CHAPTER 15

Forms and Validation

Forms allow applications to collect data from the user. In this chapter, I explain how React works with form elements, using state properties to set their value and event handlers to respond to user interactions. I show you how to work with different element types and show you how to validate the data that the user provides in a form so that the application receives data it can use. Table 15-1 puts forms and validation in context.

Table 15-1. *Putting Forms and Validation in Context*

Question	Answer
What are they?	Forms are the basic mechanism that allows applications to prompt the user for data. Validation is the process of checking that data to ensure it can be used by the application.
Why are they useful?	Most applications require some degree of data entry from the user, such as e-mail addresses, payment details, or shipping addresses. Forms allow the user to enter that data, either in free-text form or by selecting from a range of predefined choices. Validation is used to ensure that the data is in a format that the application expects and can process.
How are they used?	In this chapter, I describe controlled form elements, whose value is set using the `value` or `checked` props and whose `change` events are handled to process user editing or selection. These features are also used for validation.
Are there any pitfalls or limitations?	There are differences in the way that different form elements behave and small deviations between React and the standard HTML form elements, as described in later sections.
Are there any alternatives?	Applications do not have to use form elements at all. In some applications, uncontrolled form elements, where React is not responsible for managing the element's data, may be a more suitable choice, as described in Chapter 16.

Table 15-2 summarizes the chapter.

Table 15-2. *Chapter Summary*

Problem	Solution	Listing
Add a form element to a component	Add the element to the content rendered by the component. Set the initial value of the element using the `value` prop and respond to changes using the onChange prop.	1–10, 12, 13
Determine the state of a checkbox	Inspect the checked property of the `target` element when handling the change event	11
Validate form data	Define validation rules and apply them when the user edits a field and triggers a change event	14–25

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 15-1.

■ **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 15-1. Creating the Example Project

```
npx create-react-app forms
```

Run the commands shown in Listing 15-2 to navigate to the `forms` folder to add the Bootstrap package and a validation package to the project. (I use the validation package in the "Validating Form Data" section.)

Listing 15-2. Adding Packages to the Project

```
cd forms
npm install bootstrap@4.1.2
npm install validator@10.7.1
```

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 15-3 to the `index.js` file, which can be found in the `src` folder.

Listing 15-3. 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();
```

Defining the Example Components

Add a file called `Editor.js` to the `src` folder and add the content shown in Listing 15-4.

Listing 15-4. The Contents of the Editor.js File in the src Folder

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

export class Editor extends Component {

    render() {
        return <div className="h5 bg-info text-white p-2">
                    Form Will Go Here
                </div>
    }
}
```

I will use this component to display a form to the user. To start, however, this component renders a placeholder message. Next, add a file called `Display.js` to the `src` folder and add the content shown in Listing 15-5.

Listing 15-5. The Contents of the Display.js File in the src Folder

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

export class Display extends Component {

    formatValue = (data) => Array.isArray(data)
        ? data.join(", ") : data.toString();

    render() {
        let keys = Object.keys(this.props.data);
        if (keys.length === 0) {
            return <div className="h5 bg-secondary p-2 text-white">
                No Data
            </div>
        } else {
            return <div className="container-fluid bg-secondary p-2">
                    { keys.map(key =>
                        <div key={key} className="row h5 text-white">
```

```
                            <div className="col">{ key }:</div>
                            <div className="col">
                                { this.formatValue(this.props.data[key]) }
                            </div>
                        </div>
                    )}
                </div>
        }
    }
}
```

This component receives a `data` prop and enumerates its properties and values in a grid. Finally, change the content in the `App.js` file to replace the content added when the project was created with the component shown in Listing 15-6.

Listing 15-6. The Contents of the App.js File in the src Folder

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

export default class App extends Component {

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

    submitData = (newData) => {
        this.setState({ formData: newData});
    }

    render() {
        return <div className="container-fluid">
            <div className="row p-2">
                <div className="col-6">
                    <Editor submit={ this.submitData } />
                </div>
                <div className="col-6">
                    <Display data={ this.state.formData } />
                </div>
            </div>
        </div>
    }
}
```

Starting the Development Tools

Using the command prompt, run the command shown in Listing 15-7 in the `forms` folder to start the development tools.

Listing 15-7. 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 15-1.

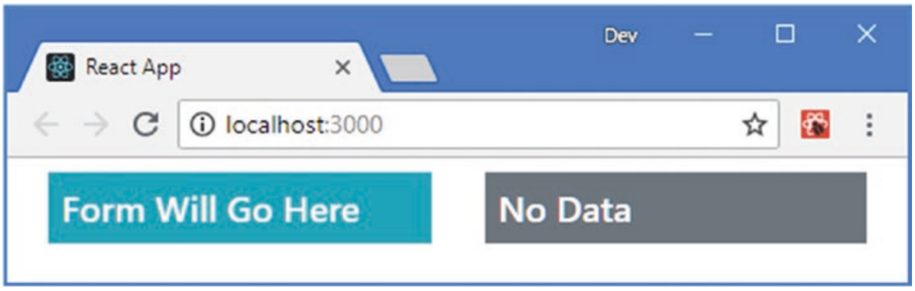


Figure 15-1. *Running the example application*

Using Form Elements

The simplest way to use form elements is to build on the React capabilities described in earlier chapters, using the state and event features. The result is known as a *controlled component*, and it will be familiar from earlier examples. In Listing 15-8, I have added an `input` element whose content is managed by React to the `Editor` component.

■ **Tip** There is also an approach called an *uncontrolled component*, which I describe in Chapter 16.

Listing 15-8. Adding a Form Element 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: ""
        }
    }
    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value },
            () => this.props.submit(this.state));
    }

    render() {
        return <div className="h5 bg-info text-white p-2">
                    <div className="form-group">
                        <label>Name</label>
                        <input className="form-control"
                            name="name"
                            value={ this.state.name }
                            onChange={ this.updateFormValue } />
                    </div>
                </div>
    }
}
```

The input element's value attribute is set using the name state property, and changes are handled using the updateFormValue method, which has been selected using the onChange prop. Most forms require multiple fields and rather than define a different event handling method for each of them, it is a good idea to use one method and ensure that the form element is configured to indicate with which state value it is associated. In this example, I have used the name prop to specify the state property's name, which I then read from the event received by the handler method:

```
...
updateFormValue = (event) => {
    this.setState({ [event.target.name]: event.target.value },
        () => this.props.submit(this.state));
}
...
```

The contents of the square brackets (the [and] characters) are evaluated to get the property name for the state update, which allows me to use the name property from the event.target object with the setState method. Not all types of form element can be processed in the same way, as you will see in later examples, but this approach reduces the number of event handling methods in a component.

■ **Tip** Set the state property to the empty string ("") if you want to present an empty input element to the user. You can see examples of empty elements in Listing 15-8. Don't use null or undefined because these values cause React to generate a warning in the browser's JavaScript console.

Notice that I have used the callback option provided by the setState method to invoke the submit function prop after the state data has been updated, which allows me to send the form data to the parent component. This means that any change in the state data of the Editor component is also pushed to the App component so that it can be shown by the Display component, with the result that typing into the input element is immediately reflected in the content presented to the user, as shown in Figure 15-2. This may seem like needless duplication of state data, but it will allow me to more easily implement validation features later in this chapter.

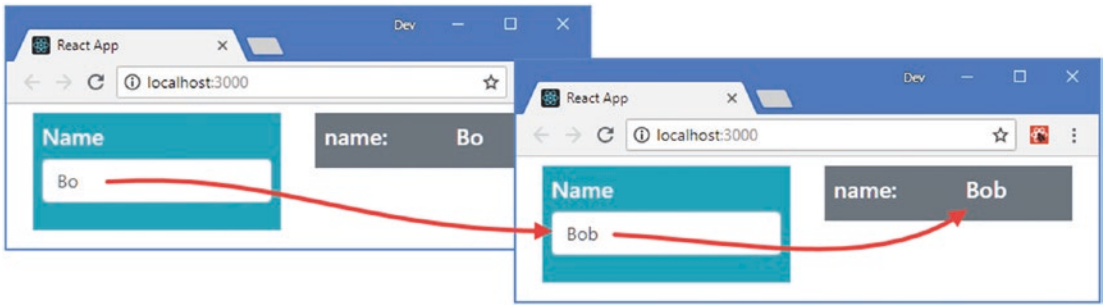


Figure 15-2. *Using a controlled component*

Using Select Elements

Once the basic structure is in place, a controller component can easily support additional form elements. In Listing 15-9, I have added two `select` elements to the `Editor` component.

Listing 15-9. Adding Select Elements 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: "Bob",
            flavor: "Vanilla",
            toppings: ["Strawberries"]
        }

        this.flavors = ["Chocolate", "Double Chocolate",
            "Triple Chocolate", "Vanilla"];
        this.toppings = ["Sprinkles", "Fudge Sauce",
                         "Strawberries", "Maple Syrup"]
    }

    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value },
            () => this.props.submit(this.state));
    }

    updateFormValueOptions = (event) => {
        let options = [...event.target.options]
            .filter(o => o.selected).map(o => o.value);
        this.setState({ [event.target.name]: options },
            () => this.props.submit(this.state));
    }
    render() {
        return <div className="h5 bg-info text-white p-2">
                    <div className="form-group">
                        <label>Name</label>
                        <input className="form-control"
                            name="name"
                            value={ this.state.name }
                            onChange={ this.updateFormValue } />
                    </div>
                    <div className="form-group">
                        <label>Ice Cream Flavors</label>
                        <select className="form-control"
                                name="flavor" value={ this.state.flavor }
                                onChange={ this.updateFormValue } >
                            { this.flavors.map(flavor =>
                                <option value={ flavor } key={ flavor }>
                                    { flavor }
                                </option>
                            )}
                        </select>
                    </div>
                    <div className="form-group">
                        <label>Ice Cream Toppings</label>
                        <select className="form-control" multiple={true}
                                name="toppings" value={ this.state.toppings }
                                onChange={ this.updateFormValueOptions }>
                            { this.toppings.map(top =>
                                <option value={ top } key={ top }>
                                    { top }
                                </option>
                            )}
                        </select>
                    </div>
                </div>
    }
}
```

The select element is easy to work with, although care has to be taken for elements that display multiple values. For a basic select element, the value property is used to set the selected value, and selections are handled using the onChange property. The option elements presented by the select element can be specified as regular HTML elements or generated programmatically, in which case they will require a key property, like this:

```
...
<select className="form-control" name="flavor" value={ this.state.flavor }
        onChange={ this.updateFormValue } >
    { this.flavors.map(flavor =>
        <option value={ flavor } key={ flavor }>{ flavor }</option>
    )}
</select>
...
```

Changes to the `select` element that presents a single item for selection can be handled using the same method defined for the `input` element since the selected value is accessed through the `event.target.value` property.

Using Select Elements That Present Multiple Items

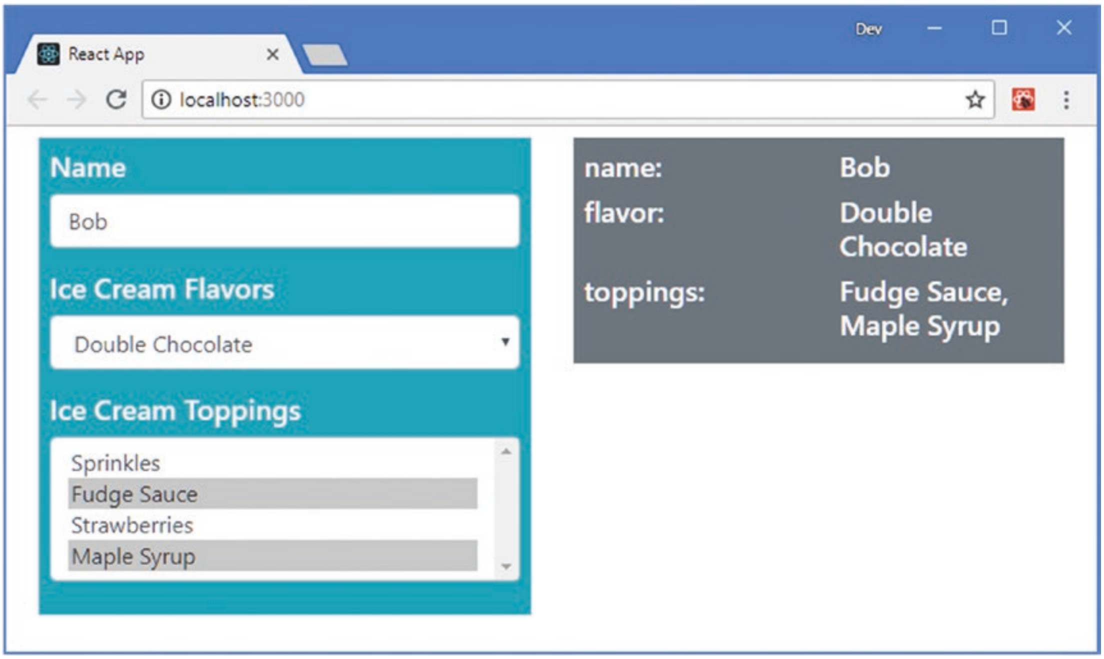
`Select` elements that allow multiple selections require a little more work. When defining the element, the `multiple` prop is set to `true` using an expression.

```
...
<select className="form-control" multiple={true} name="toppings"
    value={ this.state.toppings } onChange={ this.updateFormValueOptions }>
...
```

Using an expression avoids a common problem where assigning a string literal value to the `multiple` prop enables multiple elements, even when the string is `"false"`. Handling the user's selection requires a different handler method for the `change` event, as follows:

```
...
updateFormValueOptions = (event) => {
    let options = [...event.target.options]
        .filter(o => o.selected).map(o => o.value);
    this.setState({ [event.target.name]: options },
        () => this.props.submit(this.state));
}
...
```

The selections that the user has made are accessed through the `event.target.options` property, where the chosen items have a `selected` property whose value is `true`. In the listing, I create an array from the options, using the `filter` method to get the chosen items and the `map` method to get the `value` property, which leaves an array that contains the values from the `value` attribute of each chosen `option` element. Both `select` elements can be seen in Figure 15-3. (The data won't be shown by the `Display` component until you make a change.)



Name

Bob

Ice Cream Flavors

Double Chocolate

Ice Cream Toppings

Sprinkles
Fudge Sauce
Strawberries
Maple Syrup

name:	Bob
flavor:	Double Chocolate
toppings:	Fudge Sauce, Maple Syrup

Figure 15-3. *Using select elements*

Using Radio Buttons

Working with radio buttons requires a similar process to text `input` elements, where the user's selection can be accessed through the target element's `value` property, as shown in Listing 15-10.

Listing 15-10. Using Radio Buttons 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: "Bob",
            flavor: "Vanilla"
        }

        this.flavors = ["Chocolate", "Double Chocolate",
            "Triple Chocolate", "Vanilla"];
        this.toppings = ["Sprinkles", "Fudge Sauce",
                         "Strawberries", "Maple Syrup"]
    }

    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value },
            () => this.props.submit(this.state));
    }
    render() {
        return <div className="h5 bg-info text-white p-2">
                    <div className="form-group">
                        <label>Name</label>
                        <input className="form-control"
                            name="name"
                            value={ this.state.name }
                            onChange={ this.updateFormValue } />
                    </div>

                    <div className="form-group">
                        <label>Ice Cream Flavors</label>
                        { this.flavors.map(flavor =>
                            <div className="form-check" key={ flavor }>
                                <input className="form-check-input"
                                    type="radio" name="flavor"
                                    value={ flavor }
                                    checked={ this.state.flavor === flavor }
                                    onChange={ this.updateFormValue } />
                                <label className="form-check-label">
                                    { flavor }
                                </label>
                            </div>
                        )}
                    </div>
                </div>
    }
}
```

Radio buttons allow the user to select a single value from a list of options. The choice represented by the radio button is specified by its value property, and its checked property is used to ensure the element is selected correctly, as shown in Figure 15-4.

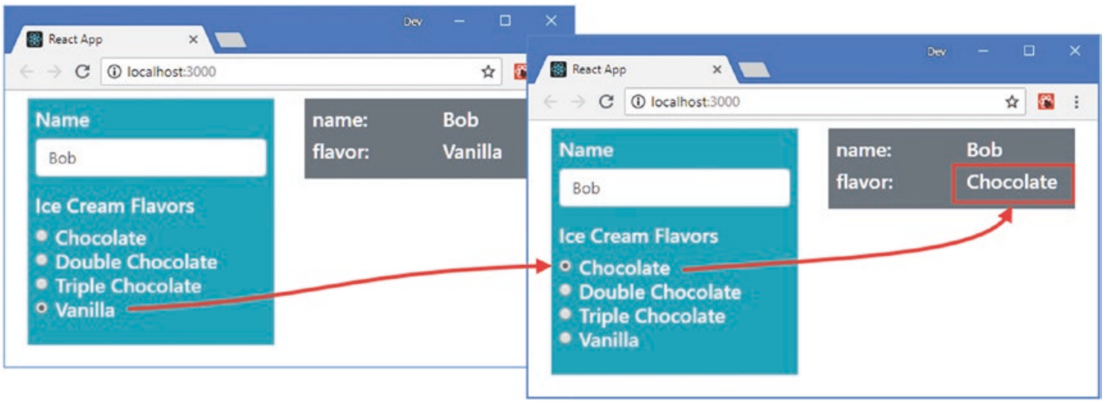


Figure 15-4. *Using radio buttons to present a choice*

Using Checkboxes

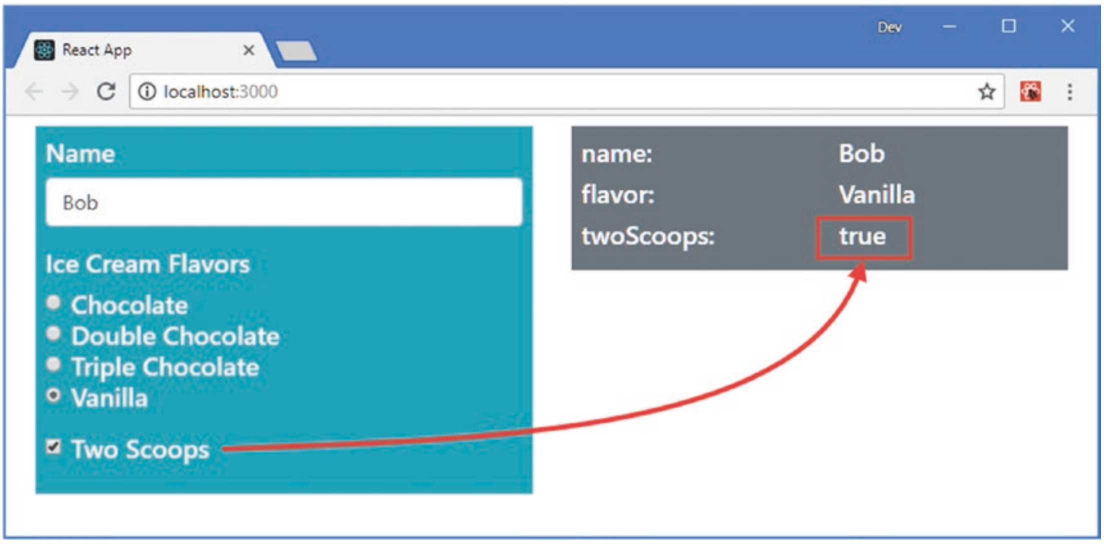
Checkboxes require a different approach because the checked property of the target element has to be read to determine whether the user has checked or unchecked the element, as shown in Listing 15-11.

Listing 15-11. Using a Checkbox 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: "Bob",
            flavor: "Vanilla",
            twoScoops: false
        }

        this.flavors = ["Chocolate", "Double Chocolate",
            "Triple Chocolate", "Vanilla"];
        this.toppings = ["Sprinkles", "Fudge Sauce",
                        "Strawberries", "Maple Syrup"]
    }

    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value },
            () => this.props.submit(this.state));
    }

    updateFormValueCheck = (event) => {
        this.setState({ [event.target.name]: event.target.checked },
            () => this.props.submit(this.state));
    }

    render() {
        return <div className="h5 bg-info text-white p-2">
                    <div className="form-group">
                        <label>Name</label>
                        <input className="form-control"
                            name="name"
                            value={ this.state.name }
                            onChange={ this.updateFormValue } />
                    </div>

                    <div className="form-group">
                        <label>Ice Cream Flavors</label>
                        { this.flavors.map(flavor =>
                            <div className="form-check" key={ flavor }>
                                <input className="form-check-input"
                                    type="radio" name="flavor"
```

```
                                        value={ flavor }
                                        checked={ this.state.flavor === flavor }
                                        onChange={ this.updateFormValue } />
                                    <label className="form-check-label">
                                        { flavor }
                                    </label>
                                </div>
                            )}
                        </div>

                        <div className="form-group">
                            <div className="form-check">
                                <input className="form-check-input"
                                    type="checkbox" name="twoScoops"
                                    checked={ this.state.twoScoops }
                                    onChange={ this.updateFormValueCheck } />
                                <label className="form-check-label">Two Scoops</label>
                            </div>
                        </div>
                    </div>
    }
}
```

The checked property is used to specify whether the checkbox should be checked when it is displayed, and the checked property is used when handling the change event to determine whether the user has checked or unchecked the element, as shown in Figure 15-5.



```
name:       Bob
flavor:     Vanilla
twoScoops:  true
```

Figure 15-5. Using a checkbox

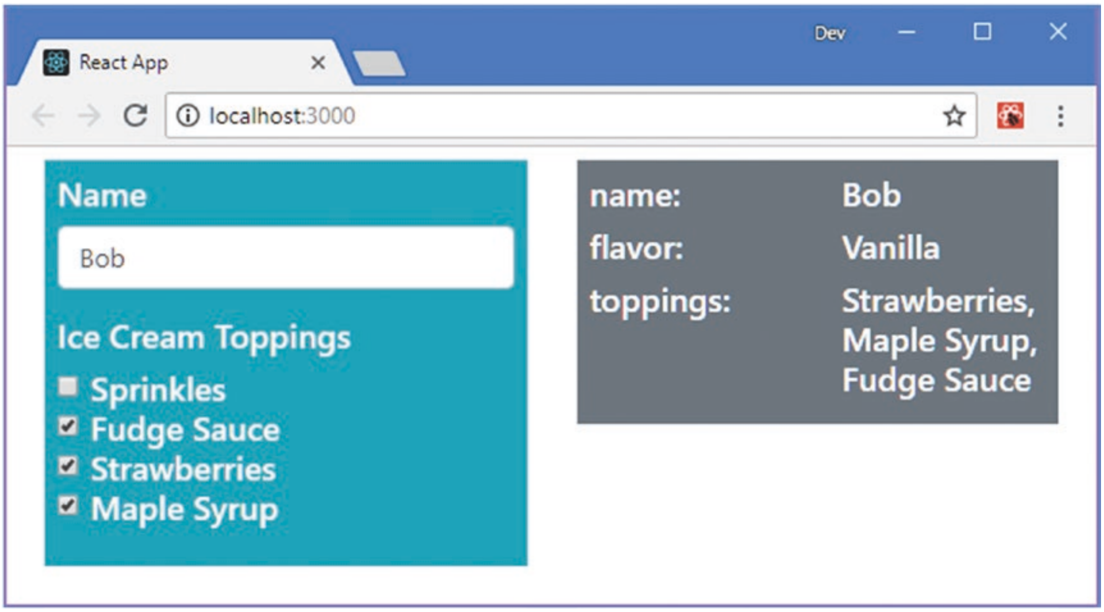
Using Checkboxes to Populate an Array

Checkboxes can also be used to populate an array, allowing users to choose from related options in a way that may be more familiar than a multi-option `select` element, as shown in Listing 15-12.

Listing 15-12. Using Related Checkboxes 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: "",
            flavor: "Vanilla",
            toppings: ["Strawberries"]
        }

        this.flavors = ["Chocolate", "Double Chocolate",
            "Triple Chocolate", "Vanilla"];
        this.toppings = ["Sprinkles", "Fudge Sauce",
                        "Strawberries", "Maple Syrup"]
    }

    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value },
            () => this.props.submit(this.state));
    }

    updateFormValueCheck = (event) => {
        event.persist();
        this.setState(state => {
            if (event.target.checked) {
                state.toppings.push(event.target.name);
            } else {
                let index = state.toppings.indexOf(event.target.name);
                state.toppings.splice(index, 1);
            }
        }, () => this.props.submit(this.state));
    }

    render() {
        return <div className="h5 bg-info text-white p-2">
                    <div className="form-group">
                        <label>Name</label>
                        <input className="form-control"
                            name="name"
                            value={ this.state.name }
                            onChange={ this.updateFormValue } />
                    </div>
                    <div className="form-group">
                        <label>Ice Cream Toppings</label>
                        { this.toppings.map(top =>
                            <div className="form-check" key={ top }>
                                <input className="form-check-input"
                                    type="checkbox" name={ top }
                                    value={ this.state[top] }
                                    checked={ this.state.toppings.indexOf(top) > -1 }
                                    onChange={ this.updateFormValueCheck } />
                                <label className="form-check-label">{ top }</label>
                            </div>
                        )}
                    </div>
                </div>
    }
}
```

The elements are generated in the same way, but changes are required to the `updateFormValueCheck` method to manage the contents of the `toppings` array so that it contains only the user's chosen values. The standard JavaScript array features are used to remove a value from the array when the corresponding checkbox is unchecked and to add a value when the checkbox is checked, producing the result shown in Figure 15-6.



Name

Bob

Ice Cream Toppings

- ☐ Sprinkles
- ☑ Fudge Sauce
- ☑ Strawberries
- ☑ Maple Syrup

name:	Bob
flavor:	Vanilla
toppings:	Strawberries, Maple Syrup, Fudge Sauce

Figure 15-6. *Using checkboxes to populate an array*

Using Text Areas

The content of a textarea element is set and read using the value property, unlike regular HTML. In Listing 15-13, I have added a textarea element to the example application and used the onChange handler to respond to edits.

Listing 15-13. Using a Text Area 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: "Bob",
            order: ""
        }
    }

    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value },
            () => this.props.submit(this.state));
    }

    render() {
        return <div className="h5 bg-info text-white p-2">
                    <div className="form-group">
                        <label>Name</label>
                        <input className="form-control"
                            name="name"
                            value={ this.state.name }
                            onChange={ this.updateFormValue } />
                    </div>

                    <div className="form-group">
                        <label>Order</label>
                        <textarea className="form-control" name="order"
                            value={ this.state.order }
                            onChange={ this.updateFormValue } />
                    </div>
                </div>
    }
}
```

Changes can be handled by the same method that I originally defined for text input elements, and the listing produces the result shown in Figure 15-7.

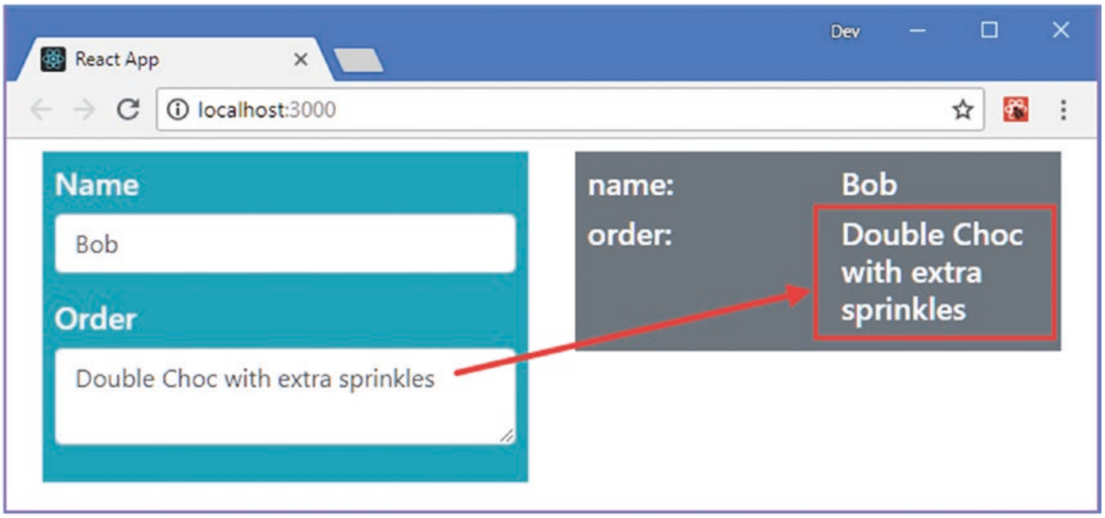

React App
localhost:3000
Name
Bob
Order
Double Choc with extra sprinkles
name:
Bob
order:
Double Choc with extra sprinkles

Figure 15-7. *Using a text area element*

Validating Form Data

Users will enter just about anything into form fields, either because they have made a mistake or because they are trying to skip through the form without filling it in, as noted in the sidebar. Validation checks the data that users provide to ensure that the application has data that it can work with. In the sections that follow, I show you how to perform form validation in a React application.

MINIMIZING THE USE OF FORMS

One reason that users will enter bad data into forms is they don't regard the result as valuable. This can occur when the form interrupts a process that is important to the user with something that is unimportant, such as an intrusive prompt to create an account when reading an article or when the same form is presented at the start of a process that the user performs often.

Validation won't help when the user doesn't value the form because they will simply enter bad data that passes the checks but that is still bad data. If you find that your intrusive prompt for an e-mail address results in a lot of `a@a.com` responses, then you should consider that your users don't think your weekly newsletter is as interesting as you do.

Use forms sparingly and only for processes that the user will regard as useful, such as providing a shipping address. For other forms, find an alternative way to solicit the data from the user that doesn't interrupt their workflow and doesn't annoy them each time they try to perform a task.

When validating forms, the different parts of the validation process can be distributed in a complex hierarchy of HTML and components. Instead of threading props to connect the different parts, I am going to use a context to keep track of validation problems. I added a file called `ValidationContext.js` to the `src` folder with the content shown in Listing 15-14. (Contexts are described in Chapter 14.)

■ **Note** The examples in this section rely on the `validator` package that was added to the project in Listing 15-2. If you skipped over the installation, you should go back and add the package before proceeding with the examples.

Listing 15-14. The Contents of the ValidationContext.js File in the src Folder

```
import React from "react";

export const ValidationContext = React.createContext({
    getMessagesForField: (field) => []
})
```

I am going to store the validation issues for each form element as an array and display messages for each of the issues alongside the element. The context provides access to a function that will return the validation messages for a specific field.

Defining the Validation Rules

Next, I added a file called `validation.js` to the `src` folder and added the code shown in Listing 15-15. This is the code that will validate the form data, using the `validator` package that was installed at the start of the chapter.

Listing 15-15. The Contents of the validation.js File in the src Folder

```
import validator from "validator";

export function ValidateData(data, rules) {
    let errors = {};
    Object.keys(data).forEach(field => {
        if (rules.hasOwnProperty(field)) {
            let fielderrors = [];
            let val = data[field];
            if (rules[field].required && validator.isEmpty(val)) {
                fielderrors.push("Value required");
            }
            if (!validator.isEmpty(data[field])) {
                if (rules[field].minlength
                        && !validator.isLength(val, rules[field].minlength)) {
                    fielderrors.push(`Enter at least ${rules[field].minlength}`
                        + " characters");
                }
                if (rules[field].alpha && !validator.isAlpha(val)) {
                    fielderrors.push("Enter only letters");
                }
                if (rules[field].email && !validator.isEmail(val)) {
                    fielderrors.push("Enter a valid email address");
                }
            }
            if (fielderrors.length > 0) {
                errors[field] = fielderrors;
            }
        }
    })
    return errors;
}
```

The ValidateData function will receive an object whose properties are the form values and an object that specifies the validation rules that are to be applied. The validation package provides methods that can be used to perform a wide range of checks, but I have focused on four validation checks for this example: ensuring that the user has supplied a value, ensuring a minimum length, ensuring a valid e-mail address, and ensuring that only alphabetic characters are used. Table 15-3 describes the methods provided by the validation package that I use in the examples that follow. See https://www.npmjs.com/package/validator for the full range of features provided by the validator package.

Table 15-3. *The validator Methods*

Name	Description
isEmpty	This method returns true if a value is an empty string.
isLength	This method returns true if a value exceeds a minimum length.
isAlpha	This method returns true if a value contains only letters.
isEmail	This method returns true if a value is a valid e-mail address.
isEqual	This method returns true if two values are the same.

Creating the Container Component

To create the validation component, I added a file called FormValidator.js to the src folder and used it to define the component shown in Listing 15-16.

Listing 15-16. The Contents of the FormValidator.js File in the src Folder

```
import React, { Component } from "react";
import { ValidateData } from "./validation";
import { ValidationContext } from "./ValidationContext";

export class FormValidator extends Component {

    constructor(props) {
        super(props);
        this.state = {
            errors: {},
            dirty: {},
            formSubmitted: false,
            getMessagesForField: this.getMessagesForField
        }
    }
    static getDerivedStateFromProps(props, state) {
        return {
            errors: ValidateData(props.data, props.rules)
        };
    }

    get formValid() {
        return Object.keys(this.state.errors).length === 0;
    }

    handleChange = (ev) => {
        let name = ev.target.name;
        this.setState(state => state.dirty[name] = true);
    }

    handleClick = (ev) => {
        this.setState({ formSubmitted: true }, () => {
            if (this.formValid) {
                this.props.submit(this.props.data)
            }
        });
    }

    getButtonClasses() {
        return this.state.formSubmitted && !this.formValid
            ? "btn-danger" : "btn-primary";
    }

    getMessagesForField = (field) => {
        return (this.state.formSubmitted || this.state.dirty[field]) ?
            this.state.errors[field] || [] : []
    }

    render() {
        return <React.Fragment>
            <ValidationContext.Provider value={ this.state }>
                <div onChange={ this.handleChange }>
                    { this.props.children }
                </div>
            </ValidationContext.Provider>

            <div className="text-center">
                <button className={ `btn ${ this.getButtonClasses() }`}
                        onClick={ this.handleClick }
                        disabled={ this.state.formSubmitted && !this.formValid } >
                    Submit
                </button>
            </div>
        </React.Fragment>
    }
}
```

The validation is performed in the getDerivedStateFromProps lifecycle method, which provides a component with a change to make changes to its state based on the props it receives. The component receives a data prop that contains the form data to be validated and a rules prop that defines the validation checks that should be applied and passes these to the ValidateData function defined in Listing 15-15. The result of the ValidateData function is assigned to the state.errors property and is an object with a property for each form field that has validation issues and an array of messages that should be presented to the user.

Form validation should not begin until the user has started to edit a field or has attempted to submit the form. Individual edits are handled by listening for the change event as it bubbles up from the form elements contained by the component, as described in Chapter 12.

```
...
<div onChange={ this.handleChange }>
    { this.props.children }
</div>
...
```

The handleChange method adds the value of the name prop of the change event's target element to the dirty state object (during validation, elements are regarded as *pristine* until the user starts editing, after which they are considered *dirty*). The component presents the user with a button element with a handler that changes the formSubmitted state property when it is clicked. If the button is clicked while there are invalid form elements, then it is disabled until the problems have been resolved and its color is changed to make it obvious that the data cannot be processed.

```
...
<button className={ `btn ${ this.getButtonClasses() }`}
        onClick={ this.handleClick }
        disabled={ this.state.formSubmitted && !this.formValid } >
    Submit
</button>
...
```

If the validation checks produce no errors, then the handleClick method invokes a function prop called submit and uses the validated data as the argument.

Displaying Validation Messages

To display validation messages alongside the form elements, I added a file called ValidationMessage.js to the src folder and used it to define the component shown in Listing 15-17.

Listing 15-17. The Contents of the ValidationMessage.js File in the src Folder

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

export class ValidationMessage extends Component {
    static contextType = ValidationContext;
    render() {
        return this.context.getMessagesForField(this.props.field).map(err =>
            <div className="small bg-danger text-white mt-1 p-1"
                    key={ err } >
                { err }
            </div>
        )
    }
}
```

This component consumes the context provided by the `FormValidator` component and uses it to get the validation messages for a single form field whose name is specified through the `field` prop. This component doesn't have any insight into the type of form element whose validation issues it reports or any knowledge of the overall validity of the form—it just requests the messages and displays them. If there are no messages to be displayed, then no content is rendered.

Applying the Form Validation

The final step is to apply the validation to the form, as shown in Listing 15-18. The `FormValidator` component must be an ancestor to the form fields so it can receive change events from them as they bubble up. It must also be an ancestor to the `ValidationMessage` components so that they have access to the validation messages through the shared context.

Listing 15-18. Applying Validation in the Editor.js File in the src Folder

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

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            name: "",
            email: "",
            order: ""
        }
        this.rules = {
            name: { required: true, minlength: 3, alpha: true },
            email: { required: true, email: true },
            order: { required: true }
        }
    }

    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value });
    }
    render() {
        return <div className="h5 bg-info text-white p-2">
                    <FormValidator data={ this.state } rules={ this.rules }
                            submit={ this.props.submit }>
                        <div className="form-group">
                            <label>Name</label>
                            <input className="form-control"
                                name="name"
                                value={ this.state.name }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="name" />
                        </div>

                        <div className="form-group">
                            <label>Email</label>
                            <input className="form-control"
                                name="email"
                                value={ this.state.email }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="email" />
                        </div>

                        <div className="form-group">
                            <label>Order</label>
                            <textarea className="form-control"
                                name="order"
                                value={ this.state.order }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="order" />
                        </div>
                    </FormValidator>
                </div>
    }
}
```

In addition to applying the validation components, I added an email field and changed the updateFormValue method so that it doesn't automatically send the data for display. The result is shown in Figure 15-8. No validation messages are shown until you start editing a field or click the button, and you can't submit the data until the data you entered meets all of the validation requirements.

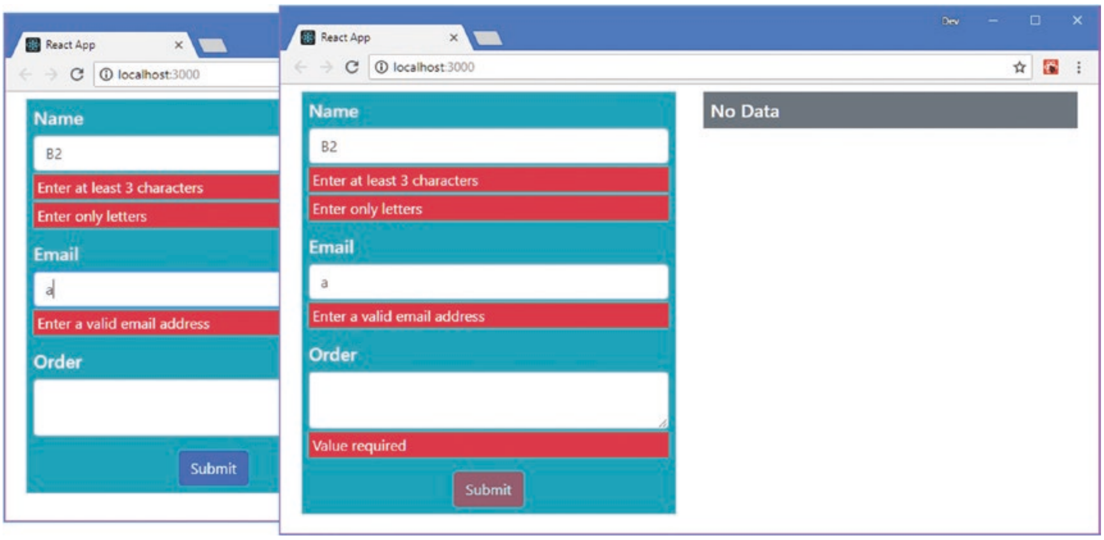


Figure 15-8. *Validating form data*

Validating Other Element and Data Types

Notice that the validation features don't deal directly with the input and `textarea` elements. Instead, the standard state and event features are used to bring the data under React's control, where it is validated and dealt with by components that have no knowledge or interest in where the data came from. This means that once the basic validation features are in place, they can be different types of form element and different types of data. Each project has its own validation requirements, but the examples in the sections that follow demonstrate some of the most commonly required approaches that you can adapt to your own needs.

Ensuring That a Checkbox Is Selected

A common validation requirement is to ensure that the user checks a box to accept terms and conditions. In Listing 15-19, I have added a check to the set of validations that ensures that a value is true, which will be the case when a checkbox element is checked.

Listing 15-19. Adding a Validation Option in the validation.js File in the src Folder

```
import validator from "validator";

export function ValidateData(data, rules) {
    let errors = {};
    Object.keys(data).forEach(field => {
        if (rules.hasOwnProperty(field)) {
            let fielderrors = [];
            let val = data[field];
            if (rules[field].true) {
                if (!val) {
                    fielderrors.push("Must be checked");
                }
            } else {
                if (rules[field].required && validator.isEmpty(val)) {
                    fielderrors.push("Value required");
                }
                if (!validator.isEmpty(data[field])) {
                    if (rules[field].minlength
                            && !validator.isLength(val, rules[field].minlength)) {
                        fielderrors.push(`Enter at least ${rules[field].minlength}`
                            + " characters");
                    }
                    if (rules[field].alpha && !validator.isAlpha(val)) {
                        fielderrors.push("Enter only letters");
                    }
                    if (rules[field].email && !validator.isEmail(val)) {
                        fielderrors.push("Enter a valid email address");
                    }
                }
            }
            if (fielderrors.length > 0) {
                errors[field] = fielderrors;
            }
        }
    })
    return errors;
}
```

The validator package that I am using to perform the validation checks operates only on string values and reports an error if it is asked to check a Boolean. To avoid problems, I have treated the new validation check as a special case that cannot be combined with other rules. In Listing 15-20, I have removed some of the existing form elements and added a checkbox, along with a validation rule that ensures it is checked.

Listing 15-20. Validating a Checkbox in the Editor.js File in the src Folder

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

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            name: "",
            terms: false
        }
        this.rules = {
            name: { required: true, minlength: 3, alpha: true },
            terms: { true: true}
        }
    }
    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value });
    }

    updateFormValueCheck = (event) => {
        this.setState({ [event.target.name]: event.target.checked });
    }

    render() {
        return <div className="h5 bg-info text-white p-2">
                    <FormValidator data={ this.state } rules={ this.rules }
                            submit={ this.props.submit }>
                        <div className="form-group">
                            <label>Name</label>
                            <input className="form-control"
                                name="name"
                                value={ this.state.name }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="name" />
                        </div>

                    <div className="form-group">
                        <div className="form-check">
                            <input className="form-check-input"
                                type="checkbox" name="terms"
                                checked={ this.state.terms }
                                onChange={ this.updateFormValueCheck } />
                            <label className="form-check-label">
                                Agree to terms
                             </label>
                        </div>
                        <ValidationMessage field="terms" />
                    </div>
                    </FormValidator>
                </div>
    }
}
```

The user is presented with a checkbox that must be checked before the form can be submitted, as shown in Figure 15-9.

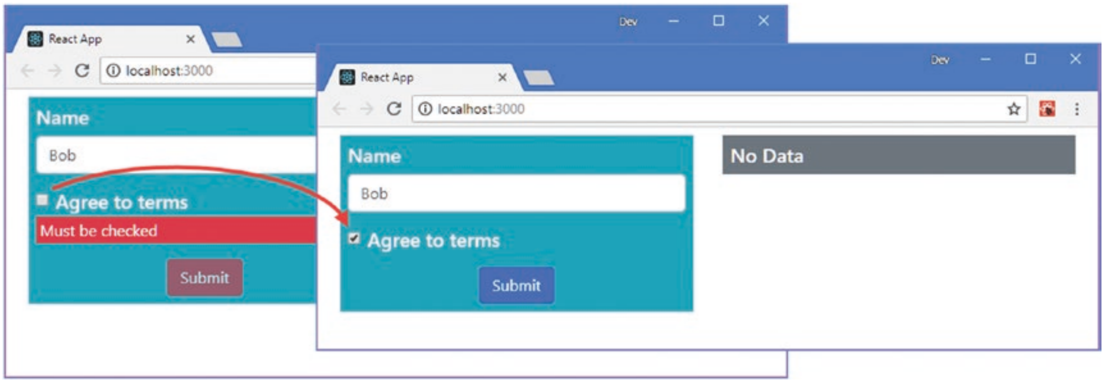


Figure 15-9. *Validating a checkbox*

Ensuring Matching Values

Some values require confirmation in two inputs, such as passwords and e-mail addresses for contact purposes. In Listing 15-21, I have added a validation rule that checks that two values are the same.

Listing 15-21. Ensuring Equal Values in the validation.js File in the src Folder

```
import validator from "validator";

export function ValidateData(data, rules) {
    let errors = {};
    Object.keys(data).forEach(field => {
        if (rules.hasOwnProperty(field)) {
            let fielderrors = [];
            let val = data[field];
            if (rules[field].true) {
                if (!val) {
                    fielderrors.push("Must be checked");
                }
            } else {
                if (rules[field].required && validator.isEmpty(val)) {
                    fielderrors.push("Value required");
                }
                if (!validator.isEmpty(data[field])) {
                    if (rules[field].minlength
                            && !validator.isLength(val, rules[field].minlength)) {
                        fielderrors.push(`Enter at least ${rules[field].minlength}`
                            + " characters");
                    }
                    if (rules[field].alpha && !validator.isAlpha(val)) {
                        fielderrors.push("Enter only letters");
                    }
                    if (rules[field].email && !validator.isEmail(val)) {
                        fielderrors.push("Enter a valid email address");
                    }
                    if (rules[field].equals
                            && !validator.equals(val, data[rules[field].equals])) {
                        fielderrors.push("Values don't match");
                    }
                }
            }
            if (fielderrors.length > 0) {
                errors[field] = fielderrors;
            }
        }
    })
    return errors;
}
```

In Listing 15-22, I have added two input elements to the Editor component and added a validation check to ensure that the user enters the same value in both fields.

Listing 15-22. Adding Related Elements in the Editor.js File in the src Folder

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

export class Editor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            name: "",
            email: "",
            emailConfirm: ""
        }
        this.rules = {
            name: { required: true, minlength: 3, alpha: true },
            email: { required: true, email: true, equals: "emailConfirm"},
            emailConfirm: { required: true, email: true, equals: "email"}
        }
    }

    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value });
    }
    render() {
        return <div className="h5 bg-info text-white p-2">
                    <FormValidator data={ this.state } rules={ this.rules }
                            submit={ this.props.submit }>
                        <div className="form-group">
                            <label>Name</label>
                            <input className="form-control"
                                name="name"
                                value={ this.state.name }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="name" />
                        </div>

                        <div className="form-group">
                            <label>Email</label>
                            <input className="form-control"
                                name="email"
                                value={ this.state.email }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="email" />
                        </div>

                        <div className="form-group">
                            <label>Confirm Email</label>
                            <input className="form-control"
                                name="emailConfirm"
                                value={ this.state.emailConfirm }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="emailConfirm" />
                        </div>
                    </FormValidator>
                </div>
    }
}
```

The result is that the form is valid only when the contents of the `email` and `emailConfirm` fields are the same, as shown in Figure 15-10.

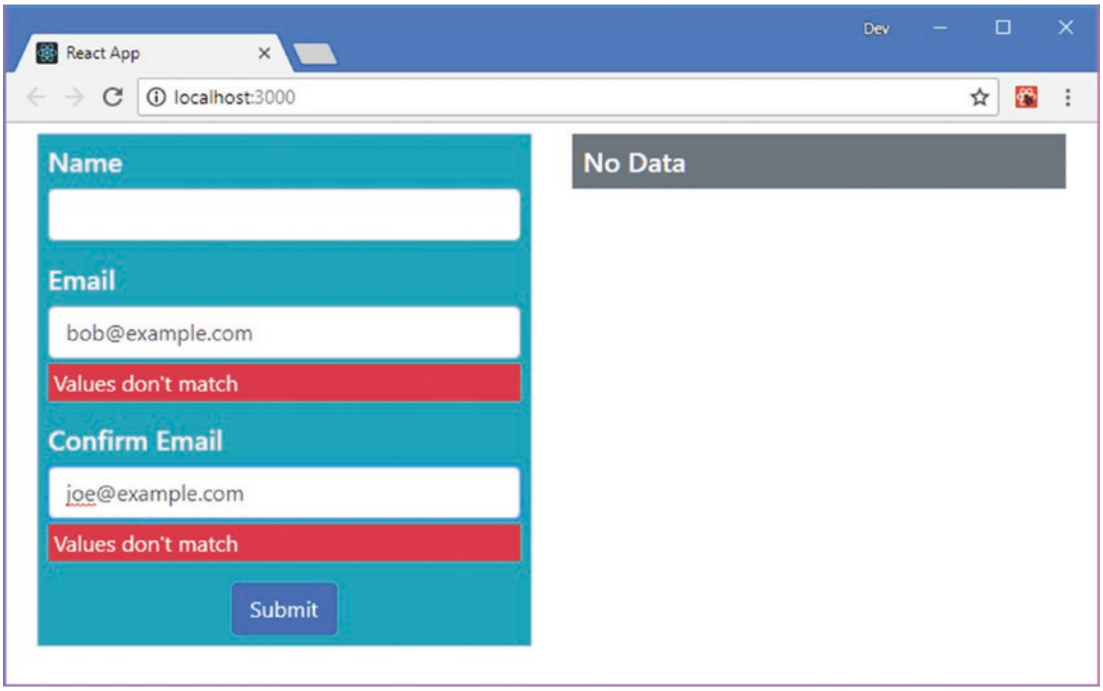


Figure 15-10. *Ensuring matching values*

Performing Whole-Form Validation

Some types of validation cannot be performed on individual values, such as ensuring that combinations of choices are consistent. This sort of validation can be performed only when the user has entered valid data into the form and submitted it, at which point an application can perform a final set of checks before processing the data.

Validation for individual fields can be applied in multiple forms using the same code, while validation on combinations of values tends to be specific to a single form. To avoid mixing general code with form-specific features, I added a file called `wholeFormValidation.js` to the `src` folder and used it to define the validation function shown in Listing 15-23.

Listing 15-23. The Contents of the wholeFormValidation.js File in the src Folder

```
export function ValidateForm(data) {
    let errors = [];
    if (!data.email.endsWith("@example.com")) {
        errors.push("Only example.com users allowed");
    }
    if (!data.email.toLowerCase().startsWith(data.name.toLowerCase())) {
        errors.push("Email address must start with name");
    }
    if (data.name.toLowerCase() === "joe") {
        errors.push("Go away, Joe")
    }
    return errors;
}
```

The ValidateForm function receives the form data and checks that e-mail addresses end with @example.com and that the name property isn't joe and that the email value begins with the name value. In Listing 15-24, I have extended the FormValidator component so that it receives a form validation function as a prop and uses it before submitting the form data.

Listing 15-24. Adding Support for Whole-Form Validation in the FormValidator.js File in the src Folder

```
import React, { Component } from "react";
import { ValidateData } from "./validation";
import { ValidationContext } from "./ValidationContext";

export class FormValidator extends Component {

    constructor(props) {
        super(props);
        this.state = {
            errors: {},
            dirty: {},
            formSubmitted: false,
            getMessagesForField: this.getMessagesForField
        }
    }

    static getDerivedStateFromProps(props, state) {
        state.errors = ValidateData(props.data, props.rules);
        if (state.formSubmitted && Object.keys(state.errors).length === 0) {
            let formErrors = props.validateForm(props.data);
            if (formErrors.length > 0) {
                state.errors.form = formErrors;
            }
        }
        return state;
    }

    get formValid() {
        return Object.keys(this.state.errors).length === 0;
    }

    handleChange = (ev) => {
        let name = ev.target.name;
        this.setState(state => state.dirty[name] = true);
    }
    handleClick = (ev) => {
        this.setState({ formSubmitted: true }, () => {
            if (this.formValid) {
                let formErrors = this.props.validateForm(this.props.data);
                if (formErrors.length === 0) {
                    this.props.submit(this.props.data)
                }
            }
        });
    }

    getButtonClasses() {
        return this.state.formSubmitted && !this.formValid
            ? "btn-danger" : "btn-primary";
    }

    getMessagesForField = (field) => {
        return (this.state.formSubmitted || this.state.dirty[field]) ?
            this.state.errors[field] || [] : []
    }

    render() {
        return <React.Fragment>
            <ValidationContext.Provider value={ this.state }>
                <div onChange={ this.handleChange }>
                    { this.props.children }
                </div>
            </ValidationContext.Provider>

            <div className="text-center">
                <button className={ `btn ${ this.getButtonClasses() }`}
                        onClick={ this.handleClick }
                        disabled={ this.state.formSubmitted && !this.formValid } >
                    Submit
                </button>
            </div>
        </React.Fragment>
    }
}
```

The changes start validating the entire form as soon as the user clicks the Submit button. In Listing 15-25, I have updated the `Editor` component so that it provides the `FormValidator` with a whole-form validation function and defines a new `ValidationMessage` component to display errors that are form-specific.

Listing 15-25. Applying Whole-Form Validation in the Editor.js File in the src Folder

```
import React, { Component } from "react";
import { FormValidator } from "./FormValidator";
import { ValidationMessage } from "./ValidationMessage";
import { ValidateForm } from "./wholeFormValidation";
```

```
export class Editor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            name: "",
            email: "",
            emailConfirm: ""
        }
        this.rules = {
            name: { required: true, minlength: 3, alpha: true },
            email: { required: true, email: true, equals: "emailConfirm"},
            emailConfirm: { required: true, email: true, equals: "email"}
        }
    }

    updateFormValue = (event) => {
        this.setState({ [event.target.name]: event.target.value });
    }

    render() {
        return <div className="h5 bg-info text-white p-2">
                    <FormValidator data={ this.state } rules={ this.rules }
                            submit={ this.props.submit }
                            validateForm={ ValidateForm }>

                        <ValidationMessage field="form" />

                        <div className="form-group">
                            <label>Name</label>
                            <input className="form-control"
                                name="name"
                                value={ this.state.name }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="name" />
                        </div>

                        <div className="form-group">
                            <label>Email</label>
                            <input className="form-control"
                                name="email"
                                value={ this.state.email }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="email" />
                        </div>
                        <div className="form-group">
                            <label>Confirm Email</label>
                            <input className="form-control"
                                name="emailConfirm"
                                value={ this.state.emailConfirm }
                                onChange={ this.updateFormValue } />
                            <ValidationMessage field="emailConfirm" />
                        </div>
                    </FormValidator>
                </div>
    }
}
```

The user is presented with additional validation messages if they try to submit data that doesn't meet the conditions checked in Listing 15-23, as shown in Figure 15-11.

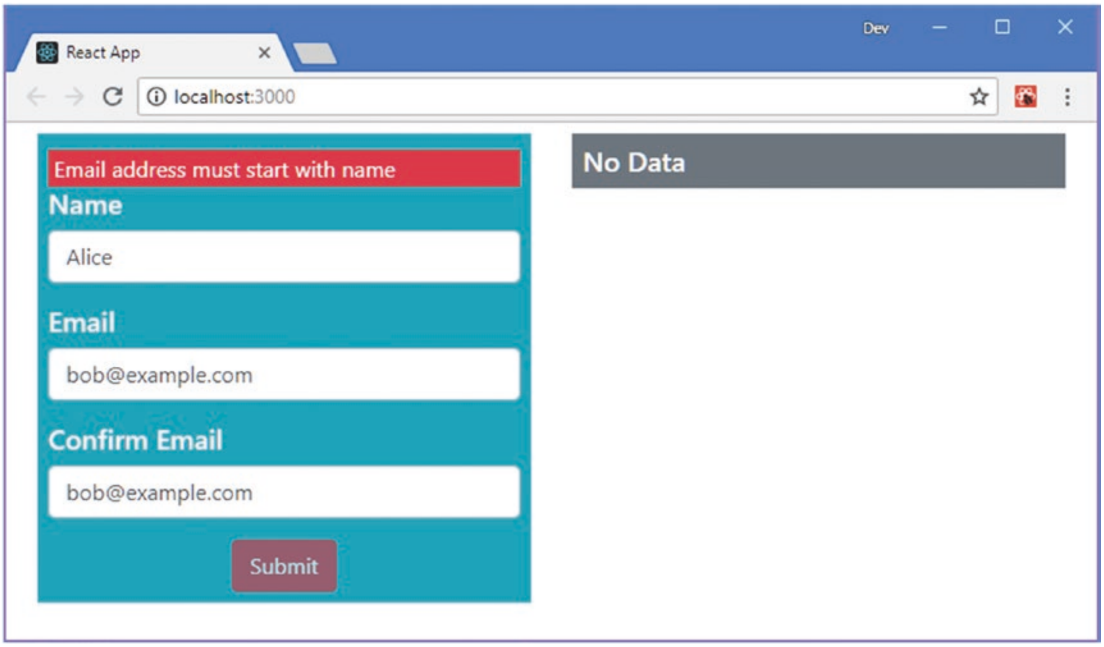


***Figure 15-11.** Performing whole-form validation*

Summary

In this chapter, I showed you how to create controlled components, which are form elements whose content is managed through a state property and whose editing is processed by an event handler. I showed you different types of form element and demonstrated how form data can be validated. Controlled form components are only one type that React supports, and in the next chapter, I introduce the refs feature and explained how uncontrolled form elements can be used.