8

HANDLING SIDE EFFECTS

LEARNING OBJECTIVES

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

- Identify side effects in your React apps.
- Understand and use the **`useEffect()`** Hook.
- Utilize the different features and concepts related to the **`useEffect()`** Hook to avoid bugs and optimize your code.
- Handle side effects related and unrelated to state changes.

INTRODUCTION

While all React examples previously covered in this book have been relatively straightforward, and many key React concepts were introduced, it is unlikely that many real apps could be built with those concepts alone.

Most real apps that you will build as a React developer also need to send HTTP requests, access the browser storage and log analytics data, or perform any other kind of similar task. And with components, props, events, and state alone, you'll often encounter problems when trying to add such features to your app. Detailed explanations and examples will be discussed later in this chapter, but the core problem is that tasks like this will often interfere with React's component rendering cycle, leading to unexpected bugs or even breaking the app.

This chapter will take a closer look at those kinds of actions, analyze what they have in common, and most importantly, teach you how to correctly handle such tasks in React apps.

WHAT'S THE PROBLEM?

Before exploring a solution, it's important to first understand the concrete problem.

Actions that are not directly related to producing a (new) user interface state often clash with React's component rendering cycle. They may introduce bugs or even break the entire web app.

Consider the following example code snippet:

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

import classes from './BlogPosts.module.css';

async function fetchPosts() {
  const response = await fetch('https://jsonplaceholder.typicode.com/
posts');
  const blogPosts = await response.json();
  return blogPosts;
}

function BlogPosts() {
  const [loadedPosts, setLoadedPosts] = useState([]);

  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
```

```
  return (
    <ul className={classes.posts}>
      {loadedPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default BlogPosts;
```

Don't execute this code as it will cause an infinite loop and send a large number of HTTP requests behind the scenes!

Next, you'll learn more about this problem, as well as a solution for it.

In this example, a React component (**`BlogPosts`**) is created. In addition, a non-component function (**`fetchPosts()`**) is defined. That function uses the built-in **`fetch()`** function (provided by the browser) to send an HTTP request to an external **application programming interface** (**API**) and fetch some data.

> **NOTE**
>
> The **`fetch()`** function is made available by the browser (all modern browsers support this function). You can learn more about **`fetch()`** at https://academind.com/tutorials/xhr-fetch-axios-the-fetch-api.
>
> The **`fetch()`** function yields a **promise**, which, in this example, is handled via **`async`**/**`await`**. Just like **`fetch()`**, promises are a key web development concept, which you can learn more about (along with **`async`**/**`await`**) at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.
>
> An API, in this context, is a site that exposes various paths to which requests can be sent—either to submit or to fetch data. jsonplaceholder.typicode.com is a dummy API, responding with dummy data. It can be used in scenarios like the preceding example, where you just need an API to send requests to. You can use it to test some concept or code without connecting or creating a real backend API. In this case, it's used to explore some React problems and concepts. Basic knowledge about sending HTTP requests with **`fetch()`** and APIs is expected for this chapter and the book overall. If needed, you can use pages such as MDN (https://developer.mozilla.org/) to strengthen your knowledge of such core concepts.

In the preceding code snippet, the **`BlogPosts`** component utilizes **`useState()`** to register a **`loadedPosts`** state value. The state is used to output a list of blog posts. Those blog posts are not defined in the app itself though. Instead, they are fetched from an external API (in this case, from a dummy backend API found at jsonplaceholder.typicode.com).

`fetchPosts()`, which is the utility function that contains the code for fetching blog posts data from that backend API using the built-in **`fetch()`** function, is called directly in the component function body. Since **`fetchPosts()`** is an **`async`** function (using **`async`**/**`await`**), it returns a promise. In **`BlogPosts`**, the code that should be executed once the promise resolves is registered via the built-in **`then()`** method (because React component functions shouldn't be **`async`** functions, **`async`**/**`await`** can't be used here).

Once the **`fetchPosts()`** promise resolves, the extracted posts data (**`fetchedPosts`**) is set as the new **`loadedPosts`** state (via **`setLoadedPosts(fetchedPosts)`**).

If you were to run the preceding code (which you should not do!), it would at first seem to work. But behind the scenes, it would actually start an infinite loop, hammering the API with HTTP requests. This is because, as a result of getting a response from the HTTP request, **`setLoadedPosts()`** is used to set a new state.

Earlier in this book (in *Chapter 4*, *Working with Events and State*), you learned that whenever the state of a component changes, React re-evaluates the component to which the state belongs. "Re-evaluating" simply means that the component function is executed again (by React, automatically).

Since this **`BlogPosts`** component calls **`fetchPosts()`** (which sends an HTTP request) directly inside the component function body, this HTTP request will be sent every time the component function is executed. And as the state (**`loadedPosts`**) is updated as a result of getting a response from that HTTP request, this process begins again, and an infinite loop is created.

The root problem, in this case, is that sending an HTTP request is a side effect—a concept that will be explored in greater detail in the next section.

UNDERSTANDING SIDE EFFECTS

Side effects are actions or processes that occur in addition to another "main process". At least, this is a concise definition that helps with understanding side effects in the context of a React app.

> **NOTE**
>
> You can also look up a more scientific definition here: https://en.wikipedia.org/wiki/Side_effect_(computer_science).

In the case of a React component, the main process would be the component render cycle in which the main task of a component is to render the user interface that is defined in the component function (the returned JSX code). The React component should return the final JSX code, which is then translated into DOM-manipulating instructions.

For this, React considers state changes as the trigger for updating the user interface. Registering event handlers such as **`onClick`**, adding refs, or rendering child components (possibly by using props) would be other elements that belong to this main process—because all these concepts are directly related to the main task of rendering the desired user interface.

Sending an HTTP request, as in the preceding example, is not part of this main process, though. It doesn't directly influence the user interface. While the response data might eventually be output on the screen, it definitely won't be used in the exact same component render cycle in which the request is sent (because HTTP requests are asynchronous tasks).

Since sending the HTTP request is not part of the main process (rendering the user interface) that's performed by the component function, it's considered a "side effect". It's invoked by the same function (the **`BlogPosts`** component function), which primarily has a different goal.

If the HTTP request were sent upon a click of a button rather than as part of the main component function body, it would not be a side effect. Consider this example:

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

import classes from './BlogPosts.module.css';

async function fetchPosts() {
  const response = await fetch('https://jsonplaceholder.typicode.com/
posts');
  const blogPosts = await response.json();
  return blogPosts;
}
```

```
function BlogPosts() {
  const [loadedPosts, setLoadedPosts] = useState([]);

  function fetchPostsHandler() {
    fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
  }

  return (
    <>
      <button onClick={fetchPostsHandler}>Fetch Posts</button>
      <ul className={classes.posts}>
        {loadedPosts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </>
  );
}

export default BlogPosts;
```

This code is almost identical to the previous example, but it has one important difference: a **`<button>`** was added to the JSX code. And it's this button that invokes a newly added **`fetchPostsHandler()`** function, which then sends the HTTP request (and updates the state).

With this change made, the HTTP request is *not* sent every time the component function re-renders (that is, is executed again). Instead, it's only sent whenever the button is clicked, and therefore, this does not create an infinite loop. The HTTP request, in this case, also doesn't postulate a side effect, because the primary goal of **`fetchPostsHandler()`** (i.e., the main process) is to fetch new posts and update the state.

SIDE EFFECTS ARE NOT JUST ABOUT HTTP REQUESTS

In the previous example, you learned about one potential side effect that could occur in a component function: an HTTP request. You also learned that HTTP requests are not always side effects. It depends on where they are created.

In general, any action that's started upon the execution of a React component function is a side effect if that action is not directly related to the main task of rendering the component's user interface.

Here's a non-exhaustive list of examples for side effects:

- Sending an HTTP request (as shown previously)
- Storing data to or fetching data from browser storage (for example, via the built-in **`localStorage`** object)
- Setting timers (via **`setTimeout()`**) or intervals (via **`setInterval()`**)
- Logging data to the console via **`console.log()`**
- Not all side effects cause infinite loops, however. Such loops only occur if the side effect leads to a state update.
- Here's an example of a side effect that would not cause an infinite loop:

```
function ControlCenter() {
  function startHandler() {
    // do something ...
  }

  console.log('Component is rendering!'); // this is a side effect!

  return (
    <div>
      <p>Press button to start the review process</p>
      <button onClick={startHandler}>Start</button>
    </div>
  );
}
```

In this example, **`console.log(...)`** is a side effect because it's executed as part of every component function execution and does not influence the rendered user interface (neither for this specific render cycle nor indirectly for any future render cycles in this case, unlike the previous example with the HTTP request).

Of course, using **`console.log()`** like this is not causing any problems. During development, it's quite normal to log messages or data for debugging purposes. Side effects aren't necessarily a problem and, indeed, side effects like this can be used or tolerated.

But you also often need to deal with side effects such as the HTTP request from before. Sometimes, you need to fetch data when a component renders—probably not for every render cycle, but typically the first time it is executed (that is, when its generated user interface appears on the screen for the first time).

React offers a solution for this kind of problem, as well.

DEALING WITH SIDE EFFECTS WITH THE USEEFFECT() HOOK

In order to deal with side effects such as the HTTP request shown previously in a safe way (that is, without creating an infinite loop), React offers another core Hook: the **`useEffect()`** Hook.

The first example can be fixed and rewritten like this:

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

import classes from './BlogPosts.module.css';

async function fetchPosts() {
  const response = await fetch('https://jsonplaceholder.typicode.com/
posts');
  const blogPosts = await response.json();
  return blogPosts;
}

function BlogPosts() {
  const [loadedPosts, setLoadedPosts] = useState([]);

  useEffect(function () {
    fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
  }, []);

  return (
    <ul className={classes.posts}>
      {loadedPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default BlogPosts;
```

In this example, the **`useEffect()`** Hook is imported and used to control the side effect (hence the name of the Hook, **`useEffect()`**, as it deals with side effects in React components). The exact syntax and usage will be explored in the next section, but if you use this Hook, you can safely run the example and get some output like this:

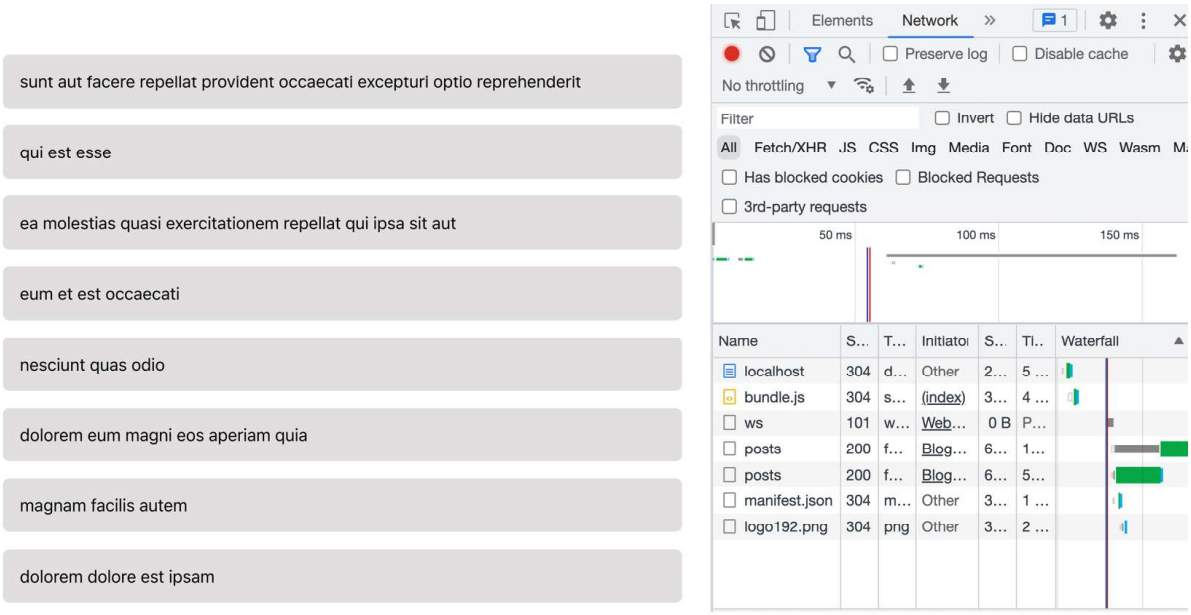

Figure 8.1: A list of dummy blog posts and no infinite loop of HTTP requests

In the preceding screenshot, you can see the list of dummy blog posts, and most importantly, when inspecting the sent network requests, you find no infinite list of requests.

`useEffect()` is therefore the solution for problems like the one outlined previously. It helps you deal with side effects so that you can avoid infinite loops and extract them from your component function's main process.

But how does **`useEffect()`** work and how is it used correctly?

HOW TO USE USEEFFECT()

As shown in the previous example code snippet, **`useEffect()`**, like all React Hooks, is executed as a function inside the component function (**`BlogPosts`**, in this case).

Though, unlike **`useState()`** or **`useRef()`**, **`useEffect()`** does not return a value, it does accept an argument (or, actually, two arguments) like those other Hooks. The first argument is *always* a function. In this case, the function passed to **`useEffect()`** is an anonymous function, created via the **`function`** keyword.

Alternatively, you could also provide an anonymous function created as an arrow function (**`useEffect(() => { ... })`**) or point at some named function (**`useEffect(doSomething)`**). The only thing that matters is that the first argument passed to **`useEffect()`** *must* be a function. It must not be any other kind of value.

In the preceding example, **`useEffect()`** also receives a second argument: an empty array (**`[]`**). The second argument must be an array, but providing it is *optional*. You could also omit the second argument and just pass the first argument (the function) to **`useEffect()`**. However, in most cases, the second argument is needed to achieve the correct behavior. Both arguments and their purpose will be explored in greater detail as follows.

The first argument is a function that will be executed by React. It will be executed *after* every component render cycle (that is, after every component function execution).

In the preceding example, if you only provide this first argument and omit the second, you will therefore still create an infinite loop. There will be an (invisible) timing difference because the HTTP request will now be sent after every component function execution (instead of as part of it), but you will still trigger a state change, which will still trigger the component function to execute again. Therefore, the effect function will run again, and an infinite loop will be created. In this case, the side effect will be extracted out of the component function technically, but the problem with the infinite loop will not be solved:

```
useEffect(function () {
  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}); // this would cause an infinite loop again!
```

Extracting side effects out of React component functions is the main job of **`useEffect()`**, and so only the first argument (the function that contains the side effect code) is mandatory. But, as mentioned previously, you will also typically need the second argument to control the frequency with which the effect code will be executed, because that's what the second argument (an array) will do.

The second parameter received by **`useEffect()`** is *always* an array (unless it's omitted). This array specifies the dependencies of the effect function. Any dependency specified in this array will, once it changes, cause the effect function to execute again. If no array is specified (that is, if the second argument is omitted), the effect function will be executed again for every component function execution:

```
useEffect(function () {
  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, []);
```

In the preceding example, the second argument was not omitted, but it's an empty array. This informs React that this effect function has no dependencies. Therefore, the effect function will never be executed again. Instead, it will only be executed once, when the component is rendered for the first time. This is React's default behavior. If you set no dependencies (by providing an empty array), React will execute the effect function *once*—directly after the component function was executed for the first time.

It's important to note that specifying an empty array is very different from omitting it. If omitted, no dependency information is provided to React. Therefore, React executes the effect function after every component re-evaluation. If an empty array is provided instead, you explicitly state that this effect has no dependencies and therefore should only run once.

This brings up another important question, though: when should you add dependencies? And how exactly are dependencies added or specified?

EFFECTS AND THEIR DEPENDENCIES

Omitting the second argument to **`useEffect()`** causes the effect function (the first argument) to execute after every component function execution. Providing an empty array causes the effect function to run only once (after the first component function invocation). But is that all you can control?

No, it isn't.

The array passed to **`useEffect()`** can and should contain all variables, constants, or functions that are used inside the effect function—if those variables, constants, or functions were defined inside the component function (or in some parent component function, passed down via props).

Consider this example:

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

import classes from './BlogPosts.module.css';

const DEFAULT_URL = 'https://jsonplaceholder.typicode.com/posts';

async function fetchPosts(url) {
  const response = await fetch(url);
```

```
  const blogPosts = await response.json();
  return blogPosts;
}

function BlogPosts() {
  const [postsUrl, setPostsUrl] = useState(DEFAULT_URL);
  const [loadedPosts, setLoadedPosts] = useState([]);

  function adjustUrlHandler(event) {
    setPostsUrl(event.target.value);
  }

  useEffect(function () {
    fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
  }, [postsUrl]);

  return (
    <>
      <input className={classes.input} type="text"
onBlur={adjustUrlHandler} />
      <ul className={classes.posts}>
        {loadedPosts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </>
  );
}

export default BlogPosts;
```

This example is based on the previous example, but it was adjusted in multiple places.

The **`BlogPosts`** component now contains a second state value (**`postsUrl`**), which is updated inside **`adjustUrlHandler()`** whenever a newly added **`<input>`** field blurs (i.e., loses focus). This allows the website visitor to enter a custom URL to which the HTTP request will be sent.

By default (**`DEFAULT_URL`**), a dummy API (**`https://jsonplaceholder.typicode.com/posts`**) will be used, but the user could insert any URL of their choice. Of course, if that API doesn't return a list of blog posts, the app won't work as intended. This component therefore might be of limited practical use, but it does show the importance of effect dependencies quite well.

A new effect (and therefore a new HTTP request) should only be triggered if **`postsUrl`** changes. That's why **`postsUrl`** was added to the dependencies array of **`useEffect()`**. If the array had been kept empty, the effect function would only run once (as described in the previous section). Therefore, any changes to **`postsUrl`** wouldn't have any effect (no pun intended) on the effect function or the HTTP request executed as part of that function. No new HTTP request would be sent.

By adding **`postsUrl`** to the dependencies array, React registers this value (in this case, a state value, but any value can be registered) and re-executes the effect function whenever that value changes (that is, whenever a new value is set).

The most common types of effect dependencies are state values, props, and functions that might be executed inside of the effect function. The latter will be analyzed in greater depth later in this chapter.

As a rule, you should add all values (including functions) that are used inside an effect function to the effect dependencies array.

With this new knowledge in mind, if you take another look at the preceding **`useEffect()`** example code, you might spot some missing dependencies:

```
useEffect(function () {
  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, [postsUrl]);
```

Why are **`fetchPosts`**, **`fetchedPosts`**, and **`setLoadedPosts`** not added as dependencies? These are, after all, values and functions used inside of the effect function. The next section will address this in detail.

UNNECESSARY DEPENDENCIES

In the previous example, it might seem as if **`fetchPosts`**, **`fetchedPosts`**, and **`setLoadedPosts`** should be added as dependencies to **`useEffect()`**, as shown here:

```
useEffect(function () {
  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, [postsUrl, fetchPosts, fetchedPosts, setLoadedPosts]);
```

However, for **`fetchPosts`** and **`fetchedPosts`**, this would be incorrect. And for **`setLoadedPosts`**, it would be unnecessary.

`fetchedPosts` should not be added because it's not an external dependency. It's a local variable (or argument, to be precise), defined and used inside the effect function. It's not defined in the component function to which the effect belongs. If you try to add it as a dependency, you'll get an error:

Figure 8.2: An error occurred – fetchedPosts could not be found

`fetchPosts`, the function that sends the actual HTTP request, is not a function defined inside of the effect function. But it still shouldn't be added because it is defined outside the component function.

Therefore, there is no way for this function to change. It's defined once (in the **`BlogPosts.js`** file), and it can't change. That said, this would not be the case if it were defined inside the component function. In that case, whenever the component function executes again, the **`fetchPosts`** function would be recreated as well. This is a scenario that will be discussed later in this chapter (in the *"Functions as Dependencies"* section).

In this example though, **`fetchPosts`** can't change. Therefore, it doesn't have to be added as a dependency (and consequently should not be). The same would be true for functions, or any kind of values, provided by the browser or third-party packages. Any value that's not defined inside a component function shouldn't be added to the dependencies array.

> **NOTE**
>
> It may be confusing that a function could change—after all, the logic is hardcoded, right? But in JavaScript, functions are actually just objects and therefore may change. When the code that contains a function is executed again (e.g., a component function being executed again by React), a new function object will be created in memory.
>
> If this is not something you're familiar with, the following resource should be helpful: https://academind.com/tutorials/javascript-functions-are-objects.

So **`fetchedPosts`** and **`fetchPosts`** should both not be added (for different reasons). What about **`setLoadedPosts`**?

`setLoadedPosts` is the state updating function returned by **`useState()`** for the **`loadedPosts`** state value. Therefore, like **`fetchPosts`**, it's a function. Unlike **`fetchPosts`**, though, it's a function that's defined inside the component function (because **`useState()`** is called inside the component function). It's a function created by React (since it's returned by **`useState()`**), but it's still a function. Theoretically, it should therefore be added as a dependency. And indeed, you can add it without any negative consequences.

But state updating functions returned by **`useState()`** are a special case: React guarantees that those functions will never change or be recreated. When the surrounding component function (**`BlogPosts`**) is executed again, **`useState()`** also executes again. However, a new state (and a new state updating function) is only created the first time a component function is called by React. Subsequent executions don't lead to a new state value or state updating function being created.

Because of this special behavior (i.e., React guaranteeing that the function itself never changes), state updating functions may (and actually should) be omitted from the dependencies array.

For all these reasons, **`fetchedPosts`**, **`fetchPosts`**, and **`setLoadedPosts`** should all not be added to the dependencies array of **`useEffect()`**. **`postsUrl`** is the only dependency used by the effect function that may change (that is, when the user enters a new URL into the input field) and therefore should be listed in the array.

To sum it up, when it comes to adding values to the effect dependencies array, there are three kinds of exceptions:

- Internal values (or functions) that are defined and used inside the effect (such as **`fetchedPosts`**)
- External values that are not defined inside a component function (such as **`fetchPosts`**)
- State updating functions (such as **`setLoadedPosts`**)

In all other cases, if a value is used in the effect function, it *must be added* to the dependencies array! Omitting values incorrectly can lead to unexpected effect executions (that is, an effect executing too often or not often enough).

CLEANING UP AFTER EFFECTS

To perform a certain task (for example, sending an HTTP request), many effects should simply be triggered when their dependencies change. While some effects can be re-executed multiple times without issue, there are also effects that, if they execute again before the previous task has finished, are an indication that the task performed needs to be canceled. Or, maybe there is some other kind of cleanup work that should be performed when the same effect executes again.

Here's an example, where an effect sets a timer:

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

function Alert() {
  const [alertDone, setAlertDone] = useState(false);

  useEffect(function () {
    console.log('Starting Alert Timer!');
    setTimeout(function () {
      console.log('Timer expired!');
      setAlertDone(true);
    }, 2000);
  }, []);

  return (
    <>
      {!alertDone && <p>Relax, you still got some time!</p>}
      {alertDone && <p>Time to get up!</p>}
    </>
  );
}

export default Alert;
```

This **`Alert`** component is used in the **`App`** component:

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

import Alert from './components/Alert';

function App() {
  const [showAlert, setShowAlert] = useState(false);
```

```
  function showAlertHandler() {
    // state updating is done by passing a function to setShowAlert
    // because the new state depends on the previous state (it's the
opposite)
    setShowAlert((isShowing) => !isShowing);
  }

  return (
    <>
      <button onClick={showAlertHandler}>
        {showAlert ? 'Hide' : 'Show'} Alert
      </button>
      {showAlert && <Alert />}
    </>
  );
}

export default App;
```

> **NOTE**
>
> You can also clone or download the full example from GitHub at https://packt.link/Zmkp9.

In the **`App`** component, the **`Alert`** component is shown conditionally. The **`showAlert`** state is toggled via the **`showAlertHandler`** function (which is triggered upon a button click).

In the **`Alert`** component, a timer is set using **`useEffect()`**. Without **`useEffect()`**, an infinite loop would be created, since the timer, upon expiration, changes some component state (the **`alertDone`** state via the **`setAlertDone`** state updating function).

The dependency array is an empty array because this effect function does not use any component values, variables, or functions. **`console.log()`** and **`setTimeout()`** are functions built into the browser (and therefore external functions), and **`setAlertDone()`** can be omitted because of the reasons mentioned in the previous section.

If you run this app and then start toggling the alert (by clicking the button), you'll notice strange behavior. The timer is set every time the **`Alert`** component is rendered. But it's not clearing the existing timer. This is due to the fact that multiple timers are running simultaneously, as you can clearly see if you look at the JavaScript console in your browser's developer tools:

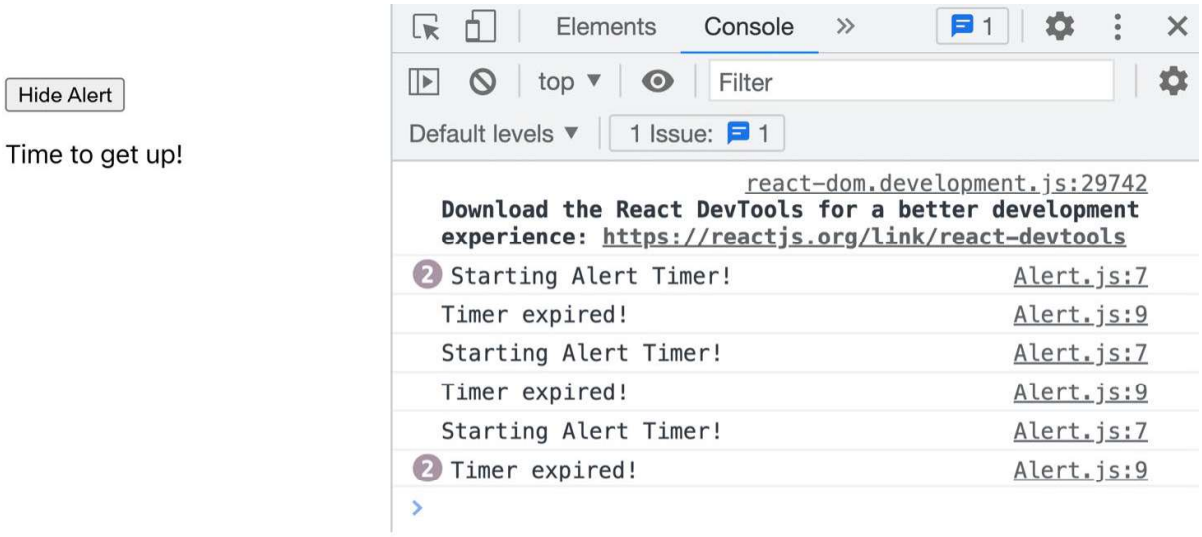

Figure 8.3: Multiple timers are started

This example is deliberately kept simple, but there are other scenarios in which you may have an ongoing HTTP request that should be aborted before a new one is sent. There are cases like that, where an effect should be cleaned up first before it runs again.

React also offers a solution for those kinds of situations: the effect function passed as a first argument to **`useEffect()`** can return an optional cleanup function. If you do return a function inside your effect function, React will execute that function every time *before* it runs the effect again.

Here's the **`useEffect()`** call of the **`Alert`** component with a cleanup function being returned:

```
useEffect(function () {
  let timer;

  console.log('Starting Alert Timer!');
  timer = setTimeout(function () {
    console.log('Timer expired!');
    setAlertDone(true);
  }, 2000);
```

```
  return function() {
    clearTimeout(timer);
  }
}, []);
```

In this updated example, a new **`timer`** variable (a local variable that is only accessible inside the effect function) is added. That variable stores a reference to the timer that's created by **`setTimeout()`**. This reference can then be used together with **`clearTimeout()`** to remove a timer.

The timer is removed in a function returned by the effect function—which is the cleanup function that will be executed automatically by React before the effect function is called the next time.

You can see the cleanup function in action if you add a **`console.log()`** statement to it:

```
return function() {
  console.log('Cleanup!');
  clearTimeout(timer);
}
```

In your JavaScript console, this looks as follows:

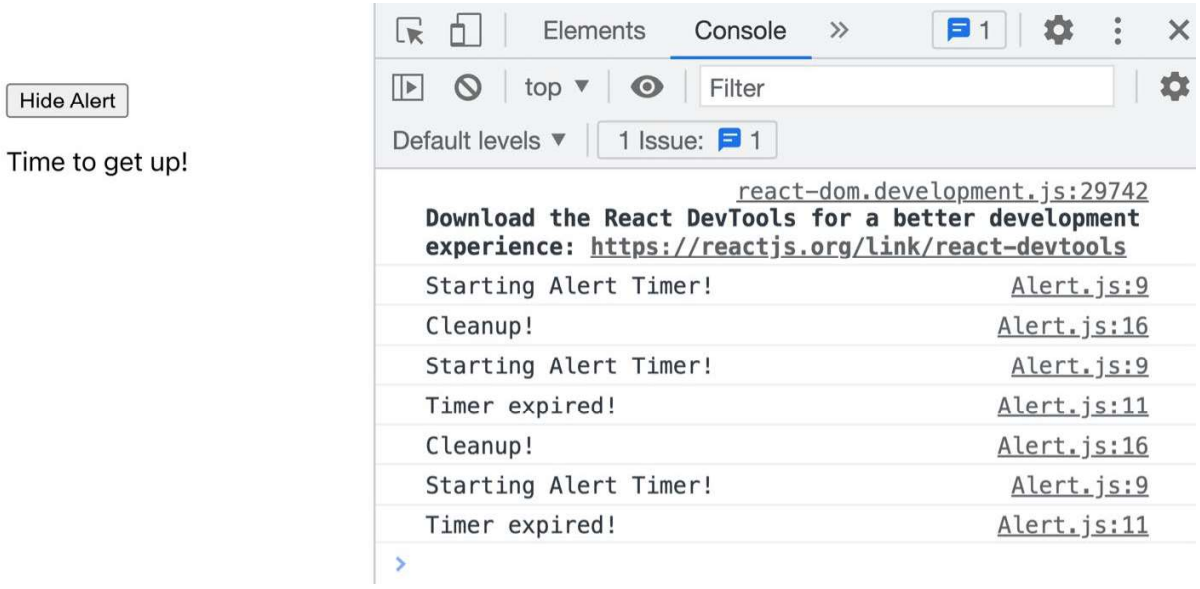

Figure 8.4: The cleanup function is executed before the effect runs again

In the preceding screenshot, you can see that the cleanup function is executed (indicated by the **`Cleanup!`** log) right before the effect function is executed again. You can also see that the timer is cleared successfully: the first timer never expires (there is no **`Timer expired!`** log for the first timer in the screenshot).

The cleanup function is not executed when the effect function is called for the first time. However, it will be called by React whenever a component that contains an effect unmounts (that is, when it's removed from the DOM).

If an effect has multiple dependencies, the effect function will be executed whenever any of the dependency values change. Therefore, the cleanup function will also be called every time some dependency changes.

DEALING WITH MULTIPLE EFFECTS

Thus far, all the examples in this chapter have dealt with only one **`useEffect()`** call. You are not limited to only one call per component though. You can call **`useEffect()`** as often as needed—and can therefore register as many effect functions as needed.

But how many effect functions do you need?

You could start putting every side effect into its own **`useEffect()`** wrapper. Every HTTP request, every **`console.log()`** statement, and every timer, you could put into separate effect functions.

That said, as you can see in some of the previous examples—specifically, the code snippet in the previous section—that's not necessary. There, you have multiple effects in one **`useEffect()`** call (three **`console.log()`** statements and one timer).

A better approach would be to split your effect functions by dependencies. If one side effect depends on state A and another side effect depends on state B, you could put them into separate effect functions (unless those two states are related), as shown here:

```
function Demo() {
  const [a, setA] = useState(0); // state updating functions aren't
called
  const [b, setB] = useState(0); // in this example


  useEffect(function() {
    console.log(a);
  }, [a]);
```

```
  useEffect(function() {
    console.log(b);
  }, [b]);

  // return some JSX code ...
}
```

But the best approach is to split your effect functions by logic. If one effect deals with fetching data via an HTTP request and another effect is about setting a timer, it will often make sense to put them into different effect functions (that is, different **`useEffect()`** calls).

FUNCTIONS AS DEPENDENCIES

Different effects have different kinds of dependencies, and one common kind of dependency is functions.

As mentioned previously, functions in JavaScript are just objects. Therefore, whenever some code that contains a function definition is executed, a new function object is created and stored in memory. When calling a function, it's that specific function object in memory that is executed. In some scenarios (for example, for functions defined in component functions), it's possible that multiple objects based on the same function code exist in memory.

Because of this behavior, functions that are referenced in code are not necessarily equal, even if they are based on the same function definition.

Consider this example:

```
function Alert() {
  function setAlert() {
    setTimeout(function() {
      console.log('Alert expired!');
    }, 2000);
  }

  useEffect(function() {
    setAlert();
  }, [setAlert]);

  // return some JSX code ...
}
```

In this example, instead of creating a timer directly inside the effect function, a separate **`setAlert()`** function is created in the component function. That **`setAlert()`** function is then used in the effect function passed to **`useEffect()`**. Since that function is used there, and because it's defined in the component function, it should be added as a dependency to **`useEffect()`**.

Another reason for this is that every time the **`Alert`** component function is executed again (e.g., because some state or prop value changes), a new **`setAlert`** function object will be created. In this example, that wouldn't be problematic because **`setAlert`** only contains static code. A new function object created for **`setAlert`** would work exactly in the same way as the previous one, therefore, it would not matter.

But now consider this adjusted example

> **NOTE**
>
> The complete app can be found on GitHub at https://packt.link/pna08.

```
function Alert() {
  const [alertMsg, setAlertMsg] = useState('Expired!');

  function changeAlertMsgHandler(event) {
    setAlertMsg(event.target.value);
  }

  function setAlert() {
    setTimeout(function () {
      console.log(alertMsg);
    }, 2000);
  }

  useEffect(
    function () {
      setAlert();
    },
```

```
    []
  );

  return <input type="text" onChange={changeAlertMsgHandler} />;
}

export default Alert;
```

Now, a new **`alertMsg`** state is used for setting the actual alert message that's logged to the console. In addition, the **`setAlert`** dependency was removed from **`useEffect()`**.

If you run this code, you'll get the following output:

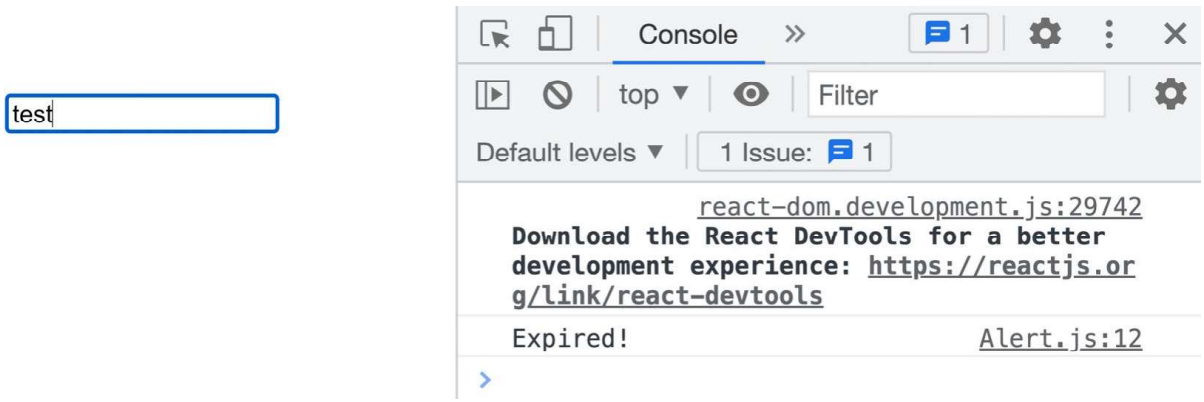

Figure 8.5: The console log does not reflect the entered value

In this screenshot, you can see that, despite a different value being entered into the input field, the original alert message is output.

The reason for this behavior is that the new alert message is not picked up. It's not used because, despite the component function being executed again (because the state changed), the effect is not executed again. And the original execution of the effect still uses the old version of the **`setAlert`** function—the old **`setAlert`** function object, which has the old alert message locked in. That's how JavaScript functions work, and that's why, in this case, the desired result is not achieved.

The solution to the problem is simple though: add **`setAlert`** as a dependency to **`useEffect()`**. You should always add all values, variables, or functions used in an effect as dependencies, and this example shows *why* you should do that. Even functions can change.

If you add **`setAlert`** to the effect dependency array, you'll get a different output:

```
useEffect(
  function () {
    setAlert();
  },
  [setAlert]
);
```

Please note that only a pointer to the **`setAlert`** function is added. You don't execute the function in the dependencies array (that would add the return value of the function as a dependency, which is typically not the goal).

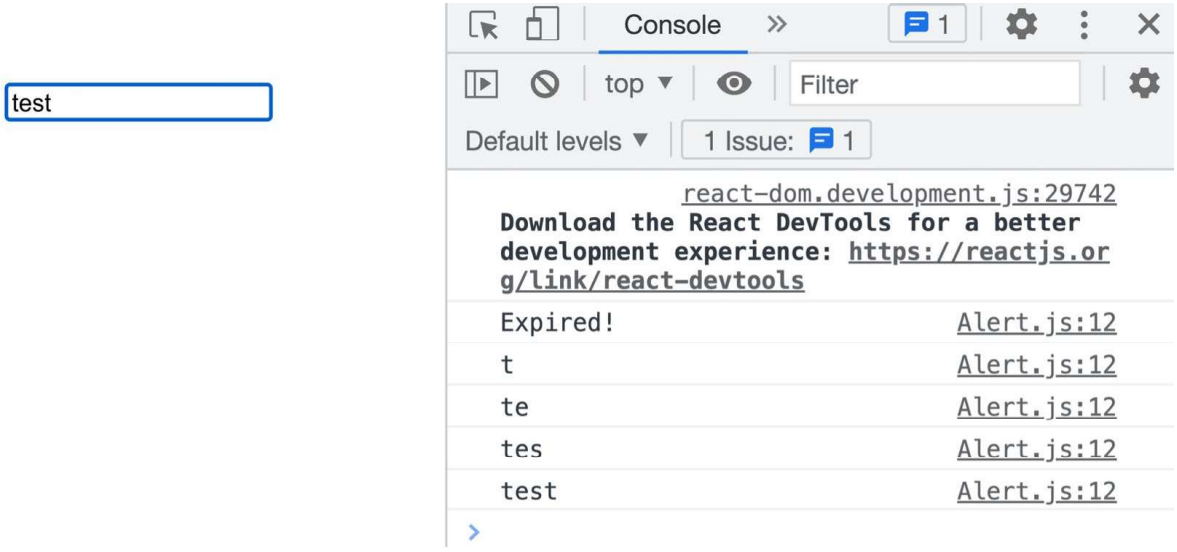

Figure 8.6: Multiple timers are started

Now, a new timer is started for every keystroke, and as a result, the entered message is output in the console.

Of course, this might also not be the desired result. You might only be interested in the final error message that was entered. This can be achieved by adding a cleanup function to the effect (and adjusting **`setAlert`** a little bit):

```
function setAlert() {
  return setTimeout(function () {
    console.log(alertMsg);
  }, 2000);
}

useEffect(
```

```
  function () {
    const timer = setAlert();

    return function () {
      clearTimeout(timer);
    };
  },
  [setAlert]
);
```

As shown in the "*Cleaning Up after Effects*" section, the timer is cleared with the help of a timer reference and **`clearTimeout()`** in the effect's cleanup function.

After adjusting the code like this, only the final alert message that was entered will be output.

Seeing the cleanup function in action again is helpful; the main takeaway is the importance of adding all dependencies, though—including function dependencies.

An alternative to including the function as a dependency would be to move the entire function definition into the effect function because any value that's defined and used inside of an effect function must not be added as a dependency:

```
useEffect(
  function () {
    function setAlert() {
      return setTimeout(function () {
        console.log(alertMsg);
      }, 2000);
    }

    const timer = setAlert();

    return function () {
      clearTimeout(timer);
    };
  },
  []
);
```

Of course, you could also get rid of the **`setAlert`** function altogether then and just move the function's code into the effect function.

Either way, you will have to add a new dependency: **`alertMsg`**, which is now used inside of the effect function. Even though the **`setAlert`** function isn't a dependency anymore, you still must add any values used (and **`alertMsg`** is used in the effect function now):

```
useEffect(
  function () {
    function setAlert() {
      return setTimeout(function () {
        console.log(alertMsg);
      }, 2000);
    }

    const timer = setAlert();

    return function () {
      clearTimeout(timer);
    };
  },
  [alertMsg]
);
```

Hence, this alternative way of writing the code just comes down to personal preferences. It does not reduce the number of dependencies.

You would get rid of a function dependency if you were to move the function out of the component function. This is because, as mentioned in the *"Unnecessary Dependencies"* section, external dependencies (for example, built into the browser or defined outside of component functions) should not be added as dependencies.

However, in the case of the **`setAlert`** function, this is not possible because **`setAlert`** uses **`alertMsg`**. Since **`alertMsg`** is a component state value, the function that uses it must be defined inside the component function; otherwise, it won't have access to that state value.

This can all sound quite confusing, but it comes down to two simple rules:

- Always add all non-external dependencies—no matter whether they're variables or functions.
- Functions are just objects and can change if their surrounding code executes again.

AVOIDING UNNECESSARY EFFECT EXECUTIONS

Since all dependencies should be added to **`useEffect()`**, you sometimes end up with code that causes an effect to execute unnecessarily.

Consider the example component below.

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

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

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

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

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

  function validateEmail() {
    if (!enteredEmail.includes('@')) {
      console.log('Invalid email!');
    }
  }

  useEffect(function () {
    validateEmail();
  }, [validateEmail]);

  return (
    <form>
      <div>
        <label>Email</label>
        <input type="email" onChange={updateEmailHandler} />
```

```
      </div>
      <div>
        <label>Password</label>
        <input type="password" onChange={updatePasswordHandler} />
      </div>
      <button>Save</button>
    </form>
  );
}

export default Alert;
```

This component contains a form with two inputs. The entered values are stored in two different state values (**`enteredEmail`** and **`enteredPassword`**). The **`validateEmail()`** function then performs some email validation and, if the email address is invalid, logs a message to the console. **`validateEmail()`** is executed with the help of **`useEffect()`**.

The problem with this code is that the effect function will be executed whenever **`validateEmail`** changes because, correctly, **`validateEmail`** was added as a dependency. But **`validateEmail`** will change whenever the component function is executed again. And that's not just the case for state changes to **`enteredEmail`** but also whenever **`enteredPassword`** changes—even though that state value is not used at all inside of **`validateEmail`**.

This unnecessary effect execution can be avoided with various solutions:

- You could move the code inside of **`validateEmail`** directly into the effect function (**`enteredEmail`** would then be the only dependency of the effect, avoiding effect executions when any other state changes).
- You could avoid using **`useEffect()`** altogether since you could perform email validation inside of **`updateEmailHandler`**. Having **`console.log()`** (a side effect) in there would be acceptable since it wouldn't cause any harm.

But in some other scenarios, you might need to use **`useEffect()`** (for example, to avoid an infinite loop). Fortunately, React also offers a solution for situations like this: you can wrap the function that's used as a dependency with another React Hook, the **`useCallback()`** Hook.

The adjusted code would look like this:

```
import { useState, useEffect, useCallback } from 'react';

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

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

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

  const validateEmail = useCallback(
    function () {
      if (!enteredEmail.includes('@')) {
        console.log('Invalid email!');
      }
    },
    [enteredEmail]
  );

  useEffect(
    function () {
      validateEmail();
    },
    [validateEmail]
  );

  // return JSX code ...
}

export default Alert;
```

`useCallback()`, like all React Hooks, is a function that's executed directly inside the component function. Like **`useEffect()`**, it accepts two arguments: another function (can be anonymous or a named function) and a dependencies array.

Unlike **`useEffect()`**, though, **`useCallback()`** does not execute the received function. Instead, **`useCallback()`** ensures that a function is only recreated if one of the specified dependencies has changed. The default JavaScript behavior of creating a new function object whenever the surrounding code executes again is (synthetically) disabled.

`useCallback()` returns the latest saved function object. Hence, that returned value (which is a function) is saved in a variable or constant (**`validateEmail`** in the previous example).

Since the function wrapped by **`useCallback()`** now only changes when one of the dependencies changes, the returned function can be used as a dependency for **`useEffect()`** without executing that effect for all kinds of state changes or component updates.

In the case of the preceding example, the effect function would then only execute when **`enteredEmail`** changes—because that's the only change that will lead to a new **`validateEmail`** function object being created.

Another common reason for unnecessary effect execution is the usage of objects as dependencies, like in this example:

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

function Error(props) {
  useEffect(
    function () {
      // performing some error logging
      // in a real app, a HTTP request might be sent to some analytics
API
      console.log('An error occurred!');
      console.log(props.message);
    },
    [props]
  );

  return <p>{props.message}</p>;
}

export default Error;
```

This **`Error`** component is used in another component, the **`Form`** component, like this:

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

import Error from './Error';

function Form() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [errorMessage, setErrorMessage] = useState('');

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

  function submitFormHandler(event) {
    event.preventDefault();
    if (!enteredEmail.endsWith('.com')) {
      setErrorMessage('Only email addresses ending with .com are
accepted!');
    }
  }

  return (
    <form onSubmit={submitFormHandler}>
      <div>
        <label>Email</label>
        <input type="email" onChange={updateEmailHandler} />
      </div>
      {errorMessage && <Error message={errorMessage} />}
      <button>Submit</button>
    </form>
  );
}

export default Form;
```

The **`Error`** component receives an error message via props (**`props.message`**) and displays it on the screen. In addition, with the help of **`useEffect()`**, it does some error logging. In this example, the error is simply output to the JavaScript console. In a real app, the error might be sent to some analytics API via an HTTP request. Either way, a side effect that depends on the error message is performed.

The **`Form`** component contains two state values, tracking the entered email address as well as the error status of the input. If an invalid input value is submitted, **`errorMessage`** is set and the **`Error`** component is displayed.

The interesting part about this example is the dependency array of **`useEffect()`** inside the **`Error`** component. It contains the **`props`** object as a dependency (**`props`** is always an object, grouping all prop values together). When using objects (props or any other object, it does not matter) as dependencies for **`useEffect()`**, unnecessary effect function executions can be the result.

You can see this problem in this example. If you run the app and enter an invalid email address (e.g., **`test@test.de`**), you'll notice that subsequent keystrokes in the email input field will cause the error message to be logged (via the effect function) for every keystroke.

> **NOTE**
>
> The full code can be found on GitHub at https://packt.link/qqaDG.

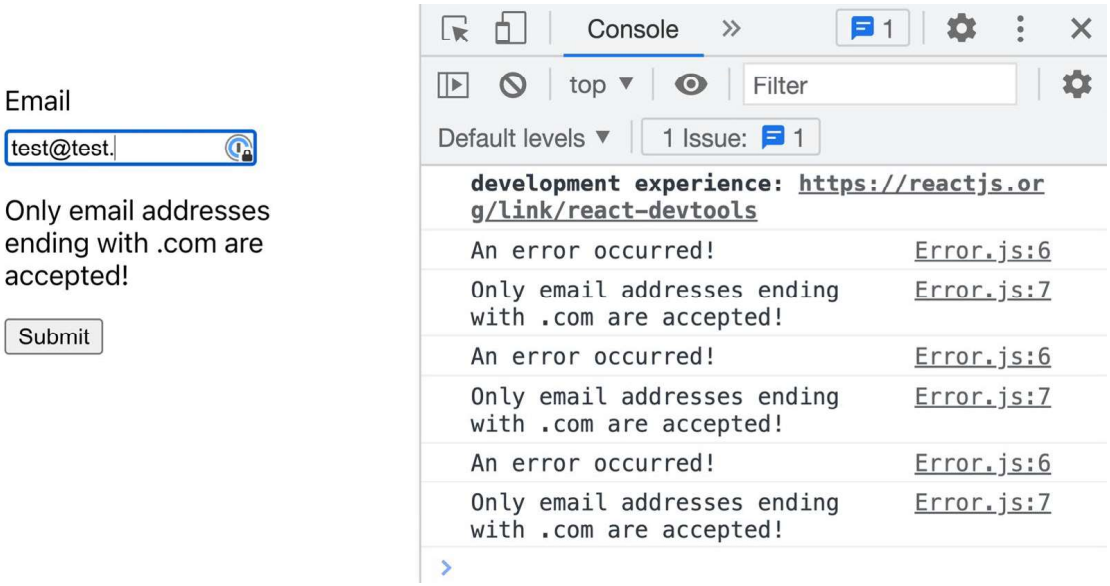

Figure 8.7: A new error message is logged for every keystroke

Those extra executions can occur because component re-evaluations (i.e., component functions being invoked again by React) will produce brand-new JavaScript objects. Even if the values of properties of those objects did not change (as in the preceding example), technically, a brand-new object in memory is created by JavaScript. Since the effect depends on the entire object, React only "sees" that there is a new version of that object and hence runs the effect again.

In the preceding example, a new props object (for the **`Error`** component) is created whenever the **`Form`** component function is called by React—even if the error message (the only prop value that's set) did not change.

In this example, that's just annoying since it clutters the JavaScript console in the developer tools. However, if you were sending an HTTP request to some analytics backend API, it could cause bandwidth problems and make the app slower. Therefore, it's best if you get into the habit of avoiding unnecessary effect executions as a general rule.

In the case of object dependencies, the best way to avoid unnecessary executions is to simply destructure the object so that you can pass only those object properties as dependencies that are needed by the effect:

```
function Error(props) {
  const { message } = props; // destructure to extract required
properties

  useEffect(
    function () {
      console.log('An error occurred!');
      console.log(props.message);
    },
    // [props] // don't use the entire props object!
    [message]
  );

  return <p>{props.message}</p>;
}
```

In the case of props, you could also destructure the object right in the component function parameter list:

```
function Error({message}) {
  // ...
}
```

Using this approach, you ensure that only the required property values are set as dependencies. Therefore, even if the object gets recreated, the property value (in this case, the value of the **`message`** property) is the only thing that matters. If it doesn't change, the effect function won't be executed again.

EFFECTS AND ASYNCHRONOUS CODE

Some effects deal with asynchronous code (sending HTTP requests is a typical example). When performing asynchronous tasks in effect functions, there is one important rule to keep in mind, though: the effect function itself should not be asynchronous and should not return a promise.

You might want to use **`async`**/**`await`** to simplify asynchronous code, but when doing so inside of an effect function, it's easy to accidentally return a promise. For example, the following code would work but does not follow best practices:

```
useEffect(async function () {
  const fetchedPosts = await fetchPosts();
  setLoadedPosts(fetchedPosts);
}, []);
```

Adding the **`async`** keyword in front of **`function`** unlocks the usage of **`await`** inside the function—which makes dealing with asynchronous code (that is, with promises) more convenient.

But the effect function passed to **`useEffect()`** should only return a normal function, if anything. It should not return a promise. Indeed, React actually issues a warning when trying to run code like the preceding:

Figure 8.8: React shows a warning about async being used in an effect function

To avoid this warning, you can use promises without **`async`**/**`await`**, like this:

```
useEffect(function () {
  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, []);
```

Alternatively, if you want to use **`async`**/**`await`**, you can create a separate wrapper function inside of the effect function, which is then executed in the effect:

```
useEffect(function () {
  async function loadData() {
    const fetchedPosts = await fetchPosts();
    setLoadedPosts(fetchedPosts);
  }

  loadData();
}, []);
```

By doing that, the effect function itself is not asynchronous (it does not return a promise), but you can still use **`async`**/**`await`**.

RULES OF HOOKS

In this chapter, two new Hooks were introduced: **`useEffect()`** and **`useCallback()`**. Both Hooks are very important—**`useEffect()`** especially, as this is a Hook you will typically use a lot. Together with **`useState()`** (introduced in *Chapter 4*, *Working with Events and State*) and **`useRef()`** (introduced in *Chapter 7*, *Portals and Refs*), you now have a solid set of key React Hooks.

When working with React Hooks, there are two rules (the so-called **rules of Hooks**) you must follow:

- Only call Hooks at the top level of component functions. Don't call them inside of if statements, loops, or nested functions.
- Only call Hooks inside of React components or custom Hooks (custom Hooks will be covered in *Chapter 11*, *Building Custom React Hooks*).

These rules exist because React Hooks won't work as intended if used in a non-compliant way. Fortunately, React will generate a warning message if you violate one of these rules, hence you will notice if you accidentally do so.

SUMMARY AND KEY TAKEAWAYS

- Actions that are not directly related to the main process of a function can be considered side effects.
- Side effects can be asynchronous tasks (for example, sending an HTTP request), but can also be synchronous (for example, `console.log()` or accessing browser storage).
- Side effects are often needed to achieve a certain goal, but it's a good idea to separate them from the main process of a function.
- Side effects can become problematic if they cause infinite loops (because of the update cycles between effect and state).
- `useEffect()` is a React Hook that should be used to wrap side effects and perform them in a safe way.
- `useEffect()` takes an effect function and an array of effect dependencies.
- The effect function is executed directly after the component function was invoked (not simultaneously).
- Any value, variable, or function used inside of an effect should be added to the dependencies array.
- Dependency array exceptions are external values (defined outside of a component function), state updating functions, or values defined and used inside of the effect function.
- If no dependency array is specified, the effect function executes after every component function invocation.
- If an empty dependency array is specified, the effect function runs once when the component first mounts (that is, when it is created for the first time).
- Effect functions can also return optional cleanup functions that are called right before an effect function is executed again (and right before a component is removed from the DOM).
- Effect functions must not return promises.
- For function dependencies, `useCallback()` can help reduce the number of effect executions.
- For object dependencies, destructuring can help reduce the number of effect executions.

WHAT'S NEXT?

Dealing with side effects is a common problem when building apps because most apps need some kind of side effects (for example, sending an HTTP request) to work correctly. Therefore, side effects aren't a problem themselves, but they can cause problems (for example, infinite loops) if handled incorrectly.

With the knowledge gained in this chapter, you know how to handle side effects efficiently with **`useEffect()`** and related key concepts.

At this point in the book, you now know all the key React concepts you need to build feature-rich web applications. The next chapter will look behind the scenes of React and explore how it works internally. You will also learn about some common optimization techniques that can make your apps more performant.

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 at https://packt.link/k0K8S:

1. How would you define a side effect?
2. What's a potential problem that could arise with some side effects in React components?
3. How does the **`useEffect()`** Hook work?
4. Which values should *not* be added to the **`useEffect()`** dependencies array?
5. Which value can be returned by the effect function? And which kind of value *must not* be returned?

APPLY WHAT YOU LEARNED

Now that you know about effects, you can add even more exciting features to your React apps. Fetching data via HTTP upon rendering a component is just as easy as accessing browser storage when some state changes.

In the following section, you'll find an activity that allows you to practice working with effects and **`useEffect()`**. As always, you will need to employ some of the concepts covered in earlier chapters (such as working with state).

ACTIVITY 8.1: BUILDING A BASIC BLOG

In this activity, you must add logic to an existing React app to render a list of blog post titles fetched from a backend web API and submit newly added blog posts to that same API. The backend API used is https://jsonplaceholder.typicode.com/, which is a dummy API that doesn't actually store any data you send to it. It will always return the same dummy data, but it's perfect for practicing sending HTTP requests.

As a bonus, you can also add logic to change the text of the submit button while the HTTP request to save the new blog post is on its way.

Use your knowledge about effects and browser-side HTTP requests to implement a solution.

> **NOTE**
>
> You can find the starting code for this activity at https://packt.link/C3bLv. 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.
>
> For this activity, you need to know how to send HTTP requests (`GET`, `POST`, and so on) via JavaScript (for example, via the `fetch()` function or with the help of a third-party library). If you don't have that knowledge yet, this resource can get you started: http://packt.link/DJ6Hx.

After downloading the code and running `npm install` in the project folder to install all required dependencies, the solution steps are as follows:

1. Send a `GET` HTTP request to the dummy API to fetch blog posts inside the `App` component (when the component is first rendered).
2. Display the fetched dummy blog posts on the screen.
3. Handle form submissions and send a `POST` HTTP request (with some dummy data) to the dummy backend API.
4. **Bonus:** Set the button caption to "Saving..." while the request is on its way (and to "Save" when it's not).

The expected result should be a user interface that looks like this:

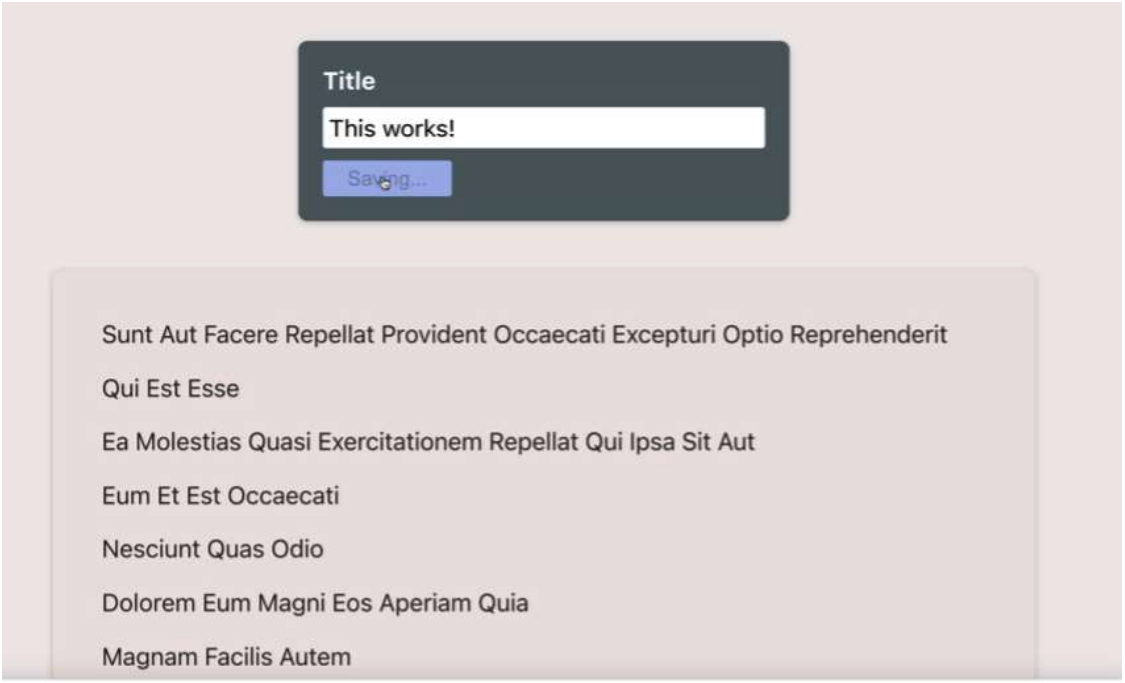

Figure 8.9: The final user interface

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

9

BEHIND THE SCENES OF REACT AND OPTIMIZATION OPPORTUNITIES

LEARNING OBJECTIVES

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

- Avoid unnecessary component update evaluations via React's **`memo()`** function.
- Avoid unnecessary code execution via the **`useMemo()`** and **`useCallback()`** Hooks.
- Load optional code lazily, only when it's needed, via React's **`lazy()`** function.
- Use React's developer tools to analyze your app.

INTRODUCTION

Using all the features covered up to this point, you can build non-trivial React apps and therefore highly interactive and reactive user interfaces.

This chapter, while introducing some new functions and concepts, will not provide you with tools that would enable you to build even more advanced web apps. You will not learn about groundbreaking, key concepts such as state or props (though you will learn about more advanced concepts in later chapters).

Instead, this chapter allows you to look behind the scenes of React. You will learn how React calculates required DOM updates, and how it ensures that such updates happen without impacting performance in an unacceptable way. You will also learn about some other optimization techniques employed by React—all with the goal of ensuring that your React app runs as smoothly as possible.

Besides this look behind the scenes, you will learn about various built-in functions and concepts that can be used to further optimize app performance. This chapter will not only introduce those concepts but also explain **why** they exist, **how** they should be used, and **when** to use which feature.

REVISITING COMPONENT EVALUATIONS AND UPDATES

Before exploring React's internal workings, it makes sense to briefly revisit React's logic for executing component functions.

Component functions are executed whenever their internal state changes or their parent component function is executed again. The latter happens because, if a parent component function is called, its entire JSX code (which points at the child component function) is re-evaluated. Any component functions referenced in that JSX code are therefore also invoked again.

Consider a component structure like this:

```
function NestedChild() {
  console.log('<NestedChild /> is called.');

  return <p id="nested-child">A component, deeply nested into the
component tree.</p>;
}

function Child() {
  console.log('<Child /> is called.');
```

```
  return (
    <div id="child">
      <p>
        A component, rendered inside another component, containing yet
another
        component.
      </p>
      <NestedChild />
    </div>
  );
}

function Parent() {
  console.log('<Parent /> is called.');

  const [counter, setCounter] = useState(0);

  function incCounterHandler() {
    setCounter((prevCounter) => prevCounter + 1);
  }

  return (
    <div id="parent">
      <p>A component, nested into App, containing another component
(Child).</p>
      <p>Counter: {counter}</p>
      <button onClick={incCounterHandler}>Increment</button>
      <Child />
    </div>
  );
}
```

In this example structure, the **`Parent`** component renders a **`<div>`** with two paragraphs, a button, and another component: the **`Child`** component. That component in turn outputs a **`<div>`** with a paragraph and yet another component: the **`NestedChild`** component (which then only outputs a paragraph).

The **`Parent`** component also manages some state (a dummy counter), which is changed whenever the button is clicked. All three components print a message via **`console.log()`**, simply to make it easy to spot when each component is called by React.

The following screenshot shows those components in action—after the button was clicked:

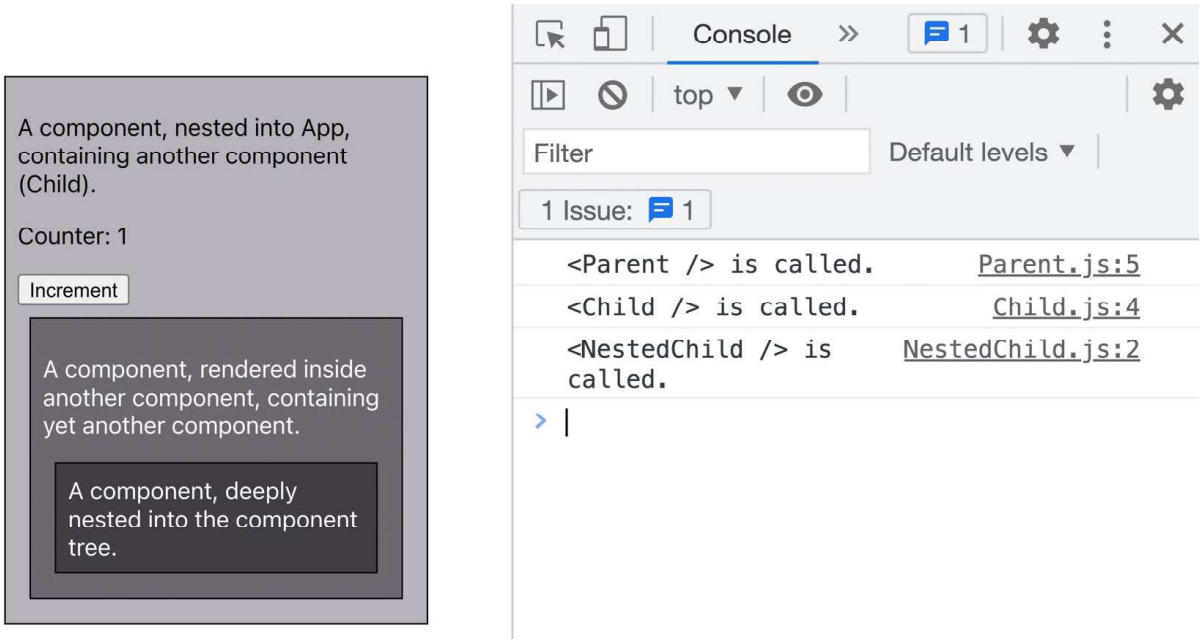

Figure 9.1: Each component function is executed

In this screenshot, you can not only see how the components are nested into each other, but also how they are all invoked by React when the **`Increment`** button is clicked. **`Child`** and **`NestedChild`** are invoked even though they don't manage or use any state. But since they are a child (**`Child`**) or descendent (**`NestedChild`**) of the **`Parent`** component, which did receive a state change, the nested component functions are called as well.

Understanding this flow of component function execution is important because this flow implies that any component function invocation also influences its descendent components. It also shows you how frequently component functions may be invoked by React and how many component functions may be affected by a single state change.

Therefore, there's one important question that should be answered: what happens to the actual page DOM (i.e., to the loaded and rendered website in the browser) when one or more component functions are invoked? Is the DOM recreated? Is the rendered UI updated?

WHAT HAPPENS WHEN A COMPONENT FUNCTION IS CALLED

Whenever a component function is executed, React evaluates whether or not the rendered user interface (i.e., the DOM of the loaded page) must be updated.

This is important: React **evaluates** whether an update is needed. It's not forcing an update automatically!

Internally, React does not take the JSX code returned by a component (or multiple components) and replace the page DOM with it.

That could be done, but it would mean that every component function execution would lead to some form of DOM manipulation—even if it's just a replacement of the old DOM content with a new, similar version. In the example shown above, the **`Child`** and **`NestedChild`** JSX code would be used to replace the currently rendered DOM every time those component functions were executed.

As you can see in the example above, those component functions are executed quite frequently. But the returned JSX code is always the same because it's static. It does not contain any dynamic values (e.g., state or props).

If the actual page DOM were replaced with the DOM elements implied by the returned JSX code, the visual result would always be the same. But there still would be some DOM manipulation behind the scenes. And that's a problem, because manipulating the DOM is quite a performance-intensive task—especially when done with a high frequency. Removing and adding or updating DOM elements should therefore only be done when needed—not unnecessarily.

Because of this, React does not throw away the current DOM and replace it with the new DOM (implied by the JSX code), just because a component function was executed. Instead, React first checks whether an update is needed. And if it's needed, only the parts of the DOM that need to change are replaced or updated.

For determining whether an update is needed (and where), React uses a concept called the **virtual DOM**.

THE VIRTUAL DOM VS THE REAL DOM

To determine whether (and where) a DOM update might be needed, React (specifically, the **`react-dom`** package) compares the current DOM structure to the structure implied by the JSX code returned by the executed component functions. If there's a difference, the DOM is updated accordingly; otherwise, it's left untouched.

However, just as manipulating the DOM is relatively performance-intensive, reading the DOM is as well. Even without changing anything in the DOM, reaching out to it, traversing the DOM elements, and deriving the structure from it is something you typically want to reduce to a minimum.

If multiple component functions are executed and each trigger a process where the rendered DOM elements are read and compared to the JSX structure implied by the invoked component functions, the rendered DOM will be hit with read operations multiple times within a very short time frame.

For bigger React apps that are made up of dozens, hundreds, or even thousands of components, it's highly probable that dozens of component function executions might occur within a single second. If that were to lead to the same amount of DOM read operations, there's a quite high chance that the web app would feel slow or laggy to the user.

That's why React does not use the real DOM to determine whether any user interface updates are needed. Instead, it constructs and manages a virtual DOM internally—an in-memory representation of the DOM that's rendered in the browser. The virtual DOM is not a browser feature, but a React feature. You can think of it as a deeply nested JavaScript object that reflects the components of your web app, including all the built-in HTML components such as **`<div>`**, **`<p>`**, etc. (that is, the actual HTML elements that should show up on the page in the end).

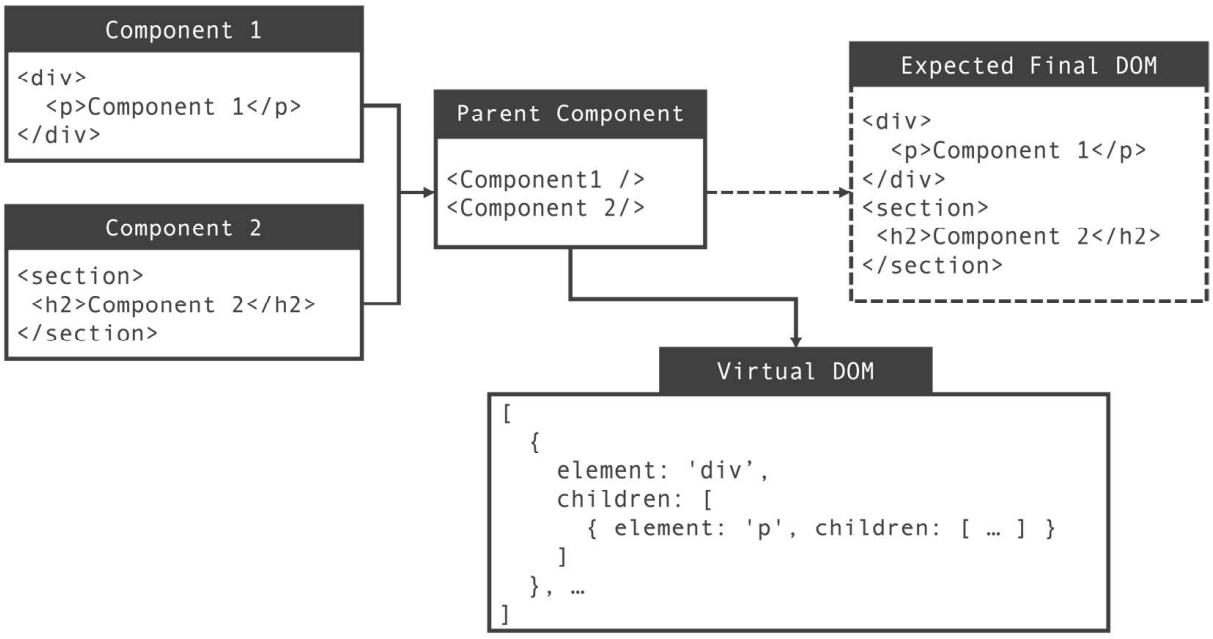

Figure 9.2: React manages a virtual representation of the expected element structure

In the figure above, you can see that the expected element structure (in other words, the expected final DOM) is actually stored as a JavaScript object (or an array with a list of objects). This is the virtual DOM, which is managed by React and used for identifying required DOM updates.

> **NOTE**
>
> Please note that the actual structure of the virtual DOM is more complex than the structure shown in the image. The chart above aims to give you an idea of what the virtual DOM is and how it might look. It's not an exact technical representation of the JavaScript data structure managed by React.

React manages this virtual DOM because comparing this virtual DOM to the expected user interface state is much less performance-intensive than reaching out to the real DOM.

Whenever a component function is called, React compares the returned JSX code to the respective virtual DOM nodes stored in the virtual DOM. If differences are detected, React will determine which changes are needed to update the DOM. Once the required adjustments are derived, these changes are applied to both the virtual and the real DOM. This ensures that the real DOM reflects the expected user interface state without having to reach out to it or update it all the time.

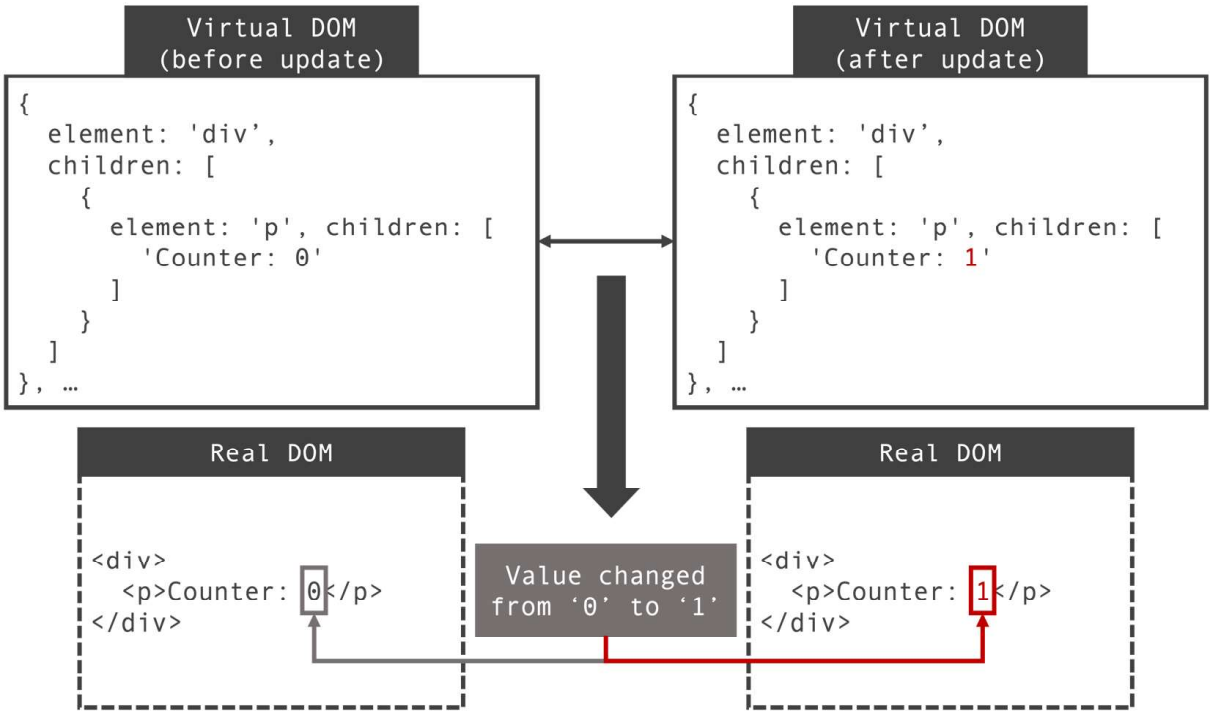

Figure 9.3: React detects required updates via the virtual DOM

In the figure above, you can see how React compares the current DOM and the expected structure with help of the virtual DOM first, before reaching out to the real DOM to manipulate it accordingly.

As a React developer, you don't need to actively interact with the virtual DOM. Technically, you don't even need to know that it exists and that React uses it internally. But it's always helpful to understand the tool (React in that case) you're working with. It's good to know that React is doing various performance optimizations for you and that you get those on top of the many other features that make your life as a developer (hopefully) easier.

STATE BATCHING

Since React uses this concept of a virtual DOM, frequent component function executions aren't a huge problem. But of course, even if comparisons are only conducted virtually, there is still some internal code that must be executed. Even with the virtual DOM, performance could degrade if lots of unnecessary (and at the same time quite complex) virtual DOM comparisons must be made.

One scenario where unnecessary comparisons are performed is in the execution of multiple sequential state updates. Since each state update causes the component function to be executed again (as well as all potential nested components), multiple state updates that are performed together (for example., in the same event handler function) will cause multiple component function invocations.

Consider this example:

```
function App() {
  const [counter, setCounter] = useState(0);
  const [showCounter, setShowCounter] = useState(false);

  function incCounterHandler() {
    setCounter((prevCounter) => prevCounter + 1);
    if (!showCounter) {
      setShowCounter(true);
    }
  }

  return (
    <>
      <p>Click to increment + show or hide the counter</p>
      <button onClick={incCounterHandler}>Increment</button>
      {showCounter && <p>Counter: {counter}</p>}
    </>
  );
}
```

This component contains two state values: **`counter`** and **`showCounter`**. When the button is clicked, the counter is incremented by **`1`**. Now, **`showCounter`** is set to **`true`** if it was set to **`false`**. Therefore, the first time the button is clicked, both the **`counter`** and the **`showCounter`** states are changed (because **`showCounter`** is **`false`** initially).

Since two state values are changed, the expectation would be that the **`App`** component function is called twice by React—because every state update causes the connected component function to be invoked again.

However, if you add a **`console.log()`** statement to the **`App`** component function (to track how often it's executed), you will see that it's only invoked once, when the **`Increment`** button is clicked:

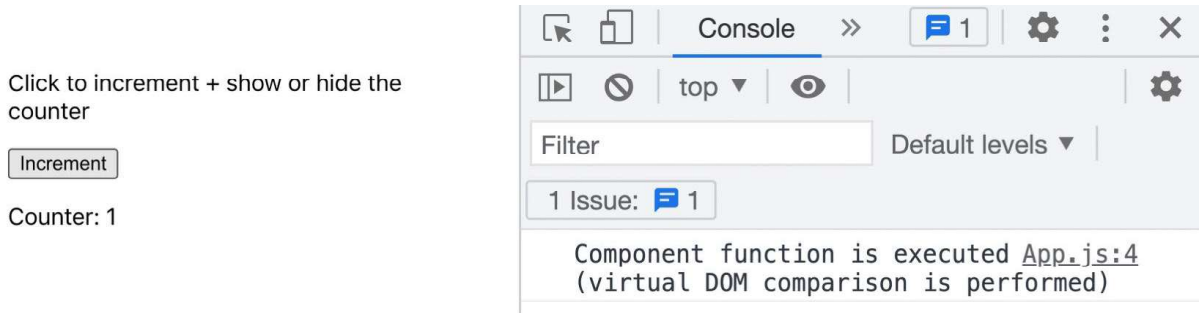

Figure 9.4: Only one console log message is displayed

> **NOTE**
>
> If you're seeing two log messages instead of one, make sure you're not using React's "Strict Mode." When running in Strict Mode during development, React executes component functions more often than it normally would.
>
> If necessary, you can disable Strict Mode by removing the **`<React.StrictMode>`** component from your **`index.js`** file. You will learn more about React's Strict Mode toward the end of this chapter.

This behavior is called **state batching**. React performs state batching when multiple state updates are initiated from the same place in your code (e.g., from inside the same event handler function).

It's a built-in functionality that ensures that your component functions are not called more often than needed. This prevents unnecessary virtual DOM comparisons.

State batching is a very useful mechanism. But there is another kind of unnecessary component evaluation that it does not prevent: child component functions that get executed when the parent component function is called.

AVOIDING UNNECESSARY CHILD COMPONENT EVALUATIONS

Whenever a component function is invoked (because its state changed, for example), any nested component functions will be called as well. See the first section of this chapter for more details.

As you saw in the example in the first section of this chapter, it is often the case that those nested components don't actually need to be evaluated again. They might not depend on the state value that changed in the parent component. They might not even depend on any values of the parent component at all.

Here's an example where the parent component function contains some state that is not used by the child component:

```
function Error({ message }) {
  if (!message) {
    return null;
  }

  return <p className={classes.error}>{message}</p>;
}


function Form() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [errorMessage, setErrorMessage] = useState();

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

  function submitHandler(event) {
    event.preventDefault();
    if (!enteredEmail.endsWith('.com')) {
      setErrorMessage('Email must end with .com.');
    }
  }

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

```
        <input
          id="email"
          type="email"
          value={enteredEmail}
          onChange={updateEmailHandler}
        />
      </div>
      <Error message={errorMessage} />
      <button>Sign Up</button>
    </form>
  );
}
```

> **NOTE**
>
> You can find the complete example code on GitHub at https://packt.link/z3Hg2.

In this example, the **`Error`** component relies on the **`message`** prop, which is set to the value stored in the **`errorMessage`** state of the **`Form`** component. However, the **`Form`** component also manages an **`enteredEmail`** state, which is not used (not received via props) by the **`Error`** component. Therefore, changes to the **`enteredEmail`** state will cause the **`Error`** component to be executed again, despite the component not needing that value.

You can track the unnecessary **`Error`** component function invocations by adding a **`console.log()`** statement to that component function:

```
function Error({ message }) {
  console.log('<Error /> component function is executed.');
  if (!message) {
    return null;
  }

  return <p className={classes.error}>{message}</p>;
}
```

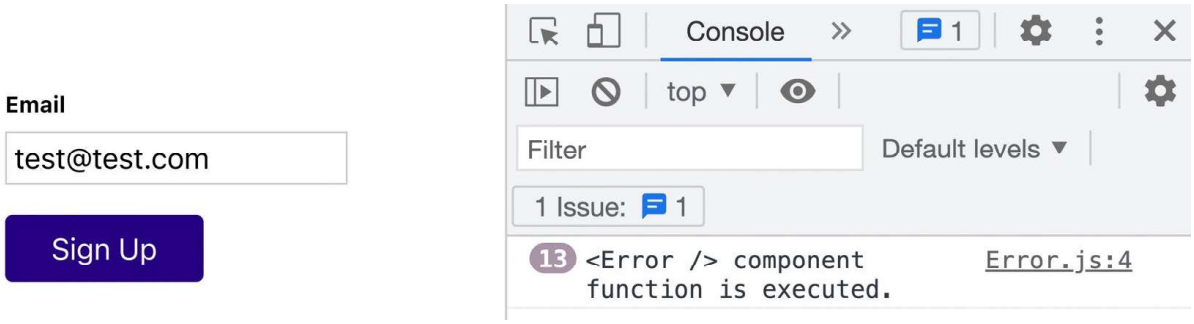

Figure 9.5: The Error component function is executed for every keystroke

In the preceding screenshot, you can see that the **`Error`** component function is executed for every keystroke on the input field (that is, once for every **`enteredEmail`** state change).

This is in line with what you have learned previously, but it is also unnecessary. The **`Error`** component does depend on the **`errorMessage`** state and should certainly be re-evaluated whenever that state changes, but executing the **`Error`** component function because the **`enteredEmail`** state value was updated is clearly not required.

That's why React offers another built-in function that you can use to control (and prevent) this behavior: the **`memo()`** function.

`memo` is imported from **`react`** and is used like this:

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

import classes from './Error.module.css';

function Error({ message }) {
  console.log('<Error /> component function is executed.');
  if (!message) {
    return null;
  }

  return <p className={classes.error}>{message}</p>;
}

export default memo(Error);
```

You wrap the component function that should be protected from unnecessary, parent-initiated re-evaluations with **`memo()`**. This causes React to check whether the component's props did change, compared to the last time the component function was called. If prop values are equal, the component function is not executed again.

By adding **`memo()`**, the unnecessary component function invocations are avoided, as shown below:

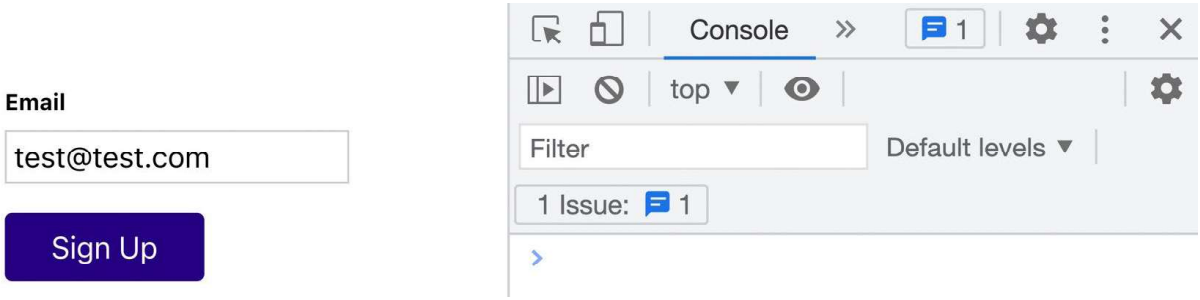

Figure 9.6: No console log messages appear

As you can see in the figure, no messages are printed to the console. This proves that unnecessary component executions are avoided (remember: before adding **`memo()`**, many messages were printed to the console).

`memo()` also takes an optional second argument that can be used to add your own logic to determine whether prop values have changed or not. This can be useful if you're dealing with more complex prop values (e.g., objects or arrays) where custom comparison logic might be needed, as in the following example:

```
memo(SomeComponent, function(prevProps, nextProps) {
  return prevProps.user.firstName !== nextProps.user.firstName;
});
```

The (optional) second argument passed to **`memo()`** must be a function that automatically receives the previous props object and the next props object. The function then must return **`true`** if the component (**`SomeComponent`**, in this example) should be re-evaluated and **`false`** if it should not.

Often, the second argument is not needed because the default behavior of **`memo()`** (where it compares all props for inequality) is exactly what you need. But if more customization or control is needed, **`memo()`** allows you to add your custom logic.

With **`memo()`** in your toolbox, it's tempting to wrap every React component function with **`memo()`**. Why wouldn't you do it? After all, it avoids unnecessary component function executions.

But there is a very good reason for not using **`memo()`** in all your component functions. Indeed, you typically only want to use it in a few selected components.

Because avoiding unnecessary component re-evaluations with using **`memo()`** comes at a cost: comparing props (old versus new) also requires some code to run. It's not "free." It's especially problematic if the result then is that the component function must be executed again (because props changed) as, in that case, you will have spent time comparing props just to then invoke the component function anyway.

Hence **`memo()`** really only makes sense if you have relatively simple props (i.e., props with no deeply nested objects that you need to compare manually with a custom comparison function) and most parent component state changes don't affect those props of the child component. And even in those cases, if you have a relatively simple component function (i.e., without any complex logic in it), using **`memo()`** still might not yield any measurable benefit.

The example code above (the **`Error`** component) is a good example: in theory, using **`memo()`** makes sense here. Most state changes in the parent component won't affect **`Error`**, and the prop comparison will be very simple because it's just one prop (the **`message`** prop, which holds a string) that must be compared. But despite that, using **`memo()`** to wrap **`Error`** will very likely not be worth it. **`Error`** is an extremely basic component with no complex logic in it. It simply doesn't matter if the component function gets invoked frequently. Hence, using **`memo()`** in this spot would be absolutely fine—but so is not using it.

A great spot to use **`memo()`**, on the other hand, is a component that's relatively close to the top of the component tree (or of a deeply nested branch of components in the component tree). If you are able to avoid unnecessary executions of that one component via **`memo()`**, you also implicitly avoid unnecessary executions of all nested components beneath that one component. This is visually illustrated in the diagram below:

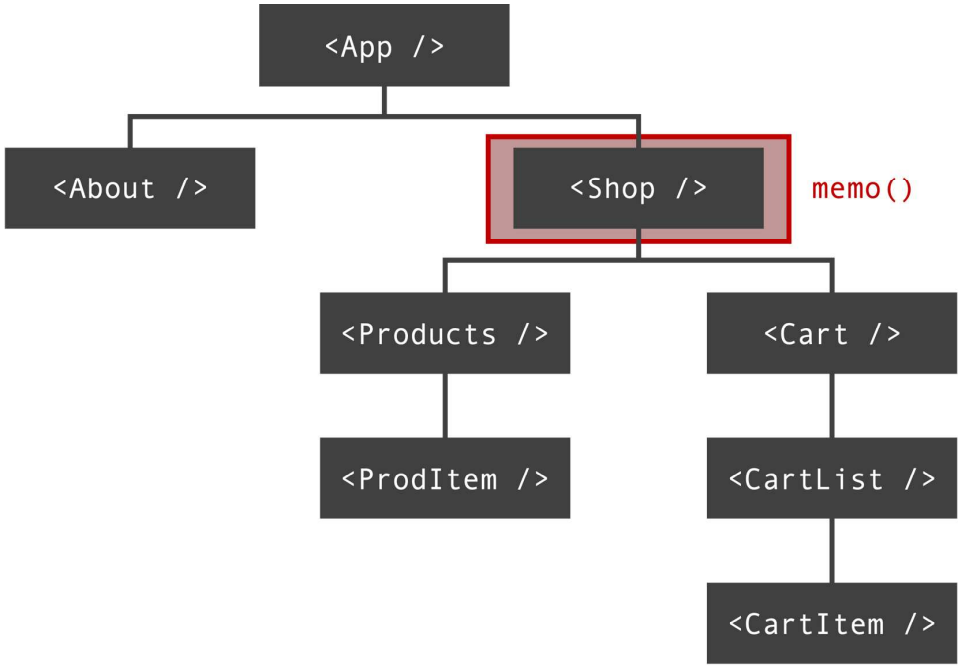

Figure 9.7: Using memo at the start of a component tree branch

In the preceding figure, **`memo()`** is used on the **`Shop`** component, which has multiple nested descendent components. Without **`memo()`**, whenever the **`Shop`** component function gets invoked, **`Products`**, **`ProdItem`**, **`Cart`**, etc. would also be executed. With **`memo()`**, assuming that it's able to avoid some unnecessary executions of the **`Shop`** component function, all those descendent components are no longer evaluated.

AVOIDING COSTLY COMPUTATIONS

The **`memo()`** function can help avoid unnecessary component function executions. As mentioned in the previous section, this is especially valuable if a component function performs a lot of work (e.g., sorting a long list).

But as a React developer, you will also encounter situations in which you have a work-intensive component that needs to be executed again because some prop value changed. In such cases, using **`memo()`** won't prevent the component function from executing again. But the prop that changed might not be needed for the performance-intensive task that is performed as part of the component.

Consider the following example:

```
function sortItems(items) {
  console.log('Sorting');
  return items.sort(function (a, b) {
    if (a.id > b.id) {
      return 1;
    } else if (a.id < b.id) {
      return -1;
    }
    return 0;
  });
}

function List({ items, maxNumber }) {
  const sortedItems = sortItems(items);

  const listItems = sortedItems.slice(0, maxNumber);

  return (
    <ul>
      {listItems.map((item) => (
        <li key={item.id}>
          {item.title} (ID: {item.id})
        </li>
      ))}
    </ul>
  );
}

export default List;
```

The **`List`** component receives two prop values: **`items`** and **`maxNumber`**. It then calls **`sortItems()`** to sort the items by **`id`**. Thereafter, the sorted list is limited to a certain amount (**`maxNumber`**) of items. As a last step, the sorted and shortened list is then rendered to the screen via **`map()`** in the JSX code.

> **NOTE**
>
> A full example app can be found on GitHub at https://packt.link/dk9ag.

Depending on the number of items passed to the **`List`** component, sorting it can take a significant amount of time (for very long lists, even up to a few seconds). It's definitely not an operation you want to perform unnecessarily or too frequently. The list needs to be sorted whenever **`items`** change, but it should not be sorted if **`maxNumber`** changes—because this does not impact the items in the list (i.e., it doesn't affect the order). But with the code snippet shared above, **`sortItems()`** will be executed whenever either of the two prop values changes, no matter whether it's **`items`** or **`maxNumber`**.

As a result, when running the app and changing the number of displayed items, you can see multiple **`"Sorting"`** log messages—implying that **`sortItems()`** was executed every time the number of items was changed.

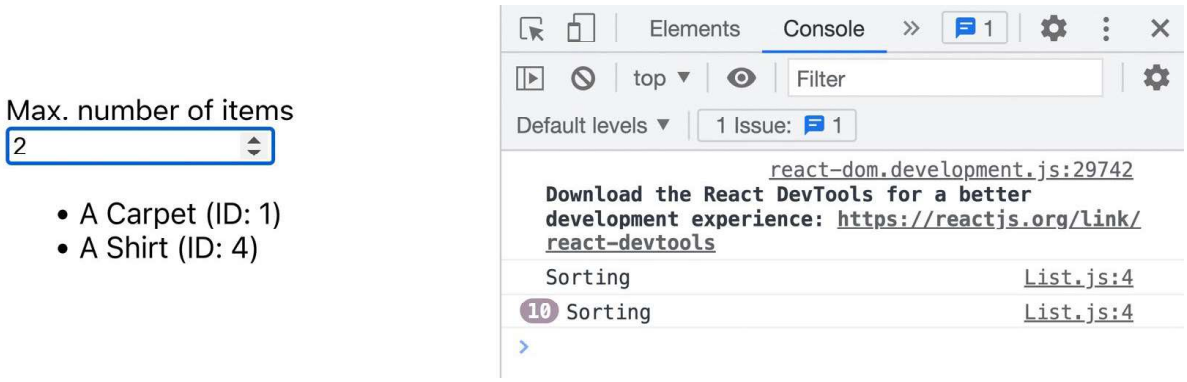

Figure 9.8: Multiple "Sorting" log messages appear in the console

The **`memo()`** function won't help here because the **`List`** component function should (and will) execute whenever **`items`** or **`maxNumber`** change. **`memo()`** does not help control partial code execution inside the component function.

For that, you can use another feature provided by React: the **`useMemo()`** Hook.

`useMemo()` can be used to wrap a compute-intensive operation. For it to work correctly, you also must define a list of dependencies that should cause the operation to be executed again. To some extent, it's similar to **`useEffect()`** (which also wraps an operation and defines a list of dependencies), but the key difference is that **`useMemo()`** runs at the same time as the rest of the code in the component function, whereas **`useEffect()`** executes the wrapped logic after the component function execution finished. **`useEffect()`** should not be used for optimizing compute-intensive tasks but for side effects.

`useMemo()`, on the other hand, exists to control the execution of performance-intensive tasks. Applied to the example mentioned above, the code can be adjusted like this:

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

function List({ items, maxNumber }) {
  const sortedItems = useMemo(
    function () {
      console.log('Sorting');
      return items.sort(function (a, b) {
        if (a.id > b.id) {
          return 1;
        } else if (a.id < b.id) {
          return -1;
        }
        return 0;
      });
    },
    [items]
  );

  const listItems = sortedItems.slice(0, maxNumber);

  return (
    <ul>
      {listItems.map((item) => (
        <li key={item.id}>
          {item.title} (ID: {item.id})
        </li>
      ))}
    </ul>
```

```
  );
}

export default List;
```

`useMemo()` wraps an anonymous function (the function that previously existed as a named function, **`sortItems`**), which contains the entire sorting code. The second argument passed to **`useMemo()`** is the array of dependencies for which the function should be executed again (when a dependency value changes). In this case, **`items`** is the only dependency of the wrapped function, and so that value is added to the array.

With **`useMemo()`** used like this, the sorting logic will only execute when items change, not when **`maxNumber`** (or anything else) changes. As a result, you see **`"Sorting"`** being output in the developer tools console only once:

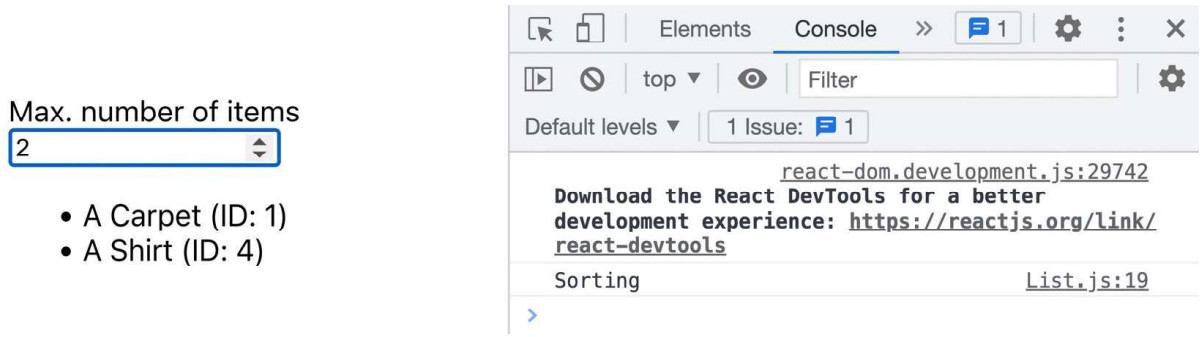

Figure 9.9: Only one "Sorting" output in the console

`useMemo()` can be very useful for controlling code execution inside of your component functions. It can be a great addition to **`memo()`** (which controls the overall component function execution). But also like **`memo()`**, you should not start wrapping all your logic with **`useMemo()`**. Only use it for very performance-intensive computations since checking for dependency changes and storing and retrieving past computation results (which **`useMemo()`** does internally) also comes at a performance cost.

UTILIZING USECALLBACK()

In the previous chapter, you learned about **`useCallback()`**. Just like **`useMemo()`** can be used for "expensive" calculations, **`useCallback()`** can be used to avoid unnecessary function re-creations. In the context of this chapter, **`useCallback()`** can be helpful because, in conjunction with **`memo()`** or **`useMemo()`**, it can help you avoid unnecessary code execution. It can help you in cases where a function is passed as a prop (i.e., where you might use **`memo()`**) or is used as a dependency in some "expensive" computation (i.e., possibly solved via **`useMemo()`**).

Here's an example where **`useCallback()`** can be combined with **`memo()`** to avoid unnecessary component function executions:

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

import classes from './Error.module.css';

function Error({ message, onClearError }) {
  console.log('<Error /> component function is executed.');
  if (!message) {
    return null;
  }

  return (
    <div className={classes.error}>
      <p>{message}</p>
      <button className={classes.errorBtn} onClick={onClearError}>X</
button>
    </div>
  );
}

export default memo(Error);
```

The **`Error`** component is wrapped with the **`memo()`** function and so will only execute if one of the received prop values changes.

The **`Error`** component is used by another component, the **`Form`** component, like this:

```
function Form() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [errorMessage, setErrorMessage] = useState();

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

  function submitHandler(event) {
    event.preventDefault();
    if (!enteredEmail.endsWith('.com')) {
      setErrorMessage('Email must end with .com.');
```

```
    }
  }

  function clearErrorHandler() {
    setErrorMessage(null);
  }

  return (
    <form className={classes.form} onSubmit={submitHandler}>
      <div className={classes.control}>
        <label htmlFor="email">Email</label>
        <input
          id="email"
          type="email"
          value={enteredEmail}
          onChange={updateEmailHandler}
        />
      </div>
      <Error message={errorMessage} onClearError={clearErrorHandler} />
      <button>Sign Up</button>
    </form>
  );
}
```

In this component, the **`Error`** component receives a pointer to the **`clearErrorHandler`** function (as a value for the **`onClearError`** prop). You might recall a very similar example from earlier in this chapter (from the *"Avoiding Unnecessary Child Component Evaluations"* section). There, **`memo()`** was used to ensure that the **`Error`** component function was not invoked when **`enteredEmail`** changed (because its value was not used in the **`Error`** component function at all).

Now with the adjusted example and the **`clearErrorHandler`** function pointer passed to **`Error`**, **`memo()`** unfortunately isn't preventing component function executions anymore. Why? Because functions are objects in JavaScript, and the **`clearErrorHandler`** function is recreated every time the **`Form`** component function is executed (which happens on every state change, including changes to the **`enteredEmail`** state).

Since a new function object is created for every state change, **`clearErrorHandler`** is technically a different value for every execution of the **`Form`** component. Therefore, the **`Error`** component receives a new **`onClearError`** prop value whenever the **`Form`** component function is invoked. To **`memo()`**, the old and new **`clearErrorHandler`** function objects are different from each other, and it therefore will not stop the **`Error`** component function from running again.

That's exactly where **`useCallback()`** can help:

```
const clearErrorHandler = useCallback(() => {
  setErrorMessage(null);
}, []);
```

By wrapping **`clearErrorHandler`** with **`useCallback()`**, the re-creation of the function is prevented, and so no new function object is passed to the **`Error`** component. Hence, **`memo()`** is able to detect equality between the old and new **`onClearError`** prop value and prevents unnecessary function component executions again.

Similarly, **`useCallback()`** can be used in conjunction with **`useMemo()`**. If the compute-intensive operation wrapped with **`useMemo()`** uses a function as a dependency, you can use **`useCallback()`** to ensure that this dependent function is not recreated unnecessarily.

AVOIDING UNNECESSARY CODE DOWNLOAD

Thus far, this chapter has mostly discussed strategies for avoiding unnecessary code execution (and that it's not always worth the effort). But it's not just the execution of code that can be an issue. It's also not great if your website visitors have to download lots of code that might never be executed at all. Because every kilobyte of JavaScript code that has to be downloaded will slow down the initial loading time of your web page—not just because of the time it takes to download the code bundle (which can be significant, if users are on a slow network and code bundles are big) but also because the browser has to parse all the downloaded code before your page becomes interactive.

For this reason, a lot of community and ecosystem effort is spent on reducing JavaScript code bundle sizes. Minification (automatic shortening of variable names and other measures to reduce the final code) and compression can help a lot and is therefore a common technique. Actually, projects created with **`create-react-app`** already come with a build workflow (initiated by running **`npm run build`**), which will produce a production-optimized code bundle that is as small as possible.

But there also are steps that can be taken by you, the developer, to reduce the overall code bundle size:

1. Try to write short and concise code.
2. Be thoughtful about including lots of third-party libraries and don't use them unless you really need to.
3. Consider using code-splitting techniques.

The first point should be fairly obvious. If you write less code, your website visitors have less code to download. Therefore, trying to be concise and write optimized code makes sense.

The second point should also make sense. For some tasks, you will actually save code by including third-party libraries that may be much more elaborate than the code solution you might come up with. But there are also situations and tasks in which you might get away with writing your own code or using some built-in function instead of including a third-party library. You should at least always think about this alternative and only include third-party libraries you absolutely need.

The last point is something React can help with.

REDUCING BUNDLE SIZES VIA CODE SPLITTING (LAZY LOADING)

React exposes a **`lazy()`** function that can be used to load component code conditionally—meaning, only when it's actually needed (instead of upfront).

Consider the following example, consisting of two components working together.

A **`DateCalculator`** component is defined like this:

```
import { useState } from 'react';
import { add, differenceInDays, format, parseISO } from 'date-fns';

import classes from './DateCalculator.module.css';

const initialStartDate = new Date();
const initialEndDate = add(initialStartDate, { days: 1 });

function DateCalculator() {
  const [startDate, setStartDate] = useState(
    format(initialStartDate, 'yyyy-MM-dd')
  );
  const [endDate, setEndDate] = useState(format(initialEndDate, 'yyyy-
```

```
MM-dd'));

  const daysDiff = differenceInDays(parseISO(endDate),
parseISO(startDate));

  function updateStartDateHandler(event) {
    setStartDate(event.target.value);
  }

  function updateEndDateHandler(event) {
    setEndDate(event.target.value);
  }

  return (
    <div className={classes.calculator}>
      <p>Calculate the difference (in days) between two dates.</p>
      <div className={classes.control}>
        <label htmlFor="start">Start Date</label>
        <input
          id="start"
          type="date"
          value={startDate}
          onChange={updateStartDateHandler}
        />
      </div>
      <div className={classes.control}>
        <label htmlFor="end">End Date</label>
        <input
          id="end"
          type="date"
          value={endDate}
          onChange={updateEndDateHandler}
        />
      </div>
      <p className={classes.difference}>Difference: {daysDiff} days</p>
    </div>
  );
}

export default DateCalculator;
```

This **`DateCalculator`** component is then rendered conditionally by the **`App`** component:

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

import DateCalculator from './components/DateCalculator';

function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);

  function openDateCalcHandler() {
    setShowDateCalc(true);
  }

  return (
    <>
      <p>This app might be doing all kinds of things.</p>
      <p>
        But you can also open a calculator which calculates the difference
        between two dates.
      </p>
      <button onClick={openDateCalcHandler}>Open Calculator</button>
      {showDateCalc && <DateCalculator />}
    </>
  );
}

export default App;
```

In this example, the **`DateCalculator`** component uses a third-party library (the **`date-fns`** library) to access various date-related utility functions (for example, a function for calculating the difference between two dates, or **`differenceInDays`**).

The component then accepts two date values and calculates the difference between those dates in days—though the actual logic of the component isn't too important here. What is important is the fact that a third-party library and various utility functions are used. This adds quite a bit of JavaScript code to the overall code bundle, and all that code must be downloaded when the entire website is loaded for the first time, even though the date calculator isn't even visible at that point in time (because it is rendered conditionally).

You can see the overall bundle being downloaded in the following screenshot:

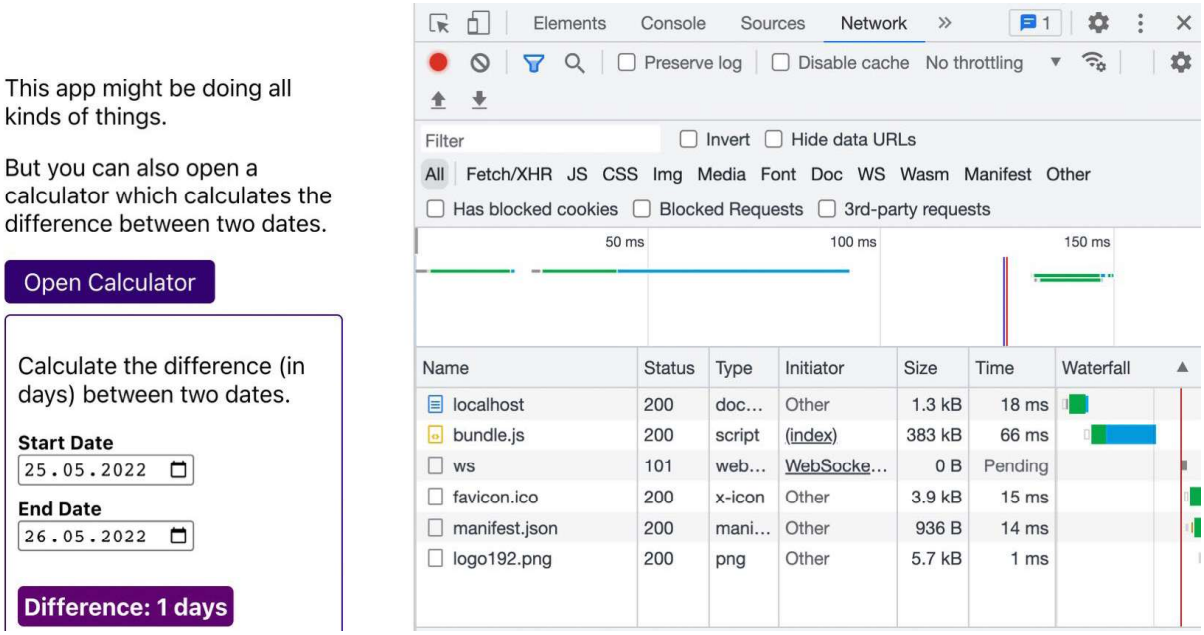

Figure 9.10: Only one bundle file is downloaded

The **`Network`** tab in the browser's developer tools reveals outgoing network requests. As you can see in the screenshot, a JavaScript bundle file is downloaded. You won't see any extra requests being sent when the button is clicked. This implies that all the code, including the code needed for **`DateCalculator`**, was downloaded upfront.

That's where code splitting with React's **`lazy()`** function becomes useful.

This function can be wrapped around a dynamic import to load the imported component only once it's needed.

> **NOTE**
>
> For further information on this topic, visit https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import#dynamic_imports.

In the preceding example, it would be used like this in the **App** component file:

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

const DateCalculator = lazy(() => import('./components/DateCalculator'));

function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);

  function openDateCalcHandler() {
    setShowDateCalc(true);
  }

  return (
    <>
      <p>This app might be doing all kinds of things.</p>
      <p>
        But you can also open a calculator which calculates the difference
        between two dates.
      </p>
      <button onClick={openDateCalcHandler}>Open Calculator</button>
      {showDateCalc && <DateCalculator />}
    </>
  );
}

export default App;
```

This alone won't do the trick though. You must also wrap the conditional JSX code, where the dynamically imported component is used, with another component provided by React: the **<Suspense>** component, like this:

```
import { lazy, Suspense, useState } from 'react';

const DateCalculator = lazy(() => import('./components/DateCalculator'));

function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);

  function openDateCalcHandler() {
    setShowDateCalc(true);
```

```
  }

  return (
    <>
      <p>This app might be doing all kinds of things.</p>
      <p>
        But you can also open a calculator which calculates the difference
        between two dates.
      </p>
      <button onClick={openDateCalcHandler}>Open Calculator</button>
      <Suspense fallback={<p>Loading...</p>}>
        {showDateCalc && <DateCalculator />}
      </Suspense>
    </>
  );
}

export default App;
```

> **NOTE**
>
> You can find the finished example code on GitHub at https://packt.link/wj5Pi.

`Suspense` is a component built into React, and you must wrap it around any conditional code that uses React's **`lazy()`** function. **`Suspense`** also has one mandatory prop that must be provided, the **`fallback`** prop, which expects a JSX value that will be rendered as fallback content until the dynamically loaded content is available.

`lazy()` leads to the overall JavaScript code being split up into multiple bundles. And the bundle that contains the **`DateCalculator`** component (and its dependencies, such as the **`date-fns`** library code) is only downloaded when it's needed—that is, when the button in the **`App`** component is clicked. If that download were to take a bit longer, the **`fallback`** content of **`Suspense`** would be shown on the screen in the meantime.

> **NOTE**
>
> React's **`Suspense`** component is not limited to being used in conjunction with the **`lazy()`** function. In the past it was, but starting with React 18, it can also be used to show some fallback content while other data (i.e., data from a database) is being fetched.
>
> Right now, capabilities are nonetheless quite limited. However, in future React versions, this should become more useful.

After adding **`lazy()`** and the **`Suspense`** component as described, a smaller bundle is initially downloaded. In addition, if the button is clicked, more code files are downloaded:

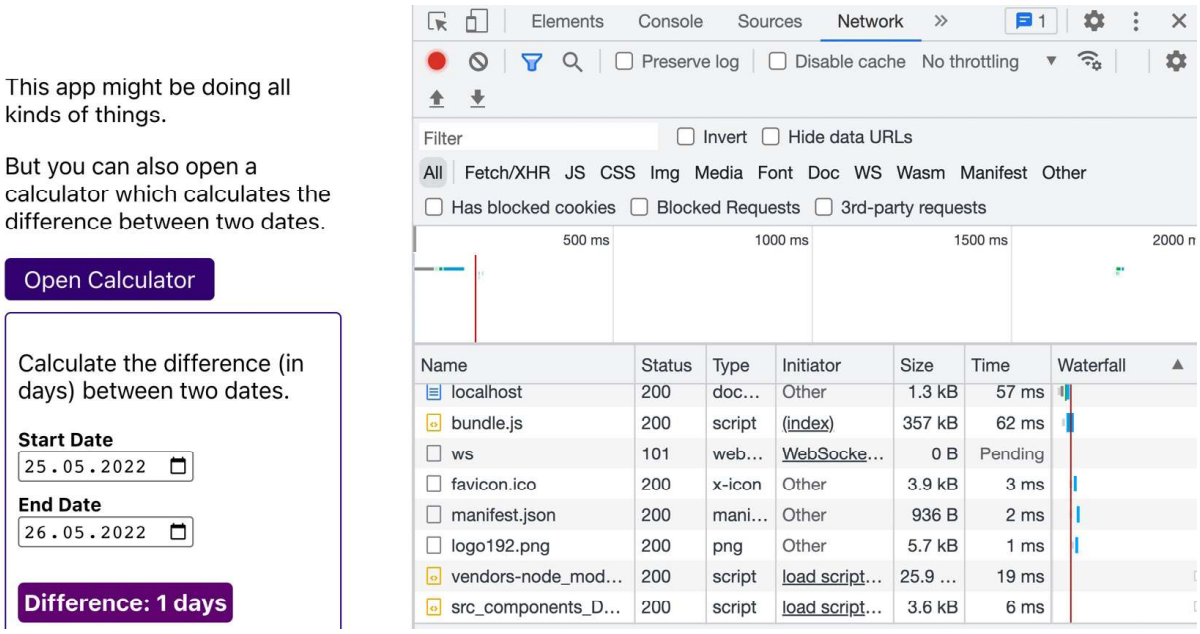

Figure 9.11: Multiple bundles are downloaded

Just as with all the other optimization techniques described thus far, the **`lazy()`** function is not a function you should start wrapping around all your imports. If an imported component is very small and simple (and doesn't use any third-party code), splitting the code isn't really worth it, especially since you have to consider that the additional HTTP request required for downloading the extra bundle also comes with some overhead.

It also doesn't make sense to use **`lazy()`** on components that will be loaded initially anyways. Only consider using it on conditionally loaded components.

STRICT MODE

Throughout this chapter, you have learned a lot about React's internals and various optimization techniques. Not really an optimization technique, but still related, is another feature offered by React called **Strict Mode**.

You may have stumbled across code like this before:

```
import React from 'react';

// ... other code ...
root.render(<React.StrictMode><App /></React.StrictMode >);
```

`<React.StrictMode>` is another built-in component provided by React. It doesn't render any visual element, but it will enable some extra checks that are performed behind the scenes by React.

Most checks are related to identifying the use of unsafe or legacy code (i.e., features that will be removed in the future). But there are also some checks that aim to help you identify potential problems with your code.

For example, when using strict mode, React will execute component functions twice and also unmount and remount every component whenever it mounts for the first time. This is done to ensure that you're managing your state and side effects in a consistent and correct way (for example, that you do have cleanup functions in your effect functions).

> **NOTE**
>
> Strict Mode only affects your app and its behavior during development. It does not influence your app once you build it for production. Extra checks of effects such as double component function execution will not be performed in production.

Building React apps with Strict Mode enabled can sometimes lead to confusion or annoying error messages. You might, for example, wonder why your component effects are executing too often.

Therefore, it's your personal decision whether you want to use Strict Mode or not. You could also enable it (by wrapping it around your **`<App />`** component) occasionally only.

DEBUGGING CODE AND THE REACT DEVELOPER TOOLS

Earlier in this chapter, you learned that component functions may execute quite frequently and that you can prevent unnecessary executions using **`memo()`** and **`useMemo()`** (and that you shouldn't always prevent it).

Identifying component executions by adding **`console.log()`** inside the component functions is one way of gaining insight into a component. It's the approach used throughout this chapter. However, for large React apps with dozens, hundreds, or even thousands of components, using **`console.log()`** can get tedious.

That's why the React team also built an official tool to help with gaining app insights. The React developer tools are an extension that can be installed into all major browsers (Chrome, Firefox, and Edge). You can find and install the extension by simply searching the web for **`"<your browser> react developer tools"`** (e.g., "chrome react developer tools").

Once you have installed the extension, you can access it directly from inside the browser. For example, when using Chrome, you can access the React developer tools extensions directly from inside Chrome's developer tools (which can be opened via the menu in Chrome). Explore the specific extension documentation (in your browser's extensions store) for details on how to access it.

The React developer tools extension offers two areas: a **`Components`** page and a **`Profile`** page:

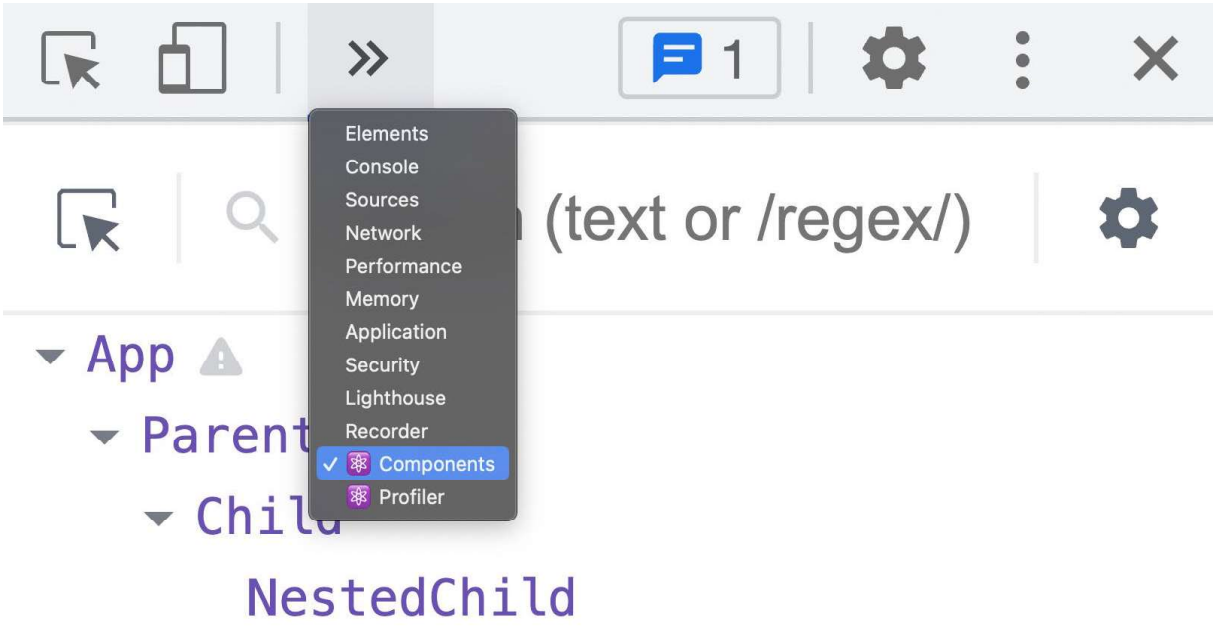

Figure 9.12: The React developer tools can be accessed via browser developer tools

The **`Components`** page can be used to analyze the component structure of the currently rendered page. You can use this page to understand the structure of your components (i.e., the "tree of components"), how components are nested into each other, and even the configuration (props, state) of components.

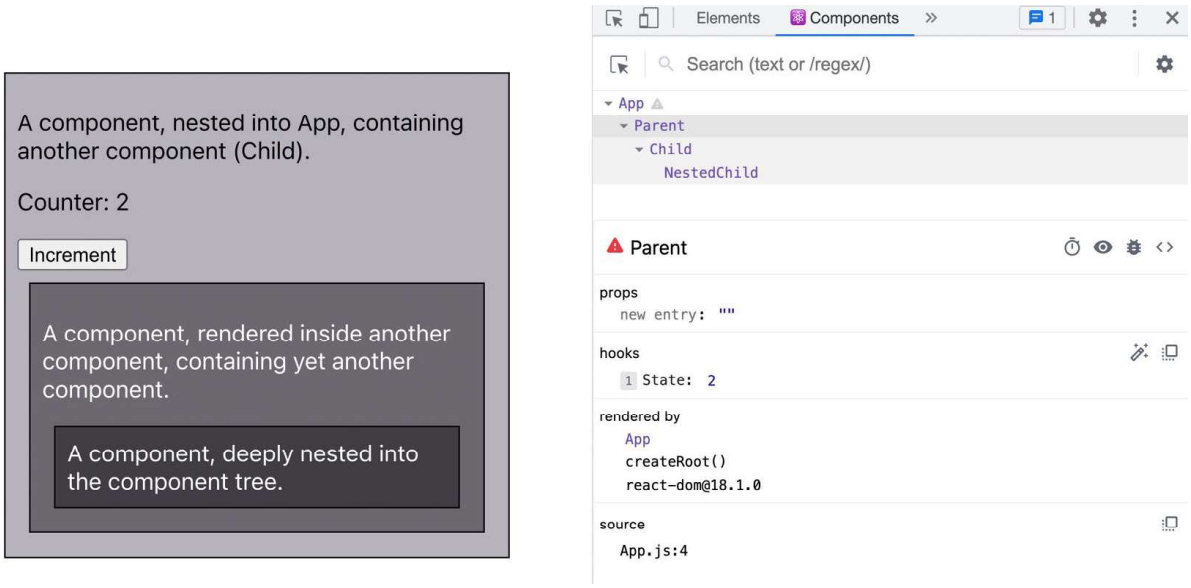

Figure 9.13: Component relations and data are shown

This page can be very useful when attempting to understand the current state of a component, how a component is related to other components, and which other components may therefore influence a component (e.g., cause it to be re-evaluated).

However, in the context of this chapter, the more useful page is the **`Profiler`** page:

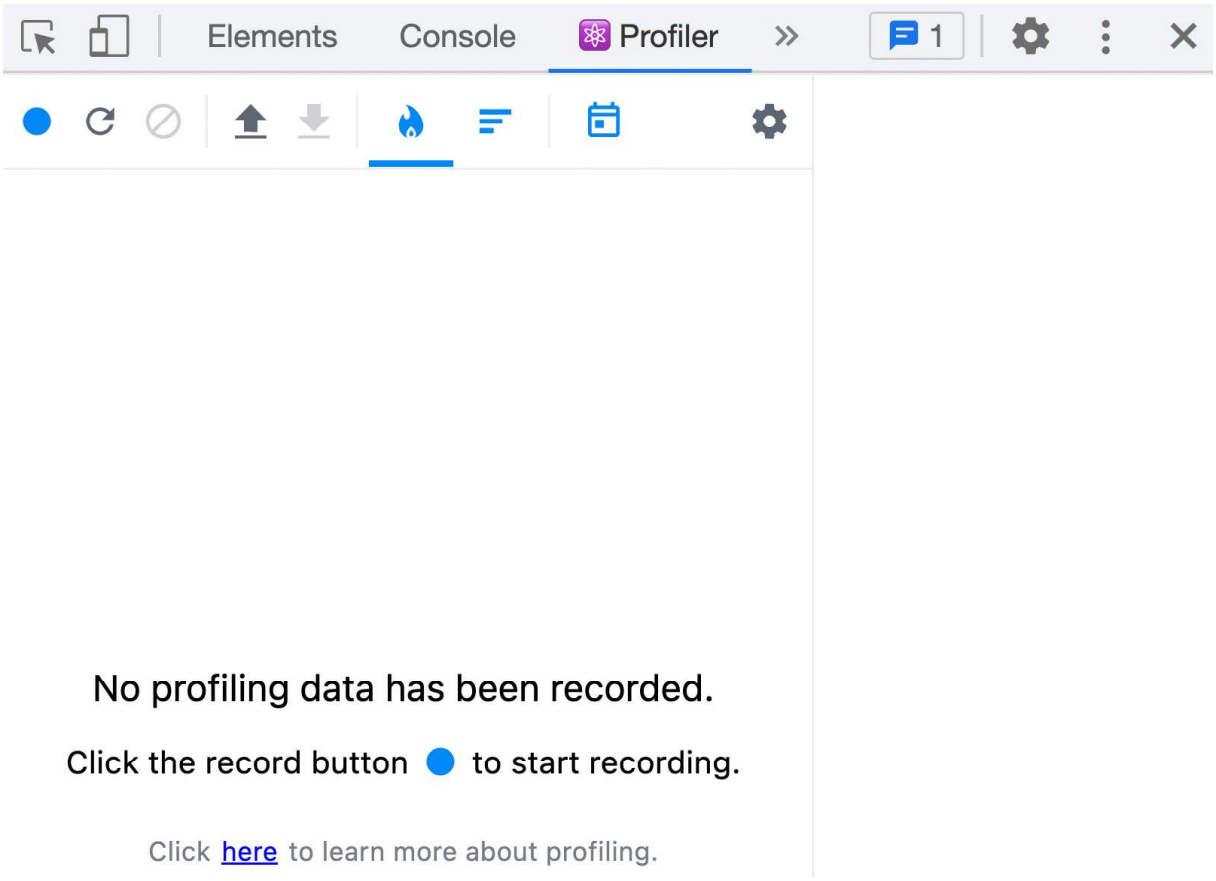

Figure 9.14: The Profiler page (without any data gathered)

On this page, you can begin recording component evaluations (i.e., component function executions). You can do this by simply pressing the **`Record`** button in the top-left corner (the blue circle). This button will then be replaced by a **`Stop`** button, which you can press to end the recording.

After recording the React app for a couple of seconds (and interacting with it during that period), an example result could look like this:

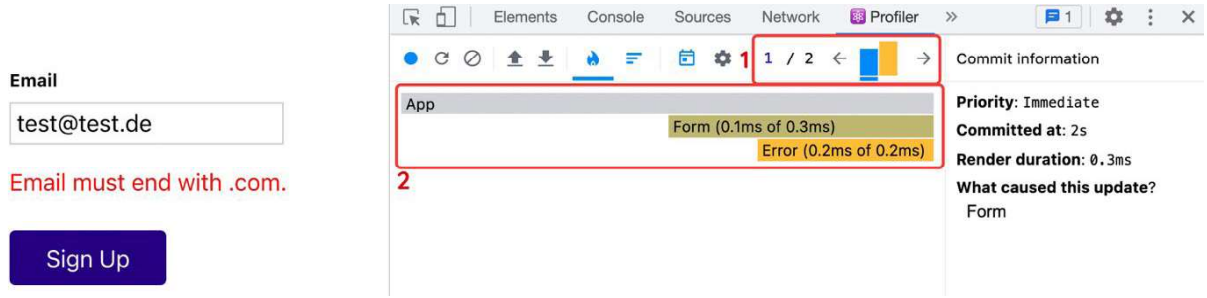

Figure 9.15: The Profiler page shows various bars after recording finished

This result consists of two main areas:

- A list of bars, indicating the number of component re-evaluations (every bar reflects one re-evaluation cycle that affected one or more components). You can click these bars to explore more details about a specific cycle.
- For the selected evaluation cycle, a list of the affected components is presented. You can identify affected components easily as their bars are colored and timing information is displayed for them.

You can select any render cycle from 1 (in this case, there are two for this recording session) to view which components were affected. The bottom part of the window (2) shows all affected components by highlighting them with some color and outputting the overall amount of time taken by this component to be re-evaluated (for example, **`0.1ms of 0.3ms`**).

> **NOTE**
>
> It's worth noting that this tool also proves that component evaluation is extremely fast—**`0.1ms`** for re-evaluating a component is way too fast for any human to realize that something happened behind the scenes.

On the right side of the window, you also learn more about this component evaluation cycle. For example, you learn where it was triggered. In this case, it was triggered by the **`Form`** component (it's the same example as discussed earlier in this chapter, in the *"Avoiding Unnecessary Child Component Evaluations"* section).

The **`Profiler`** page can therefore also help you to identify component evaluation cycles and determine which components are affected. In this example, you can see a difference if the **`memo()`** function is wrapped around the **`Error`** component:

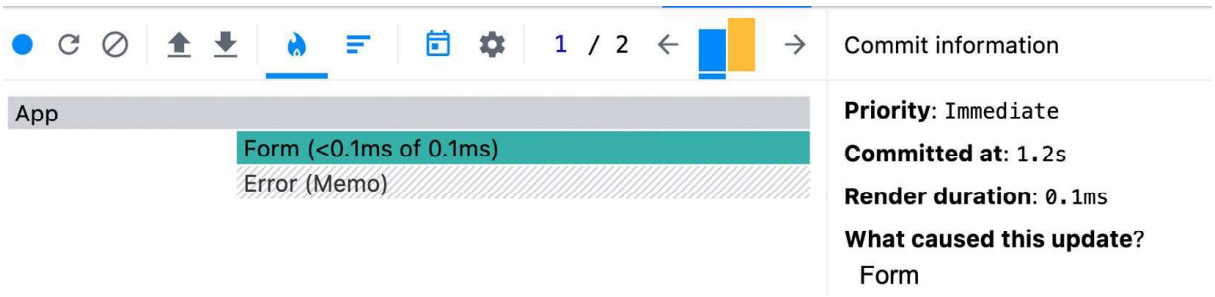

Figure 9.16: Only the Form component is affected, not the Error component

After re-adding the **`memo()`** function as a wrapper around the **`Error`** component (as explained earlier in this chapter), you can use the **`Profiler`** page of the React developer tools to confirm that the **`Error`** component is no longer unnecessarily evaluated. To do this, you should start a new recording session and reproduce the situation, where previously, without **`memo()`**, the **`Error`** component would've been called again.

The diagonal grayed-out lines across the **`Error`** component in the **`Profiler`** window signal that this component was not affected by some other component function invocation.

The React developer tools can therefore be used to gain deeper insights into your React app and your components. You can use them in addition or instead of calling **`console.log()`** in a component function.

SUMMARY AND KEY TAKEAWAYS

- React components are re-evaluated (executed) whenever their state changes or the parent component is evaluated.
- React optimizes component evaluation by calculating required user interface changes with help of a virtual DOM first.
- Multiple state updates that occur at the same time and in the same place are batched together by React. This ensures that unnecessary component evaluations are avoided.
- The **`memo()`** function can be used to control component function executions.
- **`memo()`** looks for prop value differences (old props versus new props) to determine whether a component function must be executed again.
- **`useMemo()`** can be used to wrap performance-intensive computations and only perform them if key dependencies changed.
- Both **`memo()`** and **`useMemo()`** should be used carefully since they also come at a cost (the comparisons performed).
- The initial code download size can be reduced with help of code splitting via the **`lazy()`** function (in conjunction with the built-in **`Suspense`** component)

- React's Strict Mode can be enabled (via the built-in **`<React.StrictMode>`** component) to perform various extra checks and detect potential bugs in your application.
- The React developer tools can be used to gain deeper insights into your React app (for example, component structure and re-evaluation cycles).

WHAT'S NEXT?

As a developer, you should always know and understand the tool you're working with—in this case, React.

This chapter allowed you to get a better idea of how React works under the hood and which optimizations are implemented automatically. In addition, you also learned about various optimization techniques that can be implemented by you.

The next chapter will go back to solving actual problems you might face when trying to build React apps. Instead of optimizing React apps, you will learn more about techniques and features that can be used to solve more complex problems related to component and application state management.

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 at https://packt.link/hPDal:

1. Why does React use a virtual DOM to detect required DOM updates?
2. How is the real DOM affected when a component function is executed?
3. Which components are great candidates for the **`memo()`** function? Which components are bad candidates?
4. How is **`useMemo()`** different from **`memo()`**?
5. What's the idea behind code splitting and the **`lazy()`** function?

APPLY WHAT YOU LEARNED

With your newly gained knowledge about React's internals and some of the optimization techniques you can employ in order to improve your apps, you can now apply this knowledge in the following activity.

ACTIVITY 9.1: OPTIMIZE AN EXISTING APP

In this activity, you're handed an existing React app that can be optimized in various places. Your task is to identify optimization opportunities and implement appropriate solutions. Keep in mind that too much optimization can actually lead to a worse result.

> **NOTE**
>
> You can find the starting code for this activity at https://packt.link/cYqyu. 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.
>
> The provided project also uses many features covered in earlier chapters. Take the time to analyze it and understand the provided code. This is a great practice and allows you to see many key concepts in action.

Once you have downloaded the code and run **`npm install`** in the project folder (to install all required dependencies), you can start the development server via **`npm start`**. As a result, upon visiting **`localhost:3000`**, you should see the following user interface:

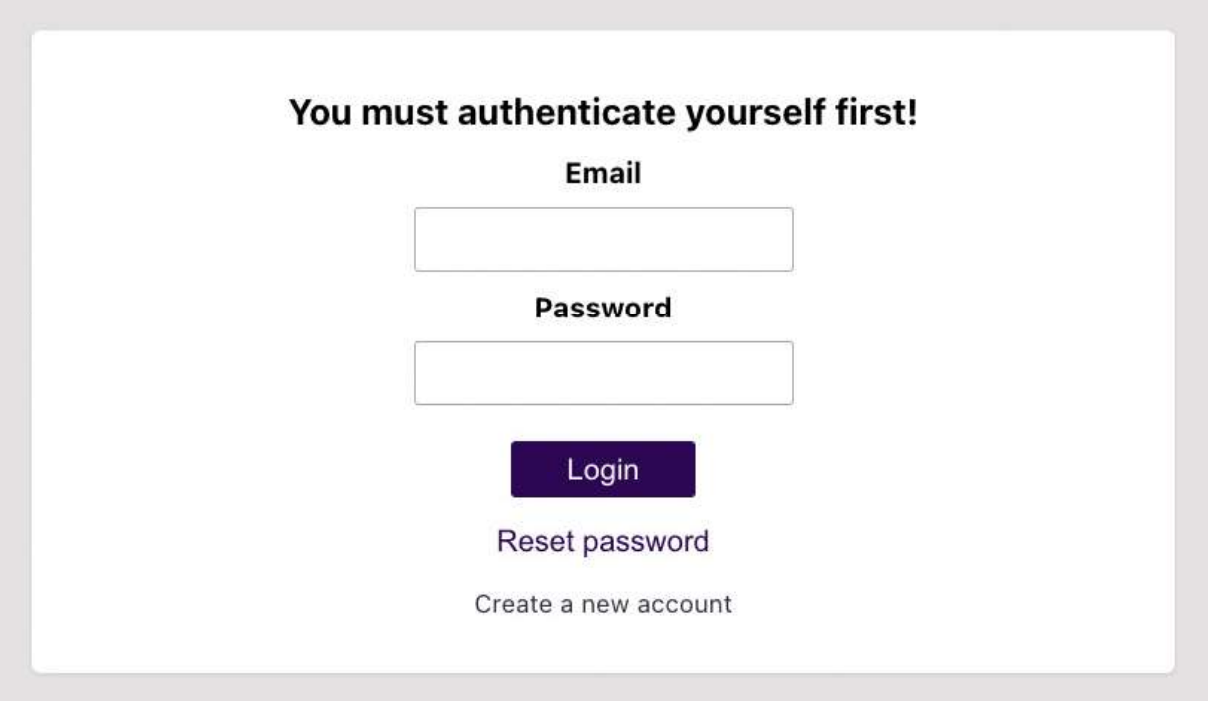

Figure 9.17: The running starting project

Take your time to get acquainted with the provided project. Experiment with the different buttons in the user interface, fill in some dummy data into the form input fields, and analyze the provided code. Please note that this dummy project does not send any HTTP requests to any server. All entered data is discarded the moment it is entered.

To complete the activity, the solution steps are as follows:

1. Find optimization opportunities by looking for unnecessary component function executions.
2. Also identify unnecessary code execution inside of component functions (where the overall component function invocation can't be prevented).
3. Determine which code could be loaded lazily instead of eagerly.
4. Use the **`memo()`** function, the **`useMemo()`** Hook, and React's **`lazy()`** function to improve the code.

You can tell that you came up with a good solution and sensible adjustments if you can see extra code fetching network requests (in the **`Network`** tab of your browser developer tools) for clicking on the **`Reset password`** or **`Create a new account`** buttons:

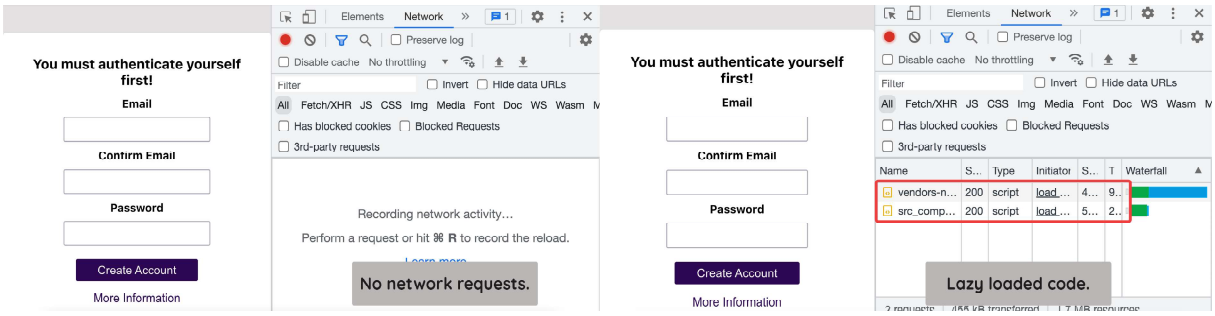

Figure 9.18: In the final solution, some code is lazy loaded

In addition, you should see no **`Validated password.`** console message when typing into the email input fields (**`Email`** **and** **`Confirm Email`**) of the signup form (that is, the form you switch to when clicking **`Create a new account`**):

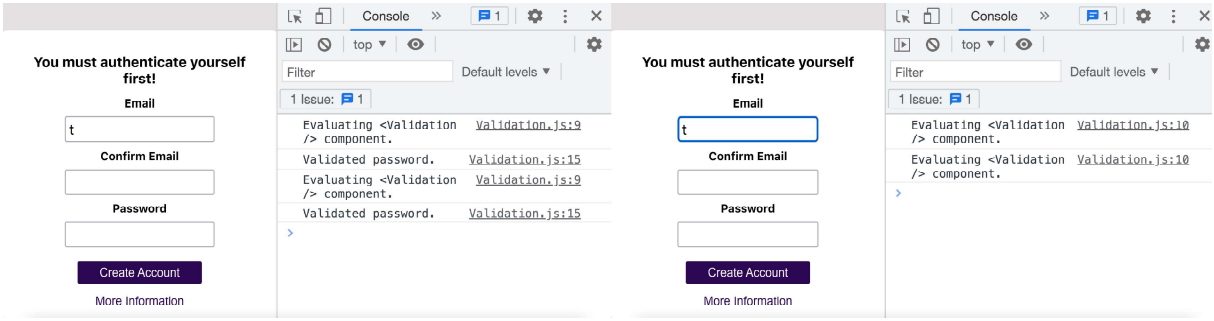

Figure 9.19: No "Validated password." output in the console

You also shouldn't get any console outputs when clicking the **`More Information`** button:

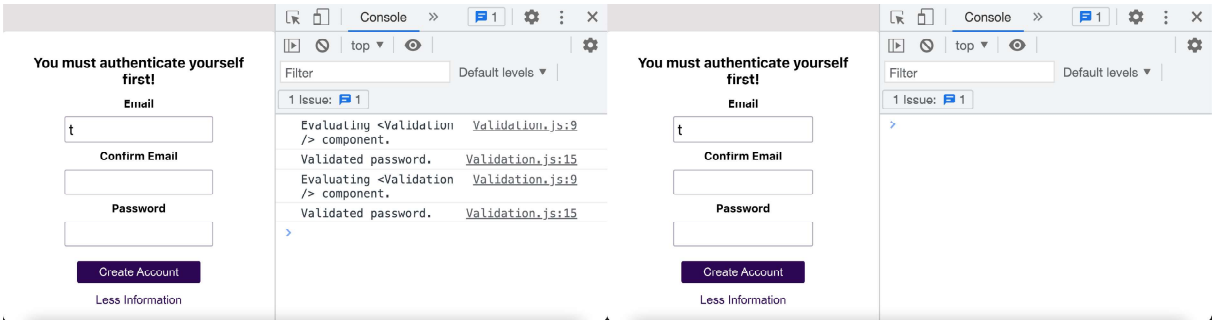

Figure 9.20: No console messages when clicking "More Information"

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

10

WORKING WITH COMPLEX STATE

LEARNING OBJECTIVES

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

- Manage cross-component or even app-wide state (instead of just component-specific state).
- Distribute data across multiple components.
- Handle complex state values and changes.

INTRODUCTION

State is one of the core concepts you must understand (and work with) to use React effectively. Basically, every React app utilizes (many) state values across many components to present a dynamic, reactive user interface.

From simple state values that contain a changing counter or values entered by users, all the way up to more complex state values such as the combination of multiple form inputs or user authentication information, state is everywhere. And in React apps, it's typically managed with the help of the **`useState()`** Hook.

However, once you start building more complex React applications (e.g., online shops, admin dashboards, and similar sites), it is likely that you'll face various challenges related to state. State values might be used in component A but changed in component B or be made up of multiple dynamic values that may change for a broad variety of reasons (e.g., a cart in an online shop, which is a combination of products, where every product has a quantity, a price, and possibly other traits that may be changed individually).

You can handle all these problems with **`useState()`**, props, and the other concepts covered by this book thus far. But you will notice that solutions based on **`useState()`** alone gain a complexity that can be difficult to understand and maintain. That's why React has more tools to offer—tools created for these kinds of problems, which this chapter will highlight and discuss.

A PROBLEM WITH CROSS-COMPONENT STATE

You don't even need to build a highly sophisticated React app to encounter a common problem: state that spans multiple components.

For example, you might be building a news app where users can bookmark certain articles. The user interface could look like this:

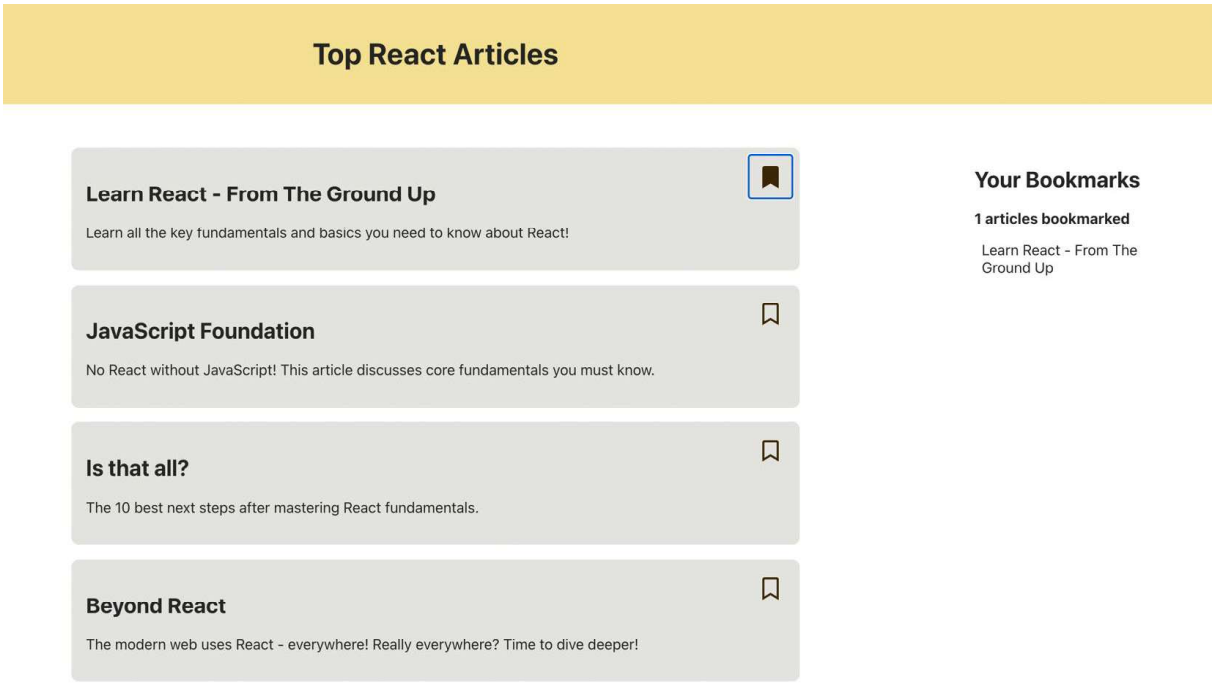

Figure 10.1: An example user interface

As you can see in the preceding figure, the list of articles is on the left, and a summary of the bookmarked articles can be found in a sidebar on the right.

A common solution is to split this user interface into multiple components. The list of articles, specifically, would probably be in its own component—just like the bookmark summary sidebar.

However, in that scenario, both components would need to access the same shared state—that is, the list of bookmarked articles. The article list component would require access in order to add (or remove) articles. The bookmark summary sidebar component would require it as it needs to display the bookmarked articles.

The component tree and state usage for this kind of app could look like this:

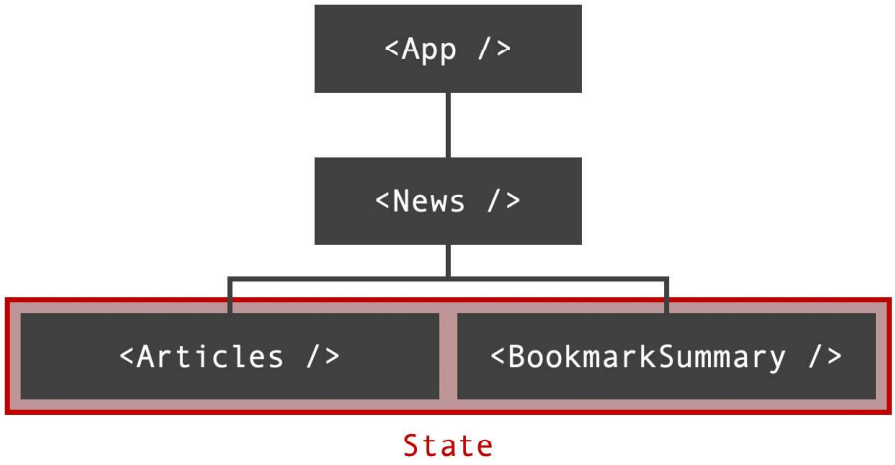

Figure 10.2: Two sibling components share the same state

In this figure, you can see that the state is shared across these two components. You also see that the two components have a shared parent component (the **`News`** component, in this example).

Since the state is used by two components, you would not manage it in either of those components. Instead, it's *lifted up*, as described in *Chapter 4*, *Working with Events and State* (in the "*Lifting State Up*" section). When lifting state up, the state values and pointers to the functions that manipulate the state values are passed down to the actual components that need access via props.

This works and is a common pattern. You can (and should) keep on using it. But what if a component that needs access to some shared state is deeply nested in other components? What if the app component tree from the preceding example looked like this?

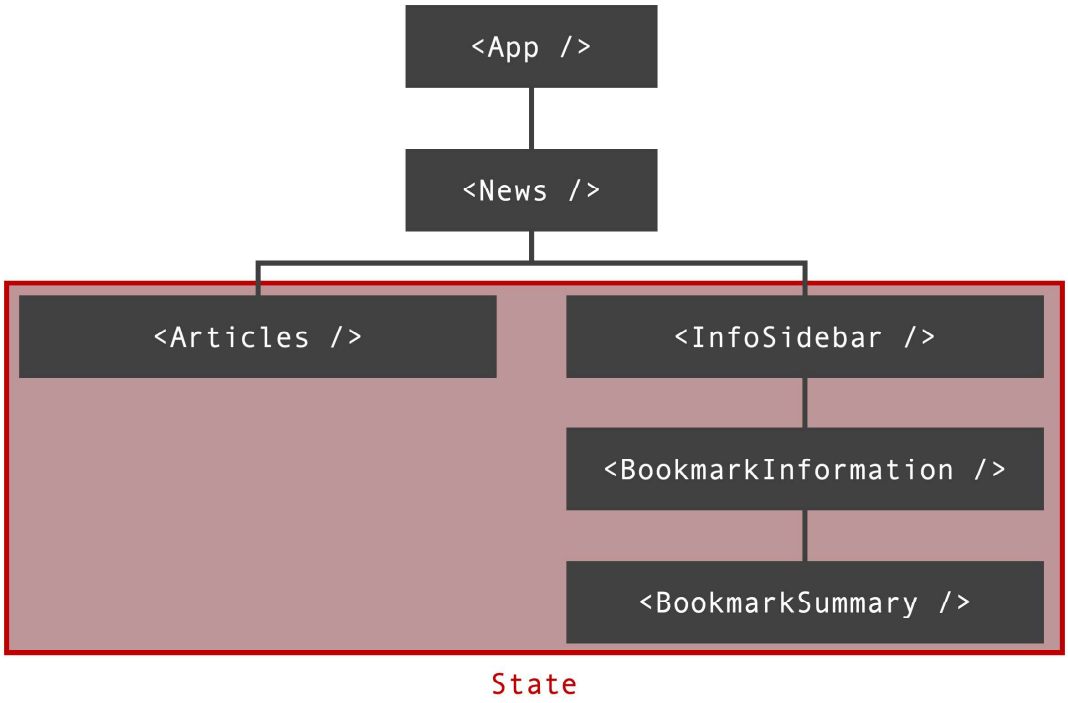

Figure 10.3: A component tree with multiple layers of state-dependent components

In this figure, you can see that the **`BookmarkSummary`** component is a deeply nested component. Between it and the **`News`** component (which manages the shared state), you have two other components: the **`InfoSidebar`** component and the **`BookmarkInformation`** component. In more complex React apps, having multiple levels of component nesting, as in this example, is very common.

Of course, even with those extra components, state values can still be passed down via props. You just need to add props to **all** components between the component that holds the state and the component that needs the state. For example, you must pass the **`bookmarkedArticles`** state value to the **`InfoSidebar`** component (via props) so that that component can forward it to **`BookmarkInformation`**:

```
import BookmarkInformation from '../BookmarkSummary/BookmarkInformation';
import classes from './InfoSidebar.module.css';

function InfoSidebar({ bookmarkedArticles }) {
  return (
    <aside className={classes.sidebar}>
```

```
      <BookmarkInformation bookmarkedArticles={bookmarkedArticles} />
    </aside>
  );
}

export default InfoSidebar;
```

The same procedure is repeated inside of the **`BookmarkInformation`** component.

> **NOTE**
>
> You can find the complete example on GitHub at https://packt.link/Ft8AM.

This kind of pattern is called **prop drilling**. Prop drilling means that a state value is passed through multiple components via props. And it's passed through components that don't need the state themselves at all—except for forwarding it to a child component (as the **`InfoSidebar`** and **`BookmarkInformation`** components are doing in the preceding example).

As a developer, you will typically want to avoid this pattern because prop drilling has a few weaknesses:

- Components that are part of prop drilling (such as **`InfoSidebar`** or **`BookmarkInformation`**) are not really reusable anymore because any component that wants to use them has to provide a value for the forwarded state prop.
- Prop drilling also leads to a lot of overhead code that has to be written (the code to accept props and forward props).
- Refactoring components becomes more work because state props have to be added or removed.

For these reasons, prop drilling is only acceptable if all components involved are only used in this specific part of the overall React app, and the probability of reusing or refactoring them is low.

Since prop drilling should be avoided (in most situations), React offers an alternative: the **context** API.

USING CONTEXT TO HANDLE MULTI-COMPONENT STATE

React's context feature is one that allows you to create a value that can easily be shared across as many components as needed, without using props.

Using the context API is a multi-step process, the steps for which are described here:

1. You must create a context value that should be shared.
2. The context must be provided in a parent component of the components that need access to the context object.
3. Components that need access (for reading or writing) must subscribe to the context.

React manages the context value (and its changes) internally and automatically distributes it to all components that have subscribed to the context.

Before any component may subscribe, however, the first step is to create a context object. This is done via React's **`createContext()`** function:

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

createContext('Hello Context'); // a context with an initial string value
createContext({}); // a context with an initial (empty) object as a value
```

This function takes an initial value that should be shared. It can be any kind of value (e.g., a string or a number), but typically, it's an object. This is because most shared values are a combination of the actual values and functions that should manipulate those values. All these things are then grouped together into a single context object.

Of course, the initial context value can also be an empty value (e.g., **`null`**, **`undefined`**, an empty string, etc.) if needed.

`createContext()` also returns a value: a context object that should be stored in a capitalized variable (or constant) because it contains a nested property that is a React component (and React components should start with capital characters).

Here's how the **`createContext()`** function can be used for the example discussed earlier in this chapter:

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

const BookmarkContext = createContext({
  bookmarkedArticles: []
});

export default BookmarkContext; // making it available outside of its file
```

Here, the initial value is an object that contains the **`bookmarkedArticles`** property, which holds an (empty) array. You could also store just the array as an initial value (i.e., **`createContext([])`**) but an object is better since more will be added to it later in the chapter.

This code is typically placed in a separate context code file that's often stored in a folder named **`store`** (because this context feature can be used as a central state store) or **`context`**. However, this is just a convention and is not technically required. You can put this code anywhere in your React app.

Of course, this initial value is not a replacement for state; it's a static value that never changes. But this was just the first of three steps related to context. The next step is to provide the context.

PROVIDING AND MANAGING CONTEXT VALUES

In order to use context values in other components, you must first provide the value. This is done using the value returned by **`createContext()`**. That function yields an object that contains a nested **`Provider`** property. And that property contains a React component that should be wrapped around all other components that need access to the context value.

In the preceding example, the **`BookmarkContext.Provider`** component could be used in the **`News`** component to wrap it around both the **`Articles`** and **`InfoSidebar`** components:

```
import Articles from '../Articles/Articles';
import InfoSidebar from '../InfoSidebar/InfoSidebar';
import BookmarkContext from '../../store/bookmark-context';

function News() {
  return (
```

```
    <BookmarkContext.Provider>
      <Articles />
      <InfoSidebar />
    </BookmarkContext.Provider>
  );
}
```

However, this code does not work because one important thing is missing: the **`Provider`** component expects a **`value`** prop, which should contain the current context value that should be distributed to interested components. While you do provide an initial context value (which could have been empty), you also need to inform React about the current context value because, very often, context values change (they are often used as a replacement for the cross-component state, after all).

Hence, the code could be altered like this:

```
import Articles from '../Articles/Articles';
import InfoSidebar from '../InfoSidebar/InfoSidebar';
import BookmarkContext from '../../store/bookmark-context';

function News() {
  const bookmarkCtxValue = {
    bookmarkedArticles: []
  }; // for now, it's the same value as used before, for the initial
context

  return (
    <BookmarkContext.Provider value={bookmarkCtxValue}>
      <Articles />
      <InfoSidebar />
    </BookmarkContext.Provider>
  );
}
```

With this code, an object with a list of bookmarked articles is distributed to interested descendent components.

The list is still static though. But that can be changed with a tool you already know about: the **`useState()`** Hook. Inside the **`News`** component, you can use the **`useState()`** Hook to manage the list of bookmarked articles, like this:

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

import Articles from '../Articles/Articles';
import InfoSidebar from '../InfoSidebar/InfoSidebar';
import BookmarkContext from '../../store/bookmark-context';

function News() {
  const [savedArticles, setSavedArticles] = useState([]);

  const bookmarkCtxValue = {
    bookmarkedArticles: savedArticles // using the state as a value now!
  };

  return (
    <BookmarkContext.Provider value={bookmarkCtxValue}>
      <Articles />
      <InfoSidebar />
    </BookmarkContext.Provider>
  );
}
```

With this change, the context changes from static to dynamic. Whenever the **`savedArticles`** state changes, the context value will change.

Therefore, that's the missing piece when it comes to providing the context. If the context should be dynamic (and changeable from inside some nested child component), the context value should also include a pointer to the function that triggers a state update.

For the preceding example, the code is therefore adjusted like this:

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

import Articles from '../Articles/Articles';
import InfoSidebar from '../InfoSidebar/InfoSidebar';
import BookmarkContext from '../../store/bookmark-context';

function News() {
```

```
  const [savedArticles, setSavedArticles] = useState([]);

  function addArticle(article) {
    setSavedArticles((prevSavedArticles) => [...prevSavedArticles,
article]);
  }

  function removeArticle(articleId) {
    setSavedArticles((prevSavedArticles) => prevSavedArticles.filter(
      (article) => article.id !== articleId)
    );
  }

  const bookmarkCtxValue = {
    bookmarkedArticles: savedArticles,
    bookmarkArticle: addArticle,
    unbookmarkArticle: removeArticle
  };

  return (
    <BookmarkContext.Provider value={bookmarkCtxValue}>
      <Articles />
      <InfoSidebar />
    </BookmarkContext.Provider>
  );
}
```

The following are two important things changed in this code snippet:

- Two new functions were added: **`addArticle`** and **`removeArticle`**.
- Properties that point at these functions were added to **`bookmarkCtxValue`**: the **`bookmarkArticle`** and **`unbookmarkArticle`** methods.

The **`addArticle`** function adds a new article (which should be bookmarked) to the **`savedArticles`** state. The function form of updating the state value is used since the new state value depends on the previous state value (the bookmarked article is added to the list of already bookmarked articles).

Similarly, the **`removeArticle`** function removes an article from the **`savedArticles`** list by filtering the existing list such that all items, except for the one that has a matching **`id`** value, are kept.

If the **`News`** component did not use the new context feature, it would be a component that uses state just as you saw many times before in this book. But now, by using React's context API, those existing capabilities are combined with a new feature (the context) to create a dynamic, distributable value.

Any components nested in the **`Articles`** or **`InfoSidebar`** components (or their descendent components) will be able to access this dynamic context value, and the **`bookmarkArticle`** and **`unbookmarkArticle`** methods in the context object, without any prop drilling.

> **NOTE**
>
> You don't have to create dynamic context values. You could also distribute a static value to nested components. This is possible but a rare scenario, since most React apps do typically need dynamic state values that can change across components.

USING CONTEXT IN NESTED COMPONENTS

With the context created and provided, it's ready to be used by components that need to access or change the context value.

To make the context value accessible by components nested inside the context's **`Provider`** component (**`BookmarkContext.Provider`**, in the preceding example), React offers a **`useContext()`** Hook that can be used.

`useContext()` requires one argument: the context object that was created via **`createContext()`**, i.e., the value returned by that function. **`useContext()`** then returns the value passed to the **`Provider`** component (via its **`value`** prop).

For the preceding example, the context value can be used in the **`BookmarkSummary`** component like this:

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

import BookmarkContext from '../../store/bookmark-context';
import classes from './BookmarkSummary.module.css';

function BookmarkSummary() {
  const bookmarkCtx = useContext(BookmarkContext);
```

```
  const numberOfArticles = bookmarkCtx.bookmarkedArticles.length;

  return (
    <>
      <p className={classes.summary}>{numberOfArticles} articles
bookmarked</p>
      <ul className={classes.list}>
        {bookmarkCtx.bookmarkedArticles.map((article) => (
          <li key={article.id}>{article.title}</li>
        ))}
      </ul>
    </>
  );
}

export default BookmarkSummary;
```

In this code, **`useContext()`** receives the **`BookmarkContext`** value, which is imported from the **`store/bookmark-context.js`** file. It then returns the value stored in the context, which is the **`bookmarkCtxValue`** found in the previous code example. As you can see in that snippet, **`bookmarkCtxValue`** is an object with three properties: **`bookmarkedArticles`**, **`bookmarkArticle`** (a method), and **`unbookmarkArticle`** (also a method).

This returned object is stored in a **`bookmarkCtx`** constant. Whenever the context value changes (because the **`setSavedArticles`** state-updating function in the **`News`** component is executed), this **`BookmarkSummary`** component will also be executed again by React, and thus **`bookmarkCtx`** will hold the latest state value.

Finally, in the **`BookmarkSummary`** component, the **`bookmarkedArticles`** property is accessed on the **`bookmarkCtx`** object. This list of articles is then used to calculate the number of bookmarked articles, output a short summary, and display the list on the screen.

Similarly, **`BookmarkContext`** can be used via **`useContext()`** in the **`Articles`** component:

```
import { useContext } from 'react';
import { FaBookmark, FaRegBookmark } from 'react-icons/fa';

import dummyArticles from '../../data/dummy-articles';
import BookmarkContext from '../../store/bookmark-context';
```

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

function Articles() {
  const bookmarkCtx = useContext(BookmarkContext);

  return (
    <ul className={classes.list}>
      {dummyArticles.map((article) => {
        // will be true if this article item is also
        // included in the bookmarkedArticles array
        const isBookmarked = bookmarkCtx.bookmarkedArticles.some(
          (bArticle) => bArticle.id === article.id
        );

        let buttonAction = () => {}; // dummy value, will be finished
later!
        // default button icon: Empty bookmark icon, because not
bookmarked
        let buttonIcon = <FaRegBookmark />;

        if (isBookmarked) {
          buttonAction = () => {}; // dummy value, will be finished later!
          buttonIcon = <FaBookmark />;
        }

        return (
          <li key={article.id}>
            <h2>{article.title}</h2>
            <p>{article.description}</p>
            <button onClick={buttonAction}>{buttonIcon}</button>
          </li>
        );
      })}
    </ul>
  );
}
export default Articles;
```

In this component, the context is used to determine whether or not a given article is currently bookmarked (this information is required in order to change the icon and functionality of the button).

That's how context values (whether static or dynamic) can be read in components. Of course, they can also be changed, as discussed in the next section.

CHANGING CONTEXT FROM NESTED COMPONENTS

React's context feature is often used to share data across multiple components without using props. It's therefore also quite common that some components must manipulate that data. For example, the context value for a shopping cart must be adjustable from inside the component that displays product items (because those probably have an **`"Add to cart"`** button).

However, to change context values from inside a nested component, you cannot simply overwrite the stored context value. The following code would not work as intended:

```
const bookmarkCtx = useContext(BookmarkContext);

// Note: This does NOT work
bookmarkCtx.bookmarkedArticles = []; // setting the articles to an empty
array
```

This code does not work. Just as you should not try to update state by simply assigning a new value, you can't update context values by assigning a new value. That's why two methods (**`bookmarkArticle`** and **`unbookmarkArticle`**) were added to the context value in the "*Providing and Managing Context Values*" section. These two methods point at functions that trigger state updates (via the state-updating function provided by **`useState()`**).

Therefore, in the **`Articles`** component, where articles can be bookmarked or unbookmarked via button clicks, these methods should be called:

```
// This code is part of the Article component function
// default button action => bookmark article, because not bookmarked yet
let buttonAction = () => bookmarkCtx.bookmarkArticle(article);
// default button icon: Empty bookmark icon, because not bookmarked
let buttonIcon = <FaRegBookmark />;

if (isBookmarked) {
  buttonAction = () => bookmarkCtx.unbookmarkArticle(article.id);
  buttonIcon = <FaBookmark />;
}
```

The **`bookmarkArticle`** and **`unbookmarkArticle`** methods are called inside of anonymous functions that are stored in a **`buttonAction`** variable. That variable is assigned to the **`onClick`** prop of the **`<button>`** (see the previous code snippet).

With this code, the context value can be changed successfully. Thanks to the steps taken in the previous section (*"Using Context in Nested Components"*), whenever the context value is updated, it is then also automatically reflected in the user interface.

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

GETTING BETTER CODE COMPLETION

In the section *"Using Context to Handle Multi-Component State"*, a context object was created via **`createContext()`**. That function received an initial context value—an object that contains a **`bookmarkedArticles`** property, in the preceding example.

In this example, the initial context value isn't too important. It's not often used because it's overwritten with a new value inside the **`News`** component regardless. However, depending on which **Integrated Development Environment (IDE**) you're using, you can get better code auto-completion when defining an initial context value that has the same shape and structure as the final context value that will be managed in other React components.

Therefore, since two methods were added to the context value in the section *"Providing and Managing Context Values"*, those methods should also be added to the initial context value in **`store/bookmark-context.js`**:

```
const BookmarkContext = createContext({
  bookmarkedArticles: [],
  bookmarkArticle: () => {},
  unbookmarkArticle: () => {}
});

export default BookmarkContext;
```

The two methods are added as empty functions that do nothing because the actual logic is set in the **`News`** component. The methods are only added to this initial context value to provide better IDE auto-completion. This step is therefore optional.

CONTEXT OR "LIFTING STATE UP"?

At this point, you now have two tools for managing cross-component state:

- You can lift state up, as described earlier in the book (in *Chapter 4*, *Working with Events and State*, in section "*Lifting State Up*").
- Alternatively, you can use React's context API as explained in this chapter.

Which of the two approaches should you use in each scenario?

Ultimately, it is up to you how you manage this, but there are some straightforward rules you can follow:

- Lift the state up if you only need to share state across one or two levels of component nesting.
- Use the context API if you have long chains of components (i.e., deep nesting of components) with shared state. Once you start to use a lot of prop drilling, it's time to consider React's context feature.
- Also use the context API if you have a relatively flat component tree but want to reuse components (i.e., you don't want to use props for passing state to components).

OUTSOURCING CONTEXT LOGIC INTO SEPARATE COMPONENTS

With the previously explained steps, you have everything you need to manage cross-component state via context.

But there is one pattern you can consider for managing your dynamic context value and state: creating a separate component for providing (and managing) the context value.

In the preceding example, the **`News`** component was used to provide the context and manage its (dynamic, state-based) value. While this works, your components can get unnecessarily complex if they have to deal with context management. Creating a separate, dedicated component for that can therefore lead to code that's easier to understand and maintain.

For the preceding example, that means that, inside of the **`store/bookmark-context.js`** file, you could create a **`BookmarkContextProvider`** component that looks like this:

```
export function BookmarkContextProvider({ children }) {
  const [savedArticles, setSavedArticles] = useState([]);

  function addArticle(article) {
    setSavedArticles((prevSavedArticles) => [...prevSavedArticles,
article]);
  }

  function removeArticle(articleId) {
    setSavedArticles((prevSavedArticles) =>
      prevSavedArticles.filter((article) => article.id !== articleId)
    );
  }

  const bookmarkCtxValue = {
    bookmarkedArticles: savedArticles,
    bookmarkArticle: addArticle,
    unbookmarkArticle: removeArticle,
  };

  return (
    <BookmarkContext.Provider value={bookmarkCtxValue}>
      {children}
    </BookmarkContext.Provider>
  );
}
```

This component contains all the logic related to managing a list of bookmarked articles via state. It creates the same context value as before (a value that contains the list of articles as well as two methods for updating that list).

The **`BookmarkContextProvider`** component does one additional thing though. It uses the special **`children`** prop (covered in *Chapter 3*, *Components and Props*, in section *"The Special 'children' Prop"*) to wrap whatever is passed between the **`BookmarkContextProvider`**'s component tags with **`BookmarkContext.Provider`**.

This allows for the use of the **`BookmarkContextProvider`** component in the **`News`** component, like so:

```
import Articles from '../Articles/Articles';
import InfoSidebar from '../InfoSidebar/InfoSidebar';
import { BookmarkContextProvider } from '../../store/bookmark-context';

function News() {
  return (
    <BookmarkContextProvider>
      <Articles />
      <InfoSidebar />
    </BookmarkContextProvider>
  );
}

export default News;
```

Instead of managing the entire context value, the **`News`** component now simply imports the **`BookmarkContextProvider`** component and wraps that component around **`Articles`** and **`BookmarkSummary`**. The **`News`** component, therefore, is leaner.

> **NOTE**
>
> This pattern is entirely optional. It's neither an official best practice nor does it yield any performance benefits.

COMBINING MULTIPLE CONTEXTS

Especially in bigger and more feature-rich React applications, it is possible (and quite probable), that you will need to work with multiple context values that are likely unrelated to each other. For example, an online shop could use one context for managing the shopping cart, another context for the user authentication status, and yet another context value for tracking page analytics.

React fully supports use cases like this. You can create, manage, provide, and use as many context values as needed. You can manage multiple (related or unrelated) values in a single context or use multiple contexts. You can provide multiple contexts in the same component or in different components. It is totally up to you and your app's requirements.

You can also use multiple contexts in the same component (meaning that you can call **`useContext()`** multiple times, with different context values).

LIMITATIONS OF USESTATE()

Thus far in this chapter, the complexity of cross-component state has been explored. But state management can also get challenging in scenarios where some state is only used inside a single component.

`useState()` is a great tool for state management in most scenarios (of course, right now, it's also the only tool that's been covered). Therefore, **`useState()`** should be your default choice for managing state. But **`useState()`** can reach its limits if you need to derive a new state value that's based on the value of another state variable, as in this example:

```
setIsLoading(fetchedPosts ? false : true);
```

This short snippet is taken from a component where an HTTP request is sent to fetch some blog posts.

> **NOTE**
>
> You'll find the complete example code on GitHub at https://packt.link/FiOCM. You will also see more excerpts from the code later in this chapter.

When initiating the request, an **`isLoading`** state value (responsible for showing a loading indicator on the screen) should be set to **`true`** only if no data was fetched before. If data was fetched before (i.e., **`fetchedPosts`** is not **`null`**), that data should still be shown on the screen, instead of some loading indicator.

At first sight, this code might not look problematic. But it actually violates an important rule related to **`useState()`**: you should not reference the current state for setting a new state value. If you need to do so, you should instead use the function form of the state updating function (see *Chapter 4*, *Working with Events and State*, section "*Updating State Based on Previous State Correctly*").

However, in the preceding example, this solution won't work. If you switch to the functional state-updating form, you only get access to the current value of the state you're trying to update. You don't get (safe) access to the current value of some other state. And in the preceding example, another state (**`fetchedPosts`** instead of **`isLoading`**) is referenced. Therefore, you must violate the mentioned rule.

This violation also has real consequences (in this example). The following code snippet is part of a function called **`fetchPosts`**, which is wrapped with **`useCallback()`**:

```
const fetchPosts = useCallback(async function fetchPosts() {
  setIsLoading(fetchedPosts ? false : true);
  setError(null);

  try {
    const response = await fetch(
      'https://jsonplaceholder.typicode.com/posts'
    );

    if (!response.ok) {
      throw new Error('Failed to fetch posts.');
    }

    const posts = await response.json();

    setIsLoading(false);
    setError(null);
    setFetchedPosts(posts);
  } catch (error) {
    setIsLoading(false);
    setError(error.message);
    setFetchedPosts(null);
  }
}, []);
```

This function sends an HTTP request and changes multiple state values based on the state of the request.

`useCallback()` is used to avoid an infinite loop related to **`useEffect()`** (see *Chapter 8*, *Handling Side Effects* to learn more about **`useEffect()`**, infinite loops, and **`useCallback()`** as a remedy). Normally, **`fetchedPosts`** should be added as a dependency to the **`dependencies`** array passed as a second argument to the **`useCallback()`** function. However, in this example, this can't be done because **`fetchedPosts`** is changed inside the function wrapped by **`useCallback()`**, and the state value is therefore not just a dependency but also actively changed. This causes an infinite loop.

As a result, a warning is shown in the terminal and the intended behavior of not showing the loading indicator if data was fetched before is not achieved:

```
WARNING in [eslint]
src/App.js
  Line 34:6:  React Hook useCallback has a missing dependency: 'fetchedPosts'. Either include it or remove the dependency array. You can
 also replace multiple useState variables with useReducer if 'setIsLoading' needs the current value of 'fetchedPosts'  react-hooks/exhau
stive-deps
```

Figure 10.4: A warning about the missing dependency is output in the terminal

Problems like the one just described are common if you have multiple related state values that depend on each other.

One possible solution would be to move from multiple, individual state slices (**`fetchedPosts`**, **`isLoading`**, and **`error`**) to a single, combined state value (i.e., to an object). That would ensure that all state values are grouped together and can thereby be accessed safely when using the functional state-updating form. The state-updating code then could look like this:

```
setHttpState(prevState => ({
  fetchedPosts: prevState.fetchedPosts,
  isLoading: prevState.fetchedPosts ? false : true,
  error: null
}));
```

This solution would work. However, ending up with ever more complex (and nested) state objects is not typically desirable as it can make state management a bit harder and bloat your component code.

That's why React offers an alternative to **`useState()`**: the **`useReducer()`** Hook.

MANAGING STATE WITH USEREDUCER()

Just like **`useState()`**, **`useReducer()`** is a React Hook. And just like **`useState()`**, it is a Hook that can trigger component function re-evaluations. But, of course, it works slightly differently; otherwise, it would be a redundant Hook.

`useReducer()` is a Hook meant to be used for managing complex state objects. You will rarely (probably never) use it to manage simple string or number values.

This Hook takes two main arguments:

- A reducer function
- An initial state value

This brings up an important question: what is a reducer function?

UNDERSTANDING REDUCER FUNCTIONS

In the context of **`useReducer()`**, a reducer function is a function that itself receives two parameters:

- The current state value
- An action that was dispatched

Besides receiving arguments, a reducer function must also return a value: the new state. It's called a reducer function because it reduces the old state (combined with an action) to a new state.

To make this all a bit easier to grasp and reason through, the following code snippet shows how **`useReducer()`** is used in conjunction with such a reducer function:

```
const initialHttpState = {
  data: null,
  isLoading: false,
  error: null,
};

function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state, // copying the existing state
      isLoading: state.data ? false : true,
      error: null,
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
      isLoading: false,
```

```
      error: null,
    };
  }

  return initialHttpState; // default value for unknown actions
}

function App() {
  useReducer(httpReducer, initialHttpState);

  // more component code, not relevant for this snippet / explanation
}
```

At the bottom of this snippet, you can see that **`useReducer()`** is called inside of the **`App`** component function. Like all React Hooks, it must be called inside of component functions or other Hooks. You can also see the two arguments that were mentioned previously (the reducer function and initial state value) being passed to **`useReducer()`**.

`httpReducer` is the reducer function. The function takes two arguments (**`state`**, which is the old state, and **`action`**, which is the dispatched action) and returns different state objects for different action types.

This reducer function takes care of all possible state updates. The entire state-updating logic is therefore outsourced from the component (note that **`httpReducer`** is defined outside of the component function).

But the component function must, of course, be able to trigger the defined state updates. That's where actions become important.

> **NOTE**
>
> In this example, the reducer function is created outside of the component function. You could also create it inside the component function, but that is not recommended. If you create the reducer function inside the component function, it will technically be recreated every time the component function is executed. This impacts performance unnecessarily since the reducer function does not need access to any component function values.

DISPATCHING ACTIONS

The code shown previously is incomplete. When calling **`useReducer()`** in a component function, it does not just take two arguments. Instead, the Hook also returns a value—an array with exactly two elements (just like **`useState()`**, though the elements are different).

`useReducer()` should therefore be used like this (in the **`App`** component):

```
const [httpState, dispatch] = useReducer(httpReducer, initialHttpState);
```

In this snippet, array destructuring is used to store the two elements (and it is always exactly two!) in two different constants: **`httpState`** and **`dispatch`**.

The first element in the returned array (**`httpState`**, in this case) is the state value returned by the reducer function. It's updated (meaning that the component function is called by React) whenever the reducer function is executed again. The element is called **`httpState`** in this example because it contains the state value, which is related to an HTTP request in this instance. That said, how you name the element in your own case is up to you.

The second element (**`dispatch`**, in the example) is a function. It's a function that can be called to trigger a state update (i.e., to execute the reducer function again). When executed, the **`dispatch`** function must receive one argument—that is, the action value that will be available inside of the reducer function (via the reducer function's second argument). Here's how **`dispatch`** can be used in a component:

```
dispatch({ type: 'FETCH_START' });
```

The element is called **`dispatch`** in the example because it's a function used for dispatching actions to the reducer function. Just as before, the name is up to you, but **`dispatch`** is a commonly chosen name.

The shape and structure of that action value are also entirely up to you, but it's often set to an object that contains a **`type`** property. The **`type`** property is used in the reducer function to perform different actions for different types of actions. **`type`** therefore acts as an action identifier. You can see the **`type`** property being used inside the **`httpReducer`** function:

```
function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state, // copying the existing state
      isLoading: state.data ? false : true,
      error: null,
```

```
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
      isLoading: false,
      error: null,
    };
  }

  return initialHttpState; // default value for unknown actions
}
```

You can add as many properties to the action object as needed. In the preceding example, some state updates access **`action.payload`** to extract some extra data from the action object. Inside a component, you would pass data along with the action like this:

```
dispatch({ type: 'FETCH_SUCCESS', payload: posts });
```

Again, the property name (**`payload`**) is up to you, but passing extra data along with the action allows you to perform state updates that rely on data generated by the component function.

Here's the complete, final code for the entire **`App`** component function:

```
// code for httpReducer etc. did not change


function App() {
  const [httpState, dispatch] = useReducer(httpReducer,
initialHttpState);

  // Using useCallback() to prevent an infinite loop in useEffect() below
  const fetchPosts = useCallback(async function fetchPosts() {
```

```
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
        <button onClick={fetchPosts}>Load Posts</button>
      </header>
      {httpState.isLoading && <p>Loading...</p>}
      {httpState.error && <p>{httpState.error}</p>}
      {httpState.data && <BlogPosts posts={httpState.data} />}
    </>
  );
}
```

In this code snippet, you can see how different actions (with different **`type`** and sometimes **`payload`** properties) are dispatched. You can also see that the **`httpState`** value is used to show different user interface elements based on the state (e.g., **`<p>Loading`**...**`</p>`** is shown if **`httpState.isLoading`** is **`true`**).

SUMMARY AND KEY TAKEAWAYS

- State management can have its challenges—especially when dealing with cross-component (or app-wide) state or complex state values.
- Cross-component state can be managed by lifting state up or by using React's Context API.
- The Context API is typically preferable if you do a lot of prop drilling (forwarding state values via props across multiple component layers).
- When using the context API, you use **`createContext()`** to create a new context object.
- The created context object yields a **`Provider`** component that must be wrapped around the part of the component tree that should get access to the context.
- Components can access the context value via the **`useContext()`** Hook.
- For managing complex state values, **`useReducer()`** can be a good alternative to **`useState()`**.
- **`useReducer()`** utilizes a reducer function that converts the current state and a dispatched action to a new state value.
- **`useReducer()`** returns an array with exactly two elements: the state value and a dispatch function, which is used for dispatching actions

WHAT'S NEXT?

Being able to manage both simple and complex state values efficiently is important. This chapter introduced two crucial tools that help with the task.

With the context API's **`useContext()`** and **`useReducer()`** Hooks, two new React Hooks were introduced. Combined with all the other Hooks covered thus far in the book, these mark the last of the React Hooks you will need in your everyday work as a React developer.

As a React developer, you're not limited to the built-in Hooks though. You can also build your own Hooks. The next chapter will finally explore how that works and why you might want to build custom Hooks in the first place.

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 the examples that can be found at https://packt.link/wc8xd:

1. What problem can be solved with React's context API?
2. Which three main steps have to be taken when using the context API?
3. When might `useReducer()` be preferred over `useState()`?
4. When working with `useReducer()`, what's the role of actions?

APPLY WHAT YOU LEARNED

Apply your knowledge about the context API and the `useReducer()` Hook to some real problems.

ACTIVITY 10.1: MIGRATING AN APP TO THE CONTEXT API

In this activity, your task is to improve an existing React project. Currently, the app is built without the context API and so cross-component state is managed by lifting the state up. In this project, prop drilling is the consequence in some components. Therefore, the goal is to adjust the app such that the context API is used for cross-component state management.

> **NOTE**
>
> You can find the starting code for this activity at https://packt.link/93LSa. 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.
>
> The provided project also uses many features covered in earlier chapters. Take your time to analyze it and understand the provided code. This is a great practice and allows you to see many key concepts in action.

Once you have downloaded the code and run **`npm install`** in the project folder (to install all required dependencies), you can start the development server via **`npm start`**. As a result, upon visiting **`localhost:3000`**, you should see the following user interface:

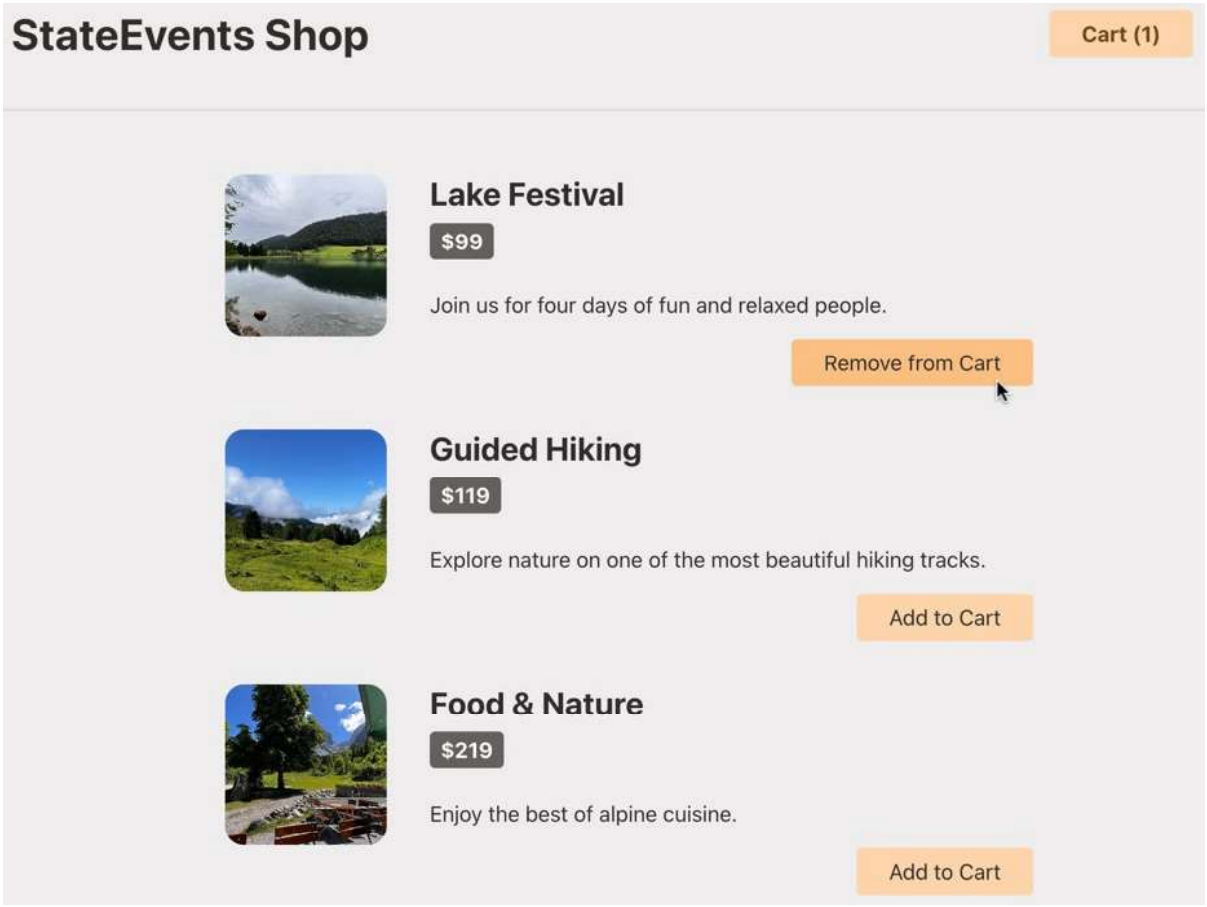

Figure 10.5: The running starting project

To complete the activity, the steps are as follows:

1. Create a new context for the cart items.
2. Create a **`Provider`** component for the context and handle all context-related state changes there.
3. Provide the context (with the help of the **`Provider`** component) and make sure all components that need access to the context have access.

4. Remove the old logic (where state was lifted up).
5. Use the context in all the components that need access to it.

The user interface should be the same as that shown in *Figure 10.5* once you have completed the activity. Make sure that the user interface works exactly as it did before you implemented React's context features.

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

ACTIVITY 10.2: REPLACING USESTATE() WITH USEREDUCER()

In this activity, your task is to replace the **`useState()`** Hooks in the **`Form`** component with **`useReducer()`**. Use only one single reducer function (and thus only one **`useReducer()`** call) and merge all relevant state values into one state object.

> **NOTE**
>
> You can find the starting code for this activity at https://packt.link/wUDJu. 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.
>
> The provided project also uses many features covered in earlier chapters. Take your time to analyze it and understand the provided code. This is a great practice and allows you to see many key concepts in action.

Once you have downloaded the code and run **`npm install`** in the project folder (to install all required dependencies), you can start the development server via **`npm start`**. As a result, upon visiting **`localhost:3000`**, you should see the following user interface:

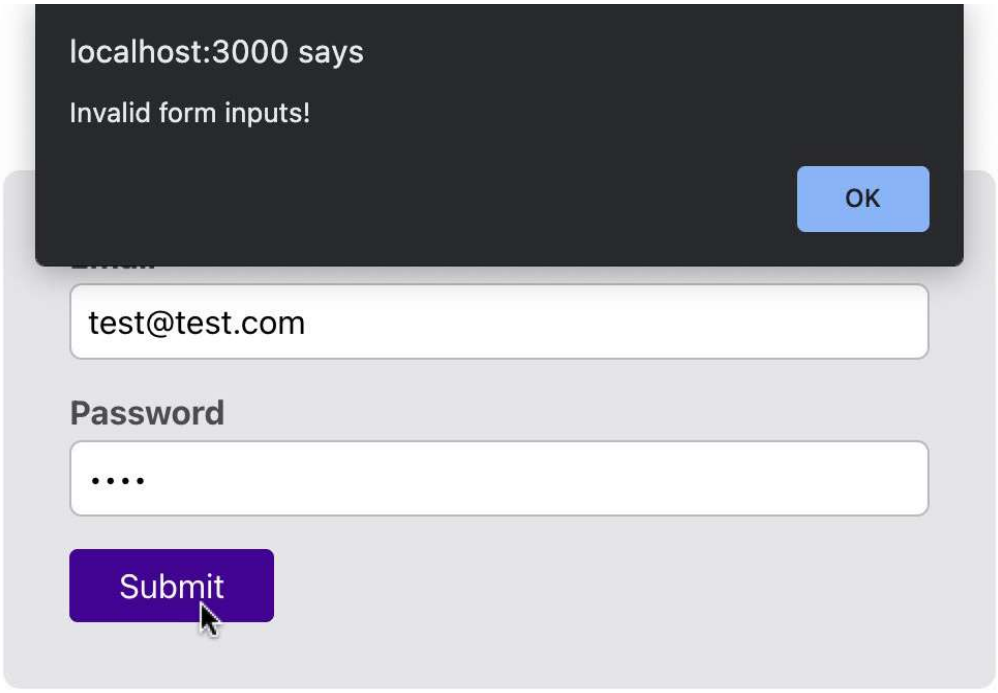

Figure 10.6: The running starting project

To complete the activity, the solution steps are as follows:

1. Remove (or comment out) the existing logic in the Form component that uses the **`useState()`** Hook for state management.
2. Add a reducer function that handles two actions (email changed and password changed) and also returns a default value.
3. Update the state object based on the dispatched action type (and payload, if needed).

4. Use the reducer function with the **`useReducer()`** Hook.
5. Dispatch the appropriate actions (with the appropriate data) in the **`Form`** component.
6. Use the state value where needed.

The user interface should be the same as that shown in *Figure 10.6* once you've finished the activity. Make sure that the user interface works exactly as it did before you implemented React's context features.

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

11

BUILDING CUSTOM REACT HOOKS

LEARNING OBJECTIVES

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

- Build your own React Hooks.
- Use custom and default React Hooks in your components.

INTRODUCTION

Throughout this book, one key React feature has been referenced repeatedly in many different variations. That feature is React Hooks.

Hooks power almost all core functionalities and concepts offered by React—from state management in a single component to accessing cross-component state (context) in multiple components. They enable you to access JSX elements via refs and allow you to handle side effects inside of component functions.

Without Hooks, modern React would not work, and building feature-rich applications would be impossible.

Thus far, only built-in Hooks have been introduced and used. However, you can build your own custom Hooks as well. In this chapter, you will learn why you might want to do this and how it works.

WHY WOULD YOU BUILD CUSTOM HOOKS?

In the previous chapter (*Chapter 10*, *Working with Complex State*), when the **`useReducer()`** Hook was introduced, an example was provided in which the Hook was utilized in sending an HTTP request. Here's the relevant, final code again:

```
const initialHttpState = {
  data: null,
  isLoading: false,
  error: null,
};

function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state, // copying the existing state
      isLoading: state.data ? false : true,
      error: null,
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
```

```
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
      isLoading: false,
      error: null,
    };
  }

  return initialHttpState; // default value for unknown actions
}

function App() {
  const [httpState, dispatch] = useReducer(httpReducer,
initialHttpState);

  // Using useCallback() to prevent an infinite loop in useEffect() below
  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
```

```
      fetchPosts();
    },
    [fetchPosts]
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
        <button onClick={fetchPosts}>Load Posts</button>
      </header>
      {httpState.isLoading && <p>Loading...</p>}
      {httpState.error && <p>{httpState.error}</p>}
      {httpState.data && <BlogPosts posts={httpState.data} />}
    </>
  );
};
```

In this code example, an HTTP request is sent whenever the **`App`** component is rendered for the first time. The HTTP request fetches a list of (dummy) posts. Until the request finishes, a loading message (**`<p>Loading`**...**`</p>`**) is displayed to the user. In the case of an error, an error message is displayed.

As you can see, quite a lot of code must be written to handle this relatively basic use case. And, especially in bigger React apps, it is quite likely that multiple components will need to send HTTP requests. They probably won't need to send the exact same request to the same URL (**`https://jsonplaceholder.typicode.com/posts`**, in this example), but it's definitely possible that different components will fetch different data from different URLs.

Therefore, almost the exact same code must be written over and over again in multiple components. And it's not just the code for sending the HTTP request (i.e., the function wrapped by **`useCallback()`**). Instead, the HTTP-related state management (done via **`useReducer()`**, in this example), as well as the request initialization via **`useEffect()`**, must be repeated in all those components.

And that is where custom Hooks come in to save the day. Custom Hooks help you avoid this repetition by allowing you to build reusable, potentially stateful "logic snippets" that can be shared across components.

WHAT ARE CUSTOM HOOKS?

Before starting to build custom Hooks, it's very important to understand what exactly custom Hooks are.

In React apps, custom Hooks are regular JavaScript functions that satisfy the following conditions:

- The function name starts with "use" (just as all built-in Hooks start with "use": **`useState()`**, **`useReducer()`**, etc.).
- The function does not just return JSX code (otherwise, it would essentially be a React component with a strange function name), though it could return some JSX code—as long as that's not the only value returned.

And that's it. If a function simply meets these two conditions, it can (and should) be called a custom (React) Hook.

Custom Hooks are special because you can call other built-in and custom React Hooks inside of their function bodies. Normally, this would not be allowed. React Hooks (both built-in and custom) can only be called inside of component functions. If you try to call a Hook in some other place (e.g., in a regular, non-Hook function), you will get an error, as shown below:

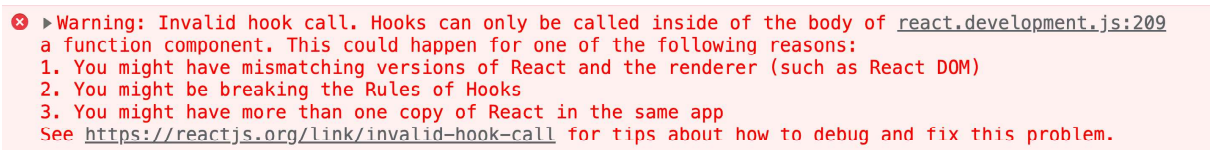

Figure 11.1: React complains if you call a Hook function in the wrong place

Hooks, whether custom or built-in, must only be called inside of component functions. And, even though the error message doesn't explicitly mention it, they may be called inside of custom Hooks.

This is an extremely important feature because it means that you can build reusable non-component functions that can contain state logic (via **`useState()`** or **`useReducer()`**), handle side effects in your reusable custom Hook functions (via **`useEffect()`**), or use any other React Hook. With normal, non-Hook functions, none of these would be possible, and you would therefore be unable to outsource any logic that involves a React Hook into such functions.

In this way, custom Hooks complement the concept of React components. While React components are reusable UI building blocks (which may contain stateful logic), custom Hooks are reusable logic snippets that can be used in your component functions. Thus, custom Hooks help you reuse shared logic across components.

Applied to the problem discussed previously (regarding the HTTP request logic), custom Hooks enable you to outsource the logic for sending an HTTP request and handling the related states (loading, error, etc.).

A FIRST CUSTOM HOOK

Before exploring advanced scenarios and solving the HTTP request problem mentioned previously, here's a more basic example for a first, custom Hook:

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

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

  function increment() {
    setCounter(oldCounter => oldCounter + 1);
  };

  function decrement() {
    setCounter(oldCounter => oldCounter - 1);
  };

  return { counter, increment, decrement };
};

export default useCounter;
```

As you can see, **`useCounter`** is a regular JavaScript function. The name of the function starts with **`use`**, and React therefore treats this function as a custom Hook (meaning you won't get any error messages when using other Hooks inside of it).

Inside **`useCounter()`**, a **`counter`** state is managed via **`useState()`**. The state is changed via two nested functions (**`increment`** and **`decrement`**), and the state, as well as the functions, is returned by **`useCounter`** (grouped together in a JavaScript object).

> **NOTE**
>
> The syntax used to group **`counter`**, **`increment`**, and **`decrement`** together uses a regular JavaScript feature: shorthand property names.
>
> If a property name in an object literally matches the name of the variable whose value is assigned to the property, you can use this shorter notation.
>
> Instead of writing **`{ counter: counter, increment: increment, decrement: decrement }`**, you can use the shorthand notation shown in the snippet above.

This custom Hook can be stored in a separate file (e.g., in a **`hooks`** folder inside the React project, such as **`src/hooks/use-counter.js`**). Thereafter, it can be used in any React component, and you can use it in as many React components as needed.

For example, the following two components (**`Demo1`** and **`Demo2`**) could use this **`useCounter`** Hook as follows:

```
import useCounter from './hooks/use-counter';

function Demo1() {
  const { counter, increment, decrement } = useCounter();

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>

    </>
  );
};

function Demo2() {
  const { counter, increment, decrement } = useCounter();
```

```
  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>
    </>
  );
};

function App() {
  return (
    <main>
      <Demo1 />
      <Demo2 />
    </main>
  );
};

export default App;
```

> **NOTE**
>
> You will find the full example code at https://packt.link/e8zRK.

The **Demo1** and **Demo2** components both execute **useCounter()** inside of their component functions. The **useCounter()** function is called a normal function because it **is** a regular JavaScript function.

Since the **useCounter** Hook returns an object with three properties (**counter**, **increment**, and **decrement**), **Demo1** and **Demo2** use object destructuring to store the property values in local constants. These values are then used in the JSX code to output the **counter** value and connect the two **<button>** elements to the **increment** and **decrement** functions.

After pressing the buttons a couple of times each, the resulting user interface might look like this:

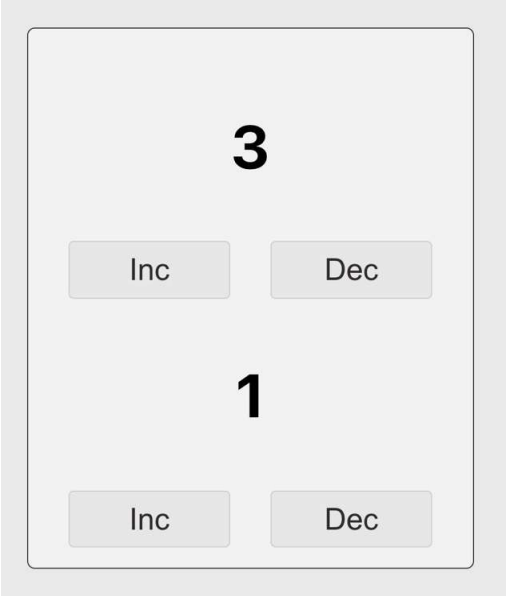

Figure 11.2: Two independent counters

In this screenshot, you can also see a very interesting and important behavior of custom Hooks. That is, if the same stateful custom Hook is used in multiple components, every component gets its own state. The **`counter`** state is not shared. The **`Demo1`** component manages its own **`counter`** state (through the **`useCounter()`** custom Hook), and so does the **`Demo2`** component.

CUSTOM HOOKS: A FLEXIBLE FEATURE

The two independent states of **`Demo1`** and **`Demo2`** show a very important feature of custom Hooks: you use them to share logic, not to share state. If you needed to share state across components, you would do so with React context (see the previous chapter).

When using Hooks, every component uses its own "instance" (or "version") of that Hook. It's always the same logic, but any state or side effects handled by a Hook are handled on a per-component basis.

It's also worth noting that custom Hooks **can** be stateful but **don't have to be**. They can manage state via **`useState()`** or **`useReducer()`**, but you could also build custom Hooks that only handle side effects (without any state management).

There's only one thing you implicitly have to do in custom Hooks: you must use some other React Hook (custom or built-in). This is because, if you didn't include any other Hook, there would be no need to build a custom Hook in the first place. A custom Hook is just a regular JavaScript function (with a name starting with **`use`**) with which you are allowed to use other Hooks. If you don't need to use any other Hooks, you can simply build a normal JavaScript function with a name that does not start with **`use`**.

You also have a lot of flexibility regarding the logic inside the Hook, its parameters, and the value it returns. Regarding the Hook logic, you can add as much logic as needed. You can manage no state or multiple state values. You can include other custom Hooks or only use built-in Hooks. You can manage multiple side effects, work with refs, or perform complex calculations. There are no restrictions regarding what can be done in a custom Hook.

CUSTOM HOOKS AND PARAMETERS

You can also accept and use parameters in your custom Hook functions. For example, the **`useCounter`** Hook from the "*A First Custom Hook*" section above can be adjusted to take an initial counter value and separate values by which the counter should be increased or decreased, as shown in the following snippet:

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

function useCounter(initialValue, incVal, decVal) {
  const [counter, setCounter] = useState(initialValue);

  function increment() {
    setCounter(oldCounter => oldCounter + incVal);
  };

  function decrement() {
    setCounter(oldCounter => oldCounter - decVal);
  };

  return { counter, increment, decrement };
};

export default useCounter;
```

In this adjusted example, the **`initialValue`** parameter is used to set the initial state via **`useState(initialValue)`**. The **`incVal`** and **`decVal`** parameters are used in the **`increment`** and **`decrement`** functions to change the **`counter`** state with different values.

Of course, once parameters are used in a custom Hook, fitting parameter values must be provided when the custom Hook is called in a component function (or in another custom Hook). Therefore, the code for the **`Demo1`** and **`Demo2`** components must also be adjusted—for example, like this:

```
function Demo1() {
  const { counter, increment, decrement } = useCounter(1, 2, 1);

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>

    </>
  );
};

function Demo2() {
  const { counter, increment, decrement } = useCounter(0, 1, 2);

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>
    </>
  );
};
```

> **NOTE**
>
> You can also find this code on GitHub at https://packt.link/bMony.

Now, both components pass different parameter values to the **`useCounter`** Hook function. Therefore, they can reuse the same Hook and its internal logic dynamically.

CUSTOM HOOKS AND RETURN VALUES

As shown with **`useCounter`**, custom Hooks may return values. And this is important: they **may** return values, but they don't have to. If you build a custom Hook that only handles some side effects (via **`useEffect()`**), you don't have to return any value (because there probably isn't any value that should be returned).

But if you do need to return a value, you decide which type of value you want to return. You could return a single number or string. If your Hook must return multiple values (like **`useCounter`** does), you can group these values into an array or object. You can also return arrays that contain objects or vice versa. In short, you can return anything. It is a normal JavaScript function, after all.

Some built-in Hooks such as **`useState()`** and **`useReducer()`** return arrays (with a fixed number of elements). **`useRef()`**, on the other hand, returns an object (which always has a **`current`** property). **`useEffect()`** returns nothing. Your Hooks can therefore return whatever you want.

For example, the **`useCounter`** Hook from prevously could be rewritten to return an array instead:

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

function useCounter(initialValue, incVal, decVal) {
  const [counter, setCounter] = useState(initialValue);

  function increment() {
    setCounter((oldCounter) => oldCounter + incVal);
  }

  function decrement() {
    setCounter((oldCounter) => oldCounter - decVal);
  }

  return [counter, increment, decrement];
}

export default useCounter;
```

To use the returned values, then, the **`Demo1`** and **`Demo2`** components need to switch from object destructuring to array destructuring, as follows:

```
function Demo1() {
  const [counter, increment, decrement] = useCounter(1, 2, 1);

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>
    </>
  );
}

function Demo2() {
  const [counter, increment, decrement] = useCounter(0, 1, 2);

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>
    </>
  );
}
```

The two components behave like before, so you can decide which return value you prefer.

> **NOTE**
>
> This finished code can also be found on GitHub at https://packt.link/adTM2.

A MORE COMPLEX EXAMPLE

The previous examples were deliberately rather simple. Now that the basics of custom Hooks are clear, it makes sense to dive into a slightly more advanced and realistic example.

Consider the HTTP request example from the beginning of this chapter:

```
const initialHttpState = {
  data: null,
  isLoading: false,
  error: null,
};

function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state, // copying the existing state
      isLoading: state.data ? false : true,
      error: null,
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
      isLoading: false,
      error: null,
    };
  }

  return initialHttpState; // default value for unknown actions
}
```

```
function App() {
  const [httpState, dispatch] = useReducer(httpReducer,
initialHttpState);

  // Using useCallback() to prevent an infinite loop in useEffect() below
  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
        <button onClick={fetchPosts}>Load Posts</button>
      </header>
      {httpState.isLoading && <p>Loading...</p>}
      {httpState.error && <p>{httpState.error}</p>}
```

```
      {httpState.data && <BlogPosts posts={httpState.data} />}
    </>
  );
};
```

In that example, the entire **`useReducer()`** logic (including the reducer function, **`httpReducer`**) and the **`useEffect()`** call can be outsourced into a custom Hook. The result would be a very lean **`App`** component and a reusable Hook that could be used in other components as well.

This custom Hook could be named **`useFetch`** (since it fetches data), and it could be stored in **`hooks/use-fetch.js`**. Of course, both the Hook name as well as the file storage path are up to you. Here's how the first version of **`useFetch`** might look:

```
import { useCallback, useEffect, useReducer } from 'react';

const initialHttpState = {
  data: null,
  isLoading: false,
  error: null,
};

function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state,
      isLoading: state.data ? false : true,
      error: null,
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
```

```
      isLoading: false,
      error: null,
    };
  }

  return initialHttpState;
}

function useFetch() {
  const [httpState, dispatch] = useReducer(httpReducer,
initialHttpState);

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );
}

export default useFetch;
```

Please note that this is not the final version.

In this first version, the **`useFetch`** Hook contains the **`useReducer()`** and **`useEffect()`** logic. It's worth noting that the **`httpReducer`** function is created outside of **`useFetch`**. This ensures that the function is not recreated unnecessarily when **`useFetch()`** is re-executed (which will happen often as it is called every time the component that uses this Hook is re-evaluated). The **`httpReducer`** function will therefore only be created once (for the entire application lifetime), and that same function instance will be shared by all components that use **`useFetch`**.

Since **`httpReducer`** is a pure function (that is, it always produces new return values that are based purely on the parameter values), sharing this function instance is fine and won't cause any unexpected bugs. If **`httpReducer`** were to store or manipulate any values that are not based on function inputs, it should be created inside of **`useFetch`** instead. This way, you avoid having multiple components accidentally manipulate and use shared values.

However, this version of the **`useFetch`** Hook has two big issues:

- Currently, no value is returned. Therefore, components that use this Hook won't get access to the fetched data or the loading state.
- The HTTP request URL is hardcoded into **`useFetch`**. As a result, all components that use this Hook will send the same kind of request to the same URL.

The first issue can be solved by returning the fetched data (or **`undefined`**, if no data was fetched yet), the loading state value, and the error value. Since these values are exactly the values that make up the **`httpState`** object returned by **`useReducer()`**, **`useFetch`** can simply return that entire **`httpState`** object, as shown here:

```
// httpReducer function and initial state did not change, hence omitted
here
function useFetch() {
  const [httpState, dispatch] = useReducer(httpReducer,
initialHttpState);

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );
```

```
      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return httpState;
}
```

The only thing that changed in this code snippet is the last line of the **`useFetch`** function. With **`return httpState`**, the state managed by **`useReducer()`** (and therefore by the **`httpReducer`** function) is returned by the custom Hook.

To fix the second problem (i.e., the hardcoded URL), a parameter should be added to **`useFetch`**:

```
// httpReducer function and initial state did not change, hence omitted
here
function useFetch(url) {
  const [httpState, dispatch] = useReducer(httpReducer,
initialHttpState);

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
```

```
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, [url]);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return httpState;
}
```

In this snippet, the **`url`** parameter was added to **`useFetch`**. This parameter value is then used inside the **`try`** block when calling **`fetch(url)`**. Please note that **`url`** was also added as a dependency to the **`useCallback()`** dependencies array.

Since **`useCallback()`** is wrapped around the fetching function (to prevent infinite loops by **`useEffect()`**), any external values used inside of **`useCallback()`** must be added to its dependencies array. Since **`url`** is an external value (meaning it's not defined inside of the wrapped function), it must be added. This also makes sense logically: if the **`url`** parameter were to change (i.e., if the component that uses **`useFetch`** changes it), a new HTTP request should be sent.

This final version of the **`useFetch`** Hook can now be used in all components to send HTTP requests to different URLs and use the HTTP state values as needed by the components.

For example, the **`App`** component can use **`useFetch`** like this:

```
import BlogPosts from './components/BlogPosts';
import useFetch from './hooks/use-fetch';

function App() {
  const { data, isLoading, error } = useFetch(
    'https://jsonplaceholder.typicode.com/posts'
```

```
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
      </header>
      {isLoading && <p>Loading...</p>}
      {error && <p>{error}</p>}
      {data && <BlogPosts posts={data} />}
    </>
  );
}

export default App;
```

The component imports and calls **`useFetch()`** (with the appropriate URL as an argument) and uses object destructuring to get the **`data`**, **`isLoading`**, and **`error`** properties from the **`httpState`** object. These values are then used in the JSX code.

Of course, the **`useFetch`** Hook could also return a pointer to the **`fetchPosts`** function (in addition to **`httpState`**) to allow components such as the **`App`** component to manually trigger a new request, as shown here:

```
// httpReducer function and initial state did not change, hence omitted
here
function useFetch(url) {
  const [httpState, dispatch] = useReducer(httpReducer,
initialHttpState);

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
```

```
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return [ httpState, fetchPosts ];
}
```

In this example, the **`return`** statement was changed. Instead of returning just **`httpState`**, **`useFetch`** now returns an array that contains the **`httpState`** object and a pointer to the **`fetchPosts`** function. Alternatively, **`httpState`** and **`fetchPosts`** could have been merged into an object (instead of an array).

In the **`App`** component, **`useFetch`** could now be used like this:

```
import BlogPosts from './components/BlogPosts';
import useFetch from './hooks/use-fetch';

function App() {
  const [{ data, isLoading, error }, fetchPosts] = useFetch(
    'https://jsonplaceholder.typicode.com/posts'
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
        <button onClick={fetchPosts}>Load Posts</button>
      </header>
      {isLoading && <p>Loading...</p>}
      {error && <p>{error}</p>}
      {data && <BlogPosts posts={data} />}
    </>
```

```
  );
}

export default App;
```

The **App** component uses array and object destructuring combined to extract the returned values (and the values nested in the **httpState** object). A newly added **<button>** element is then used to trigger the **fetchPosts** function.

This example effectively shows how custom Hooks can lead to much leaner component functions by allowing for easy logic reuse, with or without state or side effects.

SUMMARY AND KEY TAKEAWAYS

- You can create custom Hooks to outsource and reuse logic that relies on other built-in or custom Hooks.
- Custom Hooks are regular JavaScript functions with names that start with **use**.
- Custom Hooks can call any other Hooks.
- Therefore, custom Hooks can, for example, manage state or perform side effects.
- All components can use custom Hooks by simply calling them like any other (built-in) Hooks.
- When multiple components use the same custom Hook, every component receives its own "instance" (i.e., its own state value, etc.).
- Inside of custom Hooks, you can accept any parameter values and return any values of your choice.

WHAT'S NEXT?

Custom Hooks are a key React feature since they help you to write leaner components and reuse (stateful) logic across them. Especially when building more complex React apps (consisting of dozens or even hundreds of components), custom Hooks can lead to tremendously more manageable code.

Custom Hooks are the last key React feature you must know about. Combined with components, props, state (via **useState()** or **useReducer()**), side effects, and all the other concepts covered in this and previous chapters, you now have everything you need to build production-ready React apps—or, to be precise, almost everything.

Most React apps don't just consist of one single page. Instead, at least on most websites, users should be able to switch between multiple pages. For example, an online shop has a list of products, product detail pages, a shopping cart page, and many other pages.

The next chapter will therefore explore how you can build such multipage apps with React.

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 at https://packt.link/sRKeZ:

1. What is the definition of a custom Hook?
2. Which special feature can be used inside a custom Hook?
3. What happens when multiple components use the same custom Hook?
4. How can custom Hooks be made more reusable?

APPLY WHAT YOU LEARNED

Apply your knowledge about custom Hooks.

ACTIVITY 11.1: BUILD A CUSTOM KEYBOARD INPUT HOOK

In this activity, your task is to refactor a provided component such that it's leaner and no longer contains any state or side-effect logic. Instead, you should create a custom Hook that contains that logic. This Hook could then potentially be used in other areas of the React application as well.

> **NOTE**
>
> You can find the starting code for this activity at https://packt.link/rdwd9. 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.
>
> The provided project also uses many features covered in earlier chapters. Take your time to analyze it and understand the provided code. This is a great practice and allows you to see many key concepts in action.

Once you have downloaded the code and run **`npm install`** in the project folder to install all required dependencies, you can start the development server via **`npm start`**. As a result, upon visiting **`localhost:3000`**, you should see the following user interface:

Figure 11.3: The running starting project

To complete the activity, the solution steps are as follows:

1. Create a new custom Hook file (e.g., in the **`src/hooks`** folder) and create a Hook function in that file.
2. Move the side effect and state management logic into that new Hook function.
3. Make the custom Hook more reusable by accepting and using a parameter that controls which keys are allowed.
4. Return the state managed by the custom Hook.
5. Use the custom Hook and its returned value in the **`App`** component.

The user interface should be the same once you have completed the activity, but the code of the **`App`** component should change. After finishing the activity, **`App`** should contain only this code:

```
function App() {
  const pressedKey = useKeyEvent(['s', 'c', 'p']); // this is your Hook!

  let output = '';

  if (pressedKey === 's') {
    output = '😊';
  } else if (pressedKey === 'c') {
    output = '😭';
  } else if (pressedKey === 'p') {
    output = '🎉';
  }

  return (
```

```
    <main>
      <h1>Press a key!</h1>
      <p>
        Supported keys: <kbd>s</kbd>, <kbd>c</kbd>, <kbd>p</kbd>
      </p>
      <p id="output">{output}</p>
    </main>
  );
}
```

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