CHAPTER 16

Using Refs and Portals

Under normal circumstances, a component doesn't interact directly with the elements in the Document Object Model (DOM). Normal interaction is through props and event handlers, which make it possible to compose applications and for components to work together without knowledge of the content they deal with.

There are some situations where components need to interact with the elements in the DOM, and React provides two features for this purpose. The *refs* feature—short for references—provides access to the HTML elements rendered by a component after they have been added to the DOM. The *portals* feature provides access to HTML elements outside of the application's content.

These features should be used with caution because they undermine the isolation between components in an application, which makes it harder to write, test, and maintain. These features lead to "rabbit holing," where they fix one problem but introduce another, which leads to another fix and another problem and so on. If used injudiciously, these features produce components that duplicate the core functionality provided by React, which is rarely a beneficial result. Table 16-1 puts refs and portals in context.

Table 16-1. *Putting Refs and Portals in Context*

Question	Answer
What are they?	Refs are references to the elements in the DOM that have been rendered by a component. A portal allows content to be rendered outside of the application's content.
Why are they useful?	There are some features of HTML elements that cannot be easily managed without accessing the DOM directly, such as focusing an element. These features are also useful for integration with other frameworks and libraries.
How are they used?	Refs are created using the special `ref` attribute and can be created using the `React.createRef` method or using a callback function. Portals are created using the `ReactDOM.createPortal` method.
Are there any pitfalls or limitations?	These features are prone to misuse, such that they undermine component isolation and are used to duplicate features that are provided by React.
Are there any alternatives?	Refs and portals are advanced features that will not be required in many projects.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_16

Table 16-2 summarizes the chapter.

Table 16-2. *Chapter Summary*

Problem	Solution	Listing
Access the HTML element objects created for a component	Use a ref	1–9, 11, 12, 18, 19
Use a form element without using state data and an event handler	Use uncontrolled form components	10, 13–15
Prevent data loss during updates	Use the `getSnapshotBeforeUpdate` method	16, 17
Access a child component's content	Use the refs prop or ref forwarding	20–23
Project content into a specific DOM element	Use a portal	24–26

Preparing for This Chapter

To create the example project for this chapter, open a new command prompt, navigate to a convenient location, and run the command shown in Listing 16-1.

■ **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 16-1. Creating the Example Project

```
npx create-react-app refs
```

Run the commands shown in Listing 16-2 to navigate to the `refs` folder to add the Bootstrap package.

Listing 16-2. Adding the Bootstrap CSS Framework

```
cd refs
npm install bootstrap@4.1.2
```

In this chapter, I create an example that relies on jQuery. Run the command shown in Listing 16-3 in the `refs` folder to add the jQuery package to the project.

Listing 16-3. Installing jQuery

```
npm install jquery@3.3.1
```

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 16-4 to the `index.js` file, which can be found in the `src` folder.

Listing 16-4. 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();
```

Add a file called `Editor.js` file in the `src` folder and add the code shown in Listing 16-5.

Listing 16-5. The Contents of the Editor.js File in the src Folder

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

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            name: "",
            category: "",
            price: ""
        }
    }

    handleChange = (event) => {
        event.persist();
        this.setState(state => state[event.target.name] = event.target.value);
    }

    handleAdd = () => {
        this.props.callback(this.state);
        this.setState({ name: "", category:"", price:""});
    }

    render() {
        return <React.Fragment>
            <div className="form-group p-2">
                <label>Name</label>
                <input className="form-control" name="name"
                    value={ this.state.name } onChange={ this.handleChange }
                    autoFocus={ true } />
            </div>
            <div className="form-group p-2">
```

```
                <label>Category</label>
                <input className="form-control" name="category"
                    value={ this.state.category } onChange={ this.handleChange }  />
            </div>
            <div className="form-group p-2">
                <label>Price</label>
                <input className="form-control" name="price"
                    value={ this.state.price } onChange={ this.handleChange }  />
            </div>
            <div className="text-center">
                <button className="btn btn-primary" onClick={ this.handleAdd }>
                    Add
                </button>
            </div>
        </React.Fragment>
    }

}
```

The `Editor` component renders a series of `input` elements whose value are set using state data properties and whose change events are handled by the `handleChange` method. There is a `button` element whose click event invokes the `handleAdd` method, which invokes a function prop using the state data, which is then reset.

Next, add a file called `ProductTable.js` to the `src` folder and add the code shown in Listing 16-6.

Listing 16-6. The Contents of the ProductTable.js File in the src Folder

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

export class ProductTable extends Component {

    render() {
        return <table className="table table-sm table-striped">
            <thead><tr><th>Name</th><th>Category</th><th>Price</th></tr></thead>
            <tbody>
                {
                    this.props.products.map(p =>
                        <tr key={ p.name }>
                            <td>{ p.name }</td>
                            <td>{ p.category }</td>
                            <td>${ Number(p.price).toFixed(2) }</td>
                        </tr>
                    )
                }
            </tbody>
        </table>
    }
}
```

The `ProductTable` component renders a table that contains a row for each object received in the `products` prop. Next, replace the contents of the `App.js` file with the code shown in Listing 16-7.

Listing 16-7. Replacing the Contents of the App.js File in the src Folder

```
import React, { Component } from "react";
import { Editor } from "./Editor"
import { ProductTable } from "./ProductTable";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            products: []
        }
    }

    addProduct = (product) => {
        if (this.state.products.indexOf(product.name) === -1) {
            this.setState({ products: [...this.state.products, product ]});
        }
    }

    render() {
        return <div>
            <Editor callback={ this.addProduct } />
            <h6 className="bg-secondary text-white m-2 p-2">Products</h6>
                <div className="m-2">
                    {
                        this.state.products.length === 0
                            ? <div className="text-center">No Products</div>
                                : <ProductTable products={ this.state.products } />
                    }
                </div>
        </div>
    }
}
```

Using the command prompt, run the command shown in Listing 16-8 in the `refs` folder to start the development tools.

Listing 16-8. Starting the Development Tools

```
npm start
```

Once the initial preparation for the project is complete, a new browser window will open and display the URL `http://localhost:3000`, which displays the content shown in Figure 16-1. Fill out the form and click the Add button, and you will see a new entry displayed in the table.

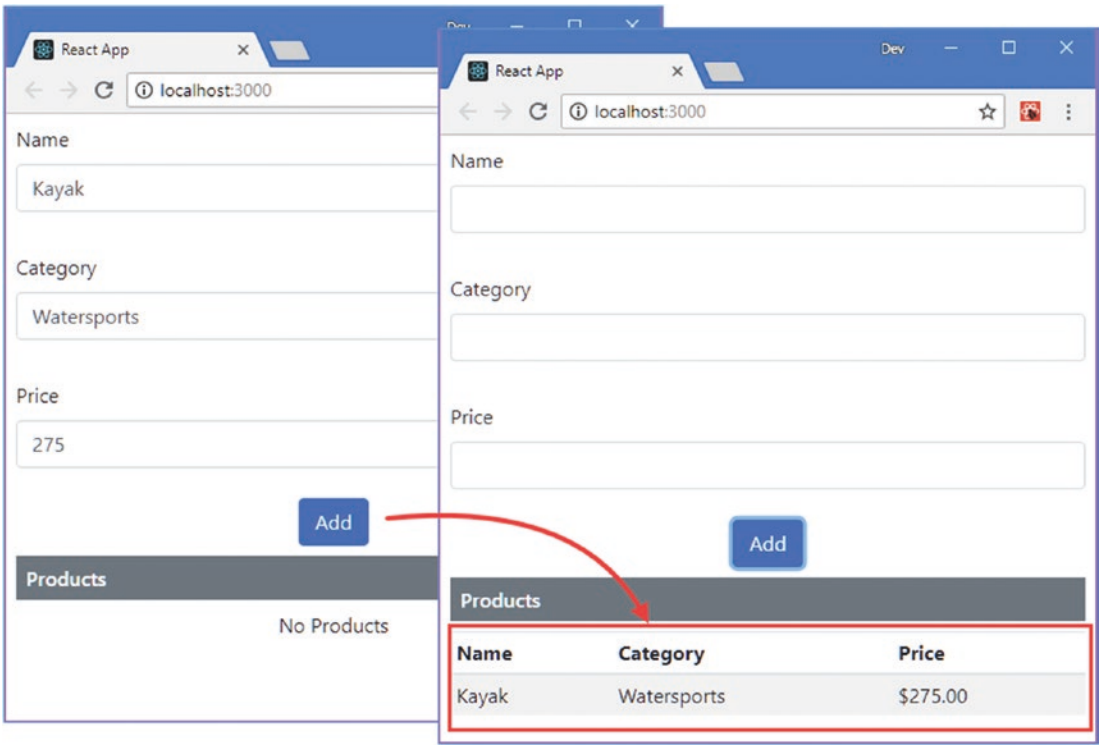

Figure 16-1. *Running the example application*

Creating Refs

Refs can be used when a component needs to access the DOM in order to use features of a specific HTML element. There are HTML features that cannot be achieved through the use of props, one of which is to ask an element to gain focus. The `autoFocus` attribute can be used to focus an element when content is first rendered, but the focus will switch to the `button` element once the user clicks it, which means that the user can't start typing to create another item until they refocus, either by clicking the `input` element or by using the Tab key.

A ref can be used to access the DOM and invoke the `focus` method on the `input` element when the event triggered by clicking the Add button is handled, as shown in Listing 16-9.

DON'T RUSH TO USE REFS

Being able to access the DOM is a natural expectation for web developers, and refs can seem like a feature that makes React development easier, especially if you are coming to React from a framework like Angular.

It is easy to get carried away with refs and end up with a component that duplicates the content handling features that should be performed by React. A component that makes excessive use of refs is difficult to manage, can create dependencies on specific browser features, and can be difficult to run on different platforms.

Use refs only as a last resort, and always consider if you can achieve the same result using the state and props features.

Listing 16-9. Using a Ref in the Editor.js File in the src Folder

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

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            name: "",
            category: "",
            price: ""
        }
        this.nameRef = React.createRef();
    }

    handleChange = (event) => {
        event.persist();
        this.setState(state => state[event.target.name] = event.target.value);
    }

    handleAdd = () => {
        this.props.callback(this.state);
        this.setState({ name: "", category:"", price:""},
            () => this.nameRef.current.focus());
    }

    render() {
        return <React.Fragment>
            <div className="form-group p-2">
                <label>Name</label>
                <input className="form-control" name="name"
                    value={ this.state.name } onChange={ this.handleChange }
                    autoFocus={ true } ref={ this.nameRef } />
            </div>
            <div className="form-group p-2">
                <label>Category</label>
                <input className="form-control" name="category"
                    value={ this.state.category } onChange={ this.handleChange }  />
            </div>
            <div className="form-group p-2">
                <label>Price</label>
                <input className="form-control" name="price"
                    value={ this.state.price } onChange={ this.handleChange }  />
            </div>
```

```
            <div className="text-center">
                <button className="btn btn-primary" onClick={ this.handleAdd }>
                    Add
                </button>
            </div>
        </React.Fragment>
    }
}
```

Refs are created using the `React.createRef` method, which is invoked in the constructor so that the result can be used throughout the component. A ref is associated with an element using the special `ref` prop, with an expression that selects the ref for the element.

```
...
<input className="form-control" name="name"
    value={ this.state.name } onChange={ this.handleChange }
    autoFocus={ true } ref={ this.nameRef } />
...
```

The ref object returned by the `createRef` method defines just one property, named `current`, that returns the `HTMLElement` object that represents the element in the DOM. I use the `current` property in the `handleAdd` method to invoke the `focus` method after the state data update has been completed, like this:

```
...
this.setState({ name: "", category:"", price:""},
    () => this.nameRef.current.focus());
...
```

The result is that the name `input` element will regain the focus when the update triggered by the Add button is complete, allowing the user to start typing for the next new product without having to manually select the element, as shown in Figure 16-2.

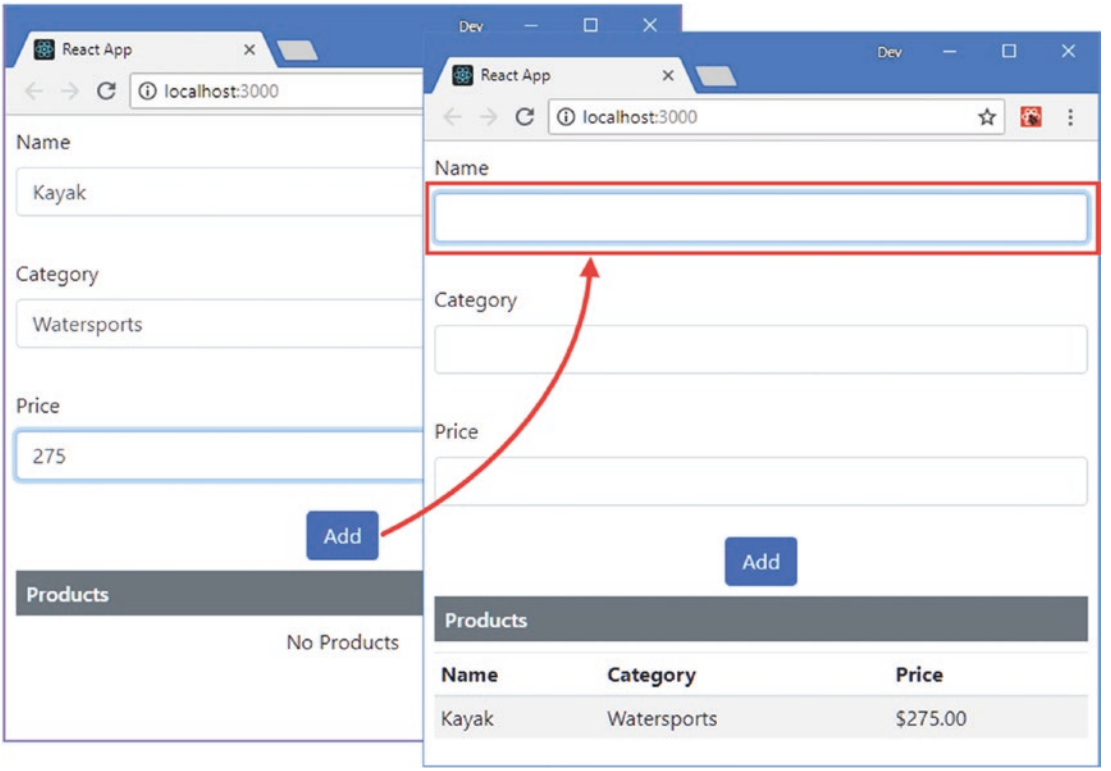

Figure 16-2. *Using a ref*

Using Refs to Create Uncontrolled Form Components

The example application uses the controlled form components technique that I introduced in Chapter 15, where React is responsible for the contents of each form element, using a state data property to store its value and an event handler to respond to changes.

Form elements already have the ability to store a value and respond to changes, but these features are not used by a controlled form component. An alternative technique is to create an *uncontrolled form component*, where a ref is used to access form elements and the browser is responsible for managing the element's value and responding to changes. In Listing 16-10, I have removed the state data used to manage the `input` elements rendered by the `Editor` component and used refs to create uncontrolled form components.

Listing 16-10. Creating Uncontrolled Form Components in the Editor.js File in the src Folder

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

export class Editor extends Component {

    constructor(props) {
        super(props);
        // this.state = {
```

```
        //     name: "",
        //     category: "",
        //     price: ""
        // }
        this.nameRef = React.createRef();
        this.categoryRef = React.createRef();
        this.priceRef = React.createRef();
    }

    // handleChange = (event) => {
    //     event.persist();
    //     this.setState(state => state[event.target.name] = event.target.value);
    // }

    handleAdd = () => {
        this.props.callback({
            name: this.nameRef.current.value,
            category: this.categoryRef.current.value,
            price: this.priceRef.current.value
        });
        this.nameRef.current.value = "";
        this.categoryRef.current.value = "";
        this.priceRef.current.value = "";
        this.nameRef.current.focus();
    }

    render() {
        return <React.Fragment>
            <div className="form-group p-2">
                <label>Name</label>
                <input className="form-control" name="name"
                    autoFocus={ true } ref={ this.nameRef } />
            </div>
            <div className="form-group p-2">
                <label>Category</label>
                <input className="form-control" name="category"
                    ref={ this.categoryRef } />
            </div>
            <div className="form-group p-2">
                <label>Price</label>
                <input className="form-control" name="price" ref={ this.priceRef } />
            </div>
            <div className="text-center">
                <button className="btn btn-primary" onClick={ this.handleAdd }>
                    Add
                </button>
            </div>
        </React.Fragment>
    }
}
```

The input elements values are not required until the user clicks the Add button. In the handleAdd method, which is invoked when the button is clicked, the refs for each of the input elements is used to read the value property. The result has the same appearance to the user as earlier examples, but behind the scenes, React is no longer responsible for managing the element values or responding to change events.

SETTING AN INITIAL VALUE FOR AN UNCONTROLLED ELEMENT

React isn't responsible for uncontrolled elements, but it can still provide an initial value, which is then managed by the browser. To set the value, use the defaultValue or defaultChecked attribute, but bear in mind that the value you specify will be used only when the element is first rendered and won't update the element when it is changed.

Creating Refs Using a Callback Function

The previous example shows how refs can be used in form elements, but the result isn't that different from the controlled form component with which I started the chapter. There is an alternative technique that can be used to create refs and that can produce more concise components, as shown in Listing 16-11, known as *callback refs*.

Listing 16-11. Using Callback Refs in the Editor.js File in the src Folder

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

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.formElements = {
            name: { },
            category: { },
            price: { }
        }
    }

    setElement = (element) => {
        if (element !== null) {
            this.formElements[element.name].element = element;
        }
    }

    handleAdd = () => {
        let data = {};
        Object.values(this.formElements)
            .forEach(v => {
                data[v.element.name] = v.element.value;
                v.element.value = "";
            });
```

```
        this.props.callback(data);
        this.formElements.name.element.focus();
    }

    render() {
        return <React.Fragment>
            <div className="form-group p-2">
                <label>Name</label>
                <input className="form-control" name="name"
                    autoFocus={ true } ref={ this.setElement } />
            </div>
            <div className="form-group p-2">
                <label>Category</label>
                <input className="form-control" name="category"
                    ref={ this.setElement } />
            </div>
            <div className="form-group p-2">
                <label>Price</label>
                <input className="form-control" name="price"
                    ref={ this.setElement } />
            </div>
            <div className="text-center">
                <button className="btn btn-primary" onClick={ this.handleAdd }>
                    Add
                </button>
            </div>
        </React.Fragment>
    }
}
```

The value of the `ref` property of the `input` elements is set to a method, which is invoked when the content is rendered. Instead of dealing with a `ref` object, the specified method receives the `HTMLElement` object directly, instead of a reference object with a `current` property. In the listing, the `setElement` method receives the elements, which are added to the `formElements` object using the `name` value so that I can differentiate between the elements.

The function you provide for a callback ref will also be invoked with `null` as the argument if the element is unmounted. For this example, I don't need to do any tidying up if the elements are removed, so I just check for the null value in the `setElement` method.

```
...
setElement = (element) => {
    if (element !== null) {
        this.formElements[element.name].element = element;
    }
}
...
```

Once you have the function for the refs in place, forms can be easily generated programmatically, as shown in Listing 16-12, because refs don't have be created and assigned to elements individually.

Listing 16-12. Generating a Form Programmatically in the Editor.js File in the src Folder

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

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.formElements = {
            name: { label: "Name", name: "name" },
            category: { label: "Category", name: "category" },
            price: { label: "Price", name: "price" }
        }
    }

    setElement = (element) => {
        if (element !== null) {
            this.formElements[element.name].element = element;
        }
    }

    handleAdd = () => {
        let data = {};
        Object.values(this.formElements)
            .forEach(v => {
                data[v.element.name] = v.element.value;
                v.element.value = "";
            });
        this.props.callback(data);
        this.formElements.name.element.focus();
    }

    render() {
        return <React.Fragment>
             {
                Object.values(this.formElements).map(elem =>
                    <div className="form-group p-2" key={ elem.name }>
                        <label>{ elem.label }</label>
                        <input className="form-control"
                            name={ elem.name }
                            autoFocus={ elem.name === "name" }
                            ref={ this.setElement }  />
                    </div>)
            }
            <div className="text-center">
                <button className="btn btn-primary" onClick={ this.handleAdd }>
                    Add
                </button>
            </div>
        </React.Fragment>
    }
}
```

The input elements are generated using the properties of the formElements object, where each property is assigned an object with label and name properties that are used in the render method to configure the element.

The code required to define and manage the form is more concise, but the effect is the same, and filling the form and clicking the Add button displays a new object, as shown in Figure 16-3.

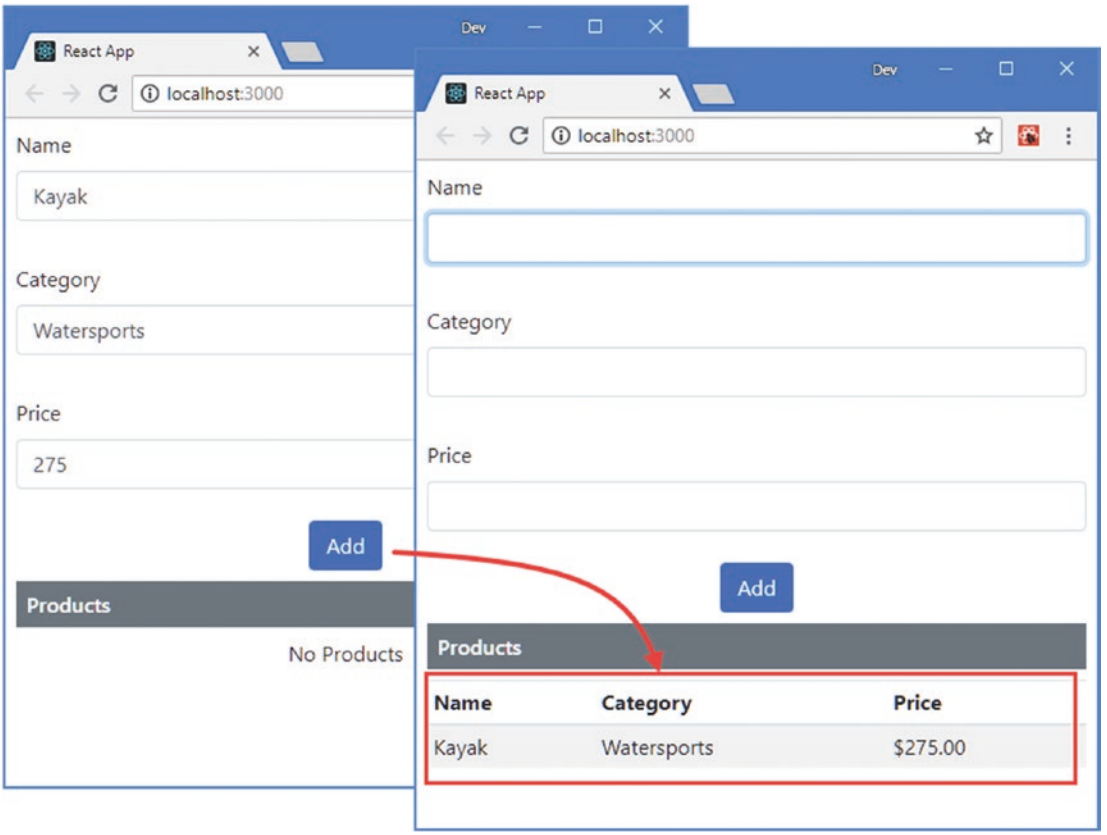

Figure 16-3. *Programmatically creating form elements and refs*

Validating Uncontrolled Form Components

Form elements have built-in validation support through the HTML Constraint Validation API, which can be accessed using refs. The validation API describes an element's validation status using an object like this one:

```
...
{
    valueMissing: true, tooShort: false, rangeUnderflow: false
}
...
```

The valueMissing property will be true when I have specified that the element must have a value but is empty. The tooShort property will be true when there are fewer characters in the element's value than specified by the validation rules. The rangeUnderflow property will be true for numeric values that are smaller than a specified minimum value.

To process this type of validation object, I added a file called ValidationMessages.js to the src folder and used it to define the function shown in Listing 16-13.

Listing 16-13. The Contents of the ValidationMessages.js File in the src Folder

```
export function GetValidationMessages(elem) {
    let errors = [];
    if (!elem.checkValidity()) {
        if (elem.validity.valueMissing) {
            errors.push("Value required");
        }
        if (elem.validity.tooShort) {
            errors.push("Value is too short");
        }
        if (elem.validity.rangeUnderflow) {
            errors.push("Value is too small");
        }
    }
    return errors;
}
```

The GetValidationMessages function receives an HTML element object and asks the browser for data validation by calling the element's checkValidity method. The checkValidity method returns true if the element's value is valid and false otherwise. If the element's value isn't valid, then the element's validity property is checked for the valueMissing, tooShort, and rangeUnderflow properties with true values and used to create an array of errors that can be shown to the user.

■ **Tip** The HTML validation features include a wider range of validation checks and validity properties than I use in this chapter. See https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation for a good description of the available features.

I added a file called ValidationDisplay.js in the src folder and used it to define a component that will display the validation messages for a single element, as shown in Listing 16-14.

Listing 16-14. The Contents of the ValidationDisplay.js File in the src Folder

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

export class ValidationDisplay extends Component {

    render() {
        return this.props.errors
            ? this.props.errors.map(err =>
                <div className="small bg-danger text-white mt-1 p-1"
                        key={ err } >
                    { err }
```

```
                </div>)
            : null
    }
}
```

This component receives an array of error messages that it should display and returns `null` to indicate no content if there are no error messages to show. In Listing 16-15, I have updated the `Editor` component so that validation attributes are applied to the form elements and validation checks are performed before the form data is used.

Listing 16-15. Applying Validation in the Editor.js File in the src Folder

```
import React, { Component } from "react";
import { ValidationDisplay } from "./ValidationDisplay";
import { GetValidationMessages } from "./ValidationMessages";

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.formElements = {
            name: { label: "Name", name: "name",
                validation: { required: true, minLength: 3 }},
            category: { label: "Category", name:"category",
                validation: { required: true, minLength: 5 }},
            price: { label: "Price", name: "price",
                validation: { type: "number", required: true, min: 5 }}
        }
        this.state = {
            errors: {}
        }
    }

    setElement = (element) => {
        if (element !== null) {
            this.formElements[element.name].element = element;
        }
    }

    handleAdd = () => {
        if (this.validateFormElements()) {
            let data = {};
            Object.values(this.formElements)
                .forEach(v => {
                    data[v.element.name] = v.element.value;
                    v.element.value = "";
                });
            this.props.callback(data);
            this.formElements.name.element.focus();
        }
    }
```

```
    validateFormElement = (name) => {
        let errors = GetValidationMessages(this.formElements[name].element);
        this.setState(state => state.errors[name] = errors);
        return errors.length === 0;
    }

    validateFormElements = () => {
        let valid = true;
        Object.keys(this.formElements).forEach(name => {
            if (!this.validateFormElement(name)) {
                valid = false;
            }
        })
        return valid;
    }

    render() {
        return <React.Fragment>
             {
                Object.values(this.formElements).map(elem =>
                    <div className="form-group p-2" key={ elem.name }>
                        <label>{ elem.label }</label>
                        <input className="form-control"
                            name={ elem.name }
                            autoFocus={ elem.name === "name" }
                            ref={ this.setElement }
                            onChange={ () => this.validateFormElement(elem.name) }
                            { ...elem.validation} />
                        <ValidationDisplay
                             errors={ this.state.errors[elem.name] } />
                    </div>)
            }
            <div className="text-center">
                <button className="btn btn-primary" onClick={ this.handleAdd }>
                    Add
                </button>
            </div>
        </React.Fragment>
    }
}
```

I included the validation attributes for each element in the objects that describes that element, like this:

```
...
name: { label: "Name", name: "name", validation: { required: true, minLength: 3 }},
...
```

The `required` attribute indicates that a value is required, and the `minLength` attribute specifies that the value should contain at least three characters. These attributes are applied to the `input` elements when they are created by the `render` method.

```
...
<input className="form-control" name={ elem.name }
    autoFocus={ elem.name === "name" } ref={ this.setElement }
    onChange={ () => this.validateFormElement(elem.name) }
    { ...elem.validation} />
...
```

I don't have to worry about the pristine/dirty element issue I described in Chapter 15 because validation isn't performed until the `checkValidity` method is invoked, which will happen in response to the `change` event, which I handle using the `onChange` event prop and the `validateFormElement` method, with the effect that validation for an element begins only when the user starts to type, as shown in Figure 16-4.

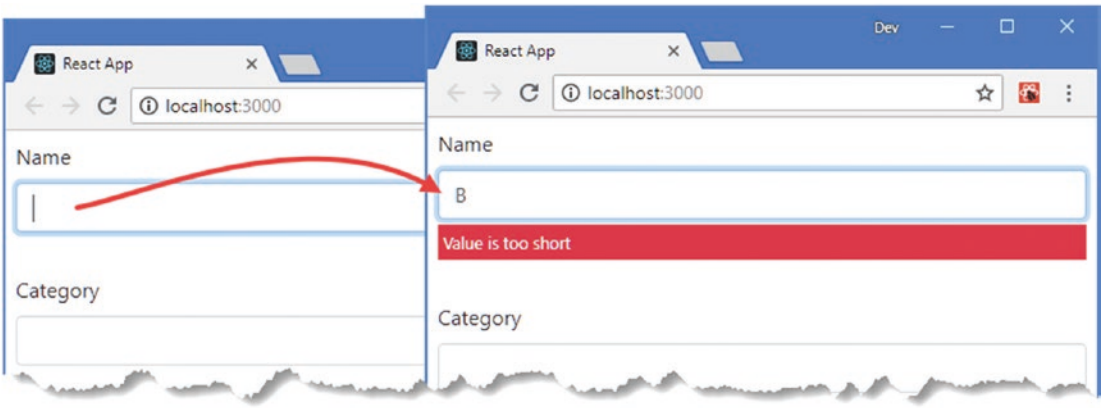

Figure 16-4. *Validating an element*

When the user clicks the Add button, the `handleAdd` method invokes the `validateFormElements` button, which validates all the elements and ensures that the form data isn't used until the problems are resolved, as shown in Figure 16-5. The effects of changes are shown immediately because each edit triggers a `change` event that causes the element's value to validated again.

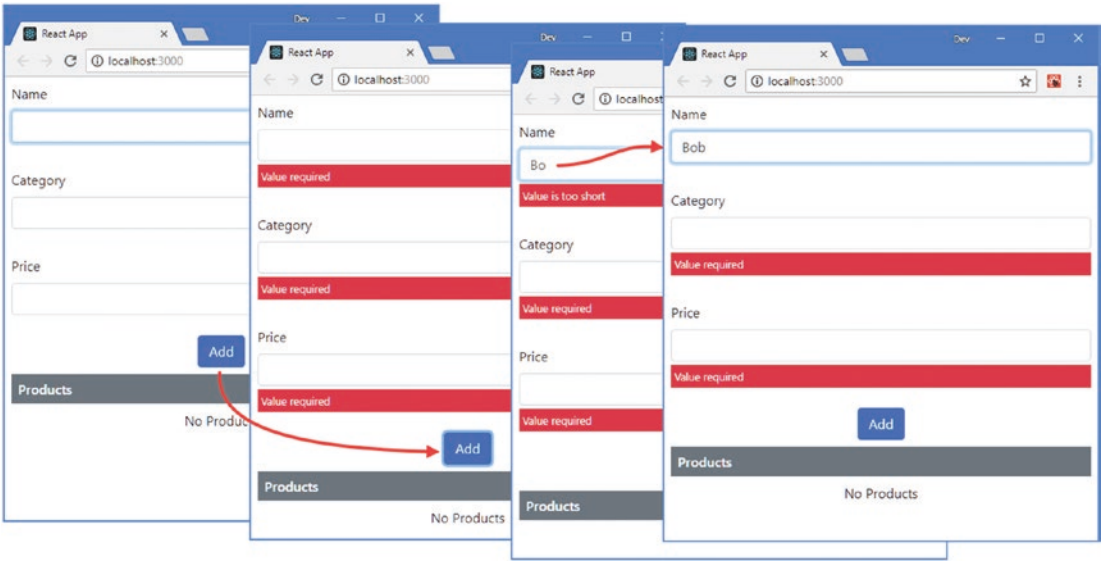

Figure 16-5. *Validating all elements*

Understanding Refs and the Lifecycle

Refs are not assigned a value until React invokes a component's `render` method. If you are using the `createRef` method, the `current` property will not be assigned a value before the component has rendered its content. Similarly, callback refs won't invoke their method until the component has rendered.

The assignment of refs may seem late in the component lifecycle, but refs provide access to DOM elements, which are not created until the rendering phase, which means that React hasn't created the elements that refs refer to until the `render` method is invoked. The element associated with a ref can be accessed only in the `componentDidMount` and `componentDidUpdate` lifecycle methods because they occur after rendering has been completed and the DOM has been populated or updated.

One consequence of using refs is that a component can't rely on the state feature to preserve its context when React replaces the elements it renders in the DOM. React tries to minimize DOM changes, but you cannot rely on the same element being used throughout the life of an application. As noted in Chapter 13, changing the top-level element rendered by a component causes React to replace its elements in the DOM, as shown in Listing 16-16.

Listing 16-16. Rendering a Different Top-Level Element in the Editor.js File in the src Folder

```
import React, { Component } from "react";
import { ValidationDisplay } from "./ValidationDisplay";
import { GetValidationMessages } from "./ValidationMessages";

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.formElements = {
            name: { label: "Name", name: "name",
```

```
                validation: { required: true, minLength: 3 }},
            category: { label: "Category", name:"category",
                validation: { required: true, minLength: 5 }},
            price: { label: "Price", name: "price",
                validation: { type: "number", required: true, min: 5 }}
        }
        this.state = {
            errors: {},
            wrapContent: false
        }
    }

    setElement = (element) => {
        if (element !== null) {
            this.formElements[element.name].element = element;
        }
    }

    handleAdd = () => {
        if (this.validateFormElements()) {
            let data = {};
            Object.values(this.formElements)
                .forEach(v => {
                    data[v.element.name] = v.element.value;
                    v.element.value = "";
                });
            this.props.callback(data);
            this.formElements.name.element.focus();
        }
    }

    validateFormElement = (name) => {
        let errors = GetValidationMessages(this.formElements[name].element);
        this.setState(state => state.errors[name] = errors);
        return errors.length === 0;
    }

    validateFormElements = () => {
        let valid = true;
        Object.keys(this.formElements).forEach(name => {
            if (!this.validateFormElement(name)) {
                valid = false;
            }
        })
        return valid;
    }

    toggleWrap = () => {
        this.setState(state => state.wrapContent = !state.wrapContent);
    }
```

```
    wrapContent(content) {
        return this.state.wrapContent
            ? <div className="bg-secondary p-2">
                    <div className="bg-light">{ content }</div>
              </div>
            : content;
    }

    render() {
        return this.wrapContent(
            <React.Fragment>
                    <div className="form-group text-center p-2">
                        <div className="form-check">
                            <input className="form-check-input"
                                type="checkbox"
                                checked={ this.state.wrapContent }
                                onChange={ this.toggleWrap } />
                            <label className="form-check-label">Wrap Content</label>
                        </div>
                    </div>
                {
                    Object.values(this.formElements).map(elem =>
                        <div className="form-group p-2" key={ elem.name }>
                            <label>{ elem.label }</label>
                            <input className="form-control"
                               name={ elem.name }
                               autoFocus={ elem.name === "name" }
                               ref={ this.setElement }
                               onChange={ () => this.validateFormElement(elem.name) }
                               { ...elem.validation} />
                            <ValidationDisplay
                                errors={ this.state.errors[elem.name] } />
                        </div>)
                }
                <div className="text-center">
                    <button className="btn btn-primary" onClick={ this.handleAdd }>
                        Add
                    </button>
                </div>
            </React.Fragment>)
    }
}
```

I have added a `wrapContent` state property that is set using a controlled checkbox and that wraps the content rendered by the component and ensures that React replaces the component's existing elements in the DOM with new ones. To see the effect, enter text into the Name field and check the Wrap Context checkbox, as shown in Figure 16-6.

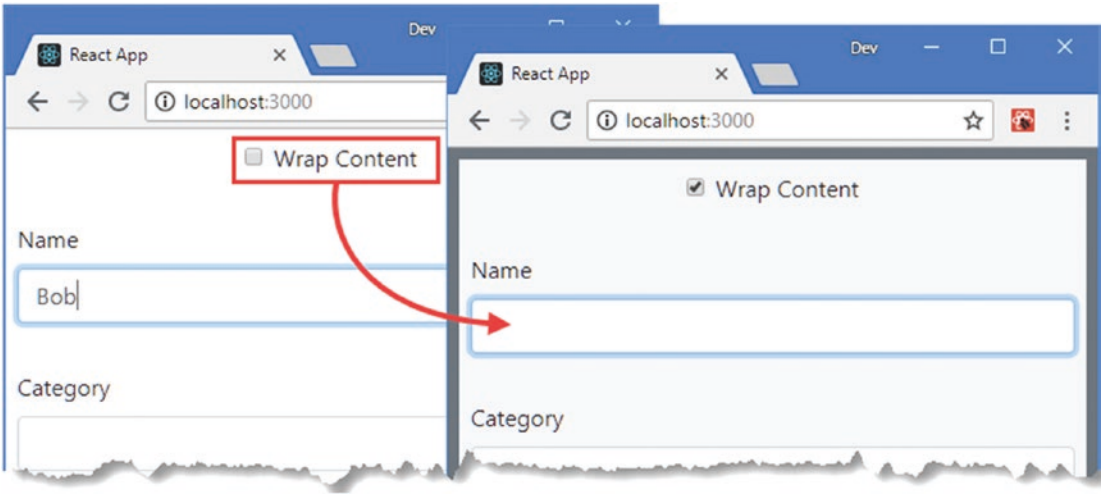

Figure 16-6. *Replacing elements*

The `input` element into which you entered text has been destroyed, and its content has been lost. To make matters more confusing for the user, any validation errors that have been detected are part of the component's state data, which means they will be displayed alongside the new `input` element, even though the data value they describe is no longer visible.

To help avoid this problem, the stateful component lifecycle includes the `getSnapshotBeforeUpdate` method, which is called between the `render` and `componentDidUpdate` methods in the update phase, as shown in Figure 16-7.

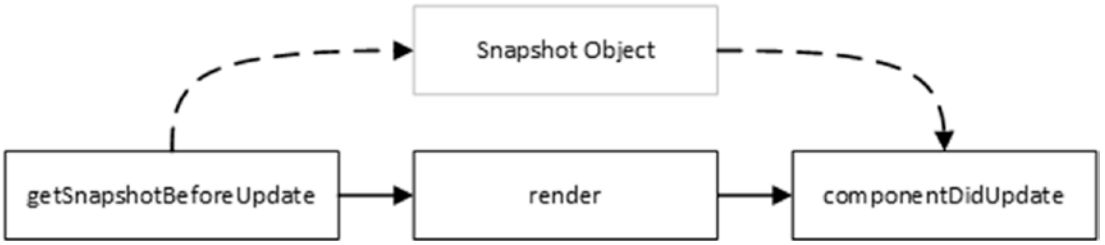

Figure 16-7. *The snapshot process*

This `getSnapshotBeforeUpdate` method allows a component to inspect its current content and generate a custom snapshot object before the `render` method is called. Once the update is complete, the `componentDidUpdate` method is called and provided with the snapshot object so that the component can modify the elements that are now in the DOM.

■ **Caution** A snapshot doesn't help preserve context if the component is unmounted and re-created, which can happen when an ancestor's content changes. In these situations, the `componentWillUnmount` method can be used to access refs, and the data can be preserved via a context, as described in Chapter 15.

In Listing 16-17, I have used the snapshot feature to capture the values entered into the input element before the update and restore those values after the update.

Listing 16-17. Taking a Snapshot in the Editor.js File in the src Folder

```
import React, { Component } from "react";
import { ValidationDisplay } from "./ValidationDisplay";
import { GetValidationMessages } from "./ValidationMessages";

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.formElements = {
            name: { label: "Name", name: "name",
                validation: { required: true, minLength: 3 }},
            category: { label: "Category", name:"category",
                validation: { required: true, minLength: 5 }},
            price: { label: "Price", name: "price",
                validation: { type: "number", required: true, min: 5 }}
        }
        this.state = {
            errors: {},
            wrapContent: false
        }
    }

    // ...other methods omitted for brevity...

    getSnapshotBeforeUpdate(props, state) {
        return Object.values(this.formElements).map(item =>
            {return { name: [item.name], value: item.element.value }})
    }

    componentDidUpdate(oldProps, oldState, snapshot) {
        snapshot.forEach(item => {
            let element = this.formElements[item.name].element
            if (element.value !== item.value) {
                element.value = item.value;
            }
        });
    }
}
```

The `getSnapshotBeforeUpdate` method receives the component's props and state as they were before the update was triggered and returns an object that will be passed to the `componentDidUpdate` method after the update. In the example, I don't need to access props or state because the data I need to preserve is contained in the `input` elements. React doesn't mandate a specific format for the snapshot object, and the `getSnapshotBeforeUpdate` method can return data in any format that will be useful. In the example, the `getSnapshotBeforeUpdate` method returns an array of objects with `name` and `value` properties.

Once React has completed the update, it calls the `componentDidUpdate` and provides the snapshot as an argument, along with the old props and state data. In the example, I process the array of objects and set the

values of the input elements. The result is that data entered into the `input` elements is preserved when the checkbox is toggled, as shown in Figure 16-8.

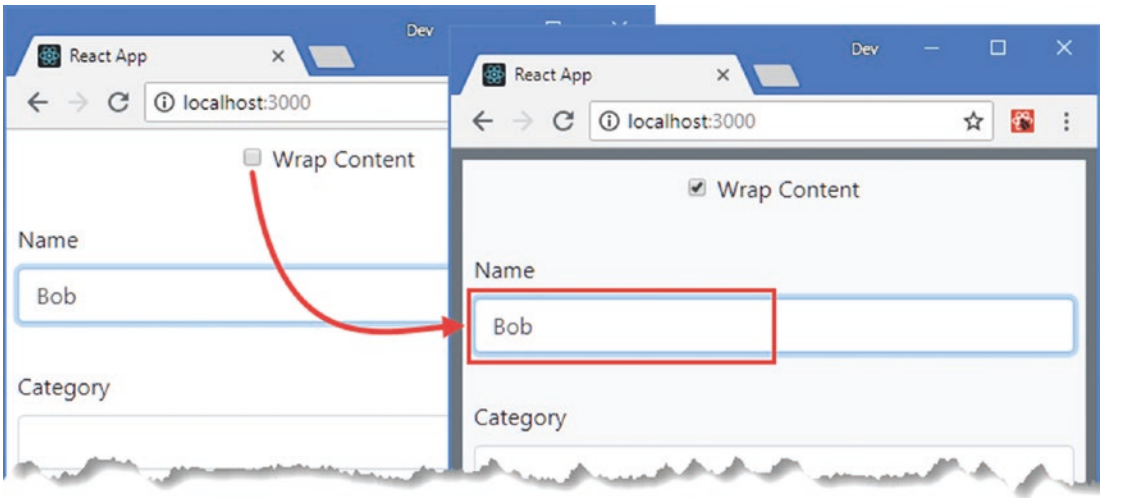

Figure 16-8. *Using snapshot data*

The `getSnapshotBeforeUpdate` and `componentDidUpdate` methods are called for every update, even when React hasn't replaced the component's elements in the DOM, which is why I apply a snapshot value only when an element's value differs from the snapshot value when the update has been completed.

UNDERSTANDING THE REFS RABBIT HOLE

There is an unintended consequence of using the HTML5 constraint validation API in the previous example. Validation is performed only when the user edits the contents of the text field and not when the value is set programmatically. When I use the snapshot data to set the value of a newly created `input` element, it will pass validation, even if the value previously failed validation. The effect is that the user can bypass validation by entering bad values into the `name` or `category` input elements, checking the wrap content checkbox, and clicking the Add button.

This is a problem that can be worked around, but the underlying issue is that using refs to access the DOM directly presents a series of small conflicts, each of which can be solved with the addition of a few lines of code. But these fixes often present other issues or compromises that require additional work, and the result is a fragile application made from complex components.

Working directly with the DOM can be essential in some projects, and there can be advantages to avoiding duplicating data and features that are already in the DOM. But use refs only when they are required because they can create as many problems as they solve.

Using Refs with Other Libraries or Frameworks

Some projects are moved to React gradually so that components have to interoperate with existing features that are written in another library or framework. The most common example is jQuery, which was the most popular choice for web application development before the era of frameworks like React and Angular and which is still widely used for simple projects. If you have an extensive set of features that are written in jQuery, for example, then you can apply them to the HTML elements rendered by a component using refs. To demonstrate, I am going to use jQuery to assign form elements with invalid elements to a class that will apply a Bootstrap style. I added a file called `jQueryColorizer.js` to the `src` folder and added the code shown in Listing 16-18.

■ **Note** This example requires the jQuery package that was added to the project in Listing 16-3. If you did not install jQuery, you should do so before proceeding.

Listing 16-18. The Contents of the jQueryColorizer.js in the src Folder

```
var $ = require('jquery');

export function ColorInvalidElements(rootElement) {
    $(rootElement)
        .find("input:invalid").addClass("border-danger")
            .removeClass("border-success")
        .end()
        .find("input:valid").removeClass("border-danger")
             .addClass("border-success");
}
```

The jQuery statement locates all the `input` elements that are assigned to the `invalid` pseudoclass and adds them to the `border-danger` class and adds any `input` elements in the `valid` pseudoclass to the `border-success` class. The `valid` and `invalid` classes are used by the HTML constraint validation API to indicate an element's validation status. In Listing 16-19, I have added a ref and used it to invoke the jQuery function from the App component.

MIXING FRAMEWORKS

Using refs to incorporate other frameworks is difficult and prone to problems. Like any use of refs, it should be done with caution and only when you are unable to rewrite the functionality in React. You may feel that you will save time by building on your existing code, but my experience is that any time saved will be spent trying to work around a long series of small problems that arise because the two frameworks work in different ways.

If you have to use another library or framework alongside React, then you should pay close attention to the way that the frameworks approach the DOM. You will find that React and the other framework expect to have complete control of the content they create, and unexpected results can arise when elements are added, removed, or changed in a way that the framework developers did not expect.

Listing 16-19. Invoking a Function in the App.js File in the src Folder

```
import React, { Component } from "react";
import { Editor } from "./Editor"
import { ProductTable } from "./ProductTable";
import { ColorInvalidElements } from "./jQueryColorizer";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            products: []
        }
        this.editorRef = React.createRef();
    }

    addProduct = (product) => {
        if (this.state.products.indexOf(product.name) === -1) {
            this.setState({ products: [...this.state.products, product ]});
        }
    }

    colorFields = () => {
        ColorInvalidElements(this.editorRef.current);
    }

    render() {
        return <div>
                <div className="text-center m-2">
                    <button className="btn btn-primary" onClick={ this.colorFields }>
                        jQuery
                    </button>
                </div>
                <div ref={ this.editorRef} >
                    <Editor callback={ this.addProduct } />
                </div>
                <h6 className="bg-secondary text-white m-2 p-2">Products</h6>
                    <div className="m-2">
                        {
                            this.state.products.length === 0
                                ? <div className="text-center">No Products</div>
                                : <ProductTable products={ this.state.products } />
                        }
                    </div>
            </div>
    }
}
```

The result is that clicking the jQuery button invokes the `colorFields` method, which uses the ref to provide the jQuery function with the HTML element it requires. The jQuery function applies borders to the input elements to indicate their validation status, as shown in Figure 16-9. (The difference in the border

colors will not be evident in the printed edition of this book, and this is an example that is best run in the browser to see the effect.)

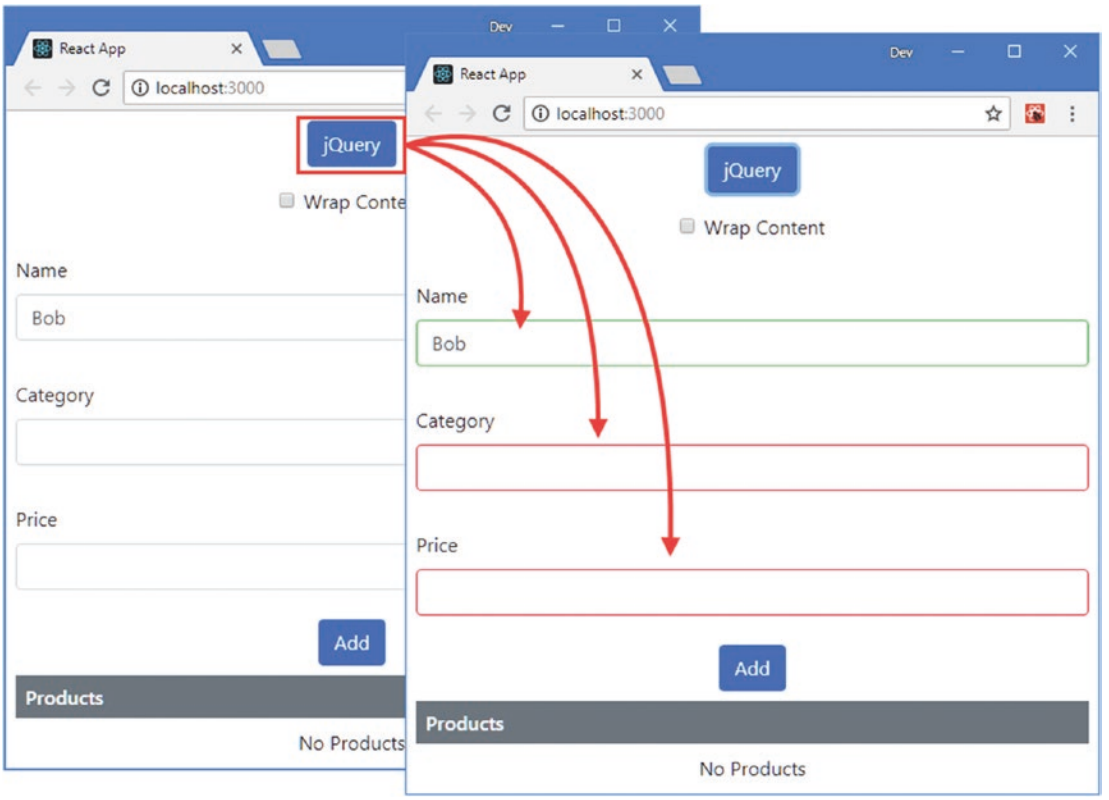

Figure 16-9. *Providing jQuery with an element via a ref*

ACCESSING COMPONENTS WITH REFS

In Listing 16-19, I added a `div` element around the `Editor` element. When React renders the content into the DOM, the `Editor` element won't be part of the HTML document, and adding the `div` element ensures that jQuery is able to access the application's content.

Refs do work with components, and if I had applied the `ref` prop to the `Editor` element, the value of the ref's `current` property will be assigned to the `Editor` object that React created when rendering the `App` component's content.

A ref to a component allows access to that component's state data and methods. It can be tempting to use refs to invoke a child component's methods because it produces a development experience that more closely resembles the way that objects are conventionally used.

Manipulating a component via a ref is bad practice. It produces tightly coupled components that end up working against React. The state data, props, and event features may feel less natural at first, but you will become accustomed to them, and the result is an application that takes full advantage of React and that is easier to write, test, and maintain.

Accessing a Child Component's Content

The `refs` prop is given special handling by React, which means that care must be taken when a component requires a ref to a DOM element rendered by one of its descendants. The simplest approach is to pass the ref object or callback function using a different name, in which case React will pass along the ref as it would any other prop. To demonstrate, I added a file called `FormField.js` to the `src` folder and used it to define the component shown in Listing 16-20.

■ **Note** Accessing a child component's content should be done with caution because it creates tightly coupled components that are harder to write and test. Where possible, you should use props to communicate between components.

Listing 16-20. The Contents of the FormField.js File in the src Folder

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

export class FormField extends Component {

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

    handleChange = (ev) => {
        this.setState({ fieldValue: ev.target.value});
    }

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

This component renders a controlled `input` element and uses a prop called `fieldRef` to associate the `ref` received from the parent with the element. In Listing 16-21, I have replaced the content rendered by the `App` component to use the `FormField` component and provide it with a ref.

Listing 16-21. Replacing the Contents of the App.js File in the src Folder

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

export default class App extends Component {
```

```
    constructor(props) {
        super(props);
        this.fieldRef = React.createRef();
    }

    handleClick = () => {
        this.fieldRef.current.focus();
    }

    render() {
        return <div className="m-2">
                <FormField label="Name" fieldRef={ this.fieldRef } />
                <div className="text-center m-2">
                    <button className="btn btn-primary"
                            onClick={ this.handleClick }>
                        Focus
                    </button>
                </div>
            </div>
    }
}
```

The App component creates a ref and passes it to the `FormField` component using the `fieldRef` prop, which is then applied to the `input` element using `ref`. The result is that clicking the Focus button, rendered by the App component, will focus the input element, rendered by its child, as shown in Figure 16-10.

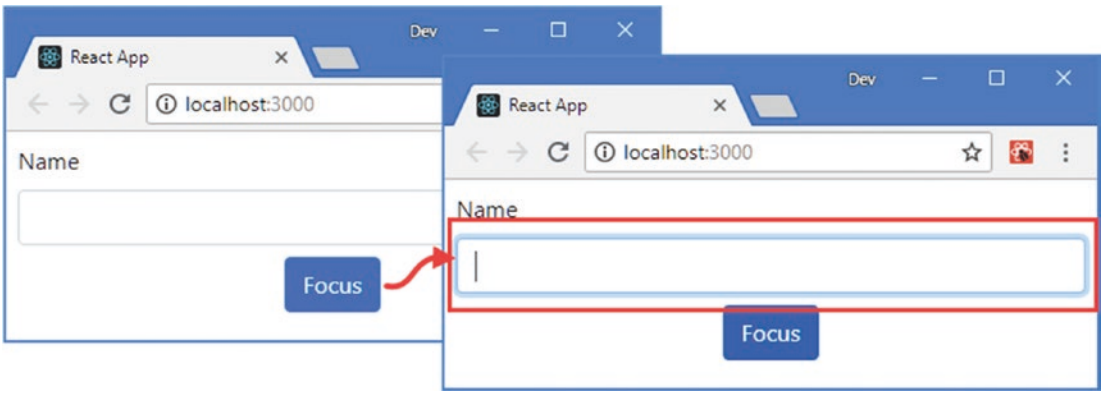

Figure 16-10. *Accessing a child's content*

Using Ref Forwarding

React provides an alternative approach to passing refs to children, known as *ref forwarding*, which allows `ref` to be used instead of a regular prop. In Listing 16-22, I have used ref forwarding for the `FormField` component.

Listing 16-22. Using Ref Forwarding in the FormField.js File in the src Folder

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

export const ForwardFormField = React.forwardRef((props, ref) =>
     <FormField { ...props } fieldRef={ ref } />
)

export class FormField extends Component {

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

    handleChange = (ev) => {
        this.setState({ fieldValue: ev.target.value});
    }

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

The `React.forwardRef` method is passed a function that receives props and the `ref` value and renders content. In this case, I receive the `ref` value and forward it to the `fieldRef` prop, which is the prop name that the `FormField` component expects to receive. I exported the result from the `forwardRef` method as `ForwardFormField`, which I have used in the `App` component, as shown in Listing 16-23.

Listing 16-23. Using Ref Forwarding in the App.js File in the src Folder

```
import React, { Component } from "react";
import { ForwardFormField } from "./FormField";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.fieldRef = React.createRef();
    }

    handleClick = () => {
        this.fieldRef.current.focus();
    }
```

```
    render() {
        return <div>
                <ForwardFormField label="Name" ref={ this.fieldRef } />
                <div className="text-center m-2">
                    <button className="btn btn-primary"
                            onClick={ this.handleClick }>
                        Focus
                    </button>
                </div>
            </div>
    }
}
```

This example produces the same effect as shown in Figure 16-10, with the advantage that the App component doesn't require any special knowledge of how the ref is handled inside the child component.

Using Portals

A portal allows a component to render its content into a specific DOM element, instead of being presented as part of its parent's content. This feature lets a component break out of the normal React component model but requires the target element to be created and managed outside of the application, meaning that you can't use portals to render the content into a different component. As a consequence, this feature is useful in a limited range of situations, such as when creating dialogs or model alerts to the user or when integrating React into content created by another framework or library. In Listing 16-24, I have added new HTML elements to the index.html file so that there is a DOM element outside of the content rendered by the example application that I can target with a portal.

Listing 16-24. Adding Elements in the index.html File in the public Folder

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <meta name="theme-color" content="#000000">
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
  <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
  <title>React App</title>
</head>

<body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>

  <div class="container">
    <div class="row">
      <div class="col">
        <div id="root"></div>
      </div>
```

```
      <div class="col">
        <div id="portal" class="m-2">
          <h6 class="bg-info text-white text-center p-2">
            This is the portal target
          </h6>
        </div>
      </div>
    </div>
  </div>
</body>
</html>
```

The new elements are assigned to Bootstrap CSS grid classes so that the portal target element is shown alongside the content rendered by the application, as shown in Figure 16-11.

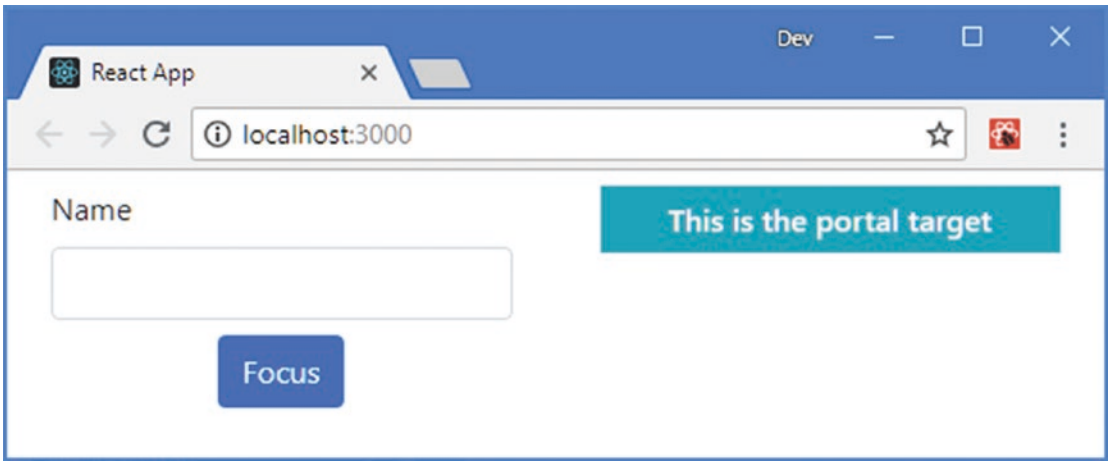

Figure 16-11. *Adding an Element to the HTML Document*

I added a file called `PortalWrapper.js` in the `src` folder and used it to define the component shown in Listing 16-25, which locates the target element in the DOM and uses it to create a portal.

Listing 16-25. The Contents of the PortalWrapper.js File in the src Folder

```
import React, { Component } from "react";
import ReactDOM from "react-dom";

export class PortalWrapper extends Component {

    constructor(props) {
        super(props);
        this.portalElement = document.getElementById("portal");
    }
```

```
    render() {
        return ReactDOM.createPortal(
            <div className="border p-3">{ this.props.children }</div>
        , this.portalElement);
    }
}
```

The PortalWrapper component is defined using the props.children property to create a container but returns its content using the ReactDOM.createPortal method, whose arguments are the content to render and the DOM target element. In this example, I use the DOM API's getElementById method to locate the target element added to the HTML file in Listing 16-24. In Listing 16-26, I have used the portal in the App component.

USING REFS FOR PORTALS

You cannot use a portal to render content to an element using a ref. Portals are used during the rendering process, and refs are not assigned elements until rendering is complete, which means that you won't be able to access an element via a ref early enough in the lifecycle for the ReactDOM.createPortal method. Use contexts, as described in Chapter 14, if you need coordination between components in different parts of the application or use one of the packages described in Part 3.

Listing 16-26. Using a Portal in the App.js File in the src Folder

```
import React, { Component } from "react";
import { ForwardFormField } from "./FormField";
import { PortalWrapper } from "./PortalWrapper";

export default class App extends Component {

    constructor(props) {
        super(props);
        this.fieldRef = React.createRef();
        this.portalFieldRef = React.createRef();
    }

    focusLocal = () => {
        this.fieldRef.current.focus();
    }

    focusPortal = () => {
        this.portalFieldRef.current.focus();
    }

    render() {
        return <div>
                <PortalWrapper>
                    <ForwardFormField label="Name" ref={ this.portalFieldRef } />
                </PortalWrapper>
```

```
                <ForwardFormField label="Name" ref={ this.fieldRef } />
                <div className="text-center m-2">
                    <button className="btn btn-primary m-1"
                            onClick={ this.focusLocal }>
                        Focus Local
                    </button>
                    <button className="btn btn-primary m-1"
                            onClick={ this.focusPortal }>
                        Focus Portal
                    </button>
                </div>
            </div>
    }
}
```

The `PortalWrapper` element is used to apply the new component as a container for a `ForwardFormField`. The content displayed by the portal is treated as though it is part of the `App` components content, such that events will bubble up as normal and refs can be assigned, even though the content of the portal is being rendered outside the application. The `App` component is unaware that a portal is being used, and clicking the Focus Local and Focus Portal buttons uses the same ref technique to focus the `input` element presented by each `ForwardFormField` component, as shown in Figure 16-12.

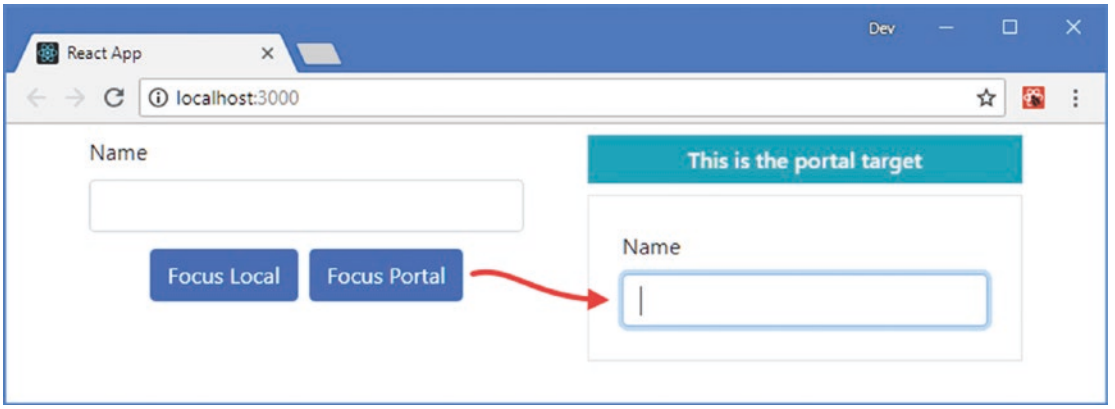

Figure 16-12. *Using a portal*

Summary

In this chapter, I described the React features for working directly with the DOM. I explained how refs can provide access to content rendered by a component and how this makes uncontrolled form elements possible. I also demonstrated a portal, which allows content to be rendered outside of the application's component hierarchy. These features can be invaluable but should be used sparingly because they undermine the normal React development model and result in closely coupled components. In the next chapter, I show you how to perform unit testing on React components.