4

WORKING WITH EVENTS AND STATE

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Add user event handlers (for example, for reacting to button clicks) to React apps.
- Update the UI via a concept called **state**.
- Build real dynamic and interactive UIs (that is, so that they are not static anymore).

INTRODUCTION

In the previous chapters, you learned how to build UIs with the help of React **components**. You also learned about **props—**a concept and feature that enables React developers to build and reuse configurable components.

These are all important React features and building blocks, but with these features alone, you would only be able to build static React apps (that is, web apps that never change). You would not be able to change or update the content on the screen if you only had access to those features. You also would not be able to react to any user events and update the UI in response to such events (for instance, to show an overlay window upon a button click).

Put in other words, you would not be able to build real websites and web applications if you were limited to just components and props.

Therefore, in this chapter, a brand-new concept is introduced: state. State is a React feature that allows developers to update internal data and trigger a UI update based on such data adjustments. In addition, you will learn how to react (no pun intended) to user events such as button clicks or text being entered into input fields.

WHAT'S THE PROBLEM?

As outlined previously, at this point in the book, there is a problem with all React apps and sites you might be building: they're static. The UI can't change.

To understand this issue a bit better, take a look at a typical React component, as you are able to build it up to this point in the book:

```
function EmailInput() {
  return (
    <div>
      <input placeholder="Your email" type="email" />
      <p>The entered email address is invalid.</p>
    </div>
  );
};
```

This component might look strange though. Why is there a **<p>** element that informs the user about an incorrect email address?

Well, the goal might be to show that paragraph only if the user *did* enter an incorrect email address. That is to say, the web app should wait for the user to start typing and evaluate the user input once the user is done typing (that is, once the input loses focus). Then, the error message should be shown if the email address is considered invalid (for example, an empty input field or a missing `@` symbol).

But at the moment, with the React skills picked up thus far, this is something you would not be able to build. Instead, the error message would always be shown since there is no way of changing it based on user events and dynamic conditions. In other words, this React app is a static app, not dynamic. The UI can't change.

Of course, changing UIs and dynamic web apps are things you might want to build. Almost every website that exists contains some dynamic UI elements and features. Therefore, that's the problem that will be solved in this chapter.

HOW NOT TO SOLVE THE PROBLEM

How could the component shown previously be made more dynamic?

The following is one solution you could come up with. (*Spoiler*, *the code won't work, so you don't need to try running it!*)

```
function EmailInput() {
  return (
    <div>
      <input placeholder="Your email" type="email" />
      <p></p>
    </div>
  );
};

const input = document.querySelector('input');
const errorParagraph = document.querySelector('p');

function evaluateEmail(event) {
  const enteredEmail = event.target.value;
  if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
    errorParagraph.textContent = ' The entered email address is
invalid.';
  } else {
```

```
    errorParagraph.textContent = '';
  }
};

input.addEventListener('blur', evaluateEmail);
```

This code won't work, because you can't select React-rendered DOM elements from inside the same component file this way. This is just meant as a dummy example of how you could try to solve this. That being said, you could put the code below the component function some place where it does execute successfully (for example, into a **`setTimeout()`** callback that fires after a second, allowing the React app to render all elements onto the screen).

Put in the right place, this code will add the email validation behavior described earlier in this chapter. Upon the built-in **`blur`** event, the **`evaluateEmail`** function is triggered. This function receives the **`event`** object as an argument (automatically, by the browser), and therefore the **`evaluateEmail`** function is able to parse the entered value from that **`event`** object via **`event.target.value`**. The entered value can then be used in an **`if`** check to conditionally display or remove the error message.

> **NOTE**
>
> All the preceding code that deals with the **`blur`** event (such as **`addEventListener`**) and the **`event`** object, including the code in the **`if`** check, is standard JavaScript code. It is not specific to React in any way.
>
> If you find yourself struggling with this non-React code, it's strongly recommended that you dive into more vanilla JavaScript resources (such as the guides on the MDN website at https://developer.mozilla.org/en-US/docs/Web/JavaScript) first.

But what's wrong with this code, if it would work in some places of the overall application code?

It's imperative code! That means you are writing down step-by-step instructions on what the browser should do. You are not declaring the desired end state; you are instead describing a way of getting there. And it's not using React.

Keep in mind that React is all about controlling the UI and that writing React code is about writing declarative code—instead of imperative code. Revisit *Chapter 2*, *What Is React?*, if that sounds brand new to you.

You could achieve your goal by introducing this kind of code, but you would be working against React and its philosophy (React's philosophy being that you declare your desired end state(s) and let React figure out how to get there). A clear indicator of this is the fact that you would be forced to find the right place for this kind of code in order for it to work.

This is not a philosophical problem, and it's not just some weird hard rule that you should follow. Instead, by working against React like this, you will make your life as a developer unnecessarily hard. You are neither using the tools React gives you nor letting React figure out how to achieve the desired (UI) state.

That does not just mean that you spend time on solving problems you wouldn't have to solve. It also means that you're passing up possible optimizations React might be able to perform under the hood. Your solution is very likely not just leading to more work (that is, more code) for you; it also might result in a buggy result that could also suffer from suboptimal performance.

The example shown previously is a simple one. Think about more complex websites and web apps, such as online shops, vacation rental websites, or web apps such as Google Docs. There, you might have dozens or hundreds of (dynamic) UI features and elements. Managing them all with a mixture of React code and standard vanilla JavaScript code will quickly become a nightmare. Again, refer to *Chapter 2* of this book to understand the merits of React.

A BETTER INCORRECT SOLUTION

The naïve approach discussed previously doesn't work well. It forces you to figure out how to make the code run correctly (for example, by wrapping parts of it in some **`setTimeout()`** call to defer execution) and leads to your code being scattered all over the place (that is, inside of React component functions, outside of those functions, and maybe also in totally unrelated files). How about a solution that embraces React, like this:

```
function EmailInput() {
  let errorMessage = '';

  function evaluateEmail(event) {
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
```

```
      errorMessage = ' The entered email address is invalid.';
    } else {
      errorMessage = '';
    }
  };

  const input = document.querySelector('input');

  input.addEventListener('blur', evaluateEmail);

  return (
    <div>
      <input placeholder="Your email" type="email" />
      <p>{errorMessage}</p>
    </div>
  );
};
```

This code again would not work (even though it's technically valid JavaScript code). Selecting JSX elements doesn't work like this. It doesn't work because **`document.querySelector('input')`** executes before anything is rendered to the DOM (when the component function is executed for the first time). Again, you would have to delay the execution of that code until the first render cycle is over (you would therefore be once again working against React).

But even though it still would not work, it's closer to the correct solution.

It's closer to the ideal implementation because it embraces React way more than the first attempted solution did. All the code is contained in the component function to which it belongs. The error message is handled via an **`errorMessage`** variable that is output as part of the JSX code.

The idea behind this possible solution is that the React component that controls a certain UI feature or element is also responsible for its state and events. You might identify two important keywords of this chapter here!

This approach is definitely going in the right direction, but it still wouldn't work for two reasons:

- Selecting the JSX **`<input>`** element via **`document.querySelector('input')`** would fail.
- Even if the input could be selected, the UI would not update as expected.

These two problems will be solved next—finally leading to an implementation that embraces React and its features. The upcoming solution will avoid mixing React and non-React code. As you will see, the result will be easier code where you have to do less work (that is, write less code).

PROPERLY REACTING TO EVENTS

Instead of mixing imperative JavaScript code such as **`document.querySelector('input')`** with React-specific code, you should fully embrace React and its features.

And since listening to events and triggering actions upon events is an extremely common requirement, React has a built-in solution. You can attach event listeners directly to the JSX elements to which they belong.

The preceding example would be rewritten like this:

```
function EmailInput() {
  let errorMessage = '';

  function evaluateEmail(event) {
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
      errorMessage = 'The entered email address is invalid.';
    } else {
      errorMessage = '';
    }
  };

  return (
    <div>
      <input placeholder="Your email" type="email" onBlur={evaluateEmail}
/>
      <p>{errorMessage}</p>
    </div>
  );
};
```

The **`onBlur`** prop was added to the built-in input element. This prop is made available by React, just as all these base HTML elements (such as **`<input>`** and **`<p>`**) are made available as components by React. In fact, all these built-in HTML components come with their standard HTML attributes as React props (plus some extra props, such as the **`onBlur`** event handling prop).

React exposes all standard events that can be connected to DOM elements as **`onXYZ`** props (where **`XYZ`** is the event name, such as **`blur`** or **`click`**, starting with a capital character). You can react to the **`blur`** event by adding the **`onBlur`** prop. You could listen to a **`click`** event via the **`onClick`** prop. You get the idea.

> **NOTE**
>
> For more information on standard events see https://developer.mozilla.org/en-US/docs/Web/Events#event_listing.

These props require values to fulfill their job. To be precise, they need a pointer to the function that should be executed when the event occurs. In the preceding example, the **`onBlur`** prop receives a pointer to the **`evaluateEmail`** function as a value.

> **NOTE**
>
> There's a subtle difference between **`evaluateEmail`** and **`evaluateEmail()`**. The first is a pointer to the function; the second actually executes the function (and yields the return value, if any). Again, this is not something specific to React but a standard JavaScript concept. If it's not clear, this resource explains it in greater detail: https://developer.mozilla.org/en-US/docs/Web/Events#event_listing.

By using these event props, the preceding example code will now finally execute without throwing any errors. You could verify this by adding a **`console.log('Hello');`** statement inside the **`evaluateEmail`** function. This will display the text **`'Hello'`** in the console of your browser developer tools, whenever the input loses focus:

```
function EmailInput() {
  let errorMessage = '';

  function evaluateEmail(event) {
    console.log('Hello');
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
      errorMessage = 'The entered email address is invalid.';
    } else {
```

```
      errorMessage = '';
    }
  };

  return (
    <div>
      <input placeholder="Your email" type="email" onBlur={evaluateEmail}
/>
      <p>{errorMessage}</p>
    </div>
  );
};
```

In the browser console, this looks as follows:

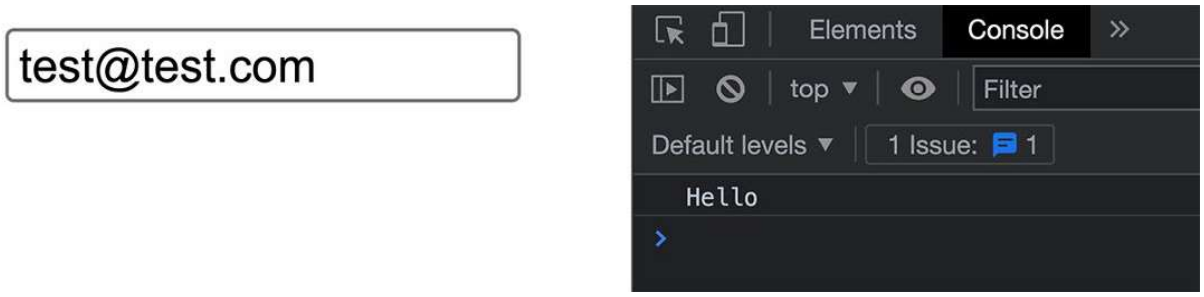

Figure 4.1: Displaying some text in the browser console upon removing focus from the input field

This is definitely one step closer to the best possible implementation, but it also still won't produce the desired result of updating the page content dynamically.

UPDATING STATE CORRECTLY

By now, you understand how to correctly set up event listeners and execute functions upon certain events. What's missing is a feature that forces React to update the visible UI on the screen and the content that is displayed to the app users.

That's where React's **state** concept comes into play. Like props, state is a key concept of React, but whereas props are about receiving external data inside a component, state is about managing and updating **internal data**. And, most importantly, whenever such state is updated, React goes ahead and updates the parts of the UI that are affected by the state change.

Here's how state is used in React (of course, the code will then be explained in detail afterward):

```
import { useState } from 'react';
```

```
function EmailInput() {
  const [errorMessage, setErrorMessage] = useState('');

  function evaluateEmail(event) {
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
      setErrorMessage('The entered email address is invalid.');
    } else {
      setErrorMessage('');
    }
  };

  return (
    <div>
      <input placeholder="Your email" type="email" onBlur={evaluateEmail}
/>
      <p>{errorMessage}</p>
    </div>
  );
};
```

Compared to the example code discussed earlier in this chapter, this code doesn't look much different. But there is a key difference: the usage of the **`useState()`** Hook.

Hooks are another key concept of React. These are special functions that can only be used inside of React components (or inside of other Hooks, as will be covered in *Chapter 11*, *Building Custom React Hooks*). Hooks add special features and behaviors to the React components in which they are used. For example, the **`useState()`** Hook allows a component (and therefore, implicitly React) to set and manage some state that is tied to this component. React provides various built-in Hooks, and they are not all focused on state management. You will learn about other Hooks and their purposes throughout this book.

The **`useState()`** Hook is an extremely important and commonly used Hook as it enables you to manage data inside a component, which, when updated, tells React to update the UI accordingly.

That is the core idea behind state management and this state concept: state is data, which, when changed, should force React to re-evaluate a component and update the UI if needed.

Using Hooks such as **`useState()`** is pretty straightforward: You import them from **`'react'`** and you then call them like a function inside your component function. You call them like a function because, as mentioned, React Hooks are functions—just special functions (from React's perspective).

A CLOSER LOOK AT USESTATE()

How exactly does the **`useState()`** Hook work and what does it do internally?

By calling **`useState()`** inside a component function, you register some data with React. It's a bit like defining a variable or constant in vanilla JavaScript. But there is something special: React will track the registered value internally, and whenever you update it, React will re-evaluate the component function in which the state was registered.

React does this by checking whether the data used in the component changed. Most importantly, React validates whether the UI needs to change because of changed data (for example, because a value is output inside the JSX code). If React determines that the UI needs to change, it goes ahead and updates the real DOM in the places where an update is needed (for example, changing some text that's displayed on the screen). If no update is needed, React ends the component re-evaluation without updating the DOM.

React's internal workings will be discussed in great detail later in the book, in *Chapter 9*, *Behind the Scenes of React and Optimization Opportunities*).

The entire process starts with calling **`useState()`** inside a component. This creates a state value (which will be stored and managed by React) and ties it to a specific component. An initial state value is registered by simply passing it as a parameter value to **`useState()`**. In the preceding example, an empty string (**`''`**) is registered as a first value:

```
const [errorMessage, setErrorMessage] = useState('');
```

As you can see in the example, **`useState()`** does not just accept a parameter value. It also returns a value: an array with exactly two elements.

The preceding example uses **array destructuring**, which is a standard JavaScript feature that allows developers to retrieve values from an array and immediately assign them to variables or constants. In the example, the two elements that make up the array returned by **`useState()`** are pulled out of that array and stored in two constants (**`errorMessage`** and **`setErrorMessage`**). You don't have to use array destructuring when working with React or **`useState()`**, though.

You could also write the code like this instead:

```
const stateData = useState('');
const errorMessage = stateData[0];
const setErrorMessage = stateData[1];
```

This works absolutely fine, but when using array destructuring, the code stays a bit more concise. That's why you typically see the syntax using array destructuring when browsing React apps and examples. You also don't have to use constants; variables (via **`let`**) would be fine as well. As you will see throughout this chapter and the rest of the book, though, the variables won't be reassigned, and so using constants makes sense (but it is not required in any way).

> **NOTE**
>
> If array destructuring or the difference between variables and constants sounds brand new to you, it's strongly recommended that you refresh your JavaScript basics before progressing with this book. As always, MDN provides great resources for that (see http://packt.link/3B8Ct for array destructuring, https://packt.link/hGjqL for information on the **`let`** variable, and https://packt.link/TdPPS for guidance on the use of **`const`**.)

As mentioned before, **`useState()`** returns an array with exactly two elements. It will always be exactly two elements—and always exactly the same kind of elements. The first element is always the current state value, and the second element is a function that you can call to set the state to a new value.

But how do these two values (the state value and the state-updating function) work together? What does React do with them internally? And how are these two array elements used (by React) to update the UI?

A LOOK UNDER THE HOOD OF REACT

React manages the state values for you, in some internal storage that you, the developer, can't directly access. Since you often do need access to a state value (for instance, some entered email address, as in the preceding example), React provides a way of reading state values: the first element in the array returned by **`useState()`**. The first element of the returned array holds the current state value. You can therefore use this element in any place where you need to work with the state value (for example, in the JSX code to output it there).

In addition, you often also need to update the state—for example, because a user entered a new email address. Since you don't manage the state value yourself, React gives you a function that you can call to inform React about the new state value. That's the second element in the returned array.

In the example shown before, you call **`setErrorMessage('Error!')`** to set the **`errorMessage`** state value to a new string (**`'Error!'`**).

But why is this managed like this? Why not just use a standard JavaScript variable that you can assign and reassign as needed?

Because React must be informed whenever there's a state that impacts the UI changes. Otherwise, the visible UI doesn't change at all, even in cases where it should. React does not track regular variables and changes to their values, and so they have no influence on the state of the UI.

The state-updating function exposed by React (that second array element returned by **`useState()`**) *does* trigger some internal UI-updating effect though. This state-updating function does more than set a new value; it also informs React that a state value changed, and that the UI might therefore be in need of an update.

So, whenever you call **`setErrorMessage('Error!')`**, React does not just update the value that it stores internally; it also checks the UI and updates it when needed. UI updates can involve anything from simple text changes up to the complete removal and addition of various DOM elements. Anything is possible there!

React determines the new target UI by rerunning (also called re-evaluating) any component functions that are affected by a state change. That includes the component function that executed the **`useState()`** function that returned the state-updating function that was called. But it also includes any child components, since an update in a parent component could lead to new state data that's also used by some child components (the state value could be passed to child components via props).

It's important to understand and keep in mind that React will re-execute (re-evaluate) a component function if a state-updating function was called in the component function or some parent component function. Because this also explains why the state value returned by **`useState()`** (that is, the first array element) can be a constant, even though you can assign new values by calling the state-updating function (the second array element). Since the entire component function is re-executed, **`useState()`** is also called again (because all the component function code is executed again) and hence a new array with two new elements is returned by React. And the first array element is still the current state value.

However, as the component function was called because of a state update, the current state value now is the updated value.

This can be a bit tricky to wrap your head around, but it is how React works internally. In the end, it's just about component functions being called multiple times by React. Just as any JavaScript function can be called multiple times.

NAMING CONVENTIONS

The **`useState()`** Hook is typically used in combination with array destructuring, like this:

```
const [enteredEmail, setEnteredEmail] = useState('');
```

But when using array destructuring, the names of the variables or constants (**`enteredEmail`** and **`setEnteredEmail`**, in this case) are up to you, the developer. Therefore, a valid question is how you should name these variables or constants. Fortunately, there is a clear convention when it comes to React and **`useState()`**, and these variable or constant names.

The **first element** (that is, the current state value) should be named such that it describes what the state value is all about. Examples would be **`enteredEmail`**, **`userEmail`**, **`providedEmail`**, just **`email`**, or similar names. You should avoid generic names such as **`a`** or **`value`** or misleading names such as **`setValue`** (which sounds like it is a function—but it isn't).

The **second element** (that is, the state-updating function) should be named such that it becomes clear that it is a function and that it does what it does. Examples would be **`setEnteredEmail`** or **`setEmail`**. In general, the convention for this function is to name it **`setXYZ`**, where **`XYZ`** is the name you chose for the first element, the current state value variable. (Note, though, that you start with an uppercase character, as in **`setEnteredEmail`**, not **`setenteredEmail`**.)

ALLOWED STATE VALUE TYPES

Managing entered email addresses (or user input in general) is indeed a common use case and example for working with state. But you're not limited to this scenario and value type.

In the case of entered user input, you will often deal with string values such as email addresses, passwords, blog posts, or similar values. But any valid JavaScript value type can be managed with the help of **`useState()`**. You could, for example, manage the total sum of multiple shopping cart items—that is, a number—or a Boolean value (for example, *"did a user confirm the terms of use?"*).

Besides managing primitive value types, you can also store and update reference value types such as objects and arrays.

> **NOTE**
>
> If the difference between primitive and reference value types is not entirely clear, it's strongly recommended that you dive into this core JavaScript concept before proceeding with this book through the following link: https://academind.com/tutorials/reference-vs-primitive-values.

React gives you the flexibility of managing all these value types as state. You can even switch the value type at runtime (just as you can in vanilla JavaScript). It is absolutely fine to store a number as the initial state value and update it to a string at a later point in time.

Just as with vanilla JavaScript, you should, of course, ensure that your program deals with this behavior appropriately, though there's nothing technically wrong with switching types.

WORKING WITH MULTIPLE STATE VALUES

When building anything but very simple web apps or UIs, you will need multiple state values. Maybe users can not only enter their email but also a username or their address. Maybe you also need to track some error state or save shopping cart items. Maybe users can click a "Like" button whose state should be saved and reflected in the UI. There are many values that change frequently and whose changes should be reflected in the UI.

Consider this concrete scenario: you have a component that needs to manage both the value entered by a user into an email input field and the value that was inserted into a password field. Each value should be captured once a field loses focus.

Since you have two input fields that hold different values, you have two state values: the entered email and the entered password. Even though you might use both values together at some point (for example, to log a user in), the values are not provided simultaneously. In addition, you might also need every value to stand alone, since you use it to show potential error messages (for example, *"password too short"*) while the user is entering data.

Scenarios like this are very common, and therefore, you can also manage multiple state values with the **`useState()`** Hook. There are two main ways of doing that:

1. Use multiple **state slices** (multiple state values)
2. Using one single, *big* state object

USING MULTIPLE STATE SLICES

You can manage multiple state values (also often called **state slices**), by simply calling **`useState()`** multiple times in your component function.

For the example described previously, a (simplified) component function could look like this:

```
function LoginForm() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [enteredPassword, setEnteredPassword] = useState('');

  function emailEnteredHandler(event) {
    setEnteredEmail(event.target.value);
  };

  function passwordEnteredHandler(event) {
    setEnteredPassword(event.target.value);
  };

  // Below, props are split across multiple lines for better readability
  // This is allowed when using JSX, just as it is allowed in standard
HTML
  return (
    <form>
      <input
        type="email"
        placeholder="Your email"
        onBlur={emailEnteredHandler} />
      <input
        type="password"
        placeholder="Your password"
        onBlur={passwordEnteredHandler} />
    </form>
  );
};
```

In this example, two state slices are managed by calling **`useState()`** twice. Therefore, React registers and manages two state values internally. These two values can be read and updated independently from each other.

> **NOTE**
>
> In the example, the functions that are triggered upon events end with **`handler`** (**`emailEnteredHandler`** and **`passwordEnteredHandler`**). This is a convention used by some React developers. Event handler functions end with ...**`handler`** to make it clear that these functions are executed upon certain (user) events. This is not a convention you have to follow. The functions could have also been named **`updateEmail`**, **`updatePassword`**, **`handleEmailUpdate`**, **`handlePasswordUpdate`**, or anything else. If the name is meaningful and follows some stringent convention, it's a valid choice.

You can register as many state slices (by calling **`useState()`** multiple times) as you need in a component. You could have one state value, but you could also have dozens or even hundreds. Typically, though, you will only have a couple of state slices per component since you should try to split bigger components (which might be doing lots of different things) into multiple smaller components to keep them manageable.

The advantage of managing multiple state values like this is that you can update them independently. If the user enters a new email address, you only need to update that email state value. The password state value doesn't matter for your purposes.

A possible disadvantage could be that multiple state slices—and therefore multiple **`useState()`** calls—leads to lots of lines of code that might bloat your component. As mentioned before though, you typically should try to break up big components (that handle lots of different slices of state) into multiple smaller components anyways.

Still, there is an alternative to managing multiple state values like this: you can also manage a single, *merged* state value object.

MANAGING COMBINED STATE OBJECTS

Instead of calling **`useState()`** for every single state slice, you can go for one *big* state object that combines all the different state values:

```
function LoginForm() {
  const [userData, setUserData] = useState({
    email: '',
    password: ''
  });

  function emailEnteredHandler(event) {
    setUserData({
      email: event.target.value,
      password: userData.password
    });
  };

  function passwordEnteredHandler(event) {
    setUserData({
      email: userData.email,
      password: event.target.value
    });
  };

  // ... code omitted, because the returned JSX code is the same as
before
};
```

In this example, **`useState()`** is called only once, and the initial value passed to **`useState()`** is a JavaScript object. The object contains two properties: **`email`** and **`password`**. The property names are up to you, but they should describe the values that will be stored in the properties.

`useState()` still returns an array with exactly two elements. That the initial value is an object does not change anything about that. The first element of the returned array is now just an object instead of a string (as it was in the examples shown earlier). As mentioned before, any valid JavaScript value type can be used when working with **`useState()`**. Primitive value types such as strings or numbers can be used just as you would reference value types such as objects or arrays (which, technically, are objects of course).

The state-updating function (**`setUserData`**, in the preceding example) is still a function created by React that you can call to set the state to a new value. And you wouldn't have to set it to an object again, though that is typically the default. You don't change value types when updating the state, unless you have a good reason for doing so (though, technically, you are allowed to switch to a different type any time).

> **NOTE**
>
> In the preceding example, the way the state-updating function is used is not entirely correct. It would work but it does violate recommended best practices. You will learn later in this chapter why this is the case and how you should use the state-updating function instead.

When managing state objects as shown in the preceding example, there's one crucial thing you should keep in mind: you must always set all properties the object contains, even the ones that didn't change. This is required because, when calling the state-updating function, you *tell* React which new state value should be stored internally.

Thus, any value you pass as an argument to the state-updating function will overwrite the previously stored value. If you provide an object that contains only the properties that changed, all other properties will be lost since the previous state object is replaced by the new one, which contains fewer properties.

This is a common pitfall and therefore something you must pay attention to. For this reason, in the example shown previously, the property that is not changed is set to the previous state value—for example, **`email: userData.email`**, where **`userData`** is the current state snapshot and the first element of the array returned by **`useState()`**, while setting **`password`** to **`event.target.value`**.

It is totally up to you whether you prefer to manage one state value (that is, an object grouping together multiple values) or multiple state slices (that is, multiple **`useState()`** calls) instead. There is no right or wrong way and both approaches have their advantages and disadvantages.

However, it is worth noting that you should typically try to break up *big* components into smaller ones. Just as regular JavaScript functions shouldn't do too much work in a single function (it is considered a good practice to have separate functions for different tasks), components should focus on one or only a few tasks per component as well. Instead of having a huge **`<App />`** component that handles multiple forms, user authentication, and a shopping cart directly in one component, it would be preferable to split the code of that component into multiple smaller components that are then combined together to build the overall app.

When following that advice, most components shouldn't have too much state to manage anyway, since managing many state values is an indicator of a component doing *too much work*. That's why you might end up using a few state slices per component, instead of large state objects.

UPDATING STATE BASED ON PREVIOUS STATE CORRECTLY

When learning about objects as state values, you learned that it's easy to accidentally overwrite (and lose) data because you might set the new state to an object that contains only the properties that changed—not the ones that didn't. That's why, when working with objects or arrays as state values, it's important to always add the existing properties and elements to the new state value.

And, in general, setting a state value to a new value that is (at least partially) based on the previous state is a common task. You might set **`password`** to **`event.target.value`** but also set **`email`** to **`userData.email`** to ensure that the stored email address is not lost due to updating a part of the overall state (that is, because of updating the password to the newly entered value).

That's not the only scenario where the new state value could be based on the previous one though. Another example would be a **`counter`** component—for example, a component like this:

```
function Counter() {
  const [counter, setCounter] = useState(0);

  function incrementCounterHandler() {
    setCounter(counter + 1);
  };

  return (
    <>
      <p>Counter Value: {counter}</p>
```

```
      <button onClick={incrementCounterHandler}>Increment</button>
    </>
  );
};
```

In this example, a **`click`** event handler is registered for **`<button>`** (via the **`onClick`** prop). Upon every click, the counter state value is incremented by **`1`**.

This component would work, but the code shown in the example snippet is actually violating an important best practice and recommendation: state updates that depend on some previous state should be performed with the help of a function that's passed to the state-updating function. To be precise, the example should be rewritten like this:

```
function Counter() {
  const [counter, setCounter] = useState(0);

  function incrementCounterHandler() {
    setCounter(function(prevCounter) { return prevCounter + 1; });
    // alternatively, JS arrow functions could be used:
    // setCounter(prevCounter => prevCounter + 1);
  };

  return (
    <>
      <p>Counter Value: {counter}</p>
      <button onClick={incrementCounterHandler}>Increment</button>
    </>
  );
};
```

This might look a bit strange. It might seem like a function is now passed as the new state value to the state-updating function (that is, the number stored in **`counter`** is replaced with a function). But indeed, that is not the case.

Technically, a function *is* passed as an argument to the state-updating function, but React won't store that function as the new state value. Instead, when receiving a function as a new state value in the state-updating function, React will call that function for you and pass the latest state value to that function. Therefore, you should provide a function that accepts at least one parameter: the previous state value. This value will be passed into the function automatically by React, when React executes the function (which it will do internally).

The function should then also return a value—the new state value that should be stored by React. And since the function receives the previous state value, you can now derive the new state value based on the previous state value (for example, by adding the number 1 to it, but any operation could be performed here).

Why is this required, if the app worked fine before this change as well? It's required because in more complex React applications and UIs, React could be processing many state updates simultaneously—potentially triggered from different sources at different times.

When *not* using the approach discussed in the last paragraphs, the order of state updates might not be the expected one and bugs could be introduced into the app. Even if you know that your use case won't be affected and the app does its job without issue, it is recommended to simply adhere to the discussed best practice and pass a function to the state-updating function if the new state depends on the previous state.

With this newly gained knowledge in mind, take another look at an earlier code example:

```
function LoginForm() {
  const [userData, setUserData] = useState({
    email: '',
    password: ''
  });

  function emailEnteredHandler(event) {
    setUserData({
      email: event.target.value,
      password: userData.password
    });
  };

  function passwordEnteredHandler(event) {
    setUserData({
      email: userData.email,
      password: event.target.value
    });
```

```
  };

  // ... code omitted, because the returned JSX code is the same as
before
};
```

Can you spot the error in this code?

It's not a technical error; the code will execute fine, and the app will work as expected. But there is a problem with this code nonetheless. It violates the discussed best practice. In the code snippet, the state in both handler functions is updated by referring to the current state snapshot via **`userData.password`** and **`userData.email`**, respectively.

The code snippet should be rewritten like this:

```
function LoginForm() {
  const [userData, setUserData] = useState({
    email: '',
    password: ''
  });

  function emailEnteredHandler(event) {
    setUserData(prevData => ({
      email: event.target.value,
      password: prevData.password
    }));
  };

  function passwordEnteredHandler(event) {
    setUserData(prevData => ({
      email: prevData.email,
      password: event.target.value
    }));
  };

  // ... code omitted, because the returned JSX code is the same as
before
  // userData is not actively used here, hence you could get a warning
  // regarding that. Simply ignore it or start using userData
  // (e.g., via console.log(userData))
};
```

By passing an arrow function as an argument to **`setUserData`**, you allow React to call that function. React will do this automatically (that is, if it receives a function in this place, React will call it) and it will provide the previous state (**`prevState`**) automatically. The returned value (the object that stores the updated **`email`** or **`password`** and the currently stored **`email`** or **`password`**) is then set as the new state. The result, in this case, might be the same as before, but now the code adheres to recommended best practices.

> **NOTE**
>
> In the previous example, an arrow function was used instead of a "regular" function. Both approaches are fine though. You can use either of the two function types; the result will be the same.

In summary, you should always pass a function to the state-updating function if the new state depends on the previous state. Otherwise, if the new state depends on some other value (for instance, user input), directly passing the new state value as a function argument is absolutely fine and recommended.

TWO-WAY BINDING

There is one special usage of React's state concept that is worth discussing: **two-way binding**.

Two-way binding is a concept that is used if you have an input source (typically an **`<input>`** element) that sets some state upon user input (for instance, upon the **`change`** event) and outputs the input at the same time.

Here's an example:

```
function NewsletterField() {
  const [email, setEmail] = useState('');

  function changeEmailHandler(event) {
    setEmail(event.target.value);
  };

  return (
    <>
      <input
        type="email"
```

```
        placeholder="Your email address"
        value={email}
        onChange={changeEmailHandler} />
    </>
  );
};
```

Compared to the other code snippets and examples, the difference here is that the component does not just store the user input (upon the **`change`** event, in this case) but that the entered value is also output in the **`<input>`** element (via the default **`value`** prop) thereafter.

This might look like an infinite loop, but React deals with this and ensures that it doesn't become one. Instead, this is what's commonly referred to as two-way binding as a value is both set and read from the same source.

You may wonder why this is being discussed here, but it is important to know that it is perfectly valid to write code like this. And this kind of code could be necessary if you don't just want to set a value (in this case, the **`email`** value) upon user input in the **`<input>`** field but also from other sources. For example, you might have a button in the component that, when clicked, should clear the entered email address.

It might look like this:

```
function NewsletterField() {
  const [email, setEmail] = useState('');

  function changeEmailHandler(event) {
    setEmail(event.target.value);
  };

  function clearInputHandler() {
    setEmail(''); // reset email input (back to an empty string)
  };

  return (
    <>
      <input
        type="email"
        placeholder="Your email address"
        value={email}
        onChange={changeEmailHandler} />
      <button onClick={clearInputHandler}>Reset</button>
```

```
    </>
  );
};
```

In this updated example, the **`clearInputHandler`** function is executed when **`<button>`** is clicked. Inside the function, the **`email`** state is set back to an empty string. Without two-way binding, the state would be updated, but the change would not be reflected in the **`<input>`** element. There, the user would still see their last input. The state reflected on the UI (the website) and the state managed internally by React would be different—a bug you absolutely must avoid.

DERIVING VALUES FROM STATE

As you can probably tell by now, state is a key concept in React. State allows you to manage data that, when changed, forces React to re-evaluate a component and, ultimately, the UI.

As a developer, you can use state values anywhere in your component (and in your child components, by passing state to them via props). You could, for example, repeat what a user entered like this:

```
function Repeater() {
  const [userInput, setUserInput] = useState('');

  function inputHandler(event) {
    setUserInput(event.target.value);
  };

  return (
    <>
      <input type="text" onChange={inputHandler} />
      <p>You entered: {userInput}</p>
    </>
  );
};
```

This component might not be too useful, but it will work, and it does use state.

Often, in order to do more useful things, you will need to use a state value as a basis to derive a new (often more complex) value. For example, instead of simply repeating what the user entered, you could count the number of entered characters and show that information to the user:

```
function CharCounter() {
  const [userInput, setUserInput] = useState('');

  function inputHandler(event) {
    setUserInput(event.target.value);
  };

  const numChars = userInput.length;

  return (
    <>
      <input type="text" onChange={inputHandler} />
      <p>Characters entered: {numChars}</p>
    </>
  );
};
```

Note the addition of the new **`numChars`** constant (it could also be a variable, via **`let`**). This constant is derived from the **`userInput`** state by accessing the **`length`** property on the string value that's stored in the **`userInput`** state.

This is important! You're not limited to working with state values only. You can manage some key value as state (that is, the value that will change) and derive other values based on that state value—such as, in this case, the number of characters entered by the user. And, indeed, this is something you will do frequently as a React developer.

You might also be wondering why **`numChars`** is a constant and outside of the **`inputHandler`** function. After all, that is the function that is executed upon user input (that is, upon every keystroke the user makes).

Keep in mind what you learned about how React handles state internally. When you call the state-updating function (**`setUserInput`**, in this case), React will re-evaluate the component to which the state belongs. This means that the **`CharCounter`** component function will be called again by React. All the code in that function is therefore executed again.

React does this to determine what the UI should look like after the state update; and, if it detects any differences compared to the currently rendered UI, React will go ahead and update the browser UI (that is, the DOM) accordingly. Otherwise, nothing will happen.

Since React calls the component function again, **`useState()`** will yield its array of values (current state value and state-updating function). The current state value will be the state to which it was set when **`setUserInput`** was called. Therefore, this new **`userInput`** value can be used to perform other calculations anywhere in the component function—such as deriving **`numChars`** by accessing the **`length`** property of **`userInput`**.

That's why **`numChars`** can be a constant. For this component execution, it won't be re-assigned. A new value might only be derived when the component function is executed again in the future (that is, if **`setUserInput`** is called again). And in that case, a brand-new **`numChars`** constant would be created (and the old one would be discarded).

WORKING WITH FORMS AND FORM SUBMISSION

State is commonly used when working with forms and user input. Indeed, most examples in this chapter dealt with some form of user input.

Up to this point, all examples focused on listening to user events that are directly attached to individual input elements. That makes sense because you will often want to listen to events such as keystrokes or an input losing focus. Especially when adding input validation (that is, checking entered values), you might want to use input events to give website users useful feedback while they're typing.

But it's also quite common to react to the overall form submission. For example, the goal could be to combine the input from various input fields and send the data to some backend server. How could you achieve this? How can you listen and react to the submission of a form?

You can do all these things with the help of standard JavaScript events and the appropriate event handler props provided by React. Specifically, the **`onSubmit`** prop can be added to **`<form>`** elements to assign a function that should be executed once a form is submitted. In order to then handle the submission with React and JavaScript, you must ensure that the browser won't do its default thing and generate (and send) an HTTP request automatically.

As in vanilla JavaScript, this can be achieved by calling the **`preventDefault()`** method on the automatically generated event object.

Here's a full example:

```
function NewsletterSignup() {
  const [email, setEmail] = useState('');
  const [agreed, setAgreed] = useState(false);

  function updateEmailHandler(event) {
    // could add email validation here
    setEmail(event.target.value);
  };

  function updateAgreementHandler(event) {
    setAgreed(event.target.checked); // checked is a default JS boolean
property
  };

  function signupHandler(event) {
    event.preventDefault(); // prevent browser default of sending a Http
request

    const userData = {userEmail: email, userAgrees: agreed};
    // doWhateverYouWant(userData);
  };

  return (
    <form onSubmit={signupHandler}>
      <div>
        <label htmlFor="email">Your email</label>
        <input type="email" id="email" onChange={updateEmailHandler}/>
      </div>
      <div>
        <input type="checkbox" id="agree"
onChange={updateAgreementHandler}/>
        <label htmlFor="agree">Agree to terms and conditions</label>
      </div>
    </form>
  );
};
```

This code snippet handles form submission via the **`signupHandler()`** function that's assigned to the built-in **`onSubmit`** prop. User input is still fetched with the help of two state slices (**`email`** and **`agreed`**), which are updated upon the inputs' change events.

> **NOTE**
>
> In the preceding code example, you might've noticed a new prop that wasn't used before in this book: **`htmlFor`**. This is a special prop, built into React and the core JSX elements it provides. It can be added to **`<label>`** elements in order to set the **`for`** attribute for these elements. The reason it is called **`htmlFor`** instead of just **`for`** is that, as explained earlier in the book, JSX looks like HTML but isn't HTML. It's JavaScript under the hood. And in JavaScript, **`for`** is a reserved keyword for **`for`** loops. To avoid problems, the prop is therefore named **`htmlFor`**.

LIFTING STATE UP

Here's a common scenario and problem: you have two components in your React app and a change or event in component A should change the state in component B. To make this less abstract, consider the following simplified example:

```
function SearchBar() {
  const [searchTerm, setSearchTerm] = useState('');

  function updateSearchTermHandler(event) {
    setSearchTerm(event.target.value);
  };

  return <input type="search" onChange={updateSearchTermHandler} />;
};

function Overview() {
  return <p>Currently searching for {searchTerm}</p>;
};

function App() {
  return (
    <>
```

```
      <SearchBar />
      <Overview />
    </>
  );
};
```

In this example, the **`Overview`** component should output the entered search term. But the search term is actually managed in another component—namely, the **`SearchBar`** component. In this simple example, the two components could of course be merged into one single component, and the problem would be solved. But it's very likely that when building more realistic apps, you'll face similar scenarios but with way more complex components. Breaking components up into smaller pieces is considered a good practice, since it keeps the individual components manageable.

Having multiple components depend on some shared piece of state is therefore a scenario you will face frequently when working with React.

This problem can be solved by *lifting state up*. When lifting state up, the state is not managed in either of the two components that use it—neither in **`Overview`**, which reads the state, nor in **`SearchBar`**, which sets the state—but in a shared ancestor component instead. To be precise, it is managed in the **closest** shared ancestor component. Keep in mind that components are nested into each other and thus a "tree of components" (with the **`App`** component as the root component) is built up in the end.

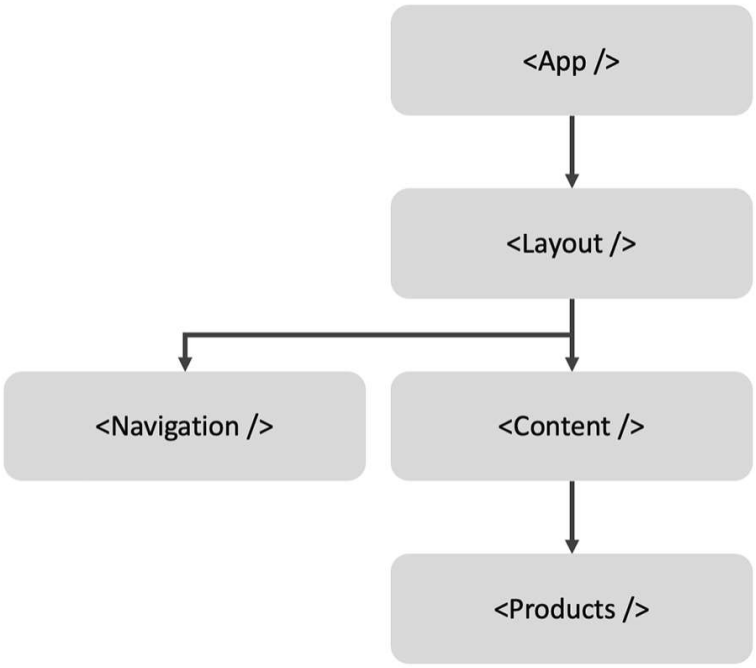

Figure 4.2: An example component tree

In the previous simple code example, the **App** component is the closest (and, in this case, only) ancestor component of both **SearchBar** and **Overview**. If the app was structured as shown in the figure, with state set in **Navigation** and used in **Products**, **Layout** would be the closest ancestor component.

State is lifted up by using props in the components that need to manipulate (that is, set) or read the state, and by registering the state in the ancestor component that is shared by the two other components. Here's the updated example from previously:

```
function SearchBar(props) {
  return <input type="search" onChange={props.onUpdateSearch} />;
};

function Overview({currentTerm}) {
  return <p>Currently searching for {currentTerm}</p>;
};

function App() {
  const [searchTerm, setSearchTerm] = useState('');

  function updateSearchTermHandler(event) {
    setSearchTerm(event.target.value);
  };

  return (
    <>
      <SearchBar onUpdateSearch={updateSearchTermHandler} />
      <Overview currentTerm={searchTerm} />
    </>
  );
};
```

The code didn't actually change that much; it mostly moved around a bit. The state is now managed inside of the shared ancestor component and **App** component, and the two other components get access to it via props.

Three key things are happening in this example:

1. The **`SearchBar`** component receives a prop called **`onUpdateSearch`**, whose value is a function—a function created in the **`App`** component and passed down to **`SearchBar`** from **`App`**.
2. The **`onUpdateSearch`** prop is then set as a value to the **`onChange`** prop on the **`<input>`** element inside of the **`SearchBar`** component.
3. The **`searchTerm`** state (that is, its current value) is passed from **`App`** to **`Overview`** via a prop named **`currentTerm`**.

The first two points could be confusing. But keep in mind that, in JavaScript, functions are first-class objects and regular values. You can store functions in variables and, when using React, pass functions as values for props. And indeed, you could already see that in action at the very beginning of this chapter. When introducing events and event handling, functions were provided as values to all these **`onXYZ`** props (**`onChange`**, **`onBlur`**, and so on).

In this code snippet, a function is passed as a value for a custom prop (that is, a prop expected in a component created by you, not built into React). The **`onUpdateSearch`** prop expects a function as a value because the prop is then itself being used as a value for the **`onChange`** prop on the **`<input>`** element.

The prop is named **`onUpdateSearch`** to make it clear that it expects a function as a value and that it will be connected to an event. Any name could've been chosen though; it doesn't have to start with **`on`**. But it's a common convention to name props that expect functions as values and that are intended to be connected to events like this.

Of course, **`updateSearch`** is not a default event, but since the function will effectively be called upon the **`change`** event of the **`<input>`** element, the prop acts like a custom event.

With this structure, the state was lifted up to the **`App`** component. This component registers and manages the state. But it also exposes the state-updating function (indirectly, in this case, as it is wrapped by the **`updateSearchTermHandler`** function) to the **`SearchBar`** component. And it also provides the current state value (**`searchTerm`**) to the **`Overview`** component via the **`currentTerm`** prop.

Since child and descendent components are also re-evaluated by React when state changes in a component, changes in the **`App`** component will also lead to the **`SearchBar`** and **`Overview`** components being re-evaluated. Therefore, the new prop value for **`searchTerm`** will be picked up, and the UI will be updated by React.

No new React features are needed for this. It's only a combination of state and props. But depending on how these features are connected and where they are used, both simple and more complex app patterns can be achieved.

SUMMARY AND KEY TAKEAWAYS

- Event handlers can be added to JSX elements via **`on[EventName]`** props (for example, **`onClick`**, **`onChange`**).
- Any function can be executed upon (user) events.
- In order to force React to re-evaluate components and (possibly) update the rendered UI, state must be used.
- State refers to data managed internally by React, and a state value can be defined via the **`useState()`** Hook.
- React Hooks are JavaScript functions that add special features to React components (for example, the state feature, in this chapter).
- **`useState()`** always returns an array with exactly two elements:
- The **first element** is the current state value.
- The **second element** is a function to set the state to a new value (the *state-updating function*).
- When setting the state to a new value that depends on the previous value, a function should be passed to the state-updating function. This function then receives the previous state as a parameter (which will be provided automatically by React) and returns the new state that should be set.
- Any valid JavaScript value can be set as state—besides primitive values such as strings or numbers. This also includes reference values such as objects and arrays.
- If state needs to change because of some event that occurs in another component, you should *lift the state up* and manage it on a higher, shared level (that is, a common ancestor component).

WHAT'S NEXT?

State is an extremely important building block because it enables you to build truly dynamic applications. With this key concept out of the way, the next chapter will dive into utilizing state (and other concepts learned thus far) to render content conditionally and to render lists of content.

These are common tasks that are required in almost any UI or web app you're building, no matter whether it's about showing a warning overlay or displaying a list of products. The next chapter will help you add such features to your React apps.

TEST YOUR KNOWLEDGE!

Test your knowledge about the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at https://packt.link/Zu02Z.

1. What problem does state solve?
2. What's the difference between props and state?
3. How is state registered in a component?
4. Which values does the `useState()` Hook provide?
5. How many state values can be registered for a single component?
6. Does state affect components other than the component in which it was registered?
7. How should state be updated if the new state depends on the previous state?
8. How can state be shared across multiple components?

APPLY WHAT YOU LEARNED

With the new knowledge gained in this chapter, you are finally able to build truly dynamic UIs and React applications. Instead of being limited to hardcoded, static content and pages, you can now use state to set and update values and force React to re-evaluate components and the UI.

Here, you will find an activity that allows you to apply all the knowledge, including this new state knowledge, you have acquired up to this point.

ACTIVITY 4.1: BUILDING A SIMPLE CALCULATOR

In this activity, you'll build a very basic calculator that allows users to add, subtract, multiply, and divide two numbers with each other.

The steps are as follows:

1. Build the UI by using React components. Be sure to build four separate components for the four math operations, even though lots of code could be reused.
2. Collect the user input and update the result whenever the user enters a value into one of the two related input fields.

   Note that when working with numbers and getting those numbers from user input, you will need to ensure that the entered values are treated as numbers and not as strings.

The final result and UI of the calculator should look like this:

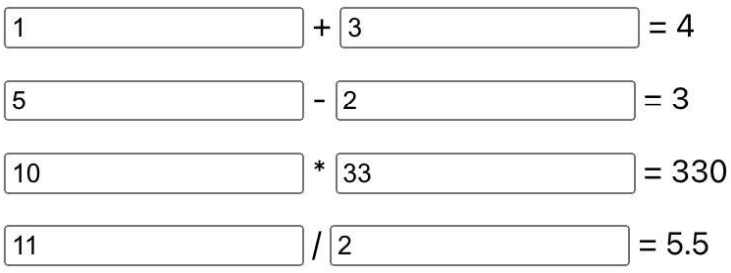

Figure 4.3: Calculator user interface

> **NOTE**
>
> The solution to this activity can be found via this link.

ACTIVITY 4.2: ENHANCING THE CALCULATOR

In this activity, you'll build upon *Activity 4.1* to make the calculator built there slightly more complex. The goal is to reduce the number of components and build one single component in which users can select the mathematical operation via a drop-down element. In addition, the result should be output in a different component—that is, not in the component where the user input is gathered.

The steps are as follows:

1. Remove three of the four components from the previous activity and use one single component for all mathematical operations.
2. Add a drop-down element (**`<select>`** element) to that remaining component (between the two inputs) and add the four math operations as options (**`<option>`** elements) to it.
3. Use state to gather both the numbers entered by the user and the math operation chosen via the drop-down (it's up to you whether you prefer one single state object or multiple state slices).
4. Output the result in another component. (Hint: Choose a good place for registering and managing the state.)

The result and UI of the calculator should look like this:

Figure 4.4: User interface of the enhanced calculator

> **NOTE**
>
> The solution to this activity can be found via this link.

5

RENDERING LISTS AND CONDITIONAL CONTENT

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Output dynamic content conditionally.
- Render lists of data and map list items to JSX elements.
- Optimize lists such that React is able to efficiently update the user interface when needed.

INTRODUCTION

By this point in the book, you are already familiar with several key concepts, including components, props, state, and events, with which you have all the core tools you need to build all kinds of different React apps and websites. You have also learned how to output dynamic values and results as part of the user interface.

But there is one topic related to outputting dynamic data that has not yet been discussed in depth: outputting content conditionally and rendering list content. Since most (if not all) websites and web apps you build will require at least one of these two concepts, it is crucial to know how to work with conditional content and list data.

In this chapter, you will therefore learn how to render and display different user interface elements (and even entire user interface sections), based on dynamic conditions. In addition, you will learn how to output lists of data (such as a to-do list with its items) and render JSX elements dynamically for the items that make up a list. This chapter will also explore important best practices related to outputting lists and conditional content.

WHAT ARE CONDITIONAL CONTENT AND LIST DATA?

Before diving into the techniques for outputting conditional content or list content, it is important to understand what exactly is meant by those terms.

Conditional content simply means any kind of content that should only be displayed under certain circumstances. Examples are as follows:

- Error overlays that should only show up if a user submits incorrect data in a form
- Additional form input fields that appear once the user chooses to enter extra details (such as business details)
- A loading spinner that is displayed while data is sent or fetched to or from a backend server
- A side navigation menu that slides into view when the user clicks on a menu button

This is just a very short list of a few examples. You could, of course, come up with hundreds of additional examples. But it should be clear what all these examples are about in the end: visual elements or entire sections of the user interface that are only shown if certain conditions are met.

In the first example (an error overlay), the condition would be that a user entered incorrect data into a form. The conditionally shown content would then be the error overlay.

Conditional content is extremely common, since virtually all websites and web apps have some content that is similar or comparable to the preceding examples.

In addition to conditional content, many websites also output lists of data. It might not always be immediately obvious, but if you think about it, there is virtually no website that does not display some kind of list content. Again, here are some examples of list content that may be outputted on a site:

- An online shop displaying a grid or list of products
- An event booking site displaying a list of events
- A shopping cart displaying a list of cart items
- An orders page displaying a list of orders
- A blog displaying a list of blog posts—and maybe a list of comments below a blog post
- A list of navigation items in the header

An endless list (no pun intended) of examples could be created here. Lists are everywhere on the web. And, as the preceding examples show, many (probably even most) websites have multiple lists with various kinds of data on the same site.

Take an online shop, for example. Here, you would have a list (or a grid, which is really just another kind of list) of products, a list of shopping cart items, a list of orders, a list of navigation items in the header, and certainly a lot of other lists as well. This is why it is important that you know how to output any kind of list with any kind of data in React-driven user interfaces.

RENDERING CONTENT CONDITIONALLY

Imagine the following scenario. You have a button that, when clicked, should result in the display of an extra text box, as shown in the following diagram:

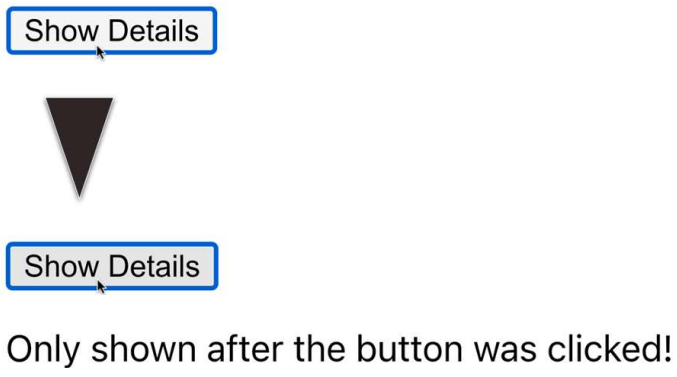

Figure 5.1: A button that, once clicked, reveals another element

This is a very simple example, but not an unrealistic one. Many websites have parts of the user interface that work like this. Showing extra information upon a button click (or some similar interaction) is a common pattern. Just think of nutrition information below a meal on a food order site or an FAQ section where answers are shown after selecting a question.

So, how could this scenario be implemented in a React app?

If you ignore the requirement of rendering some of the content conditionally, the overall React component could look like this:

```
function TermsOfUse() {
  return (
    <section>
      <button>Show Terms of Use Summary</button>
      <p>By continuing, you accept that we will not indemnify you for any
damage or harm caused by our products.</p>
    </section>
  );
}
```

This component has absolutely no conditional code in it, and therefore, both the button and the extra information box are shown all the time.

In this example, how could the paragraph with the terms-of-use summary text be shown conditionally (that is, only after the button is clicked)?

With the knowledge gained throughout the previous chapters, especially *Chapter 4*, *Working with Events and State*, you already have the skills needed to only show the text after the button is clicked. The following code shows how the component could be rewritten to show the full text only after the button is clicked:

```
import { useState } from 'react';

function TermsOfUse() {
  const [showTerms, setShowTerms] = useState(false);

  function showTermsSummaryHandler() {
    setShowTerms(true);
  }

  let paragraphText = '';

  if (showTerms) {
    paragraphText = 'By continuing, you accept that we will not indemnify
you for any damage or harm caused by our products.';
  }

  return (
    <section>
      <button onClick={showTermsSummaryHandler}>Show Terms of Use
Summary</button>
      <p>{paragraphText}</p>
    </section>
  );
}
```

Parts of the code shown in this snippet already qualify as conditional content. The **`paragraphText`** value is set conditionally, with the help of an **`if`** statement based on the value stored in the **`showTerms`** state.

But the **<p>** element itself is actually **not** conditional. It is always there, regardless of whether it contains a full sentence or an empty string. If you were to open the browser developer tools and inspect that area of the page, an empty paragraph element would be visible, as shown:

Figure 5.2: An empty paragraph element is rendered as part of the DOM

You can translate your knowledge about conditional values (such as the paragraph text) to conditional elements, however. Besides storing standard values such as text or numbers in variables, you can also store JSX elements in variables. This is possible because, as mentioned in *Chapter 1*, *React: What and Why*, JSX is just syntactic sugar. Behind the scenes, a JSX element is a standard JavaScript function that is executed by React. And, of course, the return value of a function call can be stored in a variable or constant.

With that in mind, the following code could be used to render the entire paragraph conditionally:

```
import { useState } from 'react';

function TermsOfUse() {
  const [showTerms, setShowTerms] = useState(false);

  function showTermsSummaryHandler() {
    setShowTerms(true);
  }

  let paragraph;

  if (showTerms) {
    paragraph = <p>By continuing, you accept that we will not indemnify
you for any damage or harm caused by our products.</p>;
  }

  return (
```

```
    <section>
      <button onClick={showTermsSummaryHandler}>Show Terms of Use
Summary</button>
      {paragraph}
    </section>
  );
}
```

In this example, if **`showTerms`** is **`true`**, the **`paragraph`** variable does not store text but instead an entire JSX element (the **`<p>`** element). In the returned JSX code, the value stored in the **`paragraph`** variable is outputted dynamically via **`{paragraph}`**. If **`showTerms`** is **`false`**, **`paragraph`** stores the value **`undefined`** and nothing is rendered to the DOM. Therefore, inserting **`null`** or **`undefined`** in JSX code leads to nothing being outputted by React. But if **`showTerms`** is **`true`**, the complete paragraph is saved as a value and outputted in the DOM.

This is how entire JSX elements can be rendered dynamically. Of course, you are not limited to single elements. You could store entire JSX tree structures (such as multiple, nested, or sibling JSX elements) inside variables or constants. As a simple rule, anything that can be returned by a component function can be stored in a variable.

Different Ways of Rendering Content Conditionally

In the example shown previously, content is rendered conditionally by using a variable, which is set with the help of an **`if`** statement and then outputted dynamically in JSX code. This is a common (and perfectly fine) technique of rendering content conditionally, but it is not the only approach you can use.

Alternatively, you could also do the following:

- Utilize ternary expressions.
- *Abuse* JavaScript logical operators.
- Use any other valid JavaScript way of selecting values conditionally.

Utilizing Ternary Expressions

In JavaScript (and many other programming languages), you can use ternary expressions as alternatives to **`if`** statements. Ternary expressions can save you lines of code, especially with simple conditions where the main goal is to assign some variable value conditionally.

Here is a direct comparison—first starting with a regular **`if`** statement:

```
let a = 1;
if (someCondition) {
  a = 2;
}
```

And here is the same logic, implemented with a ternary expression:

```
const a = someCondition ? 1 : 2;
```

This is standard JavaScript code, not React-specific. But it is important to understand this core JavaScript feature in order to understand how it can be used in React apps.

Translated to the previous React example, the paragraph content could be set and outputted conditionally with the help of ternary expressions like this:

```
import { useState } from 'react';

function TermsOfUse() {
  const [showTerms, setShowTerms] = useState(false);

  function showTermsSummaryHandler() {
    setShowTerms(true);
  }

  const paragraph = showTerms ? <p>By continuing, you accept that we will
not indemnify you for any damage or harm caused by our products.</p> :
null;

  return (
    <section>
      <button onClick={showTermsSummaryHandler}>Show Terms of Use
Summary</button>
      {paragraph}
    </section>
  );
}
```

As you can see, the overall code is a bit shorter than before, when an **`if`** statement was used. The paragraph constant contains either the paragraph (including the text content) or **`null`**. **`null`** is used as an alternative value because **`null`** can safely be inserted into JSX code as it simply leads to nothing being rendered in its place.

A disadvantage of ternary expressions is that readability and understandability may suffer—especially when using nested ternary expressions, like in the following example:

```
const paragraph = !showTerms ? null : someOtherCondition ? <p>By
continuing, you accept that we will not indemnify you for any damage or
harm caused by our products.</p> : null;
```

This code is difficult to read and even more difficult to understand. For this reason, you should typically avoid writing nested ternary expressions and fall back to **`if`** statements in such situations.

But, despite these potential disadvantages, ternary expressions can help you write less code in React apps, especially when using them inline, directly inside some JSX code:

```
import { useState } from 'react';

function TermsOfUse() {
  const [showTerms, setShowTerms] = useState(false);

  function showTermsSummaryHandler() {
    setShowTerms(true);
  }

  return (
    <section>
      <button>Show Terms of Use Summary</button>
      {showTerms ? <p>By continuing, you accept that we will not
indemnify you for any damage or harm caused by our products.</p> : null}
    </section>
  );
}
```

This is the same example as before, only now it's even shorter since here you avoid using the **`paragraph`** constant by utilizing the ternary expression directly inside of the JSX snippet. This allows for relatively lean component code, so it is quite common to use ternary expressions in JSX code in React apps to take advantage of this.

ABUSING JAVASCRIPT LOGICAL OPERATORS

Ternary expressions are popular because they enable you to write less code, which, when used in the right places (and avoiding nesting multiple ternary expressions), can help with overall readability.

Especially in React apps, in JSX code you will often write ternary expressions like this:

```
<div>
  {showDetails ? <h1>Product Details</h1> : null}
</div>
```

Or like this:

```
<div>
  {showTerms ? <p>Our terms of use …</p> : null}
</div>
```

What do these two snippets have in common?

They are unnecessarily long, because in both examples, the else case (**`: null`**) must be specified, even though it adds nothing to the final user interface. After all, the primary purpose of these ternary expressions is to render JSX elements (**`<h1>`** and **`<p>`**, in the preceding examples). The else case (**`: null`**) simply means nothing is rendered if the conditions (**`showDetails`** and **`showTerms`**, respectively) are not met.

This is why a different pattern is popular among React developers:

```
<div>
  {showDetails && <h1>Product Details</h1>}
</div>
```

This is the shortest possible way of achieving the intended result, rendering only the **`<h1>`** element and its content if **`showDetails`** is **`true`**.

This code uses (or abuses) an interesting behavior of JavaScript's logical operators, and specifically of the **`&&`** (logical "and") operator. In JavaScript, the **`&&`** operator returns the second value (that is, the value after **`&&`**), if the first value (that is, the value before **`&&`**) is **`true`** or truthy (that is, not **`false`**, **`undefined`**, **`null`**, **`0`**, and so on).

For example, the following code would output **`'Hello'`**:

```
console.log(1 === 1 && 'Hello');
```

This behavior can be used to write very short expressions that check a condition and then output another value, as shown in the preceding example.

> **NOTE**
>
> It is worth noting that using **`&&`** can lead to unexpected results if you're using it with non-Boolean condition values (that is, if the value in front of **`&&`** holds a non-Boolean value). If **`showDetails`** were **`0`** instead of **`false`** (for whatever reason), the number **`0`** would be displayed on the screen. You should therefore ensure that the value acting as a condition yields **`null`** or **`false`** instead of arbitrary falsy values. You could, for example, force a conversion to a Boolean by adding **`!!`** (for example, **`!!showDetails`**). That is not required if your condition value already holds **`null`** or **`false`**.

GET CREATIVE!

At this point, you have learned about three different ways of defining and outputting content conditionally (regular **`if`** statements, ternary expressions, and using the **`&&`** operator). But the most important point is that React code is ultimately just regular JavaScript code. Hence, any approach that selects values conditionally will work.

If it makes sense in your specific use case and React app, you could also have a component that selects and outputs content conditionally like this:

```
const languages = {
  de: 'de-DE',
  us: ,en-US',
  uk: ,en-GB'
};

function LanguageSelector({country}) {
  return <p>Selected Language: {languages[country]}</p>
}
```

This component outputs either **`'de-DE'`**, **`'en-US'`**, or **`'en-GB'`** based on the value of the **`country`** prop. This result is achieved by using JavaScript's dynamic property selection syntax. Instead of selecting a specific property via the dot notation (such as **`person.name`**), you can select property values via the bracket notation. With that notation, you can either pass a specific property name (**`languages['de-DE']`**) or an expression that yields a property name (**`languages[country]`**).

Selecting property values dynamically like this is another common pattern for picking values from a map of values. It is therefore an alternative to specifying multiple **`if`** statements or ternary expressions.

And, in general, you can use any approach that works in standard JavaScript—because React is, after all, just standard JavaScript at its core.

WHICH APPROACH IS BEST?

Various ways of setting and outputting content conditionally have been discussed, but which approach is best?

That really is up to you (and, if applicable, your team). The most important advantages and disadvantages have been highlighted, but ultimately, it is your decision. If you prefer ternary expressions, there's nothing wrong with choosing them over the logical **`&&`** operator, for example.

It will also depend on the exact problem you are trying to solve. If you have a map of values (such as a list of countries and their country language codes), going for dynamic property selection instead of multiple **`if`** statements might be preferable. On the other hand, if you have a single **`true`**/**`false`** condition (such as **`age > 18`**), using a standard **`if`** statement or the logical **`&&`** operator might be best.

SETTING ELEMENT TAGS CONDITIONALLY

Outputting content conditionally is a very common scenario. But sometimes, you will also want to choose the type of HTML tag that will be outputted conditionally. Typically, this will be the case when you build components whose main task is to wrap and enhance built-in components.

Here's an example:

```
function Button({isButton, config, children}) {
  if (isButton) {
    return <button {...config}>{children}</button>;
  }
  return <a {...config}>{children}</a>;
};
```

This **`Button`** component checks whether the **`isButton`** prop value is truthy and, if that is the case, returns a **`<button>`** element. The **`config`** prop is expected to be a JavaScript object, and the standard JavaScript spread operator (**`...`**) is used to then add all key-value pairs of the **`config`** object as props to the **`<button>`** element. If **`isButton`** is not truthy (maybe because no value was provided for **`isButton`**, or because the value is **`false`**), the **`else`** condition becomes active. Instead of a **`<button>`** element, an **`<a>`** element is returned.

> **NOTE**
>
> Using the spread operator (**`...`**) to translate an object's properties (key-value pairs) into component props is another common React pattern (and was introduced in *Chapter 3*, *Components and Props*). The spread operator is not React-specific but using it for this special purpose *is*.
>
> When spreading an object such as **`{link: 'https://some-url.com', isButton: false}`** onto an **`<a>`** element (via **`<a {...obj}>`**), the result would be the same as if all props had been set individually (that is, **`<a link="https://some-url.com" isButton={false}>`**).
>
> This pattern is particularly popular in situations where you build custom *wrapper components* that wrap a common core component (e.g., **`<button>`**, **`<input>`**, or **`<a>`**) to add certain styles or behaviors, while still allowing for the component to be used in the same way as the built-in component (that is, you can set all the default props).

The **`Button`** component from the preceding example returns two totally different JSX elements, depending on the **`isButton`** prop value. This is a great way of checking a condition and returning different content (that is, conditional content).

But, by using a special React behavior, this component could be written with even less code:

```
function Button({isButton, config, children}) {
  const Tag = isButton ? 'button' : 'a';
  return <Tag {...config}>{children}</Tag>;
};
```

The special behavior is that tag names can be stored (as string values) in variables or constants, and that those variables or constants can then be used like JSX elements in JSX code (as long as the variable or constant name starts with an uppercase character, like all your custom components).

The **`Tag`** constant in the preceding example stores either the **`'button'`** or **`'a'`** string. Since it starts with an uppercase character (**`Tag`**, instead of **`tag`**), it can then be used like a custom component inside of JSX code snippets. React accepts this as a component, even though it isn't a component function. This is because a standard HTML element tag name is stored, and so React can render the appropriate built-in component. The same pattern could also be used with custom components. Instead of storing string values, you would store pointers to your custom component functions through the following:

```
import MyComponent from './my-component';
import MyOtherComponent from './my-other-component';

const Tag = someCondition ? MyComponent : MyOtherComponent;
```

This is another useful pattern that can help save code and hence leads to leaner components.

OUTPUTTING LIST DATA

Besides outputting conditional data, you will often work with list data that should be outputted on a page. As mentioned earlier in this chapter, some examples are lists of products, transactions, and navigation items.

Typically, in React apps, such list data is received as an array of values. For example, a component might receive an array of products via props (passed into the component from inside another component that might be getting that data from some backend API):

```
function ProductsList({products}) {
  // … todo!
};
```

In this example, the products array could look like this:

```
const products = [
  {id: 'p1', title: 'A Book', price: 59.99},
  {id: 'p2', title: 'A Carpet', price: 129.49},
  {id: 'p3', title: 'Another Book', price: 39.99},
];
```

This data can't be outputted like this, though. Instead, the goal is typically to translate it into a list of JSX elements which fits. For example, the desired result could be the following:

```
<ul>
  <li>
    <h2>A Book</h2>
    <p>$59.99</p>
  </li>
  <li>
    <h2>A Carpet</h2>
    <p>$129.49</p>
  </li>
  <li>
    <h2>Another Book</h2>
    <p>$39.99</p>
  </li>
</ul>
```

How can this transformation be achieved?

Again, it's a good idea to ignore React and find a way to transform list data with standard JavaScript. One possible way to achieve this would be to use a **`for`** loop, as shown:

```
const transformedProducts = [];
for (const product of products) {
  transformedProducts.push(product.title);
}
```

In this example, the list of product objects (**`products`**) is transformed into a list of product titles (that is, a list of string values). This is achieved by looping through all product items in **`products`** and extracting only the **`title`** property from each product. This **`title`** property value is then pushed into the new **`transformedProducts`** array.

A similar approach can be used to transform the list of objects into a list of JSX elements:

```
const productElements = [];
for (const product of products) {
  productElements.push((
    <li>
```

```
      <h2>{product.title}</h2>
      <p>${product.price}</p>
    </li>
  ));
}
```

The first time you see code like this, it might look a bit strange. But keep in mind that JSX code can be used anywhere where regular JavaScript values (that is, numbers, strings, objects, and so on) can be used. Therefore, you can also **`push`** a JSX value onto an array of values. And since it's JSX code, you can also output content dynamically in those JSX elements (such as **`<h2>{product.title}</h2>`**).

This code is valid, and is an important first step toward outputting list data. But it is only the first step, since the current data was transformed but still isn't returned by a component.

How can such an array of JSX elements be returned then?

The answer is that it can be returned without any special tricks or code. JSX actually accepts array values as dynamically outputted values.

You can output the **`productElements`** array like this:

```
return (
  <ul>
    {productElements}
  </ul>
);
```

When inserting an array of JSX elements into JSX code, all JSX elements inside that array are outputted next to each other. So, the following two snippets would produce the same output:

```
return (
  <div>
    {[<p>Hi there</p>, <p>Another item</p>]}
  </div>
);

return (
  <div>
    <p>Hi there</p>
```

```
    <p>Another item</p>
  </div>
);
```

With this in mind, the **`ProductsList`** component could be written like this:

```
function ProductsList({products}) {
  const productElements = [];
  for (const product of products) {
    productElements.push((
      <li>
        <h2>{product.title}</h2>
        <p>${product.price}</p>
      </li>
    ));
  }

  return (
    <ul>
      {productElements}
    </ul>
  );
};
```

This is one possible approach for outputting list data. As explained earlier in this chapter, it's all about using standard JavaScript features and combining those features with JSX.

MAPPING LIST DATA

Outputting list data with **`for`** loops works, as you can see in the preceding examples. But just as with **`if`** statements and ternary expressions, you can replace **`for`** loops with an alternative syntax to write less code and improve component readability.

JavaScript offers a built-in array method that can be used to transform array items: the **`map()`** method. **`map()`** is a default method that can be called on any JavaScript array. It accepts a function as a parameter and executes that function for every array item. The return value of this function should be the transformed value. **`map()`** then combines all these returned, transformed values into a new array that is then returned by **`map()`**.

You could use **`map()`** like this:

```
const users = [
  {id: 'u1', name: 'Max', age: 35},
  {id: 'u2', name: 'Anna', age: 32}
];
const userNames = users.map(user => user.name);
// userNames = ['Max', 'Anna']
```

In this example, **`map()`** is used to transform the array of user objects into an array of usernames (that is, an array of string values).

The **`map()`** method is often able to produce the same result as that of a **`for`** loop, but with less code.

Therefore, **`map()`** can also be used to generate an array of JSX elements and the **`ProductsList`** component from before could be rewritten like this:

```
function ProductsList({products}) {
  const productElements = products.map(product => (
      <li>
        <h2>{product.title}</h2>
        <p>${product.price}</p>
      </li>
    )
  );

  return (
    <ul>
      {productElements}
    </ul>
  );
};
```

This is already shorter than the earlier **`for`** loop example. But, just as with ternary expressions, the code can be shortened even more by moving the logic directly into the JSX code:

```
function ProductsList({products}) {
  return (
    <ul>
      {products.map(product => (
          <li>
```

```
            <h2>{product.title}</h2>
            <p>${product.price}</p>
          </li>
        )
      )}
    </ul>
  );
};
```

Depending on the complexity of the transformation (that is, the complexity of the code executed inside the inner function, which is passed to the **`map()`** method), for readability reasons, you might want to consider not using this *inline* approach (such as when mapping array elements to some complex JSX structure or when performing extra calculations as part of the mapping process). Ultimately, this comes down to personal preference and judgment.

Because it's very concise, using the **`map()`** method (either with the help of an extra variable or constant, or directly *inline* in the JSX code) is the de facto standard approach for outputting list data in React apps and JSX in general.

UPDATING LISTS

Imagine you have a list of data mapped to JSX elements, and a new list item is added at some point. Or, consider a scenario in which you have a list wherein two list items swap places (that is, the list is reordered). How can such updates be reflected in the DOM?

The good news is that React will take care of that for you if the update is performed in a stateful way (that is, by using React's state concept, as explained in *Chapter 4*, *Working with Events and State*).

However, there are a couple of important aspects to updating (stateful) lists you should be aware of.

Here's a simplified example that would **not** work as intended:

```
import { useState } from 'react';

function Todos() {
  const [todos, setTodos] = useState(['Learn React', 'Recommend this
book']);

  function addTodoHandler() {
    todos.push('A new todo');
```

```
  };

  return (
    <div>
      <button onClick={addTodoHandler}>Add Todo</button>
      <ul>
        {todos.map(todo => <li>{todo}</li>)}
      </ul>
    </div>
  );
};
```

Initially, two to-do items would be displayed on the screen (**`<li>Learn React</li>`** and **`<li>Recommend this book</li>`**). But once the button is clicked and **`addTodoHandler`** is executed, the expected result of another to-do item being displayed will not materialize.

This is because executing **`todos.push('A new todo')`** will update the **`todos`** array, but React won't notice it. Keep in mind that you must only update the state via the state updating function returned by **`useState()`**; otherwise, React will not re-evaluate the component function.

So how about this code?

```
function addTodoHandler() {
  setTodos(todos.push('A new todo'));
};
```

This is also incorrect because the state updating function (**`setTodos`**, in this case) should receive the new state (that is, the state that should be set) as an argument. But the **`push()`** method doesn't return the updated array. Instead, it mutates the existing array in place. Even if **`push()`** were to return the updated array, it would still be wrong to use the preceding code, because the data would be changed (mutated) behind the scenes before the state updating function would be executed. Technically, data would be changed before informing React about that change. Following the React best practices, this should be avoided.

Therefore, when updating an array (or, as a side note, an object in general), you should perform this update in an **immutable** way—i.e., without changing the original array or object. Instead, a new array or object should be created. This new array can be based on the old array and contain all the old data, as well as any new or updated data.

Therefore, the **`todos`** array should be updated like this:

```
function addTodoHandler() {
  setTodos(curTodos => [...curTodos, 'A new todo']);
  // alternative: Use concat() instead of the spread operator:
  // concat(), unlike push(), returns a new array
  // setTodos(curTodos => curTodos.concat('A new todo'));
};
```

By using **`concat()`** or a new array, combined with the spread operator, a brand-new array is provided to the state updating function. Note also that a function is passed to the state updating function, since the new state depends on the previous state.

When updating an array (or any object) state value like this, React is able to pick up those changes. Therefore, React will re-evaluate the component function and apply any required changes to the DOM.

> **NOTE**
>
> Immutability is not a React-specific concept, but it's a key one in React apps nonetheless. When working with state and reference values (that is, objects and arrays), immutability is extremely important to ensure that React is able to pick up changes and no "invisible" (that is, not recognized by React) state changes are performed.
>
> There are different ways of updating objects and arrays immutably, but a popular approach is to create new objects or arrays and then use the spread operator (`...`) to merge existing data into those new arrays or objects.

A PROBLEM WITH LIST ITEMS

If you're following along with your own code, you might've noticed that React actually shows a warning in the browser developer tools console, as shown in the following screenshot:

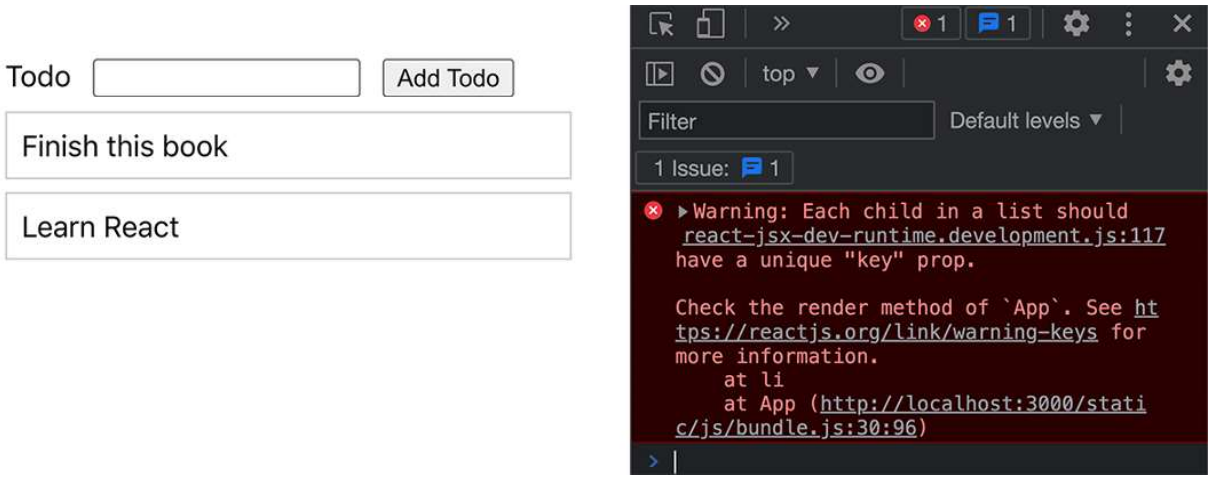

Figure 5.3: React sometimes generates a warning regarding missing unique keys

React is complaining about missing keys.

To understand this warning and the idea behind keys, it's helpful to explore a specific use case and a potential problem with that scenario. Assume that you have a React component that is responsible for displaying a list of items—maybe a list of to-do items. In addition, assume that those list items can be reordered and that the list can be edited in other ways (for example, new items can be added, existing items can be updated or deleted, and so on). Put in other words, the list is not static.

Consider this example user interface, in which a new item is added to a list of to-do items:

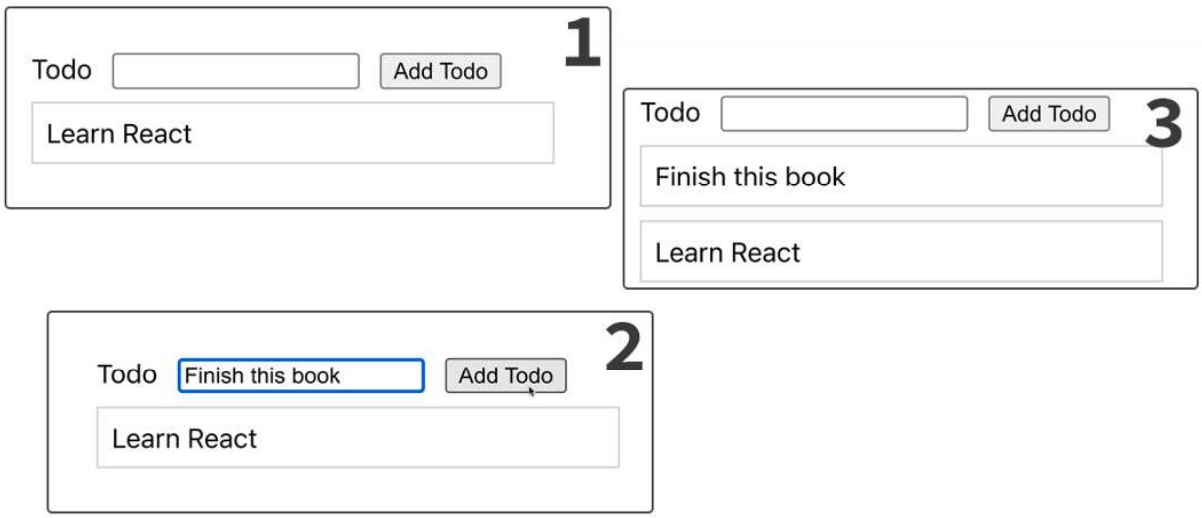

Figure 5.4: A list gets updated by inserting a new item at the top

In the preceding figure, you can see the initially rendered list (**1**), which is then updated after a user entered and submitted a new to-do value (**2**). A new to-do item was added to the top of the list (that is, as the first item of the list) (**3**).

> **NOTE**
>
> The example source code for this demo app can be found at https://packt.link/LsP33.

If you work on this app and open the browser developer tools (and then the JavaScript console), you will see the "missing keys" warning that has been mentioned before. This app also helps with understanding where this warning is coming from.

In the Chrome DevTools, navigate to the **`Elements`** tab and select one of the to-do items or the empty to-do list (that is, the **`<ul>`** element). Once you add a new to-do item, any DOM elements that were inserted or updated are highlighted by Chrome in the **`Elements`** tab (by flashing briefly). Refer to the following screenshot:

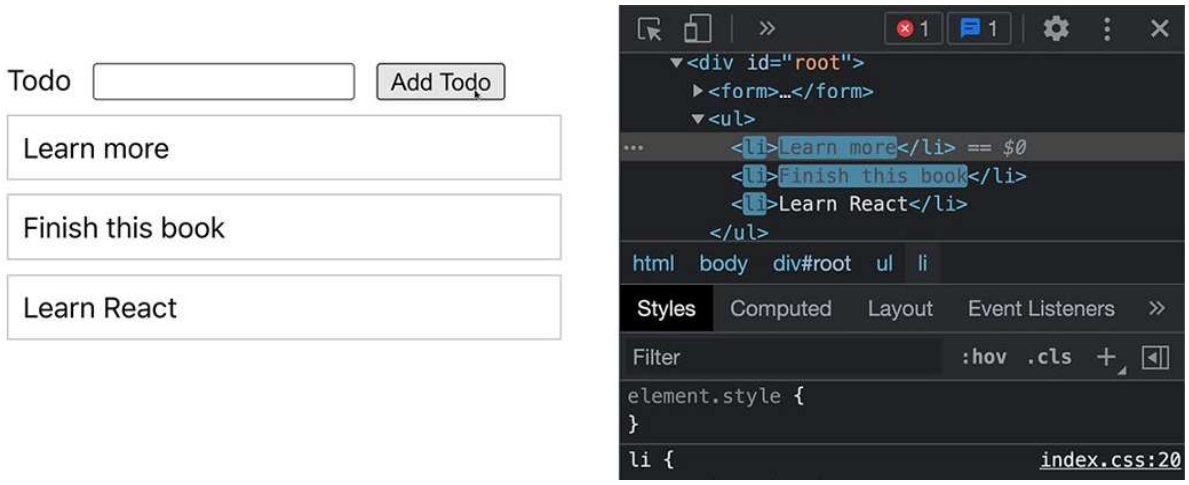

Figure 5.5: Updated DOM items are highlighted in the Chrome DevTools

The interesting part is that not only the newly added to-do element (that is, the newly inserted **`<li>`** element) is flashing. Instead, **all** existing **`<li>`** elements, which reflect existing to-do items that were not changed, are highlighted by Chrome. This implies that all these other **`<li>`** elements were also updated in the DOM—even though there was no need for that update. The items existed before, and their content (the to-do text) didn't change.

For some reason, React seems to destroy the existing DOM nodes (that is, the existing **`<li>`** items), just to then recreate them immediately. This happens for every new to-do item that is added to the list. As you might imagine, this is not very efficient and can cause performance problems for more complex apps that might be rendering dozens or hundreds of items across multiple lists.

This happens because React has no way of knowing that only one DOM node should be inserted. It cannot tell that all other DOM nodes should stay untouched because React only received a brand-new state value: a new array, filled with new JavaScript objects. Even if the content of those objects didn't change, they are technically still new objects (new values in memory).

As the developer, you know how your app works and that the content of the to-dos array didn't actually change that much. But React doesn't know that. Therefore, React determines that all existing list items (**<li>** items) must be discarded and replaced by new items that reflect the new data that was provided as part of the state update. That is why **all** list-related DOM nodes are updated (that is, destroyed and recreated) for every state update.

KEYS TO THE RESCUE!

The problem outlined previously is an extremely common one. Most list updates are incremental updates, not bulk changes. But React can't tell whether that is the case for your use case and your list.

That's why React uses the concept of **keys** when working with list data and rendering list items. Keys are simply unique **id** values that can (and should) be attached to JSX elements when rendering list data. Keys help React identify elements that were rendered before and didn't change. By allowing for the unique identification of all list elements, keys also help React to move (list item) DOM elements around efficiently.

Keys are added to JSX elements via the special built-in **key** prop that is accepted by every component:

```
<li key={todo.id}>{todo.text}</li>
```

This special prop can be added to all components, be they built-in or custom. You don't need to accept or handle the **key** prop in any way on your custom components; React will do that for you automatically.

The **key** prop requires a value that is unique for every list item. No two list items should have the same key. In addition, good keys are directly attached to the underlying data that makes up the list item. Therefore, list item indexes are poor keys because the index isn't attached to the list item data. If you reorder items in a list, the indexes stay the same (an array always starts with index **0**, followed by **1**, and so on) but the data is changed.

Consider the following example:

```
const hobbies = ['Sports', 'Cooking'];
const reversed = hobbies.reverse(); // ['Cooking', 'Sports']
```

In this example, **'Sports'** has the index **0** in the hobbies array. In the reversed array, its index would be **1** (because it's the second item now). In this case, if the index were used as a key, the data would not be attached to it.

Good keys are unique **`id`** values, such that every **`id`** belongs to exactly one value. If that value moves or is removed, its **`id`** should move or disappear with that value.

Finding good **`id`** values typically isn't a huge problem since most list data is fetched from databases anyway. No matter whether you're dealing with products, orders, users, or shopping cart items, it's all data that would typically be stored in a database. This kind of data already has unique **`id`** values since you always have some kind of unique identification criteria when storing data in databases.

Sometimes, even the values themselves can be used as keys. Consider the following example:

```
const hobbies = ['Sports', 'Cooking'];
```

Hobbies are string values, and there is no unique **`id`** value attached to individual hobbies. Every hobby is a primitive value (a string). But in cases like this, you typically won't have duplicate values as it doesn't make sense for a hobby to be listed more than once in an array like this. Therefore, the values themselves qualify as good keys:

```
hobbies.map(hobby => <li key={hobby}>{hobby}</li>);
```

In cases where you can't use the values themselves and there is no other possible key value, you can generate unique **`id`** values directly in your React app code. As a last resort, you can also fall back to using indexes; but be aware that this can lead to unexpected bugs and side effects if you reorder list items.

With keys added to list item elements, React is able to identify all items correctly. When the component state changes, it can identify JSX elements that were rendered before already. Those elements are therefore not destroyed or recreated anymore.

You can confirm this by again opening the browser DevTools to check which DOM elements are updated upon changes to the underlying list data:

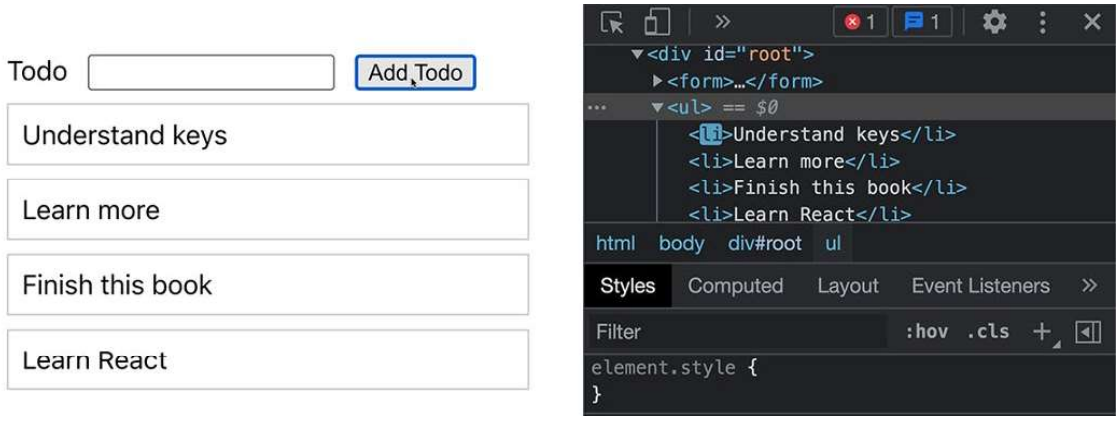

Figure 5.6: From multiple list items, only one DOM element gets updated

After adding keys, when updating the list state, only the new DOM item is highlighted in the Chrome DevTools. The other items are (correctly) ignored by React.

SUMMARY AND KEY TAKEAWAYS

- Like any other JavaScript value, JSX elements can be set and changed dynamically, based on different conditions.
- Content can be set conditionally via **`if`** statements, ternary expressions, the logical "and" operator (**`&&`**), or in any other way that works in JavaScript.
- There are multiple ways to handle conditional content—any approach that would work in vanilla JavaScript can also be used in React apps.
- Arrays with JSX elements can be inserted into JSX code and will lead to the array elements being outputted as sibling DOM elements.
- List data can be converted into JSX element arrays via **`for`** loops, the **`map()`** method, or any other JavaScript approach that leads to a similar conversion.
- Using the **`map()`** method is the most common way of converting list data to JSX element lists.
- Keys (via the **`key`** prop) should be added to the list JSX elements to help React update the DOM efficiently.

WHAT'S NEXT?

With conditional content and lists, you now have all the key tools needed to build both simple and more complex user interfaces with React. You can hide and show elements or groups of elements as needed, and you can dynamically render and update lists of elements to output lists of products, orders, or users.

Of course, that's not all that's needed to build realistic user interfaces. Adding logic for changing content dynamically is one thing, but most web apps also need CSS styling that should be applied to various DOM elements. This book is not about CSS, but the next chapter will still explore how React apps can be styled. Especially when it comes to setting and changing styles dynamically or scoping styles to specific components, there are various React-specific concepts that should be familiar to every React developer.

TEST YOUR KNOWLEDGE!

Test your knowledge about the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at https://packt.link/QyB9E.

1. What is "conditional content"?
2. Name at least two different ways of rendering JSX elements conditionally.
3. Which elegant approach can be used to define element tags conditionally?
4. What's a potential downside of using only ternary expressions (for conditional content)?
5. How can lists of data be rendered as JSX elements?
6. Why should keys be added to rendered list items?
7. Give one example each for a good and a bad key.

APPLY WHAT YOU LEARNED

You are now able to use your React knowledge to change dynamic user interfaces in a variety of ways. Besides being able to change displayed text values and numbers, you can now also hide or show entire elements (or chunks of elements) and display lists of data.

In the following sections, you will find two activities that allow you to apply your newly gained knowledge (combined with the knowledge gained in the other book chapters).

ACTIVITY 5.1: SHOWING A CONDITIONAL ERROR MESSAGE

In this activity, you'll build a basic form that allows users to enter their email address. Upon form submission, the user input should be validated and invalid email addresses (for simplicity, here email addresses that contain no `@` sign are being referred to here) should lead to an error message being shown below the form. When invalid email addresses are made valid, potentially visible error messages should be removed again.

Perform the following steps to complete this activity:

1. Build a user interface that contains a form with a label, an input field (of the text type—to make entering incorrect email addresses easier for demo purposes), and a submit button that leads to the form being submitted.

2. Collect the entered email address and show an error message below the form if the email address contains no `@` sign.

The final user interface should look and work as shown here:

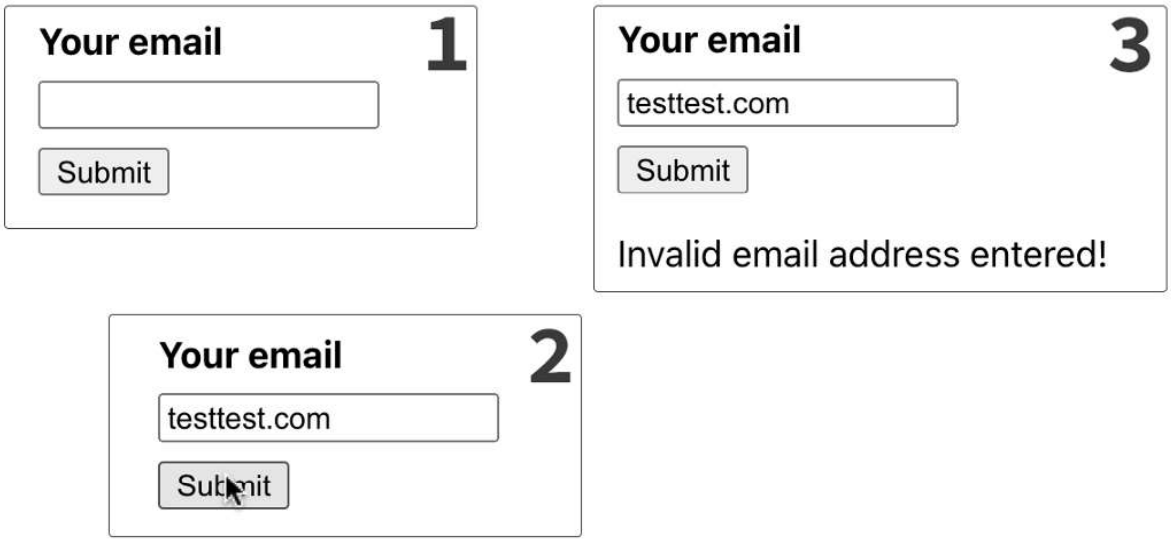

Figure 5.7: The final user interface of this activity

> **NOTE**
>
> The solution to this activity can be found via this link.

ACTIVITY 5.2: OUTPUTTING A LIST OF PRODUCTS

In this activity, you will build a user interface where a list of (dummy) products is displayed on the screen. The interface should also contain a button that, when clicked, adds another new (dummy) item to the existing list of products.

Perform the following steps to complete this activity:

1. Add a list of dummy product objects (every object should have an ID, title, and price) to a React component and add code to output these product items as JSX elements.
2. Add a button to the user interface. When clicked, the button should add a new product object to the product data list. This should then cause the user interface to update and display an updated list of product elements.

The final user interface should look and work as shown here:

Figure 5.8: The final user interface of this activity

> **NOTE**
>
> The solution to this activity can be found via this link.

6

STYLING REACT APPS

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Style JSX elements via inline style assignments or with the help of CSS classes
- Set inline and class styles, both statically and dynamically or conditionally
- Build reusable components that allow for style customization
- Utilize CSS Modules to scope styles to components
- Understand the core idea behind **`styled-components`**, a third-party CSS-in-JS library

INTRODUCTION

React.js is a frontend JavaScript library. This means that it's all about building (web) user interfaces and handling user interaction.

Up to this point, this book has extensively explored how React may be used to add interactivity to a web app. State, event handling, and dynamic content are key concepts relating to this.

Of course, websites and web apps are not just about interactivity. You could build an amazing web app that offers interactive and engaging features, and yet it may still be unpopular if it lacks appealing visuals. Presentation is key, and the web is no exception.

Therefore, like all other apps and websites, React apps and websites need proper styling. And when working with web technologies, **Cascading Style Sheets** (**CSS**) is the language of choice.

This book is not about CSS, though. It won't explain or teach you how to use CSS as there are dedicated, better resources for that (e.g., the free CSS guides at https://developer.mozilla.org). But this chapter will teach you how to combine CSS code with JSX and React concepts such as state and props. You will learn how to add styles to your JSX elements, style custom components, and make those components' styles configurable. This chapter will also teach you how to set styles dynamically and conditionally and explore popular third-party libraries that may be used for styling.

HOW DOES STYLING WORK IN REACT APPS?

Up to this point, the apps and examples presented in this book have only had minimal styling. But they at least had some basic styling, rather than no styling at all.

But how was that styling added? How can styles be added to user interface elements (such as DOM elements) when using React?

The short answer is "just as you would to non-React apps". You can add CSS styles and classes to JSX elements just as you would to regular HTML elements. And in your CSS code, you can use all the features and selectors you know from CSS. There are no React-specific changes you have to make when writing CSS code.

The code examples used up to this point (i.e., the activities or other examples hosted on GitHub) always used regular CSS styling with the help of CSS selectors to apply some basic styles to the final user interface. Those CSS rules were defined in an **`index.css`** file, which is part of every newly created React project (when using **`create-react-app`** for project creation, as shown in *Chapter 1*, *React – What and Why*).

For example, here's the **`index.css`** file used in *Activity 5.1* of the previous chapter (*Chapter 5*, *Rendering Lists and Conditional Content*):

```
body {
  margin: 2rem;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto',
'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

ul {
  list-style: none;
  margin: 0;
  padding: 0;
}

li {
  background-color: #ede6f3;
  box-shadow: 0 1px rgba(0, 0, 0, 0.2);
  padding: 0.5rem;
  margin: 1rem 0;
  max-width: 30rem;
}
```

The actual CSS code and its meaning is not important (as mentioned, this book is not about CSS). What is important, though, is the fact that this code contains no JavaScript or React code at all. As mentioned, the CSS code you write is totally independent of the fact that you're using React in your app.

The more interesting question is how that code is actually applied to the rendered web page. How is it imported into that page?

Normally, you would expect style file imports (via **`<link href="...">`**) inside of the HTML files that are being served. Since React apps are typically about building **single-page applications** (see *Chapter 1*, *React – What and Why*), you only have one HTML file—the **`index.html`** file—which can be found in the **`public/`** folder of your React project. But if you inspect that file, you won't find any **`<link href="...">`** import that would point to the **`index.css`** file (only some other **`<link>`** elements that import favicons or the web page manifest file), as you can see in the following screenshot:

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
    <title>React App</title>
  </head>
```

Figure 6.1: The <head> section of the index.html file contains no <link> import that points to the index.css file

How are the styles defined in **`index.css`** imported and applied then?

You find an **`import`** statement in the root entry file (this is the **`index.js`** file in projects generated via **`create-react-app`**):

```
import React from 'react';
import ReactDOM from 'react-dom';

import './index.css';
import App from './App';

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

The **`import './index.css';`** statement leads to the CSS file being imported and the defined CSS code being applied to the rendered web page.

It is worth noting that this is not standard JavaScript behavior. You can't import CSS files into JavaScript—at least not if you're just using vanilla JavaScript.

It works in React apps because the code is transpiled before it's loaded into the browser. Therefore, you won't find that **`import`** statement in the final JavaScript code that's executed in the browser. Instead, during the **transpilation process**, the transpiler identifies the CSS import, removes it from the JavaScript file, and injects the CSS code (or an appropriate link to the potentially bundled and optimized CSS file) into the **`index.html`** file. You can confirm this by inspecting the rendered **Document Object Model** (**DOM**) content of the loaded web page in the browser.

To do so, select the **`Elements`** tab in **`developer tools`** in Chrome, as shown below:

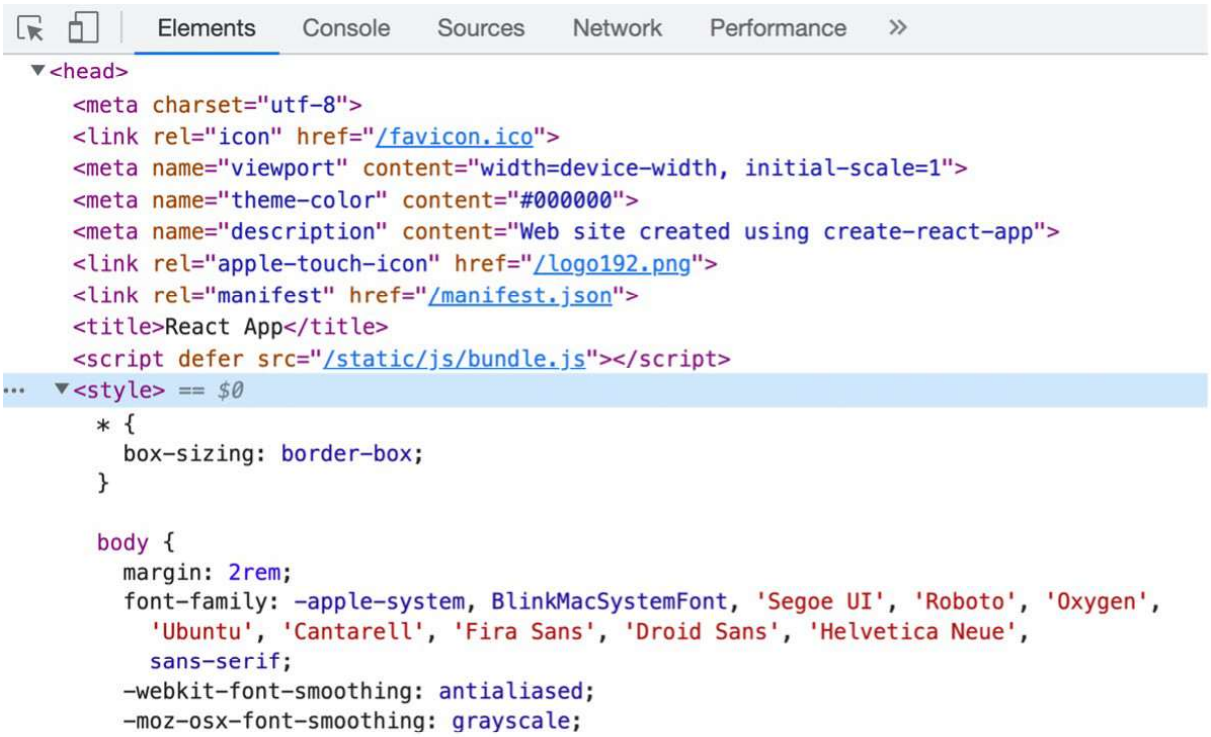

Figure 6.2: The injected CSS <style> can be found in the DOM at runtime

You can define any styles you want to apply to your HTML elements (that is, to your JSX elements in your components) directly inside of the **`index.css`** file or in any other CSS files that are imported by the **`index.css`** file.

You could also add additional CSS import statements, pointing at other CSS files, to the **`index.js`** file or any other JavaScript files (including files that store components). However, it is important to keep in mind that CSS styles are always global. No matter whether you import a CSS file in **`index.js`** or in a component-specific JavaScript file, the styles defined in that CSS file will be applied globally.

That means that styles defined in a **`goal-list.css`** file, which may be imported in a **`GoalList.js`** file, could still affect JSX elements defined in a totally different component. Later in this chapter, you will learn about techniques that allow you to avoid accidental style clashes and implement style scoping.

USING INLINE STYLES

You can use CSS files to define global CSS styles and use different CSS selectors to target different JSX elements (or groups of elements).

But even though it's typically discouraged, you can also set inline styles directly on JSX elements via the **`style`** prop.

> **NOTE**
>
> If you're wondering why inline styles are discouraged, the following discussion on Stack Overflow provides many arguments against inline styles: https://stackoverflow.com/questions/2612483/whats-so-bad-about-inline-css.

Setting inline styles in JSX code works like this:

```
function TodoItem() {
  return <li style={{color: 'red', fontSize: '18px'}}>Learn React!</li>;
};
```

In this example, the **`style`** prop is added to the **`<li>`** element (all JSX elements support the **`style`** prop) and both the **`color`** and **`size`** properties of the text are set via CSS.

This approach differs from what you would use to set inline styles when working with just HTML (instead of JSX). When using plain HTML, you would set inline styles like this:

```
<li style="color: red; font-size: 18px">Learn React!</li>
```

The difference is that the **`style`** prop expects to receive a JavaScript object that contains the style settings—not a plain string. This is something that must be kept in mind, though, since, as mentioned previously, inline styles typically aren't used that often.

Since the **`style`** object is an object and not a plain string, it is passed as a value between curly braces—just as an array, a number, or any other non-string value would have to be set between curly braces (anything between double or single quotes is treated as a string value). Therefore, it's worth noting that the preceding example does not use any kind of special "double curly braces" syntax, but instead uses one pair of curly braces to surround the non-string value and another pair to surround the object data.

Inside the **`style`** object, any CSS style properties supported by the underlying DOM element can be set. The property names are those defined for the HTML element (i.e., the same CSS property names you could target and set with vanilla JavaScript, when mutating an HTML element).

When setting styles in JavaScript code (as with the **`style`** prop shown above), JavaScript CSS property names have to be used. Those names are similar to the CSS property names you would use in CSS code but not quite the same. Differences occur for property names that consist of multiple words (e.g., **`font-size`**). When targeting such properties in JavaScript, **`camelCase`** notation must be used (**`fontSize`** instead of **`font-size`**) as JavaScript properties cannot contain dashes. Alternatively, you could wrap the property name with quotes (**`'font-size'`**).

> **NOTE**
>
> You can find more information about the HTML element style property and JavaScript CSS property names here: https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style.

SETTING STYLES VIA CSS CLASSES

As mentioned, using inline styles is typically discouraged, and therefore, CSS styles defined in CSS files (or between **`<style>`** tags in the document **`<head>`** section) are preferred.

In those CSS code blocks, you can write regular CSS code and use CSS selectors to apply CSS styles to certain elements. You could, for example, style all **`<li>`** elements on a page (no matter which component may have rendered them) like this:

```
li {
  color: red;
  font-size: 18px;
}
```

As long as this code gets added to the page (because the CSS file in which it is defined is imported in **`index.js`**, for instance), the styling will be applied.

Quite frequently, developers aim to target specific elements or groups of elements. Instead of applying some style to all **`<li>`** elements on a page, the goal could be to only target the **`<li>`** elements that are part of a specific list. Consider this HTML structure that's rendered to the page (it may be split across multiple components, but this doesn't matter here):

```
<nav>
  <ul>
    <li><a href="…">Home</a></li>
```

```
    <li><a href="…">New Goals</a></li>
  </ul>
  ...
  <h2>My Course Goals</h2>
  <ul>
    <li>Learn React!</li>
    <li>Master React!</li>
  </ul>
</nav>
```

In this example, the navigation list items should most likely not receive the same styling as the **`course goal`** list items (and vice versa).

Typically, this problem would be solved with the help of CSS classes and the class selector. You could adjust the HTML code like this:

```
<nav>
  <ul>
    <li><a href="…">Home</a></li>
    <li><a href="…">New Goals</a></li>
  </ul>
  ...
  <h2>My Course Goals</h2>
  <ul>
    <li class="goal-item">Learn React!</li>
    <li class="goal-item">Master React!</li>
  </ul>
</nav>
```

The following CSS code would then only target the **`course goal`** list items but not the navigation list items:

```
.goal-item {
  color: red;
  font-size: 18px;
}
```

And this approach almost works in React apps as well.

But if you try to add CSS classes to JSX elements as shown in the previous example, you will face a warning in the browser **developer tools**:

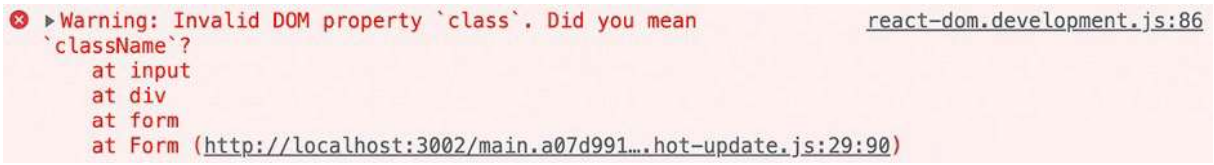

Figure 6.3: A warning output by React

As illustrated in the preceding figure, you should not add **class** as a prop but instead use **className**. Indeed, if you swap **class** for **className** as a prop name, the warning will disappear, and the class CSS styles will be applied. Hence, the proper JSX code looks like this:

```
<ul>
  <li className="goal-item">Learn React!</li>
  <li className="goal-item">>Master React!</li>
</ul>
```

But why is React suggesting you use **className** instead of **class**?

It's similar to using **htmlFor** instead of **for** when working with **<label>** objects (discussed in *Chapter 4*, *Working with Events and State*. Just like **for**, **class** is a keyword in JavaScript, and therefore, **className** is used as a prop name instead.

SETTING STYLES DYNAMICALLY

With inline styles and CSS classes (and global CSS styles in general), there are various ways of applying styles to elements. Thus far, all examples have shown static styles—that is, styles that will never change once the page has been loaded.

But while most page elements don't change their styles after a page is loaded, you also typically have some elements that should be styled dynamically or conditionally. Here are some examples:

- A **to-do** app where different **to-do** priorities receive different colors
- An input form where invalid form elements should be highlighted following an unsuccessful form submission
- A web-based game where players can choose colors for their avatars

In such cases, applying static styles is not enough, and dynamic styles should be used instead. Setting styles dynamically is straightforward. It's again just about applying key React concepts covered earlier (most importantly, those regarding the setting of dynamic values from *Chapter 2*, *Understanding React Components and JSX*, and *Chapter 4*, *Working with Events and State*).

Here's an example where the color of a paragraph is set dynamically to the color a user enters into an input field:

```
function ColoredText() {
  const [enteredColor, setEnteredColor] = useState('');

  function updateTextColorHandler(event) {
    setEnteredColor(event.target.value);
  };

  return (
    <>
      <input type="text" onChange={updateTextColorHandler}/>
      <p style={{color: enteredColor}}>This text's color changes
dynamically!</p>
    </>
  );
};
```

The text entered in the **`<input>`** field is stored in the **`enteredColor`** state. This state is then used to set the **`color`** CSS property of the **`<p>`** element dynamically. This is achieved by passing a **`style`** object with the **`color`** property set to the **`enteredColor`** value as a value to the **`style`** prop of the **`<p>`** element. The text color of the paragraph is therefore set dynamically to the value entered by the user (assuming that users enter valid CSS color values into the **`<input>`** field).

You're not limited to inline styles; CSS classes can also be set dynamically, as in the following snippet:

```
function TodoPriority() {
  const [chosenPriority, setChosenPriority] = useState('low-prio');

  function choosePriorityHandler(event) {
    setChosenPriority(event.target.value);
  };

  return (
```

```
    <>
      <p className={chosenPriority}>Chosen Priority: {chosenPriority}</p>
      <select onChange={choosePriorityHandler}>
        <option value="low-prio">Low</option>
        <option value="high-prio">High</option>
      </select>
    </>
  );
};
```

In this example, the **`chosenPriority`** state will alternate between **`low-prio`** and **`high-prio`**, depending on the drop-down selection. The state value is then output as text inside the paragraph and is also used as a dynamic CSS class name applied to the **`<p>`** element. For this to have any visual effect, there must, of course, be **`low-prio`** and **`high-prio`** CSS classes defined in some CSS file or **`<style>`** block. For example, consider the following code in **`index.css`**:

```
.low-prio {
  background-color: blue;
  color: white;
}

.high-prio {
  background-color: red;
  color: white;
}
```

CONDITIONAL STYLES

Closely related to **dynamic styles** are **conditional styles**. In fact, in the end, they are really just a special case of dynamic styles. In the previous examples, inline style values and class names were set equal to values chosen or entered by the user.

However, you can also derive styles or class names dynamically based on different conditions, as shown here:

```
function TextInput({isValid, isRecommended, inputConfig}) {
  let cssClass = 'input-default';

  if (isRecommended) {
    cssClass = 'input-recommended';
  }
```

```
  if (!isValid) {
    cssClass = 'input-invalid';
  }

  return <input className={cssClass} {...inputConfig} />
};
```

In this example, a wrapper component around the standard **<input>** element is built. (For more information about wrapper components see *Chapter 3*, *Components and Props*.) The main purpose of this wrapper component is to set some default styles for the wrapped **<input>** element. The **wrapper component** is built to provide a pre-styled input element that can be used anywhere in the app. Indeed, providing pre-styled elements is one of the most common and popular use cases for building wrapper components.

In this concrete example, the default styles are applied using CSS classes. If the **isValid** prop value is **true** and the value of the **isRecommended** prop is **false**, the **input-default** CSS class will be applied to the **<input>** element since neither of the two **if** statements become active.

If **isRecommended** is **true** (but **isValid** is **false**), the **input-recommended** CSS class would be applied. If **isValid** is **false**, the **input-invalid** class is added instead. Of course, the CSS classes must be defined in some imported CSS files (for example, in **index.css**).

Inline styles could be set in a similar way as shown in the following snippet:

```
function TextInput({isValid, isRecommended, inputConfig}) {
  let bgColor = 'black';

  if (isRecommended) {
    bgColor = 'blue';
  }

  if (!isValid) {
    bgColor = 'red';
  }

  return <input style={{backgroundColor: bgColor}} {...inputConfig} />
};
```

In this example, the background color of the **<input>** element is set conditionally, based on the values received via the **isValid** and **isRecommended** props.

COMBINING MULTIPLE DYNAMIC CSS CLASSES

In previous examples, a maximum of one CSS class was set dynamically at a time. However, it's not uncommon to encounter scenarios where multiple, dynamically derived CSS classes should be merged and added to an element.

Consider the following example:

```
function ExplanationText({children, isImportant}) {
  const defaultClasses = 'text-default text-expl';

  return <p className={defaultClasses}>{children}</p>;
}
```

Here, two CSS classes are added to **<p>** by simply combining them in one string. Alternatively, you could directly add a string with the two classes like this:

```
return <p className="text-default text-expl">{children}</p>;
```

This code will work, but what if the goal is to also add another class name to the list of classes, based on the **isImportant** prop value (which is ignored in the preceding example)?

Replacing the default list of classes would be easy, as you have learned:

```
function ExplanationText({children, isImportant}) {
  let cssClasses = 'text-default text-expl';

  if (isImportant) {
    cssClasses = 'text-important';
  }

  return <p className={cssClasses}>{children}</p>;
}
```

But what if the goal is not to replace the list of default classes? What if **text-important** should be added as a class to **<p>**, in addition to **text-default** and **text-expl**?

The **className** prop expects to receive a string value, and so passing an array of classes isn't an option. However, you can simply merge multiple classes into one string. And there are a couple of different ways of doing that:

1. String concatenation:

```
    cssClasses = cssClasses + ' text-important';
```

2. Using a template literal:

```
    cssClasses = `${cssClasses} text-important`;
```

3. Joining an array:

```
    cssClasses = [cssClasses, 'text-important'].join(' ');
```

These examples could all be used inside the **`if`** statement (**`if (isImportant)`**) to conditionally add the **`text-important`** class based on the **`isImportant`** prop value. All three approaches, as well as variations of these approaches, will work because all these approaches produce a string. In general, any approach that yields a string can be used to generate values for **`className`**.

MERGING MULTIPLE INLINE STYLE OBJECTS

When working with inline styles, instead of CSS classes, you can also merge multiple style objects. The main difference is that you don't produce a string with all values, but rather an object with all combined style values.

This can be achieved by using standard JavaScript techniques for merging multiple objects into one object. The most popular technique involves using the **spread operator**, as shown in this example:

```
function ExplanationText({children, isImportant}) {
  let defaultStyle = { color: 'black' };

  if (isImportant) {
    defaultStyle = { ...defaultStyle, backgroundColor: 'red' };
  }

  return <p style={defaultStyle}>{children}</p>;
}
```

Here, you will observe that **`defaultStyle`** is an object with a **`color`** property. If **`isImportant`** is **`true`**, it's replaced with an object that contains all the properties it had before (via the spread operator, **`...defaultStyle`**) as well as the **`backgroundColor`** property.

> **NOTE**
>
> For more information on the function and use of the spread operator, see *Chapter 5*, *Rendering Lists and Conditional Content*.

BUILDING COMPONENTS WITH CUSTOMIZABLE STYLES

As you are aware by now, components can be reused. This is supported by the fact that they can be configured via props. The same component can be used in different places on a page with different configurations to yield a different output.

Since styles can be set both statically and dynamically, you can also make the styling of your components customizable. The preceding examples already show such customization in action; for example, the **`isImportant`** prop was used in the previous example to conditionally add a **`red`** background color to a paragraph. The **`ExplanationText`** component therefore already allows for indirect style customization via the **`isImportant`** prop.

Besides this form of customization, you could also build components that accept props already holding CSS class names or style objects. For example, the following wrapper component accepts a **`className`** prop that is merged with a default CSS class (**`btn`**):

```
function Button({children, config, className}) {
  return <button {...config} className={`btn ${className}`}>{children}</
button>;
};
```

This component could be used in another component in the following way:

```
<Button config={{onClick: doSomething}} className="btn-alert">Click me!</
Button>
```

If used like this, the final **`<button>`** element would receive both the **`btn`** as well as **`btn-alert`** classes.

You don't have to use **`className`** as a prop name; any name can be used, since it's your component. However, it's not a bad idea to use **`className`** because you can then keep your mental model of setting CSS classes via **`className`** (for built-in components, you will not have that choice).

Instead of merging prop values with default CSS class names or style objects, you can also overwrite default values. This allows you to build components that come with some styling out of the box without enforcing that styling:

```
function Button({children, config, className}) {
  let cssClasses = 'btn';
  if (className) {
    cssClasses = className;
  }
```

```
  return <button {...config} className={cssClasses}>{children}</button>;
};
```

You can see how all these different concepts covered throughout this book are coming together here: props allow customization, values can be set, swapped, and changed dynamically and conditionally, and therefore, highly reusable and configurable components can be built.

CUSTOMIZATION WITH FIXED CONFIGURATION OPTIONS

Besides exposing props such as **`className`** or **`style`**, which are merged with other classes or styles defined inside a component function, you can also build components that apply different styles or class names based on other prop values.

This has been shown in the previous examples where props such as **`isValid`** or **`isImportant`** were used to apply certain styles conditionally. This way of applying styles could therefore be called "indirect styling" (though this is not an official term).

Both approaches can shine in different circumstances. For wrapper components, for example, accepting **`className`** or **`style`** props (which may be merged with other styles inside the component) enables the component to be used just like a built-in component (e.g., like the component it wraps). Indirect styling, on the other hand, can be very useful if you want to build components that provide a couple of pre-defined variations.

A good example is a text box that provides two built-in themes that can be selected via a specific prop:

```
function TextBox({children, mode}) {
  let cssClasses;

  if (mode === 'alert') {
    cssClasses = 'box-alert';
  } else if (mode === 'info') {
    cssClasses = 'box-info';
  }

  return <p className={cssClasses}>{children}</p>;
};
```

This **`TextBox`** component always yields a paragraph element. If the **`mode`** prop is set to any value other than **`'alert'`** or **`'info'`**, the paragraph doesn't receive any special styling. But if **`mode`** is equal to **`'alert'`** or **`'info'`**, specific CSS classes are added to the paragraph.

This component therefore doesn't allow direct styling via some **`className`** or **`style`** prop that would be merged, but it does offer different variations or themes that can be set with the help of a specific prop (the **`mode`** prop in this case).

THE PROBLEM WITH UNSCOPED STYLES

If you consider the different examples you've so far dealt with in this chapter, there's one specific use case that occurs quite frequently: styles are relevant to a specific component only.

For example, in the **`TextBox`** component in the previous section, **`'box-alert'`** and **`'box-info'`** are CSS classes that are likely only relevant for this specific component and its markup. If any other JSX element in the app had a **`'box-alert'`** class applied to it (even though that might be unlikely), it probably shouldn't be styled the same as the **`<p>`** element in the **`TextBox`** component.

Styles from different components could clash with each other and overwrite each other because styles are not scoped (i.e., restricted) to a specific component. CSS styles are always global, unless inline styles are used (which is discouraged, as mentioned earlier).

When working with component-based libraries such as React, this lack of scoping is a common issue. It's easy to write conflicting styles as app sizes and complexities grow (or, in other words, as more and more components are added to the code base of a React app).

That's why various solutions for this problem have been developed by members of the React community. The following are two of the most popular solutions:

- CSS Modules (supported out of the box in React projects created with **`create-react-app`**)
- Styled components (using a third-party library called **`styled-components`**)

SCOPED STYLES WITH CSS MODULES

CSS Modules is the name for an approach where individual CSS files are linked to specific JavaScript files and the components defined in those files. This link is established by transforming CSS class names, such that every JavaScript file receives its own, unique CSS class names. This transformation is performed automatically as part of the code build workflow. Therefore, a given project setup must support CSS Modules by performing the described CSS class name transformation. Projects created via **`create-react-app`** support CSS Modules by default.

CSS Modules are enabled and used by naming CSS files in a very specific and clearly defined way: **`<anything>.module.css`**. **`<anything>`** is any value of your choosing, but the **`.module`** part in front of the file extension is required as it signals (to the project build workflow) that this CSS file should be transformed according to the CSS Modules approach.

Therefore, CSS files named like this must be imported into components in a specific way:

```
import classes from './file.module.css';
```

This **`import`** syntax is different from the **`import`** syntax shown at the beginning of this section for **`index.css`**:

```
import './index.css';
```

When importing CSS files as shown in the second snippet, the CSS code is simply merged into the **`index.html`** file and applied globally. When using CSS Modules instead (first code snippet), the CSS class names defined in the imported CSS file are transformed such that they are unique for the JS file that imports the CSS file.

Since the CSS class names are transformed and are therefore no longer equal to the class names you defined in the CSS file, you import an object (**`classes`**, in the preceding example) from the CSS file. This object exposes all transformed CSS class names under keys that match the CSS class names defined by you in the CSS file. The values of those properties are the transformed class names (as strings).

Here's a complete example, starting with a component-specific CSS file (**`TextBox.module.css`**):

```
.alert {
  padding: 1rem;
  border-radius: 6px;
  background-color: salmon;
  color: red;
}

.info {
  padding: 1rem;
  border-radius: 6px;
  background-color: #d6aafa;
  color: #410474;
}
```

The JavaScript file (**`TextBox.js`**) for the component to which the CSS code should belong looks like this:

```
import classes from './TextBox.module.css';

function TextBox({ children, mode }) {
  let cssClasses;

  if (mode === 'alert') {
    cssClasses = classes.alert;
  } else if (mode === 'info') {
    cssClasses = classes.info;
  }

  return <p className={cssClasses}>{children}</p>;
}

export default TextBox;
```

> **NOTE**
>
> The full example code can also be found at https://packt.link/13nwz.

If you inspect the rendered text element in the browser developer tools, you will note that the CSS class name applied to the **`<p>`** element does not match the class name specified in the **`TextBox.module.css`** file:

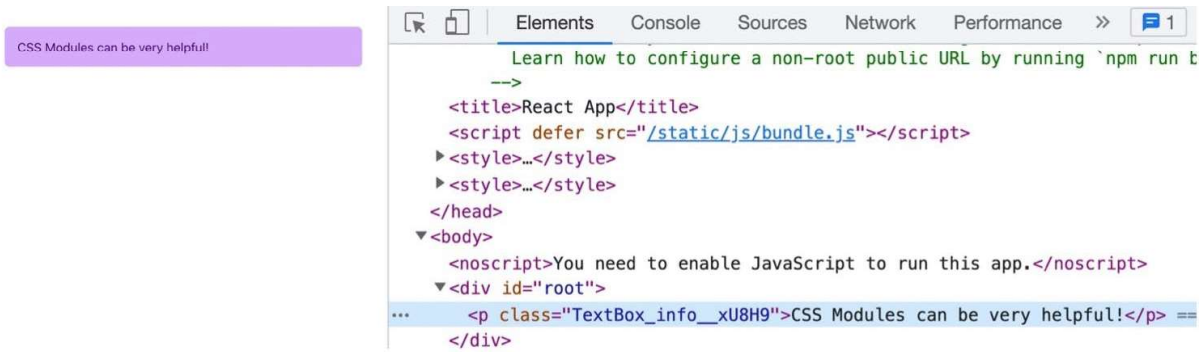

Figure 6.4: CSS class name transforms because of CSS Modules usage

This is the case because, as described previously, the class name was transformed during the build process to be unique. If any other CSS file, imported by another JavaScript file, were to define a class with the same name (**`info`** in this case), the styles would not clash and not overwrite each other because the interfering class names would be transformed into different class names before being applied to the DOM elements.

Indeed, in the example provided on GitHub, you can find another **`info`** CSS class defined in the **`index.css`** file:

```
.info {
  border: 5px solid red;
}
```

That file is still imported into **`index.js`**, and hence its styles are applied globally to the entire document. Nonetheless, the **`.info`** styles clearly aren't affecting **`<p>`** rendered by **`TextBox`** (there is no red border around the text box in *Figure 6.4*). They aren't affecting that element because it doesn't have an **`info`** class anymore; the class was renamed to **`text-box_info__vCxmZ`** by the build workflow (though the name will differ as it contains a random element).

It's also worth noting that the **`index.css`** file is still imported into **`index.js`**, as shown at the beginning of this chapter. The **`import`** statement is not changed to **`import classes from './index.css';`**, nor is the CSS file called **`index.module.css`**.

Note, too, that you can use CSS Modules to scope styles to components and can also mix the usage of CSS Modules with regular CSS files, which are imported into JavaScript files without using CSS Modules (i.e., without scoping).

One other important aspect of using CSS Modules is that you can only use CSS class selectors (that is, in your **`.module.css`** files) because CSS Modules rely on CSS classes. You can write selectors that combine classes with other selectors, such as **`input.invalid`**, but you can't add selectors that don't use classes at all in your **`.module.css`** files. For example, **`input { ... }`** or **`#some-id { ... }`** selectors wouldn't work here.

CSS Modules are a very popular way of scoping styles to (React) components, and they will be used throughout many examples for the rest of this book.

THE STYLED-COMPONENTS LIBRARY

The **`styled-components`** library is a so-called **CSS-in-JS** solution. CSS-in-JS solutions aim to remove the separation between CSS code and JavaScript code by merging them into the same file. Component styles would be defined right next to the component logic. It comes down to personal preference whether you favor separation (as enforced by using CSS files) or keeping the two languages close to each other.

Since **`styled-components`** is a third-party library that's not pre-installed in newly created React projects, you have to install this library as a first step if you want to use it. This can be done via **`npm`** (which was automatically installed together with Node.js in *Chapter 1*, *React – What and Why*):

```
npm install styled-components
```

The **`styled-components`** library essentially provides wrapper components around all built-in core components (as in, around **`p`**, **`a`**, **`button`**, **`input`**, and so on). It exposes all these wrapper components as **tagged templates**—JavaScript functions that aren't called like regular functions, but are instead executed by adding backticks (a template literal) right after the function name, for example, **`doSomething'text data'`**.

> **NOTE**
>
> Tagged templates can be confusing when you see them for the first time, especially since it's a JavaScript feature that isn't used too frequently. Chances are high that you haven't worked with them too often. It's even more likely that you have never built a custom tagged template before. You can learn more about tagged templates in this excellent documentation on MDN at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#tagged_templates.

Here is a component that imports and uses **`styled-components`** to set and scope styling:

```
import styled from 'styled-components';

const Button = styled.button`
  background-color: #370566;
  color: white;
```

```
  border: none;
  padding: 1rem;
  border-radius: 4px;
`;

export default Button;
```

This component isn't a component function but rather a constant that stores the value returned by executing the **`styled.button`** tagged template. That tagged template returns a component function that yields a **`<button>`** element. The styles passed via the tagged template (i.e., inside the template literal) are applied to that returned button element. You can see this if you inspect the button in the browser **`developer tools`**:

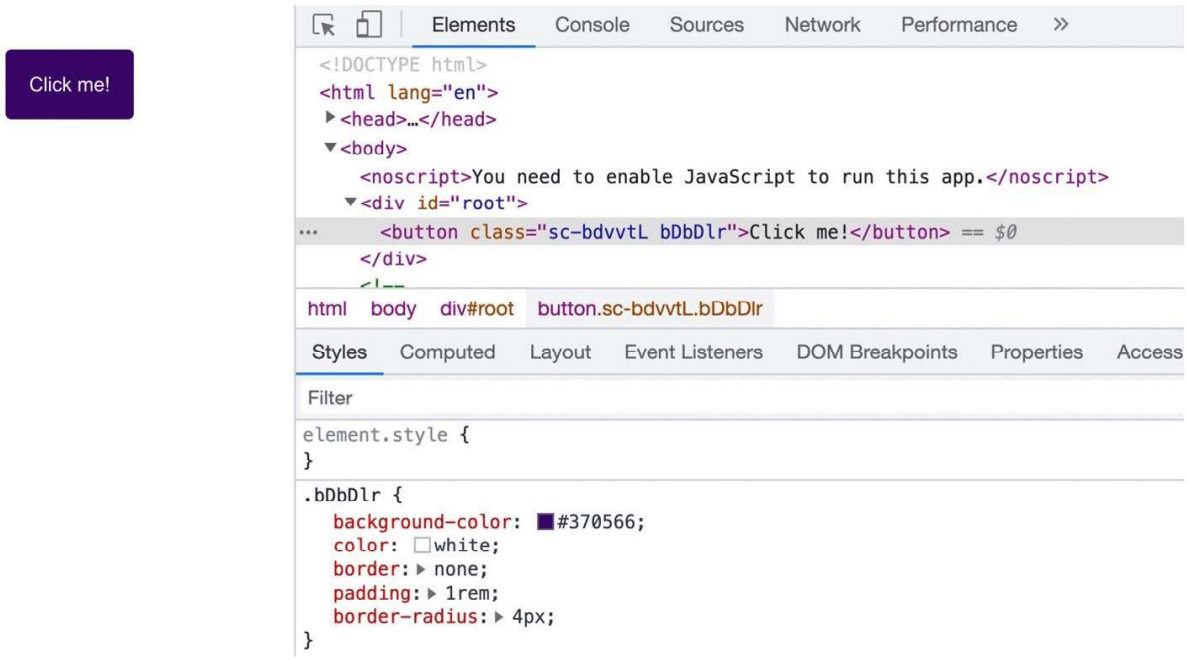

Figure 6.5: The rendered button element receives the defined component styles

In *Figure 6.5*, you can also see how the **`styled-components`** library applies your styles to the element. It extracts your style definitions from the tagged template string and injects them into a **`<style>`** element in the **`<head>`** section of the document. The injected styles are then applied via a class selector that is generated (and named) by the **`styled-components`** library. Finally, the automatically generated CSS class name is added to the element (**`<button>`**, in this case) by the library.

The components exposed by the **`styled-components`** library spread any extra props you pass to a component onto the wrapped core component. In addition, any content inserted between the opening and closing tags is also inserted between the tags of the wrapped component.

That's why the **`Button`** created previously can be used like this without adding any extra logic to it:

```
import Button from './components/button';

function App() {
  function clickHandler() {
    console.log('This button was clicked!');
  }
  return <Button onClick={clickHandler}>Click me!</Button>;
}

export default App;
```

> **NOTE**
>
> The complete example code can be found on GitHub at https://packt.link/XD6IL.

You can do more with the **`styled-components`** library. For example, you can set styles dynamically and conditionally. This book is not primarily about that library though. It's just one of many alternatives to CSS Modules. Therefore, it is recommended that you explore the official **`styled-components`** documentation if you want to learn more, which you can find at https://styled-components.com/.

USING OTHER CSS OR JAVASCRIPT STYLING LIBRARIES AND FRAMEWORKS

As mentioned, there are many third-party styling libraries that can be used in React apps. There are alternatives to **`styled-components`** or CSS Modules that help with scoping styles. But there are also other kinds of CSS libraries:

- Utility libraries that solve very specific CSS problems—independent of the fact that you're using them in a React project (for example, **`Animate.css`**, which helps with adding animations)

- CSS frameworks that provide a broad variety of pre-built CSS classes that can be applied to elements to quickly achieve a certain look (e.g., **Bootstrap** or **Tailwind**)

It is important to realize that you can use any of these CSS libraries with React. You can bring your favorite CSS framework (such as Bootstrap or Tailwind) and use the CSS classes provided by that framework on the JSX elements of your React app.

Some libraries and frameworks have React-specific extensions or specifically support React, but that does not mean that you can't use libraries that don't have this.

SUMMARY AND KEY TAKEAWAYS

- Standard CSS can be used to style React components and JSX elements.
- CSS files are typically directly imported into JavaScript files, which is possible thanks to the project build process, which extracts the CSS code and injects it into the document (the HTML file).
- As an alternative to global CSS styles (with **`element`**, **`id`**, **`class`**, or other selectors), inline styles can be used to apply styling to JSX elements.
- When using CSS classes for styling, you must use the **`className`** prop (not **`class`**).
- Styles can be set statically and dynamically or conditionally with the same syntax that is used for injecting other dynamic or conditional values into JSX code—a pair of curly braces.
- Highly configurable custom components can be built by setting styles (or CSS classes) based on prop values or by merging received prop values with other styles or class name strings.
- When using just CSS, clashing CSS class names can be a problem.
- CSS Modules solve this problem by transforming class names into unique names (per component) as part of the build workflow.
- Alternatively, third-party libraries such as **`styled-components`** can be used. This library is a CSS-in-JS library that also has the advantage or disadvantage (depending on your preference) of removing the separation between JS and CSS code.
- Other CSS libraries or frameworks can be used as well; React does not impose any restrictions regarding that.

WHAT'S NEXT?

With styling covered, you're now able to build not just functional but also visually appealing user interfaces. Even if you often work with dedicated web designers or CSS experts, you still typically need to be able to set and assign styles (dynamically) that are delivered to you.

With styling being a general concept that is relatively independent of React, the next chapter will dive back into more React-specific features and topics. You will learn about **portals** and **refs**, which are two key concepts that are built into React. You will understand which problems are solved by these concepts and how the two features are used.

TEST YOUR KNOWLEDGE!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found here: https://packt.link/vJgKI.

1. With which language are styles for React components defined?
2. Which important difference between projects with and without React has to be kept in mind when assigning classes to elements?
3. How can styles be assigned dynamically and conditionally?
4. What does "scoping" mean in the context of styling?
5. How could styles be scoped to components? Briefly explain at least one concept that helps with scoping.

APPLY WHAT YOU LEARNED

You are now not only able to build interactive user interfaces but also style those user interface elements in engaging ways. You can set and change those styles dynamically or based on conditions.

In this section, you will find two activities that allow you to apply your newly gained knowledge in combination with what you learned in previous chapters.

ACTIVITY 6.1: PROVIDING INPUT VALIDITY FEEDBACK UPON FORM SUBMISSION

In this activity, you will build a basic form that allows users to enter an email address and a password. The provided input of each input field is validated, and the validation result is stored (for each individual input field).

The aim of this activity is to add some general form styling and some conditional styling that becomes active once an invalid form has been submitted. The exact styles are up to you, but for highlighting invalid input fields, the background color of the affected input field must be changed, as well as its border color and the text color of the related label.

The steps are as follows:

1. Create a new React project and add a form component to it.
2. Output the form component in the project's root component.
3. In the form component, output a form that contains two input fields: one for entering an email address and one for entering a password.
4. Add labels to the input fields.
5. Store the entered values and check their validity upon form submission (you can be creative in forming your own validation logic).
6. Pick appropriate CSS classes from the provided **`index.css`** file (you can write your own classes as well).
7. Add them to the invalid input fields and their labels once invalid values have been submitted.

The final user interface should look like this:

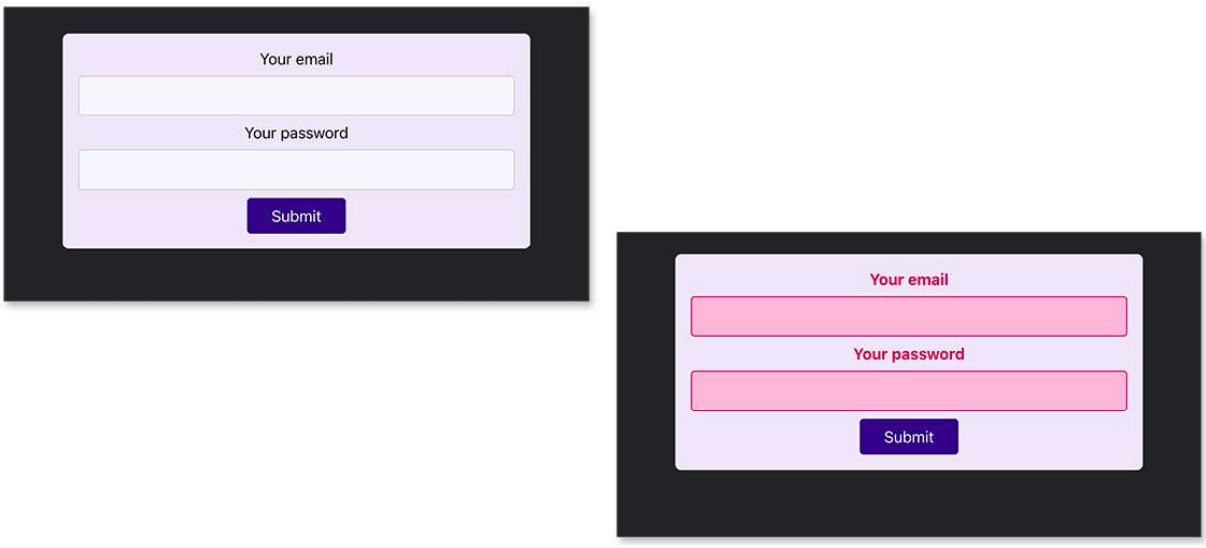

Figure 6.6: The final user interface with invalid input values highlighted in red

Since this book is not about CSS and you may not be a CSS expert, you can use the **`index.css`** file from the solution and focus on the React logic to apply appropriate CSS classes to JSX elements.

> **NOTE**
>
> The solution to this activity can be found via this link.

ACTIVITY 6.2: USING CSS MODULES FOR STYLE SCOPING

In this activity, you'll take the final app built in *Activity 6.01* and adjust it to use CSS Modules. The goal is to migrate all component-specific styles into a component-specific CSS file, which uses CSS Modules for style scoping.

The final user interface therefore looks the same as it did in the previous activity. But the styles will be scoped to the **`Form`** component so that clashing class names won't interfere with styling.

The steps are as follows:

1. Finish the previous activity or take the finished code from GitHub.
2. Identify the styles belonging specifically to the **`Form`** component and move them into a new, component-specific CSS file.
3. Change CSS selectors to class name selectors and add classes to JSX elements as needed (this is because CSS Modules require class name selectors).
4. Use the component-specific CSS file as explained throughout this chapter and assign the CSS classes to the appropriate JSX elements.

> **NOTE**
>
> The solution to this activity can be found via this link.

7

PORTALS AND REFS

LEARNING OBJECTIVES

By the end of this chapter, you will be able to do the following:

- Use direct DOM element access to interact with elements
- Expose the functions and data of your components to other components
- Control the position of rendered JSX elements in the DOM

INTRODUCTION

React.js is all about building user interfaces, and, in the context of this book, it's about building web user interfaces specifically.

Web user interfaces are ultimately all about the **Document Object Model** (**DOM**). You can use JavaScript to read or manipulate the DOM. This is what allows you to build interactive websites: you can add, remove, or edit DOM elements after a page was loaded. This can be used to add or remove overlay windows or to read values entered into input fields.

This was already discussed in *Chapter 1*, *React – What and Why*, and, as you learned there, React is used to simplify this process. Instead of manipulating the DOM or reading values from DOM elements manually, you can use React to describe the desired state. React then takes care of the steps needed to achieve this desired state.

However, there are scenarios and use cases wherein, despite using React, you still want to be able to directly reach out to specific DOM elements—for example, to read a value entered by a user into an input field or if you're not happy with the position of a newly inserted element in the DOM that was chosen by React.

React provides certain functionalities that help you in exactly these kinds of situations: **Portals** and **Refs**.

A WORLD WITHOUT REFS

Consider the following example: you have a website that renders an input field, requesting a user's email address. It could look something like this:

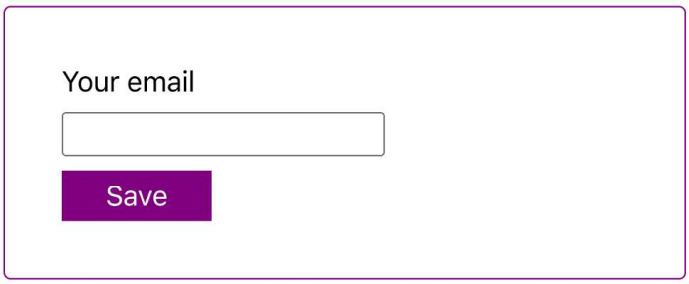

Figure 7.1: An example form with an email input field

The code for the component that's responsible for rendering the form and handling the entered email address value might look like this:

```
function EmailForm() {
  const [enteredEmail, setEnteredEmail] = useState('');

  function updateEmailHandler(event) {
  setEnteredEmail(event.target.value);
  }

  function submitFormHandler(event) {
  event.preventDefault();
  // could send enteredEmail to a backend server
  }

  return (
  <form className={classes.form} onSubmit={submitFormHandler}>
    <label htmlFor="email">Your email</label>
    <input type="email" id="email" onChange={updateEmailHandler} />
    <button>Save</button>
  </form>
  );
}
```

As you can see, this example uses the **`useState()`** Hook, combined with the **`change`** event, to register keystrokes in the **`email`** input field and store the entered value.

This code works fine and there is nothing wrong with having this kind of code in your application. But adding the extra event listener and state, as well as adding the function to update the state whenever the **`change`** event is triggered, is quite a bit of boilerplate code for one simple task: getting the entered email address.

The preceding code snippet does nothing else with the email address other than submit it. In other words, the only reason for using the **`enteredEmail`** state in the example is to read the entered value.

In scenarios such as this, quite a bit of code could be saved if you fell back to some vanilla JavaScript logic:

```
const emailInputEl = document.getElementById('email');
const enteredEmailVal = emailInputEl.value;
```

These two lines of code (which could be merged into one line theoretically) allow you to get hold of a DOM element and read the currently stored value.

The problem with this kind of code is that it does not use React. And if you're building a React app, you should really stick to React when working with the DOM. Don't start blending your own vanilla JavaScript code *that accesses the DOM* into the React code.

This can lead to unintended behaviors or bugs, especially if you start manipulating the DOM. It could lead to bugs because React would not be aware of your changes in that case; the actual rendered UI would not be in sync with React's assumed UI. Even if you're just reading from the DOM, it's a good habit to not even start merging vanilla JavaScript DOM access methods with your React code.

To still allow you to get hold of DOM elements and read values, as shown above, React gives you a special concept that you can use: **Refs**.

Ref stands for reference, and it's a feature that allows you to reference (i.e., get hold of) elements from inside a React component. The preceding vanilla JavaScript code would do the same (it also gives you access to a rendered element), but when using refs, you can get access without mixing vanilla JavaScript code into your React code.

Refs can be created using a special React Hook called the **`useRef()`** Hook.

This Hook can be executed to generate a **`ref`** object:

```
import { useRef } from 'react';

function EmailForm() {
  const emailRef = useRef();

  // other code ...
};
```

This generated ref object, **`emailRef`** in the preceding example, can then be assigned to any JSX element. This assignment is done via a special prop (the **`ref`** prop) that is automatically supported by every JSX element:

```
return (
  <form className={classes.form} onSubmit={submitFormHandler}>
  <label htmlFor="email">Your email</label>
  <input
    ref={emailRef}
    type="email"
    id="email"
```

```
    />
    <button>Save</button>
    </form>
);
```

Just like the **key** prop introduced in *Chapter 5*, *Rendering Lists and Conditional Content*, the **ref** prop is provided by React. The **ref** prop wants a ref object, e.g., one that was created via **useRef()**.

With that ref object created and assigned, you can then use it to get access to the connected JSX element (to the **<input>** element in this example). There's just one important thing to note: to get hold of the connected element, you must access a special **current** prop on the created ref object. This is required because React stores the value assigned to the ref object in a nested object, accessible via the **current** property, as shown here:

```
function submitFormHandler(event) {
  event.preventDefault();
  const enteredEmail = emailRef.current.value; // .current is mandatory!
  // could send enteredEmail to a backend server
};
```

emailRef.current yields the underlying DOM object that was rendered for the connected JSX element. In this case, it therefore allows access to the input element DOM object. Since that DOM object has a **value** property, this **value** property can be accessed without issue.

> **NOTE**
>
> For further information on this topic see https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#attributes.

With this kind of code, you can read the value from the DOM element without having to use **useState()** and an event listener. The final component code therefore becomes quite a bit leaner:

```
function EmailForm() {
  const emailRef = useRef();

  function submitFormHandler(event) {
  event.preventDefault();
  const enteredEmail = emailRef.current.value;
```

```
  // could send enteredEmail to a backend server
  }

  return (
  <form className={classes.form} onSubmit={submitFormHandler}>
    <label htmlFor="email">Your email</label>
    <input
      ref={emailRef}
      type="email"
      id="email"
    />
    <button>Save</button>
  </form>
  );
}
```

REFS VERSUS STATE

Since refs can be used to get quick and easy access to DOM elements, the question that might come up is whether you should always use refs instead of state.

The clear answer to this question is "no".

Refs can be a very good alternative in use cases like the one shown above, when you need read access to an element. This is very often the case when dealing with user input. In general, refs can replace state if you're just accessing some value to read it when some function (a form submit handler function, for example) is executed. As soon as you need to change values and those changes must be reflected in the UI (for example, by rendering some conditional content), refs are out of the game.

In the example above, if, besides getting the entered value, you'd also like to reset (i.e., clear) the email input after the form was submitted, you should use state again. While you could reset the input with the help of a ref, you should not do that. You would start manipulating the DOM and only React should do that—with its own, internal methods, based on the declarative code you provide to React.

You should not reset the email input like this:

```
function EmailForm() {
  const emailRef = useRef();

  function submitFormHandler(event) {
  event.preventDefault();
```

```
  const enteredEmail = emailRef.current.value;
  // could send enteredEmail to a backend server

  emailRef.current.value = ''; // resetting the input value
  }

  return (
  <form className={classes.form} onSubmit={submitFormHandler}>
    <label htmlFor="email">Your email</label>
    <input
      ref={emailRef}
      type="email"
      id="email"
    />
    <button>Save</button>
  </form>
  );
}
```

Instead, you should reset it by using React's state concept and by following the declarative approach embraced by React:

```
function EmailForm() {
  const [enteredEmail, setEnteredEmail] = useState('');

  function updateEmailHandler(event) {
  setEnteredEmail(event.target.value);
  }

  function submitFormHandler(event) {
  event.preventDefault();
  // could send enteredEmail to a backend server

  // reset by setting the state + using the value prop below
  setEnteredEmail('');
  }

  return (
  <form className={classes.form} onSubmit={submitFormHandler}>
    <label htmlFor="email">Your email</label>
    <input
```

```
      type="email"
      id="email"
      onChange={updateEmailHandler}
      value={enteredEmail}
    />
    <button>Save</button>
  </form>
  );
}
```

> **NOTE**
>
> As a rule, you should simply try to avoid writing imperative code in React projects. Instead, tell React how the final UI should look and let React figure out how to get there.
>
> Reading values via refs is an acceptable exception and manipulating DOM elements (with or without refs) should be avoided. A rare exception would be scenarios such as calling **`focus()`** on an input element DOM object as methods like **`focus()`** don't typically cause any DOM changes that could break the React app.

USING REFS FOR MORE THAN DOM ACCESS

Accessing DOM elements (for reading values) is one of the most common use cases for using refs. As shown above, it can help you reduce code in certain situations.

But refs are more than just "element connection bridges"; they are objects that can be used to store all kinds of values—not just pointers at DOM objects. You can, for example, also store strings or numbers or any other kind of value in a ref:

```
const passwordRetries = useRef(0);
```

You can pass an initial value to **`useRef()`** (**0** in this example) and then access or change that value at any point in time, inside of the component to which the ref belongs:

```
passwordRetries.current = 1;
```

However, you still have to use the **`current`** property to read and change the stored value, because, as mentioned above, this is where React will store the actual value that belongs to the Ref.

This can be useful for storing data that should "survive" component re-evaluations. As you learned in *Chapter 4*, *Working with Events and State*, React will execute component functions every time the state of a component changes. Since the function is executed again, any data stored in function-scoped variables would be lost. Consider the following example:

```
function Counters() {
  const [counter1, setCounter1] = useState(0);
  const counterRef = useRef(0);
  let counter2 = 0;

  function changeCountersHandler() {
  setCounter1(1);
  counter2 = 1;
  counterRef.current = 1;
  };

  return (
  <>
    <button onClick={changeCountersHandler}>Change Counters</button>
    <ul>
      <li>Counter 1: {counter1}</li>
      <li>Counter 2: {counter2}</li>
      <li>Counter 3: {counterRef.current}</li>
    </ul>
  </>
  );
};
```

In this example, counters 1 and 3 would change to 1 once the button is clicked. However, counter 2 would remain zero, even though the **`counter2`** variable gets changed to a value of 1 in **`changeCountersHandler`** as well:

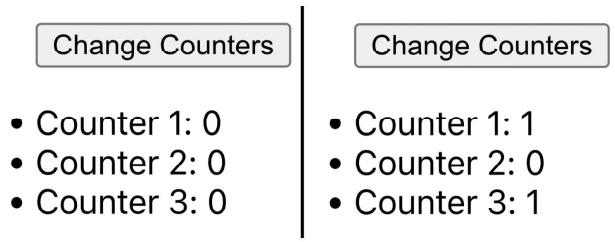

Figure 7.2: Only two of the three counter values changed

In this example, it should be expected that the state value changes, and the new value is reflected in the updated user interface. That is the whole idea behind state, after all.

The ref (**`counterRef`**) also keeps its updated value across component re-evaluations, though. That's the behavior described above: refs are not reset or cleared when the surrounding component function is executed again. The vanilla JavaScript variable (**`counter2`**) does not keep its value. Even though it is changed in **`changeCountersHandler`**, a new variable is initialized when the component function is executed again; thus the updated value (**`1`**) is lost.

In this example, it might again look like refs can replace state, but the example actually shows very well why that is **not** the case. Try replacing **`counter1`** with another ref (so that there is no state value left in the component) and clicking the button:

```
import { useRef } from 'react';

function Counters() {
  const counterRef1 = useRef(0);
  const counterRef2 = useRef(0);
  let counter2 = 0;

  function changeCountersHandler() {
  counterRef1.current = 1;
  counter2 = 1;
  counterRef2.current = 1;
  }

  return (
  <>
    <button onClick={changeCountersHandler}>Change Counters</button>
    <ul>
      <li>Counter 1: {counterRef1.current}</li>
      <li>Counter 2: {counter2}</li>
      <li>Counter 3: {counterRef2.current}</li>
    </ul>
  </>
  );
```

```
export default Counters;
```

Nothing will change on the page because, while the button click is registered and the **`changeCountersHandler`** function is executed, no state change is initiated. And state changes (initiated via the **`setXYZ`** state updating function calls) are the triggers that cause React to re-evaluate a component. Changes to ref values do **not** do that.

Therefore, if you have data that should survive component re-evaluations but should not be managed as state (because changes to that data should not cause the component to be re-evaluated when changed), you could use a ref:

```
const passwordRetries = useRef(0);
// later in the component ...
passwordRetries.current = 1; // changed from 0 to 1
// later ...
console.log(passwordRetries.current); // prints 1, even if the component changed
```

This is not a feature that's used frequently, but it can be helpful from time to time. In all other cases, use normal state values.

FORWARDING REFS

Refs cannot just be used to access DOM elements. You can also use them to access React components—including your own components.

This can sometimes be useful. Consider this example: you have a **`<Form>`** component that contains a nested **`<Preferences>`** component. The latter component is responsible for displaying three checkboxes, asking the user for their newsletter preferences:

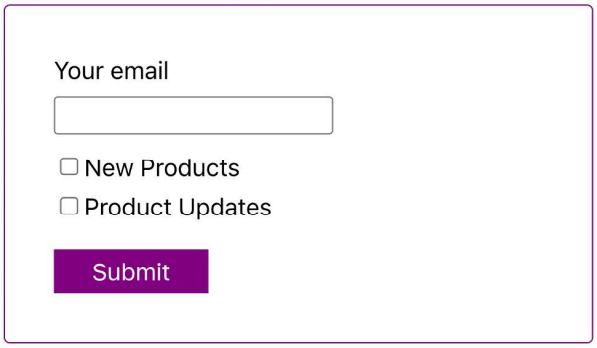

Figure 7.3: A newsletter sign-up form that shows two checkboxes to set newsletter preferences

The code of the **`Preferences`** component could look like this:

```
function Preferences() {
  const [wantsNewProdInfo, setWantsNewProdInfo] = useState(false);
  const [wantsProdUpdateInfo, setWantsProdUpdateInfo] = useState(false);

  function changeNewProdPrefHandler() {
    setWantsNewProdInfo((prevPref) => !prevPref);
  }

  function changeUpdateProdPrefHandler() {
    setWantsProdUpdateInfo((prevPref) => !prevPref);
  }

  return (
    <div className={classes.preferences}>
      <label>
        <input
          type="checkbox"
          id="pref-new"
          checked={wantsNewProdInfo}
          onChange={changeNewProdPrefHandler}
        />
        <span>New Products</span>
      </label>
      <label>
        <input
          type="checkbox"
          id="pref-updates"
          checked={wantsProdUpdateInfo}
          onChange={changeUpdateProdPrefHandler}
        />
        <span>Product Updates</span>
      </label>
    </div>
  );
};
```

As you can see, it's a basic component that essentially outputs the two checkboxes, adds some styling, and keeps track of the selected checkbox via state.

The **`Form`** component code could look like this:

```
function Form() {
  function submitHandler(event) {
    event.preventDefault();
  }

  return (
    <form className={classes.form} onSubmit={submitHandler}>
      <div className={classes.formControl}>
        <label htmlFor="email">Your email</label>
        <input type="email" id="email" />
      </div>
      <Preferences />
      <button>Submit</button>
    </form>
  );
}
```

Now imagine that upon form submission (inside of the **`submitHandler`** function), the **`Preferences`** should be reset (i.e., no checkbox is selected anymore). In addition, prior to resetting, the selected values should be read and used in the **`submitHandler`** function.

This would be straightforward if the checkboxes were not put into a separate component. If the entire code and JSX markup reside in the **`Form`** component, state could be used in that component to read and change the values. But this is not the case in this example and rewriting the code just because of this problem sounds like an unnecessary restriction.

Fortunately, Refs can help in this situation.

You can expose features (for example, functions or state values) of a component to other components by forwarding refs. Refs can essentially be used as a "communication device" between two components, just as they were used as a "communication device" with a DOM element in the previous sections.

To forward Refs, you must wrap the receiving component (**`Preferences`**, in this example) with a special function provided by React: **`forwardRef()`**.

This can be done like this:

```
const Preferences = forwardRef((props, ref) => {
  // component code ...
});

export default Preferences;
```

This looks slightly different than all the other components in this book because an arrow function is used instead of the **`function`** keyword. You can always use arrow functions instead of "normal functions", but here it's helpful to switch as it makes wrapping the function with **`forwardRef()`** very easy. Alternatively, you could stick to the **`function`** keyword and wrap the function like this:

```
function Preferences(props, ref) {
 // component code ...
};

export default forwardRef(Preferences);
```

It is up to you which syntax you prefer. Both work and both are commonly used in React projects.

The interesting part about this code is that the component function now receives **two** parameters instead of one. Besides receiving **`props`**, which component functions always do, it now also receives a special **`ref`** parameter. And this parameter is only received because the component function is wrapped with **`forwardRef()`**.

This **`ref`** parameter will contain any **`ref`** value set by the component using the **`Preferences`** component. For example, the **`Form`** component could set a **`ref`** parameter on **`Preferences`** like this:

```
function Form() {
  const preferencesRef = useRef({});

  function submitHandler(event) {
    // other code ...
  }

  return (
    <form className={classes.form} onSubmit={submitHandler}>
      <div className={classes.formControl}>
        <label htmlFor="email">Your email</label>
```

```
        <input type="email" id="email" />
      </div>
      <Preferences ref={preferencesRef} />
      <button>Submit</button>
    </form>
  );
}
```

Again, **`useRef()`** is used to create a **`ref`** object (**`preferencesRef`**), and that object is then passed via the special **`ref`** prop to the **`Preferences`** component. The created ref receives a default value of an empty object (**`{}`**); it's this object that can then be accessed via **`ref.current`**. In the **`Preferences`** component, the **`ref`** value is not received as a regular prop on the **`props`** parameter, though. Instead, it's received via this second **`ref`** parameter, which exists because of **`forwardRef()`**.

But what's the benefit of that? How can this **`preferencesRef`** object now be used inside **`Preferences`** to enable cross-component interaction?

Since **`ref`** is an object that is never replaced, even if the component in which it was created via **`useRef()`** is re-evaluated (see previous sections above), the receiving component can assign properties and methods to that object and the creating component can then use these methods and properties. The **`ref`** object is therefore used as a communication vehicle.

In this example, the **`Preferences`** component could be changed like this to use the **`ref`** object:

```
const Preferences = forwardRef((props, ref) => {
  const [wantsNewProdInfo, setWantsNewProdInfo] = useState(false);
  const [wantsProdUpdateInfo, setWantsProdUpdateInfo] = useState(false);

  function changeNewProdPrefHandler() {
    setWantsNewProdInfo((prevPref) => !prevPref);
  }

  function changeUpdateProdPrefHandler() {
    setWantsProdUpdateInfo((prevPref) => !prevPref);
  }

  function reset() {
    setWantsNewProdInfo(false);
    setWantsProdUpdateInfo(false);
  }
```

```
  ref.current.reset = reset;
  ref.current.selectedPreferences = {
    newProductInfo: wantsNewProdInfo,
    productUpdateInfo: wantsProdUpdateInfo,
  };

  // also return JSX code (has not changed) ...
});
```

In **`Preferences`**, both the state values and a pointer at a newly added **`reset`** function are stored in the received **`ref`** object. **`ref.current`** is used since the object created by React (when using **`useRef()`**) always has such a **`current`** property, and that property should be used for storing the actual values in **`ref`**.

Since **`Preferences`** and **`Form`** operate on the same object that's stored in the **`ref`** object, the properties and methods assigned to the object in **`Preferences`** can also be used in **`Form`**:

```
function Form() {
  const preferencesRef = useRef({});

  function submitHandler(event) {
    event.preventDefault();

    console.log(preferencesRef.current.selectedPreferences); // reading a
value
    preferencesRef.current.reset(); // executing a function stored in
Preferences
  }

  return (
    <form className={classes.form} onSubmit={submitHandler}>
      <div className={classes.formControl}>
        <label htmlFor="email">Your email</label>
        <input type="email" id="email" />
      </div>
      <Preferences ref={preferencesRef} />
      <button>Submit</button>
    </form>
  );
}
```

By using forward refs like this, a parent component (**`Form`**, in this case) is able to use some child component (for instance, **`Preferences`**) in an imperative way—meaning properties can be accessed and methods called to manipulate the child component (or, to be precise, to trigger some internal functions and behavior inside the child component).

CONTROLLED VERSUS UNCONTROLLED COMPONENTS

Forwarding refs is a method that can be used to allow the **`Form`** and **`Preferences`** components to work together. But even though it might look like an elegant solution at first, it should typically not be your default solution for this kind of problem.

Using forward refs, as shown in the example above, leads to more imperative code in the end. It's imperative code because instead of defining the desired user interface state via JSX (which would be declarative), individual step-by-step instructions are added in JavaScript.

If you revisit *Chapter 1*, *React – What and Why* (the "*The Problem with Vanilla JavaScript*" section), you'll see that code such as **`preferencesRef.current.reset()`** (from the example above) looks quite similar to instructions such as **`buttonElement.addEventListener(...)`** (example from *Chapter 1*). Both examples use imperative code and should be avoided for the reasons mentioned in *Chapter 1* (writing step-by-step instructions leads to inefficient micro-management and often unnecessarily complex code).

Inside the **`Form`** component, the **`reset()`** function of **`Preferences`** is invoked. Hence the code describes the desired action that should be performed (instead of the expected outcome). Typically, when working with React you should strive for describing the desired (UI) state instead. Remember, when working with React, that you should write declarative, rather than imperative, code.

When using refs to read or manipulate data as shown in the previous sections of this chapter, you are building so-called **uncontrolled components**. The components are considered "uncontrolled" because React is not directly controlling the UI state. Instead, values are read from other components or the DOM. It's therefore the DOM that controls the state (e.g., a state such as the value entered by a user into an input field).

As a React developer, you should try to minimize the use of uncontrolled components. It's absolutely fine to use refs to save some code if you only need to gather some entered values. But as soon as your UI logic becomes more complex (for example, if you also want to clear user input), you should go for **controlled components** instead.

And doing so is quite straightforward: a component becomes controlled as soon as React manages the state. In the case of the **`EmailForm`** component from the beginning of this chapter, the controlled component approach was shown before refs were introduced. Using **`useState()`** for storing the user's input (and updating the state with every keystroke) meant that React was in full control of the entered value.

For the previous example, the **`Form`** and **`Preferences`** components, switching to a controlled component approach could look like this:

```
function Preferences({newProdInfo, prodUpdateInfo, onUpdateInfo}) {
  return (
    <div className={classes.preferences}>
      <label>
        <input
          type="checkbox"
          id="pref-new"
          checked={newProdInfo}
          onChange={onUpdateInfo.bind(null, 'pref-new')}
        />
        <span>New Products</span>
      </label>
      <label>
        <input
          type="checkbox"
          id="pref-updates"
          checked={prodUpdateInfo}
          onChange={onUpdateInfo.bind(null, 'pref-updates')}
        />
        <span>Product Updates</span>
      </label>
    </div>
  );
};
```

In this example, the **`Preferences`** component stops managing the checkbox state and instead receives props from its parent component (the **`Form`** component).

`bind()` is used on the **`onUpdateInfo`** prop (which will receive a function as a value) to "pre-configure" the function for future execution. **`bind()`** is a default JavaScript method that can be called on any JavaScript function to control which arguments will be passed to that function once it's invoked in the future.

> **NOTE**
>
> You can learn more about this JavaScript feature at https://academind.com/tutorials/function-bind-event-execution.

The **`Form`** component now manages the checkbox states, even though it doesn't directly contain the checkbox elements. But it now begins to control the **`Preferences`** component and its internal state, hence turning **`Preferences`** into a controlled component instead of an uncontrolled one:

```
function Form() {
  const [wantsNewProdInfo, setWantsNewProdInfo] = useState(false);
  const [wantsProdUpdateInfo, setWantsProdUpdateInfo] = useState(false);

  function updateProdInfoHandler(selection) {
    // using one shared update handler function is optional
    // you could also use two separate functions (passed to Preferences)
as props
    if (selection === 'pref-new') {
      setWantsNewProdInfo((prevPref) => !prevPref);
    } else if (selection === 'pref-update') {
      setWantsProdUpdateInfo((prevPref) => !prevPref);
    }
  }

  function reset() {
    setWantsNewProdInfo(false);
    setWantsProdUpdateInfo(false);
  }

  function submitHandler(event) {
    event.preventDefault();
    // state values can be used here
    reset();
  }

  return (
    <form className={classes.form} onSubmit={submitHandler}>
      <div className={classes.formControl}>
        <label htmlFor="email">Your email</label>
```

```
        <input type="email" id="email" />
      </div>
      <Preferences
        newProdInfo={wantsNewProdInfo}
        prodUpdateInfo={wantsProdUpdateInfo}
        onUpdateInfo={updateProdInfoHandler}
      />
      <button>Submit</button>
    </form>
  );
}
```

`Form` manages the checkbox selection state, including resetting the state via the **`reset()`** function, and passes the managed state values (**`wantsNewProdInfo`** and **`wantsProdUpdateInfo`**) as well as the **`updateProdInfoHandler`** function, which updates the state values, to **`Preferences`**. The **`Form`** component now controls the **`Preferences`** component.

If you go through the two code snippets above, you'll notice that the final code is once again purely declarative. Across all components, state is managed and used to declare the expected user interface.

It is considered a good practice to go for controlled components in most cases. If you are only extracting some entered user input values, however, then using refs and creating an uncontrolled component is absolutely fine.

REACT AND WHERE THINGS END UP IN THE DOM

Leaving the topic of refs, there is one other important React feature that can help with influencing (indirect) DOM interaction: **Portals**.

When building user interfaces, you sometimes need to display elements and content conditionally. This was already covered in *Chapter 5*, *Rendering Lists and Conditional Content*. When rendering conditional content, React will inject that content into the place in the DOM where the overall component (in which the conditional content is defined) is located.

For example, when showing a conditional error message below an input field, that error message is right below the input in the DOM:

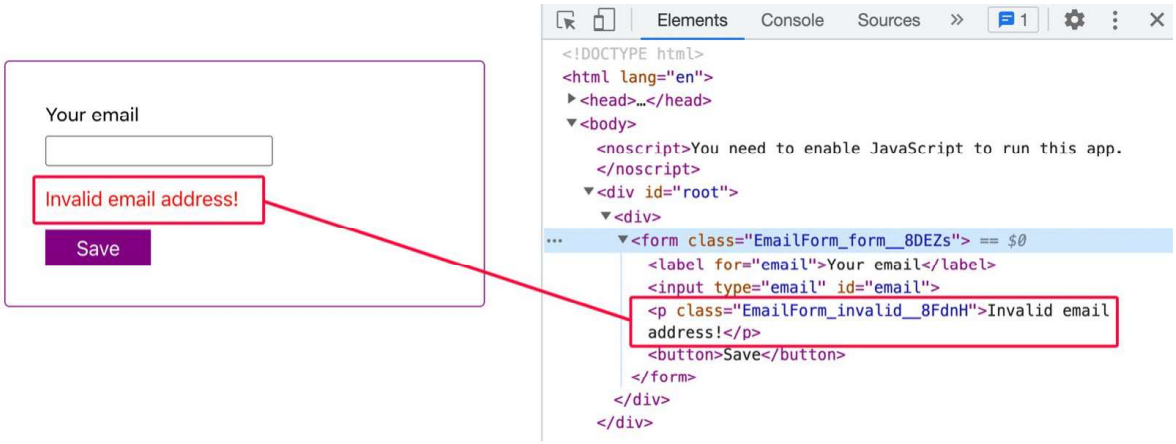

Figure 7.4: The error message DOM element sits right below the <input> it belongs to

This behavior makes sense. Indeed, it would be pretty irritating if React were to start inserting DOM elements in random places. But in some scenarios, you may prefer a (conditional) DOM element to be inserted in a different place in the DOM—for example, when working with overlay elements such as error dialogs.

In the preceding example, you could add logic to ensure that some error dialog is presented to the user if the form is submitted with an invalid email address. This could be implemented with logic similar to the **`"Invalid email address!"`** error message, and therefore the dialog element would, of course, also be injected dynamically into the DOM:

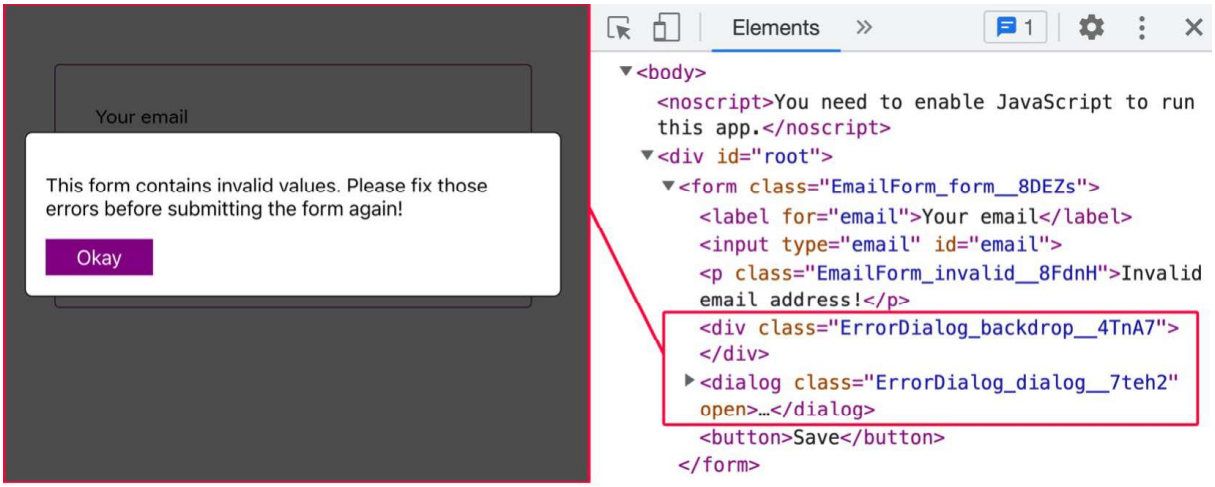

Figure 7.5: The error dialog and its backdrop are injected into the DOM

In this screenshot, the error dialog is opened as an overlay above a backdrop element, which is itself added such that it acts as an overlay to the rest of the user interface.

> **NOTE**
>
> The appearance is handled entirely by CSS, and you can take a look at the complete project (including the styling) here: https://packt.link/wFmZ7.

This example works and looks fine. However, there is room for improvement.

Semantically, it doesn't entirely make sense to have the overlay elements injected somewhere nested into the DOM next to the **`<input>`** element. It would make more sense for overlay elements to be closer to the root of the DOM (in other words, to be direct child elements of **`<div id="root">`** or even **`<body>`**), instead of being children of **`<form>`**. And it's not just a semantic problem. If the example app contains other overlay elements, those elements might clash with each other like this:

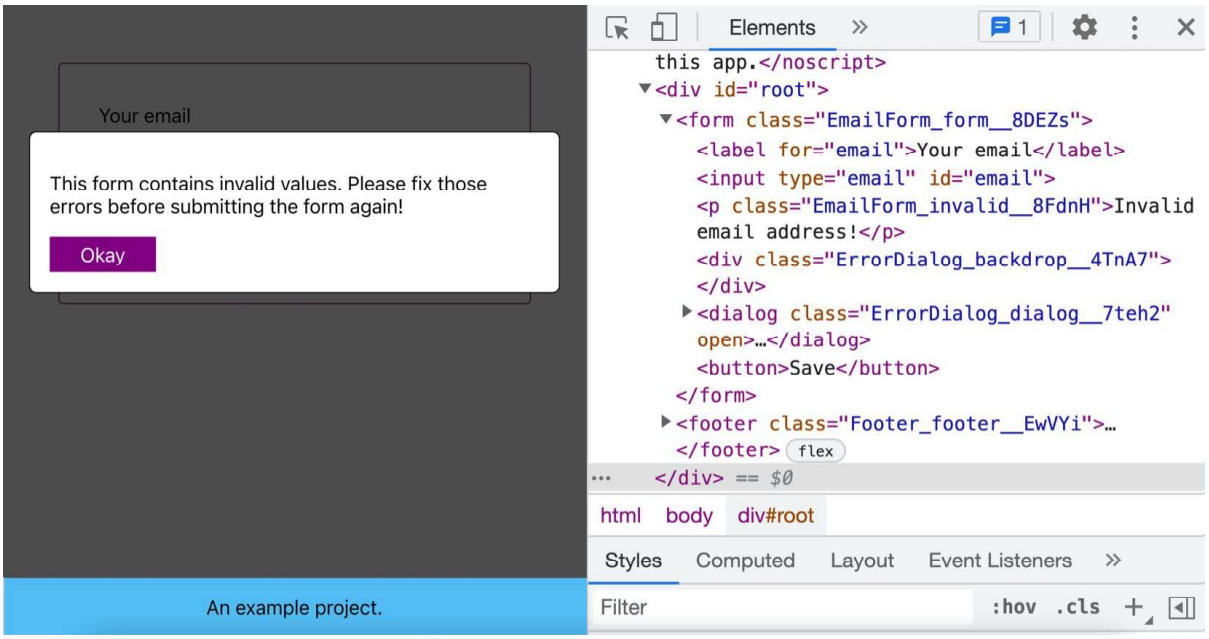

Figure 7.6: The <footer> element at the bottom is visible above the backdrop

In this example, the **`<footer>`** element at the bottom ("An example project") is not hidden or grayed out by the backdrop that belongs to the error dialog. The reason for that is that the footer also has some CSS styling attached that turns it into a de facto overlay (because of **`position: fixed`** and **`left`** + **`bottom`** being used in its CSS styles).

As a solution to this problem, you could tweak some CSS styles and, for example, use the **`z-index`** CSS property to control overlay levels. However, it would be a cleaner solution if the overlay elements (i.e., the **`<div>`** backdrop and the **`<dialog>`** error elements) were inserted into the DOM in a different place—for example, at the very end of the **`<body>`** element (but as direct children to **`<body>`**).

And that's exactly the kind of problem React **Portals** help you solve.

PORTALS TO THE RESCUE

A **Portal**, in React's world, is a feature that allows you to instruct React to insert a DOM element in a different place thanwhere it would normally be inserted.

Considering the example shown above, this portal feature can be used to "tell" React to not insert the **`<dialog>`** error and the **`<div>`** backdrop that belongs to the dialog inside the **`<form>`** element, but to instead insert those elements at the end of the **`<body>`** element.

To use this portal feature, you first must define a place wherein elements can be inserted (an "injection hook"). This can be done in the HTML file that belongs to the React app (i.e., **`public/index.html`**). There, you can add a new element (for example, a **`<div>`** element) somewhere in the **`<body>`** element:

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <div id="dialogs"></div>
</body>
```

In this case, a **`<div id="dialogs">`** element is added at the end of the **`<body>`** element to make sure that any components (and their styles) inserted in that element are evaluated last. This will ensure that their styles take a higher priority and overlay elements inserted into **`<div id="dialogs">`** would not be overlaid by other content coming earlier in the DOM. Adding and using multiple hooks would be possible, but for this example, only one "injection point" is needed. You can also use HTML elements other than **`<div>`** elements.

With the **`index.html`** file adjusted, React can be instructed to render certain JSX elements (i.e., components) in a specified hook via the **`createPortal()`** function of **`react-dom`**:

```
import { createPortal } from 'react-dom';

import classes from './ErrorDialog.module.css';

function ErrorDialog({ onClose }) {
  return createPortal(
    <>
      <div className={classes.backdrop}></div>
      <dialog className={classes.dialog} open>
        <p>
          This form contains invalid values. Please fix those errors
before
          submitting the form again!
        </p>
        <button onClick={onClose}>Okay</button>
      </dialog>
    </>,
    document.getElementById(,dialogs')
  );
}

export default ErrorDialog;
```

Inside this **`ErrorDialog`** component, which is rendered conditionally by another component (the **`EmailForm`** component, the example code for which is available on GitHub), the returned JSX code is wrapped by **`createPortal()`**. **`createPortal()`** takes two arguments: the JSX code that should be rendered in the DOM and a pointer at the element in **`index.html`** where the content should be injected.

In this example, the newly added **`<div id="dialogs">`** is selected via **`document.getElementById('dialogs')`**. Therefore, **`createPortal()`** ensures that the JSX code generated by **`ErrorDialog`** is rendered in that place in the HTML document:

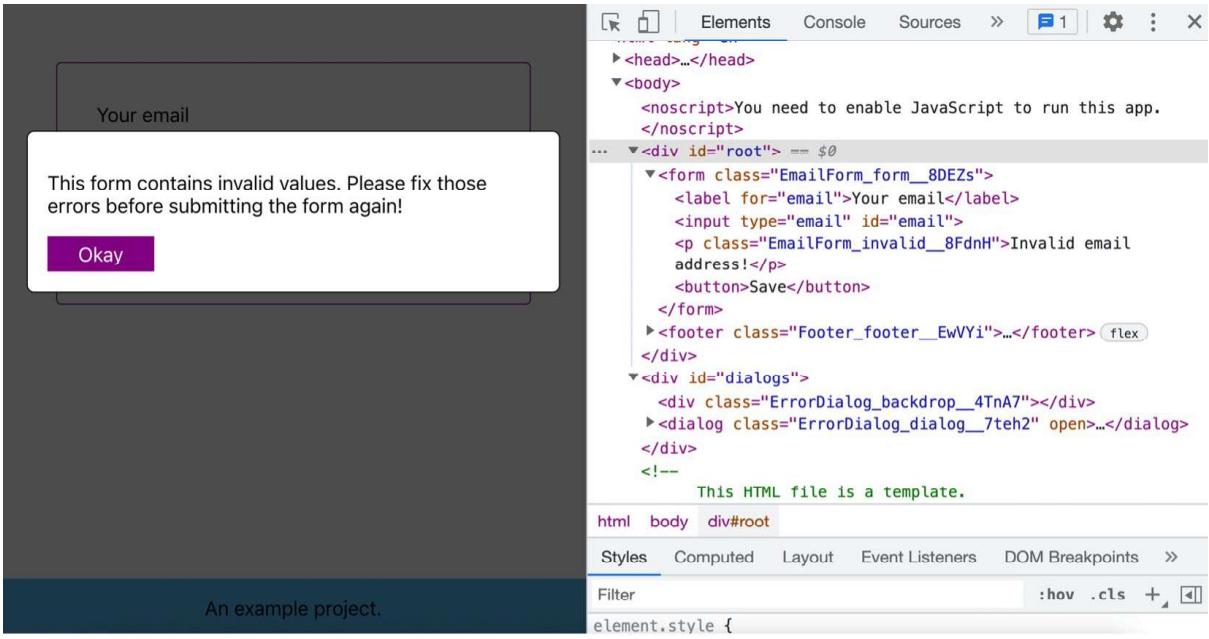

**Figure 7.7: The overlay elements are inserted into **

In this screenshot, you can see that the overlay elements (**`<div>`** backdrop and **`<dialog>`** error) are indeed inserted into the **`<div id="dialogs">`** element, instead of the **`<form>`** element (as they were before).

As a result of this change, **`<footer>`** no longer overlays the error dialog backdrop—without any CSS code changes. Semantically, the final DOM structure also makes more sense since you would typically expect overlay elements to be closer to the root DOM node.

Still, using this portal feature is optional. The same visual result (though not the DOM structure) could have been achieved by changing some CSS styles. Nonetheless, aiming for a clean DOM structure is a worthwhile pursuit, and avoiding unnecessarily complex CSS code is also not a bad thing.

SUMMARY AND KEY TAKEAWAYS

- Refs can be used to gain direct access to DOM elements or to store values that won't be reset or changed when the surrounding component is re-evaluated.
- Only use this direct access to read values, not to manipulate DOM elements (let React do this instead).
- Components that gain DOM access via refs, instead of state and other React features, are considered uncontrolled components (because React is not in direct control).

- Prefer controlled components (using state and a strictly declarative approach) over uncontrolled components unless you're performing very simple tasks such as reading an entered input value.
- Using forward refs, you can also expose features of your own components such that they may be used imperatively.
- Portals can be used to instruct React to render JSX elements in a different place in the DOM than they normally would.

WHAT'S NEXT?

At this point in the book, you've encountered many key tools and concepts that can be used to build interactive and engaging user interfaces. But, as you will learn in the next chapter, one crucial concept is still missing: a way of handling **side effects**.

The next chapter will explore what exactly **side effects** are, why they need special handling, and how React helps you with that.

TEST YOUR KNOWLEDGE!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers with examples that can be found at https://packt.link/pFCSN.

1. How can refs help with handling user input in forms?
2. What is an uncontrolled component?
3. What is a controlled component?
4. When should you **not** use refs?
5. What's the main idea behind portals?

APPLY WHAT YOU HAVE LEARNED

With this newly gained knowledge about refs and portals, it's again time to practice what you have learned.

Below, you'll find two activities that allow you to practice working with refs and portals. As always, you will, of course, also need some of the concepts covered in earlier chapters (e.g., working with state).

ACTIVITY 7.1: EXTRACT USER INPUT VALUES

In this activity, you have to add logic to an existing React component to extract values from a form. The form contains an input field and a drop-down menu and you should make sure that, upon form submission, both values are read and, for the purpose of this dummy app, output to the browser console.

Use your knowledge about Refs and uncontrolled components to implement a solution without using React state.

> **NOTE**
>
> You can find the starting code for this activity at https://packt.link/PAvKn. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (**`activities/practice-1/starting-code`** in this case) to use the right code snapshot.

After downloading the code and running **`npm install`** in the project folder (to install all the required dependencies), the solution steps are as follows:

1. Create two Refs, one for each input element that should be read (input field and drop-down menu).
2. Connect the Refs to the input elements.
3. In the submit handler function, access the connected DOM elements via the refs and read the currently entered or selected values.
4. Output the values to the browser console.

The expected result (user interface) should look like this:

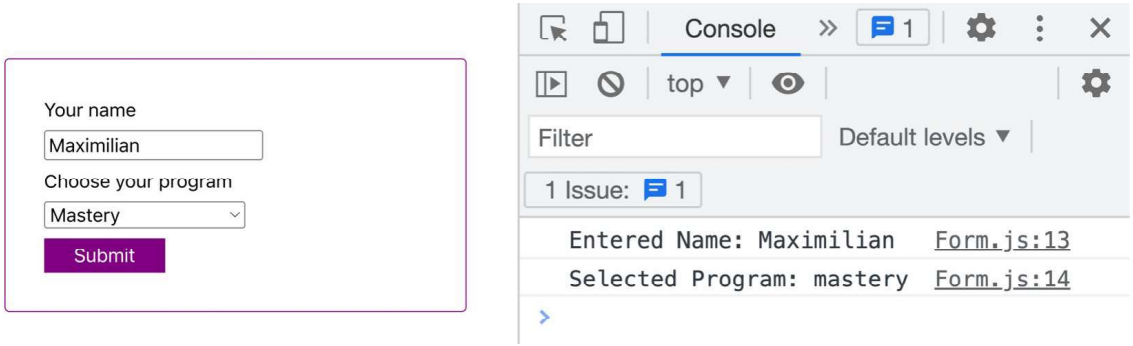

Figure 7.8: The browser developer tools console outputs the selected values

> **NOTE**
>
> The solution to this activity can be found via this link.

ACTIVITY 7.2: ADD A SIDE-DRAWER

In this activity, you will connect an already existing **`SideDrawer`** component with a button in the main navigation bar to open the side drawer (i.e., display it) whenever the button is clicked. After the side drawer opens, a click on the backdrop should close the drawer again.

In addition to implementing the general logic described above, your goal will be to ensure proper positioning in the final DOM such that no other elements are overlaid on top of the **`SideDrawer`** (without editing any CSS code). The **`SideDrawer`** should also not be nested in any other components or JSX elements.

> **NOTE**
>
> This activity comes with some starting code, which can be found here: https://packt.link/Q4RSe.

After downloading the code and running **`npm install`** to install all the required dependencies, the solution steps are as follows:

1. Add logic to conditionally show or hide the **`SideDrawer`** component in the **`MainNavigation`** component.
2. Add an "injection hook" for the side drawer in the HTML document.
3. Use React's portal feature to render the JSX elements of **`SideDrawer`** in the newly added hook.

The final user interface should look and behave like this:

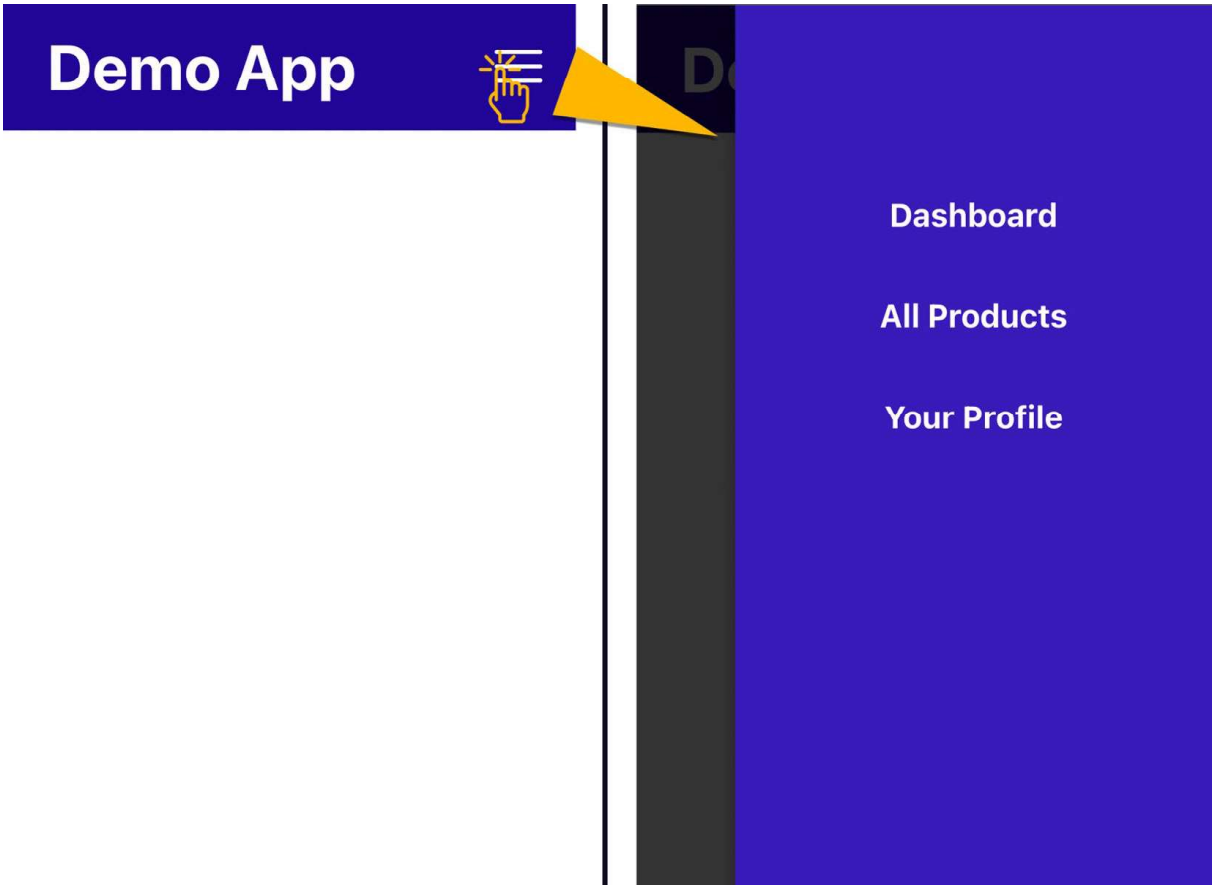

Figure 7.9: A click on the menu button opens the side drawer

Upon clicking on the menu button, the side drawer opens. If the backdrop behind the side drawer is clicked, it should close again.

The final DOM structure (with the side drawer opened) should look like this:

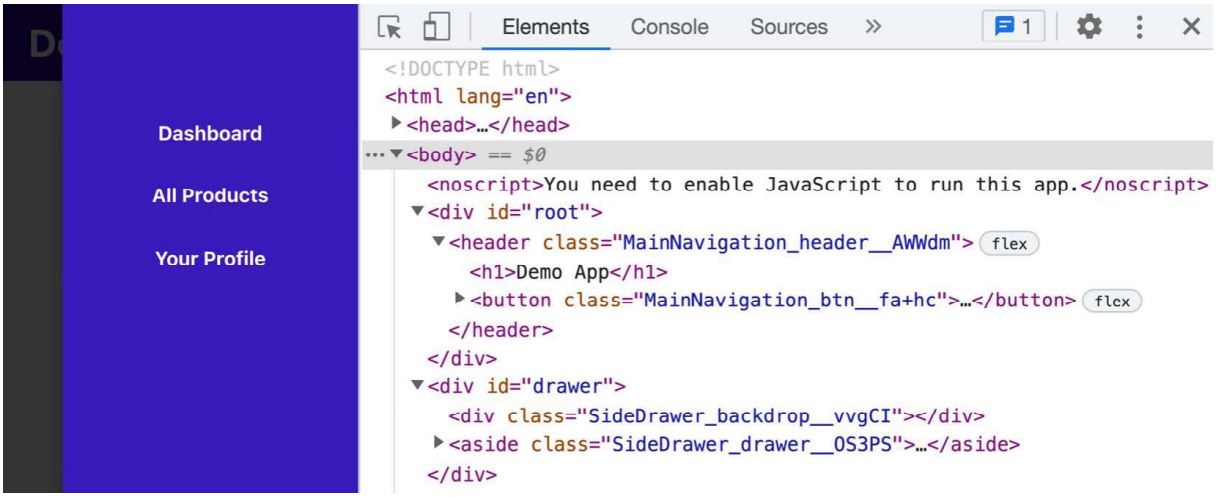

Figure 7.10: The drawer-related elements are inserted in a separate place in the DOM

The side drawer-related DOM elements (the **`<div>`** backdrop and **`<aside>`**) are inserted into a separate DOM node (**`<div id="drawer">`**).

> **NOTE**
>
> The solution to this activity can be found via this link.