CHAPTER 17

Unit Testing

In this chapter, I show you how to test React components. I introduce a package that makes testing easier and demonstrate how it can be used to test components in isolation and test their interactions with their children. Table 17-1 puts unit testing in context.

Table 17-1. *Putting Unit Testing in Context*

Question	Answer
What is it?	React components require special support for testing so that their interactions with other parts of the application can be isolated and inspected.
Why is it useful?	Isolated unit tests are able to assess the basic logic provided by a component without being influenced by the interactions with the rest of the application.
How is it used?	Projects created with `create-react-app` are configured with basic test tools that are supplemented with packages that simplify the process of working with components.
Are there any pitfalls or limitations?	Effective unit testing can be difficult, and it can take time and effort to get to the point where unit tests are easily written and run and you are sure that you have isolated the correct part of the application for testing.
Are there any alternatives?	Unit testing is not a requirement and is not adopted in all projects.

DECIDING WHETHER TO UNIT TEST

Unit testing is a contentious topic. This chapter assumes you do want to do unit testing and shows you how to set up the tools and apply them to a React application. It isn't an introduction to unit testing, and I make no effort to persuade skeptical readers that unit testing is worthwhile. If would like an introduction to unit testing, then there is a good article here: https://en.wikipedia.org/wiki/Unit_testing.

I like unit testing, and I use it in my own projects—but not all of them and not as consistently as you might expect. I tend to focus on writing unit tests for features and functions that I know will be hard to write and that are likely to be the source of bugs in deployment. In these situations, unit testing helps structure my thoughts about how to best implement what I need. I find that just thinking about what I need to test helps produce ideas about potential problems, and that's before I start dealing with actual bugs and defects.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_17

That said, unit testing is a tool and not a religion, and only you know how much testing you require. If you don't find unit testing useful or if you have a different methodology that suits you better, then don't feel you need to unit test just because it is fashionable. (However, if you don't have a better methodology and you are not testing at all, then you are probably letting users find your bugs, which is rarely ideal.)

Table 17-2 summarizes the chapter.

Table 17-2. *Chapter Summary*

Problem	Solution	Listing
Perform unit tests on React components	Use Jest (or one of the other test frameworks available) along with Enzyme to create the tests	9–11
Isolate a component for testing	Test using shallow rendering	12
Test a component along with its descendants	Test using full rendering	13
Test a component's behavior	Test using the Enzyme features for working with props, state, methods, and events	14–17

Preparing for This Chapter

For this chapter, I am going to use a new project. Open a new command prompt, navigate to a convenient location, and run the command shown in Listing 17-1 to create a project called `testapp`.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-react-16`.

Listing 17-1. Creating the Example Project

```
npx create-react-app testapp
```

Run the commands shown in Listing 17-2 to navigate to the `testapp` folder to add the Bootstrap package.

Listing 17-2. Adding the Bootstrap CSS Framework

```
cd testapp
npm install bootstrap@4.1.2
```

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 17-3 to the `index.js` file, which can be found in the `testapp/src` folder.

Listing 17-3. Including Bootstrap in the index.js File in the src Folder

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import 'bootstrap/dist/css/bootstrap.css';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: http://bit.ly/CRA-PWA
serviceWorker.unregister();
```

The `create-react-app` tool creates projects that contain basic test tools, but there are some useful additions that make testing easier. Run the commands shown in Listing 17-4 in the `testapp` folder to add the testing packages to the project.

Listing 17-4. Adding Packages to the Example Project

```
npm install --save-dev enzyme@3.8.0
npm install --save-dev enzyme-adapter-react-16@1.7.1
```

Table 17-3 describes the packages that have been added to the project.

Table 17-3. *The Unit Testing Packages*

Name	Description
`enzyme`	Enzyme is a test package created by Airbnb that makes it easy to test components by exploring the content they render and examining their props and state.
`enzyme-adapter-react-16`	Enzyme requires an adapter for the specific version of React being used. This package is for the version of React used throughout this book.

Creating Components

I need some simple components to demonstrate how React applications can be unit tested. I added a file called `Result.js` to the `src` folder and used it to define the component shown in Listing 17-5.

Listing 17-5. The Contents of the Result.js File in the src Folder

```
import React from "react";

export const Result = (props) => {
    return <div className="bg-light text-dark border border-dark p-2 ">
        { props.result || 0 }
    </div>
}
```

`Result` is a simple functional component that displays the result of a calculation, received through its result prop. Next, I added a file called `ValueInput.js` to the `src` folder and used it to define the component shown in Listing 17-6.

Listing 17-6. The Contents of the ValueInput.js File in the src Folder

```
import React, { Component } from "react";

export class ValueInput extends Component {

    constructor(props) {
        super(props);
        this.state = {
            fieldValue: 0
        }
    }

    handleChange = (ev) => {
        this.setState({ fieldValue: ev.target.value },
            () => this.props.changeCallback(this.props.id, this.state.fieldValue));
    }

    render() {
        return <div className="form-group p-2">
            <label>Value #{this.props.id}</label>
            <input className="form-control"
                value={ this.state.fieldValue}
                onChange={ this.handleChange } />
        </div>
    }
}
```

This is a stateful component that renders an input element and invokes a callback function when there is a change. Listing 17-7 shows the changes I made to the `App` component to remove the placeholder content and use the new components.

Listing 17-7. Completing the Example Application in the App.js File in the src Folder

```
import React, { Component } from "react";
import { ValueInput } from "./ValueInput";
import { Result } from "./Result";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            title: this.props.title || "Simple Addition" ,
            fieldValues: [],
            total: 0
        }
    }

    updateFieldValue = (id, value) => {
        this.setState(state => {
            state.fieldValues[id] = Number(value);
            return state;
        });
    }

    updateTotal = () => {
        this.setState(state => ({
            total: state.fieldValues.reduce((total, val) => total += val, 0)
        }))
    }

    render() {
        return <div className="m-2">
                    <h5 className="bg-primary text-white text-center p-2">
                        { this.state.title }
                    </h5>
                    <Result result={ this.state.total } />
                    <ValueInput id="1" changeCallback={ this.updateFieldValue } />
                    <ValueInput id="2" changeCallback={ this.updateFieldValue } />
                    <ValueInput id="3" changeCallback={ this.updateFieldValue } />
                    <div className="text-center">
                    <button className="btn btn-primary" onClick={ this.updateTotal}>
                        Total
                    </button>
            </div>
        </div>
    }
}
```

App creates three `ValueInput` components and configures them so that the values the user enters are stored in the `fieldValues` state array. A button is configured so that a click event invokes the `updateTotal` method, which sums the values from the `ValueInput` components and updates a state data value that is displayed by the `Result` component.

Running the Example Application

Use the command prompt to navigate to the `testapp` folder and run the command shown in Listing 17-8 to start the React developer tools.

Listing 17-8. Starting the Development Tools

```
npm start
```

A new browser window will open, and you will see the example application, as shown in Figure 17-1. Enter numeric values into the fields, and click the Total button to display a result.

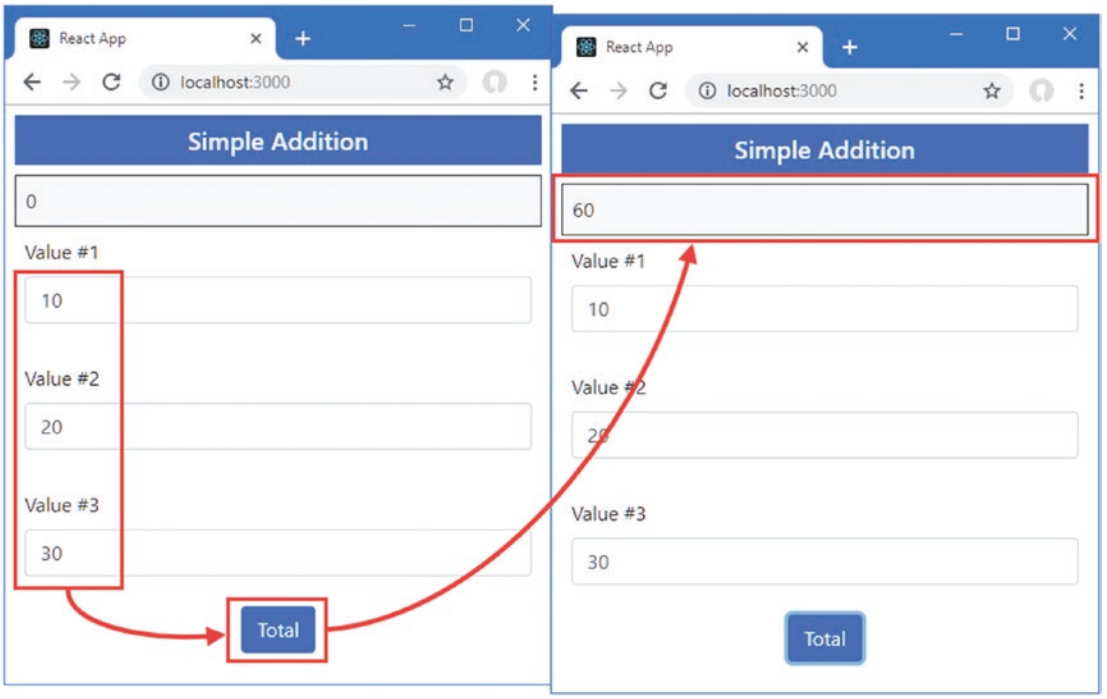

Figure 17-1. *Running the example application*

Running the Placeholder Unit Test

Projects created with `create-react-app` contain the Jest test runner, which is a tool that executes unit tests and reports the results. As part of the project setup process, a file called `App.test.js` is created, which contains the following code:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
```

```
it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
  ReactDOM.unmountComponentAtNode(div);
});
```

This is a basic unit test, which is encapsulated in the `it` function. The first argument to the function is a description of the test. The second argument is the test itself, which is a function that performs some work. In this case, the unit test renders the `App` component into a `div` element and then unmounts it. Open a new command prompt, navigate to the `testapp` folder, and run the command shown in Listing 17-9 to perform the unit test. (The test tools are designed so that you can have them running alongside the development tools.)

Listing 17-9. Running a Unit Test

```
npm run test
```

This command locates all the tests defined in the project and executes them. There is only one test at the moment, which produces the following results:

```
...
PASS  src/App.test.js
  √ renders without crashing (24ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.077s
Ran all test suites related to changed files.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
...
```

After the tests have been run, the testing tool enters watch mode. When a file changes, tests are located and executed, and the results displayed again. To see what happens when a unit test fails, add the statement shown in Listing 17-10 to the `render` method of the `App` component.

Listing 17-10. Making a Test Fail in the App.js File in the src Folder

```
...
render() {
    throw new Error("something went wrong");
    return <div className="m-2">
                <h5 className="bg-primary text-white text-center p-2">
                    { this.state.title }
                </h5>
```

```
                <Result result={ this.state.total } />
                <ValueInput id="1" changeCallback={ this.updateFieldValue } />
                <ValueInput id="2" changeCallback={ this.updateFieldValue } />
                <ValueInput id="3" changeCallback={ this.updateFieldValue } />
                <div className="text-center">
                <button className="btn btn-primary" onClick={ this.updateTotal}>
                    Total
                </button>
        </div>
    </div>
}
...
```

An error will be thrown when the render method is invoked, which is the behavior that the unit test is looking out for. When you save the change, the unit test will be performed again, but this time it will fail, giving you details of the problem that was detected.

```
...
renders without crashing

   something went wrong

     27 |
     28 |     render() {
   > 29 |         throw new Error("something went wrong");
        |               ^
     30 |         return <div className="m-2">
     31 |                     <h5 className="bg-primary text-white text-center p-2">
     32 |                         Simple Addition
...
```

The error that is thrown by the component bubbles up to the `it` function in the unit test and is treated as a test failure. To restore the application to its working state, comment out the `throw` statement from the App component, as shown in Listing 17-11.

Listing 17-11. Removing the throw Statement in the App.js File in the src Folder

```
...
render() {
    //throw new Error("something went wrong");
    return <div className="m-2">
                <h5 className="bg-primary text-white text-center p-2">
                    { this.state.title }
                </h5>
                <Result result={ this.state.total } />
                <ValueInput id="1" changeCallback={ this.updateFieldValue } />
                <ValueInput id="2" changeCallback={ this.updateFieldValue } />
                <ValueInput id="3" changeCallback={ this.updateFieldValue } />
                <div className="text-center">
```

```
                <button className="btn btn-primary" onClick={ this.updateTotal}>
                    Total
                </button>
            </div>
        </div>
}
...
```

When you save the change, the test will run again and will pass this time.

Testing a Component Using Shallow Rendering

Shallow rendering isolates a component from its children, allowing it to be tested on its own. It is an effective technique for testing the basic functions of a component without the effects caused by interaction with its content. To test the App component using shallow rendering, I added a file called `appContent.test.js` to the `src` folder and added the code shown in Listing 17-12.

■ **Tip** Jest will find tests in files whose name ends with `test.js` or `spec.js` or any file in a folder named `__tests__` (two underscores before and after `tests`).

Listing 17-12. The Contents of the appContent.test.js File in the src Folder

```
import React from "react";
import Adapter from 'enzyme-adapter-react-16';
import Enzyme, { shallow } from "enzyme";
import App from "./App";
import { ValueInput } from "./ValueInput";

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

it("Renders three ValueInputs", () => {
    const wrapper = shallow(<App />);
    const valCount = wrapper.find(ValueInput).length;
    expect(valCount).toBe(3)
});
```

This is the first real unit test in this chapter, so I will explain each part and show you how they fit together.

The first statement configures the Enzyme package and applies the adapter that allows Enzyme to work with the correct version of React.

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

The `Enzyme.configure` method is passed a configuration object whose `adapter` property is assigned the imported contents of the adapter package. If you need to test a different version of React, you can see the list of adapters available at https://airbnb.io/enzyme.

The next step is the definition of the unit test. The `it` method doesn't need to be imported because it is defined globally by the Jest test package.

```
...
it("Renders three ValueInputs", () => {
...
```

The first argument should be a meaningful description of what the test aims to establish. In this case, the test checks that `App` renders three `ValueInput` components.

The next statement sets up the component, which is done using the shallow function imported from the `enzyme` package.

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

The `shallow` function accepts the component element. A component is instantiated and is put through the lifecycle described in Chapter 13, and its contents are rendered. But, since this is shallow rendering, the child components are not used to rendered, leaving their elements in place in the output from the `App` component. That means that the `App` component's props and state data are used when rendering the content, but the child components are not processed, producing a result like this:

```
...
<div className="m-2">
    <h5 className="bg-primary text-white text-center p-2">
        Simple Addition
    </h5>
    <Result result={0} />
    <ValueInput id="1" changeCallback={[Function]} />
    <ValueInput id="2" changeCallback={[Function]} />
    <ValueInput id="3" changeCallback={[Function]} />
    <div className="text-center">
        <button className="btn btn-primary" onClick={[Function]}>
            Total
        </button>
    </div>
</div>
...
```

The output is presented in a wrapper object that can be inspected for testing. The Enzyme package provides a set of methods that can be used to inspect the content rendered from the DOM, modeled on the API provided by the popular jQuery DOM manipulation package. The most useful methods are described in Table 17-4, and the full set of features is described at https://airbnb.io/enzyme.

Table 17-4. *Useful Enzyme Methods for Inspecting Component Content*

Name	Description
`find(selector)`	This method finds all elements matched by the CSS selector, which will match element types, attributes, and classes.
`findWhere(predicate)`	This method finds all elements that are matched by the specified predicate.
`first(selector)`	Returns the first element that is matched by the selector. If the selector is omitted, then the first element of any type will be returned.
`children()`	Creates a new selection containing the children of the current element.
`hasClass(class)`	This method returns true if an element is a member of a specified class.
`text()`	This method returns the text content from an element.
`html()`	This method returns the deep rendered content from the component so that all of the descendant components are processed.
`debug()`	This method returns the shallow rendered content from the component.

These methods can be used to navigate through the content rendered by a component and to inspect the contents. The test in Listing 17-12 uses the `find` selector to select all the `ValueInput` elements rendered by the `App` component and uses the `length` property on the result to determine how many elements have been found.

```
...
const valCount = wrapper.find(ValueInput).length;
...
```

The final step in the test is to compare the result with the expected outcome, which is done using the global `expect` function provided by Jest.

```
...
expect(valCount).toBe(3)
...
```

The result of a test is passed to the `expect` function, and then a matcher method is invoked on the result. Jest supports an extensive array of matches, described at `https://jestjs.io/docs/en/expect`, and the most useful are shown in Table 17-5.

Table 17-5. *Useful Expect Matchers*

Name	Description
`toBe(value)`	This method asserts that a result is the same as the specified value (but need not be the same object).
`toEqual(object)`	This method asserts that a result is the same object as the specified value.
`toMatch(regexp)`	This method asserts that a result matches the specified regular expression.
`toBeDefined()`	This method asserts that the result has been defined.
`toBeUndefined()`	This method asserts that the result has not been defined.
`toBeNull()`	This method asserts that the result is null.
`toBeTruthy()`	This method asserts that the result is truthy.
`toBeFalsy()`	This method asserts that the result is falsy.
`toContain(substring)`	This method asserts that the result contains the specified substring.
`toBeLessThan(value)`	This method asserts that the result is less than the specified value.
`toBeGreaterThan(value)`	This method asserts that the result is more than the specified value.

Jest keeps track of which matches fail and reports the outcome when all the tests in the project have been run. The matcher in Listing 17-12 checks that there are three `ValueInput` components in the content rendered by `App`.

Jest runs the test in Listing 17-12 as soon as the file is saved, which produces the following results:

```
...
PASS  src/App.test.js
 PASS  src/App.shallow.test.js

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        2.672s
Ran all test suites.

Watch Usage: Press w to show more.
...
```

There are now two tests in the project, and both of them are run. You can leave the tests to run automatically, or you can run one or more tests on demand using the options that are shown when the W key is pressed.

Testing a Component with Full Rendering

Full rendering processes all the descendent components. The descendent component elements are left in the rendered content, which means that the App component will produce the following content when it is fully rendered:

```
...
<App>
<div className="m-2">
  <h5 className="bg-primary text-white text-center p-2">
    Simple Addition
  </h5>
  <Result result={0}>
    <div className="bg-light text-dark border border-dark p-2 ">0</div>
  </Result>
  <ValueInput id="1" changeCallback={[Function]}>
    <div className="form-group p-2">
      <label>Value #1</label>
      <input className="form-control" value={0} onChange={[Function]} />
    </div>
  </ValueInput>
  <ValueInput id="2" changeCallback={[Function]}>
    <div className="form-group p-2">
      <label>Value #2</label>
      <input className="form-control" value={0} onChange={[Function]} />
    </div>
  </ValueInput>
  <ValueInput id="3" changeCallback={[Function]}>
    <div className="form-group p-2">
      <label>Value #3</label>
      <input className="form-control" value={0} onChange={[Function]} />
    </div>
  </ValueInput>
  <div className="text-center">
    <button className="btn btn-primary" onClick={[Function]}>Total</button>
  </div>
</div>
</App>
...
```

Full rendering is performed with the mount method, as shown in Listing 17-13.

Listing 17-13. Fully Rendering a Component in the appContent.test.js File in the src Folder

```
import React from "react";
import Adapter from 'enzyme-adapter-react-16';
import Enzyme, { shallow, mount } from "enzyme";
import App from "./App";
import { ValueInput } from "./ValueInput";

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

```
it("Renders three ValueInputs", () => {
    const wrapper = shallow(<App />);
    const valCount = wrapper.find(ValueInput).length;
    expect(valCount).toBe(3)
});
```

```
it("Fully renders three inputs", () => {
    const wrapper = mount(<App title="tester" />);
    const count = wrapper.find("input.form-control").length
    expect(count).toBe(3);
});

it("Shallow renders zero inputs", () => {
    const wrapper = shallow(<App />);
    const count = wrapper.find("input.form-control").length
    expect(count).toBe(0);
})
```

The first new test uses the Enzyme `mount` function to fully render `App` and its descendants. The wrapper returned by `mount` supports the methods described in Table 17-5, and the full set of features is described at https://airbnb.io/enzyme/docs/api/mount.html. I use the find method to locate `input` elements that have been assigned to the `form-control` class and use `expect` to make sure that there are three of them. The second new test locates the same elements but does so using shallow rendering and checks that there are no `input` elements in the content.

When the changes to the file are saved, the tests will be run and produce the following results:

```
...
PASS  src/App.test.js
PASS  src/appContent.test.js

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        3.109s
Ran all test suites.

Watch Usage: Press w to show more.
...
```

Testing with Props, State, Methods, and Events

The content that a component renders can change in response to user input or updates in the application state. To help test the behavior of a component, Enzyme provides the methods described in Table 17-6.

Table 17-6. *Enzyme Methods for Testing Behavior*

Name	Description
`instance()`	This method returns the component object so that its methods can be invoked.
`prop(key)`	This method returns the value of the specified prop.
`props()`	This method returns all of the component's props.
`setProps(props)`	This method is used to specify new props, which are merged with the component's existing props before it is updated.
`state(key)`	This method is used to get a specified state value. If no value is specified, then all of the component's state data is returned.
`setState(state)`	This method changes the component's state data and then re-renders the component.
`simulate(event, args)`	This method dispatches an event to the component.
`update()`	This method forces the component to re-render its content.

The simplest test of behavior is to ensure that a component reflects its props. I created a file called `appBehavior.test.js` in the `src` folder and used it to define the test shown in Listing 17-14.

Listing 17-14. Testing a Prop in the appBehavior.test.js File in the src Folder

```
import React from "react";
import Adapter from 'enzyme-adapter-react-16';
import Enzyme, { shallow } from "enzyme";
import App from "./App";

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

it("uses title prop", () => {

    const titleVal = "test title"
    const wrapper = shallow(<App title={ titleVal } />);

    const firstTitle = wrapper.find("h5").text();
    const stateValue = wrapper.state("title");

    expect(firstTitle).toBe(titleVal);
    expect(stateValue).toBe(titleVal);
});
```

The App component is configured with a `title` prop when it is passed to the `shallow` method. The test checks that the prop is used to override the default value by locating the `h5` element and getting its text content and also by reading the value of the `title` state property. The test passes only if both the contents of the `h5` element and the `state` property are the same as the value of the `title` prop.

Testing the Effect of Methods

The `instance` method is used to obtain the component object, which can then be used to invoke its methods. In Listing 17-15, I have defined a test that invokes the `updateField` and `updateTotal` methods and checks the effect on the component's state data.

Listing 17-15. Invoking Methods in the appBehavior.test.js File in the src Folder

```
import React from "react";
import Adapter from 'enzyme-adapter-react-16';
import Enzyme, { shallow } from "enzyme";
import App from "./App";

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

it("uses title prop", () => {

    const titleVal = "test title"
    const wrapper = shallow(<App title={ titleVal } />);

    const firstTitle = wrapper.find("h5").text();
    const stateValue = wrapper.state("title");

    expect(firstTitle).toBe(titleVal);
    expect(stateValue).toBe(titleVal);
});

it("updates state data", () => {
    const wrapper = shallow(<App />);
    const values = [10, 20, 30];

    values.forEach((val, index) =>
        wrapper.instance().updateFieldValue(index + 1, val));
    wrapper.instance().updateTotal();

    expect(wrapper.state("total"))
        .toBe(values.reduce((total, val) => total + val), 0);
});
```

The new test shallow renders an `App` component and then calls the `updateFieldValue` method with an array of values before then invoking the `updateTotal` method. The `state` method is used to get the value of the total `state` property, which is compared to the sum of the values passed to the `updateFieldValue` method.

Testing the Effects of an Event

The `simulate` method is used to send an event to the component's event handlers. Care must be taken with this type of test because it is easy to end up testing React's ability to dispatch events rather than the component's ability to handle them. In most cases, it is more useful to invoke the methods that will be executed in response to an event. Listing 17-16 locates the `button` element rendered by the `App` component and triggers a `click` event in order to ensure that it leads to the total being calculated.

Listing 17-16. Simulating an Event in the appBehavior.test.js File in the src Folder

```
import React from "react";
import Adapter from 'enzyme-adapter-react-16';
import Enzyme, { shallow } from "enzyme";
import App from "./App";

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

it("uses title prop", () => {

    const titleVal = "test title"
    const wrapper = shallow(<App title={ titleVal } />);

    const firstTitle = wrapper.find("h5").text();
    const stateValue = wrapper.state("title");

    expect(firstTitle).toBe(titleVal);
    expect(stateValue).toBe(titleVal);
});

it("updates state data", () => {
    const wrapper = shallow(<App />);
    const values = [10, 20, 30];

    values.forEach((val, index) =>
        wrapper.instance().updateFieldValue(index + 1, val));
    wrapper.instance().updateTotal();

    expect(wrapper.state("total"))
        .toBe(values.reduce((total, val) => total + val), 0);
})

it("updates total when button is clicked", () => {
    const wrapper = shallow(<App />);
    const button = wrapper.find("button").first();

    const values = [10, 20, 30];
    values.forEach((val, index) =>
        wrapper.instance().updateFieldValue(index + 1, val));

    button.simulate("click")

    expect(wrapper.state("total"))
        .toBe(values.reduce((total, val) => total + val), 0);
})
```

The new test simulates the `click` event, the handler for which invokes the component's `updateTotal` method. To ensure that the event has been handled, the value of the `total` state data property is read.

Testing the Interaction Between Components

The ability to navigate the content rendered by a component can be combined with the methods described in Table 17-6 to test the interaction between components, as shown in Listing 17-17.

Listing 17-17. Testing Component Interaction in the appBehavior.test.js File in the src Folder

```
import React from "react";
import Adapter from 'enzyme-adapter-react-16';
import Enzyme, { shallow, mount } from "enzyme";
import App from "./App";
import { ValueInput } from "./ValueInput";

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

it("uses title prop", () => {

    const titleVal = "test title"
    const wrapper = shallow(<App title={ titleVal } />);

    const firstTitle = wrapper.find("h5").text();
    const stateValue = wrapper.state("title");

    expect(firstTitle).toBe(titleVal);
    expect(stateValue).toBe(titleVal);
});

it("updates state data", () => {
    const wrapper = shallow(<App />);
    const values = [10, 20, 30];

    values.forEach((val, index) =>
        wrapper.instance().updateFieldValue(index + 1, val));
    wrapper.instance().updateTotal();

    expect(wrapper.state("total"))
        .toBe(values.reduce((total, val) => total + val), 0);
})

it("updates total when button is clicked", () => {
    const wrapper = shallow(<App />);
    const button = wrapper.find("button").first();

    const values = [10, 20, 30];
    values.forEach((val, index) =>
        wrapper.instance().updateFieldValue(index + 1, val));

    button.simulate("click")

    expect(wrapper.state("total"))
        .toBe(values.reduce((total, val) => total + val), 0);
})
```

```
it("child function prop updates state", () => {
    const wrapper = mount(<App />);
    const valInput = wrapper.find(ValueInput).first();
    const inputElem = valInput.find("input").first();

    inputElem.simulate("change", { target: { value: "100"}});
    wrapper.instance().updateTotal();

    expect(valInput.state("fieldValue")).toBe("100");
    expect(wrapper.state("total")).toBe(100);
})
```

The new test locates the input element rendered by the first `ValueInput` and triggers its change event, supplying an argument that will provide the component's handler with the values it needs. The `instance` method is used to invoke the `updateTotal` method of the `App` component, and the `state` method is used to check that the state data for both the `App` and `ValueInput` components has been updated correctly.

Summary

In this chapter, I showed you how to perform unit tests on React components. I showed you how to run tests using Jest and how to perform those tests with shallow and full rendering, provided by the Enzyme package. I explained how to examine the content rendered by a component, how to invoke its methods, how to explore its state, and how to manage its props. Together, these features allow a component to be tested in isolation and in combination with its children. In the next part of the book, I describe how to supplement the core React features to create complete web applications.