SECTION -2

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

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 (important: don't execute this code as it will cause an infinite loop and send a large number of HTTP requests behind the scenes):

```
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;
```

So what's the problem with this code? Why does it create an infinite loop?

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 the external API mentioned in the note box.

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.

Note

async/await is not used directly in the component function body because regular React components must not be async functions. Such functions automatically return a promise as a value (even without an explicit return statement), which is an invalid return value for a React component.

That being said, there are indeed React components that are allowed to use async/await and return a promise. So-called **React Server Components** are not restricted to returning JSX code, strings, etc. This feature will be discussed in detail in Chapter 16, React Server Components & Server Actions.

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.

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 handleFetchPosts() {
    fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
  }
  return (
    <>
      <button onClick={handleFetchPosts}>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 handleFetchPosts() 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 handleFetchPosts() (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 of 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 handleStart() {
    // 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={handleStart}>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.

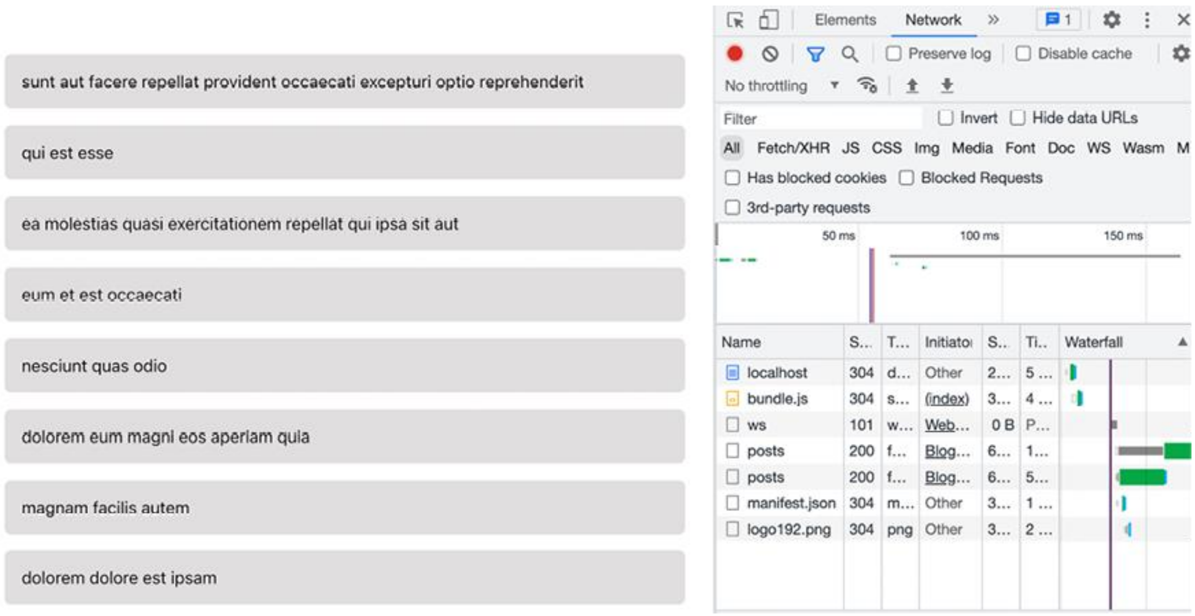
Dealing with Side Effects with the useEffect() Hook

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:



sunt aut facere repellat provident occaecati excepturi optio reprehenderit

qui est esse

ea molestias quasi exercitationem repellat qui ipsa sit aut

eum et est occaecati

nesciunt quas odio

dolorem eum magni eos aperiam quia

magnam facilis autem

dolorem dolore est ipsam

Name	S...	T...	Initiator	S...	Ti...	Waterfall
localhost	304	d...	Other	2...	5 ...	
bundle.js	304	s...	(index)	3...	4 ...	
ws	101	w...	Web...	0 B	P...	
posts	200	f...	Blog...	6...	1...	
posts	200	f...	Blog...	6...	5...	
manifest.json	304	m...	Other	3...	1 ...	
logo192.png	304	png	Other	3...	2 ...	

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 post titles, 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).

Although, unlike useState() or useRef(), useEffect() does not return a value, though 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. 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 it is 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';

async function fetchPosts(url) {

  const response = await fetch(url);

  const blogPosts = await response.json();

  return blogPosts;

}

function BlogPosts({ url }) {
  const [loadedPosts, setLoadedPosts] = useState([]);
  useEffect(function () {
    fetchPosts(url)
     .then((fetchedPosts) => setLoadedPosts(fetchedPosts));
  }, [url]);
  return (
    <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 was adjusted in one important place: BlogPosts now accepts a url prop.

Therefore, this component can now be used and configured by other components. Of course, if some other component sets a URL that 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.

But if that other component changes the URL (e.g., due to some user input there), a new request should be sent, of course. So BlogPosts should send another fetch request every time the url prop value changes.

That's why url 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 url 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 url to the dependencies array, React registers this value (in this case, a prop value, but any value can be registered) and re-executes the effect function whenever that value changes (that is, whenever a new url prop value is set by the component that uses BlogPosts).

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(url)
    .then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, [url]);
```

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.

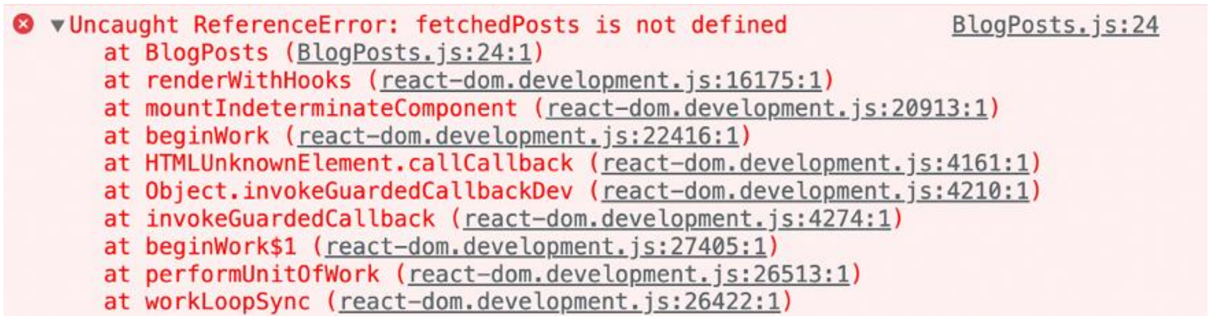
Unnecessary Dependencies

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(url)
    .then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, [url, 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:



```
▼Uncaught ReferenceError: fetchedPosts is not defined          BlogPosts.js:24
    at BlogPosts (BlogPosts.js:24:1)
    at renderWithHooks (react-dom.development.js:16175:1)
    at mountIndeterminateComponent (react-dom.development.js:20913:1)
    at beginWork (react-dom.development.js:22416:1)
    at HTMLUnknownElement.callCallback (react-dom.development.js:4161:1)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4210:1)
    at invokeGuardedCallback (react-dom.development.js:4274:1)
    at beginWork$1 (react-dom.development.js:27405:1)
    at performUnitOfWork (react-dom.development.js:26513:1)
    at workLoopSync (react-dom.development.js:26422:1)
```

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.jsx 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 updating function is only created the first time a component function is called by React. Subsequent executions don't lead to a new 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(). url 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:

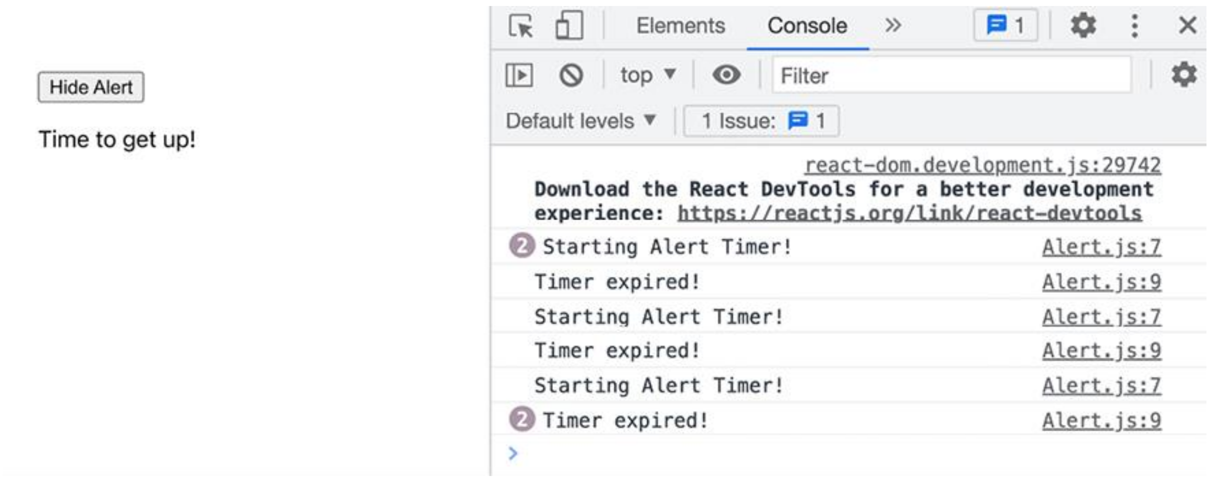
```
import { useState } from 'react';

import Alert from './components/Alert.jsx';

function App() {

  const [showAlert, setShowAlert] = useState(false);

  function handleShowAlert() {

    // 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={handleShowAlert}>

        {showAlert ? 'Hide' : 'Show'} Alert

      </button>

      {showAlert && <Alert />}

    </>

  );

}

export default App;
```

In the App component, the Alert component is shown conditionally.
The showAlert state is toggled via the handleShowAlert 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:



```
react-dom.development.js:29742
Download the React DevTools for a better development
experience: https://reactjs.org/link/react-devtools
2 Starting Alert Timer!                    Alert.js:7
Timer expired!                             Alert.js:9
Starting Alert Timer!                      Alert.js:7
Timer expired!                             Alert.js:9
Starting Alert Timer!                      Alert.js:7
2 Timer expired!                           Alert.js:9
```

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 provides 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);
  }
}, []);
```

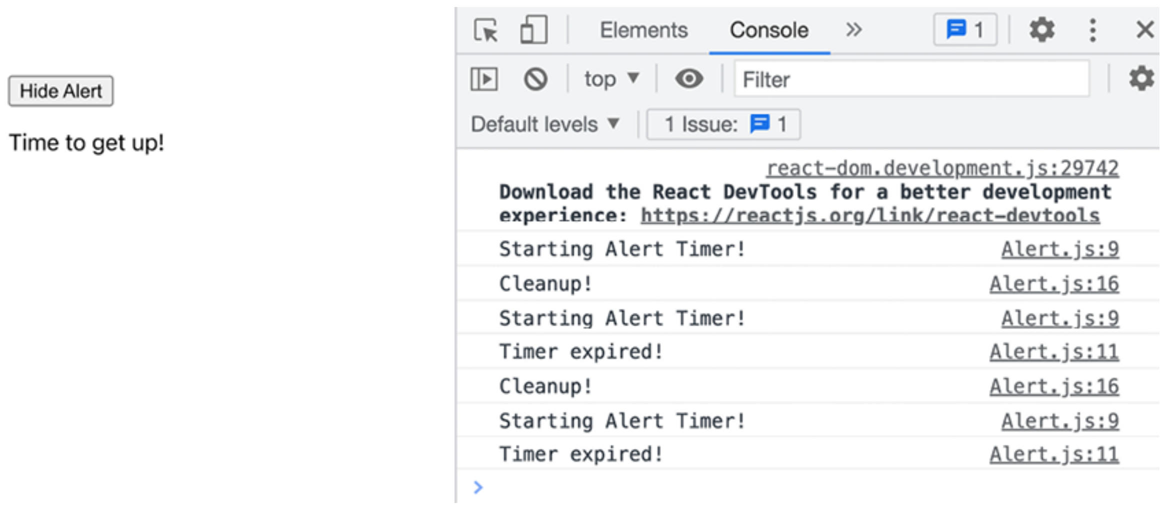
In this updated example, a new timer variable (a local variable that is only accessible inside the effect function) is added. That variable stores a reference to the timer that's created by setTimeout(). This reference can then be used together with clearTimeout() to remove a timer.

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:



```
react-dom.development.js:29742
Download the React DevTools for a better development
experience: https://reactjs.org/link/react-devtools
Starting Alert Timer!                    Alert.js:9
Cleanup!                                 Alert.js:16
Starting Alert Timer!                    Alert.js:9
Timer expired!                           Alert.js:11
Cleanup!                                 Alert.js:16
Starting Alert Timer!                    Alert.js:9
Timer expired!                           Alert.js:11
```

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. You could put every HTTP request, every console.log() statement, and every timer 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().

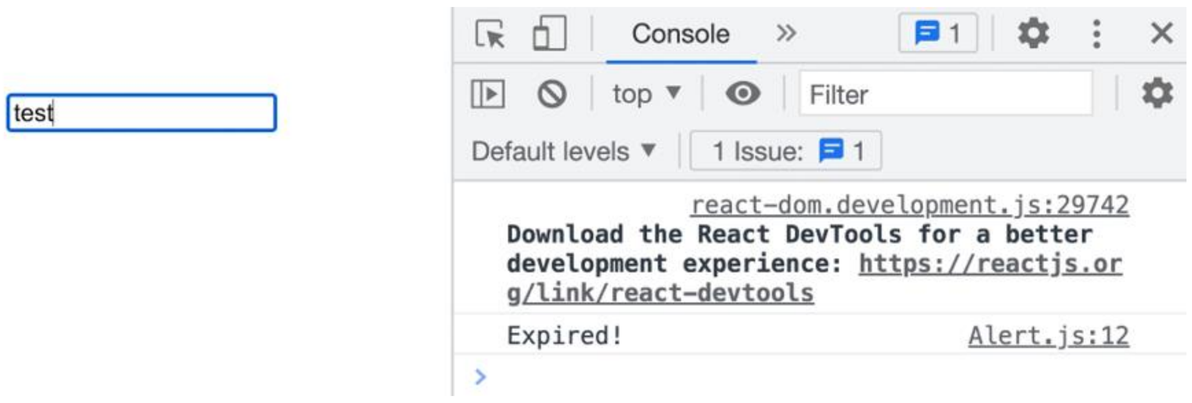
Another reason for this is that every time the Alert component function is executed again (e.g., because some state or prop value changes), a
new setAlert function object will be created. In this example, that wouldn’t be problematic because setAlert only contains static code. A new function object created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

But now consider this adjusted example:

```
function Alert() {
 const [alertMsg, setAlertMsg] = useState('Expired!');
 function handleChangeAlertMsg(event) {
  setAlertMsg(event.target.value);
 }
 function setAlert() {
  setTimeout(function () {
   console.log(alertMsg);
  }, 2000);
 }
 useEffect(
  function () {
   setAlert();
  },
  []
 );
 return <input type="text" onChange={handleChangeAlertMsg} />;
}
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:



```
react-dom.development.js:29742
Download the React DevTools for a better
development experience: https://reactjs.or
g/link/react-devtools
Expired!                          Alert.js:12
```

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.

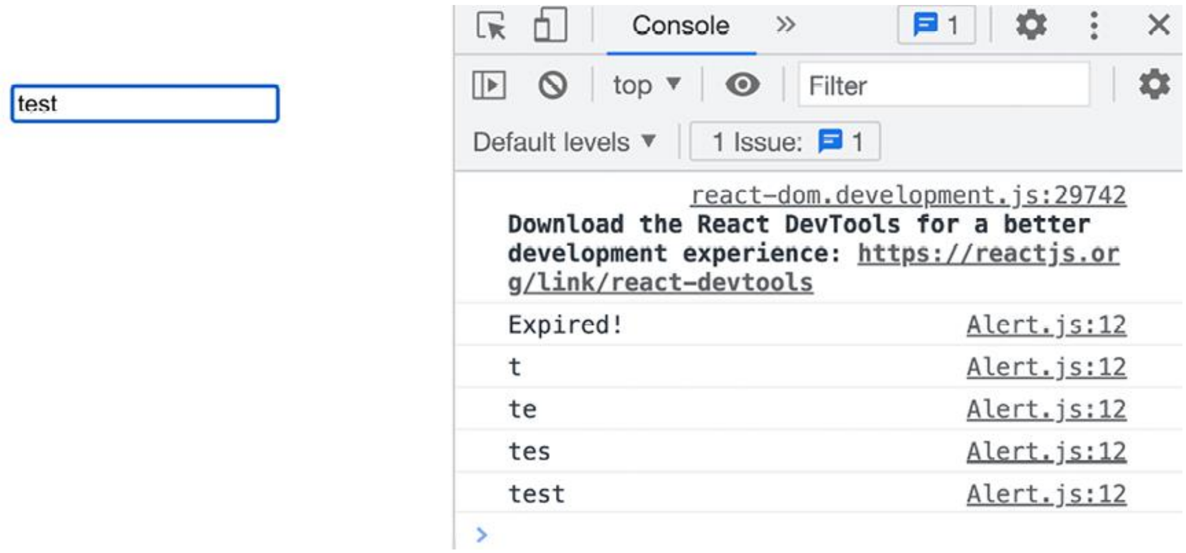
The reason for this behavior is that the new alert message is not picked up. It's not used because, despite the component function being executed again (because the state changed), the effect is not executed again. And the original execution of the effect still uses the old version of the setAlert function—the 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).



```
react-dom.development.js:29742
Download the React DevTools for a better
development experience: https://reactjs.or
g/link/react-devtools
Expired!                                  Alert.js:12
t                                         Alert.js:12
te                                        Alert.js:12
tes                                       Alert.js:12
test                                      Alert.js:12
```

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, those 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:

```
import { useState, useEffect } from 'react';
function Alert() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [enteredPassword, setEnteredPassword] = useState('');
  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }
  function handleUpdatePassword(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={handleUpdateEmail} />
     </div>
     <div>
      <label>Password</label>
      <input type="password" onChange={handleUpdatePassword} />
     </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 handleUpdateEmail. Having console.log() (a side effect) in there would be acceptable since it wouldn't cause any harm.
- You could call validateEmail() directly in the component function—since it doesn't change any state, it wouldn't trigger an infinite loop.

Note

There is an article in the official React documentation that highlights scenarios where you might not need useEffect(): https://react.dev/learn/you-might-not-need-an-effect.

.

Of course, in some other scenarios, you might need to use useEffect(). 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 handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }
  function handleUpdatePassword(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 (which 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:

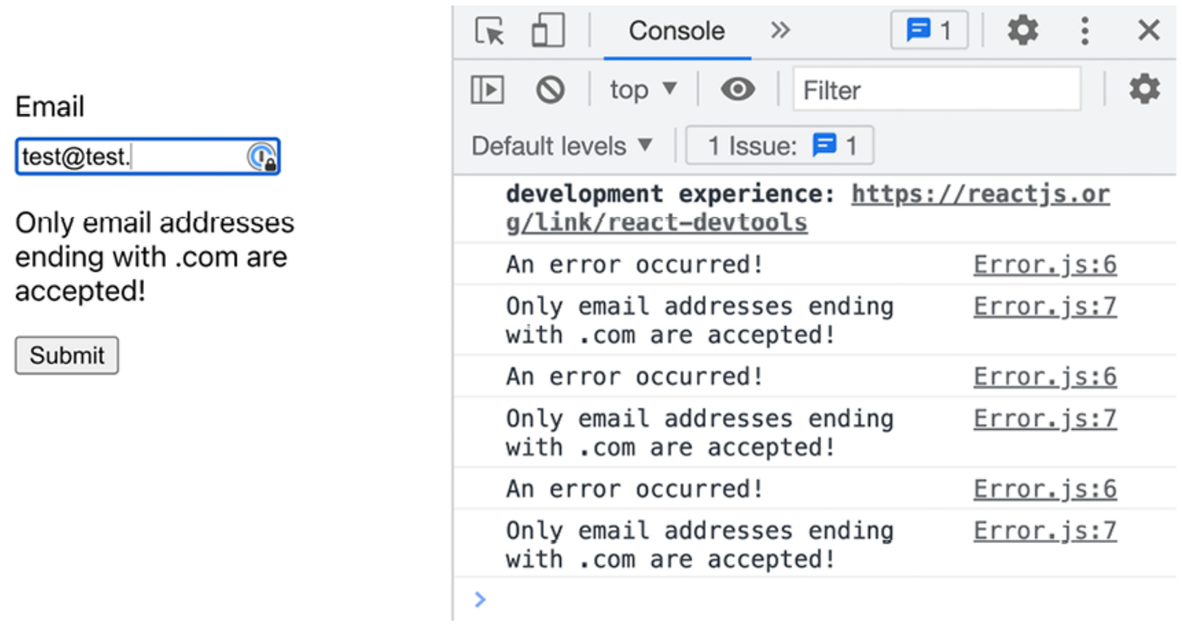
```
import { useState } from 'react';
import Error from './Error.jsx';
function Form() {
 const [enteredEmail, setEnteredEmail] = useState('');
 const [errorMessage, setErrorMessage] = useState('');
 function handleUpdateEmail(event) {
  setEnteredEmail(event.target.value);
 }
 function handleSubmitForm(event) {
  event.preventDefault();
  if (!enteredEmail.endsWith('.com')) {
   setErrorMessage('Only email addresses ending with .com are accepted!');
  }
 }
 return (
  <form onSubmit={handleSubmitForm}>
   <div>
    <label>Email</label>
    <input type="email" onChange={handleUpdateEmail} />
   </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.



Only email addresses ending with .com are accepted!

```
development experience: https://reactjs.org/link/react-devtools
An error occurred!                      Error.js:6
Only email addresses ending             Error.js:7
with .com are accepted!
An error occurred!                      Error.js:6
Only email addresses ending             Error.js:7
with .com are accepted!
An error occurred!                      Error.js:6
Only email addresses ending             Error.js:7
with .com are accepted!
```

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(message);
  },
  // [props] // don't use the entire props object!
  [message]
 );
 return <p>{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

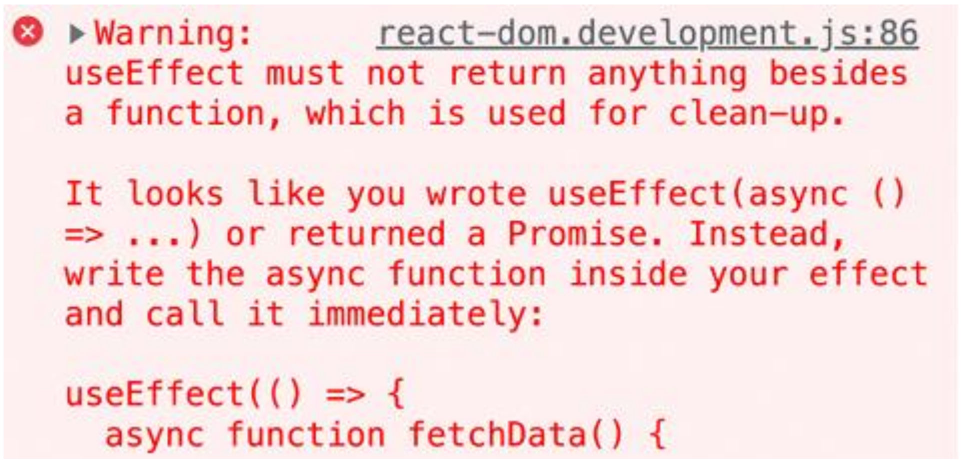
Some effects deal with asynchronous code (sending HTTP requests is a typical example). When performing asynchronous tasks in effect functions, there is one important rule to keep in mind, though: the effect function itself should not be asynchronous and should not return a promise. This does not mean that you can't work with promises in effects—you just must 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 snippet:



```
▶Warning:        react-dom.development.js:86
useEffect must not return anything besides
a function, which is used for clean-up.

It looks like you wrote useEffect(async ()
=> ...) or returned a Promise. Instead,
write the async function inside your effect
and call it immediately:

useEffect(() => {
  async function fetchData() {
```

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));
}, []);
```

This works because the effect function doesn't return the promise.

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 12, 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 is 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.

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: 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.

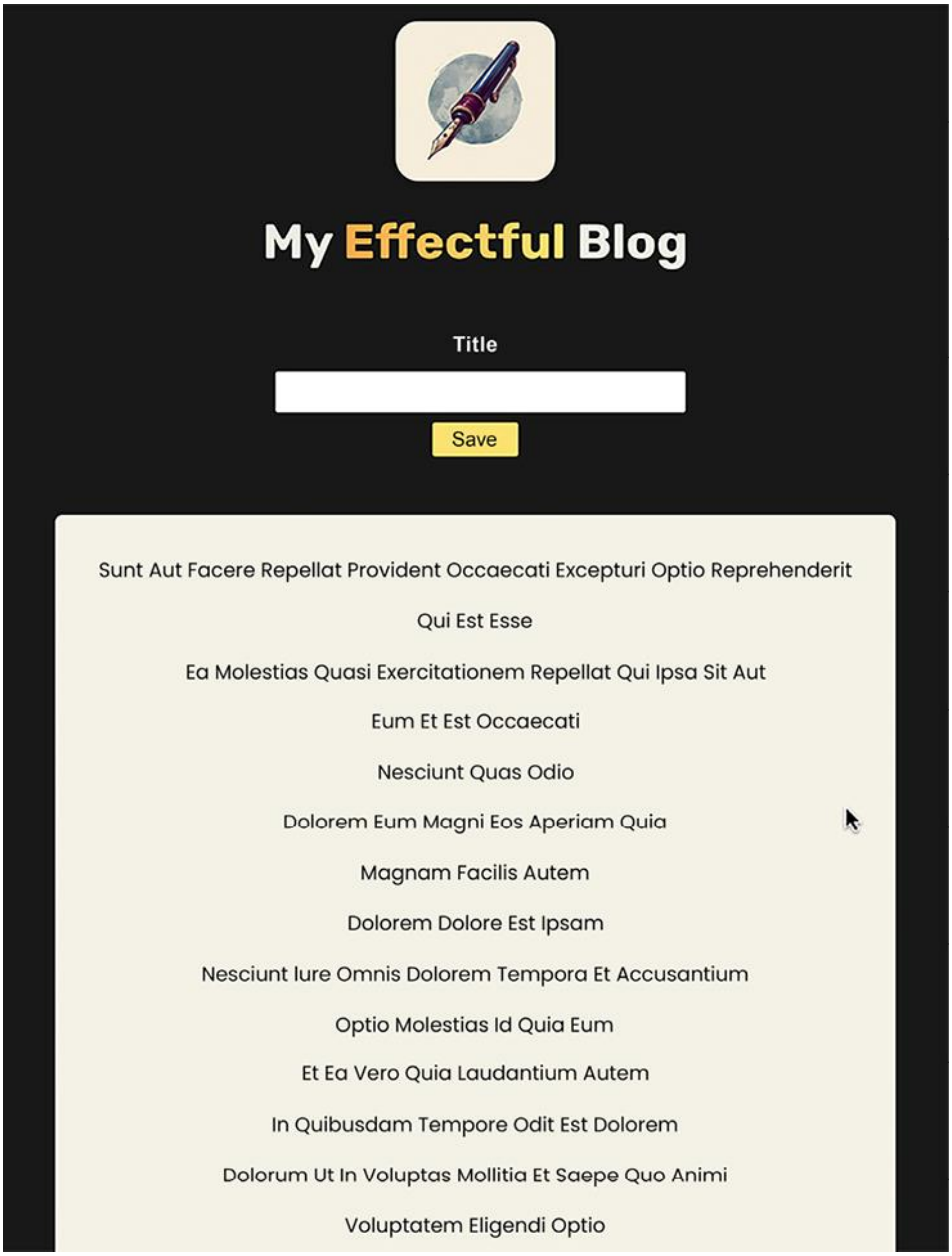
Use your knowledge about effects and browser-side HTTP requests to implement a solution.

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).

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:



Sunt Aut Facere Repellat Provident Occaecati Excepturi Optio Reprehenderit

Qui Est Esse

Ea Molestias Quasi Exercitationem Repellat Qui Ipsa Sit Aut

Eum Et Est Occaecati

Nesciunt Quas Odio

Dolorem Eum Magni Eos Aperiam Quia

Magnam Facilis Autem

Dolorem Dolore Est Ipsam

Nesciunt Iure Omnis Dolorem Tempora Et Accusantium

Optio Molestias Id Quia Eum

Et Ea Vero Quia Laudantium Autem

In Quibusdam Tempore Odit Est Dolorem

Dolorum Ut In Voluptas Mollitia Et Saepe Quo Animi

Voluptatem Eligendi Optio

Figure 8.9: The final user interface

MULTIPAGE APPS WITH REACT ROUTER

Learning Objectives

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

- Build multipage single-page applications (and understand why this is not an oxymoron)
- Use the React Router package to load different React components for different URL paths
- Create static and dynamic routes (and understand what routes are in the first place)
- Navigate the website via both links and programmatic commands
- Build nested page layouts

By now you should know how to build React components and web apps, as well as how to manage components and app-wide state, and how to share data between components (via props or context).

But even though you know how to compose a React website from multiple components, all these components are on the same **single website page**. Sure, you can display components and content conditionally, but users will **never switch to a different page**. This means that the URL path will never change; users will always stay on **your-domain.com**. Also, at this point in time, your React apps don't support any paths such as your-domain.com/products or your-domain.com/blog/latest.

Note

Uniform Resource Locators (**URLs**) are references to web resources. For example, https://academind.com/courses is a URL that points to a specific page of the website. In this example, academind.com is the **domain name** of the website and /courses is the **path** to a specific website page.

For React apps, it might make sense that the path of the loaded website never changes.

But even though it might make sense, it's also quite a serious limitation.

One Page Is Not Enough

Having just a single page means that complex websites that would typically consist of multiple pages (e.g., an online shop with pages for **products**, **orders**, and more) become quite difficult to build with React. Without **multiple pages**, you have to fall back to state and conditional values to display different content on the screen.

But without changing URL paths, your website visitors can't share links to anything but the starting page of your website. Also, any conditionally loaded content will be lost when a new visitor visits that starting page. That will also be the case if users simply reload the page they're currently on. A reload fetches a new version of the page, and so any state (and therefore user interface) changes are lost.

For example , if you want to share the path to admin page of a site like x.com/admin (and the page may be using different layouts,navigation etc. from the home page style) ,it's not possible in the SPA concept we learned

For these reasons, you absolutely need a way of including multiple pages (with different URL paths) in a single React app for most React websites. Thanks to modern browser features and a highly popular third-party package, that is indeed possible (and the default for most React apps).

Via the **React Router** package, your React app can listen to URL path changes and display different components for different paths. For example, you could define the following path-component mappings:

- `<domain>/` => `<Home />` component is loaded.
- `<domain>/products` => `<ProductList />` component is loaded.
- `<domain>/products/p1` => `<ProductDetail />` component is loaded.
- `<domain>/about` => `<AboutUs />` component is loaded.

Technically, it will still be a SPA because there's still only one HTML page being sent to website users. But in that single-page React app, different components are rendered conditionally by the React Router package based on the specific URL paths that are being visited. As the developer of the app, you don't have to manually manage this kind of state or render content conditionally—React Router will do it for you. In addition, your website is able to handle different URL paths, and therefore, individual pages can be shared or reloaded.

Getting Started with React Router and Defining Routes

React Router is a third-party React library that can be installed in any React project. Once installed, you can use various components in your code to enable the aforementioned features.

Inside your React project, the package is installed via this command:

```
npm install react-router-dom
```

Once installed, you can import and use various components (and Hooks) from that library.

To start supporting multiple pages in your React app, you need to set up **routing** by going through the following steps:

1. Create different components for your different pages (e.g., `Dashboard` and `Orders` components).
2. Use the `createBrowserRouter()` function and the `RouterProvider` component from the React Router library to enable routing and define the **routes** that should be supported by the React app.

In this context, the term **routing** refers to the React app being able to load different components for different URL paths (e.g., different components for the `/` and `/orders` paths). A route is a definition that's added to the React app that defines the URL path for which a predefined JSX snippet should be rendered (e.g., the `Orders` component should be loaded for the `/orders` path).

In an example React app that contains `Dashboard` and `Orders` components, and wherein the React Router library was installed via `npm install`, you can enable routing and navigation between these two components by editing the root component (in `src/App.jsx`) like this:

```
import {
  createBrowserRouter,
  RouterProvider
} from 'react-router-dom';
```

```
import Dashboard from './routes/Dashboard.jsx';
import Orders from './routes/Orders.jsx';
const router = createBrowserRouter([
  { path: '/', element: <Dashboard /> },
  { path: '/orders', element: <Orders /> }
]);
function App() {
  return <RouterProvider router={router} />;
}
export default App;
```

In the preceding code snippet, React Router's `createBrowserRouter()` function is called to create a `router` object that contains the application's route configuration (a list of available routes). The array passed to `createBrowserRouter()` contains route definition objects, where every object defines a `path` for which the route should be matched and an `element` that should be rendered.

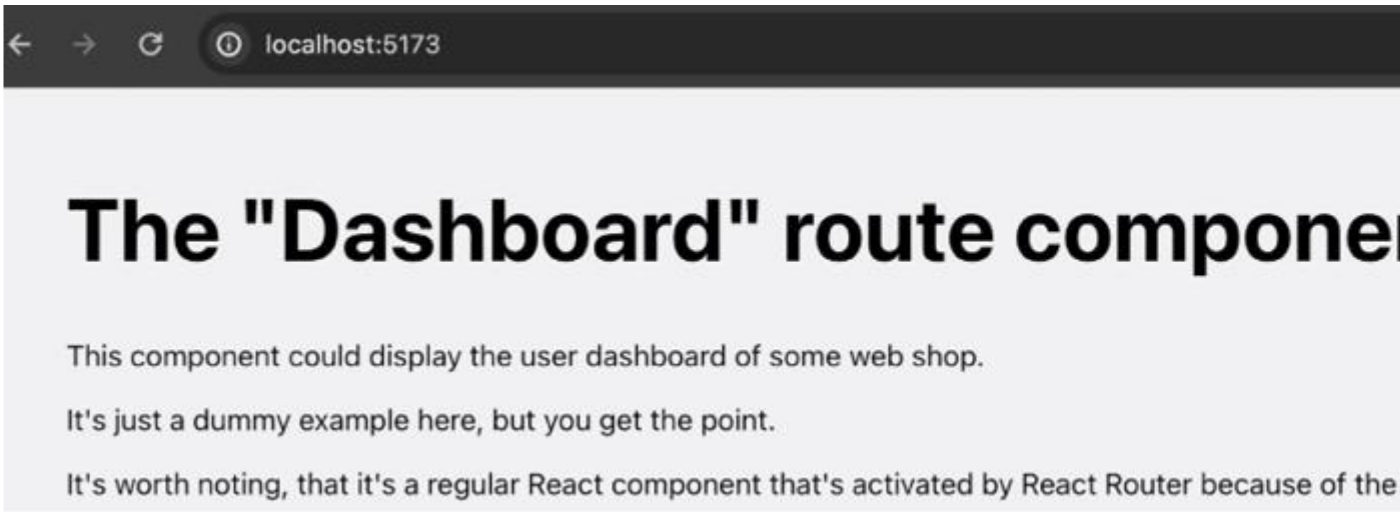
React Router's `RouterProvider` component is then used to set the `router` configuration and define a place for the active route elements to be rendered.

You can think of the `<RouterProvider />` element being replaced with the content defined via the `element` property once a route becomes active. Therefore, the positioning of the `RouterProvider` component matters. In this case (and probably in most React apps), it's the root application component—i.e., React Router, that should control the entire application component tree.

Note

The code for this example can be found at https://github.com/TalipotTech/ReactConcepts/tree/main/examples/03-naive-navigation-problem

If you run the provided example React app (via `npm run dev`), you'll see the following output on the screen:



The "Dashboard" route compone

This component could display the user dashboard of some web shop.

It's just a dummy example here, but you get the point.

It's worth noting, that it's a regular React component that's activated by React Router because of the

Figure 13.1: The Dashboard component content is loaded

The content of the `Dashboard` component is displayed on the screen if you visit `localhost:5173`. Please note that the visible page content is not defined in the `App` component (in the code snippet shared previously). Instead, only two route definitions were added: one for the `/` path (i.e., for `localhost:5173/` or just `localhost:5173`, without the trailing forward slash—it's handled in the same way) and one for the `/orders` path (`localhost:5173/orders`).

Note

`localhost` is a local address that's typically used for development. When you deploy your React app (i.e., you upload it to a web server), you will receive a different domain—or assign a custom domain. Either way, it will not be `localhost` after deployment.

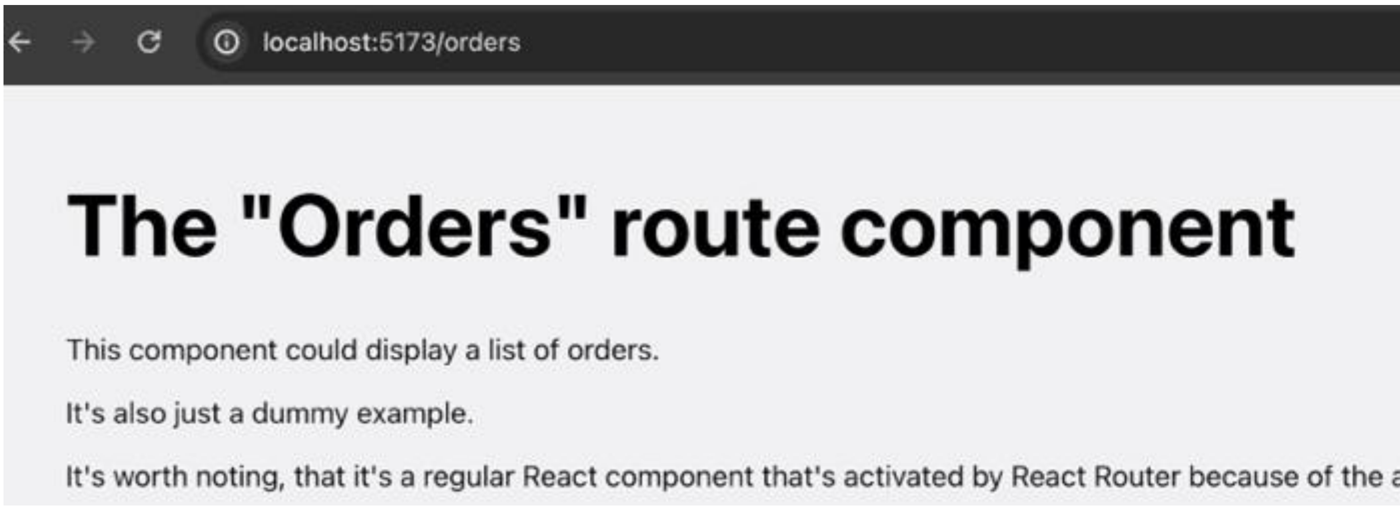
The part after `localhost` (`:5173`) defines the network port to which the request will be sent. Without the additional port information, ports `80` or `443` (as the default HTTP(S) ports) are used automatically. During development, however, these are not the ports you want. Instead, you would typically use ports such as `5173`, `8000`, or `8080` as these are normally unoccupied by any other system processes and hence can be used safely. Projects created via Vite typically use port `5173`.

Since `localhost:5173` is loaded by default (when running `npm run dev`), the first route definition (`{ path: '/', element: <Dashboard /> }`) becomes active. This route is active because its path (`'/'`) matches the path of `localhost:5173` (since this is the same as `localhost:5173/`).

As a result, the JSX code defined via `element` is rendered in place of the `<RouterProvider>` component by React Router. In this case, this means that the content of the `Dashboard` component is displayed because the `element` property value of this route definition is `<Dashboard />`. It is quite common to use single components (such as `<Dashboard />`, in this example), but you could set any JSX content as a value for the `element` property.

In the preceding example, no complex page is displayed. Instead, only some text shows up on the screen. This will change later in this chapter, though.

But it gets interesting if you manually change the URL from just `localhost:5173` to `localhost:5173/orders` in the browser address bar. In any of the previous chapters, this would not have changed the page content. But now, with routing enabled and the appropriate routes being defined, the page content does change, as shown:



The "Orders" route component

This component could display a list of orders.

It's also just a dummy example.

It's worth noting, that it's a regular React component that's activated by React Router because of the a

Figure 13.2: For /orders, the content of the Orders component is displayed

Once the URL changes, the content of the `Orders` component is displayed on the screen. It's again just some basic text in this first example, but it shows that different code is rendered for different URL paths.

However, this basic example has a major flaw (besides the quite boring page content). Right now, users must enter URLs manually. But, of course, that's not how you typically use websites.

Adding Page Navigation

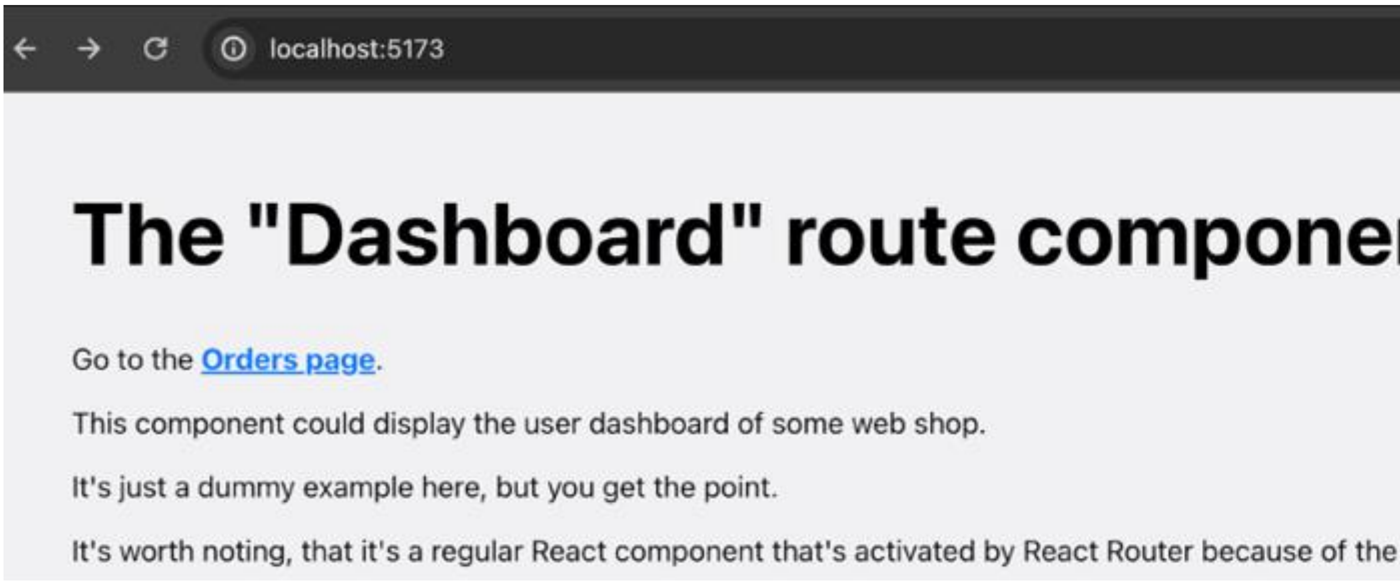
To allow users to switch between different website pages without editing the browser address bar manually, websites normally contain links, typically added via the `<a>` HTML element (the anchor element), like this:

```
<a href="/orders">Past Orders</a>
```

For this example, on-page navigation could therefore be added by modifying the `Dashboard` component code like this:

```
function Dashboard() {
  return (
    <>
   <h1>The "Dashboard" route component</h1>
      <p>Go to the <a href="/orders">Orders page</a>.</p>
   {/* <p> elements omitted */}
  </>
 );
}
export default Dashboard;
```

In this code snippet, a link to the `/orders` route has been added. Website visitors therefore see this page now:


localhost:5173

The "Dashboard" route compone

Go to the Orders page.

This component could display the user dashboard of some web shop.

It's just a dummy example here, but you get the point.

It's worth noting, that it's a regular React component that's activated by React Router because of the

Figure 13.3: A navigation link was added

When website users click this link, they are therefore taken to the `/orders` route and the content of the `Orders` component is displayed on the screen.

This approach works but has a major flaw: the website is reloaded every time a user clicks the link. You can tell that it's reloaded because the browser's refresh icon changes to a cross (briefly) whenever you click a link.

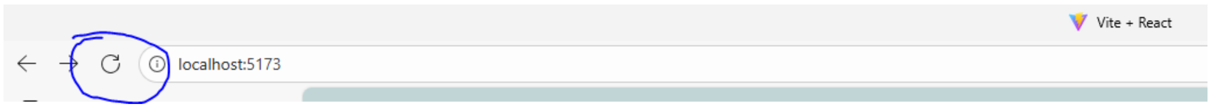


Fig:13.2.1 The blue rounded in picture is the refresh button in Edge brower.Watch carefully when you click the link. The icon changes to a cross (briefly) whenever you click a link

This happens because the browser sends a new HTTP request to the server whenever a link is clicked. Even though the server always returns the same single HTML page, the page is reloaded during that process (because of the new HTTP request that was sent).

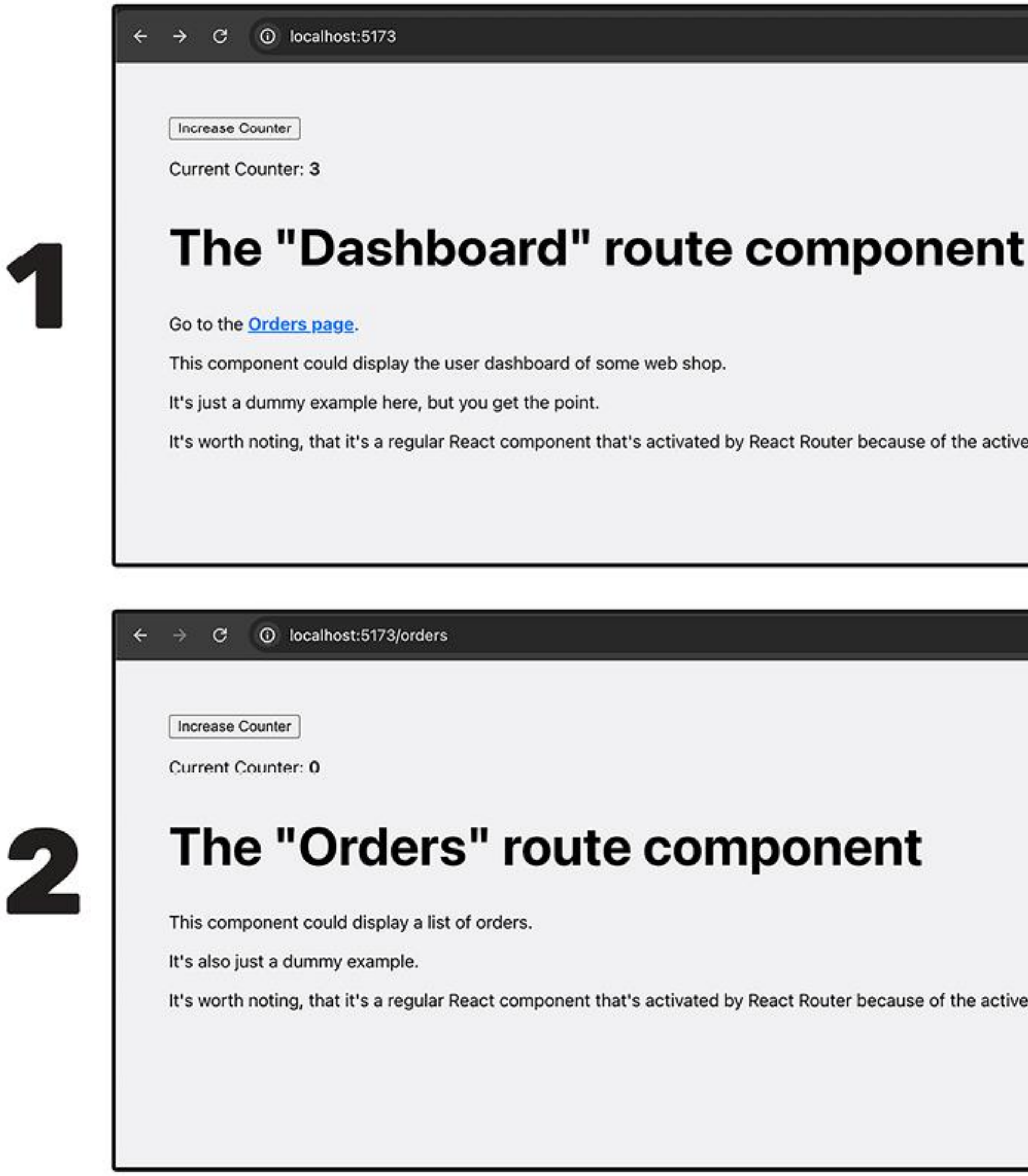
While that's not a problem on this simple demo page, it would be an issue if you had some shared state (e.g., app-wide state managed via context) that should not be reset during a page change. In addition, every new request takes time and forces the browser to download all website assets (e.g., script files) again. Even though those files might be cached, this is an unnecessary step that may impact website performance.

The following, slightly adjusted, example `App` component illustrates the state-resetting problem:

```
import { useState } from 'react';
import {
  createBrowserRouter,
  RouterProvider
} from 'react-router-dom';
import Dashboard from './routes/Dashboard.jsx';
import Orders from './routes/Orders.jsx';
const router = createBrowserRouter([
  { path: '/', element: <Dashboard /> },
  { path: '/orders', element: <Orders /> },
]);
function App() {
  const [counter, setCounter] = useState(0);
  function handleIncCounter() {
     setCounter((prevCounter) => prevCounter + 1);
  }
  return (
     <>
        <p>
           <button onClick={handleIncCounter}>Increase Counter</button>
        </p>
        <p>Current Counter: <strong>{counter}</strong></p>
    <RouterProvider router={router} />
  </>
  );
}
export default App;
```

In this example, a simple counter was added to the `App` component.
Since `<RouterProvider>` is rendered in that same component, below the counter, the `App` component should not be replaced when a user visits a different page (instead, it’s `<RouterProvider>` that should be replaced—not the entire `App` component JSX code).

At least, that’s the theory. But, as you can see in the following screenshot, the `counter` state is lost whenever any link is clicked:



Current Counter: 3

The "Dashboard" route component

Go to the Orders page.

This component could display the user dashboard of some web shop.

It's just a dummy example here, but you get the point.

It's worth noting, that it's a regular React component that's activated by React Router because of the active

Current Counter: 0

The "Orders" route component

This component could display a list of orders.

It's also just a dummy example.

It's worth noting, that it's a regular React component that's activated by React Router because of the active

Figure 13.4: The counter state is reset when switching the page

In the screenshot, you can see that the counter is initially set to `3` (because the button was clicked thrice). After navigating from `Dashboard` to the `Orders` page (via clicking the `Orders page` link), the counter changes to `0`.

That happens because the page is reloaded due to the HTTP request that's sent by the browser.

To work around this issue and avoid this unintended page reload, you must prevent the browser's default behavior. Instead of sending a new HTTP request, the browser URL address should just be updated
(from `localhost:5173` to `localhost:5173/orders`) and the target component (`Orders`) should be loaded. Therefore, to the website user, it would seem as if a different page was loaded. But behind the scenes, it's just the page document (the DOM) that was updated.

Thankfully, you don't have to implement the logic for this on your own. Instead, the React Router library exposes a special `Link` component that should be used instead of the anchor `<a>` element.

To use this new component, the code in `src/routes/Dashboard.jsx` must be adjusted like this:

```
import { Link } from 'react-router-dom';
function Dashboard() {
  return (
     <>
    <h1>The "Dashboard" route component</h1>
    <p>Go to the <Link to="/orders">Orders page</Link>.</p>
    <p>
      This component could display the user dashboard
      of some web shop.
    </p>
    <p>It's just a dummy example here, but you get the point.</p>
    <p>
      It's worth noting, that it's a regular React component
      that's activated by React Router because of the
      active route configuration.
    </p>
   </>
  );
}
export default Dashboard;
```

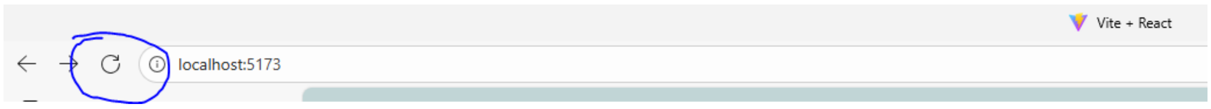
Note

The code for this example can be found at

[ReactConcepts/examples/04-react-router-navigation at main · TalipotTech/ReactConcepts]()

Inside this updated example, the new `Link` component is used. That component requires a `to` prop, which is used to define the URL path that should be loaded.

By using this component in place of the `<a>` anchor element, the counter state is no longer reset. This is because React Router now prevents the browser's default behavior (i.e., the unintended page reload described above) and displays the correct page content.



Now watch the Refresh Button behaviour while your click .You can see that the page is not refreshed

Under the hood, the `Link` component still renders the built-in `<a>` element. But React Router controls it and implements the behavior described above.

The `Link` component is therefore the default component that should be used for internal links. For external links, the standard `<a>` element should be used instead since the link leads away from the website, hence there is no state to preserve or page reload to prevent.

Working with Layouts & Nested Routes

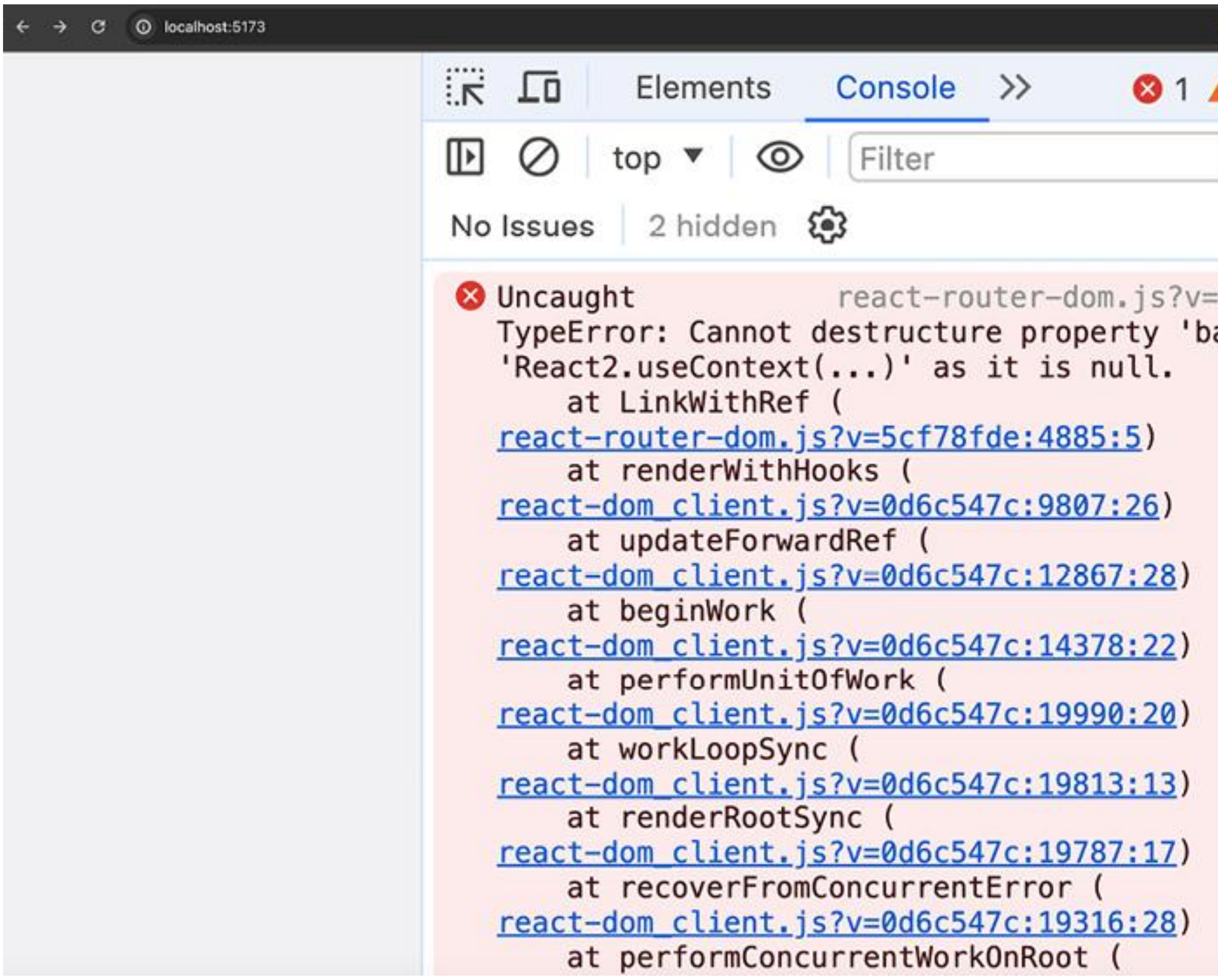
Most websites require some form of page-wide navigation (and hence navigation links) or other page sections that should be shared across some or all routes.

Consider the previous example website with the routes `/` and `/orders`. The example website would also benefit from having a top navigation bar that allows users to switch between the starting page (i.e., the `Dashboard` route) and the `Orders` page.

Therefore, `App.jsx` could be adjusted to have a top navigation bar inside a `<header>` above `<RouterProvider>`:

```
import {
  createBrowserRouter,
  RouterProvider,
  Link
} from 'react-router-dom';
import Dashboard from './routes/Dashboard.jsx';
import Orders from './routes/Orders.jsx';
const router = createBrowserRouter([
  { path: '/', element: <Dashboard /> },
  { path: '/orders', element: <Orders /> },
]);
function App() {
  return (
    <>
     <header>
          <nav>
            <ul>
              <li>
                <Link to="/">My Dashboard</Link>
              </li>
              <li>
                <Link to="/orders">Past Orders</Link>
              </li>
            </ul>
          </nav>
        </header>
     <RouterProvider router={router} />
  </>
  );
}
export default App;
```

But if you try to run this application, you'll see a blank page and encounter an error message in the JavaScript console(Press F12) in the browser developer tools.



```
Uncaught              react-router-dom.js?v=
TypeError: Cannot destructure property 'ba
'React2.useContext(...)' as it is null.
    at LinkWithRef (
react-router-dom.js?v=5cf78fde:4885:5)
    at renderWithHooks (
react-dom_client.js?v=0d6c547c:9807:26)
    at updateForwardRef (
react-dom_client.js?v=0d6c547c:12867:28)
    at beginWork (
react-dom_client.js?v=0d6c547c:14378:22)
    at performUnitOfWork (
react-dom_client.js?v=0d6c547c:19990:20)
    at workLoopSync (
react-dom_client.js?v=0d6c547c:19813:13)
    at renderRootSync (
react-dom_client.js?v=0d6c547c:19787:17)
    at recoverFromConcurrentError (
react-dom_client.js?v=0d6c547c:19316:28)
    at performConcurrentWorkOnRoot (
```

Figure 13.5: React Router seems to complain about something

The error message is a bit cryptic, but the problem is that the above code tries to use `<Link>` outside of a component controlled by React Router.

Only components loaded via `<RouterProvider>` are controlled by React Router, hence React Router features like its `Link` component can only be used in route components (or their descendent components).

Therefore, setting up the main navigation inside of the `App` component (which is **not** loaded by React Router) does not work.

To wrap or enhance multiple route components with some shared component and JSX markup, you must define a new route that wraps the existing routes. Such a route is also sometimes called a **layout route** since it can be used to provide some shared layout. The routes wrapped by this route would be called **nested routes**.

A layout route is defined like any other route inside the route definitions array. It then becomes a layout route by wrapping other routes via a special `children` property that's accepted by React Router. That `children` property receives an array of nested routes—child routes to the wrapping parent route.

Here's the adjusted route definition code for this example app:

```
import Root from './routes/Root.jsx';
import Dashboard from './routes/Dashboard.jsx';
import Orders from './routes/Orders.jsx';
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    children: [
      { index: true, element: <Dashboard /> },
      { path: '/orders', element: <Orders /> },
    ],
  },
]);
```

In this updated code snippet, a new root layout route is defined—a route that registers the existing routes (the `Dashboard` and `Orders` components) as child routes. This setup therefore allows the `Root` component to be active simultaneously to the `Dashboard` or `Orders` route component.

You might also note that the `Dashboard` route no longer has a `path`. Instead, it now has an `index` property, which is set to `true`. That `index` property is a property that can be used when working with nested routes. It tells React Router which nested route to activate (and therefore which component to load) if the parent route path is matched exactly.

In this example, when the `/` path is active (i.e., if a user visits `<domain>/`), the `Root` and `Dashboard` components will be rendered.
For `<domain>/orders`, `Root` and `Orders` would become visible.

The `Root` component is a newly added component in this example. It's a standard component (like `Dashboard` or `Orders`) with one special feature: it defines the place where the child route components should be inserted via a special `Outlet` component that's provided by React Router:

```
import { Link, Outlet } from 'react-router-dom';
function Root() {
  return (
      <>
    <header>
      <nav>
        <ul>
          <li>
            <Link to="/">My Dashboard</Link>
          </li>
          <li>
            <Link to="/orders">Past Orders</Link>
          </li>
        </ul>
      </nav>
    </header>
           <Outlet />
  </>
   );
}
export default Root;
```

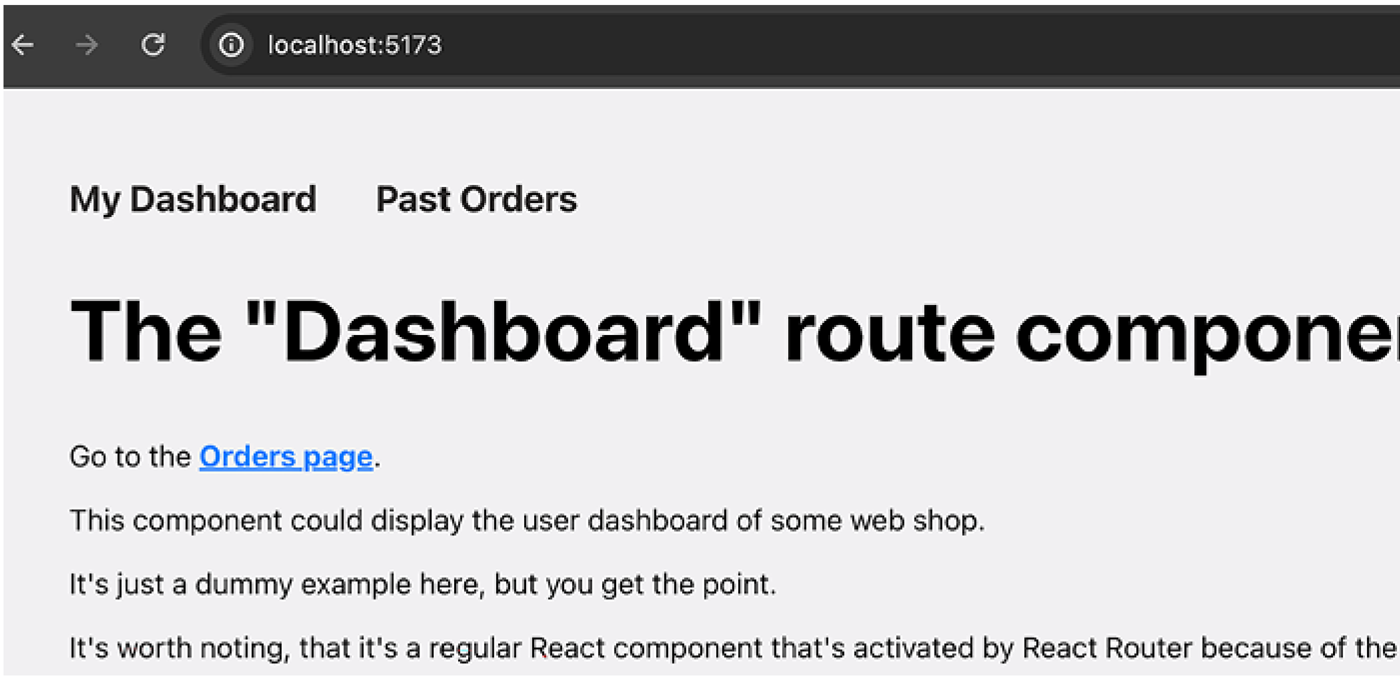
The `<Outlet />` placeholder is needed since React Router must know where to render the route components of the routes passed to the `children` property.

Note

You can find the complete example code on GitHub at

ReactConcepts/examples/05-layouts-nested-routes at main · TalipotTech/ReactConcepts.

Since the `Root` component itself is also rendered by React Router, it now is a component that has access to the `<Link>` tag. Therefore, this `Root` component can be used to share common markup (like the navigation `<header>`) across all nested routes.



My Dashboard Past Orders

The "Dashboard" route compone

Go to the Orders page.

This component could display the user dashboard of some web shop.

It's just a dummy example here, but you get the point.

It's worth noting, that it's a regular React component that's activated by React Router because of the

Figure 13.6: A shared navigation bar is displayed at the top (for all routes)

Hence, nested routes and layout routes (or wrapper routes) are crucial features offered by React Router.

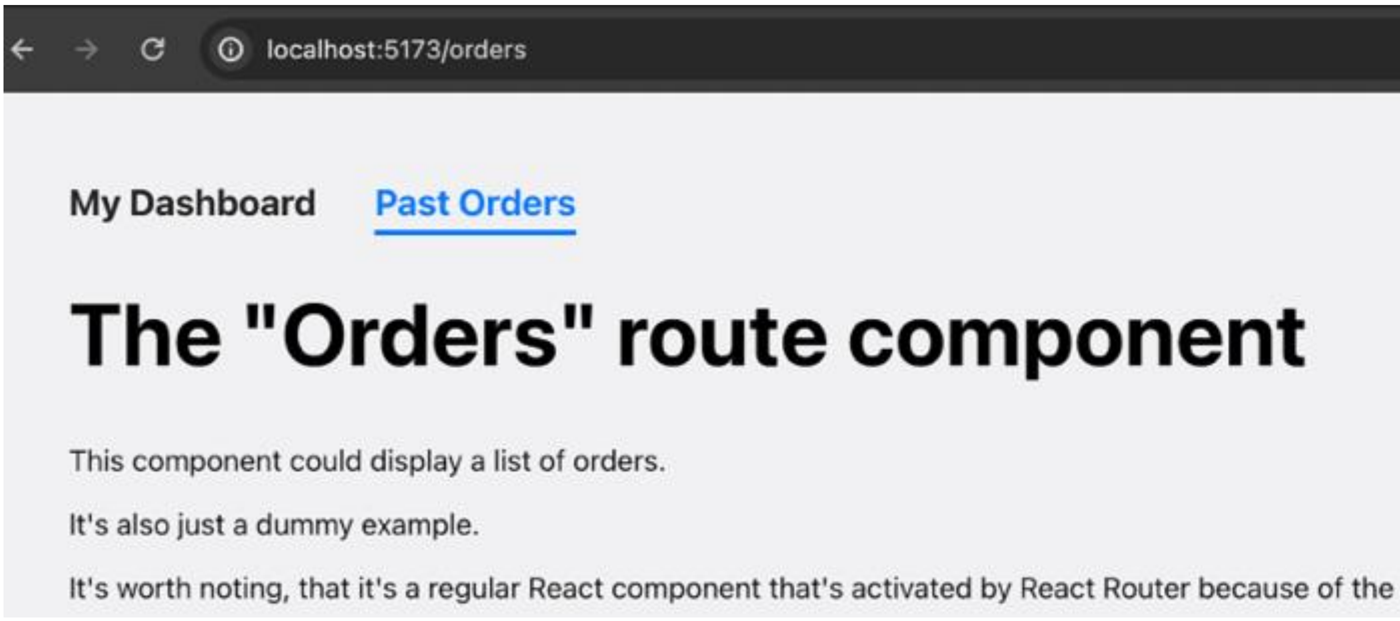
It's also worth noting that you can add as many levels of route nesting as needed by your application—you're **not** restricted to having just one layout route that wraps child routes.

From Link to NavLink

In a shared navigation, you often want to highlight the link that led to the currently active page. For example, if a user clicked the `Past Orders` link (and hence navigates to `/orders`), that link should change its appearance (e.g., its color).

Consider the example from previously (Figure 13.6)—there, in the top navigation bar, it's not immediately obvious whether the user is on the `Dashboard` page or the `Orders` page. Of course, the URL address and the main page content do change, but the navigation items don't adjust visually.

To prove this point, compare the previous screenshot to the following one:


localhost:5173/orders

The "Orders" route component

This component could display a list of orders.

It's also just a dummy example.

It's worth noting, that it's a regular React component that's activated by React Router because of the

Figure 13.7: The highlighted “Past Orders” navigation link is underlined and changes its color

In this version of the website, it’s immediately clear that the user is on the `"Orders"` page since the `Past Orders` navigation link is highlighted. It’s subtle things such as this that make websites more usable and can ultimately lead to higher user engagement.

But how can this be achieved?

To do this, you would not use the `Link` component, but instead, a special alternative component offered by `react-router-dom`: the `NavLink` component:

```
import { NavLink, Outlet } from 'react-router-dom';
function Root() {
  return (
    <>
   <header>
    <nav>
     <ul>
      <li>
       <NavLink to="/">My Dashboard</NavLink>
      </li>
      <li>
                <NavLink to="/orders">Past Orders</NavLink>
      </li>
```

```
        </ul>
      </nav>
    </header>
    <Outlet />
  </>
  ) ;
}
export default Root;
```

Add the css styles to **App.css**

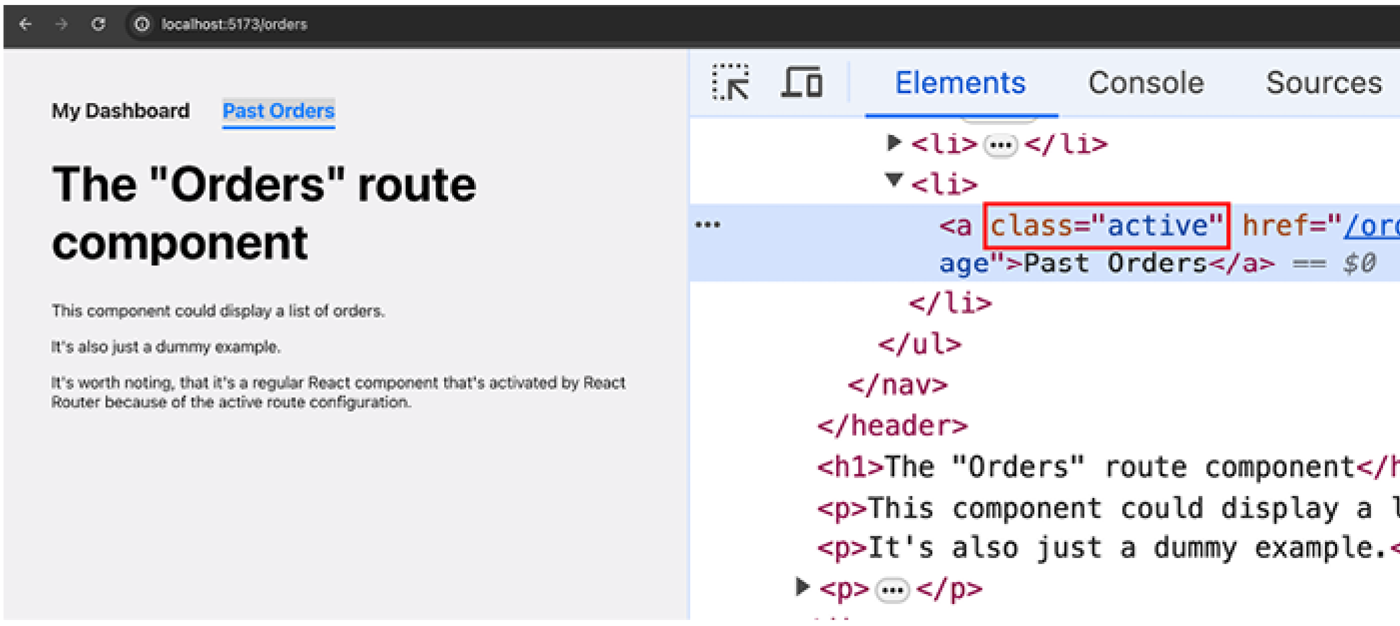
Then in App.tsx include

```
import './App.css'



header {

  margin: 1rem 0;

}



header ul {

  display: flex;

  gap: 2rem;

  list-style: none;

  margin: 0;

  padding: 0;

}
```

```
header a {
  font-size: 1.25rem;
  color: #272525;
  text-decoration: none;
  padding-bottom: 0.25rem;
  border-bottom: 3px solid transparent;
}

header a:hover {
  color: #007bff;
  border-color: #007bff;
}

header a.active,
header a.active:hover {
  color: #007bff;
  border-color: #007bff;
}
a {
  color: #007bff;
  font-weight: bold;
}

a:hover {

  color: #0056b3;

}
```

The `NavLink` component is used pretty much like the `Link` component. You wrap it around some text (the link's caption), and you define the target path via the `to` prop. However, the `NavLink` component has some extra **styling-related features** the regular `Link` component does not have.

To be precise, the `NavLink` component by default applies a CSS class called `active` to the rendered anchor element when the link is active.



The "Orders" route component

This component could display a list of orders.

It's also just a dummy example.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

```
▶<li>⋯</li>
▼<li>
   <a class="active" href="/or
   age">Past Orders</a> == $0
  </li>
 </ul>
</nav>
</header>
<h1>The "Orders" route component</h
<p>This component could display a l
<p>It's also just a dummy example.<
▶<p>⋯</p>
```

Figure 13.8: The rendered <a> element received an "active" CSS class

In case you want to apply a different CSS class name or inline styles when a link becomes active, `NavLink` also allows you to do that.

Because `NavLink`'s `className` and `style` props behave slightly differently than they do on other elements. Besides accepting string values (`className`) or style objects (`style`), both props also accept functions that will automatically be called by React Router upon every navigation action. For example, the following code could be used to ensure that a certain CSS class or style is applied:

```
<NavLink
  className={({ isActive }) => isActive ? 'loaded' : ''}
  style={({ isActive }) => isActive ? { color: 'red' } : undefined}
to="/">
    My Dashboard
</NavLink>

Add the following className to App.css
.loaded
{
  color: blue;
}
```

In the above code snippet, both `className` and `style` take advantage of the function that will be executed by React Router. This function automatically receives an object as an input argument—an object that's created and provided by React Router, and that contains an `isActive` property. React Router sets `isActive` to `true` if the link leads to the currently active route, and to `false` otherwise.

You can therefore return any CSS class names or style objects of your choosing in those functions. React Router will then apply them to the rendered `<a>` element.

Note

You can find the finished code for this example on GitHub at

[ReactConcepts/examples/06-navlinks at main · TalipotTech/ReactConcepts]()

One important note is that `NavLink` will consider a route to be active if its path matches the current URL path or if its path starts with the current URL path. For example, if you had a `/orders/details` route, a `NavLink` component that points at just `/orders` would be considered active if the current route is `/orders/details` (because that route path starts with `/orders`). If you don't want this behavior, you can add the special `end` prop to the `NavLink` component, as follows:

```
<NavLink
  to="/orders"
  style={({ isActive }) => isActive ? { color: 'red' } : undefined}
  end>
    Orders
</NavLink>
```
CopyExplain

With this special prop added, this `NavLink` would only be considered active if the current route is exactly `/orders` —for `/orders /details` , the link would not be active.

An exception from that rule would be links to just `/`. Since all routes technically start with this "empty path," React Router by default only considers `<NavLink to="/">` as active if the user is currently on `<domain>/`. For other paths (e.g., `/orders`), `<NavLink to="/">` would not be marked as active.

Caveats

`NavLink` is always the preferred choice when the styling of a link depends on the currently active route. For all other internal links, use `Link`. For external links, `<a>` is the element of choice.

Route Components versus "Normal" Components

It's worth mentioning and noting that, in the previous examples, the `Dashboard` and `Orders` components were regular React components. You could use these components anywhere in your React app—**not just as values for the `element` property of a route definition**.

However, the two components are special in that both are stored in the `src/routes` folder in the project directory. They are not stored in the `src/components` folder, which was used for the example components we created so far

That's not something you have to do, though. Indeed, the folder names are entirely up to you. These two components could be stored in `src/components`. You could also store them in an `src/elements` folder. But using `src/routes` is quite common for components that are exclusively used for routing. Popular alternatives are `src/screens`, `src/views`, and `src/pages` (again, it is up to you).

If your app includes any other components that are not used as routing elements, you would still store those in `src/components` (i.e., in a different path). This is not a hard rule or a technical requirement, but it does help with keeping your React projects manageable. Splitting your components across multiple folders makes it easier to quickly understand which components fulfill which purposes in the project.

In the example project mentioned previously, you can, for example, refactor the code such that the navigation code is stored in a separate component (e.g., a `MainNavigation` component, stored in `src/components/shared/MainNavigation.jsx`). The component file code looks like this:

```
import { NavLink } from 'react-router-dom';
import classes from './MainNavigation.module.css';
function MainNavigation() {
  return (
      <header className={classes.header}>
    <nav>
      <ul>
        <li>
          <NavLink
            to="/"
            className={({ isActive }) =>
              isActive ? classes.active : undefined
            }
            end
          >
            My Dashboard
          </NavLink>
        </li>
        <li>
          <NavLink
            to="/orders"
            className={({ isActive }) =>
              isActive ? classes.active : undefined
            }
          >
            Past Orders
          </NavLink>
        </li>
      </ul>
    </nav>
  </header>
  );
}
export default MainNavigation;
```

In this code snippet, the `NavLink` component is adjusted to assign a CSS class named `active` to any link that belongs to the currently active route. This is required when using CSS Modules since the class names are changed during the build process. Besides that, it's essentially the same navigation menu code as that used earlier in this chapter.

This `MainNavigation` component can then be imported and used in the `Root.jsx` file like this:

```
import { Outlet } from 'react-router-dom';
import MainNavigation from '../../components/shared/MainNavigation.jsx';
function Root() {
  return (
    <>
      <MainNavigation />
    <Outlet />
  </>
  );
}
export default Root;


MainNavigation.module.css
.header {
  margin: 1rem 0;
}

.header ul {
  display: flex;
  gap: 2rem;
  list-style: none;
  margin: 0;
  padding: 0;
}

.header a {
  font-size: 1.25rem;
  color: #272525;
  text-decoration: none;
  padding-bottom: 0.25rem;
  border-bottom: 3px solid transparent;
}

.header a:hover {
  color: #007bff;
  border-color: #007bff;
}

.header a.active,
.header a.active:hover {
  color: #007bff;
  border-color: #007bff;
}
```

Importing and using the `MainNavigation` component leads to a leaner `Root` component and yet preserves the same functionality as before.

These changes show how you can combine routing components that are only used for routing (`Dashboard` and `Orders`) and components that are used outside of routing (`MainNavigation`).

Note

You can find the finished code for this example on GitHub at ReactConcepts/examples/07-routing-and-normal-cmp at main · TalipotTech/ReactConcepts.

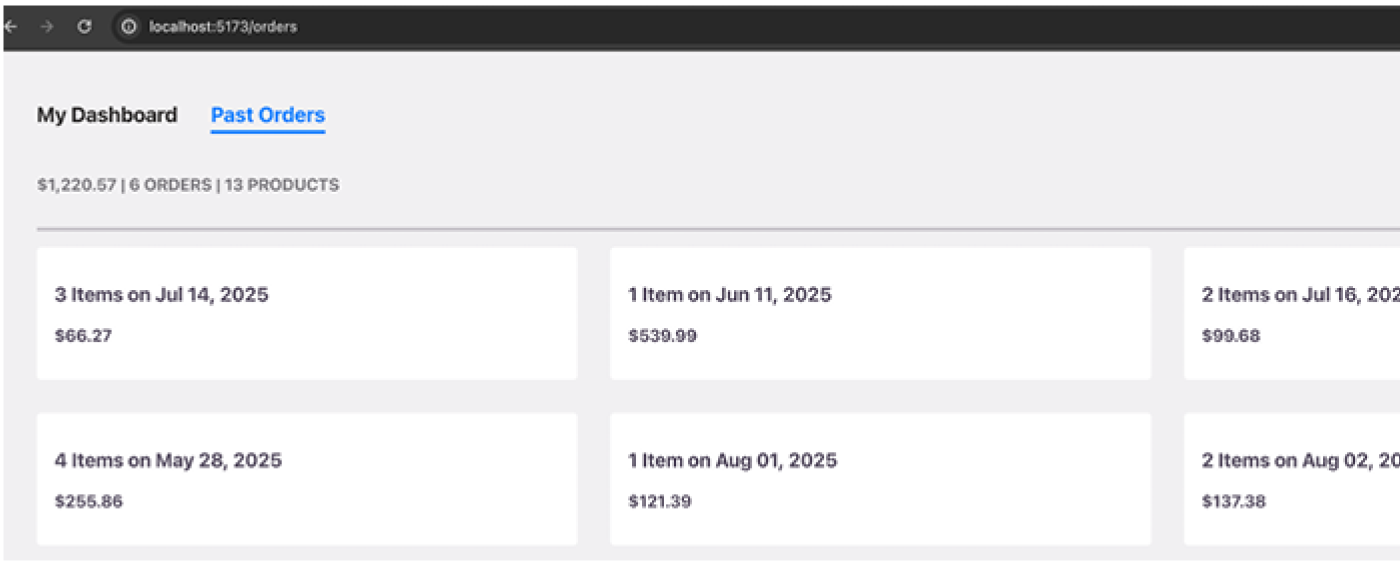
But even with those markup and style improvements, the demo application still suffers from an important problem: it only supports **static, predefined routes**. But, for most websites, those kinds of routes are not enough.

From Static to Dynamic Routes

Thus far, all examples have had two routes: `/` for the `Dashboard` component and `/orders` for the `Orders` component. But you can, of course, add as many routes as needed. If your website consists of 20 different pages, you can (and should) add 20 route definitions (i.e., 20 `Route` components) to your `App` component.

On most websites, however, you will also have some routes that can’t be defined manually—because not all routes and their exact paths are known in advance.

Consider the example, enriched with additional components and some dummy data:


localhost:5173/orders

My Dashboard Past Orders

$1,220.57 | 6 ORDERS | 13 PRODUCTS

3 Items on Jul 14, 2025
$66.27

1 Item on Jun 11, 2025
$539.99

2 Items on Jul 16, 202
$99.68

4 Items on May 28, 2025
$255.86

1 Item on Aug 01, 2025
$121.39

2 Items on Aug 02, 20
$137.38

Figure 13.9: A list of order items

Note

You can find the code for this example on GitHub at ReactConcepts/examples/08-dynamic-routes-problem at main · TalipotTech/ReactConcepts. In the code, you'll notice that many new components and style files were added. The code does not use any new features, though. It's just used to display a more realistic user interface and output some dummy data.

In the preceding screenshot, Figure 13.9, you can see a list of order items being output on the `Past Orders` page (i.e., by the `Orders` component).

In the underlying code, every order item is wrapped with a `Link` component so that a separate page with more details can be loaded for each item:

```
function OrdersList() {
  return (
    <ul className={classes.list}>
   {orders.map((order) => (
     <li key={order.id}>
       <Link to='/orders'><OrderItem order={order} /></Link>
     </li>
   ))}
  </ul>
  );
}
```

In this code snippet, the path for the `Link` component is set to `/orders`. However, that's not the final value that should be assigned. Instead, this example highlights an important problem: while it's the same route and component that should be loaded for every order item (i.e., some component that displays detailed data about the selected order), the exact content output by that component depends on which order item was selected. It's the same route and component with different data.

Outside of routing, you would use props to reuse the same component with different data. But with routing, it's not just about the component. You also must support different paths—because the detailed data for different orders should be loaded via different paths (e.g., `/orders/o1`, `/orders/o2`, etc.). Otherwise, you would again end up with URLs that are not shareable or reloadable.

Therefore, the path must include not only some static identifier (such as `/orders`) but also a dynamic value that's different for every order item. For three order items with `id` values `o1`, `o2`, and `o3`, the goal could be to support the `/orders/o1`, `/orders/o2`, and `/order/o3` paths.

For this reason, the following three route definitions could be added:

```
{ path: '/orders/o1', element: <OrderDetail id="o1" /> },
{ path: '/orders/o2', element: <OrderDetail id="o2" /> },
{ path: '/orders/o3', element: <OrderDetail id="o3" /> }
```

But this solution has a major flaw. Adding all these routes manually is a huge amount of work. And that's not even the biggest problem. You typically don't even know all values in advance. In this example, when a new order is placed, a new route would have to be added. But you can't adjust the source code of your website every time a visitor places an order.

Clearly, then, a better solution is needed. React Router offers that better solution as it supports **dynamic routes**.

Dynamic routes are defined just like other routes, except that, when defining their `path` values, you will need to include one or more **dynamic path segments** with identifiers of your choice.

The `OrderDetail` route definition therefore looks like this:

```
{ path: '/orders/:id', element: <OrderDetail /> }
```

The following three key things have changed:

- It's just one route definition instead of a (possibly) infinite list of definitions.
- `path` contains a dynamic path segment (`:id`).
- `OrderDetail` no longer receives an `id` prop.

The `:id` syntax is a special syntax supported by React Router. Whenever a segment of a path starts with a colon, React Router treats it as a **dynamic segment**. That means that it will be replaced with a different value in the actual URL path. For the `/orders/:id` route path, the `/orders/o1`, `/orders/o2`, and `/orders/abc` paths would all match and therefore activate the route.

Of course, you don't have to use `:id`. You can use any identifier of your choice. For the preceding example, `:orderId`, `:order`, or `:oid` would also make sense.

The identifier will help your app access the correct data inside the page component that should be loaded for the dynamic route (i.e., the `OrderDetail` route component in the example code snippets above). That's why the `id` prop was removed from `OrderDetail` in the last code snippet. Since only one route is defined, only one specific `id` value could be passed via props. That won't help. Therefore, a different way of loading order-specific data must be used.

Extracting Route Parameters

In the previous example, when a website user visits `/orders/o1` or `/orders/o2` (or the same path for any other order ID), the `OrderDetail` component is loaded. This component should then output more information about the specific order that was selected (i.e., the order whose ID is encoded in the URL path).

By the way, that's not just the case for this example; you can think of many other types of websites as well. You could also have, for example, an online shop with routes for products (`/products/p1`, `/products/p2`, etc.), or a travel blog where users can visit individual blog posts (`/blog/post1`, `/blog/post2`, etc.).

In all these cases, the question is how do you get access to the data that should be loaded for the specific identifier (e.g., the ID) that's included in the URL path? Since it's always the same component that's loaded, you need a way of dynamically identifying the order, product, or blog post for which the detail data should be fetched.

One possible solution would be the usage of props. Whenever you build a component that should be reusable yet configurable and dynamic, you can use props to accept different values. For example, the `OrderDetail` component could accept an `id` prop and then, inside the component function body, load the data for that specific order ID.

However, as mentioned in the previous section, this is not a possible solution when loading the component via routing. **Keep in mind that the `OrderDetail` component is created when defining the route:**

```
{ path: '/orders/:id', element: <OrderDetail /> }
```

Since the component is created when defining the route in the App component, you can't pass in any dynamic, ID-specific prop values.

Fortunately, though, that's not necessary. React Router gives you a solution that allows you to extract the data encoded in the URL path from inside the component that's displayed on the screen (when the route becomes active): the `useParams()` Hook.

This Hook can be used to get access to the route parameters of the currently active route. Route parameters are simply the dynamic values encoded in the URL path—`id`, in the case of this `OrderDetail` example.

Inside the `OrderDetail` component, `useParams()` can therefore be used to extract the specific order ID and load the appropriate order data, as follows:

```
import { useParams } from 'react-router-dom';
import Details from '../components/orders/Details.jsx';
import { getOrderById } from '../data/orders.js';
function OrderDetail() {
  const params = useParams();
  const orderId = params.id; // orderId is "o1", "o2" etc.
  const order = getOrderById(orderId);
```

```
   return  <Details order={order} />;
}
export  default  OrderDetail;
```

As you can see in this snippet, `useParams()` returns an object that contains all route parameters of the currently active route as properties. Since the route path was defined as `/orders/:id`, the `params` object contains an `id` property. The value of that property is then the actual value encoded in the URL path (e.g., `o1`). If you choose a different identifier name in the route definition (e.g., `/orders/:orderId` instead of `/orders/:id`), that property name must be used to access the value in the `params` object (i.e., access `params.orderId`).

Note

You can find the complete code on GitHub at

ReactConcepts/examples/09-dynamic-routes at main · TalipotTech/ReactConcepts.

By using route parameters, you can thus easily create dynamic routes that lead to different data being loaded. But, of course, defining routes and handling route activation are not that helpful if you do not have links leading to dynamic routes.

Creating Dynamic Links

As mentioned earlier in this chapter (in the Adding Page Navigation section), website visitors should be able to click on links that should then take them to the different pages that make up the overall website—meaning, those links should activate the various routes defined with the help of React Router.

As explained in the Adding Page Navigation and From Link to NavLink sections, for internal links (i.e., links leading to routes defined inside the React app), the `Link` or `NavLink` components are used.

So, for static routes such as `/orders`, links are created like this:

```
<Link to="/orders">Past Orders</Link> // or use <NavLink> instead
```

When building a link to a dynamic route such as `/orders/:id`, you can therefore simply create a link like this:

```
<Link to="/orders/o1">Past Orders</Link>
```

This specific link loads the `OrderDetails` component for the order with the ID `o1`.

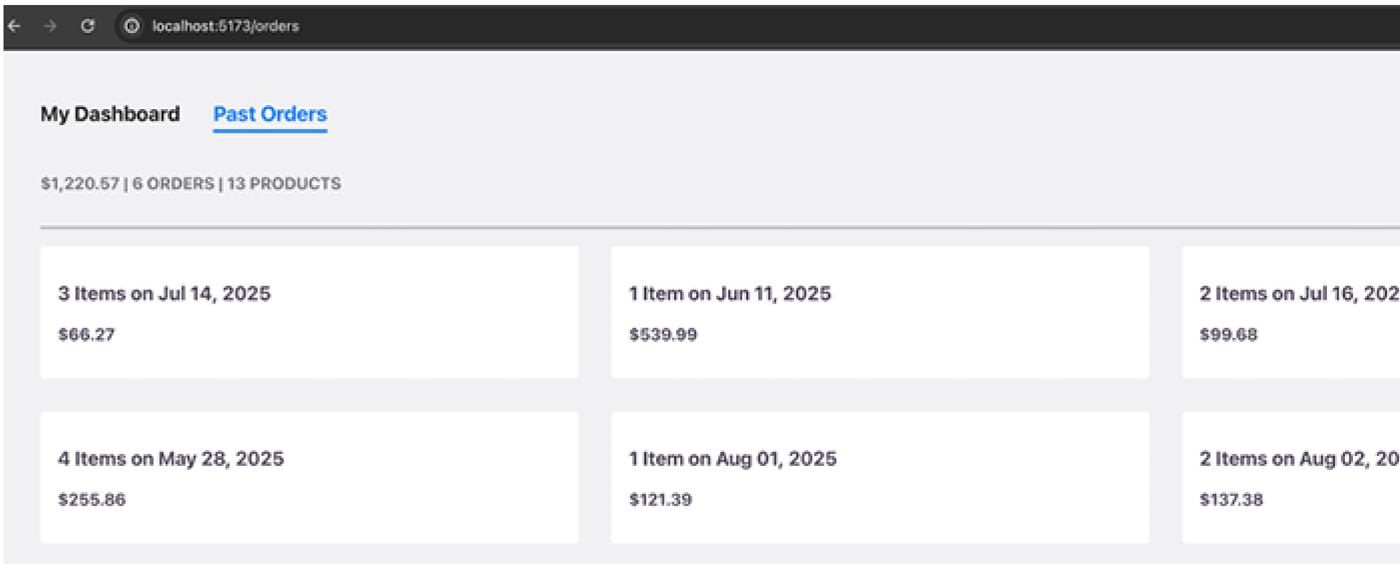
Building the link as follows would be incorrect:

```
<Link to="/orders/:id">Past Orders</Link>
```

The dynamic path segment syntax (`:id`) is only used when defining the route—not when creating a link. The link has to lead to a specific resource (a specific order, in this case).

However, creating links to specific orders, as shown previously, is not very practical. Just as it wouldn't make sense to define all dynamic routes individually (see the From Static to Dynamic Routes section), it doesn't make sense to create the respective links manually.

Sticking to the orders example, there is also no need to create links like that as you already have a list of orders that's output on one page (the `Orders` component, in this case). Similarly, you could have a list of products in an online shop. In all these cases, the individual items (orders, products, etc.) should be clickable and lead to details pages with more information.



$1,220.57 | 6 ORDERS | 13 PRODUCTS

3 Items on Jul 14, 2025
$66.27

1 Item on Jun 11, 2025
$539.99

2 Items on Jul 16, 202
$99.68

4 Items on May 28, 2025
$255.86

1 Item on Aug 01, 2025
$121.39

2 Items on Aug 02, 20
$137.38

Figure 13.10: A list of clickable order items

Therefore, the links can be generated dynamically when rendering the list of JSX elements. In the case of the orders example, the code looks like this:

```
function  OrdersList( )  {
   return  (
      <ul className={classes.list}>
    {orders.map((order) => (
      <li key={order.id}>
        <Link
                    to={`/orders/${order.id}`}>
                    <OrderItem order={order} />
        </Link>
      </li>
    ))}
   </ul>
   ) ;
}
```

In this code example, the value of the `to` prop is set dynamically equal to a string that includes the `order.id` value. Therefore, every list item receives a unique link that leads to a different details page. Or, to be precise, the link always leads to the same component but with a different order `id` value, hence loading different order data.

Note

In this code snippet (which can be found at ReactConcepts/examples/10-dynamic-links at main · TalipotTech/ReactConcepts), the string is created as a **template literal**. That's a default JavaScript feature that simplifies the creation of strings that include dynamic values.

You can learn more about template literals on MDN at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals.

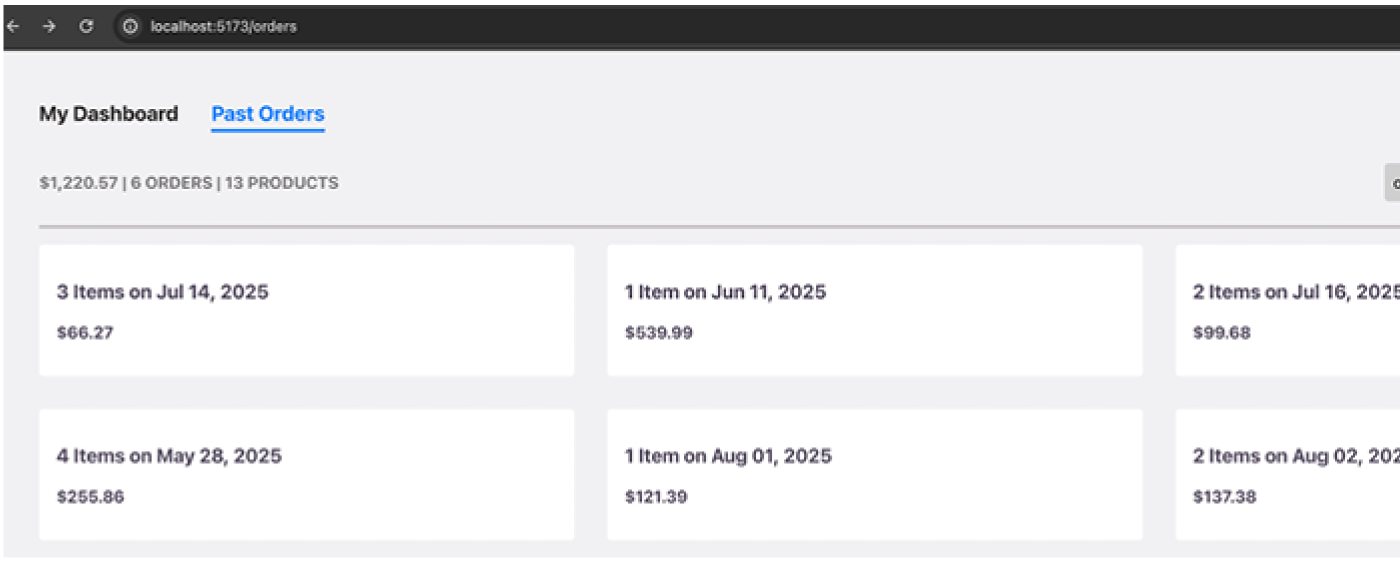
Navigating Programmatically

In the previous section, as well as earlier in this chapter, user navigation was enabled by adding links to the website. Indeed, links are the default way of adding navigation to a website. But there are scenarios where programmatic navigation is required instead.

Programmatic navigation means that a new page should be loaded via JavaScript code (rather than using a link). This kind of navigation is typically required if the active page changes in response to some action—e.g., upon form submission.

If you take the example of form submission, you will normally want to extract and save the submitted data. But thereafter, the user will sometimes need to be redirected to a different page. For example, it makes no sense to keep the user on a `Checkout` page after processing the entered credit card details. You might want to redirect the user to a `Success` page instead.

In the example discussed throughout this chapter, the `Past Orders` page could include an input field that allows users to directly enter an order ID and load the respective order data after clicking the `Find` button.


localhost:5173/orders

My Dashboard Past Orders

$1,220.57 | 6 ORDERS | 13 PRODUCTS

3 Items on Jul 14, 2025
$66.27

1 Item on Jun 11, 2025
$539.99

2 Items on Jul 16, 2025
$99.68

4 Items on May 28, 2025
$255.86

1 Item on Aug 01, 2025
$121.39

2 Items on Aug 02, 202
$137.38

Figure 13.11: An input field that can be used to quickly load a specific order

In this example, the entered order ID is first processed and validated before the user is sent to the respective details page. If the provided ID is invalid, an error message is shown instead. The code looks like this:

```
import orders, { getOrdersSummaryData } from '../../data/orders.js';
import classes from './OrdersSummary.module.css';
function OrdersSummary() {
  const { quantity, total } = getOrdersSummaryData();
  const formattedTotal = new Intl.NumberFormat('en-US', {
    style: 'currency',
    currency: 'USD',
  }).format(total);
  function findOrderAction(formData) {
    const orderId = formData.get('order-id');
    const orderExists = orders.some((order) => order.id ===
orderId);
    if (!orderExists) {
      alert('Could not find an order for the entered id.');
      return;
    }
  }
  return (
    <div className={classes.row}>
   <p className={classes.summary}>
     {formattedTotal} | {orders.length} Orders |
      {quantity} Products
   </p>
   <form className={classes.form} action={findOrderAction}>
     <input
      type="text"
      name="order-id"
      placeholder="Enter order id"
      aria-label="Find an order by id."
     />
     <button>Find</button>
   </form>
  </div>
  );
}
export default OrdersSummary;
```

The code snippet does not yet include the code that will actually trigger the page change, but it does show how the user input is read and validated.

Therefore, this is a perfect scenario for the use of programmatic navigation. A link can’t be used here since it would immediately trigger a page change—without allowing you to validate the user input first (at least not after the link was clicked).

The React Router library also supports programmatic navigation for cases like this. You can import and use the special `useNavigate()` Hook to gain access to a navigation function that can be used to trigger a navigation action (i.e., a page change):

```
import { useNavigate } from 'react-router-dom';
const navigate = useNavigate();
navigate('/orders');
// programmatic alternative to <Link to="/orders">
```

Hence, the `OrdersSummary` component from previously can be adjusted like this to use this new Hook:

```
function OrdersSummary() {
  const navigate = useNavigate();
  const { quantity, total } = getOrdersSummaryData();
  const formattedTotal = new Intl.NumberFormat('en-US', {
    style: 'currency',
    currency: 'USD',
  }).format(total);
  function findOrderAction(formData) {
    const orderId = formData.get('order-id');
    const orderExists = orders.some((order) => order.id ===
orderId);
    if (!orderExists) {
      alert('Could not find an order for the entered id.');
      return;
    }
    navigate(`/orders/${orderId}`);
  }
  // returned JSX code did not change, hence omitted
}
```

It's worth noting that the value passed to `navigate()` is a dynamically constructed string. Programmatic navigation supports both static and dynamic paths.

Note

The code for this example can be found at [ReactConcepts/examples/11-programmatic-navigation at main · TalipotTech/ReactConcepts]

Redirecting

Thus far, all the explored navigation options (links and programmatic navigation) forward a user to a specific page.

In most cases, that's the intended behavior. But in some cases, the goal is to redirect a user instead of forwarding them.

The difference is subtle but important. When a user is forwarded, they can use the browser's navigation buttons (`Back` and `Forward`) to go back to the previous page or forward to the page they came from. For redirects, that's not possible. Whenever a user is redirected to a specific page (rather than forwarded), they can't use the `Back` button to return to the previous page.

Redirecting users can, for example, be useful for ensuring that users can't go back to a login page after authenticating successfully.

When using React Router, the default behavior is to forward users. But you can easily switch to redirecting by adding the special `replace` prop to the `Link` (or `NavLink`) components, as follows:

```
<Link to="/success" replace>Confirm Checkout</Link>
```

When using programmatic navigation, you can pass a second, optional argument to the `navigate()` function. That second parameter value must be an object that can contain a `replace` property that should be set to `true` if you want to redirect users:

```
navigate('/dashboard', { replace: true });
```

Being able to redirect or forward users allows you to build highly user-friendly web applications that offer the best possible user experience for different scenarios.

Handling Undefined Routes

Previous sections in this chapter have all assumed that you have predefined routes that should be reachable by website visitors. But what if a visitor enters a URL that's simply not supported?

For example, the demo website used throughout this chapter supports the `/`, `/orders`, and `/orders/<some-id>` paths. But it does not support `/home`, `/products/p1`, `/abc`, or any other path that's not one of the defined route paths.

To show a custom Not Found page, you can define a "catch all" route with a special path—the `*` path:

```
{ path: '*', element: <NotFound /> }
```

When adding this route to the list of route definitions in the `App` component, the `NotFound` component will be displayed on the screen when no other route matches the entered or generated URL path.

Lazy Loading

Lazy loading is a technique that can be used to load certain pieces of the React application code only when needed.

Code splitting makes a lot of sense if some components will be loaded conditionally and may not be needed at all. Hence, routing is a perfect scenario for lazy loading. When applications have multiple routes, some routes may never be visited by a user. Even if all routes are visited, not all the code for all app routes (i.e., for their components) must be downloaded right at the start when the application loads. Instead, it makes sense to only download code for individual routes when they actually become active.

Thankfully, React Router has built-in support for lazy loading and route-based code splitting. It provides a `lazy` property that can be added to a route definition. That property expects a function that dynamically imports the lazily loaded file (which contains the component that should be rendered). React Router then takes care of the rest—for example, you don't need to wrap `Suspense` around any components:

```
import {
  createBrowserRouter,
  RouterProvider
} from 'react-router-dom';
import Root from './routes/Root.jsx';
import Dashboard from './routes/Dashboard.jsx';
```

```
// Removed static imports of Orders.jsx and OrderDetail.jsx
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    children: [
      { index: true, element: <Dashboard /> },
      {
        path: '/orders',
        lazy: () => import('./routes/Orders.jsx')
      },
      {
        path: '/orders/:id',
        lazy: () => import('./routes/OrderDetail.jsx')
      },
    ],
  },
]);
function App() {
  return <RouterProvider router={router} />;
}
export default App;
```

In this example, both the `/orders` and `/orders/:id` routes are set up to load their respective components lazily.

For the above code to work, there's one important adjustment you must apply to your route component files when using this built-in lazy-loading support: you must replace the default component function export (`export default SomeComponent`) with a named export where the component function is named `Component`.

For example, the `Orders` component code needs to be changed to look like this:

```
import OrdersList from '../components/orders/OrdersList.jsx';
import OrdersSummary from '../components/orders/OrdersSummary.jsx';
function Orders() {
  return (
    <>
   <OrdersSummary />
   <OrdersList />
  </>
  );
}
export const Component = Orders;  // named export as "Component"
```

In this code snippet, the `Orders` component function is exported as `Component`. This name is required since React Router looks for a component function named `Component` when activating a lazy-loaded route.

Note

The code for this example can be found at

ReactConcepts/examples/12-lazy-loading at main · TalipotTech/ReactConcepts.

Adding lazy loading can improve your React application's performance considerably. You should always consider using lazy loading, but you should not use it for every route. It would be especially illogical for routes that are guaranteed to be loaded early, for instance. In the previous example, it would not make too much sense to lazy load the `Dashboard` component since that's the default route (with a path of `/`).

But routes that are not guaranteed to be visited at all (or at least not immediately after the website is loaded) are great candidates for lazy loading.

Summary and Key Takeaways

- Routing is a key feature for many React apps.
- With routing, users can visit multiple pages despite being on an **SPA**.
- The most common package that helps with routing is the React Router library (`react-router-dom`).
- Routes are defined with the help of the `createBrowserRouter()` function and the `RouterProvider` component (typically in the `App` component or the `main.jsx` file, but you can do it anywhere).
- Route definition objects are typically set up with a `path` (for which the route should become active) and an `element` (the content that should be displayed) property.

- Content and markup can be shared across multiple routes by setting up layout routes—i.e., routes wrapping other nested routes.
- Users can navigate between routes by manually changing the URL path, by clicking links, or because of programmatic navigation.
- Internal links (i.e., links leading to application routes defined by you) should be created via the `Link` or `NavLink` components, while links to external resources use the standard `<a>` element.
- Programmatic navigation is triggered via the `navigate()` function, which is yielded by the `useNavigate()` Hook.
- You can define static and dynamic routes: static routes are the default, while dynamic routes are routes where the path (in the route definition) contains a dynamic segment (denoted by a colon, e.g., `:id`).
- The actual values for dynamic path segments can be extracted via the `useParams()` Hook.
- You can use lazy loading to load route-specific code only when the route is actually visited by the user.

What's Next?

Routing is a feature that's not supported by React out of the box but still matters for most React applications. That's why it's included in this book and why the React Router library exists. Routing is a crucial concept that completes your knowledge about the most essential React ideas and concepts, allowing you to build both simple and complex React applications.

The next chapter builds upon this chapter and dives even deeper into React Router, exploring its data fetching and manipulation capabilities.

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 ReactConcepts/exercises/questions-answers.md at main · TalipotTech/ReactConcepts:

1. How is routing different from loading content conditionally?
2. How are routes defined?
3. How should you add links to different routes to your pages?
4. How can dynamic routes (e.g., details for one of many products) be added to your app?
5. How can dynamic route parameter values be extracted (e.g., to load product data)?
6. What's the purpose of nested routes?

Apply What You Learned

Apply your knowledge about routing to the following activities.

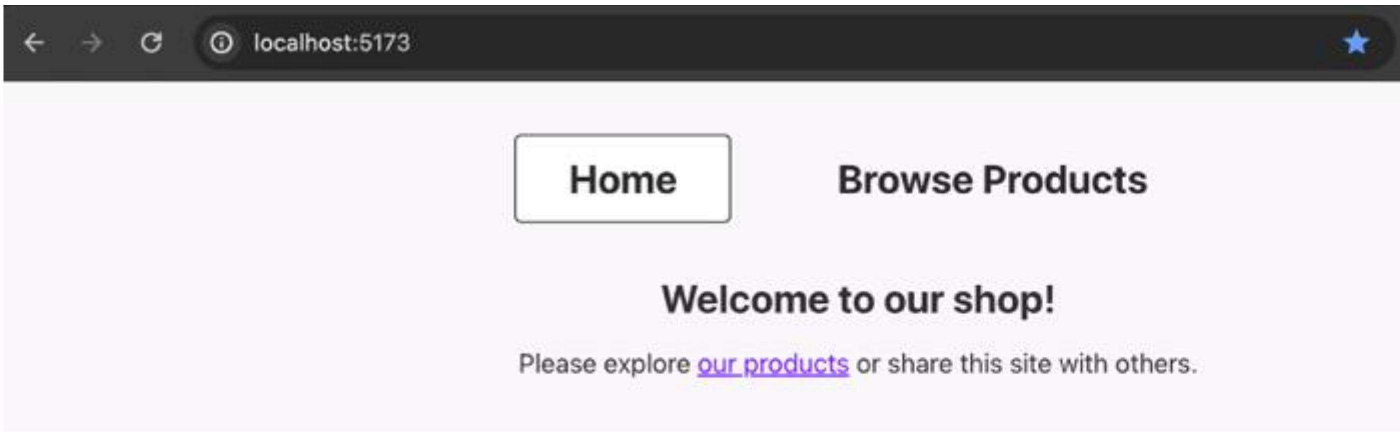
Activity 13.1: Creating a Basic Three-Page Website

In this activity, your task is to create a very basic first draft for a brand-new online shop website. The website must support three main pages:

- A welcome page
- A products overview page that shows a list of available products
- A product details page, which allows users to explore product details

Final website styling, content, and data will be added by other teams, but you should provide some dummy data and default styling. You must also add a shared main navigation bar at the top and implement route-based lazy loading.

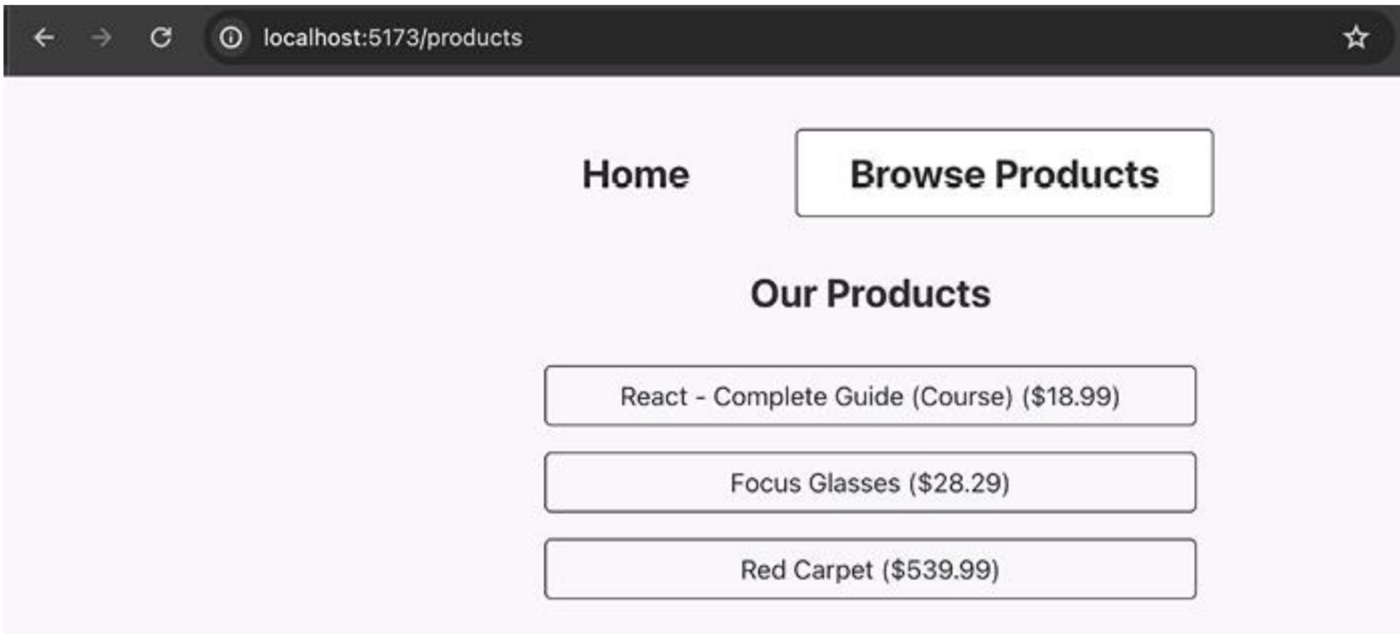
The finished pages should look like this:



Welcome to our shop!

Please explore our products or share this site with others.

Figure 13.12: The welcome page.



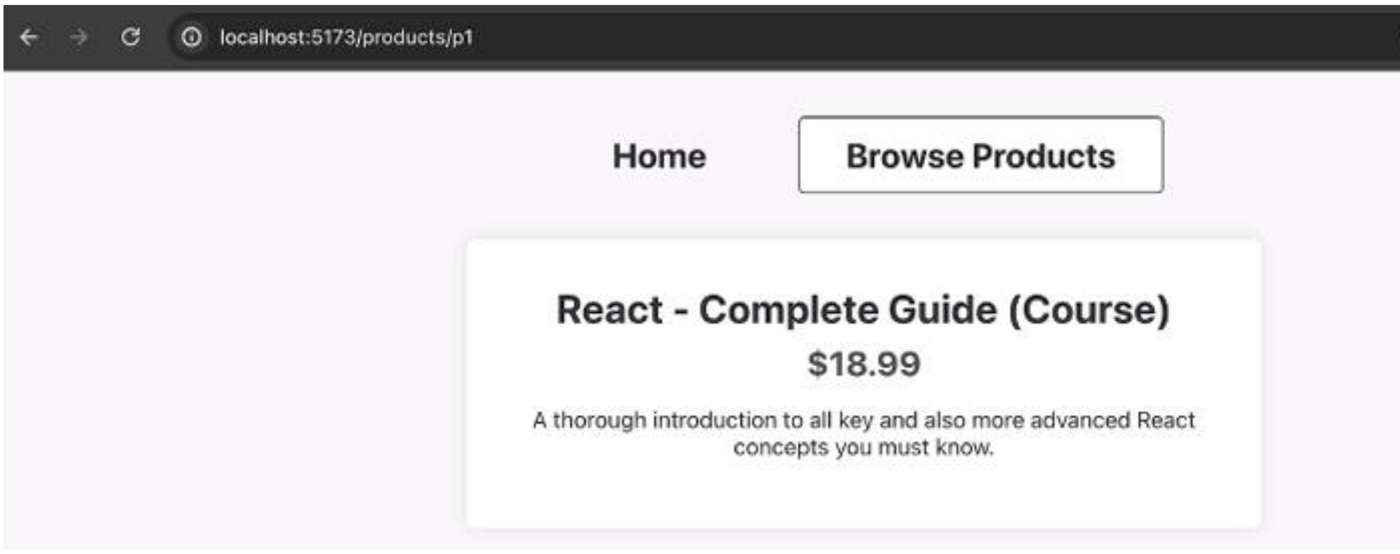
Our Products

React - Complete Guide (Course) ($18.99)

Focus Glasses ($28.29)

Red Carpet ($539.99)

Figure 13.13: A page showing some dummy product placeholders



React - Complete Guide (Course)

$18.99

A thorough introduction to all key and also more advanced React concepts you must know.

Figure 13.14: The final product details page with some placeholder data and styles

Note

For this activity, you can, of course, write all CSS styles on your own. But if you want to focus on the React and JavaScript logic, you can also use the finished CSS file from the solution at ReactConcepts/activities/practice-1/src/index.css at main · TalipotTech/ReactConcepts.

If you use that file, explore it carefully to ensure you understand which IDs or CSS classes might need to be added to certain JSX elements of your solution. You can also use the solution's dummy data instead of creating your own dummy product data. You will find the data for this at ReactConcepts/activities/practice-1/src/data/products.js at main · TalipotTech/ReactConcepts.

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

1. Create a new React project and install the React Router package.
2. Create components (with the content shown in the preceding screenshot) that will be loaded for the three required pages.
3. Enable routing and add the route definitions for the three pages.
4. Add a main navigation bar that's visible for all pages.
5. Add all required links and ensure that the navigation bar links reflect whether or not a page is active.
6. Implement lazy loading (for routes where it makes sense).

Note

The full code, and solution, for this activity can be found here: ReactConcepts/activities/practice-1 at main · TalipotTech/ReactConcepts

MANAGING DATA WITH REACT ROUTER

This section covers how to use React Router to manage data fetching and submission in React applications. It explains the tight coupling between data fetching and routing, and introduces the loader() function and useLoaderData() Hook provided by React Router to simplify data fetching. The section also discusses how to handle dynamic routes, share data across nested routes, and use layout routes to manage shared components and data. Additionally, it touches on the request object and its use in loaders, and emphasizes the importance of understanding that all code executes on the client side.

Learning Objectives

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

- Use React Router to fetch or send data without using `useEffect()` or `useState()`
- Share data between different routes without using React's context feature
- Update the UI based on the current data submission status
- Create page and action routes
- Improve the user experience by deferring the loading of non-critical data

Data Fetching and Routing Are Tightly Coupled

As mentioned previously, most websites do need to fetch (or send) data, and most websites do need more than one page. But it's important to realize that these two concepts are typically closely related.

Whenever a user visits a new page (such as `/posts`), it's likely that some data will need to be fetched. In the case of a `/posts` page, the required data is probably a list of blog posts that is retrieved from a backend server. The rendered React component (such as `Posts`) must therefore send an HTTP request to the backend server, wait for the response, handle the response (as well as potential errors), and, ultimately, display the fetched data.

Of course, not all pages need to fetch data. Landing pages, "About Us" pages, and "Terms & Use" pages probably don't need to fetch data when a user visits them. Instead, data on those pages is likely to be static. It might even be included in the source code as it doesn't change frequently.

But many pages do need to get data from a **backend every time** they're loaded—for instance, "Products," "News," "Events," or other infrequently updated pages like the "User Profile."

And data fetching isn't everything. Most websites also contain features that require data submission—be it a blog post that can be created or updated, product data that's administered, or a user comment that can be added. Hence, sending data to a backend is also a very common use case.

And beyond requests, components might also need to interact with other browser APIs, such as `localStorage`. For example, user settings might need to be fetched from storage as a certain page loads.

Naturally, all these interactions happen on pages. But it might not be immediately obvious how tightly data fetching and submission are coupled to routing.

Most of the time, **data is fetched when a route becomes active**, i.e., when a component (the page component) is rendered for the first time. Sure, users might also be able to click a button to refresh the data, but while this is optional, data fetching upon initial page load is almost always required.

And when it comes to sending data, there is also a close connection to routing. At first sight, it's not clear how it's related because, while it makes sense to fetch data upon page load, it's less likely that you will need to send some data immediately (except perhaps tracking or analytics data).

But it's very likely that after sending data, you will want to navigate to a different page, meaning that it's actually the other way around, and instead of initiating data fetching as a page loads, you want to load a different page after sending some data.

For example, after an administrator enters some product data and submits the form, they should typically be redirected to a different page (for example, from `/products/new` to the `/products` page).

The connection between data fetching, submission, and routing can therefore be summarized by the following points:

- **Data fetching** often should be initiated when a route becomes active (if that page needs data)
- After **submitting data**, the user should often be redirected to another route

Because these concepts are tightly coupled, React Router provides extra features that vastly simplify the process of working with data.

Sending HTTP Requests without React Router

Working with data is not just about sending HTTP requests. As mentioned in the previous section, you may also need to store or retrieve data via `localStorage` or perform some other operation as a page gets loaded. But sending HTTP requests is an especially common scenario and will therefore be the main use case considered for the majority of this chapter. Nonetheless, it's vital to keep in mind that what you learn in this chapter is not limited to sending HTTP requests.

As you will see, React Router provides various features that help with sending HTTP requests (or using other data fetching and manipulation APIs), but you can also send HTTP requests (or interact with `localStorage` or other APIs) without these features. In topic on, Handling Side Effects, already taught you how HTTP requests can be sent from inside React components with the help of `useEffect()`.

When using React Router's data fetching capabilities, you can get rid of `useEffect()` and manual state management.

Note

Besides starting how data fetching with useEffect() works via this code example on GitHub: . ReactConcepts/examples/01-data-fetching-classic at manage-router-data · TalipotTech/ReactConcepts

Loading Data with React Router

With React Router, fetching data can be simplified down to this, shorter, code snippet:

```
import { useLoaderData } from 'react-router-dom';
function Posts() {
  const loadedPosts = useLoaderData();
  return (
    <main>
   <h1>Your Posts</h1>
   <ul className="posts">
    {loadedPosts.map((post) => (
     <li key={post.id}>{post.title}</li>
    ))}
   </ul>
  </main>
  );
}
export default Posts;
export async function loader() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```

Believe it or not, it really is that much less code than in the examples of useEffect(). Back then, when using `useEffect()`, separate state slices(const [loadedPosts, setLoadedPosts] = useState(); const [isLoading, setIsLoading] = useState(false);

const [error, setError] = useState();) had to be managed to handle loading and error states as well as the received data. Though, to be fair, the content that should be displayed in case of an error is missing here. It's in a separate file (which will be shown later), but it would only add three extra lines of code.

In the preceding code snippet, you see a couple of new features .
The `loader()` function and the `useLoaderData()` Hook are added by React Router. These features, along with many others that will be explored throughout this chapter, are made available by the React Router package.

With that library installed, you can set an extra `loader` prop on your route definitions. This prop accepts a function that will be executed by React Router whenever this route (or one of its child routes, if defined) is activated:

```
{ path: '/posts', element: <Posts />, loader: () => {...} }
```

This function can be used to perform any data fetching or other tasks required to successfully display the page component. The logic for getting that required data can therefore be extracted from the component and moved into a separate function.

Since many websites have dozens or even hundreds of routes, adding these loader functions inline (as arrow functions as shown above **() => {...}**) in the route definition objects quickly leads to complex and confusing route definitions. For this reason, you will typically add (and export) the `loader()` function in the same file that contains the component that needs the data.

When setting up the route definitions, you can then import the component and its `loader` function and use it like this:

```
import Posts, { loader as postsLoader } from './components/Posts.jsx';
// … other code …
const router = createBrowserRouter([
  { path: '/posts', element: <Posts />, loader: postsLoader }
]);
```
CopyExplain

Assigning an alias (`postsLoader`, in this example) to the imported `loader` function is optional but recommended since you most likely have multiple `loader` functions from different components, which would otherwise lead to name clashes.

Note

Technically, you don't need to name your functions `loader`. You could use any name and assign them as values for the `loader` property in the route definition.

But using `loader` as a function name does not just follow the convention; it also has the advantage that React Router's built-in lazy loading support (covered in the previous chapter) lazy-loads the `loader` function when needed. It fails to do that if you pick any other name.

With this `loader` defined, React Router will execute the `loader()` function whenever a route is activated. To be precise, the `loader()` function is called before the component function is executed (that is, before the component is rendered).

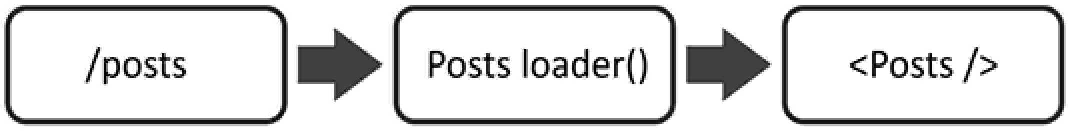


Figure 14.1: The Posts component is rendered after the loader is executed

This also explains why the `Posts` component example at the beginning of this section contained no code that handled any loading state. There simply is no loading state since a component function is only executed after its loader has finished (and the data is available). React Router won't finish the page transition until the `loader()` function has finished its job (though, as you will learn toward the end of this chapter, there is a way of changing this behavior).

The `loader()` function can perform any operation of your choice (such as sending an HTTP request, or reaching out to browser storage via the `localStorage` API). Inside that function, you should return the data that should be exposed to the component function. It's also worth noting that the `loader()` function can return any kind of data. It may also return a `Promise` object that then resolves to any kind of data. In that case, React Router will automatically wait for the `Promise` to be fulfilled before executing the related route component function. The `loader()` function can thus perform both asynchronous and synchronous tasks.

Note

It's important to understand that the `loader()` function, like all the other code that makes up your React app, executes on the client side (that is, in the browser of a website visitor). Therefore, you may perform any action that could be performed anywhere else (for example, inside `useEffect()`) in your React app as well.

You must not try to run code that belongs to the server side. Directly reaching out to a database, writing to the file system, or performing any other server-side tasks will fail or introduce security risks, meaning that you might accidentally expose database credentials on the client side.

Getting Access to Loaded Data

Of course, the component that belongs to a `loader` (that is, the component that's part of the same route definition) needs the data returned by the `loader`. This is why React Router offers a new Hook for accessing that data: the `useLoaderData()` Hook.

When called inside a component function, this Hook yields the data returned by the `loader` that belongs to the active route. If that returned data is a `Promise`, React Router (as mentioned earlier) will automatically wait for that `Promise` to resolve and provide the resolved data when `useLoaderData()` is called.

The `loader()` function may also return an HTTP response object (or a `Promise` resolving to a `Response`) object. This is the case in the preceding example because the `fetch()` function yields a `Promise` that resolves to an object of type `Response`. In that instance, React Router automatically extracts the response body and provides direct access to the data that was attached to the response (via `useLoaderData()`).

Note

If a response should be returned, the returned object must adhere to the standard `Response` interface, as defined here: https://developer.mozilla.org/en-US/docs/Web/API/Response.

Returning responses might be strange at first. After all, the `loader()` code is still executed inside the browser (not on a server). Therefore, technically, no request was sent, and no response should be required (since the entire code is executed in the same environment, that is, the browser).

For that reason, you can but don't have to return a response; you may return any kind of value. React Router just also supports responses as one possible return value type.

`useLoaderData()` can be called in any component rendered by the currently active route component. That may be the route component itself (`Posts`, in the preceding example), but it may also be any nested component.

For example, `useLoaderData()` can also be used in a `PostsList` component that's included in the `Posts` component (which has a `loader` added to its route definition):

```
import { useLoaderData } from 'react-router-dom';
function PostsList() {
  const loadedPosts = useLoaderData();
  return (
      <main>
     <h1>Your Posts</h1>
     <ul className="posts">
       {loadedPosts.map((post) => (
         <li key={post.id}>{post.title}</li>
       ))}
     </ul>
   </main>
  );
}
export default PostsList;
```

For this example, the `Posts` component file looks like this:

```
import PostsList from '../components/PostsList.jsx';
function Posts() {
  return (
      <main>
     <h1>Your Posts</h1>
     <PostsList />
   </main>
  );
}
```

```
export default Posts;
export async function loader() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```
CopyExplain

This means that `useLoaderData()` can be used in exactly the place where you need the data. The `loader()` function can also be defined wherever you want but it must be added to the route where the data is required.

Note

Depending on the React Router version being used, you might get a warning related to “No HydrateFallback” element being provided. You can ignore this warning as it only matters when using server-side rendering.

Note

You can also explore this code example on GitHub: ReactConcepts/examples/02-data-fetching-react-router at manage-router-data · TalipotTech/ReactConcepts

Loading Data for Dynamic Routes

For most websites, it’s unlikely that static, pre-defined routes alone will be sufficient to meet your needs. For instance, if you created a blogging site with exclusively static routes, you would be limited to a simple list of blog posts on `/posts`. To add more details about a selected blog post on routes such as `/posts/1` or `/posts/2` (for posts with different `id` values) you would need to include dynamic routes.

Of course, React Router also supports data fetching with the help of the `loader()` function for dynamic routes:

```
{
  path: "/posts/:id",
  element: <PostDetails />,
  loader: postDetailsLoader
}
```

The `PostDetails` component and its `loader` function can be implemented like this:

```
import { useLoaderData } from 'react-router-dom';
function PostDetails() {
  const post = useLoaderData();
  return (
    <div id="post-details">
   <h1>{post.title}</h1>
   <p>{post.body}</p>
  </div>
  );
}
export default PostDetails;
export async function loader({ params, request }) {
  console.log(request);
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts/' + params.id
  );
  if (!response.ok) {
    throw new Error('Could not fetch post for id ' + params.id);
  }
  return response;
}
```

If it looks very similar to the `Posts` component in the Loading Data with React Router section, that's no coincidence. Because the `loader()` function works in exactly the same way, there is just one extra feature being used to get hold of the dynamic path segment value: a `params` object that's made available by React Router.

Note

You can also explore this code example on GitHub: [ReactConcepts/examples/03-dynamic-routes at manage-router-data · TalipotTech/ReactConcepts].

When adding a `loader()` function to a route definition, React Router calls that function whenever the route becomes active, right before the component is rendered. When executing that function, React Router passes an object that contains extra information as an argument to `loader()`.

This object passed to `loader()` includes two main properties:

- A `request` property that contains an object with more details about the request that led to the route activation
- A `params` property that yields an object containing a key-value map of all dynamic route parameters for the active route

The `request` object doesn't matter for this example and will be discussed in the next section. But the `params` object contains an `id` property that carries the `id` value of the post for which the route is loaded. The property is named `id` because, in the route definition, `/posts/:id` was chosen as a path. If a different placeholder name had been chosen, a property with that name would have been available on `params` (for example, for `/posts/:postId`, this would be `params.postId`). This behavior is similar to the `params` object yielded by `useParams()`, as explained in Chapter, Multipage Apps with React Router.

With the help of the `params` object and the post `id`, the appropriate post `id` can be included in the outgoing request URL (for the `fetch()` request), and hence the correct post data can be loaded from the backend API. Once the data arrives, React Router will render the `PostDetails` component and expose the loaded post via the `useLoaderData()` Hook.

Loaders, Requests, and Client-Side Code

In the preceding section, you learned about a `request` object being provided to the `loader()` function. Getting such a `request` object might be confusing because React Router is a client-side library—all the code executes in the browser, not on a server. Therefore, no request should reach the React app (as HTTP requests are sent from the client to the server, not between JavaScript functions on the client side).

And, indeed, there is no request being sent via HTTP. Instead, React Router creates a request object via the browser's built-in `Request` interface to use it as a "data vehicle." This request is not sent via HTTP, but it's used as a value for the `request` property on the data object that is passed to your `loader()` function.

Note

For more information on the built-in `Request` interface, visit https://developer.mozilla.org/en-US/docs/Web/API/Request.

This `request` object will be unnecessary in many `loader` functions, but there are occasional scenarios in which you can extract useful information from that object—information that might be needed in the `loader` to fetch the right data.

For example, you can use the `request` object and its `url` property to get access to any search parameters (query parameters) that may be included in the currently active page's URL:

```
export async function loader({ request }) {
  // e.g. for localhost:5173/posts?sort=desc
  const sortDirection = new
URL(request.url).searchParams.get('sort');
  // Fetch sorted posts, based on local 'sort' query param value
  const response = await fetch(
    'https://example.com/posts?sorting=' + sortDirection
  );
  return response;
}
```

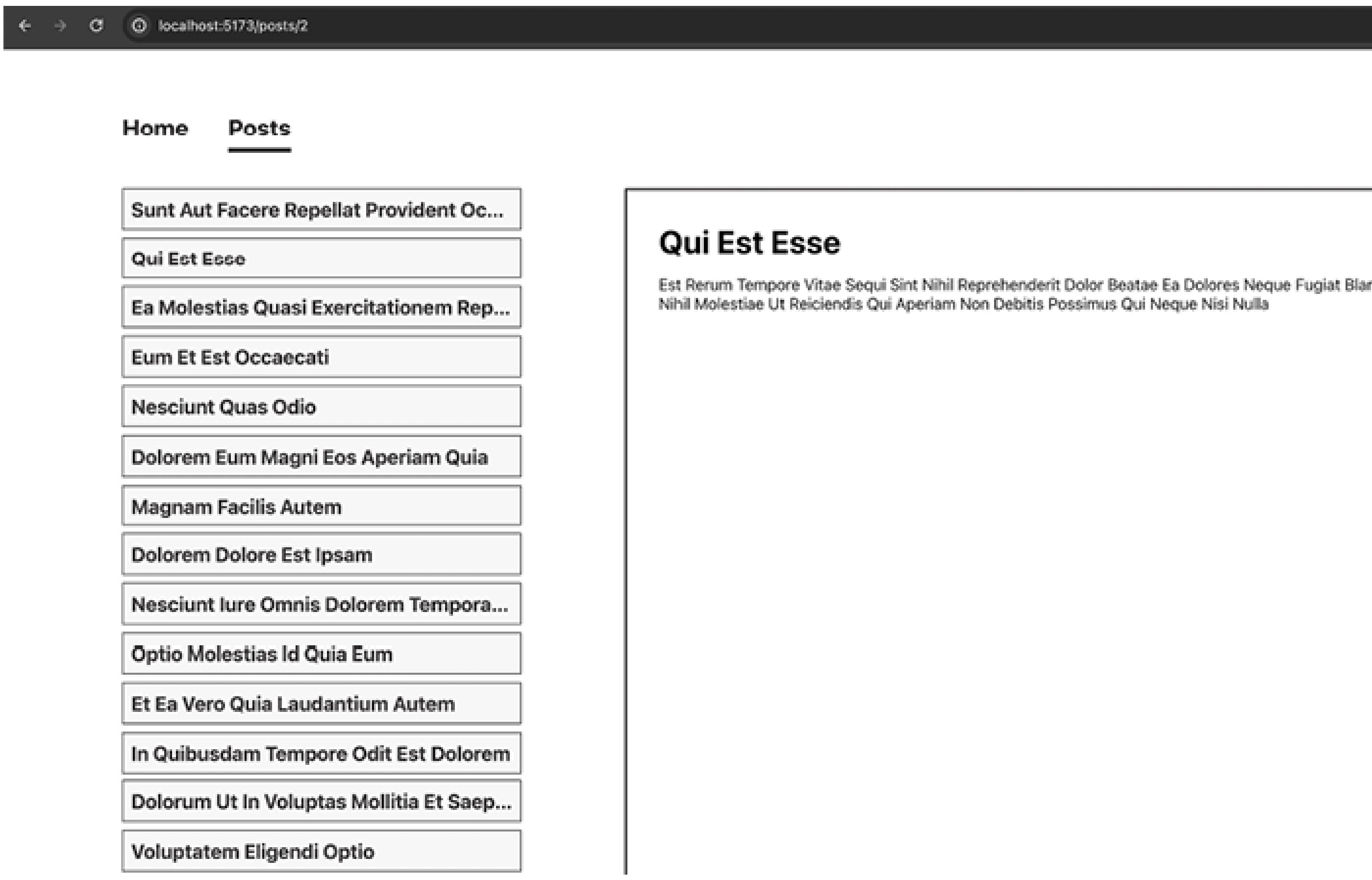
In this code snippet, the `request` value is used to get hold of a query parameter value that's used in the React app URL. That value is then used in an outgoing request.

However, it is vital that you keep in mind that the code inside your `loader()` function, just like all your other React code, always executes on the client side. If, instead, you want to execute code on a server (and, for example, fetch data on the server side), you need to use **server-side rendering** (**SSR**) or some React framework that implements SSR, like Next.js..

Layouts Revisited

React Router supports the concept of layout routes. These are routes that contain other routes and render those other routes as nested children. As you may recall, this concept was introduced in Multipage Apps with React Router.

Conveniently, layout routes can also be used for sharing data across nested routes. Consider this example website:



localhost:5173/posts/2

Home Posts

Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem
Dolorum Ut In Voluptas Mollitia Et Saep...
Voluptatem Eligendi Optio

Qui Est Esse

Est Rerum Tempore Vitae Sequi Sint Nihil Reprehenderit Dolor Beatae Ea Dolores Neque Fugiat Blan
Nihil Molestiae Ut Reiciendis Qui Aperiam Non Debitis Possimus Qui Neque Nisi Nulla

Figure 14.2: A website with a header, a sidebar, and some main content

This website has a header with a navigation bar, a sidebar showing a list of available posts, and a main area that displays the currently selected blog post.

This example includes two layout routes that are nested into each other:

- The root layout route, which includes the top navigation bar that is shared across all pages
- A posts layout route, which includes the sidebar and the main content of its child routes (for example, the details for a selected post)

The route definitions code looks like this:

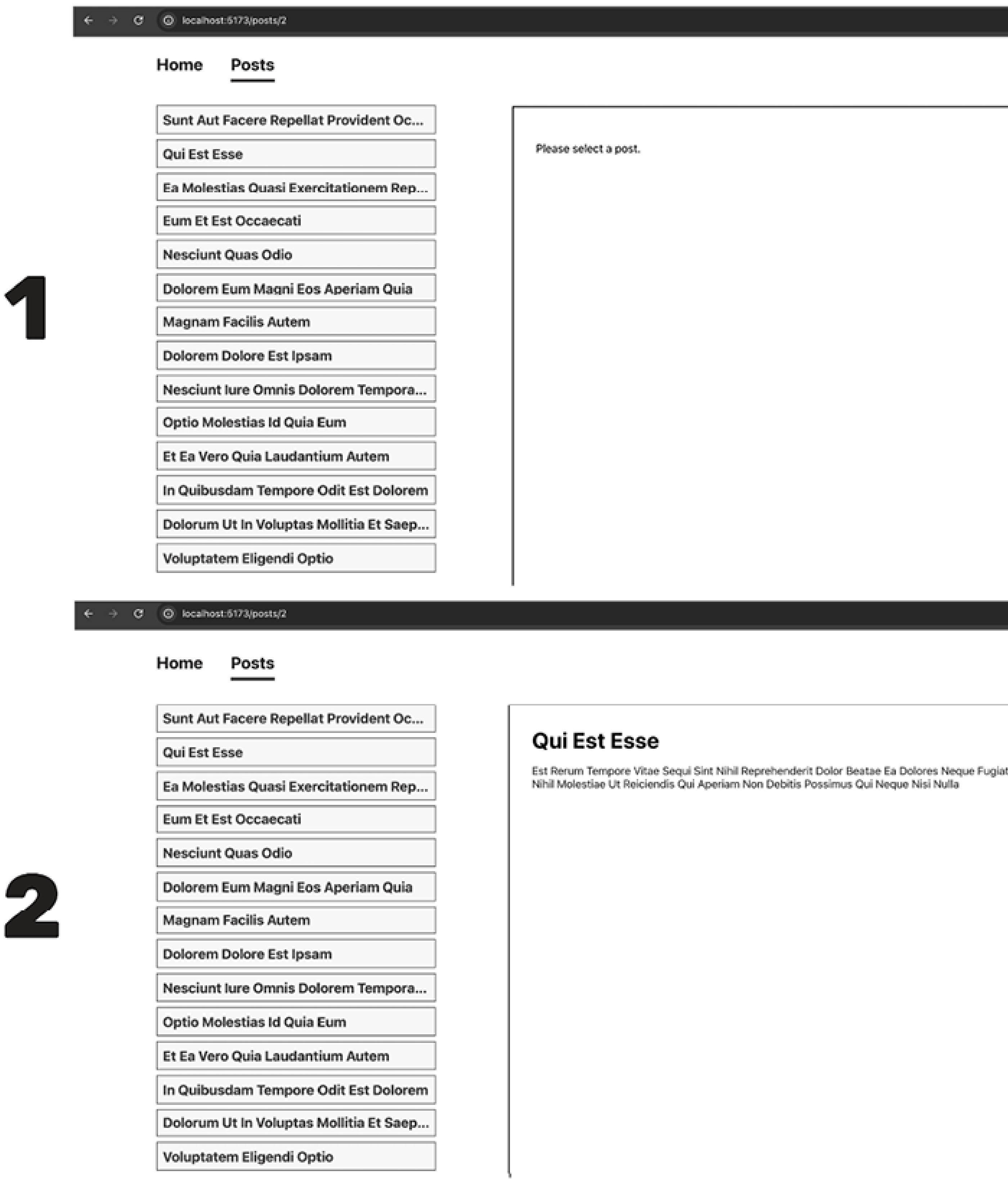
```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,  // main layout, adds navigation bar
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        element: <PostsLayout />,  // posts layout, adds posts sidebar
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          {
            path: ':id',
            element: <PostDetails />,
            loader: postDetailsLoader
          },
        ],
      },
    ],
  },
]);
```

With this setup, both the `<Posts />` and the `<PostDetails />` components are rendered next to the sidebar (since the sidebar is part of the `<PostsLayout />` element).

The interesting part is that the `/posts` route (i.e., the layout route) loads the post data, as it has the `postsLoader` assigned to it, and so the `PostsLayout` component file looks like this:

```
import { Outlet, useLoaderData } from 'react-router-dom';
import PostsList from '../components/PostsList.jsx';
function PostsLayout() {
  const loadedPosts = useLoaderData();
  return (
      <div id="posts-layout">
    <nav>
      <PostsList posts={loadedPosts} />
    </nav>
    <main>
      <Outlet />
    </main>
    </div>
  );
}
export default PostsLayout;
export async function loader() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```

Since layout routes are also regular routes, you can add `loader()` functions and use `useLoaderData()` just as you could in any other route. However, because layout routes are activated for multiple child routes, their data is also displayed for different routes. In the preceding example, the list of blog posts is always displayed on the left side of the screen, no matter if a user visits `/posts` or `/posts/10`:



localhost:5173/posts/2

Home Posts

- Sunt Aut Facere Repellat Provident Oc...
- Qui Est Esse
- Ea Molestias Quasi Exercitationem Rep...
- Eum Et Est Occaecati
- Nesciunt Quas Odio
- Dolorem Eum Magni Eos Aperiam Quia
- Magnam Facilis Autem
- Dolorem Dolore Est Ipsam
- Nesciunt Iure Omnis Dolorem Tempora...
- Optio Molestias Id Quia Eum
- Et Ea Vero Quia Laudantium Autem
- In Quibusdam Tempore Odit Est Dolorem
- Dolorum Ut In Voluptas Mollitia Et Saep...
- Voluptatem Eligendi Optio

Please select a post.

localhost:5173/posts/2

Home Posts

- Sunt Aut Facere Repellat Provident Oc...
- Qui Est Esse
- Ea Molestias Quasi Exercitationem Rep...
- Eum Et Est Occaecati
- Nesciunt Quas Odio
- Dolorem Eum Magni Eos Aperiam Quia
- Magnam Facilis Autem
- Dolorem Dolore Est Ipsam
- Nesciunt Iure Omnis Dolorem Tempora...
- Optio Molestias Id Quia Eum
- Et Ea Vero Quia Laudantium Autem
- In Quibusdam Tempore Odit Est Dolorem
- Dolorum Ut In Voluptas Mollitia Et Saep...
- Voluptatem Eligendi Optio

Qui Est Esse

Est Rerum Tempore Vitae Sequi Sint Nihil Reprehenderit Dolor Beatae Ea Dolores Neque Fugiat
Nihil Molestiae Ut Reiciendis Qui Aperiam Non Debitis Possimus Qui Neque Nisi Nulla

Figure 14.3: The same layout and data are used for different child routes

In this screenshot, the layout and data used do not change as different child routes are activated. React Router also avoids unnecessary data re-fetching (for the blog posts list data) as you switch between child routes. It's smart enough to realize that the surrounding layout hasn't changed.

Reusing Data across Routes

Layout routes do not just help you share components and markup. They also allow you to load and share data across a layout route and its child routes.

For example, the `PostDetails` component (that is, the component that's rendered for the `/posts/:id` route) needs the data for a single post, and that data can be retrieved via a `loader` attached to the `/posts/:id` route:

```
export async function loader({ params }) {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts/' + params.id
  );
  if (!response.ok) {
    throw new Error('Could not fetch post for id ' + params.id);
  }
  return response;
}
```

This example was discussed earlier in this chapter in the Loading Data for Dynamic Routes section. This approach is fine, but in some situations, this extra HTTP request can be avoided. For example, the following route configuration can be simplified, and the extra `postDetailsLoader` on the child route can be avoided:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,  // main layout, adds navigation bar
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        element: <PostsLayout />,  // posts layout, adds posts sidebar
        loader: postsLoader,
        children: [
            { index: true, element: <Posts /> },
            {
              path: ':id',
              element: <PostDetails />,
              loader: postDetailsLoader  // can be removed
            },
          ],
        },
      ],
    },
]);
```

CopyExplain

In this example, the `PostsLayout` route already fetches a list of all posts. That layout component is also active for the `PostDetails` route. In such a scenario, fetching a single post is unnecessary, since all the data has already been fetched for the list of posts. Of course, a specific `postDetailsLoader` loader for the `PostDetails` child route would be required if the request for the list of posts (by `postsLoader` on the `PostsLayout` route) didn't yield all the data required by `PostDetails`.

But if all the data is available, React Router allows you to tap into the loader data of a parent route component via the `useRouteLoaderData()` Hook.

This Hook can be used like this:

```
const posts = useRouteLoaderData('posts');
```

`useRouteLoaderData()` requires a route identifier as an argument. It requires an identifier assigned to the ancestor route that contains the data that should be reused. You can assign such an identifier via the `id` property to your routes as part of the route definitions code:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,  // main layout, adds navigation bar
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts', // the id value is up to you
        element: <PostsLayout />,  // posts layout, adds posts sidebar
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          {
            path: ':id',
            element: <PostDetails />,  // details loader was removed
          },
        ],
      },
    ],
  },
]);
```

The `useRouteLoaderData()` Hook then returns the same data `useLoaderData()` yields in that route to which you added the `id`. In this example, it would provide a list of blog posts.

In `PostDetails`, this Hook can therefore be used like this:

```
import { useParams,useRouteLoaderData,  } from 'react-router-dom';
function PostDetails() {
  const params = useParams();
  const posts = useRouteLoaderData('posts');
  const post = posts.find((post) => post.id.toString() === params.id);
  return (
    <div id="post-details">
   <h1>{post.title}</h1>
   <p>{post.body}</p>
  </div>
  );
}
export default PostDetails;
```

The `useParams()` Hook is used to get access to the dynamic route parameter value, and the `find()` method is used on the list of posts to identify a single post with a fitting `id` property. In this example, you would thus avoid sending an unnecessary HTTP request by reusing data that's already available.

Therefore, the `postDetailsLoader` that was part of the `/posts/:id` route definition can be removed.

Handling Errors

In the first example at the very beginning of this chapter (where the HTTP request was sent with the help of `useEffect()`), the code did not just handle the success case but also possible errors. In all the React Router-based examples since then, error handling has been omitted. Error handling was not discussed up to this point because, while React Router plays an important role in error handling, it's vital to first gain a solid understanding of how React Router works in general and how it helps with data fetching. But, of course, errors can't always be avoided and definitely should not be ignored.

Thankfully, handling errors is also very straightforward and easy when using React Router's data capabilities. You can set an `errorElement` property on your route definitions and define the element that should be rendered when an error occurs:

```
// ... other imports
import Error from './components/Error.jsx';
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />,
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
        ],
      },
    ],
  },
]);
```

This `errorElement` property can be set on any route definition of your choice, or even multiple route definitions simultaneously. React Router will render the `errorElement` of the route closest to the place where the error was thrown.

In the preceding snippet, no matter which route produced an error, it would always be the root route's `errorElement` that was displayed (since that's the only route definition with an `errorElement`). But if you also added an `errorElement` to the `/posts` route, and the `:id` route produced an error, it would be the `errorElement` of the `/posts` route that was shown on the screen, as follows:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />, // for all errors not handled elsewhere
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        // used if /posts or /posts/:id throws an error
        errorElement: <PostsError />, // handles /posts related errors
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
        ],
      },
    ],
  },
]);
```

This allows you, the developer, to set up fine-grained error handling.

Try http://localhost:5173/post/1

this in browser . As there is no post buts posts it should show the error page

Inside the component used as a value for the `errorElement`, you can get access to the error that was thrown via the `useRouteError()` Hook:

```
import { useRouteError } from 'react-router-dom';
function Error() {
  const error = useRouteError();
  return (
    <>
   <h1>Oh no!</h1>
   <p>An error occurred</p>
   <p>{error.message}</p>
  </>
  );
}
export default Error;
```

With this simple yet effective error-handling solution, React Router allows you to avoid managing error states yourself. Instead, you simply define a standard React element (via the `element` prop) that should be displayed when things go right and an `errorElement` to be displayed if things go wrong.

Code at https://github.com/TalipotTech/ReactConcepts/tree/manage-router-data/examples/05-errors

Onward to Data Submission

Thus far, you've learned a lot about data fetching. But as mentioned earlier in this chapter, React Router also helps with data submission.

Consider the following example component:

```
function NewPost() {
  return (
    <form id="post-form">
   <p>
    <label htmlFor="title">Title</label>
    <input type="text" id="title" name="title" />
   </p>
```

```
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" name="text" rows={3} />
      </p>
      <button>Save Post</button>
    </form>
  );
}
export default NewPost;
```

This component renders a `<form>` element that allows users to enter the details for a new post. Due to the following route configuration, the component is displayed whenever the `/posts/new` route becomes active:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />,
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
          { path: 'new', element: <NewPost /> },
        ],
      },
    ],
  },
]);
```

Without React Router’s data-related features, you might handle form submission like this:

```
function NewPost() {
  const navigate = useNavigate();
  async function submitAction(formData) {
    const enteredTitle = formData.get('title');
    const enteredText = formData.get('text');
    const postData = {
      title: enteredTitle,
      text: enteredText
    };
    await fetch('https://jsonplaceholder.typicode.com/posts', {
      method: 'POST',
      body: JSON.stringify(postData),
      headers: {'Content-Type': 'application/json'}
    });
    navigate('/posts');
  }
  return (
    <form action={submitAction}>
   <p>
    <label htmlFor="title">Title</label>
    <input type="text" id="title" name="title" />
   </p>
   <p>
    <label htmlFor="text">Text</label>
    <textarea id="text" rows={3} name="text" />
   </p>
   <button>Save Post</button>
  </form>
  );
}
```

Just as before when fetching data, this requires quite a bit of code and logic to be added to the component function. You must manually extract the submitted data, send the HTTP request, and navigate to a different page after receiving an HTTP response.

In addition, you might also need to manage loading state and potential errors (excluded in the preceding example).

Again, React Router offers some help. Where a `loader()` function can be added to handle data loading, an `action()` function can be defined to handle data submission.

When using the new `action()` function, the preceding example component looks like this:

```
import { Form, redirect } from 'react-router-dom';
function  NewPost()  {
    return  (
        <Form method="post" id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title"/>
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" rows={3} name="text" />
      </p>
      <button>Save Post</button>
    </Form>
    ) ;
}
export  default  NewPost;
export async function action({ request }) {
    const formData = await request.formData();
    const enteredTitle = formData.get('title');
    const enteredText = formData.get('text');
    const postData = {
        title: enteredTitle,
        text: enteredText
    };
    await fetch('https://jsonplaceholder.typicode.com/posts', {
        method: 'POST',
        body: JSON.stringify(postData),
        headers: { 'Content-Type': 'application/json' },
    });
    return redirect('/posts');
}
```

This code might be similar in length but it has the advantage of moving all the data submission logic out of the component function into a special `action()` function.

Besides the addition of the `action()` function, the example code snippet includes the following important changes and features:

- A `<Form>` component that's used instead of `<form>`.
- The `method` prop is set on the `<Form>` (to `"post"`).
- The submitted data is extracted as `FormData` by calling `request.formData()`.
- The user is redirected via a newly added `redirect()` function (instead of `useNavigate()` and `navigate()`).

But what are these elements about?

Working with action() and Form Data

Just like `loader()`, `action()` is a special function that can be added to route definitions, as follows:

```
import NewPost, { action as newPostAction } from './components/NewPost.jsx';
// ...
{ path: 'new', element: <NewPost />, action: newPostAction },
```

With the `action` prop set on a route definition, the assigned function is automatically called whenever a `<Form>` (not `<form>`!) targeting this route is submitted. `Form` is a component provided by React Router that should be used instead of the default `<form>` element.

Internally, `Form` uses the default `<form>` element but prevents the browser default of creating and sending an HTTP request upon form submission. Instead, React Router creates a `FormData` object and calls the `action()` function defined for the route that's targeted by the `<Form>`, passing a request object, based on the built-in `Request` interface, to it. The passed request object contains the form data generated by React Router. Later in this chapter, in the Controlling Which <Form> Triggers Which Action section, you'll learn how to control which `action()` function of which route will be executed by React Router.

Note

Handling form submissions with the help of "actions" might sound familiar—Handling User Input & Forms with Form Actions

But whereas we discussed a feature built into React (which was not related or dependent on routing), this chapter explores a core concept of React Router.

Ultimately, you can use either approach for handling form submissions. Or you could use none of the two and instead handle the `submit` event manually via `onSubmit`.

But when using routing with React Router, you'll often end up with cleaner, more concise code that integrates smoothly with other routing features like redirects when using React Router's `<Form>` component and `action()` function.

The form data object that is created by calling `request.formData()` includes all form input values entered into the submitted form. To be registered, an input element such as `<input>`, `<select>`, or `<textarea>` must have the `name` attribute assigned to it. The values set for those `name` attributes can later be used to extract the entered data.

The `request` object (that contains the form data) received by the `action()` function is created by React Router when the form is submitted.

The `Form` component defines the HTTP method of the request object. By setting the `Form`'s `method` prop to either `"get"` (the default) or `"post"`, you control what happens when the form is submitted. When setting `method="get"` (or when not setting `method` at all), a regular URL navigation will occur—just as if a link to a certain path were clicked. Any entered form values will be encoded as URL search parameters in that case. To trigger an `action()` function, `<Form>`'s `method` must be set to `"post"` instead.

However, it's important to understand that the request is not sent via HTTP since `action()`, just like `loader()` or the component function, still executes in the browser rather than on a server.

The `action()` function then receives an object with a `request` property that contains the created request object with the included form data. This `request` object can be used to extract the values entered into the form input fields like this:

```
export async function action({ request }) {
    const formData = await request.formData();
    const postData = Object.fromEntries(formData);
    // ...
}
```

The built-in `formData()` method yields a `Promise` that resolves to a `FormData` object that offers a `get()` method that can be used to get an entered value by its identifier (that is, by the `name` attribute value set on the input element). For example, the value entered into `<input name="title">` could be retrieved via `formData.get('title')`.

Alternatively, you can follow the approach chosen in the preceding code snippet and convert the `formData` object to a simple key-value object via `Object.fromEntries(formData)`. This object (`postData`, in the preceding example) contains the names set on the form input elements as properties and the entered values as values for those properties (meaning that `postData.title` would yield the value entered in `<input name="title">`).

Note

React Router also supports the other main HTTP verbs (`"patch"`, `"put"`, and `"delete"`), and setting `method` to one of these verbs will indeed also trigger the `action()` function.

This can be useful when working with multiple forms that should trigger the same `action()`. By using different methods, you can use one single action to run different code based on the value extracted from `request.method` inside the `action()` function.

But it's worth noting that using methods other than `'get'` and `'post'` is not in line with the HTML standard. Therefore, React Router could remove support for these methods in the future.

Hence, when working with multiple forms that trigger the same `action()`, a more stable solution can be to include a hidden input field with a unique identifier (e.g., `<input type="hidden" name="_method" value="DELETE">`). This value can then be extracted and used (e.g., in an `if` statement) in the `action()` function.

The extracted data can then be used for any operations of your choice. That could be an extra validation step or an HTTP request sent to some backend API, where the data may get stored in a database or file:

```
export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);
  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(postData),
    headers: { 'Content-Type': 'application/json' },
  });
  return redirect('/posts');
}
```

Finally, once all intended steps are performed, the `action()` function must return a value—any value of any type, but at least `null`. Not returning anything (i.e., omitting the `return` statement) is not allowed. Though, as with the `loader()` function, you may also return a response, for example, a redirect response like this:

```
export async function action({ request }) {
  // action logic …
  return new Response("", {
    status: 302,
    headers: {
      Location: '/posts'
    }
  });
}
```
CopyExplain

Indeed, for actions, it's highly likely that you will want to navigate to a different page once the action has been performed (e.g., once an HTTP request to an API has been sent). This may be required to navigate the user away from the data input page to a page that displays all available data entries (for example, from `/posts/new` to `/posts`).

To simplify this common pattern, React Router provides a `redirect()` function that yields a response object that causes React Router to switch to a different route. You can therefore return the result of calling `redirect()` in your `action()` function to ensure that the user is navigated to a different page. It's the equivalent of calling `navigate()` (via `useNavigate()`) when manually handling form submissions.

```
export async function action({ request }) {
  // action logic …
  return redirect('/posts')
}
```

In this snippet, React Router's `redirect()` function is used instead of manually constructing a `Response` object.

Returning Data Instead of Redirecting

As mentioned, your `action()` functions may return anything. You don't have to return a response object. While it is quite common to return a redirect response, you may occasionally want to return some raw data instead.

One scenario in which you might not want to redirect the user is after validating the user's input. Inside the `action()` function, before sending the entered data to some API, you may wish to validate the provided values first. If an invalid value (such as an empty title) is detected, a great user experience is typically achieved by keeping the user on the route with the `<Form>`. The values entered by the user shouldn't be cleared and lost; instead, the form should be updated to present useful validation error information to the user. This information can be passed from the `action()` to the component function so that it can be displayed there (for example, next to the form input fields).

In situations like this, you can return a "normal" value (that is, not a redirect response) from your `action()` function:

```
export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);
  let validationErrors = [];
  if (postData.title.trim().length === 0) {
    validationErrors.push('Invalid post title provided.')
  }
  if (postData.text.trim().length === 0) {
    validationErrors.push('Invalid post text provided.')
  }
  if (validationErrors.length > 0) {
    return validationErrors;
  }
  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(postData),
    headers: { 'Content-Type': 'application/json' },
  });
  return redirect('/posts');
}
```

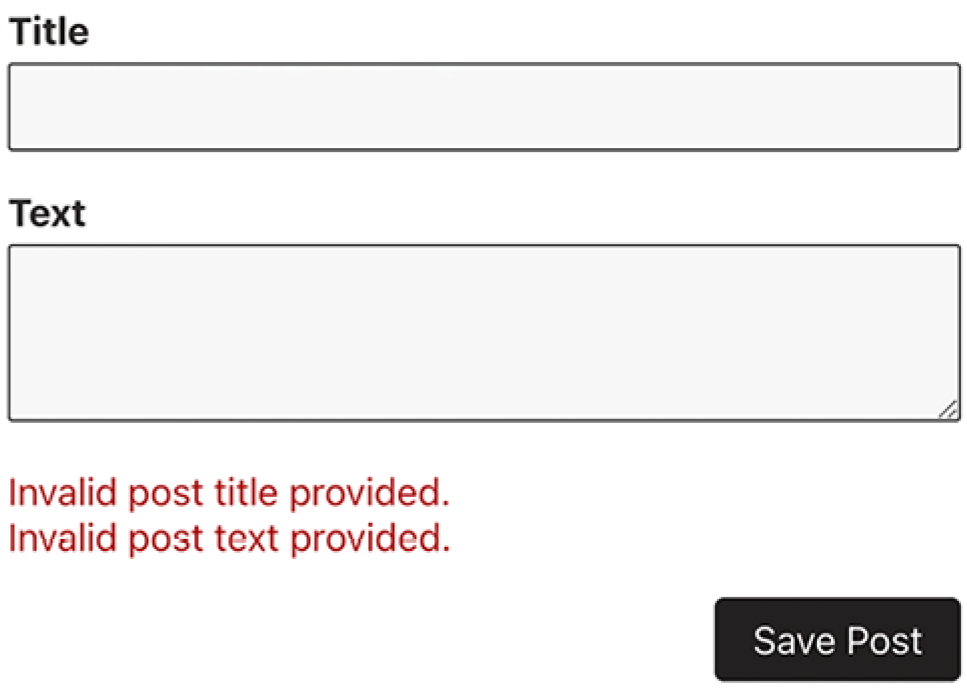
In this example, a `validationErrors` array is returned if the entered `title` or `text` values are empty.

Data returned by an `action()` function can be used in the route component (or any other nested component) via the `useActionData()` Hook:

```
import { Form, redirect, useActionData } from 'react-router-dom';
function NewPost() {
  const validationErrors = useActionData();
  return (
     <Form method="post" id="post-form">
    <p>
      <label htmlFor="title">Title</label>
      <input type="text" id="title" name="title" />
    </p>
    <p>
      <label htmlFor="text">Text</label>
      <textarea id="text" name="text" rows={3} />
    </p>
        <ul>
          {validationErrors &&
            validationErrors.map((err) => <li key={err}>{err}</li>)}
        </ul>
    <button>Save Post</button>
  </Form>
  );
}
CopyExplain
```

`useActionData()` works a lot like `useLoaderData()`, but unlike `useLoaderData()`, it's not guaranteed to yield any data. This is because while `loader()` functions always get called before the route component is rendered, the `action()` function only gets called once the `<Form>` is submitted.

In this example, `useActionData()` is used to get access to the `validationErrors` returned by `action()`. If `validationErrors` is truthy (that is, is not `undefined`), the array will be mapped to a list of error items that are displayed to the user:



Invalid post title provided.
Invalid post text provided.

Figure 14.4: Validation errors are output below the input fields

The `action()` function is therefore quite versatile in that you can use it to perform an action and redirect away as well as to conduct more than one operation and return different values for different use cases.

Controlling Which <Form> Triggers Which Action

Earlier in this chapter, in the section Working with action() and Form Data, you learned that when `<Form>` is used instead of `<form>`, React Router will execute the targeted `action()` function. But which `action()` function is targeted by `<Form>`?

By default, it's the `action()` function assigned to the route that also renders the form (either directly or via some descendent component). Consider this route definition:

```
{ path: '/posts/new', element: <NewPost />, action: newPostAction }
```
CopyExplain

With this definition, the `newPostAction()` function would be triggered whenever any `<Form>` inside of the `NewPost` component (or any nested component) is submitted.

In many cases, this default behavior is exactly what you want. But you can also target `action()` functions defined on other routes by setting the `action` prop on `<Form>` to the path of the route that contains the `action()` that should be executed:

```
// form rendered in a component that belongs to /posts
<Form method="post" action="/save-data">
  ...
</Form>
```

This form would cause React Router to execute the `action` belonging to the `/save-data` route—even though the `<Form>` component may be rendered as part of a component that belongs to a different route (e.g., `/posts`).

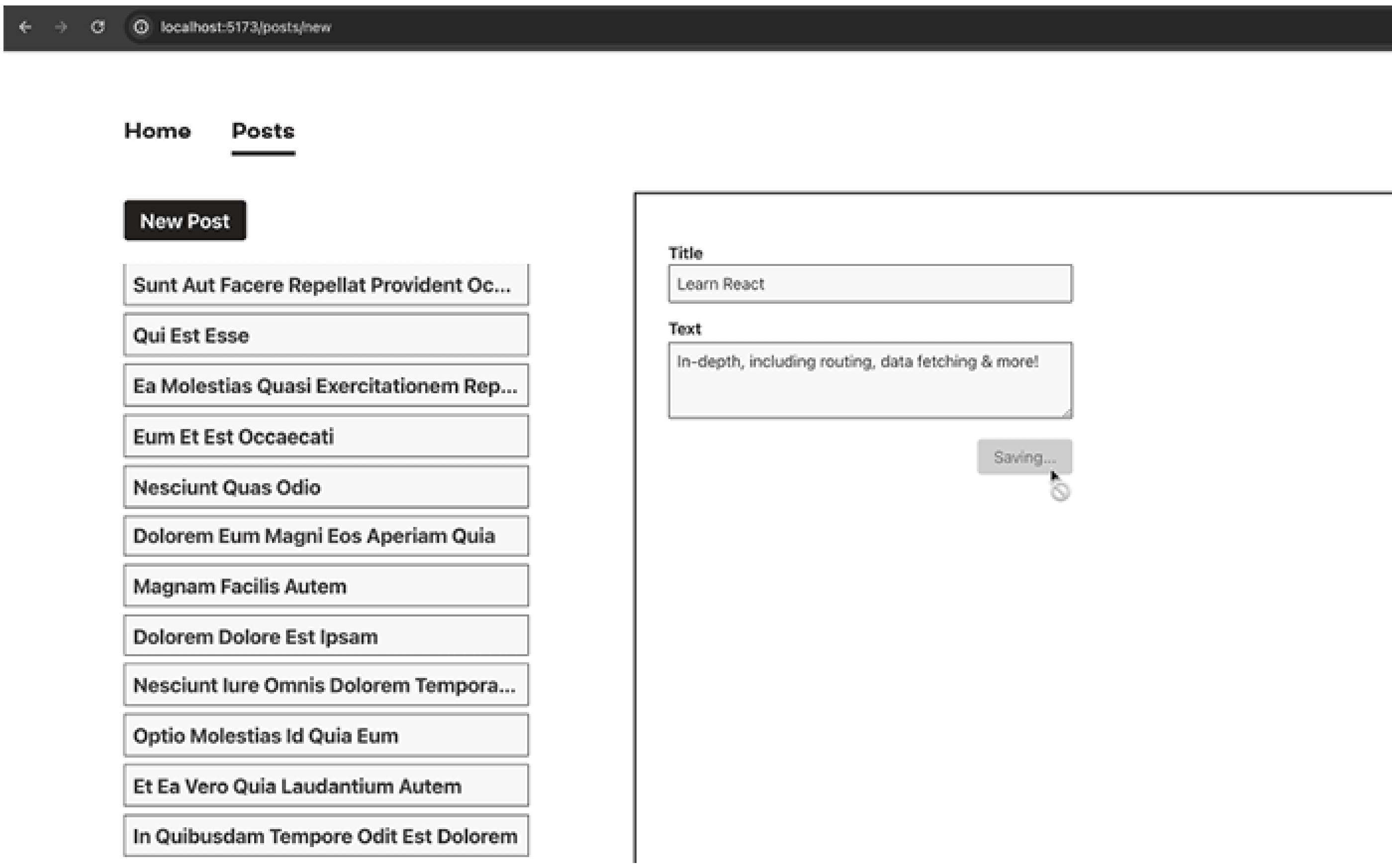
It is worth noting, though, that targeting a different route will lead to a page transition to that route's path, even if your action does not return a redirect response. In a later section of this chapter, entitled Behind-the-Scenes Data Fetching and Submission, you will learn how that behavior can be avoided.

Reflecting the Current Navigation Status

After submitting a form, the `action()` function that's triggered may need some time to perform all intended operations. Sending HTTP requests to APIs in particular can take up to a few seconds.

Of course, it's not a great user experience if the user doesn't get any feedback about the current data submission status. It's not immediately clear if anything happened at all after the submit button was clicked.

For that reason, you might want to show a loading spinner or update the button caption while the `action()` function is running. Indeed, one common way of providing user feedback is to disable the submit button and change its caption like this:


localhost:5173/posts/new

Home Posts

New Post

Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem

Title: Learn React

Text: In-depth, including routing, data fetching & more!

Saving...

Figure 14.5: The submit button is grayed out

You can get the current React Router status (that is, whether it's currently transitioning to another route or executing an `action()` function) via the `useNavigation()` Hook. This Hook provides a navigation object that contains various pieces of routing-related information.

Most importantly, this object has a `state` property that yields a string describing the current navigation status. This property is set to one of the following three possible values:

- `submitting`: If an `action()` function is currently executing
- `loading`: If a `loader()` function is currently executing (for example, because of a `redirect()` response)
- `idle`: If no `action()` or `loader()` functions are currently being executed

You can therefore use this `state` property to find out whether React Router is currently navigating to a different page or executing an `action()`. Hence, the submit button can be updated as shown in the preceding screenshot via this code:

```
import {
  Form,
  redirect,
  useActionData,
  useNavigation
} from 'react-router-dom';
function NewPost() {
  const validationErrors = useActionData();
  const navigation = useNavigation();
  const isSubmitting = navigation.state !== 'idle';
  return (
      <Form method="post" id="post-form">
     <p>
       <label htmlFor="title">Title</label>
       <input type="text" id="title" name="title" />
     </p>
     <p>
       <label htmlFor="text">Text</label>
       <textarea id="text" name="text" rows={3} />
     </p>
     <ul>
       {validationErrors &&
         validationErrors.map((err) => <li key={err}>{err}</li>)}
     </ul>
     <button disabled={isSubmitting}>
       {isSubmitting ? 'Saving...' : 'Save Post'}
     </button>
    </Form>
  );
}
```

In this example, the `isSubmitting` constant is `true` if the current navigation state is anything but `'idle'`. This constant is then used to disable the submit button (via the `disabled` attribute) and adjust the button's caption.

Submitting Forms Programmatically

In some cases, you won't want to instantly trigger an `action()` when a form is submitted—for example, if you need to ask the user for confirmation first such as when triggering actions that delete or update data.

For such scenarios, React Router allows you to submit a form (and therefore trigger an `action()` function) programmatically. Instead of using the `Form` component provided by React Router, you handle the form submission manually using the default `<form>` element. As part of your code, you can then use a `submit()` function provided by React Router's `useSubmit()` Hook to trigger the `action()` manually once you're ready for it.

Consider this example:

```
import {
  redirect,
  useParams,
  useRouteLoaderData,
  useSubmit,
} from 'react-router-dom';
function PostDetails() {
  const params = useParams();
  const posts = useRouteLoaderData('posts');
  const post = posts.find((post) => post.id.toString() === params.id);
  const submit = useSubmit();
  function handleSubmit(event) {
    event.preventDefault();
    const proceed = window.confirm('Are you sure?');
    if (proceed) {
      submit(
        { message: 'Your submitted data, if needed' },
        {
          method: 'post',
        }
      );
    }
  }
  return (
    <div id="post-details">
   <h1>{post.title}</h1>
   <p>{post.body}</p>
```

```
     <form onSubmit={handleSubmit}>
              <button>Delete</button>
           </form>
   </div>
   );
}
export default PostDetails;
// action must be added to route definition!
export async function action({ request }) {
   const formData = await request.formData();
   console.log(formData.get('message'));
   console.log(request.method);
   return redirect('/posts');
}
```

In this example, the `action()` is manually triggered by programmatically submitting data via the `submit()` function provided by `useSubmit()`. This approach is required as it would otherwise be impossible to ask the user for confirmation (via the browser's `window.confirm()` method).

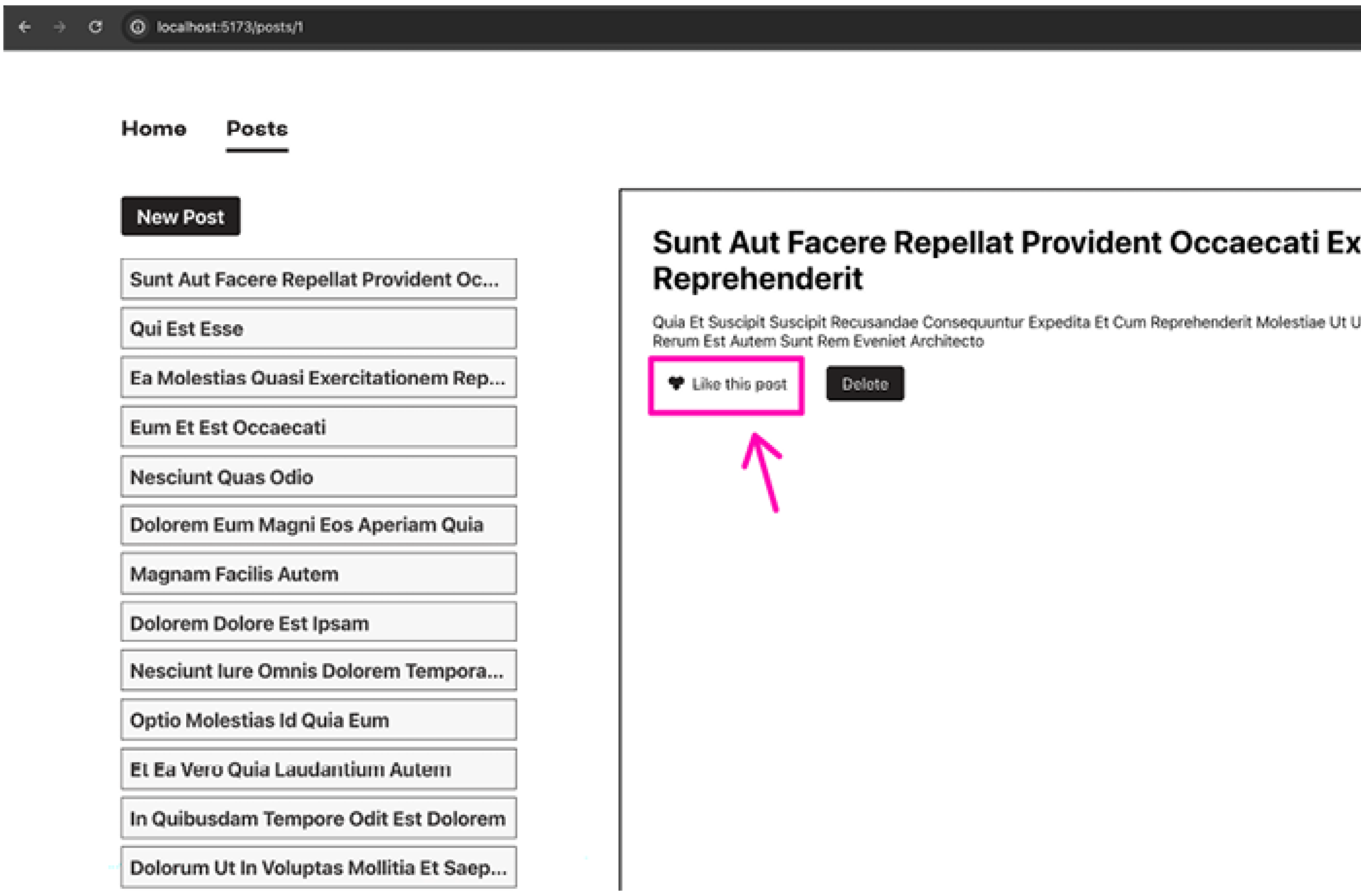
Because data is submitted programmatically, the default `<form>` element should be used and the `submit` event handled manually. As part of this process, the browser's default behavior of sending an HTTP request must also be prevented manually.

Typically, using `<Form>` instead of programmatic submission is preferable. But in certain situations, such as the preceding example, being able to control form submission manually can be useful.

Behind-the-Scenes Data Fetching and Submission

There are also situations in which you may need to trigger an action or load data without causing a page transition.

A Like button would be an example. When it's clicked, a process should be triggered in the background (such as storing information about the user and the liked post), but the user should not be directed to a different page:



Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem
Dolorum Ut In Voluptas Mollitia Et Saep...

Sunt Aut Facere Repellat Provident Occaecati Ex
Reprehenderit

Quia Et Suscipit Suscipit Recusandae Consequuntur Expedita Et Cum Reprehenderit Molestiae Ut U
Rerum Est Autem Sunt Rem Eveniet Architecto

Figure 14.6: A Like button below a post

To achieve this behavior, you could wrap the button into a `<Form>` and, at the end of the `action()` function, simply redirect back to the page that is already active.

But technically, this would still lead to an extra navigation action. Therefore, `loader()` functions would be executed and other possible side-effects might occur (the current scroll position could be lost, for example). For that reason, you might want to avoid this kind of behavior.

Thankfully, React Router offers a solution: the `useFetcher()` Hook, which yields an object that contains a `submit()` method. Unlike the `submit()` function provided by `useSubmit()`, the `submit()` method yielded by `useFetcher()` is meant for triggering actions (or `loader()` functions) without starting a page transition.

A Like button, as described previously, can be implemented like this (with the help of `useFetcher()`):

```
import {
  // ... other imports
  useFetcher,
} from 'react-router-dom';
import { FaHeart } from 'react-icons/fa';
function PostDetails() {
  // ... other code & logic
  const fetcher = useFetcher();
  function handleLikePost() {
    fetcher.submit(null, {
      method: 'post',
      action: `/posts/${post.id}/like`,
      // targeting an action on another route
    });
  }
  return (
    <div id="post-details">
   <h1>{post.title}</h1>
   <p>{post.body}</p>
   <div className="actions">
          <button className="icon-btn" onClick={handleLikePost}>
     <FaHeart />
     <span>Like this post</span>
    </button>
    <form onSubmit={handleSubmit}>
     <button>Delete</button>
    </form>
   </div>
  </div>
  );
}
```

The `fetcher` object returned by `useFetcher()` has various properties. For example, it also contains properties that provide information about the current status of the triggered action or loader (including any data that may have been returned).

But this object also includes two important methods:

- `load()`: To trigger the `loader()` function of a route (e.g., `fetcher.load('/route-path')`)

- `submit()`: To trigger an `action()` function with the provided data and configuration

In the code snippet above, the `submit()` method is called to trigger the action defined on the `/posts/<post-id>/like` route. Without `useFetcher()` (i.e., when using `useSubmit()` or `<Form>`), React Router would switch to the selected route path when triggering its action. With `useFetcher()`, this is avoided, and the action of that route can be called from inside another route (meaning the action defined for `/posts/<post-id>/like` is called while the `/posts/<post-id>` route is active).

This also allows you to define routes that don't render any element (that is, in which there is no page component) and, instead, only contain a `loader()` or `action()` function. For example, the `/posts/<post-id>/like` route file (`pages/like.js`) looks like this:

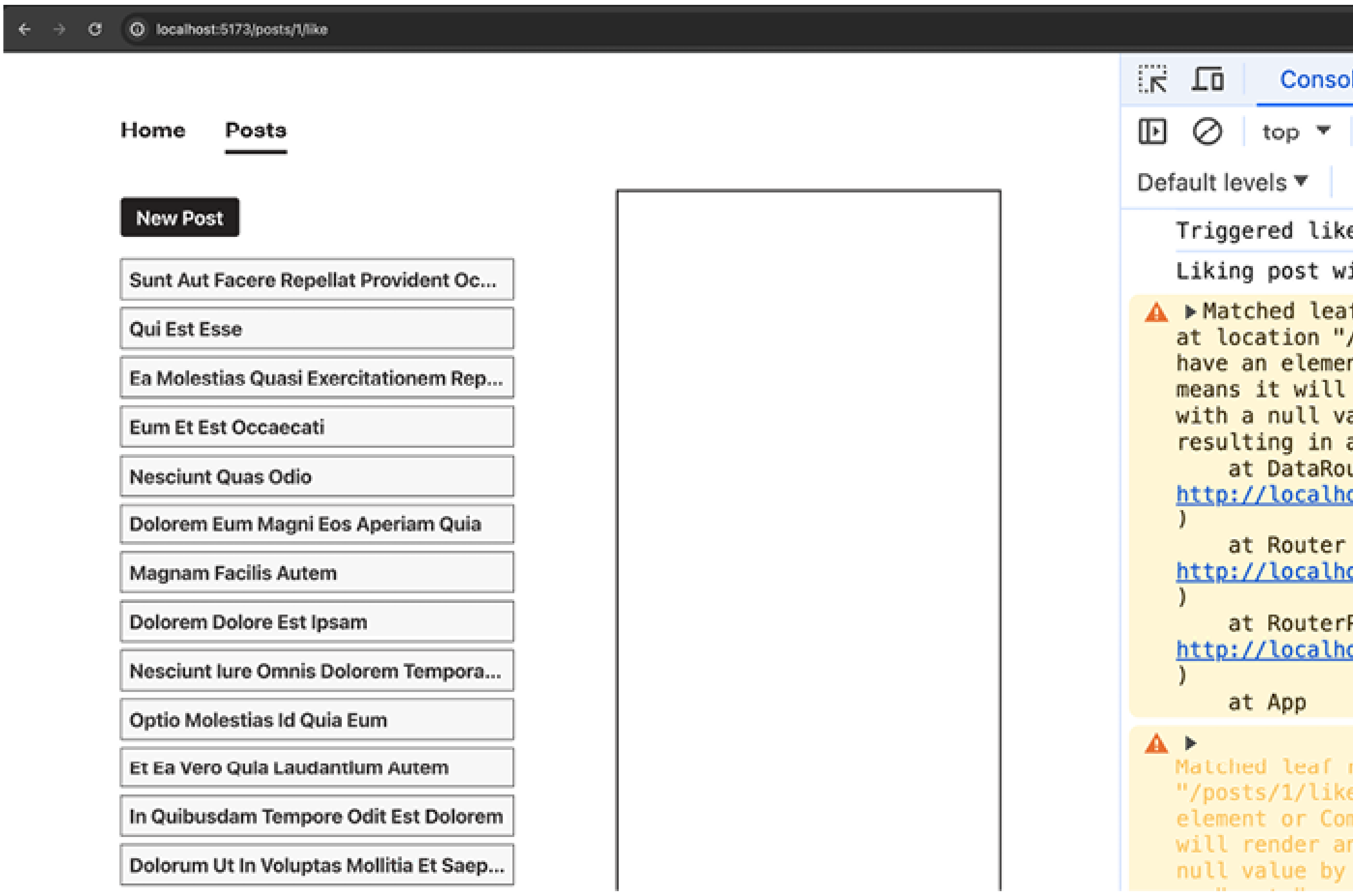
```
// there is no component function in this file!
export function action({ params }) {
  console.log('Triggered like action.');
  console.log(`Liking post with id ${params.id}.`);
  // Do anything else
  // May return data or response, including redirect() if needed
  return null; // something must be returned, even if it's just null
}
```
CopyExplain

As mentioned in the code snippet, any data may be returned in this action. But you must at least return `null`—avoiding the `return` statement and not returning anything is not allowed and will cause an error.

It's registered as a route as follows:

```
import { action as likeAction } from './pages/like.js';
// ...
{ path: ':id/like', action: likeAction },
```

This works because this `action()` is only triggered via the `submit()` method provided by `useFetcher()`. `<Form>` and the `submit()` function yielded by `useSubmit()` would instead initiate a route transition to `/posts/<post-id>/like`. Without the `element` property being set on the route definition, this transition would lead to an empty page, as shown here:



```
localhost:5173/posts/1/like
```

Home
Posts
New Post
Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem
Dolorum Ut In Voluptas Mollitia Et Saep...
Conso
top
Default levels

```
Triggered like
Liking post wi
▶Matched leaf
at location "/
have an elemen
means it will
with a null va
resulting in a
    at DataRou
http://localho
)
    at Router
http://localho
)
    at RouterP
http://localho
)
    at App
▶
Matched leaf r
"/posts/1/like
element or Com
will render an
null value by
```

Figure 14.7: An empty (nested) page is displayed, along with a warning message

Because of the extra flexibility it offers, `useFetcher()` can be very useful when building highly interactive user interfaces. It's not meant as a replacement for `useSubmit()` or `<Form>`, but rather, as an additional tool for situations where no route transition is required or wanted.

Deferring Data Loading

Up to this point in the chapter, all data fetching examples have assumed that a page should only be displayed once all its data has been fetched. That's why there was never any loading state that would have been managed (and hence no loading fallback content that would have been displayed).

In many situations, this is exactly the behavior you want as it does not often make sense to show a loading spinner or similar fallback content for a fraction of a second just to then replace it with the actual page data.

But there are also situations in which the opposite behavior might be desirable—for example, if you know that a certain page will take quite a while to load its data (possibly due to a complex database query that must be executed on the backend) or if you have a page that loads different pieces of data and some pieces are much slower than others.

In such scenarios, it may make sense to render the page component even though some data is still missing. React Router also supports this use case by allowing you to defer data loading, which, in turn, enables the page component to be rendered before the data is available.

Deferring data loading is as simple as returning a promise from the loader (instead of awaiting it there):

```
// ... other imports
export async function loader() {
  return {
    posts: getPosts()
  };
}
```
CopyExplain

In this example, `getPosts()` is a function that returns a (slow) `Promise`:

```
async function getPosts() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  await wait(3); // utility function, simulating a slow response
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  const data = await response.json();
  return data;
}
```
CopyExplain

React Router allows you to return raw promises. When doing so, you can wait for the actual values yielded by those promises in the client-side code.

Inside the component function where `useLoaderData()` is used, you must also use a new component provided by React Router: the `Await` component. It's used like this:

```
import { Suspense } from 'react';
import { Await } from 'react-router-dom';
// ... other imports
function PostsLayout() {
  const data = useLoaderData();
  return (
    <div id="posts-layout">
   <nav>
    <Suspense fallback={<p>Loading posts...</p>}>
           <Await resolve={data.posts}>
             {(loadedPosts) => <PostsList posts={loadedPosts} />}
           </Await>
         </Suspense>
   </nav>
   <main>
    <Outlet />
   </main>
  </div>
  );
}
```
CopyExplain

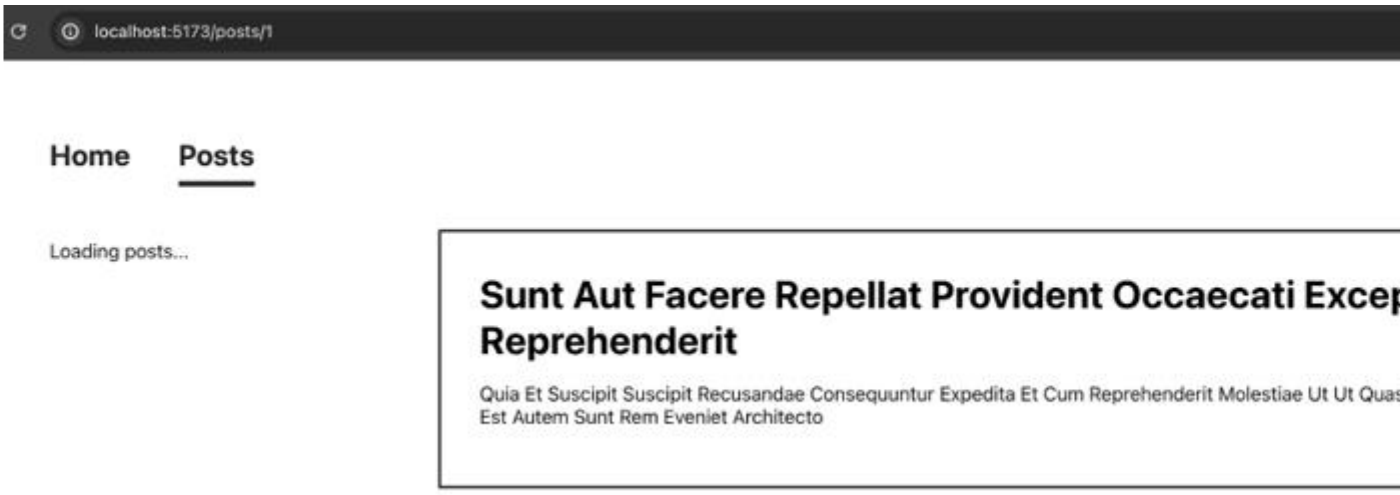
The `<Await>` element takes a `resolve` prop that receives a value of type `Promise` from the loader data. It's wrapped by the `<Suspense>` component provided by React.

The value passed to resolve is a `Promise` that was stored in the object returned by the `loader()` function. There, a key named posts was used to hold that `Promise`. The value for that key was the `Promise` returned by `getPosts()`. It's this `Promise` that's passed as a value to `resolve` via `<Await resolve={data.posts}>`. If a different key name were used (e.g., `blogPosts`), that key name had to be referenced when setting `resolve` (e.g., `<Await resolve={data.blogPosts}>`).

`Await` automatically waits for the `Promise` to resolve before then calling the function that's passed to `<Await>` as a child (that is, the function passed between the `<Await>` opening and closing tags). This function is executed by React Router once the data of the deferred operation is available. Therefore, inside that function, `loadedPosts` is received as a parameter, and the final user interface elements can be rendered.

The `Suspense` component that's used as a wrapper around `<Await>` defines some fallback content that is rendered as long as the deferred data is not yet available. In Chapter 10, Behind the Scenes of React and Optimization Opportunities, the `Suspense` component was used to show some fallback content until the missing code was downloaded. Now, it's used to bridge the time until the required data is available.

As shown in Figure 14.8, when returning a `Promise` (and using `<Await>`) like this, other parts of the website, that are not loaded via `<Await>`, are already rendered and displayed while waiting for the posts data.



Sunt Aut Facere Repellat Provident Occaecati Excep
Reprehenderit

Quia Et Suscipit Suscipit Recusandae Consequuntur Expedita Et Cum Reprehenderit Molestiae Ut Ut Quas
Est Autem Sunt Rem Eveniet Architecto

Figure 14.8: Post details are already visible while the list of posts is loading

Another big advantage of returning a `Promise` and awaiting it in the client-side code is that you can easily combine multiple fetching processes and control which processes should be deferred and which ones should not. For example, a route might be fetching different pieces of data. If only one process tends to be slow, you could defer only the slow one like this:

```
export async function loader() {
  return {
        posts: getPosts(), // slow operation => deferred
        userData: await getUserData() // fast operation => NOT deferred
     };
}
```

In this example, `getUserData()` is not deferred because the `await` keyword is added in front of it. Therefore, JavaScript waits for that `Promise` (the `Promise` returned by `getUserData()`) to resolve before returning from `loader()`. Hence, the route component is rendered once `getUserData()` finishes but before `getPosts()` is done.

Summary and Key Takeaways

- React Router can help you with data fetching and submission.
- You can register `loader()` functions for your routes, causing data fetching to be initialized as a route becomes active.
- `loader()` functions return data (or responses, wrapping data) that can be accessed via `useLoaderData()` in your component functions.
- `loader()` data can be used across components via `useRouteLoaderData()`.
- You can also register `action()` functions on your routes that are triggered upon form submissions.
- To trigger `action()` functions, you must use React Router's `<Form>` component or submit data programmatically via `useSubmit()` or `useFetcher()`.
- `useFetcher()` can be used to load or submit data without initiating a route transition.
- When fetching slow data, you can return promises without awaiting them in the `loader()` to defer loading some or all of a route's data.

What's Next?

Fetching and submitting data are extremely common tasks, especially when building more complex React applications.

Typically, those tasks are closely connected to route transitions, and React Router is the perfect tool for handling this kind of operation. That's why the React Router package offers powerful data management capabilities that vastly simplify these processes.

In this chapter, you learned how React Router assists you with fetching or submitting data and which advanced features help you handle both basic and more complex data manipulation scenarios.

Therefore, this chapter concludes the list of core React Router features you need to know.

The next chapters will explore React's server-side capabilities and how you may build fullstack applications with React, load data on a server, and use the Next.js framework.

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 found at https://github.com/TalipotTech/ReactConcepts/blob/manage-router-data/exercises/questions-answers.md

:

1. How are data fetching and submission related to routing?
2. What's the purpose of `loader()` functions?
3. What's the purpose of `action()` functions?
4. What's the difference between `<Form>` and `<form>`?
5. What's the difference between `useSubmit()` and `useFetcher()`?
6. What's the idea behind returning promises instead of awaiting them in a `loader()`?

Apply What You Learned

Apply your knowledge about routing, combined with data manipulation, to the following activity.

Activity 14.1: A To-Dos App

In this activity, your task is to create a basic to-do list web app that allows users to manage their daily to-do tasks. The finished page must allow users to add to-do items, update to-do items, delete to-do items, and view a list of to-do items.

The following paths must be supported:

- `/`: The main page, responsible for loading and displaying a list of to-do items
- `/new`: A page, opened as a modal above the main page, allowing users to add a new to-do item
- `/:id`: A page, also opened as a modal above the main page, allowing users to update or delete a selected to-do item

If no to-do items exist yet, a fitting info message should be shown on the `/` page. If users try to visit `/:id` with an invalid to-do ID, an error modal should be displayed.

Note

For this activity, there is no backend API you could use. Instead, use `localStorage` to manage the to-do data. Keep in mind that the `loader()` and `action()` functions are executed on the client side and can therefore use any browser APIs, including `localStorage`.

You will find example implementations for adding, updating, deleting, and getting to-do items
from `localStorage` at https://github.com/TalipotTech/ReactConcepts/blob/manage-router-data/activities/practice-1/src/data/todos.js

.

Also, don't be confused by the pages that open as modals above other pages. Ultimately, these are simply nested pages, styled as modal overlays. In case you get stuck, you can use the example `Modal` wrapper component found at https://github.com/TalipotTech/ReactConcepts/blob/manage-router-data/activities/practice-1/src/components/Modal.jsx

.

For this activity, you can write all CSS styles on your own if you so choose. But if you want to focus on the React and JavaScript logic, you can also use the finished CSS file from the solution at https://github.com/TalipotTech/ReactConcepts/blob/manage-router-data/activities/practice-1/src/index.css

.

If you use that file, explore it carefully to ensure you understand which IDs or CSS classes might need to be added to certain JSX elements of your solution.

To complete the activity, perform the following steps:

1. Create a new React project and install the React Router package.
2. Create components (with the content shown in the screenshots below) that will be loaded for the three required pages. Also, add links (or programmatic navigation) between these pages.
3. Enable routing and add the route definitions for the three pages.
4. Create `loader()` functions to load (and use) all the data needed by the individual pages.
5. Add `action()` functions for adding, updating, and deleting to-dos.

Hint: If you need to submit multiple forms for different actions from the same page, you could include a hidden input field that sets some value you can check for in your `action()` function, e.g., `<input type="hidden" name="_method" value="DELETE">`. Alternatively, you can also set `<Form method="delete">` (or set it to `"patch"`, `"put"`, or other HTTP verbs) and check for `request.method` in your `action()` function.

6. Add error handling in case data loading or saving fails.

The finished pages should look like this:

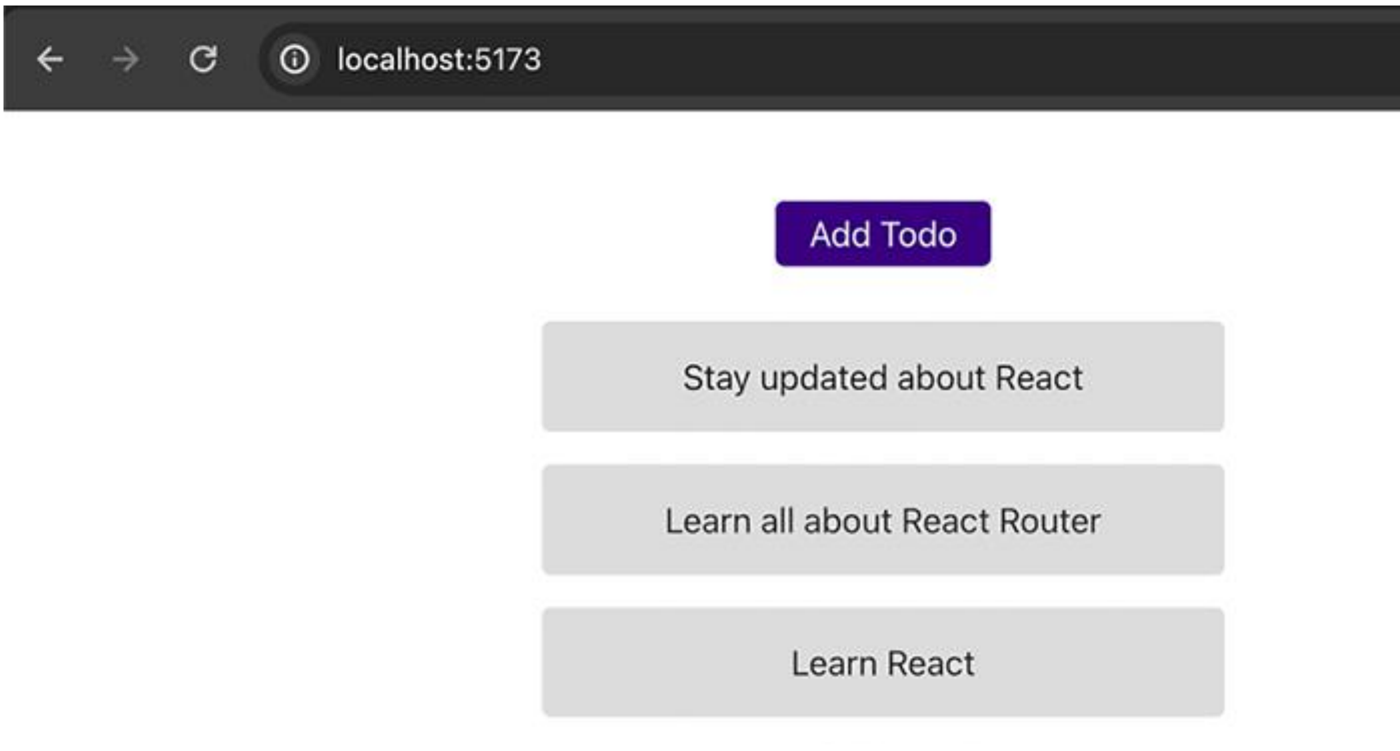


Figure 14.9: The main page displaying a list of to-dos

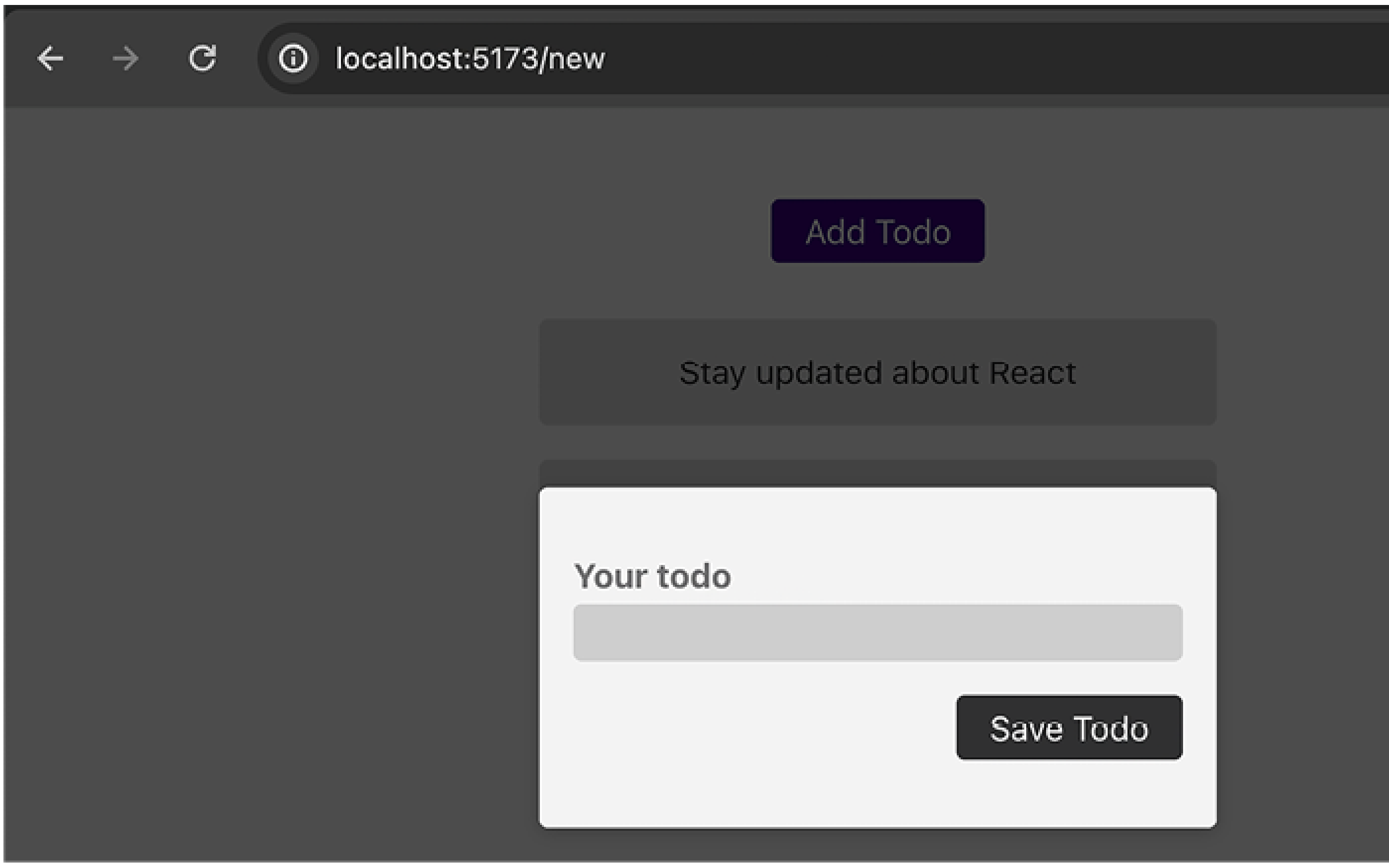


Figure 14.10: The /new page, opened as a modal, allowing users to add a new to-do

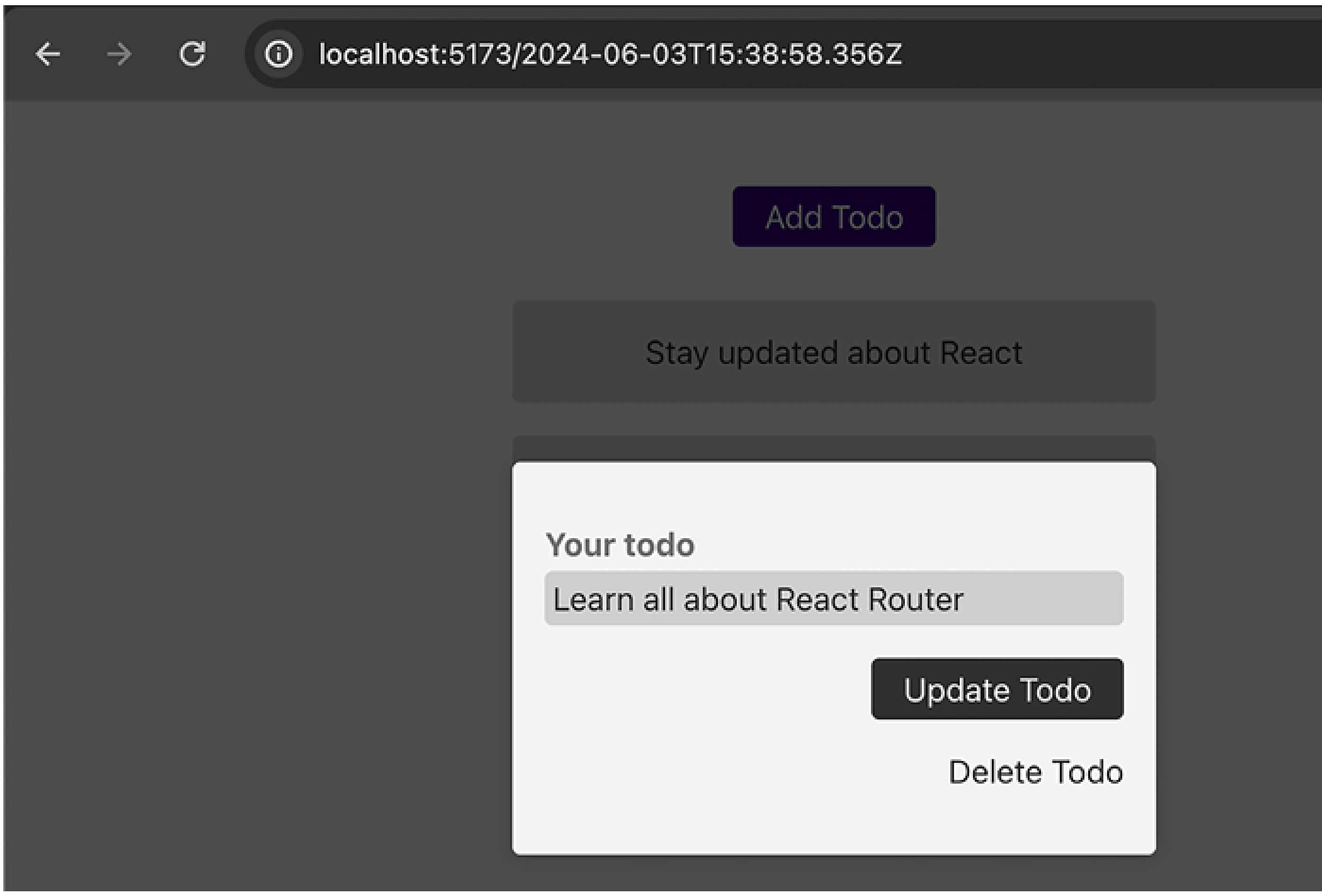

localhost:5173/2024-06-03T15:38:58.356Z
Add Todo
Stay updated about React
Your todo
Learn all about React Router
Update Todo
Delete Todo

Figure 14.11: The /:id page, also opened as a modal, allowing users to edit or delete a to-do

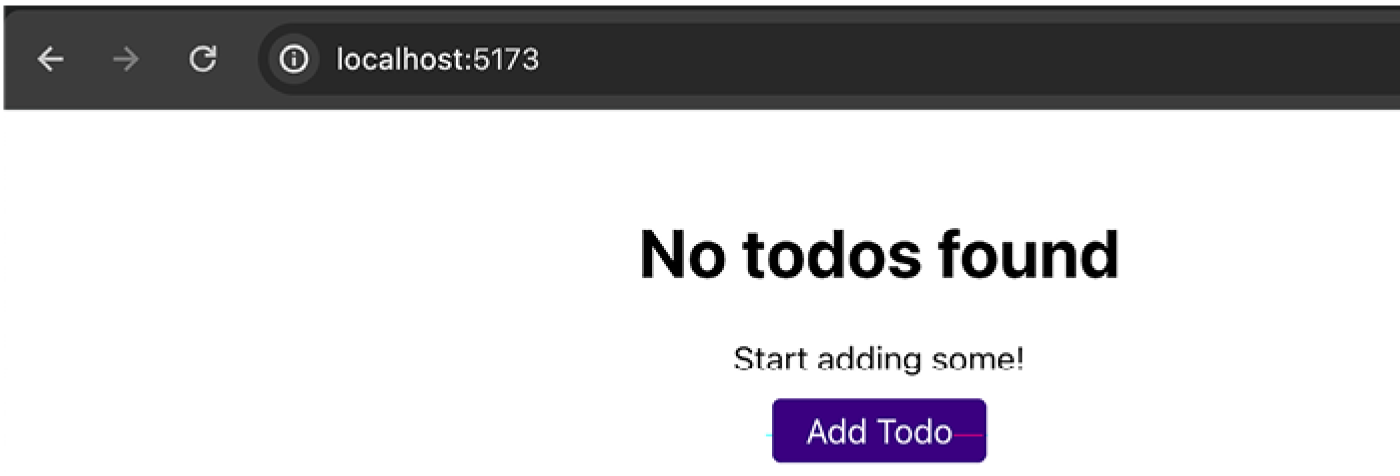


Figure 14.12: An info message, displayed if no to-dos were found

Note

The full code, and solution, to this activity can be found at https://github.com/TalipotTech/ReactConcepts/tree/manage-router-data/activities/practice-1

.

Understanding React Suspense & The use() Hook

Learning Objectives

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

- Describe the purpose and functionality of React's Suspense feature
- Use Suspense with RSCs to show fallback content on a granular level
- Use Suspense for client components via React's `use()` Hook
- Apply different Suspense strategies for data fetching and fallback content

Introduction

In this chapter, you'll learn that React's `Suspense` component used for data fetching to show some temporary content while data is being loaded (e.g., from a database). Though, as you will also learn, `Suspense` can only be used for data fetching if the data is fetched in a certain way.

In addition, this chapter will revisit the `use()` Hook.As you will learn, besides using it for getting access to context values, this Hook can be used in conjunction with `Suspense` as well.

Showing Granular Fallback Content with Suspense

When fetching data or downloading a resource (e.g., a code file), loading delays can occur—delays that can lead to a bad user experience. You should therefore consider showing some temporary fallback content while waiting for the requested resource.

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 handleUpdateStartDate(event) {
    setStartDate(event.target.value);
  }
  function handleUpdateEndDate(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={handleUpdateStartDate}
        />
      </div>
      <div className={classes.control}>
        <label htmlFor="end">End Date</label>
        <input
          id="end"
          type="date"
          value={endDate}
          onChange={handleUpdateEndDate}
        />
      </div>
      <p className={classes.difference}>
        Difference: {daysDiff} days
      </p>
    </div>
  );
}
export default DateCalculator;
```

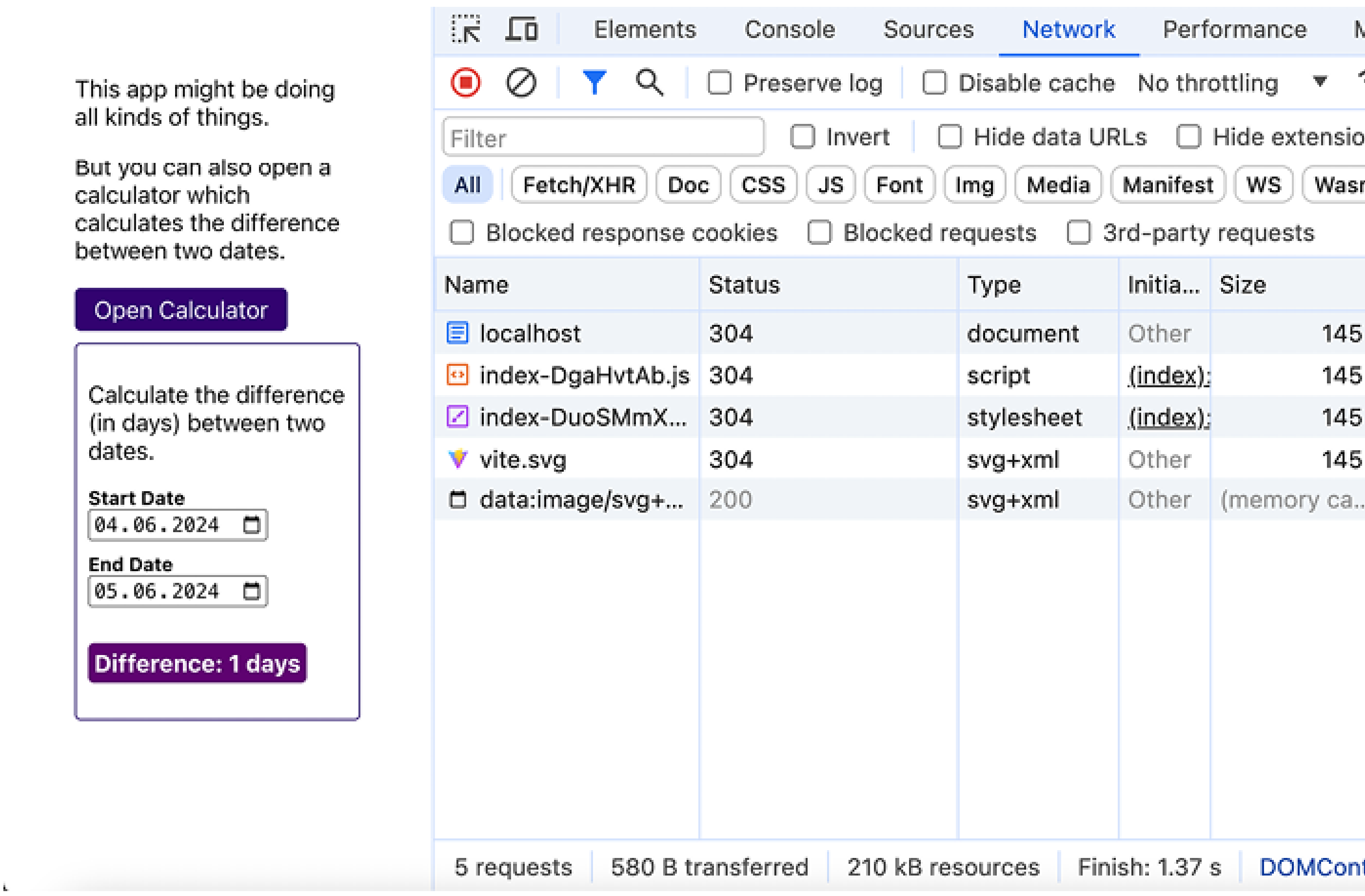
This `DateCalculator` component is then rendered conditionally by the `SuspenseEx.jsx` component:

```
import { useState } from 'react';
import DateCalculator from './components/DateCalculator.jsx';
function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);
  function handleOpenDateCalc() {
    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={handleOpenDateCalc}>Open Calculator</button>
      {showDateCalc && <DateCalculator />}
    </>
  );
}
export default SuspenseEx;
```

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).

After building the app for production (via `npm run build`), when previewing that production version (via `npm run preview`), you can see one main code bundle file being downloaded in the following screenshot:


This app might be doing all kinds of things.

But you can also open a calculator which calculates the difference between two dates.

Calculate the difference (in days) between two dates.

Start Date 04.06.2024

End Date 05.06.2024

Difference: 1 days

Name	Status	Type	Initia...	Size
localhost	304	document	Other	145
index-DgaHvtAb.js	304	script	(index):	145
index-DuoSMmX...	304	stylesheet	(index):	145
vite.svg	304	svg+xml	Other	145
data:image/svg+...	200	svg+xml	Other	(memory ca...

5 requests | 580 B transferred | 210 kB resources | Finish: 1.37 s | DOMCont

Figure 10.10: One main bundle file is downloaded

The `Network` tab in the browser's developer tools reveals outgoing network requests. As you can see in the screenshot, one main 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

Dynamic imports are a native JavaScript feature that allows for dynamically importing JavaScript code files. For further information on this topic, visit https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import.

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.jsx'
  )
);
function App() {
   const [showDateCalc, setShowDateCalc] = useState(false);
   function handleOpenDateCalc() {
      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={handleOpenDateCalc}>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:

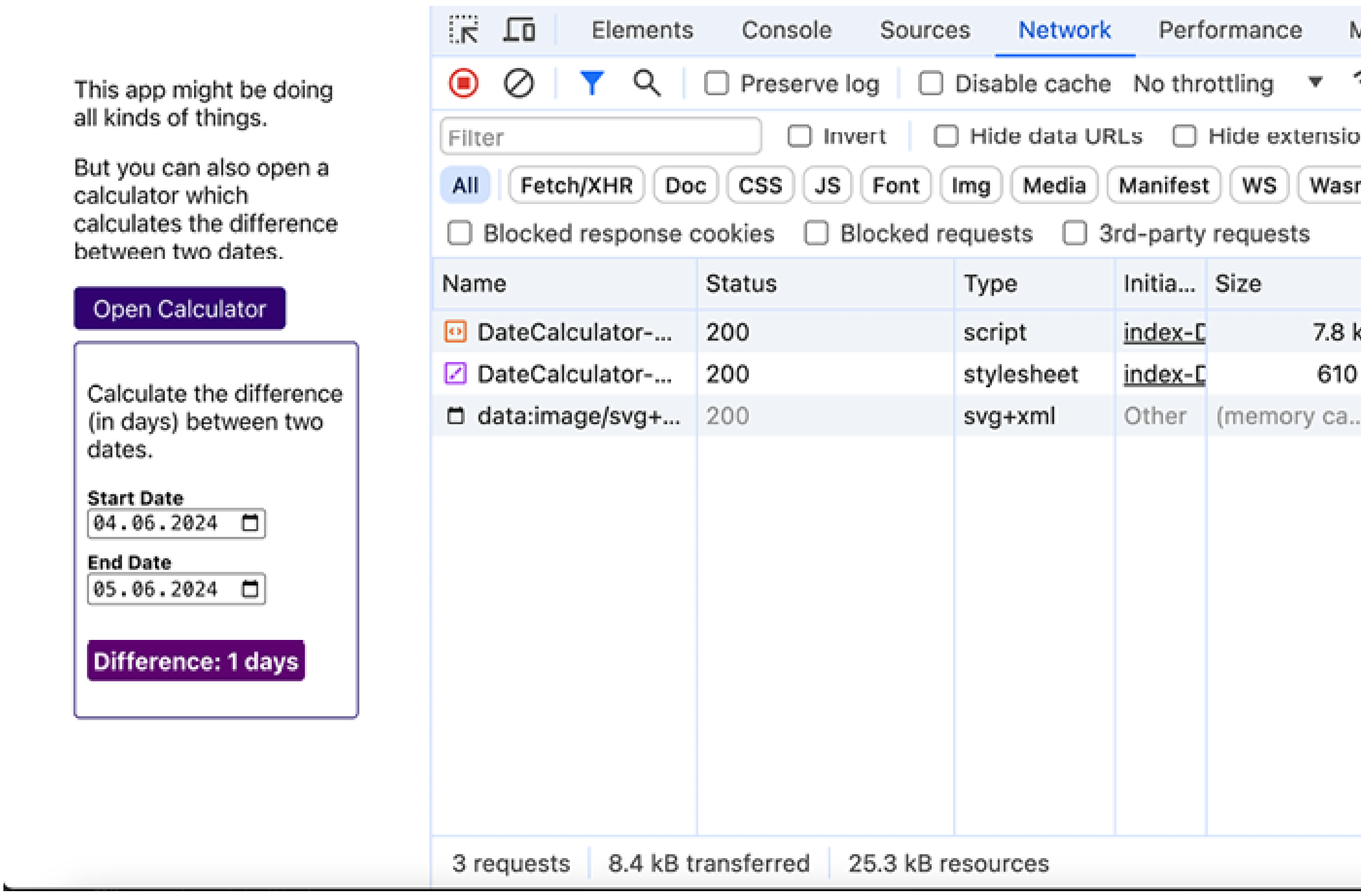
Note

You can find the finished example code on GitHub at https://github.com/TalipotTech/ReactConcepts/tree/10-behind-scenes/examples/06-code-splitting

`Suspense` is a component built into React that aims to display fallback content while some resource or data is loading. Therefore, when using it for lazy loading, 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. 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.

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:


This app might be doing all kinds of things.

But you can also open a calculator which calculates the difference between two dates.

Calculate the difference (in days) between two dates.

Start Date 04.06.2024

End Date 05.06.2024

Difference: 1 days

Name	Status	Type	Initia...	Size
DateCalculator-...	200	script	index-D	7.8 k
DateCalculator-...	200	stylesheet	index-D	610
data:image/svg+...	200	svg+xml	Other	(memory ca...

3 requests | 8.4 kB transferred | 25.3 kB resources

Figure 10.11: After clicking the button, an extra code file is 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 anyway. Only consider using it on conditionally loaded components.

For that reason, to simplify the process of rendering fallback content while waiting for some resource, React offers its `Suspense` component. You can use the `Suspense` component as a wrapper around React elements that fetch some code or data. For example, when using it in the context of code splitting, you can show some temporary fallback content like this:

```
import { lazy, Suspense, useState } from 'react';
const DateCalculator = lazy(() => import(
    './components/DateCalculator.jsx'
  )
);
function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);
  function handleOpenDateCalc() {
    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={handleOpenDateCalc}>Open Calculator</button>
       <Suspense fallback={<p>Loading...</p>}>
    {showDateCalc && <DateCalculator />}
       </Suspense>
  </>
  );
}
```

In this example (which is from a regular Vite-based React project), React's `Suspense` component is wrapped around the conditionally rendered `DateCalculator` component. `DateCalculator` is created with the help of React's `lazy()` function, which is used to lazily (i.e., on demand) load the code bundle that belongs to this component.

As a result, the entire other page content is shown right from the start. Only the conditionally displayed `DateCalculator` component is replaced with the fallback content (`<p>Loading...</p>`) while the code is being fetched. Thus, `Suspense` is used to render some fallback JSX code on a very granular level. Instead of replacing the entire page or component markup with some temporary content, only a small part of the UI is replaced.

Of course, `Suspense` therefore provides a functionality that would also be nice to have when fetching data—after all, delays occur frequently there, too.

Using Suspense for Data Fetching with Next.js

As explained in the previous chapter, in the Managing Loading States with Next.js section, the process of data fetching also often comes with waiting times that can negatively impact user experience. That's why, in that same section, you learned that Next.js allows you to define a `loading.js` file that contains some fallback component that's rendered during such a delay.

However, using that approach essentially replaces the entire page (or the main area of that page) with the loading fallback component content. But that's not always ideal—you instead might want to display some loading fallback content on a more granular level when fetching data.

Thankfully, in Next.js projects, you can use `Suspense` in a similar way, as shown in the example from the previous section, to wrap it around components that fetch data. Since Next.js supports HTTP response streaming, it's able to render the rest of the page immediately while streaming the content that depends on the fetched data to the client side once it's available. Until the data is loaded and available, `Suspense` will render its defined fallback.

Therefore, coming back to the example from the Managing Loading States with Next.js section of Chapter 16, React Server Components & Server Actions, you can take advantage of `Suspense` by outsourcing the data fetching code into a separate `UserGoals` component:

```
import fs from 'node:fs/promises';
async function fetchGoals() {
  await new Promise((resolve) => setTimeout(resolve, 3000)); // delay
  const goals = await fs.readFile('./data/user-goals.json', 'utf-8');
  return JSON.parse(goals);
}
export default async function UserGoals() {
  const fetchedGoals = await fetchGoals();
  return (
      <ul>
    {fetchedGoals.map((goal) => (
      <li key={goal}>{goal}</li>
    ))}
   </ul>
  );
}
```

This `UserGoals` component can then be wrapped with `Suspense` in the `GoalsPage` component like this:

```
import { Suspense } from 'react';
import UserGoals from '../../components/UserGoals';
export default async function GoalsPage() {
  return (
      <>
    <h1>Top User Goals</h1>
    <Suspense fallback={
      <p id="fallback">Fetching user goals...</p>}
    >
      <UserGoals />
    </Suspense>
   </>
  );
}
```
CopyExplain

This code now utilizes React's `Suspense` component to show a fallback paragraph while the `UserGoals` component is fetching data.

Note

You can find the complete demo project code on GitHub: https://github.com/TalipotTech/ReactConcepts/tree/17-suspense-use/examples/02-data-fetching-suspense

As a result, when users navigate to `/goals`, they immediately see the title (the `<h1>` element) in combination with the fallback content. There is no need for a separate `loading.js` file anymore.

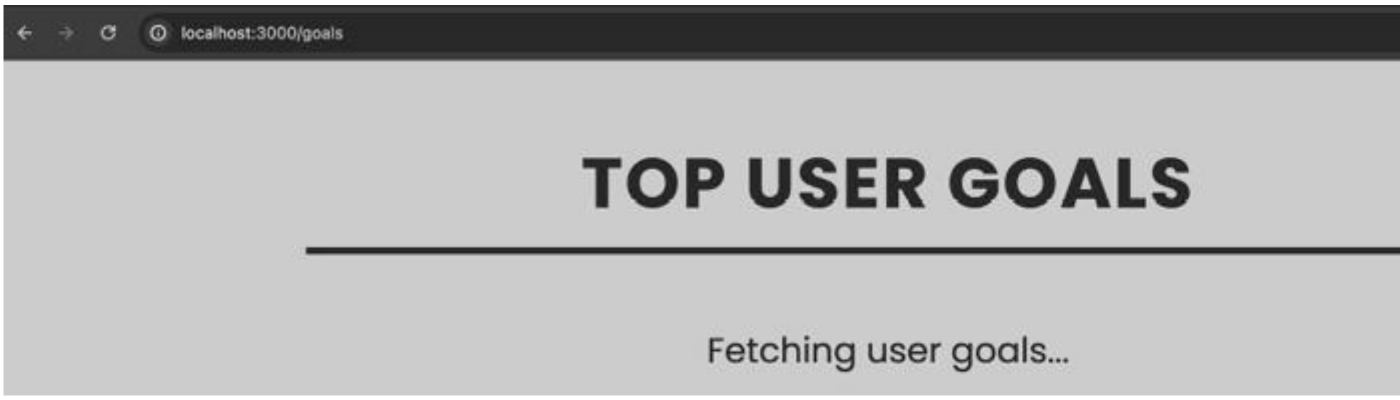


Figure 17.1: The fallback content is shown as part of the target page, instead of entirely replacing it

However, the advantage of using `Suspense` in this situation is not just that the `loading.js` file isn't needed anymore. Instead, data fetching and fallback content can now be managed on a very granular level.

For example, in a more complex online shop application, you could have a component like this:

```
function ShopOverviewPage() {
  return (
    <>
   <header>
     <h1>Find your next deal!</h1>
     <MainNavigation />
   </header>
   <main>
     <Suspense fallback={<DailyDealSkeleton />}>
       <DailyDeal />
     </Suspense>
     <section id="search">
       <h2>Looking for something specific?</h2>
       <Search />
     </section>
     <Suspense fallback={<p>Fetching products...</p>}>
       <Products />
     </Suspense>
   </main>
  </>
  );
}
```

In this example, the `<header>` and `<section id="search">` elements are always visible and rendered. On the other hand, `<DailyDeal />` and `<Products />` are only rendered once their data has been fetched. Until then, their respective fallbacks are displayed.

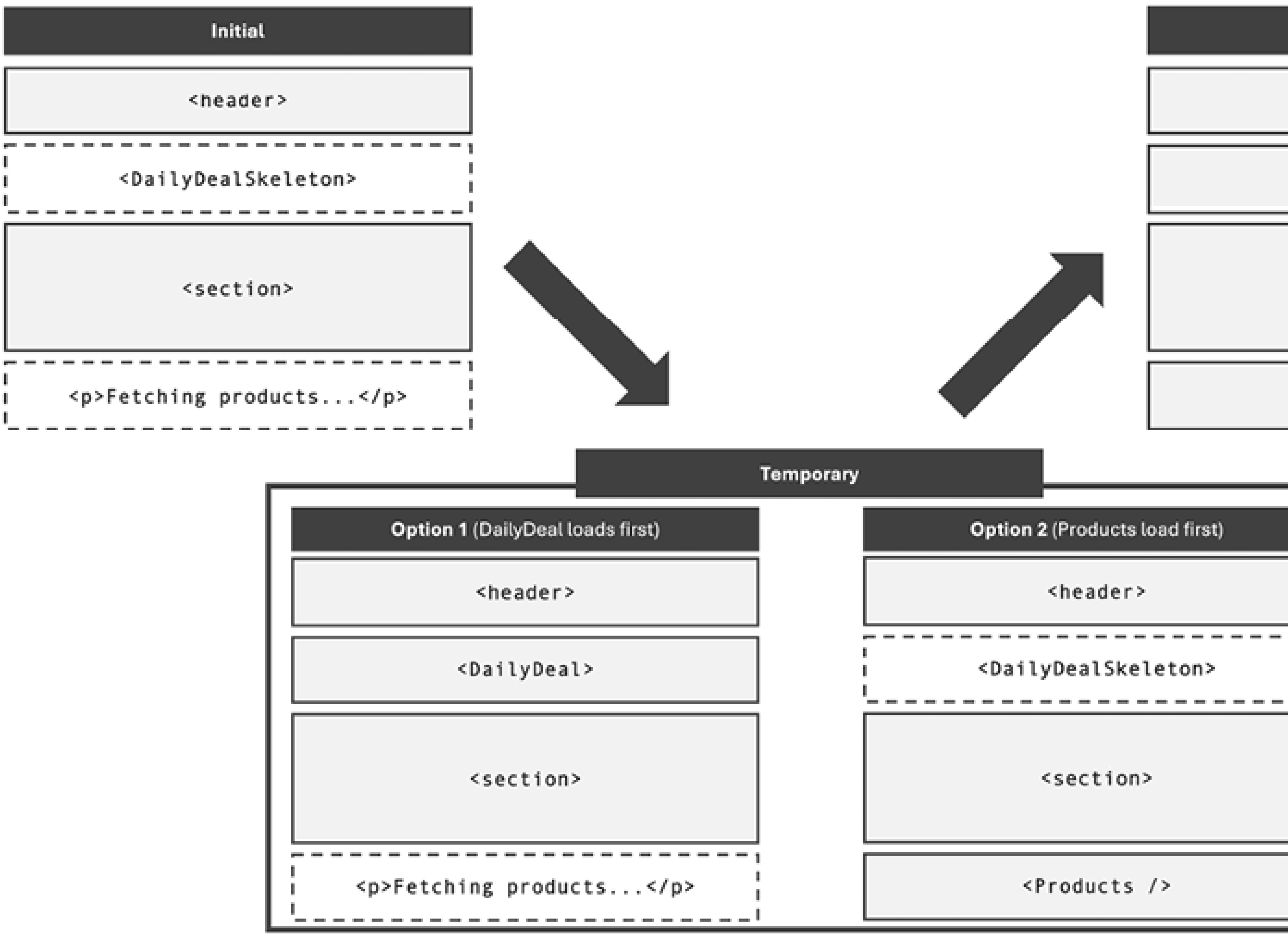


Figure 17.2: Placeholders are shown initially until loaded data is streamed in and rendered to the screen

`<DailyDeal />` and `<Products />` will be loaded and rendered independently from each other since they're wrapped by two different `Suspense` blocks. Consequently, users will immediately see the header and search area, and then eventually see the daily deal and products—though either of the two may load and render first.

What's important about these examples is that the components wrapped by `Suspense` are RSCs that use `async/await`. As you will learn in the next section, not all React components will interact with the `Suspense` component. But React Server Components, in Next.js projects, will.

Using Suspense in Other React Projects—Possible, But Tricky

The previous section explored how you may take advantage of `Suspense` for data fetching with RSCs in Next.js projects.

However, `Suspense` is not a Next.js-specific feature or concept—instead, it's provided by React itself. Consequently, you can use it in any React project to show fallback content while data is being fetched.

At least, that's the theory. But as it turns out, you can't use it with all components and data fetching strategies.

Suspense Does Not Work with useEffect()

Since fetching data via `useEffect()` is a common strategy, you might be inclined to use `Suspense` in conjunction with this Hook to show some fallback content while data is being loaded via the effect function.

For example, the following `BlogPosts` components uses `useEffect()` to load and display some blog posts:

```
import { useEffect, useState } from 'react';
function BlogPosts() {
  const [posts, setPosts] = useState([]);
  useEffect(() => {
    async function fetchBlogPosts() {
      // simulate slow network
      await new Promise((resolve) => setTimeout(resolve, 3000));
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );
      const posts = await response.json();
      setPosts(posts);
    }
    fetchBlogPosts();
  }, []);
  return (
    <ul>
```

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

You could wrap this component with `Suspense` like this:

```
import { Suspense } from 'react';
import BlogPosts from './components/BlogPosts.jsx';
function App() {
  return (
      <>
    <h1>All posts</h1>
    <Suspense fallback={<p>Fetching blog posts...</p>}>
      <BlogPosts />
    </Suspense>
  </>
  );
}
```

Unfortunately, this will not work in the intended way, though. Instead of displaying the fallback content, nothing will be rendered while the data is being fetched.

The reason for this behavior is that `Suspense` is intended to suspend when fetching data during the component rendering process—not when fetching inside of some effect function.

It helps to recall how `useEffect()` works: the effect function is executed after the component function is executed, i.e., after the first component render cycle is done.

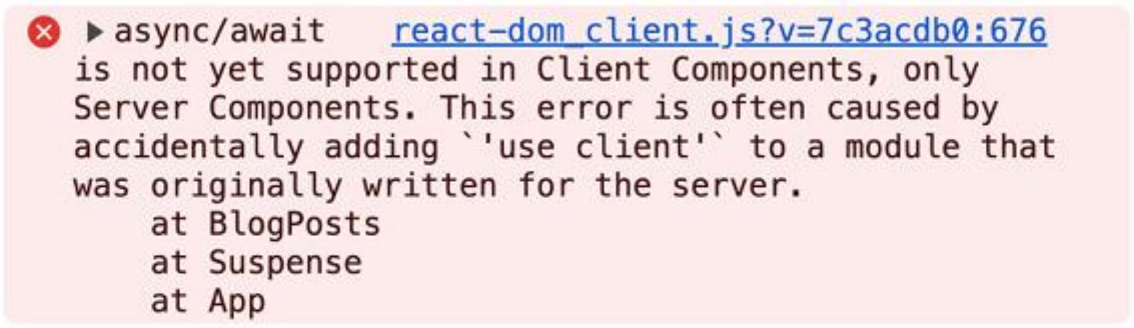
As a result, you can't use `Suspense` to show fallback content when fetching data via `useEffect()`. Instead, in those cases, you need to manually manage and use some loading state in the component that performs the data fetching (i.e., by manually managing different state slices like `isLoading`—for example, in the Limitations of useState() and Managing State with useReducer() sections.

Fetching Data while Rendering—the Incorrect Way

Since `Suspense` intends to show fallback content while a component is fetching data during its rendering process, you could try to re-write the `BlogPosts` component to look like this:

```
async function BlogPosts() {
  await new Promise((resolve) => setTimeout(resolve, 3000));
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  const posts = await response.json();
  return (
    <ul>
   {posts.map((post) => (
    <li key={post.id}>{post.title}</li>
   ))}
  </ul>
  );
}
```

But trying to use this code will yield an error in the browser developer tools:



```
▶async/await    react-dom_client.js?v=7c3acdb0:676
is not yet supported in Client Components, only
Server Components. This error is often caused by
accidentally adding `'use client'` to a module that
was originally written for the server.
    at BlogPosts
    at Suspense
    at App
```

Figure 17.3: React complains about async components on the client side

React does not support the usage of `async/await` in client components. Only React Server Components may use that syntax (and therefore return promises). Consequently, regular React projects, which are not set up to support RSCs, can't use this solution.

Of course, you could come up with a (problematic) alternative solution like this:

```
function BlogPosts() {
  const [posts, setPosts] = useState([]);
  new Promise(() => setTimeout(() => {
    return fetch(
      'https://jsonplaceholder.typicode.com/posts'
    ).then(response => response.json())
     .then(fetchedPosts => setPosts(fetchedPosts));
  }, 3000));
  return (
    <ul>
   {posts.map((post) => (
     <li key={post.id}>{post.title}</li>
   ))}
  </ul>
  );
}
```

The code creates an infinite loop.

So, fetching data as part of a component's rendering process is really difficult when not working with RSCs.

Getting Suspense Support Is Tricky

Since `Suspense` requires data fetching to occur during the rendering process, which is difficult to set up manually, the React documentation (https://react.dev/reference/react/Suspense#displaying-a-fallback-while-content-is-loading) itself mentions that "only Suspense-enabled data sources will activate the Suspense component," further stating that those data sources include:

- Data fetching with Suspense-enabled frameworks like Relay and Next.js
- Lazy-loading components code with `lazy()`
- Reading the value of a Promise with `use()`

On the same page, the official documentation highlights that "Suspense-enabled data fetching without the use of an opinionated framework is not yet supported."

Note

Documentation may change over time—and so may React. But even though the exact wording may differ at the point of time you're reading this, the way of using `Suspense`, and the fact that it can't be used without special libraries or features like `lazy()`, is highly unlikely to change.

Therefore, unless you plan on building your own Suspense-enabled library, you either have to stick to using `Suspense` for code-splitting (via `lazy()`), use a third-party framework or library that integrates with `Suspense`, or explore the usage of that `use()` Hook.

Using Suspense for Data Fetching with Supporting Libraries

As you learned in the Using Suspense for Data Fetching with Next.js section, you can use `Suspense` for data fetching when working with Next.js. But while Next.js is one of the most popular React frameworks that supports `Suspense`, it's not the only option you have.

For example, TanStack Query (formerly known as React Query) is another popular third-party library that unlocks `Suspense` for data fetching. This library, unlike Next.js, is not a library that aims to help with building full-stack React apps or running code on the server side, though. Instead, TanStack Query is a library that's all about helping with client-side data fetching, data mutations, and asynchronous state management. Since it runs on the client side, it therefore works in React projects that do not integrate with SSR and RSCs, too—although you can also use it in such projects.

TanStack Query is a complex, feature-rich library—we could probably write an entire book about it. But the following short code snippet (which is from a Vite-based project, not from a Next.js project) shows how you may fetch data with the help of that library:

```
import { useSuspenseQuery } from '@tanstack/react-query';
async function fetchPosts() {
  await new Promise((resolve) => setTimeout(resolve, 3000));
  const response = await fetch('https://jsonplaceholder.typicode.com/posts');
  const posts = await response.json();
  return posts;
}
function BlogPosts() {
  const {data} = useSuspenseQuery({
    queryKey: ['posts'],
    queryFn: fetchPosts
  });
  return (
    <ul>
   {data.map((post) => (
    <li key={post.id}>{post.title}</li>
   ))}
  </ul>
  );
}
```

In this example, the `BlogPosts` component uses TanStack Query's `useSuspenseQuery()` Hook, in conjunction with a custom `fetchPosts()` function, to fetch data via an HTTP request. As the name of the Hook implies, it integrates with React's `Suspense` component.

As a result, the `BlogPosts` component can then be wrapped with `Suspense` like this:

```
import { Suspense } from 'react';
import BlogPosts from './components/BlogPosts.jsx';
function App() {
  return (
    <>
   <h1>All posts</h1>
   <Suspense fallback={<p>Fetching blog posts...</p>}>
    <BlogPosts />
   </Suspense>
  </>
  );
}
```

As you can tell, `Suspense` is used in the same way it was used with `lazy()` or Next.js. So, its functionality and usage don't change—if you're wrapping it around a component that integrates with `Suspense` (like `BlogPost` does, via TanStack Query's `useSuspenseQuery()` Hook), `Suspense` can be used to output some fallback content while some data fetching process is underway.

Note

You can find the complete example project on GitHub: https://github.com/TalipotTech/ReactConcepts/tree/17-suspense-use/examples/05-tanstack-query

Of course, this is just a simple example. You can do more with TanStack Query, and there also are other libraries that can be used in conjunction with `Suspense`. It's just important to understand that there are other options than Next.js. But it's also crucial to keep in mind that not all code (and also not all libraries) will work with `Suspense`.

Besides using libraries that directly integrate with `Suspense` (like TanStack Query via its `useSuspenseQuery()` Hook), you can also use `Suspense` for data fetching with the help of React's built-in `use()` Hook.

use()ing Data while Rendering

You can use the `use()` Hook during a component's rendering process to extract and use the value of a promise. `use()` will automatically interact with any wrapping `Suspense` component and let it know about the current status of the data fetching process (i.e., if the promise has been resolved or not).

The example from the Fetching Data while Rendering—the Incorrect Way section can therefore be adjusted to use the `use()` Hook like this:

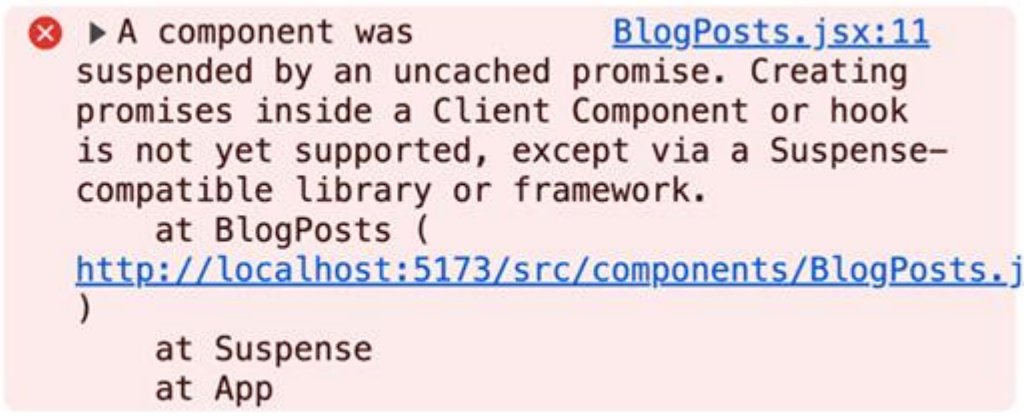
```
import { use } from 'react';
async function fetchPosts() {
  await new Promise((resolve) => setTimeout(resolve, 3000));
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  const posts = await response.json();
  return posts;
}
function BlogPosts() {
  const posts = use(fetchPosts());
  return (
    <ul>
   {posts.map((post) => (
    <li key={post.id}>{post.title}</li>
    ))}
   </ul>
   );
}
```

The `BlogPosts` component is now no longer a component that uses `async/await`. Instead, it uses the imported `use()` Hook to read the value of the promise produced by calling `fetchPosts()`.

As mentioned, `use()` interacts with `Suspense`, hence `BlogPosts` can be wrapped with `Suspense` like this:

```
import { Suspense } from 'react';
import BlogPosts from './components/BlogPosts.jsx';
function App() {
  return (
      <>
     <h1>All posts</h1>
     <Suspense fallback={<p>Fetching blog posts...</p>}>
       <BlogPosts />
     </Suspense>
   </>
   );
}
```

When running this code, it might work as intended (depending on the React version you're using), but it's more likely to not yield any results or even show an error message in the browser developer tools:



```
▶A component was          BlogPosts.jsx:11
suspended by an uncached promise. Creating
promises inside a Client Component or hook
is not yet supported, except via a Suspense-
compatible library or framework.
    at BlogPosts (
http://localhost:5173/src/components/BlogPosts.j
)
    at Suspense
    at App
```

Figure 17.4: The use() Hook only works with promises created by Suspense-compatible libraries

As explained by this error message, the `use()` Hook is not intended to be used with regular promises as created in the previous example. Instead, it should be used on promises that are provided by Suspense-compatible libraries or frameworks.

Note

If you want to go against the official recommendation and try to build promises that support `use()` and `Suspense`, you can explore the official `Suspense` demo projects linked in the official React documentation (https://19.react.dev/reference/react/Suspense)—for example, this project: https://codesandbox.io/p/sandbox/strange-black-6j7nnj.

Please note that, as mentioned in the documentation, the approach used in that demo project uses unstable APIs and may not work with future React versions.

So, again, support from a third-party framework or library is needed. No matter if you try to use `Suspense` with components that fetch data as part of the rendering process with or without `use()`, you end up needing help.

Put in other words: to take advantage of `Suspense`, you either need to directly fetch data via a Suspense-compatible library or framework, or you need to use the `use()` Hook on a promise that's generated by a Suspense-compatible library or framework.

One such framework is, again, Next.js. Besides using `Suspense` around RSCs, as shown in the section Using suspense for Data Fetching with Next.js, you can also use `Suspense` in conjunction with the `use()` Hook on promises produced by Next.js.

Using use() with Promises Created by Next.js

Next.js projects are able to create promises that will work with `use()` and `Suspense`. To be precise, any promise you create in an RSC and pass to a (client) component via props qualifies as a `use()`able promise.

Consider this example code:

```
import fs from 'node:fs/promises';
import UserGoals from '../../components/UserGoals';
```

```
async function fetchGoals() {
  await new Promise((resolve) => setTimeout(resolve, 3000)); // delay
  const goals = await fs.readFile('./data/user-goals.json', 'utf-8');
  return JSON.parse(goals);
}
export default function GoalsPage() {
  const fetchGoalsPromise = fetchGoals();
  return (
    <>
   <h1>Top User Goals</h1>
   <UserGoals promise={fetchGoalsPromise} />
  </>
  );
}
```

In this code snippet, a promise is created by calling `fetchGoals()` and stored in a constant called `fetchGoalsPromise`. The created promise (`fetchGoalsPromise`) is then passed as a value for the `promise` prop to the `UserGoals` component.

Along with another component, this `UserGoals` component is defined in the `UserGoals.js` file like this:

```
import { use, Suspense } from 'react';
function Goals({ fetchGoalsPromise }) {
  const goals = use(fetchGoalsPromise);
  return (
    <ul>
   {goals.map((goal) => (
     <li key={goal}>{goal}</li>
   ))}
  </ul>
  );
}
export default function UserGoals({ promise }) {
  return (
    <Suspense fallback={<p id="fallback">Fetching user goals...</p>}>
   <Goals fetchGoalsPromise={promise} />
  </Suspense>
  );
}
```

In this code example, the `UserGoals` component uses `Suspense` to wrap the `Goals` component to which it essentially forwards the received `promise` prop value (via the `fetchGoalsPromise` prop). The `Goals` component then reads that promise value via the `use()` Hook.

Since the promise is created in an RSC (`GoalsPage`) that's managed by Next.js, React will not complain about this code—Next.js creates promises that work with `use()`. Instead, it will show the fallback content (`<p id="fallback">Fetching user goals...</p>`) while data is being fetched and renders the final user interface once the data has arrived and has been streamed to the client.

As explained before, any elements not wrapped by Suspense (i.e., the `<h1>` element, in this example) will be displayed right from the start.

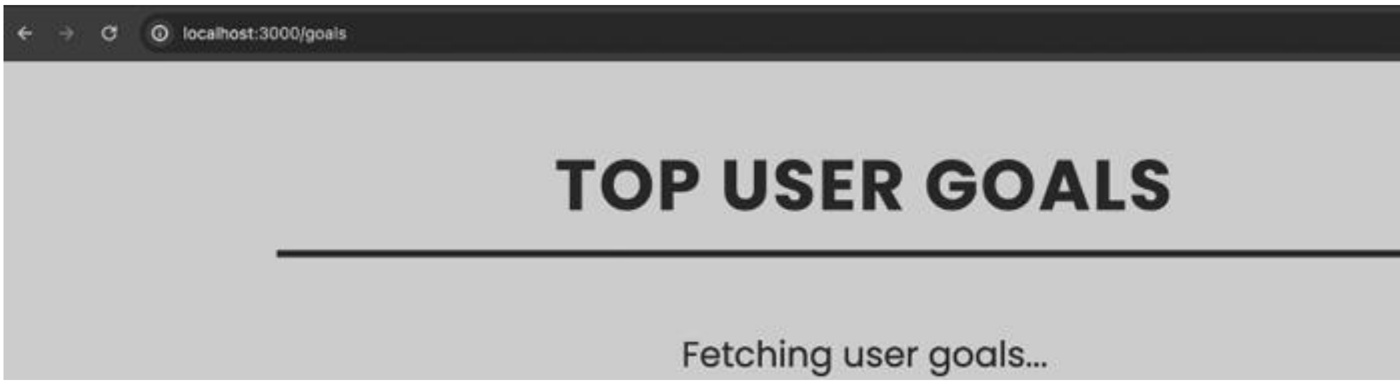


Figure 17.5: The fallback text is shown next to the title while data is fetched via use()

It's also worth noting that both `UserGoals` and `Goals` are RSCs, too—nonetheless, they can use the `use()` Hook.

Normally, Hooks can't be used in RSCs but the `use()` Hook is special. Just as it may be used inside `if` statements or loops (as explained in Chapter 11, Working with Complex State), it can be executed in both server and client components.

However, when working with a server component, you can also simply use `async/await` instead of `use()`. Thus, the `use()` Hook is really only useful when it comes to reading promise values in client components—there, `async/await` is not available.

Using use() in Client Components

Besides using it for accessing context, the `use()` Hook was introduced to help with reading values from promises in client components—i.e., in situations where you can't use `async/await`.

Consider this updated user goals example, where some state is managed and a side effect is triggered:

```
'use client';
import { use, Suspense, useEffect,useState  } from 'react';
// sendAnalytics() is a dummy function that just logs to the console
import { sendAnalytics } from '../lib/analytics';
function Goals({ fetchGoalsPromise }) {
   const [mainGoal, setMainGoal] = useState();
   const goals = use(fetchGoalsPromise);
   function handleSetMainGoal(goal) {
      setMainGoal(goal);
   }
   return (
      <ul>
    {goals.map((goal) => (
      <li
        key={goal}
        id={goal === mainGoal ? 'main-goal' : undefined}
        onClick={() => handleSetMainGoal(goal)}
      >
        {goal}
      </li>
    ))}
   </ul>
   );
}
export default function UserGoals({ promise }) {
   useEffect(() => {
      sendAnalytics('user-goals-loaded', navigator.userAgent);
   }, []);
   return (
      <Suspense fallback={<p id="fallback">Fetching user goals...</p>}>
    <Goals fetchGoalsPromise={promise} />
   </Suspense>
   );
}
```

In this example, the `Goals` component uses `useState()` to manage the information of which goal was marked as the main goal by the user. Furthermore, the `UserGoals` component (which uses `Suspense`) utilizes the `useEffect()` Hook to send an analytics event once the component renders (i.e., before the suspended `Goals` component is displayed). Due to the usage of all these client-side exclusive features, the `'use client'` directive is required.

As a result, `async/await` can't be used in the `Goals` and `UserGoals` components. But since the `use()` Hook can be used in client components, it offers a possible solution for situations like this. And, since this example is from a Next.js application, React will not complain about the kind of promise being consumed by `use()`. Instead, this example code would lead to the fallback content being displayed while the goals data is fetched.

Suspense Usage Patterns

As you have learned, the `Suspense` component can be wrapped around components that fetch data as part of their rendering process—as long as they do it in a compliant way.

Of course, in many projects, you may have multiple components that fetch data and that should display some fallback content while doing so. Thankfully, you can use the `Suspense` component as often as needed—you can even combine multiple `Suspense` components with each other.

Revealing Content Together

Thus far, in all examples, `Suspense` was always wrapped around exactly one component. But there is no rule that would stop you from wrapping `Suspense` around multiple components.

For example, the following code is valid:

```
function Shop() {
  return (
    <>
   <h1>Welcome to our shop!</h1>
   <Suspense fallback={<p>Fetching shop data...</p>}>
```

```
      <DailyDeal />
      <Products />
    </Suspense>
  </>
  );
}
```

In this code snippet, data fetching in the `DailyDeal` and `Products` components starts simultaneously. Since both components are wrapped by one single `Suspense` component, the fallback content is displayed unti both components are done fetching data. So, if one component (e.g., `DailyDeal`) is done after one second, and the other component (`Products`) takes five seconds, both components are only revealed (and replace the fallback content) after five seconds.

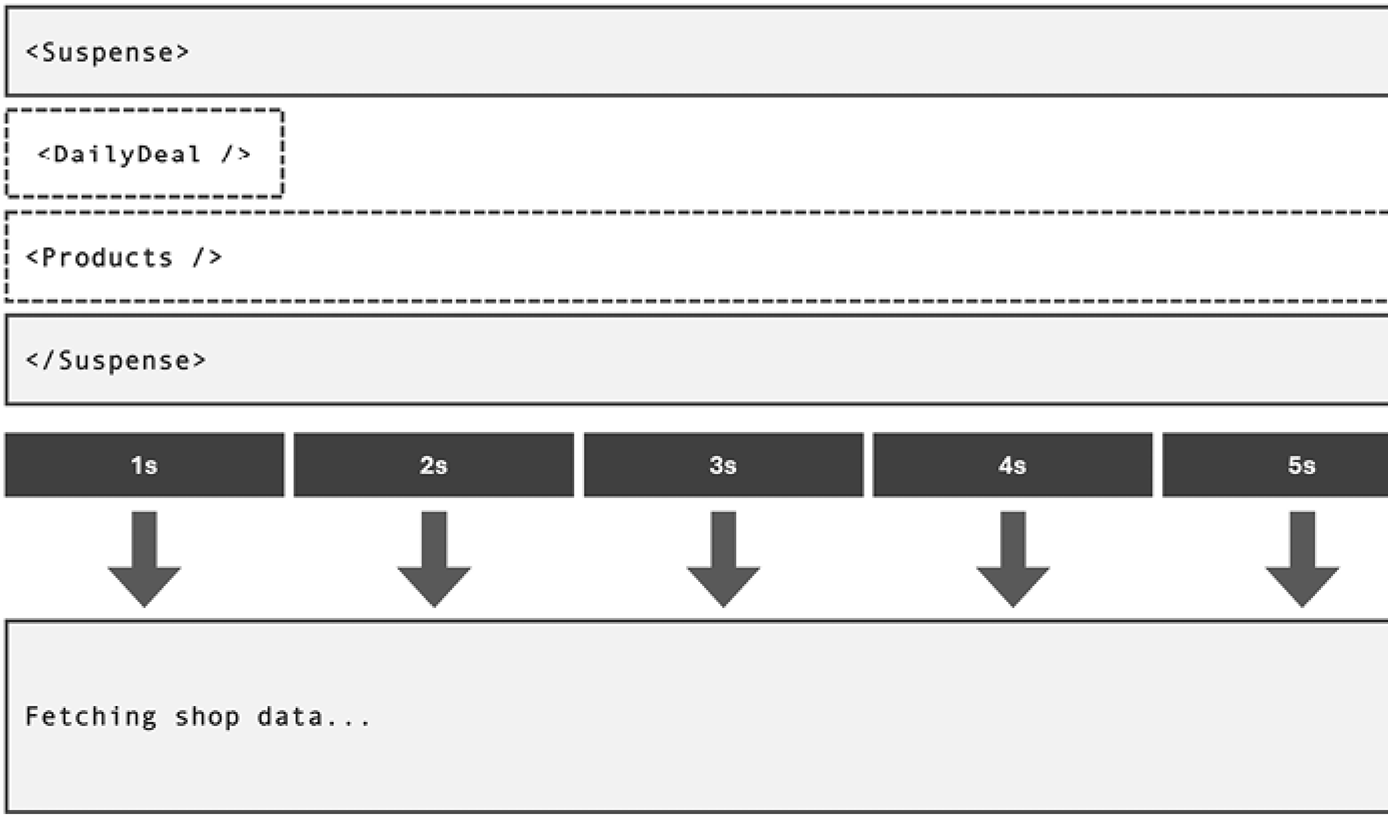


Figure 17.6: Data is fetched in parallel, and fallback content is shown via Suspense until all components are done

Revealing Content as Soon as Possible

Of course, there are situations where you might want to display fallback content for multiple components, but where you don't want to wait for all components to finish fetching data before showing any fetched content.

In such situations, you can use `Suspense` multiple times:

```
function Shop() {
  return (
    <>
   <h1>Welcome to our shop!</h1>
   <Suspense fallback={<p>Fetching daily deal data...</p>}>
     <DailyDeal />
   </Suspense>
   <Suspense fallback={<p>Fetching products data...</p>}>
     <Products />
   </Suspense>
  </>
  );
}
```
CopyExplain

In this adjusted code example, `DailyDeal` and `Products` are wrapped with two different instances of the `Suspense` component. Thus, each component's content will be revealed once available, independent from the other component's data fetching status.

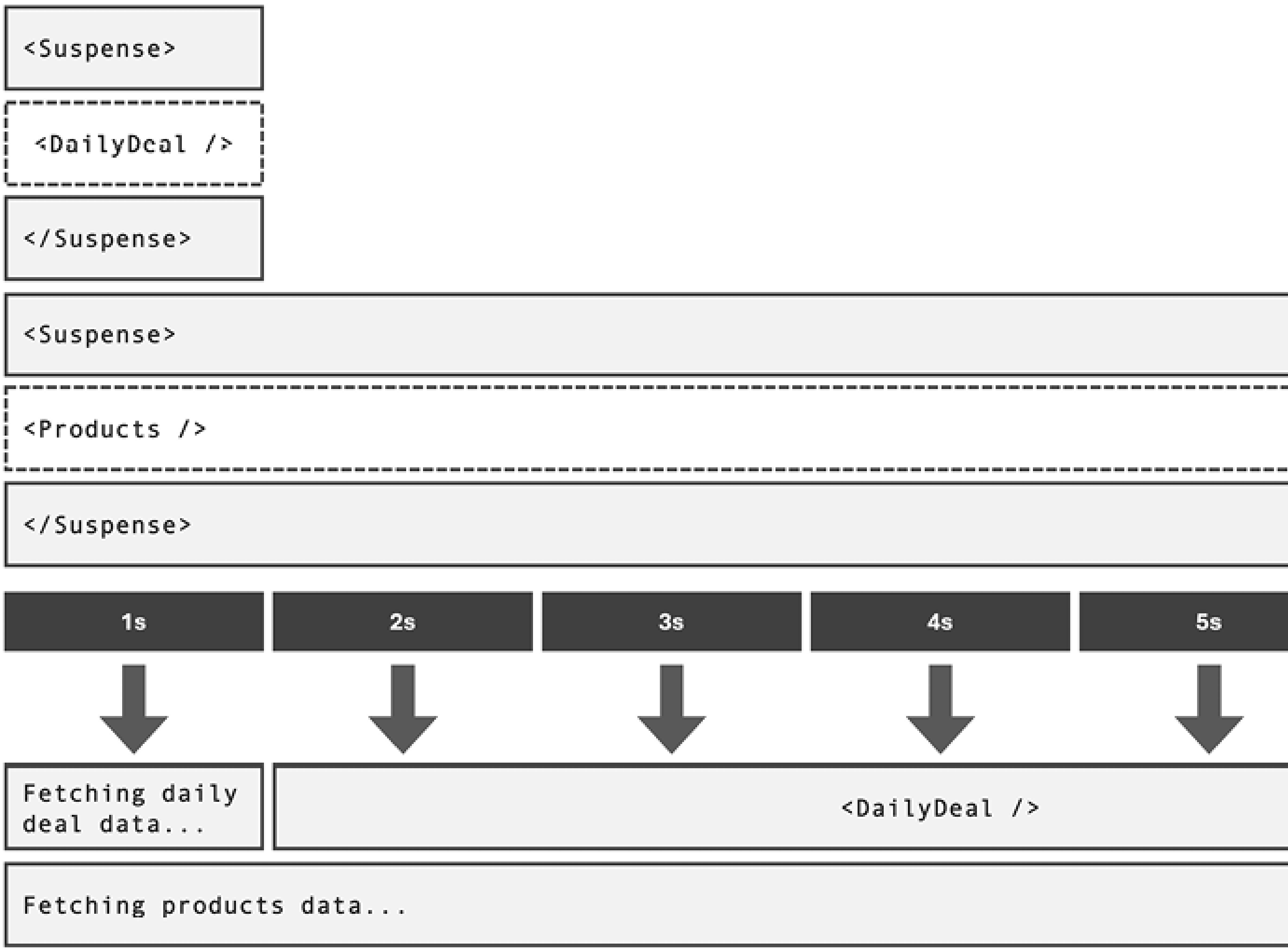


Figure 17.7: Each component replaces its fallback content with the final content once it's done fetching

Nesting Suspended Content

Besides fetching in parallel, you can also create more complex loading sequences with nested `Suspense` components.

Consider this example:

```
function Shop() {
  return (
     <>
    <h1>Welcome to our shop!</h1>
    <Suspense fallback={<p>Fetching shop data...</p>}>
      <DailyDeal />
      <Suspense fallback={<p>Fetching products data...</p>}>
        <Products />
      </Suspense>
    </Suspense>
  </>
  );
}
```
CopyExplain

In this case, initially, the paragraph with the text `Fetching shop data` is displayed. Behind the scenes, data fetching in the `DailyDeal` and `Products` components starts.

Once the `DailyDeal` component is done fetching data, its content is displayed. At the same time, below `DailyDeal`, the fallback of the nested `Suspense` block is rendered if the `Products` component is still fetching data.

Finally, once `Products` has received its data, the inner `Suspense` component's fallback content is removed, and the `Products` component is rendered instead.

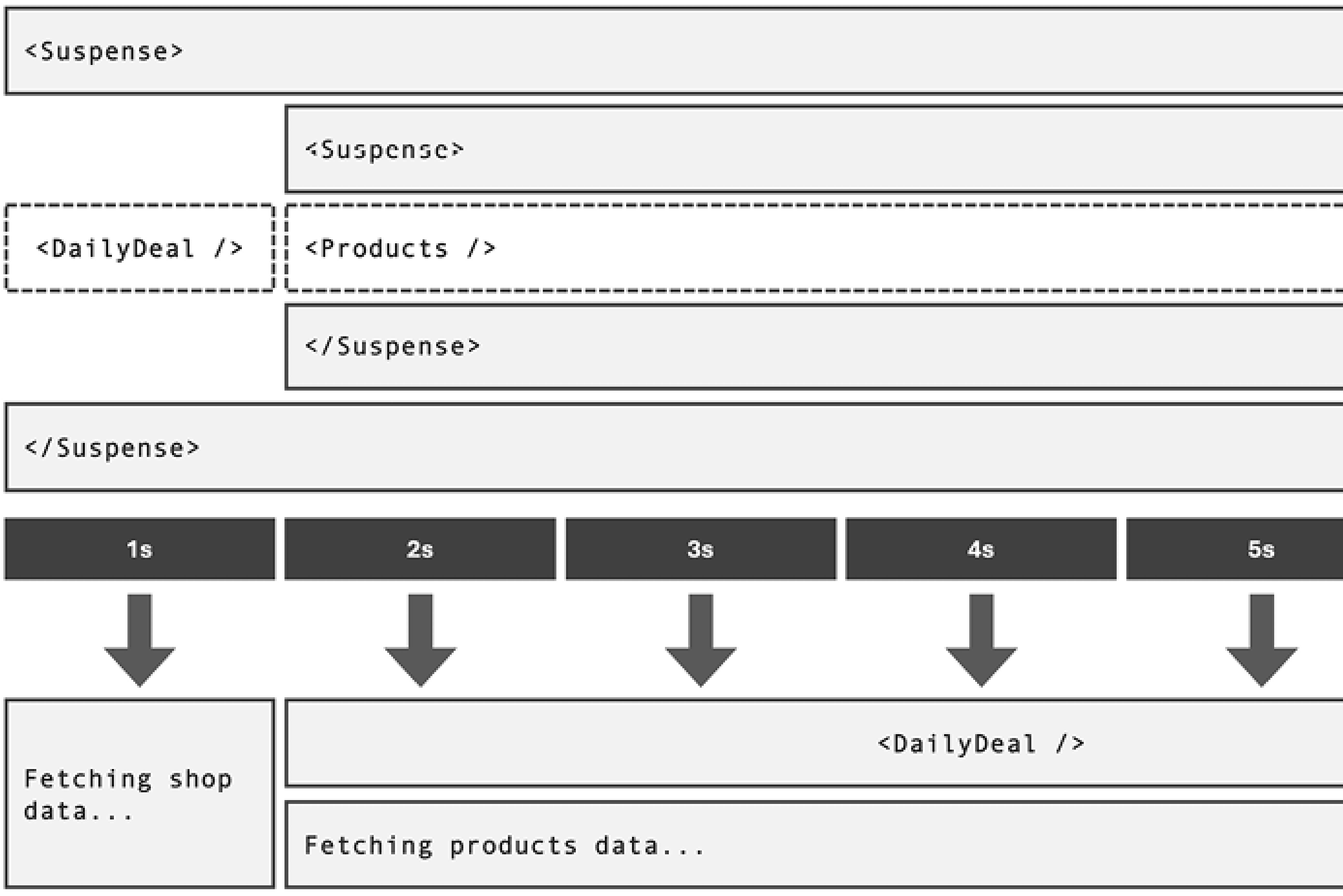


Figure 17.8: Nested Suspense blocks lead to sequential data fetching and content revelation

Therefore, as you can see, you can use `Suspense` multiple times. In addition, you can combine different `Suspense` components such that you can create exactly the loading sequence and user experience you need.

Should You Fetch Data via Suspense or useEffect()?

As you learned throughout this chapter, you can use `Suspense` in conjunction with RSCs, `Suspense`-enabled libraries, or the `use()` Hook (which also requires supporting libraries) to fetch data and show some fallback content while data is being fetched.

Alternatively, as covered in Chapter 11, Working with Complex State, you can also fetch data and manually show fallback content via `useEffect()` and `useState()` or `useReducer()`. In that case, you essentially manage the state that determines whether to show some loading fallback content on your own; with `Suspense`, React does that for you.

Consequently, it's up to you which approach you prefer. Using `Suspense` can save you quite a bit of code since you don't need to manage these different state slices manually. Combined with frameworks like Next.js or libraries like TanStack Query, data fetching can therefore become significantly easier than when doing it manually via `useEffect()`. In addition, `Suspense` integrates with RSCs and SSR and therefore can be used to fetch data on the server side—unlike `useEffect()`, which has no effect (no pun intended) on the server side.

However, if you're not using any library or framework that supports `Suspense` or `use()`-enabled promises, you don't have much of a choice other than to fall back to `useEffect()` (and hence not use `Suspense` for data fetching). This may change with future React versions, since they might provide tools that help with building promises that work with `use()`. But for the time being, it's basically a decision between using (the right) libraries and `Suspense` or no libraries and `useEffect()`.

Summary and Key Takeaways

- The `Suspense` component can be used to show fallback content while data is fetched, or code is downloaded.
- For data fetching, `Suspense` only works with components that fetch data via Suspense-enabled data sources during their rendering process.
- Libraries and frameworks like TanStack Query and Next.js support using `Suspense` for data fetching.
- Using Next.js, you can wrap `Suspense` around server components that use `async/await`.
- Alternatively, `Suspense` can be wrapped around components that use React's `use()` Hook for reading a promise value.
- `use()` should only be used to read values of promises that resolve with `Suspense` in mind—e.g., promises created by Suspense-compatible third-party libraries.
- When using Next.js, promises created in RSCs and passed to (client) components via props may be consumed via `use()`.
- The `use()` Hook helps with reading values and using `Suspense` in components that also need to use client-specific features like `useState()`.
- `Suspense` can be wrapped around as many components as needed to fetch data and display content simultaneously.
- `Suspense` can also be nested to create complex loading sequences.

What's Next?

React's `Suspense` feature can be very useful since it helps with granularly showing fallback content while code or data is being fetched. At the same time, when it comes to data fetching, it can be tricky to use `Suspense` since it only works with components that fetch data in the correct way (e.g., via the `use()` Hook, if the promise passed to the Hook is `Suspense`-compatible).

That's why this chapter also explored how to use `Suspense` and `use()` with Next.js, and how that framework simplifies the process of fetching data and showing fallback content with `Suspense` and `use()`.

Despite the potential complexity, `Suspense` can help with creating great user experiences since it allows you to easily show fallback content while a resource is pending.

This chapter also concludes the list of core React features you must know about as a React developer. Of course, you can always dive deeper to explore more patterns and third-party libraries. The next (and last) chapter will share some resources and possible next steps you could dive into after finishing this book.

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://github.com/TalipotTech/ReactConcepts/blob/17-suspense-use/exercises/questions-answers.md

:

1. What's the purpose of React's `Suspense` component?
2. How do components need to fetch data in order to work with `Suspense`?
3. How may `Suspense` be used when working with Next.js?
4. What's the purpose of the `use()` Hook?
5. Which kind of promises can be read by the `use()` Hook?
6. List three ways of using `Suspense` with multiple components.

Apply What You Learned

With all the newly gained knowledge about Next.js, it's time to apply it to a real demo project.

In the following section, you'll find an activity that allows you to practice working with Next.js and `Suspense`. As always, you will also need to employ some of the concepts covered in earlier chapters.

Activity 17.1: Implement Suspense in the Mini Blog

In this activity, your job is to build upon the finished project from Activity 16.1. There, a very simple blog was built. Now, your task is to enhance this blog to show some fallback content while the list of blog posts or the details for an individual blog post are loading. To prove your knowledge, you should fetch data via `async/await` on the starting page (`/`), and via the `use()` Hook on the `blog/<some-id>` page.

In addition, the list of available blog posts should also be displayed below the details for a single blog post. Of course, while fetching that list data, some fallback text must be displayed—though, that text should be displayed independently from the fallback content for the blog post details.

Note

You can find a starting project snapshot for this activity at

https://github.com/TalipotTech/ReactConcepts/tree/17-suspense-use/activities/practice-1-start

. 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-start`, in this case) to use the right code snapshot.

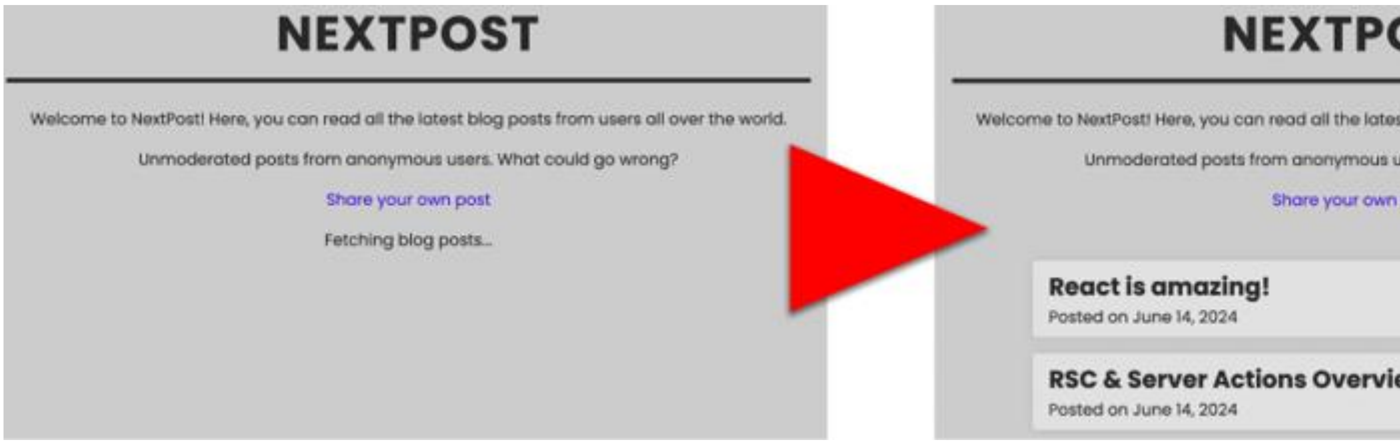
In the provided starting project, you'll find functions for fetching all blog posts and a single post. These functions contain artificial delays to simulate slow servers.

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

1. Outsource the logic for fetching and displaying a list of posts into a separate component.
2. Use that component on the starting page and use React's `Suspense` component to display some fitting fallback content while the blog posts are being fetched.
3. Also, outsource the logic for retrieving and rendering the details for a single blog post into a separate client (!) component. Output that newly created component on the `/blog/<some-id>` page.

4. Pass a promise for fetching the details of a blog to that newly created component, and use the `use()` Hook to read its value. Also, take advantage of the `Suspense` component to output some fallback content.
5. Re-use the component that fetches and renders a list of blog posts and output it below the blog post details on the `/blog/<some-id>` page. Use `Suspense` to show some fallback content, independently from the data fetching status of the blog post details.

The final page should look as shown in the following screenshots:



NEXTPOST

Welcome to NextPost! Here, you can read all the latest blog posts from users all over the world.

Unmoderated posts from anonymous users. What could go wrong?

Share your own post

Fetching blog posts...

NEXTPO

Welcome to NextPost! Here, you can read all the lates

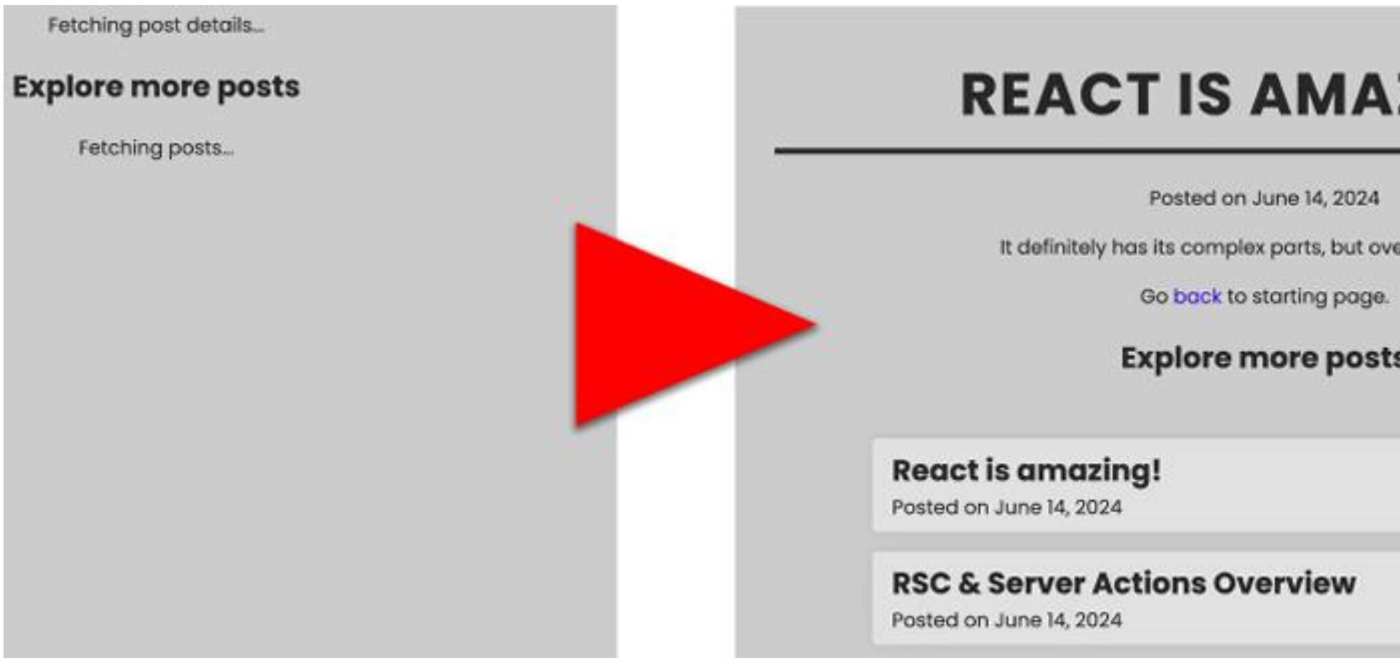
Unmoderated posts from anonymous u

Share your own

React is amazing!
Posted on June 14, 2024

RSC & Server Actions Overvi
Posted on June 14, 2024

Figure 17.9: Fallback content is shown while the blog posts are fetched



Fetching post details...

Explore more posts

Fetching posts...

REACT IS AMA

Posted on June 14, 2024

It definitely has its complex parts, but ove

Go back to starting page.

Explore more posts

React is amazing!
Posted on June 14, 2024

RSC & Server Actions Overview
Posted on June 14, 2024

Figure 17.10: Fallback content is shown while fetching blog post details and the list of blog posts

Note

You can find the full code for this activity, and an example solution, here: .

https://github.com/TalipotTech/ReactConcepts/tree/17-suspense-use/activities/practice-1