react react-dom
```

And the `TestUtils`, which we will need to render our components in the test environment:

```
npm install --save-dev react-addons-test-utils
```

The basic functionalities for testing with Mocha are ready but to get feature parity with Jest we need three more packages.

The first one is `chai`, which lets us write expectations in a similar way to how we wrote them with Jest. The second is `chai-spies`, which provides the `spies` functionality that we can use to inspect the `onClick` callback and check if it has been called.

Last but not least, we must install `jsdom`, which is the package that we use to create the detached DOM and let the TestUtils do their job in the console without a real browser:

```
npm install --save-dev chai chai-spies jsdom
```

We are now ready to write the tests and we'll use the same `button.js` created earlier. It's already been implemented so we will not follow the TDD process this time, but the goal here is to show the main differences between the two frameworks.

Mocha expects the tests to be in the test folder so we can create it and put a `button.spec.js` file inside it.

The first thing we have to do is import all the dependencies:

```
import chai from 'chai'
import spies from 'chai-spies'
import { jsdom } from 'jsdom'
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import Button from '../button'
```

As you may notice here, there are way more compared to Jest because Mocha gives you the freedom to choose your tools.

The next step is telling `chai` to use the spies package:

```
chai.use(spies)
```

And then we extract the functionalities that we need from the `chai` package. We will use them later in the test implementation:

```
const { expect, spy } = chai
```

Another operation is to set up `jsdom` and provide the DOM object needed to render the React components:

```
global.document = jsdom('')
global.window = document.defaultView
```

Finally, we can write the first test. Typically with Mocha, you have two functions to define your test: the first one is describe, which wraps a set of tests for the same module, and then there is it, which is where tests are implemented.

In this case we describe the behavior of the button:

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

And then we write the first test where we expect the type and the text to be the correct ones:

```
it('renders with text', () => {
```

We define the text variable that we will use to check the validity of the expectation:

```
const text = 'text'
```

We then shallow-render the components as we did before:

```
const renderer = TestUtils.createRenderer()
renderer.render(<Button text={text} />)
const button = renderer.getRenderOutput()
```

And we finally write the expectations:

```
expect(button.type).to.equal('button')
expect(button.props.children).to.equal(text)
```

As you can see, here the syntax is slightly different. Instead of having the `toBe` function, we use the chain `to.equal` provided by `chai`. The result is the same: a comparison between the two values.

Now we can close the first test block:

```
})
```

The second test block where we test that the `onClick callback` is fired looks like this:

```
it('fires the onClick callback', () => {
```

We then create the spy, in a similar way before:

```
const onClick = spy()
```

We render the button into a detached DOM using the `TestUtils`:

```
const tree = TestUtils.renderIntoDocument(
  <Button onClick={onClick} />
)
```

And we use the `tree` to find the component by tag:

```
const button = TestUtils.findRenderedDOMComponentWithTag(
  tree,
  'button'
)
```

The next step is to simulate the button click:

```
TestUtils.Simulate.click(button)
```

Finally, we write the expectation:

```
expect(onClick).to.be.called()
```

Again, the syntax is slightly different but the concept does not change: we check for the spy to be called.

Now, run the `npm` test inside the `root` folder with Mocha, and we should get the following message:

```
Button
✓ renders with text
✓ fires the onClick callback

2 passing (847ms)
```

This means that our tests have passed and we are now ready to use Mocha to test real components.

JavaScript testing utilities for React

At this point, it should be clear how to test basic components with Jest and Mocha and what the pros and cons are of both.

You have also learned what the TestUtils are and the difference between Shallow rendering and full DOM Rendering.

You may have noticed that the TestUtils, even if they provide a useful tool to help testing components, are verbose and sometimes it is not easy to find the right approach to get the reference to the elements and their properties.

That is the reason why the developers at *AirBnb* decided to create Enzyme, a tool built on top of the TestUtils that makes it easy to manipulate the rendered components.

The API is nicer, similar to jQuery, and it provides many useful utilities to interact with components, their states, and their properties.

Let's see what it means to convert our Jest tests to use Enzyme instead of the TestUtils.

So, please go back to the `Jest` project we previously created and install `enzyme`:

```
npm install --save-dev enzyme
```

Now open the `button.spec.js` and change the import statements to be similar to the following snippets:

```
import React from 'react'
import { shallow } from 'enzyme'
import Button from './button'
```

As you can see, instead of importing the `TestUtils` we are importing the shallow function from Enzyme. As the name suggests the shallow function provides Shallow rendering functionalities and it has some special features.

First of all, with Enzyme it is possible to simulate events even with shallow-rendered elements, which we couldn't do with the `TestUtils`. And, most importantly, the object returned by the function is not a simple React element but rather **ShallowWrapper**, a special object with some useful properties and functions, which we'll look at next.

Let's begin updating the tests, starting with the **renders with text** ones. The first line stays the same because we still need the text variable:

```
const text = 'text'
```

The Shallow rendering part becomes way more simple and intuitive. We can, in fact, replace the three lines of the `TestUtils` with the following one:

```
const button = shallow(<Button text={text} />)
```

The button object represents a ShallowWrapper that has some useful methods that we will use in the new expectation statements:

```
expect(button.type()).toBe('button')
expect(button.text()).toBe(text)
```

As you can see here, instead of checking object properties (they are likely to change as React gets updated), we are using utility functions that abstract the functionalities.

The function `type` returns the type of the rendered elements, and the function `text` returns the text being rendered inside the component. In our case, it's the same text that we passed as prop.

Your test should now look like this:

```
test('renders with text', () => {
  const text = 'text'
  const button = shallow(<Button text={text} />)

  expect(button.type()).toBe('button')
  expect(button.text()).toBe(text)
})
```

This is way more concise and clean than before.

The next thing to do is to update the tests related to the `onClick` event handler. Again, the first line stays the same:

```
const onClick = jest.fn()
```

We still ask Jest to create a mock that we can spy on in the expectations.

We can replace the line where we used the `renderIntoDocument` method to render the component into the detached DOM with the following one:

```
const button = shallow(<Button onClick={onClick} />)
```

We also don't need to find the button with `findRenderedDOMComponentWithTag` because Shallow rendering already references it.

The syntax to simulate an event is slightly different from the `TestUtils` but it is still intuitive:

```
button.simulate('click')
```

Every ShallowWrapper has a `simulate` function that we can call, passing the name of the event and some parameters as a second argument. In this case, we do not need any argument but we will see in the following section how it can be useful when testing forms.

Finally, the expectation remains the same:

```
expect(onClick).toBeCalled()
```

The final code is something like this:

```
test('fires the onClick callback', () => {
  const onClick = jest.fn()
  const button = shallow(<Button onClick={onClick} />)

  button.simulate('click')

  expect(onClick).toBeCalled()
})
```

Migrating to Enzyme is relatively easy, and it makes the code more readable.

The library provides some very useful APIs to find nested elements, or to use selectors to search elements given their class names or types.

There are methods to help make assertions and expectations on props, and functions to inject arbitrary state and context into components.

Other than Shallow rendering, which we can use to cover most scenarios, there is a `mount` method that can be imported from the library and renders the tree into the DOM.

A real-world testing example

We have a working test set up and a clear understanding of what the different tools and libraries can do for us. The next thing to do is to test a real-world component.

The `Button` we used in the previous example was great and we should always try to make our components as simple as possible; but sometimes we need to implement different kinds of logic and state, which makes the tests more challenging.

The component we are going to test is **TodoTextInput** from the Redux **TodoMVC** example:

https://github.com/reactjs/redux/blob/master/examples/todomvc/src/components/TodoTextInput.js

You can easily copy it into your `Jest` project folder.

It represents a nice example because it has various props, its class names change according to the props it receives, and it has three event handlers that implement a bit of logic that we can test.

The TodoMVC example is a *standard* way of creating a real-world application with the different frameworks in order to compare their features and make it easier for developers to choose.

The result is a simple app that lets users add to-do items and mark them as done. The component we are targeting is the text input used to add and edit items.

It is worth going quickly through the implementation of the component so that we can understand what it does before testing.

First of all, the component is defined using a class:

```
class TodoTextInput extends Component
```

The `propTypes` are defined as a static prop of the class:

```
static propTypes = {
  onSave: PropTypes.func.isRequired,
  text: PropTypes.string,
  placeholder: PropTypes.string,
  editing: PropTypes.bool,
  newTodo: PropTypes.bool
}
```

To make class properties work with Babel we need a plugin called transform-class-properties, which we can easily install with:

```
npm install --save-dev babel-plugin-transform-class-properties
```

And then we add it to the list of Babel plugins in our `.babelrc`:

```
"plugins": ["transform-class-properties"]
```

The state is defined as a `class` property as well:

```
state = {
  text: this.props.text || ''
}
```

Its default value can be the text coming from the props or an empty string.

Then there are three event handlers, which are defined using a `class` property and an arrow function so that they do not need to be manually bound in the constructor.

The first one is the submit handler:

```
handleSubmit = e => {
  const text = e.target.value.trim()
  if (e.which === 13) {
    this.props.onSave(text)
    if (this.props.newTodo) {
      this.setState({ text: '' })
    }
  }
}
```

The function receives the event; it trims the value of the target element, then it checks if the key that fired the event is the *Enter* key (13) and, if it is, it passes the trimmed value to the `onSave` prop function. If the `newTodo` prop is `true`, then the state is set again to an empty string.

The second event handler is the change handler:

The second event handler is the change handler:

```
handleChange = e => {
  this.setState({ text: e.target.value })
}
```

Apart from its being defined as a `class` property you should be able to recognize the method that keeps the state updated for a controlled input element.

The last one is the blur handler:

```
handleBlur = e => {
  if (!this.props.newTodo) {
    this.props.onSave(e.target.value)
  }
}
```

This fires the `onSave` prop function with the value of the field if the prop `newTodo` is `false`.

Finally, there is the `render` method, where the input element is defined with all its properties:

```
render() {
  return (
    <input className={
      classnames({
        edit: this.props.editing,
        'new-todo': this.props.newTodo
      })}
      type="text"
      placeholder={this.props.placeholder}
      autoFocus="true"
      value={this.state.text}
      onBlur={this.handleBlur}
      onChange={this.handleChange}
      onKeyDown={this.handleSubmit} />
  )
}
```

To apply the class name, the `classnames` function is used; it comes from a very useful package, authored by *Jed Watson*, which makes it simple to apply classes using a conditional logic.

Then some static attributes such as `type` and `autofocus` are set, and the value to control the input is set using the `text` property, which is updated on each change. Finally, the event handlers are set to the event properties of the element.

Before starting, it is important to have a clear idea about what we want to test and why. Looking at the component, it is pretty easy to figure out which are the important parts to cover. In this case, it's testing legacy code that we may inherit from other teams, or code that we might find when joining a new company.

The following list represents more or less all the variations and functionalities of the component that are worth testing:

- The state is initialized with the value coming from the props
- The placeholder prop is correctly used in the element
- The right class names are applied following the conditional logic
- The state is updated whenever the value of the field changes
- The onSave callback is fired according to the different states and conditions

It is now time to start writing code and create a file called `TodoTextInput.spec.js` with the following import statements:

```
import React from 'react'
import { shallow } from 'enzyme'
import TodoTextInput from './TodoTextInput'
```

We import React, the Shallow rendering function from Enzyme, and the component to test. We also create a utility function that we will pass to the required `onSave` property in some tests:

```
const noop = () => {}
```

We can now write the first test, where we check that the value passed to the component as a text prop is used as a default value of the element:

```
test('sets the text prop as value', () => {
  const text = 'text'
  const wrapper = shallow(
    <TodoTextInput text={text} onSave={noop} />
  )

  expect(wrapper.prop('value')).toBe(text)
})
```

The code is pretty straightforward: We create a text variable, which we pass to the component when we shallow-render it. As you can see, we pass the `noop` utility function to the `onSave` prop because we do not care about it, but it is required and React would complain if we passed `null`.

Finally, we render the component and check that the value prop of the output element is equal to the given text. If we now run `npm test` in the console we can see the following output, which means that the test passes:

```
PASS  ./TodoTextInput.spec.js
✓ sets the text prop as value (10ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.384s
Ran all test suites.
```

Awesome, let's continue with the next tests. The second test is pretty similar to the first one, except that we test the `placeholder` property:

Awesome, let's continue with the next tests. The second test is pretty similar to the first one, except that we test the `placeholder` property.

```
test('uses the placeholder prop', () => {
  const placeholder = 'placeholder'
  const wrapper = shallow(
    <TodoTextInput placeholder={placeholder} onSave={noop} />
  )

  expect(wrapper.prop('placeholder')).toBe(placeholder)
})
```

If we run `npm test`, it will now tell us that the two tests are green.

Let's now delve into some more interesting stuff and test whether the class names get applied to the element when the related props are present:

```
test('applies the right class names', () => {
  const wrapper = shallow(
    <TodoTextInput editing newTodo onSave={noop} />
  )

  expect(wrapper.hasClass('edit new-todo')).toBe(true)
})
```

In this test we add the editing and `newTodo` props and we check inside the expect function that the classes have been applied to the output.

We could check the classes separately to make sure that one is not affecting the other but you should get the point.

Now the tests start getting more complex because we want to test the behavior of the component when the key down event happens.

In particular, we want to check that, if the *Enter* key is pressed, the `onSave` callback gets called with the value of the element:

```
test('fires onSave on enter', () => {
  const onSave = jest.fn()
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('keydown', { target: { value }, which: 13 })

  expect(onSave).toHaveBeenCalledWith(value)
})
```

We first create a mock using the `jest.fn()` function; we then create a value variable to store the value of the event, which we also use to double-check that the function gets called with the same value. Then we'll render the component and finally we'll simulate the key down event passing an event object.

The event object has two properties: the target, which represents the element that initiated the event and has a `value` property, which represents the key code of the key that has been pressed.

The expectation here is that the mock `onSave` callback gets called with the value of the event.

The `npm test` now tells us that four tests have passed.

A similar test to the previous one can check that, if the pressed key is not the *Enter* key, nothing happens:

```
  test('does not fire onSave on key down', () => {
  const onSave = jest.fn()
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('keydown', { target: { value: '' } })

  expect(onSave).not.toBeCalled()
})
```

The test is pretty similar to the previous one but it is important to notice the difference in the expect statement. We are using a new property, `.not`, which asserts the opposite on the following function; in this case `toBeCalled` is supposed to be `false`.

As you can see, the way we write expectations is very similar to the spoken language.

With five green tests, we can move into the next one:

```
test('clears the value after save if new', () => {
  const value = 'value'
  const wrapper = shallow(<TodoTextInput newTodo onSave={noop} />)

  wrapper.simulate('keydown', { target: { value }, which: 13 })

  expect(wrapper.prop('value')).toBe('')
})
```

The difference with the previous test is that now there is the `newTodo` prop on the element, which forces the value to be reset.

Running `npm test` in the console gives us a green bar with six passed tests.

The following test is a very common one:

```
test('updates the text on change', () => {
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={noop} />)

  wrapper.simulate('change', { target: { value } })

  expect(wrapper.prop('value')).toBe(value)
})
```

This test checks that the controlled input works properly and, if you think about the forms we discussed in Chapter 6 , *Write Code for the Browser*, this is a must-have for all the forms in your application.

We simulate the change event passing a set value and we check that the `value` property of the output element is equal to it.

The green tests are now seven in total and there is only one more to go.

In the last test we check that the blur event fires the callback only when the To-do item is not new:

```
test('fires onSave on blur if not new', () => {
```

```
  const onSave = jest.fn()
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('blur', { target: { value } })

  expect(onSave).toHaveBeenCalledWith(value)
})
```

We set up a mock, create the expected value, simulate the blur event using the target, and then we check that the onSave callback has been called with the given value.

If we run npm test now, everything should be green and the list of passed tests is now pretty long:

Copy

```
PASS  ./TodoTextInput.spec.js
✓ sets the text prop as value (10ms)
✓ uses the placeholder prop (1ms)
✓ applies the right class names (1ms)
✓ fires onSave on enter (3ms)
✓ does not fire onSave on key down (1ms)
✓ clears the value after save if new (5ms)
✓ updates the text on change (1ms)
✓ fires onSave on blur if not new (2ms)
Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        2.271s
Ran all test suites.
```

Good job, the component is now covered with tests; and if we need to refactor it, change its behavior, or add some features, the tests will help us to discover whether the new code breaks any of the old functionalities.

This makes us more confident with our code so that we can touch any line without worrying about regressions.

React tree Snapshot Testing

Now that you have seen a real-world testing example you may think that writing so many tests for a single component is a time-consuming task and not worth it.

Checking for each variation of text, value, and class name is laborious and it requires much code to cover all instances. However, most of the time, whenever we test components, the most important thing for us is that the output is correct and that it does not change unexpectedly. There is a new feature introduced in Jest that helps with this problem and it's called **Snapshot Testing**.

Snapshots are *pictures* of the component with some props at a given point in time. Every time we run the tests, Jest creates new *pictures* and it compares them with the previous ones to check if something has changed.

The content of the snapshot is the output of the `render` method of a package called `react-test-renderer`, which has to be installed with the following command:

Copy

```
npm install --save-dev react-test-renderer
```

Once the test rendered is ready, we can create a new file called `TodoTextInput-snapshot.spec.js` with the following import statements:

Copy

```
import React from 'react'
import renderer from 'react-test-renderer'
import TodoTextInput from './TodoTextInput'
```

We import React to be able to use JSX, the `renderer` that creates the tree for the snapshot, and finally the target component that we want to test.

All the dependencies are now available and we can write a simple test:

Copy

```
test('snapshots are awesome', () => {
```

In the first line we render the component using the `renderer` that we previously imported:

Copy

```
const component = renderer.create(
  <TodoTextInput onSave={() => {}} />
)
```

What we receive back is the instance of the component, which has a special function called `toJSON` that we call in the following line:

Copy

```
const tree = component.toJSON()
```

The resulting tree is the React element returned from the original component that Jest will use to generate the snapshot to be compared with the next ones.

If we log the tree in the console, we can see what it looks like:

Copy

```
{ type: 'input',
       props:
        { className: '',
          type: 'text',
          placeholder: undefined,
          autoFocus: 'true',
          value: '',
          onBlur: [Function],
          onChange: [Function],
          onKeyDown: [Function] },
       children: null }
```

Finally, we write the expectation statement checking if the tree matches the previously saved snapshot:

Copy

```
expect(tree).toMatchSnapshot()
```

The first time we run the tests with `npm test` the snapshot is newly created and stored in a `__snapshots__` folder.

Each file inside that folder represents a snapshot. If we look into it we don't find the React element object, but a human-readable version of the rendered output:

```
exports[`test snapshots are awesome 1`] = `
<input
  autoFocus="true"
  className=""
  onBlur={[Function]}
  onChange={[Function]}
  onKeyDown={[Function]}
  placeholder={undefined}
  type="text"
  value="" />
`;
```

If we now go back to the test, add the `editing` prop to the component, and run `npm test` again, we get the following response in the console:

```
 FAIL  ./TodoTextInput-snapshot.spec.js
● snapshots are awesome
    expect(value).toMatchSnapshot()
    Received value does not match stored snapshot 1.
    - Snapshot
    + Received
    @@ -1,8 +1,8 @@
     <input
       autoFocus="true"
    -  className=""
    +  className="edit"
       onBlur={[Function]}
       onChange={[Function]}
       onKeyDown={[Function]}
       placeholder={undefined}
       type="text"
```

It shows us what's changed in the current snapshot. In this case, it's the `className` property, which was empty before and now contains the edit string.

A few lines below, we can see this message:

```
Inspect your code changes or run with npm test -- -u to update
    them.
```

Snapshots make the developer experience incredibly smooth; with a simple flag, we can confirm if the new Snapshot reflects the correct version of the component. Running `npm test -- -u` updates the snapshot automatically.

If the component has been changed by mistake instead, we can go back to the code and fix it.

As you can see, Snapshot Testing is a powerful feature that makes testing components easier and helps developers save time by avoiding the need to write tests for all the variants.

Code coverage tools

There are many reasons to write tests and we went through some of them in the previous section. The main one is always to provide value to our codebase and make it more robust.

Because of that, I am always skeptical when it comes to counting the number of tests, the lines of code, and the test coverage. I suggest people don't focus on numbers but on the value that the tests can provide.

However, in some scenarios it is useful to get some measurement of the coverage and keep track of the numbers. In big projects with many different modules, doing so makes it easy to spot files that have not been adequately tested or that have not been tested at all.

Once again, Jest provides all the equipment you need to run your tests and of course it provides the functionalities to measure and store the code coverage information.

It uses Istanbul, one of the most popular code coverage libraries, which you will have to install manually if you are using Mocha.

Running the coverage with Jest is pretty straightforward: You just need to add the `--coverage` flag to the `Jest` command in the `npm` scripts. Alternatively, you can create a configuration section for Jest in `package.json` and set the `collectCoverage` option to true.

```
"jest": {
  "collectCoverage": true
}
```

If you now run `npm test` again you can see a different output in the console, where a table shows some information about the coverage:

```
-------------------|----------|----------|----------|----------|-------------------|
File               |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines    |
-------------------|----------|----------|----------|----------|-------------------|
All files          |      100 |     87.5 |      100 |      100 |                   |
 TodoTextInput.js  |      100 |     87.5 |      100 |      100 |                   |
-------------------|----------|----------|----------|----------|-------------------|
```

As you can see, our file is almost fully covered. The first column shows how many statements are covered; the second column shows the different branches of conditional statements; the third measures the functions that have been tested; and the fourth column shows the lines of code covered by tests. The last column, which is currently empty, would tell you which lines are uncovered, and that is pretty useful information when it comes to quickly finding the parts of the code that need more attention.

Currently, the only value that is not 100% covered is the branches and, in fact, I left one branch of the last test uncovered on purpose so we can achieve full coverage together.

If you open the `TodoTextInput.js` file and check the `onBlur` handler, you'll notice that there are two branches:

```
handleBlur = e => {
  if (!this.props.newTodo) {
    this.props.onSave(e.target.value)
  }
}
```

When the to-do is not new, the `onSave` prop function is fired with the current value of the field; when the to-do is new nothing happens.

We have tested only the first path, which is the most obvious but often it's worth testing all the different branches to make sure that everything is working properly.

Let's move back to the `TodoTextInput.spec.js` and add a new test:

```
test('does not fire onSave on blur if new', () => {
  const onSave = jest.fn()
  const wrapper = shallow(
    <TodoTextInput newTodo onSave={onSave} />
  )

  wrapper.simulate('blur')

  expect(onSave).not.toBeCalled()
})
```

The test is similar to the last one in the file except that we pass the `newTodo` prop to the component and we check that the `onSave` callback does not get called.

If we now run `npm test` again, we can see that all the columns are now showing 100%.

Common testing solutions

In this last section about testing, we will go through some common patterns that are useful to know when testing complex components.

Testing components should now be familiar to you, and you should have all the information to start writing tests for your applications. However, sometimes is not easy to figure out the best strategy to test, for example, **Higher-Order Components**.

Testing Higher-Order Components

As we have seen in previous chapters, we can use Higher-Order Components to share functionalities between different components across the application. HoCs are functions that take a component and return an enhanced version of it.

Testing this kind of component is not as intuitive as testing simple ones, so it is worth looking at some common solutions together.

Our target component is going to be the `withData` HoC created in Chapter 5, *Proper Data Fetching*. We will only apply a small variation in the way data fetching is performed.

The `withData` function has the following signature:

```
const withData = URL => Component => (...)
```

It takes the URL of the endpoint where the data has to be loaded, and it provides such data to the target component. The URL can be a function that receives the current props or a static string.

The `withData` function returns a class defined as follows:

```
class extends React.Component
```

With a `constructor` where the data is initialized:

```
constructor(props) {
  super(props)

  this.state = { data: [] }
}
```

The data is loaded in the `componentDidMount` lifecycle hook:

```
componentDidMount() {
  const endpoint = typeof url === 'function'
    ? url(this.props)
    : url

  getJSON(endpoint).then(data => this.setState({ data }))
}
```

As you can see, there is a small difference from the example in Chapter 5, *Proper Data Fetching*, because, instead of using the fetch function directly, we are using `getJSON` so that you can learn how to mock external modules.

It is also best practice to wrap third-party libraries and abstract API calls into separate modules, so when we test a component we can isolate it from its dependencies.

The `getJSON` function is imported at the top of the file:

```
import getJSON from './get-json'
```

It returns a promise with the JSON data retrieved from the endpoint.

Finally, we `render` the target component spreading the props and the state:

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

Now, there are a few different things that we want to cover with tests in this function and we will start with the most simple ones. For example, a simple task could be to check whether the props received from the enhanced component are correctly passed to the target.

After that, we'd test if the logic of creating the endpoint from the URL works for both function and static strings.

Ultimately, we want to test that, as soon as the `getJSON` function returns the data, the target component receives it.

In the test file we first load all the dependencies:

```
import React from 'react'
import { shallow, mount } from 'enzyme'
import withData from './with-data'
import getJSON from './get-json'
```

We are importing both the `shallow` and `mount` functions from Enzyme because the simple tests can be run without the DOM, but since we need to test what happens during the lifecycle hooks we need `mount` as well.

Then, we create a couple of variables that we will use within the tests:

```
const data = 'data'
const List = () => <div />
```

This is the mock data that we use to check whether the information is correctly passed from the fetch to the component, and a dummy `List` component.

Creating a dummy component is pretty common practice when testing HoC because we need a target component to enhance, so we can figure out if all the features are correctly working.

Now comes the hardest part and the most powerful one:

```
jest.mock('./get-json', () => (
  jest.fn(() => ({ then: callback => callback(data) }))
))
```

As we said, we are using an external component to fetch the data. One thing that we want to avoid is loading real data and, most importantly, we do not want our test to fail if the external module fails for some reasons. With Jest it is very easy to isolate and mock the dependencies.

Using the `jest.mock` function we are telling the test runner to replace the external module with the function we passed as a second parameter. The function returns a mock function from `jest.fn` returns an object that looks like a promise but is synchronous. It has a then function that fires the received callback, passing to it the fake data we previously defined.

From now on we can unit-test the Higher-Order Component without worrying about the behavior or the bugs of the `getJSON` function.

We are now ready to write the real tests, with the first one being where we check if the props are passed correctly down to the target:

```
test('passes the props to the component', () => {
  const ListWithGists = withData()(List)
  const username = 'gaearon'

  const wrapper = shallow(<ListWithGists username={username} />)

  expect(wrapper.prop('username')).toBe(username)
})
```

It should be pretty clear, but let's see together what it does. We first enhance the dummy `List` that we are passing to the HoC, we then define a prop that we pass to the component and we shallow-render it.

Finally, we check if the output has a prop with the same value. Great; if we run `npm test` we see that the first test passes.

Let's now move to the tests that require mounting the component to the detached DOM. First of all, we check whether both the URL function and the static string work. The static string is pretty easy to test:

```
test('uses the string url', () => {
  const url = 'https://api.github.com/users/gaearon/gists'
  const withGists = withData(url)
  const ListWithGists = withGists(List)

  mount(<ListWithGists />)

  expect(getJSON).toHaveBeenCalledWith(url)
})
```

We define a URL and we use the partial application to generate a new function; we then use it to enhance the dummy `List`.

Then, we mount the component and check that the `getJSON` function is called with the URL that we passed. Easy: two green tests.

Now we want to check if the URL function works:

```
test('uses the function url', () => {
  const url = jest.fn(props => (
    `https://api.github.com/users/${props.username}/gists`
  )
  const withGists = withData(url)
  const ListWithGists = withGists(List)
  const props = { username: 'gaearon' }

  mount(<ListWithGists {...props} />)

  expect(url).toHaveBeenCalledWith(props)
  expect(getJSON).toHaveBeenCalledWith(
    'https://api.github.com/users/gaearon/gists'
  )
})
```

We first generate the URL function using a Jest mock so that we can write an expectation on it, then we enhance the `List` and define the prop that gets passed to the component.

Finally, we write two expectations:

- The first one checks that the URL function has been called with the given props
- The second one checks again that the `getJSON` function is fired with the right endpoint

Now comes the final part where we check that the data returned to the `getJSON` module is passed correctly to the target component:

```
test('passes the data to the component', () => {
  const ListWithGists = withData()(List)

  const wrapper = mount(<ListWithGists />)

  expect(wrapper.prop('data')).toEqual(data)
})
```

We first enhance the `List` using the HoC, we then mount the component, and we save a reference of the wrapper. Then we search for the `List` component inside the mounted wrapper and check that its `data` property is the same as the data returned from the fetching.

If we run `npm test` again we see that four tests are now passing.

What you learned in this section is how to test Higher-Order Components using a dummy child and how to isolate the current component mocking external dependencies.

The Page Object pattern

Let's now move into another common way of writing tests when the tree of components becomes more complex and there are multiple nested children.

For this example, we will use the **Controlled form component** created in Chapter 6 , *Write Code for the Browser*:

```
class Controlled extends React.Component
```

Let's go quickly through its function to remind ourselves how it works, and then we can talk about testing.

Inside the `constructor` we initialize the state and bind the handlers:

```
constructor(props) {
  super(props)

  this.state = {
    firstName: 'Dan',
    lastName: 'Abramov',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

The `handleChange` handler keeps the value of the fields updated inside the state:

```
handleChange({ target }) {
  this.setState({
    [target.name]: target.value,
  })
}
```

Then there is the `handleSubmit` handler, where we use the `preventDefault` function of the event object to disable the default behavior of the browser when the form is submitted. We also call the `onSubmit` prop function received from the parent, passing it the value of the fields concatenated.

This last function was not present in the original controlled example, but we need it here it to show how to test the component properly:

```
handleSubmit(e) {
  e.preventDefault()

  this.props.onSubmit(
    `${this.state.firstName} ${this.state.lastName}`
  )
}
```

Last but not least, we have the `render` method where we define the fields and we attach the handler functions:

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input
        type="text"
        name="firstName"
        value={this.state.firstName}
        onChange={this.handleChange}
      />
      <input
        type="text"
        name="lastName"
        value={this.state.lastName}
        onChange={this.handleChange}
      />
      <button>Submit</button>
    </form>
  )
}
```

One basic functionality that we may want to test here is typing something into the fields and submitting the form results in the `onSumbit` callback being called with the entered values.

It should be clear to you how to write a simple test to cover this case with Enzyme, so let's check it together:

```
test('submits the form', () => {
```

First of all we define an `onSubmit` mock function using Jest and we mount the component storing a reference to the wrapper:

```
const onSubmit = jest.fn()
const wrapper = shallow(<Controlled onSubmit={onSubmit} />)
```

Secondly, we find the first field and we fire the change event on it, passing the value that we want to update:

```
const firstName = wrapper.find('[name="firstName"]')
firstName.simulate(
  'change',
  { target: { name: 'firstName', value: 'Christopher' } }
)
```

We then do the same thing with the second field:

```
const lastName = wrapper.find('[name="lastName"]')
lastName.simulate(
  'change',
  { target: { name: 'lastName', value: 'Chedeau' } }
```

```
)
```

Once the fields are updated, we submit the form simulating an event:

```
const form = wrapper.find('form')
form.simulate('submit', { preventDefault: () => {} })
```

Now, let's write the expectations:

```
expect(onSubmit).toHaveBeenCalledWith('Christopher Chedeau')
```

And close the test block:

```
})
```

Running `npm test` in the console shows a green message, which is good; but if you look at the implementation of the test you can easily spot some problems and potential optimization issues.

The most visible one is that the code for filling the fields is duplicated, apart from some variables. The code is verbose and, most importantly, it's coupled with the structure of the markup.

If we have multiple tests like this and then we change the markup, we have to update the code in many parts of the file. Wouldn't it be nice to remove the duplication and move the selectors to one single place so that it is easier to change them if the form changes?

This is where the **Page Object pattern** come to the rescue. If we create a Page object that represents the elements of the page and hides the selectors, and we use it to fill the fields and submit the form, we'll get many benefits and avoid duplicated code.

It is fair to say that usually being **Don't Repeat Yourself (DRY)** in the tests is not the best approach because the risk is to add more bugs and complexity, but in this case it is worth it.

Let's see how the controlled form test can be improved thanks to the Page Object pattern.

First of all we have to create a `Page` object using a class:

```
class Page
```

The class has a `constructor` that receives the root `wrapper` from Enzyme and stores it for future usage:

```
constructor(wrapper) {
  this.wrapper = wrapper
}
```

Then we define a generic function to fill the fields that accepts `name` and `values` and fires the change event:

```
fill(name, value) {
  const field = this.wrapper.find(`[name="${name}"]`)
  field.simulate('change', { target: { name, value } })
}
```

Then we implement a `submit` function to abstract the part where we look for the button and simulate the browser event:

```
submit() {
  const form = this.wrapper.find('form')
  form.simulate('submit', { preventDefault() {} })
}
```

We are now able to rewrite the previous test in the following way:

```
test('submits the form with the page object', () => {
  const onSubmit = jest.fn()
  const wrapper = shallow(<Controlled onSubmit={onSubmit} />)

  const page = new Page(wrapper)
  page.fill('firstName', 'Christopher')
  page.fill('lastName', 'Chedeau')
  page.submit()

  expect(onSubmit).toHaveBeenCalledWith('Christopher Chedeau')
})
```

As you can see, we created an instance of the Page Object and we used its function to fill the fields and submit the form.

With the Page Object the code looks much cleaner and has no unnecessary repetitions. If something changes in the component, instead of updating multiple tests, we can just modify the way the Page Object works in a transparent and easy way.

React Dev Tools

When testing in the console is not enough, and we want to inspect our application while it is running inside the browser, we can use the React Developer Tools.

You can install them as a Chrome extension from the following URL:

https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en

The installation adds a tab to the Chrome Dev Tools called **React** where you can inspect the rendered tree of components and check which properties they have received and what their state is at a particular point in time.

Props and state can be read, and they can be changed in real time to trigger updates in the UI and see the results straightaway.

This is a must-have tool and in the most recent versions it has a new feature that can be enabled by ticking the **Trace React Updates** checkbox.

When this functionality is enabled we can use our application and visually see which components get updated when we perform a particular action. The updated components are highlighted with colored rectangles and it becomes easy to spot possible optimizations.

Error handling with React

Even if we write excellent code and we cover all the code with tests, errors will still happen. The different browsers and environments, and real user data, are all variables that we cannot control and sometimes our code will fail. As developers, that is something we must accept.

The best thing we can do when problems happen in our applications is:

- Notify the users and help them understand what happened and what they should do
- Collect all useful information about the error and the state of the application in order to reproduce it and fix bugs quickly

The way React handle errors is slightly counter-intuitive in the beginning.

Suppose you have the following components:

```
const Nice => <div>Nice</div>
```

And:

```
const Evil => (
  <div>
    Evil
    {this.does.not.exist}
  </div>
)
```

Rendering the following `App` into the DOM, we would expect different things to happen:

```
const App = () => (
  <div>
    <Nice />
    <Evil />
    <Nice />
  </div>
)
```

We may expect, for example, that the first `Nice` component gets rendered and the rendering stops because `Evil` throws an exception. Otherwise, we might expect both `Nice` components to be rendered and the `Evil` not to be shown. What really happens is that nothing is displayed on the screen.

In React, if one single component throws an exception, it stops rendering the entire tree. This is a decision made to improve safety and avoid inconsistent states.

Wouldn't it be nice if the broken component fail, in isolation, letting the rest of the tree keep on rendering? The only possible way to achieve this result would be by monkey-patching the render method and wrapping it into a `try...catch` block. That's clearly bad practice and it should be avoided; but in some cases it can be useful for debugging.

There is a library called `react-component-errors` monkey-patches all the component's methods and wraps them into a `try...catch` block so that they do not make the entire tree fail.

This approach has a downside in terms of performance and compatibility with the library, but as soon as you understand the risks you can choose to try it.

We first install the library using:

```
npm install --save react-component-errors
```

Then we import it inside your component file:

```
import wrapReactLifecycleMethods from 'react-component-errors'
```

And then we decorate your classes in the following way:

```
@wrapReactLifecycleMethods
class MyComponents extends React.Component
```

This library not only avoids breaking the entire tree when a single components fails, but it also provides a way to set a custom error handler and get some valuable information when an exception occurs.

We have to import the `config` object from the package like this:

```
import { config } from 'react-component-errors'
```

Then, we can set a custom error handler in the following way:

```
config.errorHandler = errorReport => { ... }
```

The function that we define as `errorHandler` receives an error report containing useful information to reproduce and fix the error.

The report, in fact, apart from the original error object, gives us the component name and the function name that generated the issue. It also provides all the props that the component received. All this information should be enough to write a test, reproduce the issue, and fix it quickly.

It is worth stressing that the technique used by this library should be avoided, and it could create some problems with your application. Most importantly, it should be disabled in production.

Summary

In this chapter, you learned about the benefits of testing, and the frameworks you can use to cover your React components with tests. Jest is a *fully-featured tool* while Mocha lets you *customize your*

experience.

The TestUtils let you *render your components* outside a browser and Enzyme is a powerful tool to *access the output of rendering* within the tests. We have seen how to test components using mocks and writing expectations.

We learned how Snapshot Testing can make it even easier to test the output of components and its code coverage tools helps you monitor the testing state of the codebase.

It is important to bear in mind common solutions when it comes to testing complex components such as Higher-Order Cmponents or forms with multiples nested fields.

Finally, you have learned how the React Developer Tools help debugging and how to approach error handling in React.

Previous Chapter

End of Chapter 10

Next Chapter