React Quickly

SECOND EDITION

MORTEN BARKLUND
AZAT MARDAN

MEAP

MANNING

MEAP Edition
Manning Early Access Program

React Quickly, Second Edition

Version 5

welcome

Thanks a bunch for purchasing *React Quickly, Second Edition*.

Oh boy, are we excited about bringing this book to you! React is one of the most sought-after web technologies, despite having quite a few miles on the odometer. React remains an extremely popular tool for writing web applications by everyone from the hotshots at Facebook, Airbnb, Discord, Uber, Bloomberg, all the way to fancy new startups and even hobbyists. Because React is still so popular, we have decided to write this updated version of *React Quickly*.

A lot has happened since the first edition of this book was published back in 2018. React has changed completely, especially as hooks came out with React 16.8. While hooks seemed like a minor technological improvement in itself, they caused a huge shift in everyday React usage. The community has completed turned around and reinvented how we write, organize, combine and generally work with components. We have tried to capture these new patterns in this edition.

React is currently in that weird stage of web technology, where it is neither new nor old. It is not the fancy new kid on the block anymore, so the initial hype has long worn off. But neither is it one of the old established bastions in the genre of web development tools, that has been around forever and comes with a long and storied past. This means that React still feels new while being a solid and thoroughly tested development platform.

React Quickly, Second Edition, will take you from a React beginner or novice to an expert and will enable you to contribute to a larger React codebase in no time – either on your own or as part of a team. The book comes in two parts: The first part introduces you to all the React basics, that you need to know in order to create reusable components. The second part introduces you to the React ecosystem and how you can create real-world applications using a variety of tools to ease development and integrate with other systems.

While we expect you to have a basic understanding of HTML, CSS, JavaScript, and web technologies in general, you do not need any prior experience with React or similar frameworks.

One great benefit of React's ecosystem is the well-crafted build tools and project generators. This means that you do not have to mess around with installers and downloads. All you need is a computer with a terminal, (reasonably new versions of) node and npm installed, and you should be good to go.

We hope that you will enjoy reading this book and going through the examples learning with us. If you enjoy the experience just half as much as we have enjoyed creating it, you will not be disappointed!

We definitely encourage you to provide feedback both in form of comments on particular elements in the book, but also general questions, about what you feel the book should have covered or does not fully explain. We will use all feedback in the liveBook Discussion forum to improve the experience for you and other readers.

—Morten Barklund and Azat Mardan

brief contents

1 Meeting React

This chapter covers

- **Understanding what React is**
- **Solving problems with React**
- **Fitting React into your web applications**
- **Writing your first React app: *Hello World***

React is the ground-breaking tool, that web developers did not even know they needed, but can't let go of once they have tried it. This is definitely true for us authors as well as for many other enthusiastic web developers out there. React is immensely popular and for a good reason.

If you were doing web development in the early 2000s, all you needed was some HTML and a server-side language like Perl or PHP. Ah, the good old days of putting in `alert()` boxes just to debug your front-end code. The internet has evolved a lot since then, and the complexity of building websites has increased dramatically. Websites have become web applications with complex user interfaces, business logic, and data layers that require changes and updates over time—and often in real-time.

Many JavaScript template libraries have been written to try to solve the problems of dealing with complex user interfaces (UIs). But they still require developers to adhere to the old separation of concerns—which splits style (CSS), data and structure (HTML), and dynamic interactions (JavaScript)—and they don't meet modern-day needs (remember the term *DHTML*?).

In contrast, React offers a new approach that streamlines frontend web development. React is a powerful UI library that offers an alternative that many big firms such as Facebook, Netflix, and Airbnb have adopted and see as the way forward. Instead of defining a one-off template for your UIs, React allows you to create reusable UI components in JavaScript that you can use again and again on your sites.

Do you need a captcha control or date picker? Then use React to define a `<Captcha />` or `<DatePicker />` component that you can add to your form: a simple drop-in component with all the functionality and logic to communicate with the back end. Do you need an autocomplete box that asynchronously queries a database once the user has typed four or more letters? Define an `<Autocomplete charNum="4"/>` component to make that asynchronous query. You can choose whether it has a text box UI or has no UI and instead uses another custom form element—perhaps `<Autocomplete textbox="..." />`.

This approach isn't new. Creating composable UIs has been around for a long time, but React is the first to use pure JavaScript without templates to make this possible. And this approach has proven easier to maintain, reuse, and extend.

React is a great library for UI's, and it should be part of your front-end web toolkit, but it isn't a complete solution for all front-end web development. In this chapter, we'll look at the pros and cons of using React in your applications and how you might fit it into your existing web development stack.

In this book, we will cover the basics of React and enable you to work as a fully-capable React developer either on your own or as part of a development team. We will mostly be working on creating small prototypes and examples, but will also make sure that you are equipped to work on a large project. We will achieve this by introducing popular libraries and tools used around React, that you will see used by many development teams, as well as architecture and design patterns that commonly arise in both small and large codebases.

Part 1 of the book focuses on React's primary concepts and features, and part 2 looks at working with libraries related to React to build more complex front-end apps (a.k.a. *React stack* or *React and friends*). Each part demonstrates both greenfield (brand new applications) and brownfield (updating legacy systems) development with React and the most popular libraries, so you can get an idea of how to approach working with it in real-world scenarios.

Source code for this chapter

The source code for the example in this chapter is available at https://github.com/rq2e/rq2e/tree/main/ch01.

1.1 Benefits of using React

Every new library or framework claims to be better than its predecessors in some respect. In the beginning, we had jQuery, and it was leaps and bounds better for writing cross-browser code in native JavaScript. If you remember, a single AJAX call taking many lines of code had to account for Internet Explorer and WebKit-like browsers. With jQuery, this takes only a single call: `$.ajax()`, for example. Back in the day, jQuery was called a framework—but not anymore! Now a framework is something bigger and more powerful.

Similarly, with Backbone and then Angular, each new generation of JavaScript frameworks has brought something new to the table. React isn't unique in this. What is new is that React challenges some of the core concepts used by most popular front-end frameworks: for example, the idea that you need to have templates.

The following list highlights some of the benefits of React versus other libraries and frameworks that existed at the time React emerged:

- *Simpler apps*—React has a CBA with pure JavaScript; a declarative style; and powerful, developer-friendly DOM abstractions (and not just DOM, but also iOS, Android, and so on).
- *Fast UIs*—React provides outstanding performance thanks to its virtual DOM and smart-reconciliation algorithm, which, as a side benefit, lets you perform testing without spinning up (starting) a headless browser.
- *Less code to write*—React's great community and vast ecosystem of components provide developers with a variety of libraries and components. This is important when you're considering what framework to use for development.

Many features made React simpler to work with than most other front-end frameworks available at its infancy. However, many new frameworks have spawned since React came around. Partially due to the popularity of React, some of these new frameworks have been developed with similar benefits or thoughts, each slightly altered in different ways. Some other frameworks might just be inspired by the overall idea, but work completely differently, whereas others are very similar to React, just with a smaller API requiring you to sometimes write more code, but other times end up with a much smaller application codebase.

For now, we'll consider the benefits that made React unique at its infancy - and still are the main selling points of React, although other modern frameworks have similar benefits today. Let's start to unpack these benefits one by one, starting with how wonderfully simple React is to use.

1.1.1 Simplicity

The concept of simplicity in computer science is highly valued by developers and users. It doesn't equate to ease of use. Something simple can be hard to implement, but in the end, it will be more elegant and efficient. And often, an easy thing will end up being complex. Simplicity is closely related to the KISS principle (keep it simple, stupid). The gist is that simpler systems work better.

React's approach allows for simpler solutions via a dramatically better web-development experience for software engineers. When we began working with React, it was a considerable shift in a positive direction that reminded us of switching from using plain, no-framework JavaScript to jQuery.

In React, this simplicity is achieved with the following features:

- *Declarative over imperative* style—React embraces declarative style over imperative by updating views automatically.
- *Component-based architecture using pure JavaScript*—React doesn't use domain-specific languages (DSLs) for its components, just pure JavaScript. And there's no separation when working on the same functionality.
- *Powerful abstractions*—React has a simplified way of interacting with the DOM, allowing you to normalize event handling and other interfaces that work similarly across browsers.

Let's cover these one by one.

DECLARATIVE OVER IMPERATIVE STYLE

First, React embraces declarative style over imperative. Declarative style means developers write how it *should* be, not what to do, step-by-step (imperative). But why is declarative style a better choice? The benefit is that declarative style reduces complexity and makes your code easier to read and understand.

The distinction between imperative and declarative coding styles can to some extent quickly become academic, and when taken to the extreme, declarative programming can become really complex to read unless you understand some pretty complex concepts well such as monads and functors.

Here are a few different ways to describe the difference between the two styles:

- **Statements vs expressions**: Imperative style programming often works with independent statements that individually advance the program state, while declarative programming uses expressions that build upon each other to progress the flow of logic.
- **Reserved word usage**: Imperative style programming often uses many reserved words such as `for`, `while`, `switch`, `if`, `else`, while declarative style programming uses array methods, arrow functions, object access, boolean expressions, and ternary operators to achieve the same results.
- **Function composition**: Imperative style programming often uses independent function calls and method invocations while declarative style programming uses function composition to build upon the previous expression and make small generalized pieces of logic that when composed achieve the desired result.
- **Mutability**: Imperative style programming often uses mutable objects and manipulates existing structures while declarative style programming uses immutable data and creates new structures from old ones rather than editing an existing one.

Let's create a simple example to illustrate these different points.

The goal of this task is to create a function, `countGoodPasswords`, that given a list of passwords will return how many of the passwords are *good*. Here we will define a *good* password as any password at least 9 characters long.

This is a great simple task that can be solved in any programming language in a multitude of ways. Some programming languages inherently make one style more natural to reach for, but JavaScript is a bit special, as it's a member of both worlds. You can solve this task either imperatively or declaratively.

Let's start with a (very) naïve imperative solution:

```
function countGoodPasswords(passwords) {
  const goodPasswords = [];    #A
  for (let i = 0; i < passwords.length; i++) {    #B
    const password = passwords[i];    #A
    if (password.length < 9) {    #B
      continue;    #B
    }
    goodPasswords.push(password);    #A, #C
  }
  return goodPasswords.length;
}
```

#A New statement changing the program state
#B Reserved word to control program flow
#C Mutating an existing object

This is of course partially taken to an extreme, and even under a fully imperative programming paradigm, this could be much shorter.

Let's implement this same example using a declarative programming mindset:

```
function countGoodPasswords(passwords) {
  return passwords.filter(p => p.length >= 9).length;
}
```

We arrive directly at the goal in a single statement by manipulating an object in several tempi using function composition to arrive at the target. We filter the original array to arrive at a temporary value, which is the array of only good passwords. However, we never store this array anywhere, we go directly to the next step of taking the length of that array.

That was just some generic JavaScript code. How does this relate to React? React takes the same declarative approach when you compose UIs. First, React developers describe UI elements in a declarative style. Then, when there are changes to views generated by those UI elements, React takes care of the updates. Yay!

The convenience of React's declarative style fully shines when you need to make changes to the view. Those are called changes of the *internal state*. When the state changes, React updates the view accordingly.

NOTE **We will cover how states work in chapter 5.**

COMPONENT-BASED ARCHITECTURE USING PURE JAVASCRIPT

Component-based architecture existed before React came on the scene. Separation of concerns, loose coupling, and code reuse are at the heart of this approach because it provides many benefits; software engineers, including web developers, love CBA. A building block of CBA in React is the component class. As with other CBAs, it has many benefits, with code reuse being the main one (you can write less code!).

What was lacking before React was a pure JavaScript implementation of this architecture. When you're working with Angular, Backbone, Ember, or most of the other MVC-like front-end frameworks, you have one file for JavaScript and another for the template. (Angular uses the term directives for components.) There are a few issues with having two languages (and two or more files) for a single component.

The HTML and JavaScript separation worked well when you had to render HTML on the server, and JavaScript was only used to make your text blink. Now, single-page applications (SPAs) handle complex user input and perform rendering on the browser. This means HTML and JavaScript are closely coupled functionally. For developers, it makes more sense if they don't need to separate between HTML and JavaScript when working on a piece of a project (component).

Under the hood, React uses a *virtual DOM* to find differences (the delta) between what's already in the browser and the new view. This process is called *DOM diffing* or *reconciliation of state and view* (bringing them back to similarity). This means developers don't need to worry about explicitly changing the view; all they need to do is update the state, and the view will be updated automatically as needed.

Conversely, with jQuery, you'd need to implement updates imperatively. By manipulating the DOM, developers can programmatically modify parts of the web page without re-rendering the entire page. DOM manipulation is what you do when you invoke jQuery methods.

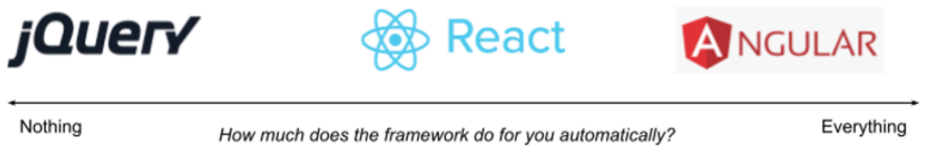


Figure 1.1: How much does the framework help you?

Think of the help provided by the underlying framework on a scale as in figure 1.1. At one end of the scale, you have a "framework" that doesn't actually help you at all. If you build your application in plain JavaScript, you would be at this extreme. Using jQuery would make it easier to manipulate the DOM, but you would still have no help from the framework when things update. You would have to manually make sure that your jQuery views update when your jQuery data updates.

At the other end of the scale we have frameworks such as Angular. It is another very popular framework, comparable to React in every way. It however works in a fundamentally different way with a lot more "magic" happening behind the scenes. You often merely described how your components fit together and Angular will try to connect things correctly behind the scenes. The problem with Angular is often that you don't have the fine-grained control you might want, if things don't work correctly. Many things are hidden for you, that make things unnecessarily complex.

React strikes that happy medium, where the framework helps you with a lot of the tedious work of connecting various things behind the scenes, but without locking you out of the fine-grained control required to make complex web applications.

Powerful abstractions

React comes with some great abstractions that make life as a React developer easier. These include:

- Synthetic events abstracting out browser differences in native events
- JSX abstracting out the JS document object model (DOM)
- Browser-independence allowing rendering in non-browser environments (e.g. on the server)

React has a powerful abstraction of the browser event model. In other words, it hides the underlying interfaces and provides normalized/synthesized methods and properties. For example, when you create an `onClick` event in React, the event handler will receive not a native browser-specific event object, but a synthetic event object that's a wrapper around native event objects. You can expect the same behavior from synthetic events regardless of the browser in which you run the code. React also has a set of synthetic events for touch events, which are great for building web apps for mobile devices.

Then there's JSX, which is one of the more controversial elements of React. For some, the abstraction of JSX is a strong argument *for* using React, while for others JSX has been a stumbling block or even a deterrent.

If you're familiar with Angular, then you've already had to write a lot of JavaScript in your template code. This is because, in modern web development, plain HTML is too static and is hardly any use by itself. Our advice: give React the benefit of the doubt, and give JSX a fair run.

JSX is a bit of *syntactical sugar* on top of JavaScript for writing React elements in JavaScript using HTML-like notation with <>. React pairs nicely with JSX because developers can better implement and read the code. Think of JSX as a mini-language that's compiled into native JavaScript. So, JSX isn't run on the browser but is used as the source code for compilation. Here's a compact snippet written in JSX:

```
if (user.session) {
  return <a href="/logout">Logout</a>;
} else {
  return <a href="/login">Login</a>;
}
```

Even if you load a JSX file in your browser with the runtime transformer library that compiles JSX into native JavaScript on the run, you still don't run the JSX; you run JavaScript instead. In this sense, JSX is akin to CoffeeScript. You compile these languages into native JavaScript to get better syntax and features than that provided by regular JavaScript.

We know that to some of you, it looks bizarre to have HTML interspersed within JavaScript code. It takes every new React developer (even us) a while to adjust because you are expecting an avalanche of syntax error messages. And yes, using JSX is optional. For these two reasons, we're not covering JSX until chapter 3; but trust us: it's very powerful and even addictive once you get familiar with it.

Another example of React's DOM abstraction is that you can render React elements on the server. This can be handy for better search engine optimization (SEO) and/or improving performance.

There are many options when it comes to rendering React components in both DOM and HTML strings on the server. You can even use hybrid approaches where your templates are rendered with some content on the server and later re-hydrated with live data in the browser. We'll talk a lot more about this in section 1.3. And, speaking of the DOM, one of the most sought-after benefits of React is its splendid performance.

1.1.2 Speed and testability

In addition to the necessary DOM updates, your framework may perform unnecessary updates, which makes the performance of complex UIs even worse. This becomes especially noticeable and painful for users when you have a lot of dynamic UI elements on your web page.

On the other hand, React's virtual DOM exists only in the JavaScript memory. Every time there's a data change, React first compares the differences using its virtual DOM; only when the library knows there has been a change in the rendering will it update the actual DOM. Figure 1.2 shows a high-level overview of how React's virtual DOM works when there are data changes.

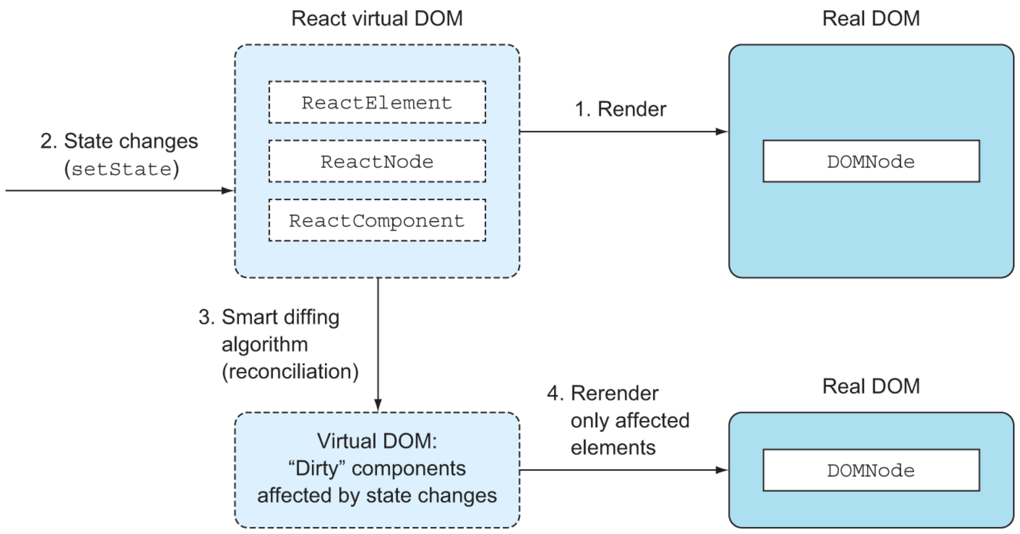

React virtual DOM
Real DOM
ReactElement
ReactNode
ReactComponent
1. Render
DOMNode
2. State changes (setState)
3. Smart diffing algorithm (reconciliation)
4. Rerender only affected elements
Real DOM
Virtual DOM: "Dirty" components affected by state changes
DOMNode

Figure 1.2 Once a component has been rendered, if its state changes, it's compared to the in-memory virtual DOM and re-rendered if necessary.

Ultimately, React updates only those parts that are absolutely necessary so that the internal state (virtual DOM) and the view (real DOM) are the same. For example, if there's a `<p>` element and you augment the text via the state of the component, only the text will be updated (that is, `innerHTML`), not the element itself. This results in increased performance compared to re-rendering entire sets of elements or, even more so, entire pages (server-side rendering).

Note

If you like to geek out on algorithms and Big O notation, these two articles do a great job of explaining how the React team managed to turn an O(n^3) problem into an O(n) one:

- **"Reconciliation," on the React website (http://mng.bz/PQ9X)**
- **"React's Diff Algorithm" by Christopher Chedeau (http://mng.bz/68L4).**

The added benefit of the virtual DOM is that you can do unit testing without headless browsers like PhantomJS (http://phantomjs.org). There are several libraries out there, including Jest and React-Testing-Library, that allow you to test your components directly from the command line.

We'll obsess quite a bit more on unit testing React components and hooks in later chapters.

1.1.3 Ecosystem and community

Last, but not least, React is supported by the developers of a juggernaut web application called Facebook, as well as by their peers at Instagram. As with Angular and some other libraries, having a big company behind the technology provides a sound testing ground (it's deployed to millions of browsers), reassurance about the future, and an increase in contribution velocity.

There is a lot of great, existing content out there already created for React by the community. Most of the time when you need some kind of component or interface, you can just search the web for "`react [name-of-component]`" and in more than 95% of the instances, you will find something worthwhile.

The history of open-source software clearly shows that the marketing of open-source projects is as important to its wide adoption and success as the code itself. By that, we mean that if a project has a poor website, lacks documentation and examples, or has an ugly logo, most developers won't take it seriously—especially now, when there are so many JavaScript libraries. Developers are picky, and they won't use an ugly duckling library.

As the saying goes, *"don't judge a book by its cover."* This might sound controversial; but, sadly, most people, including software engineers, are prone to biases such as good branding. Luckily, React has a great engineering reputation backing it. And, speaking of book covers, we hope you didn't buy this book just for its cover!

1.2 Disadvantages of React

Of course, almost everything has its drawbacks. This is true with React, but the full list of cons depends on whom you ask. Some of the differences, like declarative versus imperative, are highly subjective. They can be both pros and cons depending on your personal preference.

Here's our list of React's disadvantages. As with any such list, it may be biased:

- React isn't a full-blown, Swiss Army knife–type of framework. Developers need to pair it with a library like Redux or React Router to achieve functionality comparable to Angular or Ember. This can also be an advantage if you need a minimalistic UI library to integrate with your existing stack.
- React uses a somewhat new approach to web development, and JSX and functional programming can be intimidating to beginners. Especially in the early days, there was a lack of best practices, good books, courses, and resources available for mastering React and similar frameworks.
- React only has a one-way binding. Although one-way binding is better for complex apps and removes a lot of complexity, some developers (especially Angular developers) who got used to a two-way binding will find themselves writing a bit more code. We'll explain how React's one-way binding works compared to Angular's two-way binding in chapter 8, which covers working with form data.
- React isn't reactive (as in reactive programming and architecture, which are more event-driven, resilient, and responsive) out of the box. Developers need to use other libraries to make your application integrate with external content seamlessly and responsively.

To continue with this introduction to React, let's look at how it fits into a web application.

1.3 How React can fit into your website

Websites come in many variants, and React can be used to create interactive content in many different types of websites, either as a replacement for other technologies or as a way to add new functionality to your website. React can be used on both "classic" websites that are mostly rendered by a server, as well as client-side web applications, also known as *single-page applications*.

The React core library is a user interface library first and foremost. The core library alone is comparable to other user interface libraries, but not directly comparable to more full-fledged web application frameworks such as Angular. However, combined with other libraries, either developed by the React team or other parties (e.g. React Router and Redux), React can be a full competitor to any web application framework.

If you are using another single-page application framework (such as Angular, Vue, Ember, Backbone, etc) to render your web application today, you will probably need to replace the entire thing with a React-based stack. It is very difficult bordering on impossible to create a hybrid single-page application with some parts rendered by e.g. Angular and others by React.

You can use React for just part of your UI, if you have a website with smaller interactive UI elements (or *widgets*). In such a case you can replace your widgets one by one with small React applications, without changing everything else. These existing widgets might be written in plain JavaScript, jQuery or even Angular or similar frameworks. As you go along converting widgets to React, you can evaluate what is the best fit for your organization.

React is back-end agnostic for front-end development. In other words, you don't have to rely on a JavaScript-based backend (Node or Deno) to use React. It's fine to use React with any other back-end technology like Java, Ruby, Go, or Python. React is a UI library, after all. You can integrate it with any back end and any front-end data library (Backbone, Angular, Meteor, and so on).

Another popular use case for React is static site generators. In such a setup, React is used to define your website locally on your environment, but when deployed to the live server, it is rendered "down" to a plain HTML website with JavaScript only doing a minimal bit of work to add interactivity. All your templates etc will have been resolved. This was in the beginning mostly popular for smaller websites such as blogs, that don't update too frequently.

Recent advances in server side React rendering has made this pre-rendered approach more and more popular even for larger single-page applications updating often. You can do this with popular frameworks built on top of React such as Next.js or Remix.run. These are considered partially server-rendered web applications, where your React code runs on both the server and in the client. This can sound a bit daunting, but newer frameworks make it relatively easy to work with.

To summarize how React fits into a website, it's most often used in these scenarios:

- As a UI library in a single-page application, such as React+React Router+Redux
- As a drop-in widget in any front-end stack, such as a React autocomplete input component in a website built using any other combination of technologies
- As a static website rendered on deployment to serve infrequently updated content
- As a partially server-side rendered website or single-page application built on top of a more powerful framework
- As a UI library in mobile apps, such as a React Native iOS app

React works nicely with some front-end technologies, but it's mostly used as a part of single-page applications. We cover how React fits into a single-page application in the next section.

1.3.1 Single-page applications and React

Single-page applications (SPAs) are a subset of websites in general. A website is considered a single-page application if it has a lot of functionality directly available in the browser and not just information. It includes things like Facebook, Google Docs, Gmail, etc.

Single-page applications are built using a multitude of technologies, of which React is only one potential part in the stack. You can't even use React alone, it needs to go together with at least a few other technologies in order to be usable as a stand-alone application.

In this section, we will go over what a single-page application is in general and only once this has been established does it make sense to point out how React fits into this structure.

Single-page applications are also known as a *thick client*, because the browser, being a client, holds more logic and performs functions such as rendering of the HTML, validation, UI changes, and so on. Contrast this with a thin client, where the browser client is only used to display information that has been pre-rendered by a server. In a thin client, that browser does very little work.

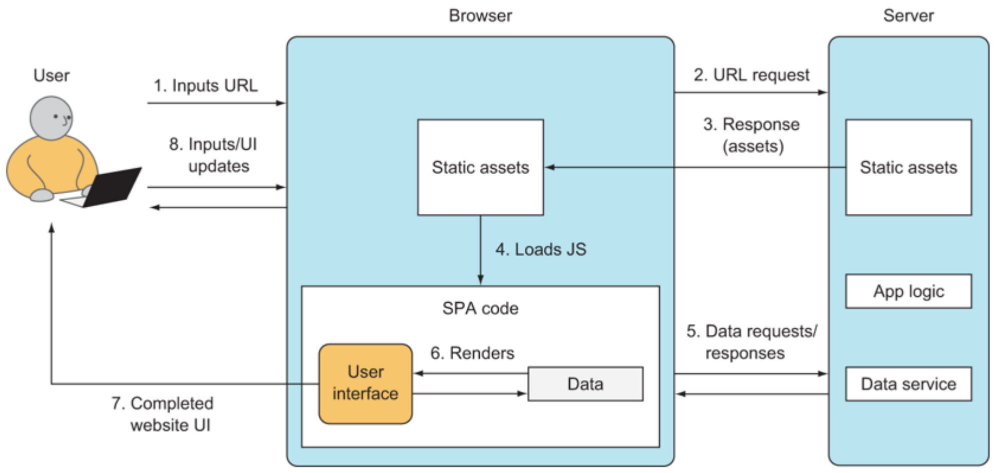


Figure 1.3 A typical single-page application architecture

Figure 1.3 is a very high-level example of a generic single-page application regardless of the technology used. It shows a bird's-eye view of a typical architecture with a user, a browser, and a server. The figure depicts a user making a request, and input actions like clicking a button, drag-and-drop, mouse hovering, and so on:

1. The user types a URL in the browser to open a new page.
2. The browser sends a URL request to the server.
3. The server responds with static assets such as HTML, CSS, and JavaScript. In most cases, the HTML is bare-bones—that is, it has only a skeleton of the web page. Usually there's a "Loading ..." message and/or rotating spinner GIF.
4. The static assets include the JavaScript code for the application. When loaded, this code makes additional requests for data (AJAX/XHR requests).
5. The data comes back in JSON, XML, or any other format.
6. Once the application receives the data, it can render missing HTML (the User Interface block in the figure). In other words, UI rendering happens on the browser by means of the application hydrating templates with data.

7. Once the browser rendering is finished, the browser updates the displayed content, and the user can work with the application.
8. The user sees a beautiful web page. The user may interact with the page (Inputs in the figure), triggering new requests from the application to the server, and the cycle of steps 2–6 continues. At this stage, browser routing may happen if the application implements it, meaning navigation to a new URL will trigger not a new page reload from the server, but rather an application re-render in the browser.

To summarize, in a single-page application most rendering for UIs happens in the browser. Only data travels to and from the browser. Contrast that with a "classic" website which is not a single-application application, where all the rendering happens on the server.

React fits into this single-page application architecture in steps 6 and 8 - rendering content based on data as well as handling user input and updating the content based on the updated data that results from these inputs.

1.3.2 The React stack

React isn't a full-blown, front-end JavaScript single-page application framework. React is minimalistic. It doesn't enforce a particular way of doing things like data modeling, styling, or routing (it's non-opinionated). Because of that, developers often need to pair React with a routing and/or data library.

While you can use React as a smaller part of your stack, developers most often opt to use a React-centric stack, which consists of React core itself as well as data, routing, and styling libraries created to be used specifically with React, such as:

- *Data-model libraries and back ends*—For example react-query (https://react-query.tanstack.com/), Redux (http://redux.js.org), Recoil.js (https://recoiljs.org/), or Apollo (https://www.apollographql.com/)
- *Routing library*—Often React Router (https://github.com/reactjs/react-router) or a similar router implemented in many frameworks
- *Styling libraries*—this can be either a predefined set of styled components such as Material UI (https://mui.com/) or Bootstrap (https://react-bootstrap.github.io/) or a library to easily work with CSS inside React components such as Styled-Components (https://styled-components.com/), Vanilla Extra (https://vanilla-extract.style/), or even Tailwind CSS (https://tailwindcss.com/)

The ecosystem of libraries for React is growing every day. Also, React's ability to describe composable components (self-contained chunks of the UI) helps reuse code. There are many components packaged as npm modules.

Here's a great (curated) list of a lot of various React components for many different purposes: https://github.com/brillout/awesome-react-components. This list has everything from UI components (including tons of form elements) over complete UI frameworks to development utilities and testing tools.

1.3.3 React website frameworks

Another category of React frameworks is the full-blown server side framework, which takes care of everything for you. Such frameworks come in two variants, but sometimes a framework can work in either way:

- Static site generators
- Dynamic server-rendered React

Static site generators are just that - a framework that will generate a completely static website for you fully ready to deploy to any static website host, which requires very little work on your part and no expensive hosting. This is particularly popular for smaller personal websites such as blogs, but can also be used for smaller businesses and even e-commerce websites (that don't require updates too often).

Dynamic server-rendered React frameworks are more complex and will take care of pre-rendering your React application on the server before serving the HTML over the wire to your visitors' browser. This means it's good for SEO, shareability and has many other benefits.

We will list three such frameworks here. We will however also revisit some of them in later projects in this book, as they really are great and *very* popular frameworks:

- **Gatsby**: a very popular blogging framework but also useful for many other types of static websites.
- **Next.js**: probably the most popular React website framework out there - useful for both small static websites and huge dynamic behemoths.
- **Remix.run**: a fairly new kid on the block, that is gaining traction and popularity very quickly in serving super-fast dynamic React websites.

All of these frameworks–and many, *many* more–are different extensions of React each functioning by their own paradigms. They all add extra functionality on top of React and sometimes also come with a set of React components, that helps you create your website in order to utilize the framework to its fullest.

By now, you have an understanding of what React is, its stack, its place in the higher-level web applications, and how you can use tools built on top of React to generate complex websites. It's time to get your hands dirty and write your first React application.

1.4 Your first React app: Hello World

Let's explore your first React application by implementing a *Hello World* application—the quintessential example used for learning programming languages. If we don't do this, the gods of programming might punish us.

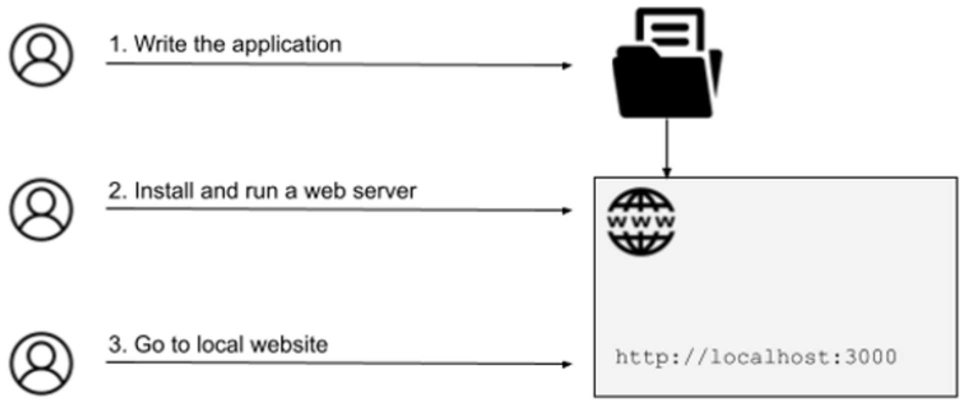


Figure 1.4 The process to create your first React application has just 3 simple steps.

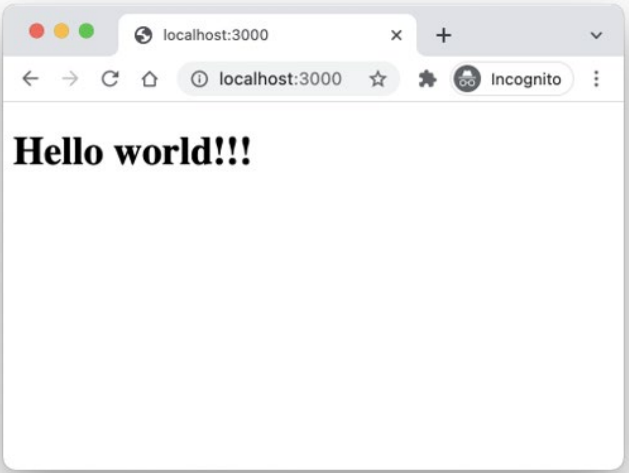
You will need a few things before you can get going. You don't need a lot, fortunately, because we're developing something that runs in the browser, so no need for all sorts of compilers or libraries. Here's a list anyway of things that you need before you can get started:

1. A text editor
2. Knowledge of how to use the terminal on your system
3. Have `npm` version 5.2 or newer installed — given that version 5.2 has been around since July 2017, odds are strong that your `npm` version is good enough if you have one
4. Have a modern browser installed — any recent version of Edge, Firefox, Chrome, or Safari will work

And that's about it. If you have the above, you are good to go for this first example. When we get to future examples in future chapters, you will actually not need a lot more than the above. A few of the later chapters will explore creating React applications in other frameworks, so they will have some more setup, but the above list is sufficient for almost the entire book!

1.4.1 The result

The project will print a *"Hello world!!!"* heading (`<h1>`) on a web page. Figure 1.5 shows what it will look like when you're finished (unless you're not quite that enthusiastic and prefer just a single exclamation point).



Hello world!!!

Figure 1.5 Hello World

You won't be using JSX yet, just plain JavaScript. You will also not be using JSX in chapter 2 and only start using it from chapter 3 and onward.

Learning React first without JSX

Although all React developers write React using JSX, browsers will only run standard JavaScript and not understand JSX directly. That's why it's beneficial to be able to understand React code in pure JavaScript. Another reason we're starting with plain JS is to show that JSX is optional, albeit the de facto standard template language for React. Finally, preprocessing JSX requires a bit more tooling, but it will actually make the whole setup simpler because you will see less of how the sausage is made, and do more of the fun stuff—writing awesome React components.

We want to get you started with React as soon as possible without spending too much time on setups in this chapter. You'll be introduced to how to start a new application in chapter 2 and we'll add JSX to the mix already in chapter 3.

1.4.2 Write the application

This project is so simple, it'll only consist of a single HTML file. This file will include links to the most recent versions of React 17 (the most stable version at the time of writing) of the React Core and React-DOM libraries. And then it will of course include a tiny bit of JavaScript code required to render the very simple application that we're building.

The code for the HTML file is simple and starts with the inclusion of the libraries in `<head>`. In the `<body>` element, you will create a <div> container with the ID `root` and a `<script>` element (that's where the app's code will go later), as shown in the following listing.

Listing 1.1 Loading React libraries and code

```
<!DOCTYPE html>
<html>
  <head>
    <script src="//unpkg.com/react@17/umd/react.development.js"></script>    #A
    <script src="//unpkg.com/react-dom@17/umd/react-dom.development.js"></script>    #B
  </head>
  <body>
    <div id="root"></div>    #C
    <script type="text/javascript">    #D
      ...    #E
    </script>
  </body>
</html>
```

#A Imports the React library
#B Imports the ReactDOM library
#C Defines an empty <div> element to mount the React UI
#D Creates a script node that will hold our JavaScript
#E The actual JavaScript code will go in here

Just type this code using your text editor and save it as a file named `index.html` in some folder on your machine.

Why do we have to create a `<div>` node to render the content in, you might ask? Why not render the React element directly in the `<body>` element? Because doing so can lead to conflict with other libraries and browser extensions that manipulate the document body. If you try attaching an element directly to the body, you'll get this console error:

Warning: Rendering components directly into document.body is discouraged, ...

This is another good thing about React: it has great warnings and error messages!

> **NOTE** **React warning and error messages aren't part of the production build, to reduce noise, increase security, and minimize the distribution size. The production build is the minified file from the React Core library: for example,** `react.min.js`**. The development version with the warnings and error messages is the unminified version: for example,** `react.development.js`.

By including the libraries in the HTML file, you get access to the React and ReactDOM global objects: `window.React` and `window.ReactDOM`. You'll need two methods from those objects: one to create an element (`React`) and another to render it in the `<div>` container (`ReactDOM`), as shown in listing 1.2.

To create a React element, all you need to do is call `React.createElement(elementName, data, children)` with three arguments that have the following meanings:

- `elementName`—HTML tag as a string (for example, `'h1'`) or a custom component class as an object. We don't have any custom components just yet but will start creating those in chapter 2.
- `data`—A data object containing attributes and properties for the element. We don't need any properties now, so we just pass `null`. We will get back to using properties in chapter 2.
- `children`—Child elements or inner HTML/text content. In this example, it's just `"Hello world!"`

Listing 1.2 Creating and rendering an h1 element

```
const reactElement = React.createElement('h1', null, 'Hello world!!!');    #A
const domNode = document.getElementById('root');    #B
ReactDOM.render(reactElement, domNode);    #C
```

#A Creates an h1 React element with the text "Hello world!!!"
#B Grabs a reference to the DOM element on the page with ID "root"
#C Renders the h1 React element into the DOM element

The code in Listing 1.2 goes into the `<script>` tag in the HTML file, that you created before in place of the `...`, that we originally put there as a placeholder.

This listing gets a React element and stores the reference to this object into the `reactElement` variable. The `reactElement` variable isn't an actual DOM node; rather, it's an instantiation of the React `h1` component (element). You can name it any way you want: `helloWorldHeading`, for example. In other words, React provides an abstraction over the DOM.

Once the element is created and stored in the variable, you render it to the DOM node/element with ID `root` using the `ReactDOM.render()` method shown in listing 1.2.

If you prefer, you can move both variables directly into the `render` call. The result is the same, except you don't use the two extra variables:

Listing 1.3 Single statement

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('root')
);
```

Using the first snippet, the full HTML file should now look like this:

Listing 1.4 Creating and rendering an h1 element

```
<!DOCTYPE html>
<html>
  <head>
    <script src="//unpkg.com/react@17/umd/react.development.js"></script>
    <script src="//unpkg.com/react-dom@17/umd/react-dom.development.js"></script>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/javascript">
      const reactElement = React.createElement('h1', null, 'Hello world!!!');    #A
      const domNode = document.getElementById('root');    #A
      ReactDOM.render(reactElement, domNode);    #A
    </script>
  </body>
</html>
```

#A The inserted JavaScript located in its proper place.

With the HTML file completed, we now need to see this in action by serving the content to our browser.

1.4.3 Install and run a web server

Now comes the next step, serving the HTML page to a browser. Why do we need to serve the content–can't we just open the HTML file directly in the browser? Due to cross-origin restrictions, you cannot open a file located on your local hard drive in the browser and have it access content on other domains (such as the React libraries loaded from unpkg.com). Browsers simply don't allow this. You can try to open the file in your browser directly by double-clicking it, but it will just show an empty white page. So that's no good.

Instead, we need to serve the content using a local development web server. That might sound terribly complex, but it's surprisingly simple to do in this day and age.

If you have node set up as recommended in the introduction, this will be enough to get you going. Just type the following command in the folder where you saved your `index.html` file:

```
$ npx serve
```

That's it. You might be asked to install a package (if you haven't used this command before, simply press enter to confirm if so), but after a few seconds, once the tool reports that everything is rolling, your web server is running.

Local development web server

In this very first example you, unfortunately, have to worry about setting up your own local web server. Luckily that is very simple, but it's a bit annoying to do here.

If for some reason the given command doesn't work for you, there's a couple of other ways to easily serve the current folder as a local web server.

If you have node, but `serve` for some reason doesn't work for you, you can instead try this command:

```
$ npx http-server
```

Alternatively, if you have a working Python 2 installation on your computer, you can just do:

```
$ python -m SimpleHTTPServer 3000
```

Or if you have a working Python 3 installation:

```
$ python -m http.server 3000
```

And finally, if you have a PHP setup working locally, you can do this:

```
$ php -S localhost:3000
```

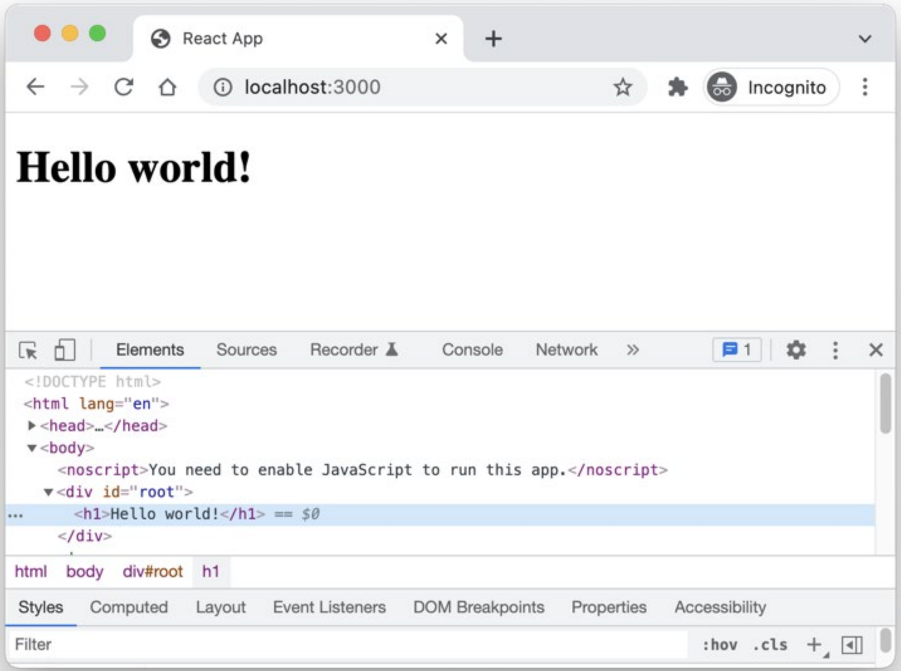
Any of those commands will run a local web server on your computer in the folder where you run the command serving your HTML file to `http://localhost:3000.`

1.4.4 Go to local website

With the web server running, you can now use your browser and go to this site:

```
http://localhost:3000
```

Here you should be able to see your application in action and it should look pretty much like figure 1.5 at the start of this chapter.



```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
  ▼<div id="root">
      <h1>Hello world!</h1> == $0
    </div>
```

Figure 1.6 Inspecting the Hello World app as rendered by React

Figure 1.6 shows the Elements tab in Chrome DevTools with the `<h1>` element selected. And you know that React must have done something here, because, in your source HTML file, there's no `<h1>` element inside the root node - it was completely empty.

One quick note: you can abstract the JavaScript code into a separate file instead of including the script directly in the HTML file (Listing 1.1). For example, you can create a file named `script.js` and copy and paste the entire snippet from either Listing 1.2 or Listing 1.3 into that file. Then, in the HTML file, you need to link to your script.js after the `<div>` with ID `root` rather than directly include a script tag, like this:

```
<div id="root"></div>
<script src="script.js"></script>
```

Congratulations! You've just implemented your first React application!

From the next chapter going forward, we will not be creating our React applications like this. We will be using a small tool to quickly generate and set up our React application basics for us which will make this entire process much more smooth. It will take care of serving our content as well, so you don't have to worry about web servers anymore.

1.5 Quiz

To be formatted with the 5 answers distributed over a 2-page spread.

1. React is a complete framework in and of itself and you can create many applications using nothing but React. *True* or *false*?
2. What is the primary problem that React solves:

   a) Fetching data from the server
   b) Creating beautiful HTML widgets
   c) Rendering dynamic data in a UI layer

3. React components are rendered into the DOM with which of the following methods? (Beware, it's a tricky question!):

   a) `ReactDOM.renderComponent`
   b) `React.render`
   c) `ReactDOM.append`
   d) `ReactDOM.render`

4. You have to use Node.js on the server to be able to use React in your SPA. *True* or *false*?
5. You must include `react-dom.js` to render React elements on a web page. *True* or *false*?

1.6 Summary

- React is declarative; it's only a view or UI layer.
- React uses components that you bring into existence with `ReactDOM.render()`.
- You use pure JavaScript to develop and compose UIs in React.
- You don't need to use JSX (an HTML-like syntax for React objects); JSX is optional when developing with React.
- React can fit into your web stack in many different ways from just a small widget on some page to the foundation of your entire website.
- To summarize the definition of React: React for the web consists of the React Core and ReactDOM libraries. React Core is a library geared toward building and sharing composable UI components using JavaScript and (optionally) JSX in a universal manner. On the other hand, to work with React in the browser, you can use the ReactDOM library, which has methods for DOM rendering as well as for server-side rendering.

1.7 Quiz answers

To be formatted with the 5 answers distributed over a 2-page spread.

1. *False*. You almost always have to use other frameworks or libraries to create the vast majority of React applications.
2. While you can create beautiful HTML widgets in React, the primary problem that React solves is to *render dynamic data in a UI layer* (answer c).

3. `ReactDOM.render`.
4. *False*. You can use any back-end technology.
5. *True*. You need the ReactDOM library.

2 Baby steps with React

This chapter covers

- **Creating a new React project**
- **Nesting elements**
- **Creating a component class**
- **Working with properties**

This chapter will teach you how to create a new React project and how to create custom components to render HTML. Both of these concepts will serve as the basis for all the future chapters.

First off, we're going to examine how to create a new React project. While doing so, we will teach you both how to start your own React projects, but also how we can utilize the React template system to very quickly instantiate the examples and projects that we're going to work on in this book. It's quite magical how you in a single line can get the code downloaded and ready to go with everything set up for you!

As we go through the process of starting our first React project, we'll take the opportunity to introduce several fundamental React concepts that you'll use frequently. These React basics include elements, components, and properties. In a nutshell, elements are instances of components that can be passed properties. What are their use cases, and why do you use them? Hang tight for this information until section 2.3, because right now we're going to discuss how to create a new React application.

> **NOTE** The source code for the examples in this chapter is available at https://github.com/rq2e/rq2e/tree/main/ch02. In section 2.2 you will however learn that you don't have to download anything yourself manually. You can instantiate all the examples from this chapter and onward directly from the command line using a single command.

2.1 Create a new React app

In this section, we'll introduce you to a magical command-line program that will make all your React setups go smoothly.

In just three short commands and a couple of minutes, you will download a fully functioning dummy React application, compile it, run it through a webserver and see it in your own browser. See figure 2.1 for an overview of what is going to happen.

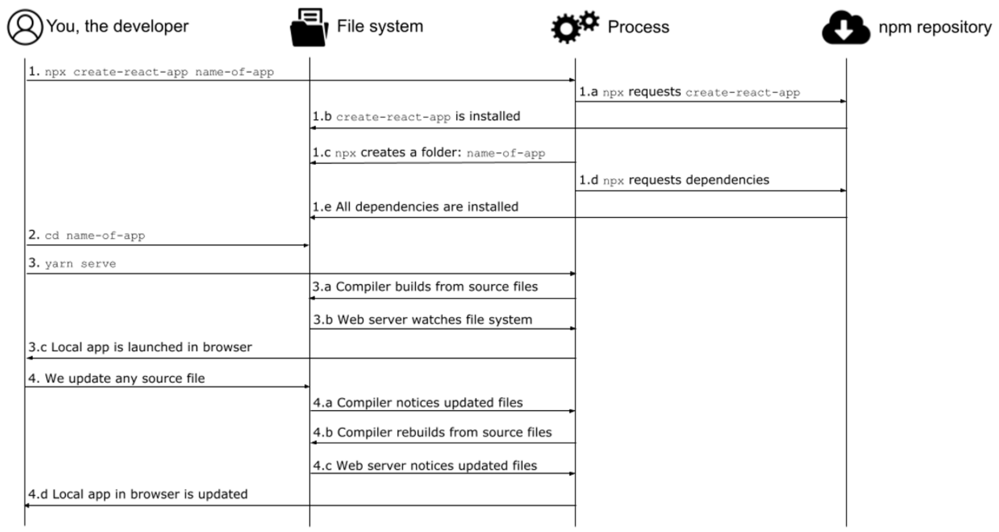

You, the developer
File system
Process
npm repository
1. npx create-react-app name-of-app
1.a npx requests create-react-app
1.b create-react-app is installed
1.c npx creates a folder: name-of-app
1.d npx requests dependencies
1.e All dependencies are installed
2. cd name-of-app
3. yarn serve
3.a Compiler builds from source files
3.b Web server watches file system
3.c Local app is launched in browser
4. We update any source file
4.a Compiler notices updated files
4.b Compiler rebuilds from source files
4.c Web server notices updated files
4.d Local app in browser is updated

Figure 2.1: Three commands that will take you from nothing to a working React application. From there, you can update the source files, and the system will automatically re-compile and update your application in the browser.

If you have a modern version of node (and npm) installed as advised in the introduction, you should be able to write the following command in your terminal:

```
$ npx create-react-app name-of-app
```

> **NOTE** `npx` **is not a typo –** `npx` **is a package runner tool that comes with npm.**

Run this command and you will have a new React application set up for you! The first time you run this command, npm will ask for confirmation to download the `create-react-app` utility (just press enter to confirm that). This refers to steps 1.a and 1.b in figure 2.1. Every time you use the command after that, there will be no questions asked, just React goodness ready. We will be referring to the `create-react-app` tool as *CRA* in the following sections.

The command will create a new folder with the passed name - in the above case `name-of-app`. Inside this folder, the utility will initialize a new git project, download the required resources for the application, and then download and locally install all the dependencies required by the project.

The command will run for a short while, probably around 1-3 minutes depending on the project complexity and network conditions. Once the command is complete, you will see something along the lines of:

```
Success! Created name-of-app at <folder>    #A
Inside that directory, you can run several commands:

  npm start    #B
    Starts the development server.

  npm run build    #C
    Bundles the app into static files for production.

  npm test    #C
    Starts the test runner.

  npm run eject    #C
    Removes this tool and copies build dependencies, configuration
    files and scripts into the app directory. If you do this, you
    can't go back!

We suggest that you begin by typing:

  cd name-of-app    #D
  npm start    #B

Happy hacking!
```

#A name-of-app and <folder> will of course be replaced by the actual name and folder location of your project
#B This is the first of four commands, that you can run in your application - we will discuss "start" right below
#C These three other commands will be discussed in the next subsection
#D This is simply a command to change the folder to the newly created project

Uh, exciting. Note that if your output mentions a command called `yarn` rather than `npm`, don't worry. See the sidebar for an explanation.

npm versus yarn

There are two popular package managers for JavaScript projects that work on the same package repository and structure, but with slightly different commands.

These two managers are npm and yarn respectively. The first, npm, comes pre-installed with node and is the default manager that many people use.

However, you can opt to install a different manager, yarn, which will have a slightly simpler command structure. For the purposes of this book, there's no other difference between using npm and yarn, other than some slightly different syntax when typing commands. If you have yarn, you (most likely) also have npm installed, so that will (most likely) always work for you.

If you want to run a command defined in a JavaScript project named "`build`" you have to type `npm run build` when using npm, but just `yarn build`, when using yarn. If you just type `npm build`, it will not work correctly.

Do note that some run commands are aliased in npm, so `npm run start` can be typed as just `npm start`, and `npm run test` can be typed as `npm test`. These are two special cases though and not the general case for all other commands.

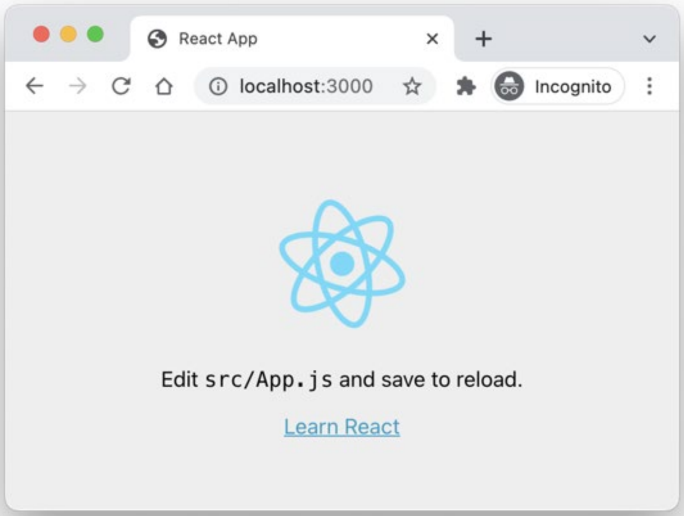
Let's do what's suggested above and run those two commands:

```
$ cd name-of-app
$ npm start
```

Now the third part of the magic happens. A React development server starts up, compiling all the files and resources used (action 3.a in figure 2.1) and spins up a local development web server (action 3.b in figure 2.1). This takes a few seconds and when done, the command line will say something like this:

```
Compiled successfully!
You can now view name-of-app in the browser.
  Local:            http://localhost:3000
```

Moreover, the application will already have been launched in your browser, as the command also launches a browser window at the proper URL (action 3.c in figure 2.1). But if not, simply open `localhost:3000` in your browser, to see the application.



Edit `src/App.js` and save to reload.

Figure 2.2: The default React application launched by a new React project. Your application will most likely be in *dark mode* with white text on a dark background color. The colors in this screenshot have been inverted for better print results.

This browser window will display a React application (as in figure 2.2), that's been created for you by a template. It's the default template used for new React applications that do not specify a specific template to use. We will discuss templates a bit later.

Note that this last command that you ran, `npm start`, is a continuously running command that stays active in the terminal. It will actually watch your source files, recompile the whole application when any source file changes, and even reload the browser with the updated application (actions 4.-4.d in figure 2.1)! Now that is pure magic.

If and when you want to abort this command, simply press Ctrl+C in your terminal and you will be back to your regular terminal prompt. However, your application no longer works, because you also stopped the local development web server.

You might have noticed that the previous output from creating our application listed not only the `start` command that we just used but also three other commands: `build`, `test`, and `eject`. We'll go over all these four commands in some more detail in the next subsection.

2.1.1 React project commands

Now that you have this React application source code available on your system, there's probably quite a few ways you want to interact with it.

The two primary things you want to be able to do is to see what you're developing as you're developing it, as well as deploy your application to a web server.

Another thing that you might also want to be able to do, is to run all tests in your application to verify that everything is (still) working as designed. Finally, you might want to be able to escape the confines of CRA to be able to tinker with the engine underneath. CRA abstracts some things away from you that you don't need to worry about at first, but when applications get more advanced, you might want to access the innards of your application configuration.

For these four purposes, a new React application created with CRA comes with these four commands:

- `start`: Launch a local development web server and continuously compile the project as it changes, serving it to any local browser.
- `build`: compile all resources into a production-ready package deployable to the right web host.
- `test`: Launch a test runner that will run all unit tests defined in your project.
- `eject`: Reveal the insides of the project and make it fully configurable.

Let's go over these one by one and discuss how and when to use them.

START

The `start` command is your primary command, the one you use every time you start a new project or pick up an old project to start work on it again. At the beginning of your coding session, you will run the start command in a terminal, and then you'll be ready to code in your editor while automatically being served the updated content in your browser.

The start command will build your project in the background continuously using the development version of React and its utility libraries. This is distinct from the production version of React used in the build command. The development version of React includes much better error messages and warnings as well as options for debugging the application as it is running in the browser. However, the development version of React is, for those reasons, also much larger in terms of sheer file size, so you do not want to publish your application using this version. It will make your application unnecessarily large and hinder users trying to access it.

The start command will also reload the application in the browser as it is running, but in a much smarter way than just reloading the whole browser window. React will try to reload only the relevant bits of logic that have changed and otherwise leave the application as-is. This means that for instance if you have clicked on a button to collapse a section that would otherwise be open by default when the application launches, React will be able to inject updated code while keeping the state in the browser, so this section remains collapsed while the logic is otherwise updated.

BUILD

This is the command to run when you are ready to see your application deployed to a real web server and have users try it out.

When you run the build command, you will be using the production version of React, which is much leaner and optimized for deployment. The result of the build will be put in the `/build` folder.

By default, nothing else really happens, but you can set up direct deployment to your cloud web hosting solution in the build command as well if you want to. Check your cloud web hosting provider's documentation for help on how to do this.

We will not be using this command in this book, as we will be using another template for deploying applications in the project, where deployment to the cloud will be an option.

TEST

If you want to run all unit tests defined in your project, run this command. You can do that on the empty default template as well, as the default template even comes with a default test file!

We will discuss testing React applications a lot more in a future chapter.

EJECT

This command can be a bit dangerous. If you eject your application, you will have access to a lot more configurable options inside the React setup than you do without it, but you also lose the option of automatically updating to newer versions of all the tools involved.

We will not cover ejecting your application in this book but will be discussing it again briefly in the pros and cons section ahead.

2.1.2 File structure

When you create a project with CRA, it will (almost) always follow the same file structure. Custom templates could do things differently, but rarely do.

The structure includes these important elements:

```
/
  public/
    index.html
  src/
    index.js
    App.js
  package.json
```

With these two folders and three files, you're good to go.

The public folder is for files that will just be served directly via the web server. This includes the `index.html` that serves your entire application as well as binary files that you do not wish to bundle inside your application. This includes content that is required by the `index.html` file directly (such as favicon, CSS, fonts, or images for sharing) as well as large files such as videos and images.

The source (`src`) folder is where all your bundled JavaScript will go as well as any other content that you want to bundle as a single package. This is mostly just JavaScript, but could potentially also include CSS, icons, small images, JSON files, and more. The bundling starts at the `index.js` file inside the source folder. It is commonplace to have the main application reside in a file named `App.js` or `app.js` depending on personal preferences. Other than that, you have free flexibility. Some templates structure the content inside the src folder in subfolders which is necessary to structure larger projects.

`package.json` is the main configuration file for your project as required by npm and yarn. It is the starting file for your project and defines the dependencies as well as the commands that you can run as covered in section 2.1.1.

The root folder will often contain a ton of other configuration files required for various libraries included in the project. It is not uncommon to see custom templates with 20+ other configuration files at the root of the project.

Now let's move on to cover what custom templates are and how they help you.

2.1.3 Templates

While the default application that we saw in figure 2.2 is pretty nice, it doesn't always help you a lot. The default application does set you up to create a simple app in the same style as that app is created, but that might not be what you are looking for.

If you want to create an app using a specific technology stack or using React in a particular way, you probably want to use a different starting template to set you up correctly.

When you use CRA, you can specify a template to use. The default template is the one we saw above with the (spinning) React logo. If you want to specify another template, you can do so as an argument:

```
$ npx create-react-app name-of-app --template name-of-template
```

You can only use the name of a template that already exists. If it doesn't exist, the application will abort. Often people don't bother with choosing a template at all and just work with the default one. But if you know that you want a specific setup or want to start your codebase at a certain state, you can use a template that sets you up for exactly that.

Some commonly used templates include:

- Minimal templates with even fewer features than the default one–e.g. `--template minimal`. This one comes without images, CSS, tests, web vitals, and other minor niceness used in the default template.
- Variants of the default or the minimal template using TypeScript–e.g. `--template typescript` or `--template minimal-typescript`. This is useful for starting a new project using TypeScript. We will briefly introduce you to the world of TypeScript in chapter 13.

- Complex boilerplate setups created by other developers where you have a stack of certain dependencies already baked into your new application–e.g. `--template redux-typescript`, which comes prepackaged with Redux and TypeScript, or `--template rb`, which is a popular *React boilerplate* (hence the *rb*), that comes prepackaged with a ton of reputable libraries including Redux + Saga, Styled-Components, ESLint, husky, and much more.

One of the very useful things about the template system for CRA is that it is fully decentralized. Anyone can publish a package to npm and structure it in a way that allows you to use it as a base for your own applications.

That is of course also one of the downsides. If you find a template on npm, there's no saying whether it's actually any good or even does what it says. Here you should probably trust the wisdom of the crowd, so if it's popular, it's probably good.

One of the benefits of the fact that they allow any idiot to publish a template on npm is, that this includes *us*, the authors of this book. We will be using custom React templates for all examples and projects in this book. We'll get back to that in a second. First, we will discuss the advantages and drawbacks of using CRA.

2.1.4 Pros and cons

There's a big number of advantages of using CRA to create a new React application but as always such advantages have consequences. We've already discussed many advantages, but let's list them here anyway:

- **Simplicity**: You have less to worry about when setting up a new application. You get JSX transpiling, bundling, testing, automatic reload, and more for free, without dealing with all the interdependencies.
- **Upgradability**: You can easily upgrade to newer versions of React and all the other libraries used. We haven't actually discussed how to do that, but it's surprisingly simple. Just run `npm install --exact react-scripts@VERSION` and you can upgrade your entire project to the specific version of React scripts. Check the changelog for `react-scripts` for details.
- **Community**: With the deluge of available CRA templates and the very easy path to making more, you can probably always find a premade template that has just the right combination of tools that you want to use, so you don't have to deal with mixing them correctly.
- **Customization**: On top of a variety of templates, you still have the option of adding all the other plugins and libraries that you need for your project. Does your project interface with e.g. both Google Maps and Amazon AWS? Just go ahead and add their libraries and you should be good to go.

However, there are also some drawbacks. Some of them can be ignored or glossed over, but in some situations, you have to seek out other setups than what CRA can provide. We'll cover some of these situations below as well:

- **Understanding**: Without setting the whole project up from scratch, you will not actually know what goes into setting such a project up from scratch. If you find yourself in a position where you need a unique setup but have always relied on CRA, you might find yourself stranded quickly, because you never really paid attention to it. But that's the duality of all abstractions: you gain the benefit of not worrying at the cost of not actually knowing what's going on underneath.
- **Control**: You do lose control over which libraries are used. CRA (currently) uses webpack + BabelJS for JSX bundling and transpiling, but that's by no means the only player around. Recently tools such as esbuild, bun, SWC, and Rome have emerged that partially cover the same ground and you can't easily switch to one of those. You're stuck to the technology stack that CRA currently has chosen for you. On the other hand, that's also an advantage, because when another tool becomes standard and maybe even superior to Babel, CRA will adapt and use that instead–completely without you having to worry about it. For the instances where you insist on using a specific stack, you do have to set your project up from scratch. Or you can eject your application as described in section 2.1.1, which gives you extra configurability and control at the cost of losing upgradability.
- **Integration**: If you want to integrate your application in a server-side setup, CRA currently cannot help you. For projects based on website frameworks as described in the first chapter, you have to use the setups provided by those frameworks rather than CRA.

Weighing all over the above, we arrived at the conclusion that CRA is perfect for new developers. You get a lot of simplicity and fewer worries. Once you get more experienced, you can start to experiment outside of CRA. That's why we have used CRA for the examples and projects in this book.

2.2 Book examples

As mentioned, we will be using CRA for (almost) every project and example in this book. We do this with a few exceptions. Some of the examples and *one* of the projects require a more complex setup not supported by this tool so you will be starting it slightly differently. Also, the very first example you completed in the first chapter did not use this approach. But we promise that the very next example even in this chapter will.

All the templates will be named according to this structure:

```
rqXX-NAME
```

`rq` of course refers to *React Quickly*. The `XX` will be replaced with the chapter number and the last bit will be a custom short name for each example. Every example and project, that will be using CRA, will display the template name in a notification like this:

```
  rq02-nesting
```

Sometimes examples will contain multiple variants of the source code, and in such a case, each variant will come with its own template as shown above. There are also examples that come with suggestions for extra homework. In such a case, there will be a template specifically pointed out as the starting point for that extra homework as well as another template with *one* possible solution to said work. You can use the solution template as either inspiration or compare it with your own solution. Beware that all such homework of course has infinite possible solutions, so just because your work doesn't match the template, does not mean it's wrong. It's just different.

For ease of use, you can often use the template name as your application name as well. So let's say you want to start working on the next example in this book. The template name is `rq02-nesting` so let's use that as the app name as well:

```
$ npx create-react-app rq02-nesting --template rq02-nesting
```

Just type that in your console, and you're already up and running and ready to tackle the example to work on the problem along with us if you so desire. You can however also just read the chapter and view the code in the listings in the book. If you find some things odd or need to get your fingers into the code to try some things out, only then could you instantiate the templates and see the examples in action.

Now let's get on with this example, which seems to be about nesting something.

2.3 Nesting elements

In the last chapter, you learned how to create a React element. As a reminder, the method you use is `React.createElement()`. For example, you can create a link element like this:

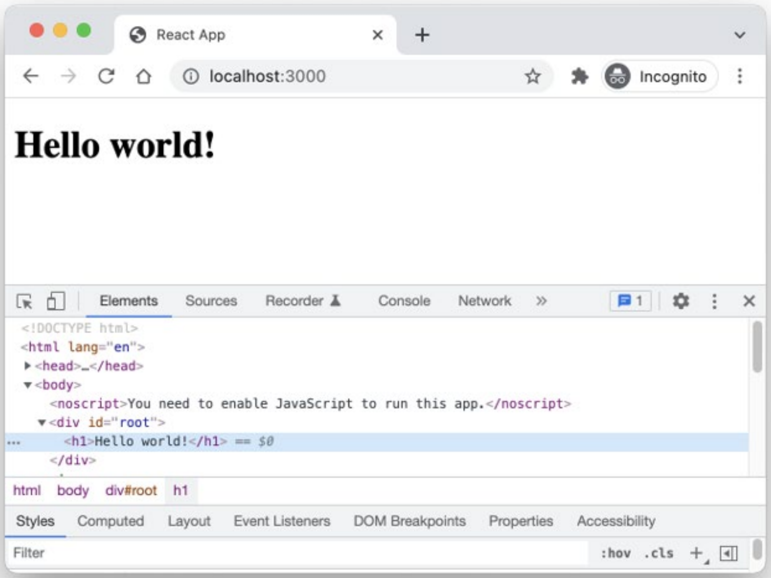
```
const reactLinkElement = React.createElement('a',
  { href: 'http://reactjs.org' },
  'React Website'
)
```

This is fine as long as we're creating just a single element. The problem is that every website ever has more than one element - otherwise how would you have any other information than just a single paragraph?

The solution to creating multi-element structures in a hierarchical manner is nesting elements. In the previous chapter, you implemented your first React code by creating a single React element and rendering it in the DOM with `ReactDOM.render()`:

```
const reactElement = React.createElement('h1', null, 'Hello world!');
const domElement = document.getElementById('content');
ReactDOM.render(reactElement, domElement)
```

It's important to note that `ReactDOM.render()` can only take a single (root) React element as an argument, which is `reactElement` in this example. The resulting application is shown in figure 2.3.



Hello world!

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
   ▼<div id="root">
      <h1>Hello world!</h1> == $0
    </div>
```

html body div#root h1

Figure 2.3 Rendering a single heading element

```
rq02-nesting
```

When you check out the template `rq02-nesting`, you will have the above application just this time created using CRA.

Remember that when you use `createElement`, the third argument is the child of the element. In this case, we just supply simple text as the child. But that text is actually another element - at least in the resulting DOM. In React, it does not have a specific element type, but it still functions as an element to some extent. We can show this relationship in a very simple diagram:

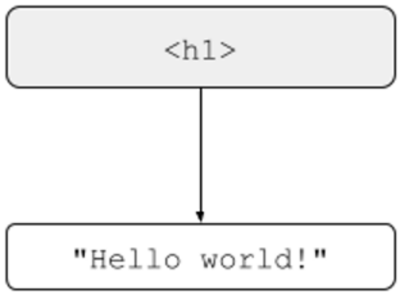


Figure 2.4 The gray node is a real React element, whereas the white element is just a text element.

2.3.1 Node hierarchy

Before we look at how we can create complex HTML structures, we need a bit of basic terminology in place first. The HTML document is often represented as an upside-down tree. Nodes in a tree are often described in a family-like fashion (parent, child, etc). Please refer to figure 2.5 for the following descriptions.

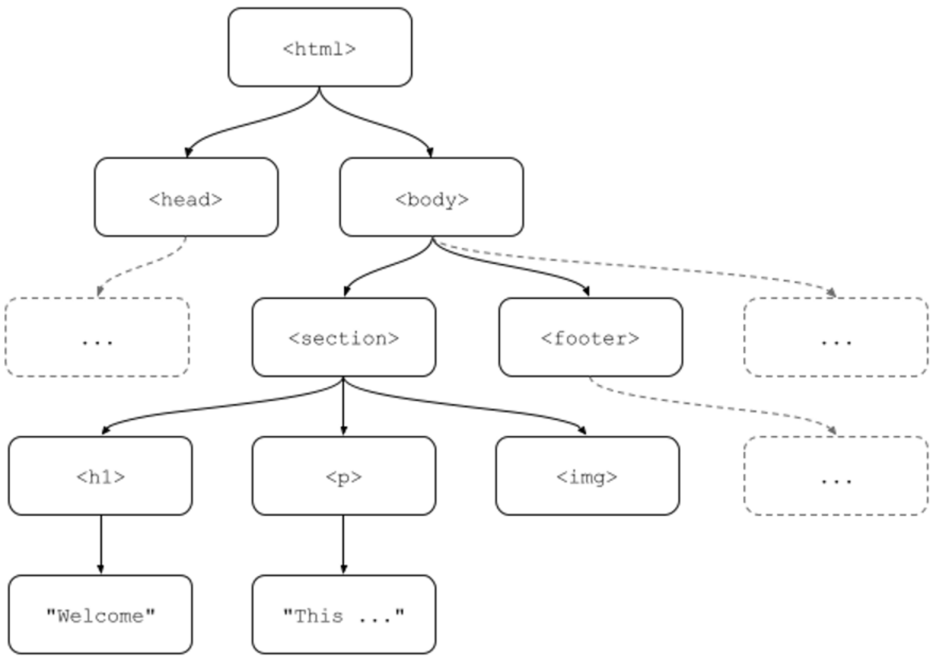


Figure 2.5 The document tree with various nodes

Node: Any member in the tree is a node. This includes both HTML elements and text nodes. All the boxes in figure 2.5 are nodes. The two bottom-most boxes are text nodes and all the others are element nodes.

Root: The first (top-most) node is the root of the tree. In figure 2.5 the `<html>` node is the root node.

Parent: The node directly above a given node is its parent. Every node in a tree has only one parent. The node above that can be called the grandparent, and so on. In figure 2.5, the parent node of the `<body>` is the `<html>` node. The root node doesn't have a parent, as the only node in the tree without a parent.

Child: Any node directly below a given node is a child of that node. A node can have multiple children. The children of the `<section>` node are the `<h1>`, `<p>`, and `<img>` nodes. Not all nodes have children. The `<img>` element does not have children. Text nodes never have children.

Sibling: Two nodes that have the same parent are considered sibling nodes. The `<p>` node has two sibling nodes: the `<h1>` and `<img>` nodes.

Ascendants: The parent of a node and its parent and its parent, etc all the way up to the root, are called the ascendants of a node. The `<h1>` node has three ascendants: The `<section>`, `<body>` and `<html>` nodes.

Descendants: The children of a node and all their children and all their children, etc, are called the descendants of a node. The `<section>` node has 5 descendants. Its three direct children as well as the two text nodes that are the grandchildren of the first two of those.

Nesting: Nesting is the process of organizing nodes in a tree and deciding which nodes will be the children of which other nodes and thus creating the document tree. In figure 2.5, we have decided to nest the `<h1>`, `<p>`, and `<img>` nodes inside the `<section>` node.

2.3.2 Simple nesting

Let's say you want to render the word *world* in italics in the string *"Hello world!"*, but still put all of it in an `h1` element. As shown in figure 2.6, you create an `em` element with the string `"world"` as a child and another `h1` element with three children:

1. the string `"Hello "` (note the space at the end),
2. the `em` element from before, and
3. the string `"!"`.

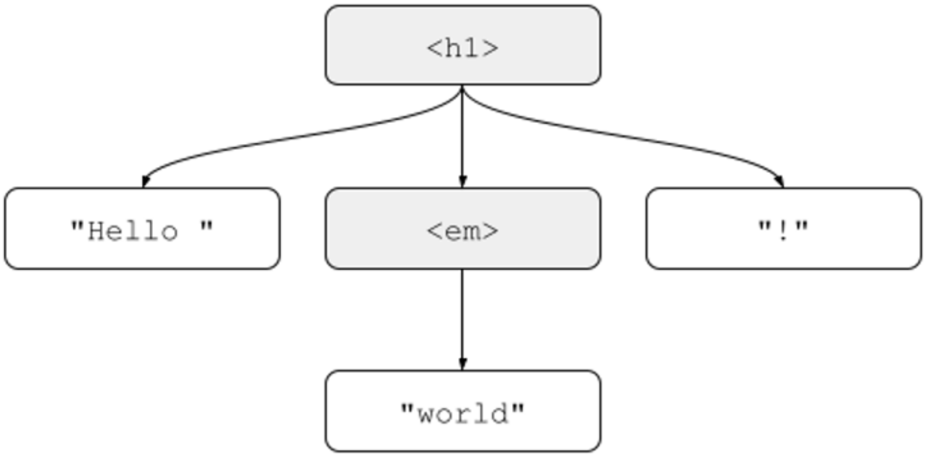


Figure 2.6: The two React elements and three text elements needed to render our slightly emphasized welcome message.

Using `React.createElement` this becomes:

```
const world = React.createElement('em', null, 'world');    #A
const title = React.createElement(
  'h1', null, 'Hello ', world, '!',    #B
);
```

#A createElement with 3 arguments
#B createElement with 5 arguments

As you can see here, we are passing five arguments to `createElement` now. First the element type, then the properties, and finally the children of the element. You can pass as many arguments as children to an element as you need. You can also pass the child elements as an array as in:

```
const title = React.createElement('h1', null, ['Hello ', world, '!'])
```

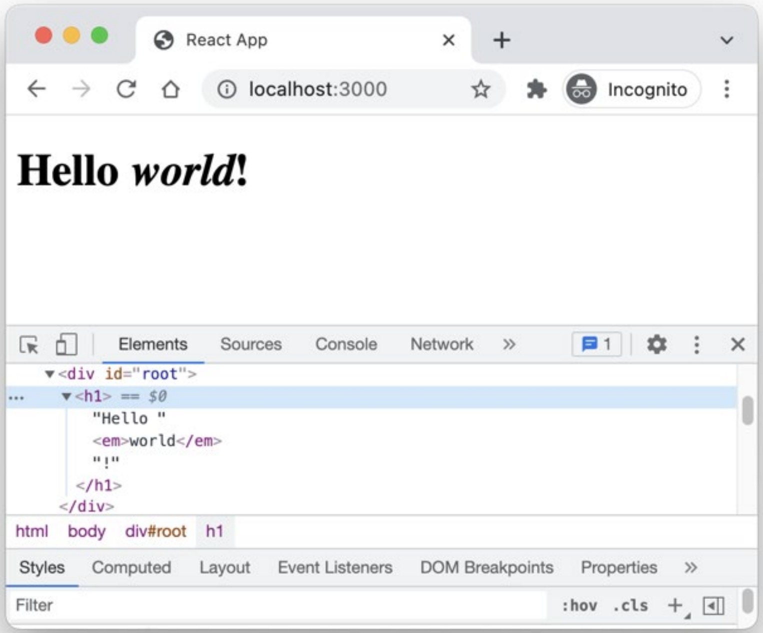
In this case, it does not make sense to put the elements into an array before passing them as an argument, but if we already had an array of elements, we could just pass that as an argument by itself.

Putting this all together (without using an array), the whole script becomes listing 2.1.

Listing 2.1 Emphatically greeting the world

```
const world = React.createElement('em', null, 'world');
const title = React.createElement('h1', null, 'Hello ', world, '!');
const domElement = document.getElementById('root');
ReactDOM.render(title, domElement);
```

If we put this into action, we end up with our application looking like figure 2.6 in the browser.



Hello *world*!

```
<div id="root">
  <h1> == $0
    "Hello "
    <em>world</em>
    "!"
  </h1>
</div>
```

Figure 2.7: Emphasized greeting in the browser

```
rq02-nesting-italic
```

But what if you wanted to put an element *after* the h1 and not just *inside* it? We'll cover sibling elements in the next section.

2.3.3 Siblings

In many instances, you can only use a single React element at the "top-level". This goes for the `ReactDOM.render` method - only a single element can be rendered into the DOM as the root element. We will also see how custom components can only return a single element a bit later.

But what if you wanted to show a headline **and** then a link after it in our example from before as in figure 2.8? That would be two different elements next to each other. You cannot render that directly using `ReactDOM.render()`.

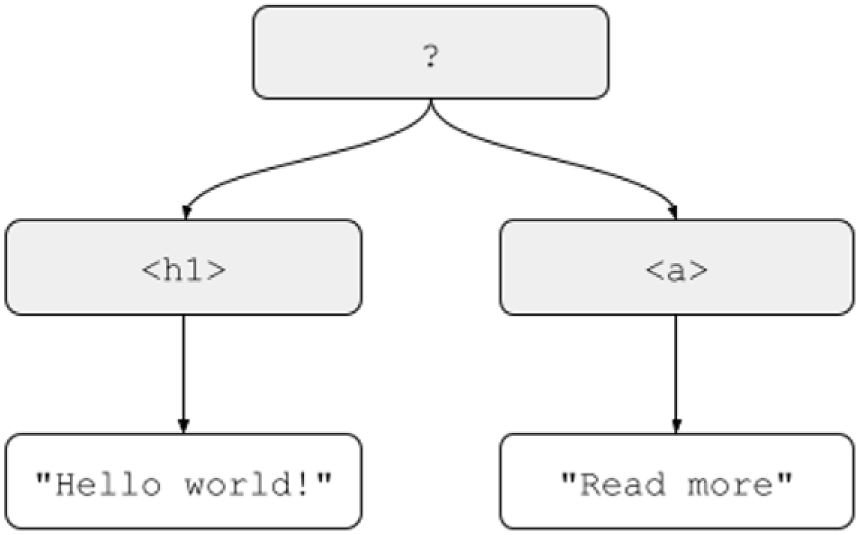
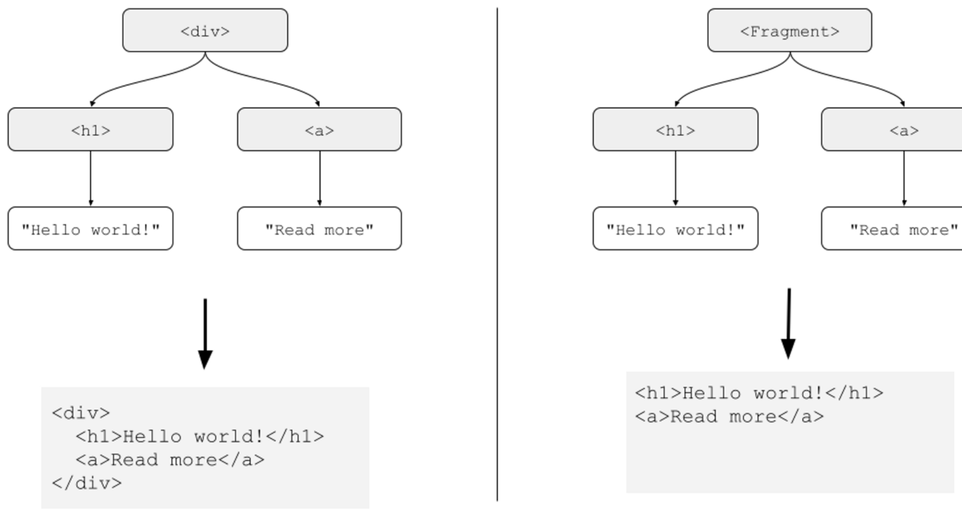


Figure 2.8: Two sibling React elements to be rendered in the root.

Instead you have to wrap them in another element (something in place of the `?` in figure 2.8). You have two different options here. One option is to use a neutral DOM element, which is easy, but would actually add a "physical" element to the output HTML. The alternative is to use a React fragment element, which works like any other element, but does not actually result in any output HTML itself. See the difference between these approaches in figure 2.9.



```
<div>
  <h1>Hello world!</h1>
  <a>Read more</a>
</div>
```

```
<h1>Hello world!</h1>
<a>Read more</a>
```

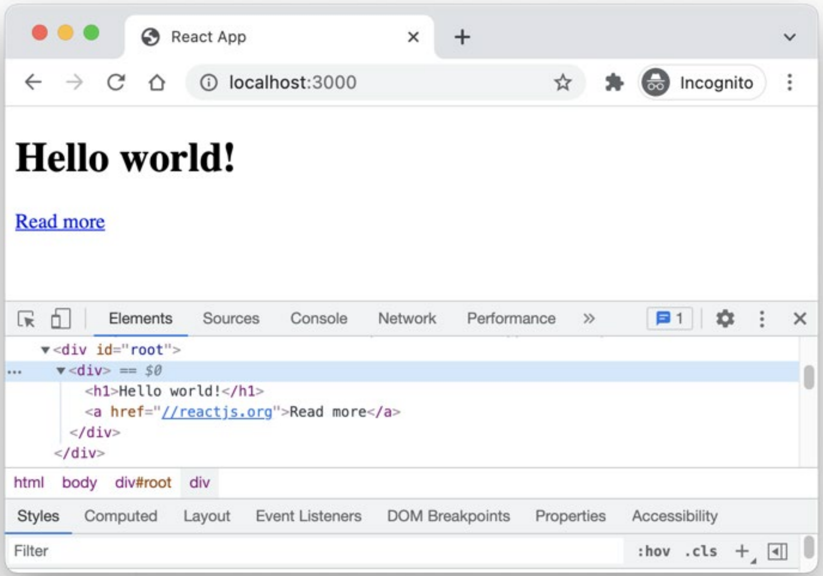
Figure 2.9: Two different approaches to sibling elements with different outputs

If you want to use a neutral DOM element, you can for instance use a `<div>` to group them as in listing 2.2.

Listing 2.2 Two elements in a grouping container

```
const title = React.createElement('h1', null, 'Hello world!');
const link = React.createElement('a', { href: '//reactjs.org' }, 'Read more');
const group = React.createElement('div', null, title, link);
const domElement = document.getElementById('root');
ReactDOM.render(group, domElement);
```

This results in the following HTML on the page as seen in figure 2.10.



```
▼<div id="root">
  ▼<div> == $0
      <h1>Hello world!</h1>
      <a href="//reactjs.org">Read more</a>
    </div>
  </div>
```

Figure 2.10: Title and link in a grouping element

```
rq02-siblings-div
```

The `<div>` container is usually a good choice for block-level content, or `<span>` for inline-level content.

But you don't actually have to use a "real" element. You can also create an empty React element, whose only purpose is to group multiple other elements, but doesn't actually output itself into the HTML on the page. This can be done with the magical component called `React.Fragment` and it can be used as the grouping element type. Let's do that in listing 2.3.

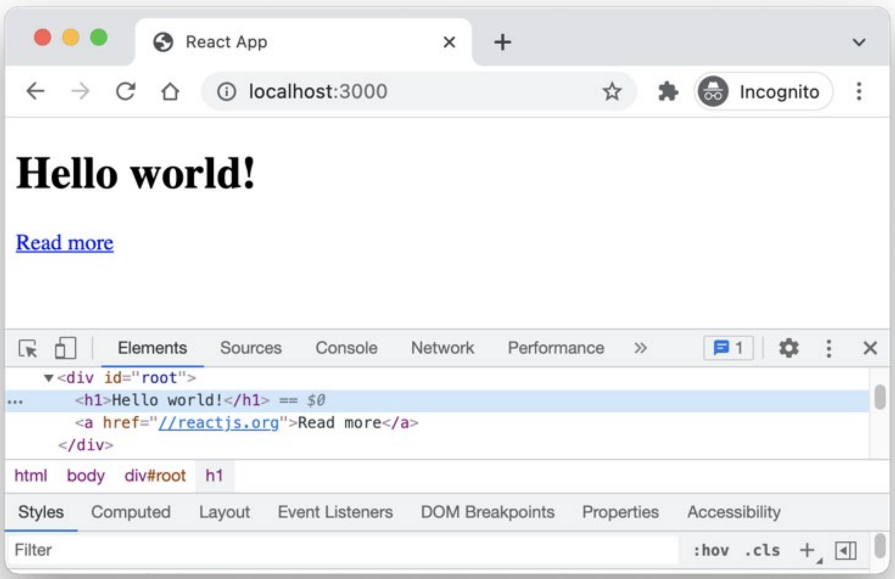
Listing 2.3 Two elements in a fragment

```
const title = React.createElement('h1', null, 'Hello world!');
const link = React.createElement('a', { href: '//reactjs.org' }, 'Read more');
const group = React.createElement(React.Fragment, null, title, link);    #A
const domElement = document.getElementById('root');
ReactDOM.render(group, domElement);
```

#A Notice the use of React.Fragment as the first argument to createElement

```
rq02-siblings-fragment
```

The output of this now becomes as in figure 2.11 in the browser.



```
<div id="root">
  <h1>Hello world!</h1> == $0
  <a href="//reactjs.org">Read more</a>
</div>
```

Figure 2.11 Title and link without a grouping element

You can of course also render the whole element in a single statement as follows:

```
const group = React.createElement(
  React.Fragment,
  null,
  React.createElement(
    'h1',
    null,
    'Hello world!',
  ),
  React.createElement(
    'a',
    { href: '' },
    'Read more',
  ),
);
```

This is functionally equivalent to the previous code, it just uses less variables. Some would argue it becomes more obvious, others that it becomes less readable.

So far, you've mostly provided string values as the first parameter of `createElement()`. But the first parameter can have two types of input, as we just saw with the fragments:

- Standard HTML tag as a string; for example, `'h1'`, `'div'`, or `'p'` (without the angle brackets). The name is in lowercase.

- React component as a reference (not a string). The name is normally capitalized.

The first approach renders standard HTML elements. You can actually use any string as a HTML tag name, regardless of whether it has a meaning in the browser by default. So while you will mostly be using normal HTML elements such as `div`, `main`, `section`, etc. there is nothing stopping you from creating a `tiny-horse` element, which would render as `<tiny-horse>` in the browser. It has no meaning and no default styling, but it would work.

In the second approach listed above, we can supply a React component as a reference. By this we do not mean the name of a React component as a string, but a direct reference to the component in question. We already saw one instance of that by using `React.Fragment`. Now let's look at how we can create our own custom components in the next section.

2.4 Creating custom components

After nesting elements with React, you'll soon stumble across the next problem: there are a lot of elements and a lot of repetition. You need to use the component-based architecture described in chapter 1, which lets you reuse code by separating the functionality into loosely coupled parts. Meet component classes, or just components, as they're often called for brevity (not to be confused with web components).

Think of standard HTML tags as building blocks. You can use them to compose your own React components, which you can use to create custom elements (instances of components). By using custom elements, you can encapsulate and abstract logic in composable reusable components. This abstraction allows teams to reuse UIs in large, complex applications as well as in different projects. Examples include panels, inputs, buttons, menus, and so on.

For this example, we want to create 3 identical links. It doesn't make a whole lot of sense to create identical links, but for now, we cannot customize them, so let's just go with this scenario. We want to create three links, that all say *"Read more about React"* and link to the React website at reactjs.org. And we want to wrap each link in a paragraph, so they go on separate lines.

There are two different approaches to this. We can do it the naive way by having three identical copies of elements or we can do it the smart way by creating a reusable Link component, and then instantiating it three times. This is illustrated in figure 2.11.

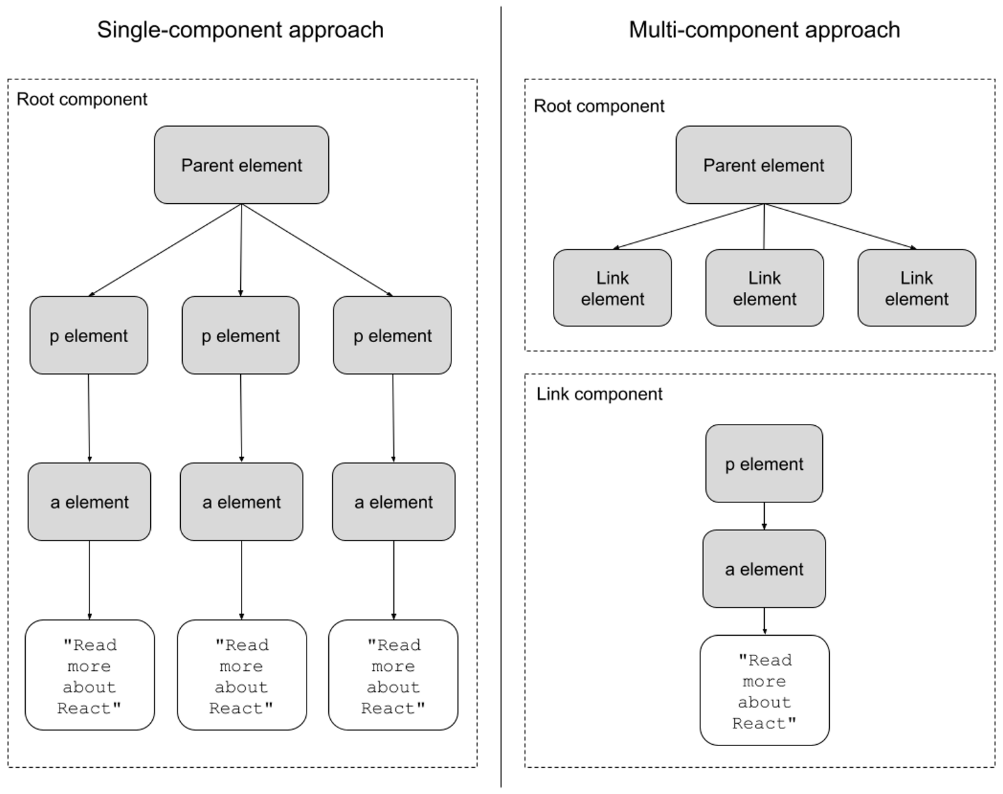


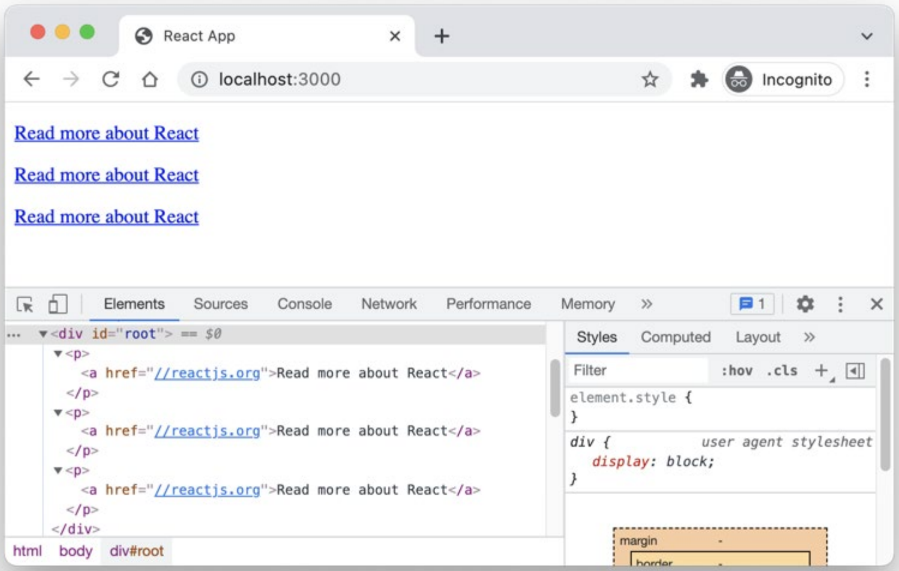
Figure 2.12 Two approaches to creating duplicate elements

Let's first look at the former approach, where we only use a single component with the copies manually duplicated. We want three independent links inside independent paragraphs, and we can do that in a fairly verbose way in listing 2.4.

Listing 2.4 Three links one time each

```
const link1 = React.createElement('a', {href: '//reactjs.org'}, 'Read more about React');
const para1 = React.createElement('p', null, link1);
const link2 = React.createElement('a', {href: '//reactjs.org'}, 'Read more about React');
const para2 = React.createElement('p', null, link2);
const link3 = React.createElement('a', {href: '//reactjs.org'}, 'Read more about React');
const para3 = React.createElement('p', null, link3);
const group = React.createElement(React.Fragment, null, para1, para2, para3);
const domElement = document.getElementById('root');
ReactDOM.render(group, domElement);
```

If we open this in the browser, we have the result you can see in Figure 2.13, which is exactly what we wanted.



Read more about React

Read more about React

Read more about React

```
▼<div id="root"> == $0
  ▼<p>
     <a href="//reactjs.org">Read more about React</a>
   </p>
  ▼<p>
     <a href="//reactjs.org">Read more about React</a>
   </p>
  ▼<p>
     <a href="//reactjs.org">Read more about React</a>
   </p>
 </div>
```

```
element.style {
}
div {                user agent stylesheet
   display: block;
}
```

Figure 2.13 Three identical links in our application

But we're repeating ourselves a lot. That is of course not desirable. The whole point of React and similar frameworks is to stop repeating ourselves at all. This calls for a custom component!

A custom component is a named object that contains other elements and component instances. So in this case, we could create a single Link component, that would render the link that we need in the correct way, and then we would include three instances of the Link component rather than the "raw" `<a>` element with all its properties.

You create a React component class by extending the `React.Component` class with `class CHILD extends PARENT` ES6 syntax. Let's create a custom `HelloWorld` component class using `class HelloWorld extends React.Component`.

The one mandatory thing you must implement for this new class is the `render()` method. This method must return a single root element created using `createElement()`, which is created from another custom component class or an HTML tag. Either can have nested elements if you so desire as long as there is only one root element.

Listing 2.5 Creating and rendering a React component class

```
class Link extends React.Component {                                              #A
  render() {                                                                      #B
    return React.createElement(                                                   #C
      'p',
      null,
      React.createElement(
        'a',
         {href: '//reactjs.org'},
        'Read more about React',
      )
    );
  }
}
const link1 = React.createElement(Link);                                          #D
const link2 = React.createElement(Link);                                          #D
const link3 = React.createElement(Link);                                          #D
const group = React.createElement(React.Fragment, null, link1, link2, link3);
const domElement = document.getElementById('root');
ReactDOM.render(group, domElement);
```

#A Defines a React component class with the capitalized name Link
#B Creates a render() method as an expression (function returning a single element)
#C Returns a new element with whatever we need for this component
#D Creates an instance of the new Link component

```
 rq02-custom-links
```

By convention, the names of variables containing React components are capitalized. This isn't required in regular JS. You could use the lowercase class name `someLink` in the above code instead of `Link` and it would still work. But because it's necessary for JSX (which we will cover in the next chapter), we apply this convention here as well.

Analogous to `ReactDOM.render()`, the `render()` method in `createClass()` can only return a single element. If you need to return multiple same-level elements, wrap them in a container component – either an HTML element or a React Fragment. If we now run this code in the browser, we get the exact same HTML as before as shown in Figure 2.13.

This new code is much more compact. We don't unnecessarily repeat ourselves and we have compartmentalized a part of our code, that we can reuse as much as we want.

This is the power of component reusability! It leads to faster development and fewer bugs. Components also have properties, lifecycle events, states, DOM events, and other features that let you make them interactive and self-contained; these are covered in the following chapters.

Right now, the links are all the same. Wouldn't it be awesome, if you could set element attributes and modify their content and/or behavior individually? Of course you can! Meet properties.

2.5 Working with properties

Properties are a cornerstone of the declarative style that React uses. Think of properties as unchangeable values within an element. They allow elements to have different variations if used in a view, such as changing a link URL by passing a new value for a property:

```
React.createElement('a', {href: '//reactjs.org'}, 'React')
```

One thing to remember is that properties are immutable within their components. A parent assigns properties to its children upon their creation. The child element isn't supposed to modify its properties. For instance, you can pass a property `PROPERTY_NAME` with the value `VALUE` to a component of type `Link`, like this:

```
React.createElement(Link, { PROPERTY_NAME: VALUE });
```

Properties closely resemble HTML attributes (as shown with the `href` on the link above). This is one of their purposes, but they also have another: you can use the properties of an element in your code as you wish. Properties can be used as follows:

- To render standard HTML attributes of an element: `href`, `title`, `style`, `class`, and so on.
- As custom instructions for components to make them render individually.

The object of properties can be accessed inside a component using `this.props`. This object is a frozen (immutable) object, from which you can only read values, not set them.

Frozen objects in JavaScript

Internally, React uses `Object.freeze()` which is a built-in function in JavaScript to make the `this.props` object immutable. To check whether an object is frozen, you can use the `Object.isFrozen()` method. For example, you can determine whether this statement will return true:

```
class Test extends React.Component {
 render() {
  console.log(Object.isFrozen(this.props))
  return React.createElement('div')
 }
}
```

The details of this are pretty complex, but for now just know that you should never try to edit or add properties inside a component itself. That is something you do in the parent context.

2.5.1 A single property

Let's start with a very simple example. We want the name of the framework in the links we created before to be customized. So we can say "Read more about React" in one link, "Read more about Vue" in the second, and "Read more about Angular" in the third.

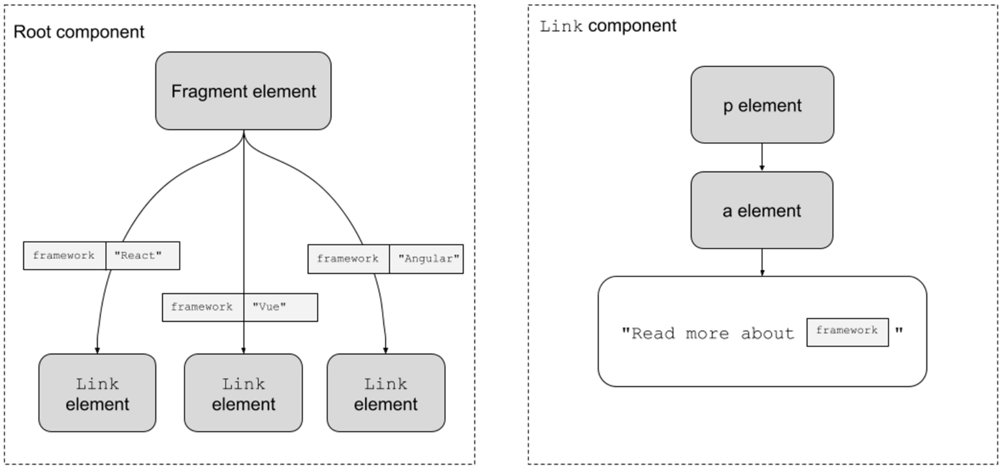


Figure 2.14: Passing a property to components and using the property inside the component.

To do this we need to do two things:

1. Pass a property to our component instances
2. Use the property inside the component

First of all, we need to pass a new property to the link instances. So rather than just doing this:

```
const link1 = React.createElement(Link);
```

We will instead be supplying an object as the second argument with a single property:

```
const link1 = React.createElement(Link, { framework: 'React' });
```

We used the variable name `framework` here. That is an arbitrary choice that we get to make as the component creator. We just need to make sure to use the same variable name in the second step.

We now need to use this passed property inside our class. Given that we call the variable `framework`, we will access it through `this.props.framework`.

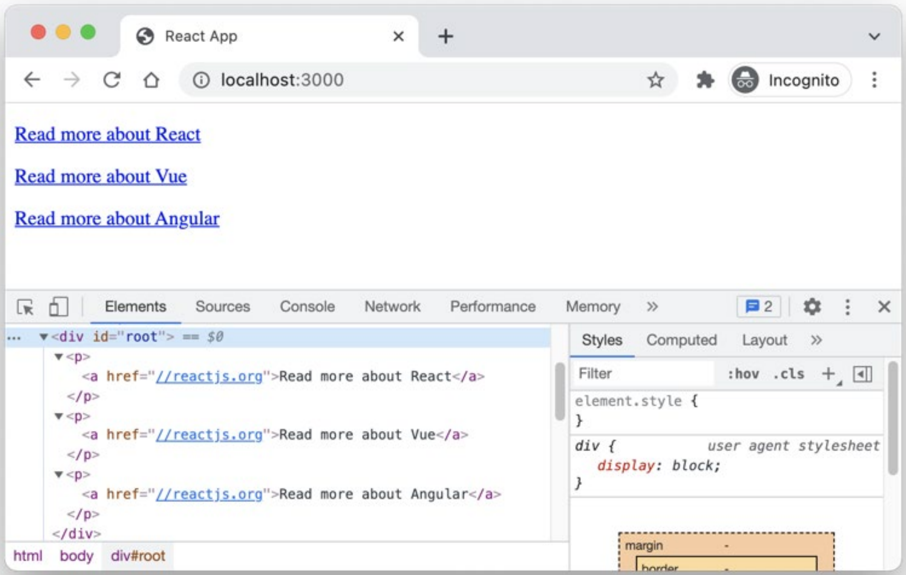
So our code all in all becomes as you can see in listing 2.6.

Listing 2.6 Link instances with different text

```
class Link extends React.Component {
  render() {
    return React.createElement(
      'p',
      null,
      React.createElement(
        'a',
        { href: '//reactjs.org' },
        `Read more about ${this.props.framework}`,    #A
      ),
    );
  }
}
const link1 = React.createElement(Link, { framework: 'React' });    #B
const link2 = React.createElement(Link, { framework: 'Vue' });    #C
const link3 = React.createElement(Link, { framework: 'Angular' });    #D
const group = React.createElement(React.Fragment, null, link1, link2, link3);
const domElement = document.getElementById('root');
ReactDOM.render(group, domElement);
```

#A We render the text content of the link by combining this.props.framework with some static content.
#B The first instance of our link component uses "React" as the framework property
#C The second instance of our link component uses "Vue" as the framework property
#D The third instance of our link component uses "Angular" as the framework property

You can see this in action in the browser in figure 2.15.



Read more about React

Read more about Vue

Read more about Angular

```
▼<div id="root"> == $0
  ▼<p>
     <a href="//reactjs.org">Read more about React</a>
   </p>
  ▼<p>
     <a href="//reactjs.org">Read more about Vue</a>
   </p>
  ▼<p>
     <a href="//reactjs.org">Read more about Angular</a>
   </p>
 </div>
```

```
element.style {
}
div {                    user agent stylesheet
  display: block;
}
```

Figure 2.15 Three links with different text

2.5.2 Multiple properties

You may have noticed that the links still point to the same URL, which is the React website. That is of course no good, we need the URLs to be different. Using the same approach, we simply invent a new property, `url`, and use it inside the component as well as in the component instances. You can see that illustrated in the diagram in figure 2.16 and implemented in the code in listing 2.7.

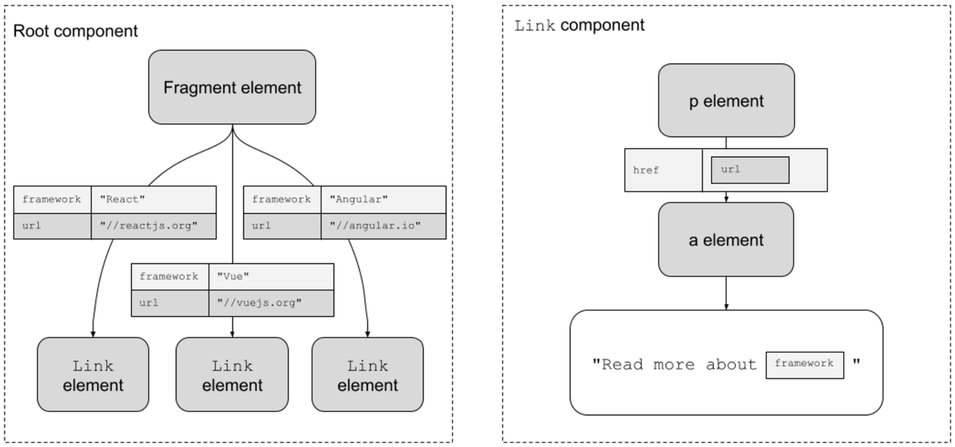


Figure 2.16: Two different properties are passed to our components.

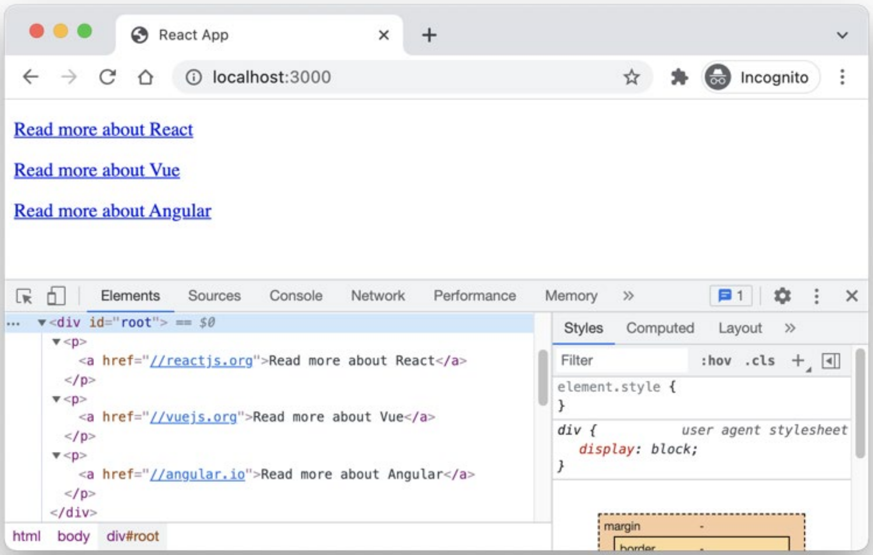
Listing 2.7 Link instances with different text and URLs

```
class Link extends React.Component {
  render() {
    return React.createElement(
      'p',
      null,
      React.createElement(
        'a',
        { href: this.props.url },    #A
        `Read more about ${this.props.framework}`,
      ),
    );
  }
}
const link1 = React.createElement(
  Link,
  { framework: 'React', url: '//reactjs.org' }    #B
);
const link2 = React.createElement(
  Link,
  { framework: 'Vue', url: '//vuejs.org' }    #C
);
const link3 = React.createElement(
  Link,
  { framework: 'Angular', url: '//angular.io' }    #D
);
const group = React.createElement(React.Fragment, null, link1, link2, link3);
const domElement = document.getElementById('root');
ReactDOM.render(group, domElement);
```

#A Using the url property to set the href property on the <a> element
#B The URL for React is reactjs.org
#C The URL for Vue is vuejs.org
#D The URL for Angular is angular.io

```
rq02-link-props
```

You can see this in action in the browser in figure 2.17.



Read more about React

Read more about Vue

Read more about Angular

```
▼<div id="root"> == $0
  ▼<p>
     <a href="//reactjs.org">Read more about React</a>
   </p>
  ▼<p>
     <a href="//vuejs.org">Read more about Vue</a>
   </p>
  ▼<p>
     <a href="//angular.io">Read more about Angular</a>
   </p>
 </div>
```

```
element.style {
}
div {                user agent stylesheet
   display: block;
}
```

Figure 2.17 Three links with different text and URL

As you can see, we can use properties on both custom components (which are used inside the component to customize the returned structure) and HTML elements (which set HTML attributes).

What happens if you mess up - what happens if you set a custom property on an HTML element? React will render it anyway. Before React 16 invalid properties would be filtered out, but because modern web applications often use other third-party libraries, that might rely on some custom properties, React 16 and onward will allow you to use whichever properties you feel like.

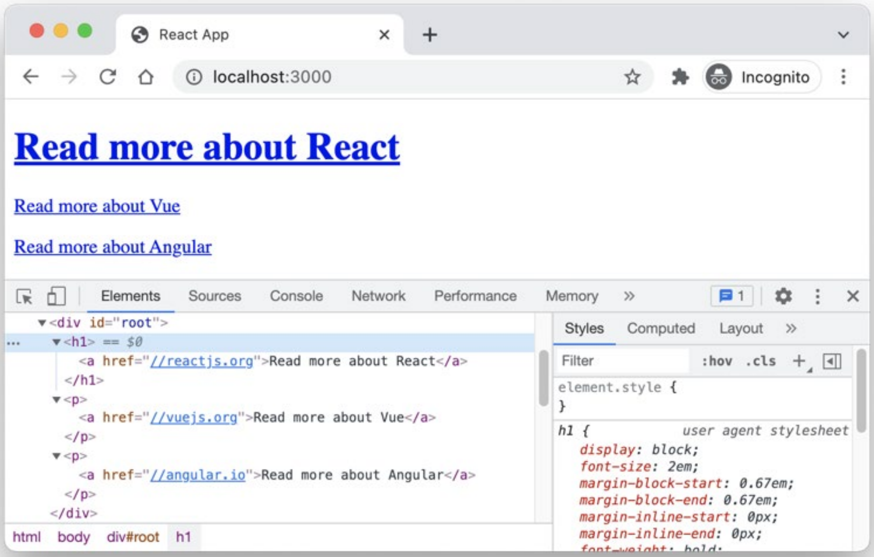
You can completely modify the rendered elements based on the value of a property. For example, we can examine the `framework` property and return a huge title with a link in case the framework name is "React":

```
class Link extends React.Component {
  render() {
    const link = React.createElement(     #A
      'a',
      { href: this.props.url },
      `Read more about ${this.props.framework}`,
    );
    if (this.props.framework === 'React') {    #B
      return React.createElement('h1', null, link);    #C
    }
    return React.createElement('p', null, link);    #D
  }
}
```

#A Create a link element and store it in a variable
#B Check if the framework matches "React"
#C If it does, return an h1 element with the link inside
#D Otherwise, return a paragraph element with the link inside

This is also a great example of React elements being just plain old JavaScript. We can create an element and store it in a variable – and then later use that variable as we see fit. And we can create branching using regular JavaScript functionality.

If we render this new component in the browser, suddenly the links are not identical anymore as you can see in figure 2.18.



Read more about React

Read more about Vue

Read more about Angular

```
<div id="root">
  <h1> == $0
    <a href="//reactjs.org">Read more about React</a>
  </h1>
  <p>
    <a href="//vuejs.org">Read more about Vue</a>
  </p>
  <p>
    <a href="//angular.io">Read more about Angular</a>
  </p>
</div>
```

```
element.style {
}
h1 {            user agent stylesheet
  display: block;
  font-size: 2em;
  margin-block-start: 0.67em;
  margin-block-end: 0.67em;
  margin-inline-start: 0px;
  margin-inline-end: 0px;
  font-weight: bold;
```

Figure 2.18 Three links but React is a lot more important

We've now covered several permutations of some very simple HTML that's almost useless by itself. But by starting small, we're building a solid foundation for future, more advanced topics. Believe us, you can achieve a lot of great things with custom components.

It's very important to know how React works in regular JavaScript if you (like many React developers) plan to use JSX. This is because in the end, browsers will still run regular JavaScript, and you'll need to understand the results of the JSX-to-JS transpiling from time to time. Going forward, we'll be using JSX, which is covered in the next chapter. But before we get to that, we need to discuss the structure of a React application a bit.

2.5.3 The special property: children

React elements take a special property, `children`. This is not a property you specify in the normal way, but you do use it as any other property.

Let's change our example a little bit and instead create a list of links where the text is just the framework name and not the text "Read more about" before it. This would look like this figure 2.19.

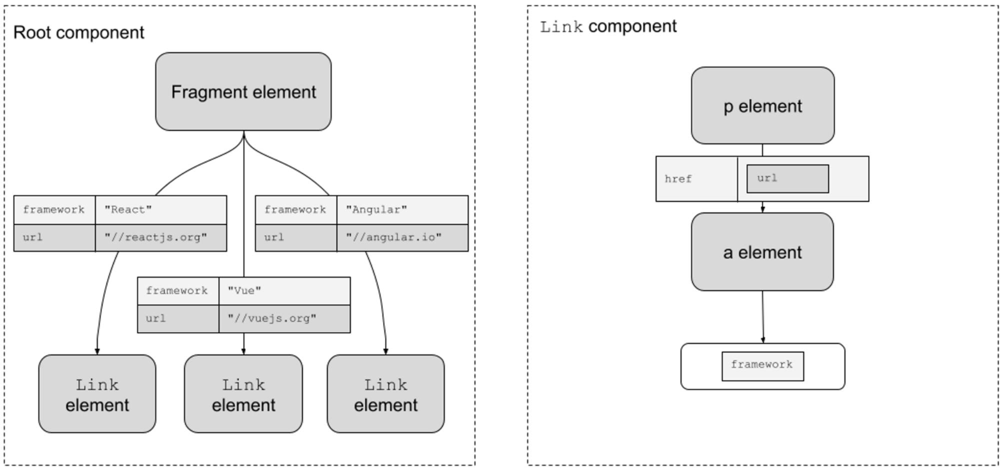


Figure 2.19 Our new structure with the links only containing the framework name.

Now let's take this one step further. Let's say we want the framework for React to be displayed in bold. We already know how to make a bold element, we just wrap it in an element as:

```
React.createElement("strong", null, "React");
```

But how are we going to pass that in as a property? Can we do that? We can, actually. We can create the node for the React framework as this:

```
const boldReact = React.createElement('strong', null, 'React');    #A
const link1 = React.createElement(
  Link,
  { framework: boldReact, url: '//reactjs.org' }    #B
);
```

#A We create a React element in the variable named boldReact
#B And then we pass that variable in as the property framework on the Link element

But that's a bit weird. We are now creating elements, but passing them in as properties. That's not what we normally do. What if we instead could create an element and pass it in as a child element?

Remember how the arguments 3 and onward to `React.createElement` are the children of the element. We haven't used that for custom components, but we can. All the nodes passed as the children to a custom element are accessible through `this.props.children`. That property is either a single node (if only passed one child element) or an array of nodes (if passed multiple child elements).

So, let's change our root component to contain three links, where link text is not passed in as a property named `framework`, but rather as a child node. For the first link, we still want to make the text bold. That would then look like figure 2.20.

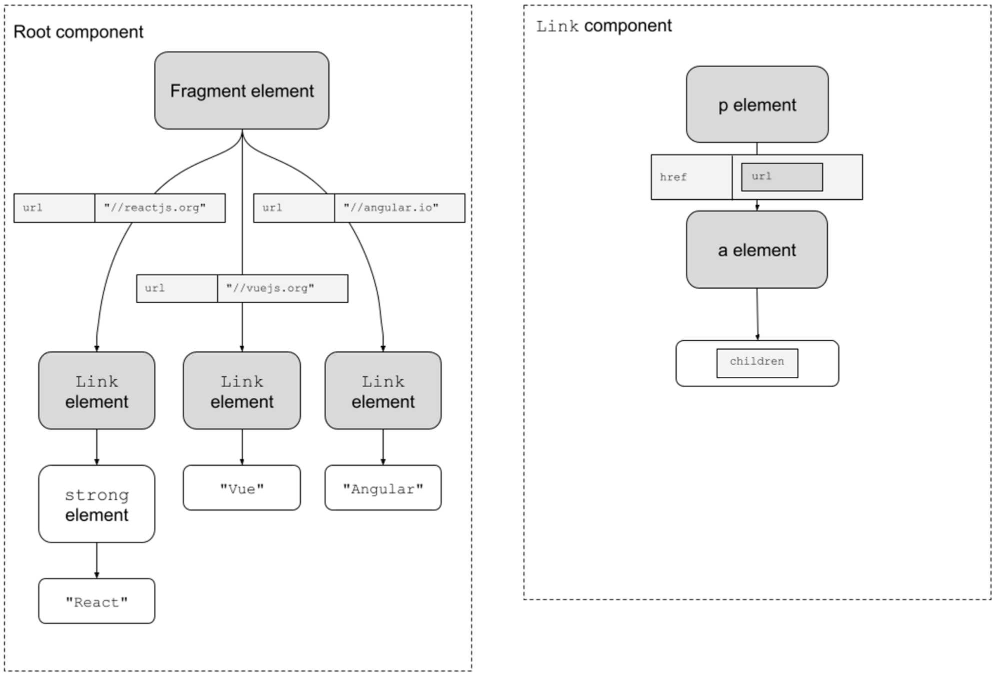


Figure 2.20 The component tree when we pass the link text as a child node rather than as a regular property

Let's go ahead and implement this in listing 2.8.

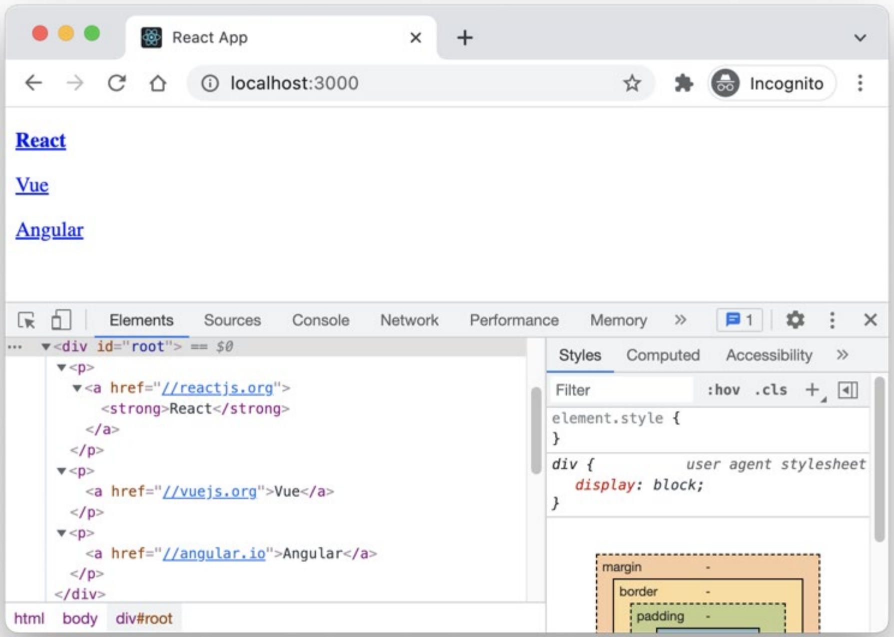
Listing 2.8 Links with text as child nodes .

```
class Link extends React.Component {
  render() {
    return React.createElement(
      'p',
      null,
      React.createElement(
        'a',
        { href: this.props.url },
        this.props.children,    #A
      ),
    );
  }
}
const link1 = React.createElement(
  Link,
  { url: '//reactjs.org' },    #B
  React.createElement('strong', null, 'React'),    #C
);
const link2 = React.createElement(
  Link,
  { url: '//vuejs.org' },    #B
  'Vue',    #D
);
const link3 = React.createElement(
  Link,
  { url: '//angular.io' },    #B
  'Angular',    #D
);
const group = React.createElement(React.Fragment, null, link1, link2, link3);
const domElement = document.getElementById('root');
ReactDOM.render(group, domElement);
```

#A Note how we use the property named children just as if it was any other property
#B We now only pass a single named property to each custom component, the url property.
#C But now we also pass a child node, which will become the children property. And for React, it's an HTML
#D For Vue and Angular, the child node is just a regular text node.

```
 rq02-links-children
```

If you run this in the browser, you get the output displayed in figure 2.21.



```
▼<div id="root"> == $0
  ▼<p>
    ▼<a href="//reactjs.org">
        <strong>React</strong>
      </a>
    </p>
  ▼<p>
      <a href="//vuejs.org">Vue</a>
    </p>
  ▼<p>
      <a href="//angular.io">Angular</a>
    </p>
  </div>
```

```
element.style {
}
div {                    user agent stylesheet
   display: block;
}
```

Figure 2.21 Our link components using children properties where the first link is bolded.

The distinction between using a normal property and the children property might seem insignificant at this point, but in the next chapter, when we start using JSX, you will see how it starts to make a lot of sense when used correctly.

2.6 Application structure

From the next chapter on, we're going to structure our applications in the same organized way with similar patterns for easy recognition. And we are going to follow the standard structure that the default CRA template also provides.

In the previous examples in this chapter, we directly put our application in the index.js file inside the source folder.

From now on, we are going to use a custom App component as the root element of our applications, and we will render that as the single child rendered to the browser. This means, that we will not have to touch `src/index.js` again at all. It will remain the same identical file for all future applications going forward that use CRA.

For this purpose, we will rewrite our application with three links from before as two new components. One is the root App and the other is the Link component. We will use the latter three times in the former. And finally, we will destructure some properties from the React namespace to shorten our component definition slightly. We will place all this in `src/App.js`. See figure 2.22 and listing 2.9.

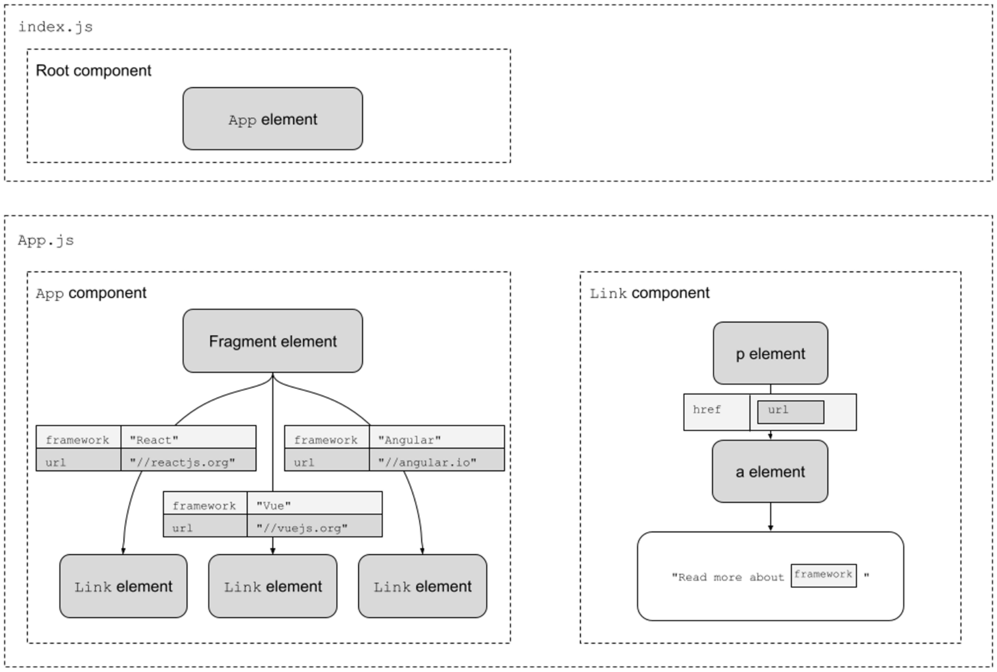


Figure 2.22: Our new file structure with our two components inside the app file.

Listing 2.9 Application that goes in src/App.js

```
import React, { Fragment, Component } from 'react';    #A
class Link extends Component {    #B
  render() {
    return React.createElement(
      'p',
      null,
      React.createElement(
        'a',
        {href: this.props.url},
        `Read more about ${this.props.framework}`,
      ),
    );
  }
}
class App extends Component {    #C
  render() {
    const link1 = React.createElement(
      Link,
      { framework: 'React', url: '//reactjs.org' }
    );
    const link2 = React.createElement(
      Link,
      { framework: 'Vue', url: '//vuejs.org' }
    );
    const link3 = React.createElement(
      Link,
       { framework: 'Angular', url: '//angular.io' }
    );
    return React.createElement(Fragment, null, link1, link2, link3);    #D
  }
}
export default App;    #E
```

#A Destructuring the import of React to be able to directly reference Fragment and Component
#B Our Link component exactly as in the previous listing
#C A new App component, that renders the root of our application
#D Our App component returns a single element as all components must
#E In App.js, we export the App component, as the single accessible asset inside this file

We then change `src/index.js` to import our App from `App.js` and render that into the root DOM element as in listing 2.10.

Listing 2.10 src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';    #A

ReactDOM.render(
  React.createElement(App),    #B
  document.getElementById('root')    #C
);
```

#A Importing our application from App.js and storing it in the local variable App
#B Creating a single root element from the App component
#C Render this into the HTML element with id="root".

This new `src/index.js` is now basically complete. We don't ever need to edit it again, we only edit `src/App.js` as we need to customize our future applications.

```
rq02-links-app
```

As our apps grow larger, we will grow out of including everything inside a single file in src/App.js. When we need to grow, we can just create new files and import those as we need to. It is customary to create a single file per component and name the file after the component (including the uppercase first letter), but it is not a strict rule. If a component needs several other small components to function, you can freely decide whether you want to put it all in one file, as we did with the Link and App in Listing 2.10, or split it up into multiple files.

Let's see how we would do this, if we wanted to. Please refer to figure 2.23.

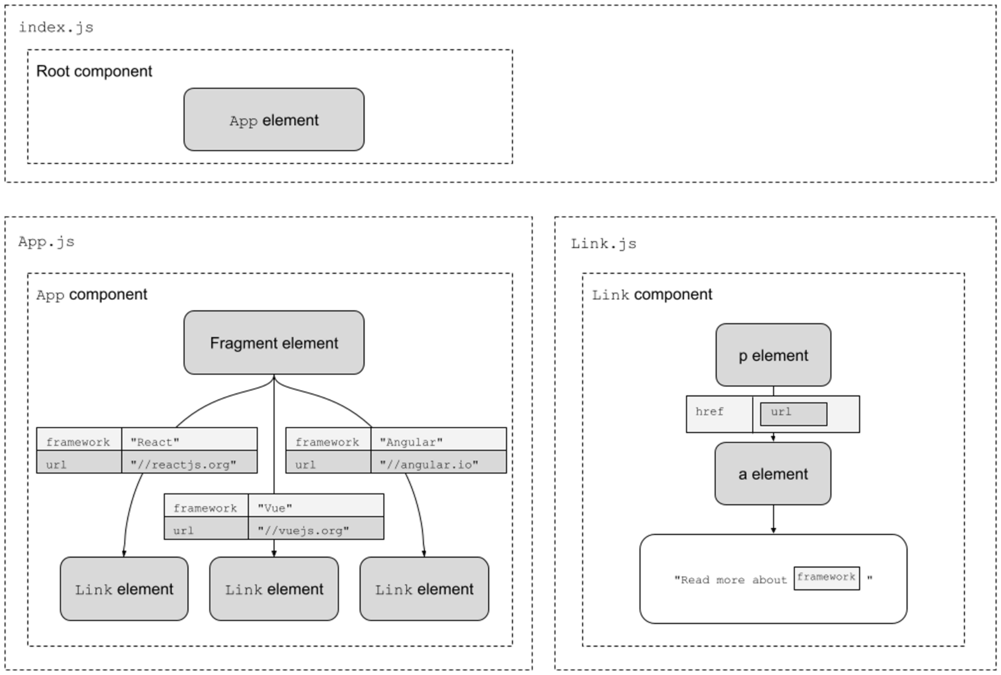


Figure 2.23: Using one file per component, our file structure looks like this.

We would need to update `src/App.js` to import the Link component from `src/Link.js` and we do that in listing 2.11.

Listing 2.11 One component per file: src/App.js

```
import React, { Fragment, Component } from 'react';
import Link from './Link';    #A
class App extends Component {
  render() {
    const link1 = React.createElement(
      Link,
      { framework: 'React', url: '//reactjs.org' }
    );
    const link2 = React.createElement(
      Link,
      { framework: 'Vue', url: '//vuejs.org' }
    );
    const link3 = React.createElement(
      Link,
       { framework: 'Angular', url: '//angular.io' }
    );
    return React.createElement(Fragment, null, link1, link2, link3);
  }
}
export default App;
```

#A Importing the Link component from another file

Then we must create the new `src/Link.js` with only the Link component and remember to export that at the end. See listing 2.12.

Listing 2.12 One component per file: src/Link.js

```
import React, { Component } from 'react';
class Link extends Component {    #A
  render() {
    return React.createElement(
      'p',
      null,
      React.createElement(
        'a',
        {href: this.props.url},
        `Read more about ${this.props.framework}`,
      ),
    );
  }
}
export default Link;    #B
```

#A The Link component definition is now alone in this file
#B Remember to export the component at the end

```
 rq02-links-app-alt
```

We will be using both approaches throughout the book. In the next few chapters, our applications will be small and compact, so we will put everything inside `src/App.js`. But in later chapters, our applications grow larger and we will start growing out of just a single file. We will also be creating files for other things than components. It could be files for commonly used functions or other shared functionality that we want to use in many files. We will also be using separate files for custom hooks when we get to that topic in chapter 9.

When projects grow even larger, folder structures are introduced. There are no defined standards for how to use folders to structure a React project, so teams often come up with their own best practices.

2.7 Quiz

1. A custom React component can be created with which of the following statements?
   a) `const Name = React.createComponent()`
   b) `class Name extends React.Component`
   c) `const Name = React.createElement()`
   d) `class Name extends React.Class`
2. The only mandatory member of a React component is which of the following?
   a) `function`
   b) `return`
   c) `name`
   d) `render`
   e) `class`
3. To access the `url` property inside a component, you use which of the following?
   a) `this.properties.url`
   b) `this.data.url`
   c) `this.props.url`
   d) `url`
4. React properties are immutable inside the component itself. *True* or *false*?
5. React components allow developers to create reusable UIs. *True* or *false*?

2.8 Summary

- You can create new React projects using the command-line program create-react-app
- New React projects can be created from a specified template and all examples in this book come with a template to allow you to see the example locally in three short commands without having to locate or download anything directly.
- You can nest React elements using third, fourth, and so on arguments in `createElement()`.
- Create elements from custom component classes.
- Modify the resulting elements using properties.
- You can pass properties to child element(s).
- To use a component-based architecture (one of the features of React), you create custom components.
- Examples in this book will all follow a very simple file structure.

2.9 Quiz answers

1. `class Name extends React.Component`. Also, there's no `React.Class` nor `React.createComponent` and `React.createElement` is used for creating component instances, not component definitions.
2. `render()` because it's the only required method. Also because `function`, `return`, and `class` are not valid method names.
3. `this.props.url` because only `this.props` gives the properties object.
4. True. It's impossible to change a property inside the component itself.
5. True. Developers use components to create reusable UIs.

3 Introduction to JSX

This chapter covers

- **Understanding JSX and its benefits**
- **Using JSX to implement custom components faster and easier**
- **React and JSX gotchas**

JSX is a syntax extension to JavaScript. It is one of the things that make React great, but also one of the more controversial elements of React when it was introduced back in the day.

This is an example of using JSX in JavaScript:

```
const link = <a href="//reactjs.org">React</a>;
```

JSX is the element there between the angle brackets: `<a href="//reactjs.org">React</a>`. It's not a string. It's not a template literal. It's not HTML. It's a JavaScript object, but created with the syntax extension called JSX. It makes creating React elements much faster and more compact, but also makes reading React elements much easier. And the latter is at least as important as the former.

JSX is made for developers only. It does not by itself do anything to make better nor faster web applications. It is converted to the exact same code as not using JSX would do.

JSX is not a requirement, but is universally accepted as the only way to write React components. You may find a few teams out there not using JSX, but they are by far the minority.

In this chapter we'll dive a bit more into the reasons for using JSX in the first place, then discuss all the different parts of applying JSX in practice, and finally some tricks that you need to pay attention to when using JSX. Along the way we will also briefly discuss converting JSX to JavaScript, so-called *transpiling*. Luckily it's not something you have to worry too much about.

> **NOTE** The source code for the examples in this chapter is available at https://github.com/rq2e/rq2e/tree/main/ch03. But as you learned in chapter 2, you can instantiate all the examples directly from the command line using a single command.

3.1 Why do we use JSX?

JSX is a JavaScript extension that provides *syntactic sugar* (i.e. making it easier to type, otherwise functionally equivalent) for function calls and object construction, particularly a replacement for `React.createElement()`. It may look like a template engine or HTML, but it isn't. JSX produces React elements while allowing you to harness the full power of JavaScript.

JSX is a great way to write React components. Its benefits include the following:

- *Improved developer experience (DX)*—Code is easier to read because it's more eloquent, thanks to an XML-like syntax that's better at representing nested declarative structures.
- *More-productive team members*—Casual developers (such as designers) can modify code more easily, because JSX looks like HTML, which is already familiar to them.
- *Fewer syntax errors*—Developers have less code to type, which means they make fewer mistakes.

Although JSX isn't required for React, it fits in nicely and is highly recommended by us and React's creators. You will have a hard time finding any team in the real world that uses React but not JSX. While we cannot say that *all* recent React projects in the world use JSX, we're pretty confident that *almost all* do.

3.1.1 Before and after JSX

To demonstrate the eloquence of JSX, this is the snippet required to create an element with a few custom components followed by a link:

```
const element = <main>
  <Title>Welcome</Title>
  <Carousel images={6} />
  <a href="/blog">Go to the blog</a>
</main>;
```

That's identical to the following snippet implemented without the benefit of JSX:

```
const element = React.createElement(
  'main',
  null,
  React.createElement(Title, null, 'Welcome'),
  React.createElement(Carousel, {images: 6}),
  React.createElement('a', {href: "/blog"}, 'Go to the blog'),
);
```

We can probably all agree that the JSX version is much easier to understand at a glance. It looks like HTML, which is very easy to read, and it's partially identical to the HTML output that will be rendered, except for the custom components of course.

3.1.2 Keeping HTML and JavaScript together

In essence, JSX is a small language with an XML-like syntax. It has changed the way people write UI components. Previously, developers wrote HTML—and JS code for controllers and views—in an MVC-like manner, jumping between various files. That stemmed from the separation of concerns in the early days. This approach served the web well when it consisted of static HTML, a little CSS, and a tiny bit of JS to make text blink.

This is no longer the case; today, we build highly interactive UIs, and JS and HTML are tightly coupled to implement various pieces of functionality. This violates the principle of separation of concern, which is a fundamental principle sought after in most software development. This principle is about separating unrelated items, but keeping related items together. If you seek to obey this principle, you should break your code down in such a way that every bit in isolation performs one and only one concern, and these "bits" can then be used in different connections. If you split your template and your view logic, but they only work if combined, then you have needlessly separate two items that belong together.

React fixes this invalidated principle by bringing together the description of the UI and the JS logic; and with JSX, the code looks like HTML and is easier to read and write. If for no other reason, we would use React and JSX just for this new approach to writing UIs.

JSX is compiled by various transformers (tools) into standard ECMAScript (see figure 3.1). You probably know that JavaScript is ECMAScript, too; but JSX isn't part of the specification, and it doesn't have any defined semantics. That means, that if you tried to compile JavaScript with embedded JSX in a normal JavaScript compiler without transpiling the JSX first, you would get errors. JSX is not valid JavaScript on its own and cannot be compiled directly by a JavaScript compiler.

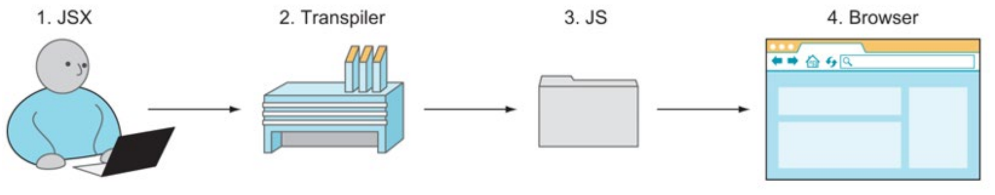


Figure 3.1 JSX is transpiled into regular JavaScript.

NOTE We call it *transpiling* rather than *compiling*, because we translate it from one source language (JSX) into another source language (JavaScript). The resulting JavaScript will then in turn be interpreted by a "real" compiler that actually runs the code. Transpiling is merely a conversion of syntax rather than interpreting the code.

When your browser executes your React application, your browser will only see the `React.createElement` statements required to generate the structure that you need. It is only in the editor that the JSX exists. The transpiler converts your files with JSX in them to pure JavaScript with `React.createElement()`'s all over the place to save you the trouble.

You may wonder, why you should bother with JSX at all. Considering how counterintuitive JSX code looks to begin with for new developers, it's no surprise that a few developers are turned off by this amazing technology. As an example, this bit of JavaScript has JSX in the middle of it, mixing in angle brackets where they normally would never exist:

```
const title = <h1>Hello</h1>;
```

What makes JSX amazing are the shortcuts to `React.createElement(NAME, ...)`. Instead of writing that function call over and over, you can instead use `<NAME />`. And as mentioned earlier, the less you type, the fewer mistakes you make. With JSX, DX is the only concern making it easier for developers to create components and applications faster and with fewer errors.

The main reason to use JSX is that many people find code with angle brackets (<>) easier to read than code with a lot of `React.createElement()` statements (even when they're aliased). And once you get into the habit of thinking about `<NAME />` not as XML, but as an alias to JavaScript code, you'll get over the perceived weirdness of JSX syntax. Knowing and using JSX can make a big difference when you're developing React components and, subsequently, React-powered applications.

Alternative shortcuts

To be fair, there were some alternatives to JSX when React was originally introduced to avoid typing verbose `React.createElement()` calls. One of them was to use the alias `React.DOM.*`. For example:

```
React.DOM.h1(null, 'Hey')
```

However, the whole `React.DOM` object is now deprecated and does not exist in React anymore since React 16, so you would have to import it from another package called `'react-dom-factories'`.

There's another alternative that is still possible today and still comes recommended by the official React documentation for situations where JSX is impractical (for example, when there's no build process). You can use a short variable as an alias for `React.createElement`. For example, you can create a variable `E` as follows:

```
const E = React.createElement;
return E('h1', null, 'Hey world?');
```

As mentioned earlier, JSX needs to be transpiled into regular JavaScript before browsers can execute the code. In most setups, you will never have to actually worry about this, but we'll discuss some transpilers in section 3.3 if you really need to do it on your own. For now we will dig in to fully understand JSX.

3.2 Understanding JSX

Let's explore how to work with JSX. You can read this section and keep it bookmarked for your reference, or (if you prefer to have some of the code examples running on your computer) start working on the examples using the CRA templates listed throughout. With CRA you get JSX transpiling "for free", so you don't have to worry about setting it up yourself.

3.2.1 Creating elements with JSX

Creating React elements with JSX is straightforward. See table 3.1 for some examples of the JavaScript that you have previously used and its JSX equivalent.

Table 3.1. JavaScript code versus JSX

JavaScript	JSX equivalent
`React.createElement('h1')`	`<h1 />`

`React.createElement(`
`  'h1',`
`  null,`
`  'Welcome',`
`);`
	`<h1>`
	`  Welcome`
	`</h1>`

`React.createElement(`
`  Title,`
`  null,`
`  'Welcome',`
`);`
	`<Title>`
	`  Welcome`
	`</Title>`

`React.createElement(`
`  Title,`
`  {size: 6},`
`  'Welcome'`
`);`
	`<Title size="6">`
	`  Welcome`
	`</Title>`

`React.createElement(`
`  Title,`
`  {size: 6},`
`  'Welcome to ',`
`  React.createElement(`
`    'strong',`
`    null,`
`    'Narnia',`
`  ),`
`);`
	`<Title size="6">`
	`  Welcome to`
	`  <strong>Narnia</strong>`
	`</Title>`

In the JSX code, the attributes and their values (e.g. `size={6}`) come from the second argument of `createElement()`. We'll focus on working with properties later in this chapter.

For now, let's look at an example of JSX elements without properties. Here is one of our early examples from the last chapter, upgraded to the recommended structure using a custom App component. It's just an `h1` element with the text "Hello world!" where the word "world" is in italic in listing 3.1.

Listing 3.1 Emphasized greeting without JSX

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return React.createElement(
      'h1',
      null,
      'Hello ',
      React.createElement('em', null, 'world'),
      '!',
     );
  }
}
export default App;
```

Implementing this with JSX is so much simpler as can be seen in listing 3.2.

Listing 3.2 Emphasized greeting with JSX

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return <h1>Hello <em>World</em>!</h1>;
  }
}
export default App;
```

You can even store objects created with JSX syntax in variables, because JSX is just a syntactic improvement of `React.createElement()`. This example stores the reference to the generated element in a variable before returning it:

```
const title = <h1>Hello <em>World</em>!</h1>;
return title;
```

This is completely identical to listing 3.2, it just uses an extra variable before returning.

3.2.2 Using JSX with custom components

The previous example used the `<h1>` JSX tag, which is also a standard HTML tag name. When working with custom components, you apply the same syntax. The only difference is that the component class name must start with a capital letter, as in `<Title />`.

Here's a more advanced iteration of our three link application from chapter 2, rewritten in JSX. In this case, you create a new component class and use JSX to create an element from it. Remember our Link example from the last chapter? The code looked like listing 3.3 without JSX (converted to the recommended App structure).

Listing 3.3 Three identical links without JSX

```
import React, { Component, Fragment } from 'react';
class Link extends Component {
  render() {
    return React.createElement(
      'p',
      null,
      React.createElement(
        'a',
        {href: '//reactjs.org'},
        'Read more about React',
      ),
    );
  }
}
class App extends Component {
  render() {
    const link1 = React.createElement(Link);
    const link2 = React.createElement(Link);
    const link3 = React.createElement(Link);
    const group = React.createElement(Fragment, null, link1, link2, link3);
    return group;
  }
}
export default App;
```

Using JSX, this now becomes listing 3.4.

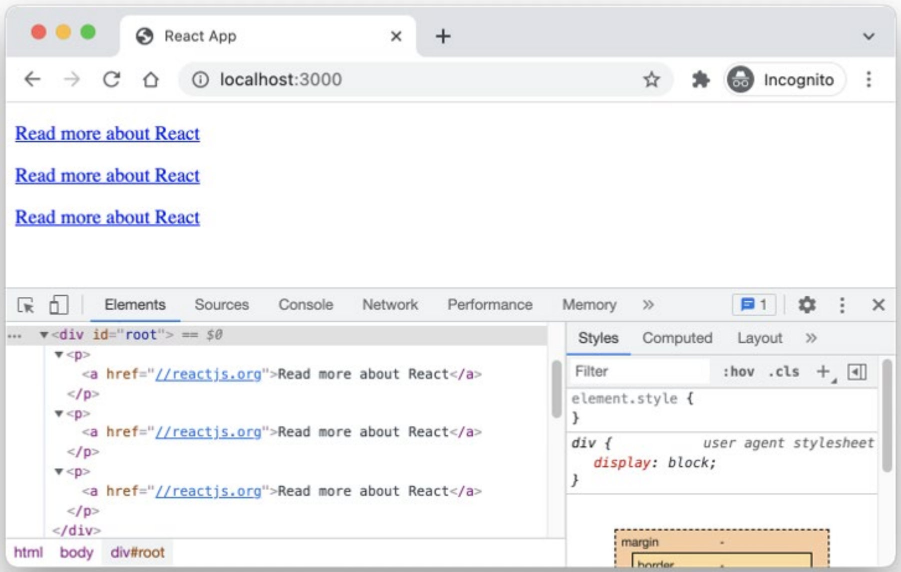
Listing 3.4 Three identical links with JSX

```
import React, { Component, Fragment } from 'react';
class Link extends Component {
  render() {
    return (
      <p>
        <a href="//reactjs.org">Read more about React</a>
      </p>
    );
  }
}
class App extends Component {
  render() {
    return (    #A
      <Fragment>    #B
        <Link />    #C
        <Link />    #C
        <Link />    #C
      </Fragment>
    );    #D
  }
}
export default App;
```

#A Opening parenthesis that starts the returned multiline JSX expression
#B React fragments are elements just like any other and can be rendered using JSX
#C Three identical instances of the Link component
#D Closing parenthesis that completes the returned multiline JSX expression

```
rq03-jsx-links
```

If you run this in the browser, you get the exact same result as we did in figure 2.13, which we have included again in figure 3.2.



Read more about React

Read more about React

Read more about React

```
▼<div id="root"> == $0
  ▼<p>
     <a href="//reactjs.org">Read more about React</a>
   </p>
  ▼<p>
     <a href="//reactjs.org">Read more about React</a>
   </p>
  ▼<p>
     <a href="//reactjs.org">Read more about React</a>
   </p>
 </div>
```

```
element.style {
}
div {                    user agent stylesheet
  display: block;
}
```

Figure 3.2 Three identical links in our application but now written using JSX

3.2.3 Multiline JSX objects

You might have noticed the parentheses around the returned multiline JSX object in listing 3.4. You have to include these parentheses if you start a multiline JSX object on a separate line after e.g. a `return`. This is the way to create multiline JSX objects when not starting on the same line:

```
return (
  <main>
    <h1>Hello world</h1>
  </main>
);
```

Alternatively, you can start your root element on the same line as `return` and avoid the parentheses. For example, this is valid as well:

```
return <main>
  <h1>Hello world</h1>
</main>;
```

A downside of this second approach is the reduced visibility of the opening `<main>` tag. It may be easy to miss in the code. The choice is up to you. We will exclusively use the former style of using parentheses around multiline JSX content for consistency.

Note that the exact same thing goes for any other use of multiline JSX objects - for example when you save them in a variable. We will also be using parentheses there as in:

```
const message = (
  <main>
    <h1>Hello world</h1>
  </main>
);
```

3.2.4 Outputting variables in JSX

When you compose components, you want them to be smart enough to change the view based on some code. For example, it would be useful if a date component uses the current date and time, and not just a hardcoded value.

When working with JavaScript-only React, you have to use string template literals (i.e. backticks) to mix strings with variables–or, even worse, concatenation. For example, to use a variable in a string context in a `DateTimeNow` component without JSX, you would write this code:

```
class DateTimeNow extends React.Component {
  render() {
    const dateTimeNow = new Date().toLocaleString()
    return React.createElement(
      'span',
      null,
      `Current date and time is ${dateTimeNow}.`
    )
  }
}
```

In JSX you can use curly braces `{}` notation to output variables dynamically, which reduces code complexity substantially:

```
class DateTimeNow extends React.Component {
  render() {
    const dateTimeNow = new Date().toLocaleString()
    return <span>Current date and time is {dateTimeNow}.</span>
    )
  }
}
```

If you reference a variable, that is a React element (optionally itself created using JSX), you can directly insert that other bit of JSX in the current context:

```
const now = <date>{dateTimeNow}</date>;
const message = <p>Today is {now}</p>;
```

This is equivalent to directly inserting the element:

```
const message = <p>Today is <date>{dateTimeNow}</date></p>;
```

The inserted variables can also be properties, not just locally defined variables:

```
<p>Hello {this.props.userName}, today is {dateTimeNow}.</p>
```

You can also invoke methods of your component that you create yourself. That is a common practice to isolate bits of functionality:

Listing 3.5 ButtonList using a method

```
import React, { Component } from 'react';
class ButtonList extends Component {
  getButton(text) {    #A
    return <button disabled={this.props.disabled}>{text}</button>;    #B
  }
  render() {
    return (
      <aside>
        {this.getButton('Up')}    #C
        {this.getButton('Down')}    #C
      </aside>
    );
  }
}
export default App;
```

#A Here we define a method getButton that takes an argument text, which will be the label on the button
#B Our button depends on another prop passed to our component
#C Here we invoke our method to get a button inserted with the proper text

The example in listing 3.5 is of course overly simplified. Most of the time you would probably be using an extra component for such a use case, but there are situations where component methods do come in handy. The purpose of this example is to show that you can invoke component methods directly in JSX.

You can in fact execute arbitrary JavaScript expressions inside the curly braces. For example, you can format a date directly:

```
<p>Today is {new Date(Date.now()).toLocaleTimeString()}.</p>
```

Now let's rewrite our emphasized greeting to store the italicized word in a variable first, before outputting it:

Listing 3.6 Emphasized greeting using JSX and a variable

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    const world = <em>World</em>;
    return <h1>Hello {world}!</h1>;
  }
}
export default App;
```

Next, let's discuss how you work with properties in JSX.

3.2.5 Working with properties in JSX

We touched on this topic earlier, when we introduced JSX. Element properties are defined using attribute syntax. That is, you use `key1=value1 key2=value2…` notation inside of the JSX tag to define both HTML attributes and React component properties. This is similar to attribute syntax in HTML/XML.

In other words, if you need to pass properties, write them in JSX as you would in normal HTML. You render standard HTML attributes by setting element properties (discussed in section 2.3) on a React element with an HTML tag. For example, this code sets a standard HTML attribute `href` for the anchor element `<a>`:

```
return <a href="//reactjs.org">Let's do React!</a>;
```

You use the exact same method to set properties on custom components. If we had our Link component from the last chapter, we could use it in JSX as:

```
return <Link url="//reactjs.org" framework="React" />;
```

Using hardcoded values for attributes isn't all that flexible of course. If you want to reuse the link component, then the `href` must change to reflect a different address each time. This is called dynamically setting values versus hardcoding them. So, next we'll go a step further and consider a component that can use dynamically generated values for attributes. Those values can come from component properties (`this.props`). After that, everything's easy. All you need to do is use curly braces (`{}`) inside angle braces (`<>`) to pass dynamic values of properties to elements.

For example, suppose you're building a component that will be used to link to user accounts. You need some attributes on your `<a>` tag but `href` and `title` must be different for each component and not hardcoded. Let's create a dynamic component `ProfileLink` that renders a link `<a>` using the properties `url` and `label` for `href` and `title`, respectively. You pass the properties to `<a>` using `{}`:

```
class ProfileLink extends React.Component {
  render() {
    return (
      <a
        href={this.props.url}
        title={this.props.label}
        target="_blank">Profile
      </a>
    );
  }
}
```

Where do the property values come from? They're defined when the `ProfileLink` is created—that is, in the component that creates `ProfileLink`, a.k.a. its parent. For example, this is how the values for `url` and `label` are passed when a `ProfileLink` instance is created, which results in the render of the `<a>` tag with those values:

```
<ProfileLink url="/users/johnny" label="Profile for Johnny" />
```

From the previous chapter, you should remember that when rendering standard elements (`<h>`, `<p>`, `<div>`, `<a>`, and so on), React will render any and all properties even if they don't have any semantic meaning in HTML. That's not specific for JSX, that's default React behavior.

If you have an object of properties, that you want to render on an element, you can render them all one by one as follows:

```
return (
  <Post
    id={post.id}
    title={post.title}
    content={post.content}
  />
);
```

This works great and is a safe solution. However, if you have an object with values, and you want to render *all of them*, you can do so using the spread operator as follows:

```
return <Post {...post} />;
```

Note that this will render *every* property of the post object, regardless of whether that makes sense or not. Only use this process, when you are sure that the object only has the properties that you need or at least sure that any excess properties are ignored.

This will even allow you to render all the properties passed to a component to another element inside that component by spreading `this.props`:

```
return <input value={this.value} {...this.props} />;
```

This is a bit dangerous though, as it allows the parent component to pass in arbitrary values that would supersede any values that you passed to it. If `this.props` contained a `value` property, it would override the `value` property that you set in the component before the spread. Be extra careful when spreading objects and in particular when spreading all props passed to a component.

We will get back to the spreading operator in the next chapter and cover some other common examples of its use.

THE SPECIAL PROPERTY: CHILDREN

If you remember back to the previous chapter, we introduced the special property `children`, which only looks like a property inside a custom component, not from the outside.

When using JSX, the `children` property becomes a lot neater to use. If you remember back to the example we had with child nodes in chapter 2, it looked like this tree structure in figure 3.3.

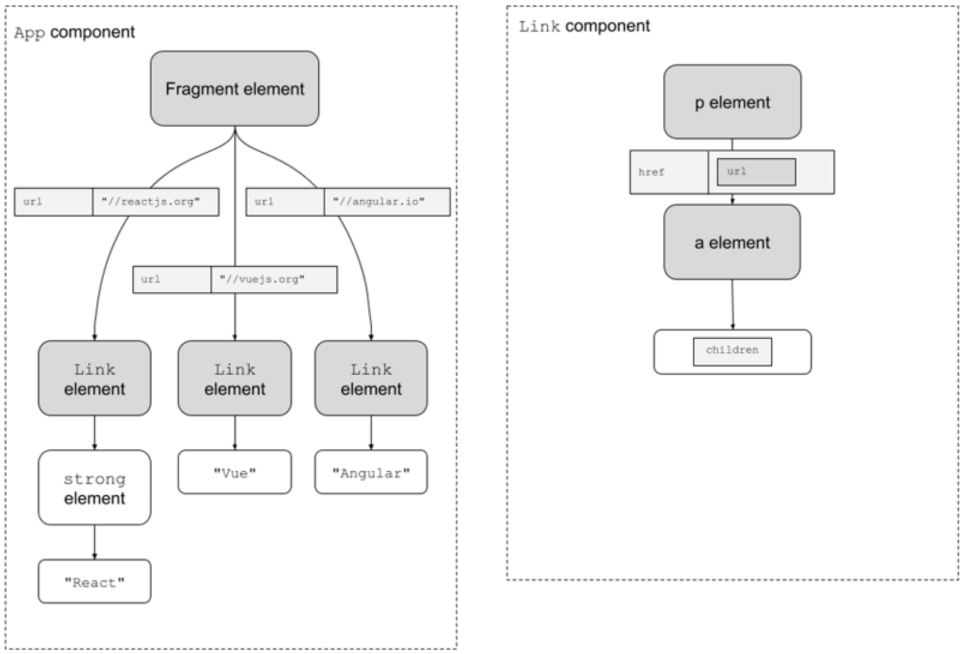


Figure 3.3 The component tree when we use child nodes as link content

Let's reimplement this one in JSX. We know all the things we need to do, so let's just go ahead and do it in listing 3.7.

Listing 3.7 Link list with child nodes in JSX

```
import React, { Component } from 'react';
class Link extends Component {
  render() {
    return (
      <a href={this.props.url}>    #A
        {this.props.children}    #B
      </aside>
    );
  }
}
class App extends Component {
  render() {
    <Fragment>
      <Link url="//reactjs.org">
        <strong>React</strong>    #C
      </Link>
      <Link url="//vuejs.org">
        Vue    #C
      </Link>
      <Link url="//angular.io">
        Angular    #C
      </Link>
    </Fragment>
  }
}
export default App;
```

#A We still use the url property as we did before
#B And we also just use this.props.children as if it was any other property
#C Note how elegantly these child nodes are added in JSX. It looks just like the rest of the code.

```
 rq03-children
```

The difference between using properties and child nodes suddenly becomes a lot more obvious. We could have passed the link content in as a property, but it would have looked pretty bad. If we used the regular property approach, it would have looked like this:

```
<Link
  url="//reactjs.org"
  content={<strong>React</strong>}
/>
```

But when we use the children approach it becomes:

```
<Link url="//reactjs.org">
  <strong>React</strong>
</Link>
```

We definitely know what we prefer—the latter approach.

3.2.6 Branching in JSX

Branching is always important in coding. If something, do this, otherwise do this. Because JSX is just JavaScript, we can basically use the exact same constructs that we do in regular coding to create branching in our components.

That being said, some patterns have emerged that most developers follow about how to use branching in React components using JSX.

- Use early return for rendering nothing
- Use the ternary operator for rendering alternative elements
- Use logical and for rendering optional elements
- Use object maps for rendering between many different elements
- Use extra components for more complex branching

We will go through each of these in the next subsections to explain how we use branching in JSX and custom React components.

USE EARLY RETURN FOR RENDERING NOTHING

Imagine that you have a component that only renders something relevant when a certain condition is true. For example, imagine a countdown component that only renders a value when the number of remaining seconds is larger than 0.

If a component doesn't render anything, we can simply return `null` from said component. However, to optimize our components, we try to do this as early as possible - basically to short-circuit the execution. The purpose of this is to branch out to the easiest case as quickly as we can, to avoid doing extra calculations or creating JSX objects where we don't need them.

We could create our Countdown component like this:

```
class Countdown extends Component {
  render() {
    const seconds = this.props.remaining % 60;
    const minutes = Math.floor(this.props.remaining / 60);
    const message = <p>{minutes}:{seconds}</p>;
    if (seconds > 0 || minutes > 0) {
      return message;
    } else {
      return null;
    }
  }
}
```

There's nothing inherently wrong with this. It works and it's fully functional. But you will see many developers use the approach of aborting early if the component renders nothing. We can detect this case of rendering nothing before calculating the number of seconds and minutes and before creating the JSX object:

```
class Countdown extends Component {
  render() {
    if (this.props.remaining === 0) {
      return null;
    }
    const seconds = this.props.remaining % 60;
    const minutes = Math.floor(this.props.remaining / 60);
    return <p>{minutes}:{seconds}</p>;
  }
}
```

Here we also use the fact that when we return from inside an `if` block, we don't actually need an `else` block. The else is implicit in that anything after the `if` block is only visited if the condition failed.

USE TERNARY FOR ALTERNATIVES

Another very common case in React components is to render different elements based on whether some condition is true or false.

Let's for instance imagine a shopping cart. If there are any items, display the items, but if there are no items, display a message saying just that.

We could do this in JSX by using a variable and assigning it different values using a regular if/else statement block. However, that's a bit lengthy and it's a lot more common in React to use the ternary operator. The ternary operator is unlike the if/else statement an expression and can thus be used inline directly in JSX:

```
<p>User is {this.props.isOnline ? 'Online' : 'Offline'}<p>
```

Using this, we can create our shopping cart component from before:

```
class ShoppingCart extends Component {
  render() {
    return (
      <aside>
        <h1>Shopping cart</h1>
        {this.props.items.length === 0 ? (
          <p>Your cart is empty. Go buy something!</p>
        ) : (
          <CartItems items={this.props.items} />
        )}
      </aside>
    );
  }
}
```

USE LOGICAL OPERATORS FOR OPTIONALS

Another common pattern is the need to optionally render an element only if a condition is true, but render nothing if not.

As an example, we want to display a little checkmark next to a user name if the user is a verified user, but nothing for the unverified plebeians. We can do this using *logical and* and the fact that logical operators "short-circuit" by returning as soon as the truthiness of the entire expression is known. So, when doing `a && b`, JavaScript returns a if a is *falsy* or b if a is *truthy*. If a is truthy, it doesn't actually matter what b is. It will be returned regardless. Combine this with the fact that React renders false as the empty string (more on that latter).

Truthiness

In JavaScript, a *truthy* value translates to true when evaluated as a Boolean. For example, in an if statement:

```
if (someVariable) {
  // this happens if and only if someVariable is truthy.
}
```

The value is *truthy* if it's not *falsy*. That is literally the official definition, not kidding.

There are only six *falsy* values:

- false
- 0
- "" (empty string)
- null
- Undefined
- NaN (not a number)

We can use this to render conditional elements, by making our logical and expression return false if the user is not verified, and a React element if the user is verified:

```
class UserName extends Component {
  render() {
    return (
      <p>
        {this.props.username}
        {this.props.isVerified && <Checkmark />}
      </p>
    );
  }
}
```

You will encounter this pattern very often in React components so it's a good one to know.

USE OBJECTS FOR SWITCHING

So far we've dealt with the case of rendering either an element or nothing, or rendering one element or another, but what if we want to render more than two types of elements based on a condition?

For this scenario, we want to render an icon based on some blog post status. If the post is in the draft state, we render a draft icon. If the post is in the published state, we render a published icon. And if the post is in the deleted state, we render a trash icon.

Well, we could nest ternaries, to first check if `status === "draft"`, then if not check if `status === "published"`, and if not, assume that it must be deleted:

```
class PostStatus extends Component {
  render() {
    return this.props.status === 'draft ?
      <DraftIcon /> :
      this.props.status === 'published' ?
      <PublishedIcon /> :
      <TrashIcon />;
  }
}
```

This would work. It's not very pretty though. Another alternative is to use a `switch` statement and simply return the different values in each case. But a more declarative approach here is to use an object with properties for the different cases resolving the different outcomes:

```
const status2icon = {
  draft: <DraftIcon />,
  published: <PublishedIcon />,
  deleted: <TrashIcon />,
};
class PostStatus extends Component {
  render() {
    return status2icon[this.props.status];
  }
}
```

That's actually rather short and neat, no?

However do realize, that this doesn't handle the case, where the status is none of those things. Before, the component would render the trash icon in case the status was neither draft nor published, but now it will only render the trash icon if the status is deleted.

If we want to handle the case of the status being any other unexpected value, we need to add a *logical or* at the end so in case the object indexing resolves to nothing, we still render an alternative. Let's say we just render the trash icon in any unknown case:

```
class PostStatus extends Component {
  render() {
    return status2icon[this.props.status] || status2icon.deleted;
  }
}
```

This pattern is probably less common in React, but you will still see it for simple cases as the above.

USE EXTRA COMPONENTS FOR COMPLEX BRANCHING

The above scenarios only cover some pretty simple branching cases. What do you do if your component has more complicated logic than that?

Let's say we have a shopping cart component as before and we have some buttons at the bottom. We have to implement the following business logic as dictated by some customer.

- If the user is logged in, there will be just a *"checkout"* button.
- If the user is not logged in, there will be a *"login"* button as well as a *"checkout as guest"* button.
- If any item is out of stock or if the cart is empty, the *"checkout"* or the *"checkout as guest"* button will be disabled.
- If the user is logged in but has not added a credit card yet, instead show an *"add credit card"* button.
- If the user logged in, has a credit card, and also has entered an address, show a *"one-click buy"* button next to the *"checkout"* button. This button will be disabled according to the same logic as the *"checkout"* button.

Well, let's implement all this with all the tricks that we have learned so far in listing 3.8.

Listing 3.8 Complex shopping cart

```
class ShoppingCart extends Component {
  render() {
    const hasItems = this.props.items.length > 0;
    const isLoggedIn = this.props.user !== null;
    const hasCreditCard = isLoggedIn && this.props.user.creditcard !== null;
    const hasAddress = isLoggedIn && this.props.user.address !== null;
    const isAvailable = this.props.items.every(item => !item.outOfStock);
    return isLoggedIn ? (    #A
      hasCreditCard ? (    #B
        <Fragment>
          <button disabled={!hasItems || !isAvailable}>    #C
            Checkout
          </button>
          {hasAddress && (
            <button disabled={!hasItems || !isAvailable}>    #D
              One-click buy
            </button>
          )}
        </Fragment>
      ) : (
        <button>Add credit card</button>
      )
    ) : (
      <Fragment>
        <button>Login</button>
        <button disabled={!hasItems || !isAvailable}>    #C
          Checkout as guest
        </button>
      </Fragment>
    );
  }
}
```

#A First ternary operator
#B Second ternary operator
#C Repeated logic for disabled button
#D Logical and to optionally render a button

```
rq03-cart-single
```

Okay, that seems to cover everything. However, this is getting a bit complicated with the nested conditionals and duplicated attributes. For such a complex case, it is often a good idea to split things into multiple components that deal with each of the different cases one by one.

Here we can create new components `<UserButtons />` and `<GuestButtons />` and at the top level just select which of these components to use, and then inside each of these add the necessary extra checks and conditionals.

So, let's do just that in listing 3.9.

Listing 3.9 Simplified multi-component shopping cart

```
class UserButtons extends Component {
  render() {
    const hasCreditCard = this.props.user.creditcard !== null;
    const hasAddress = this.props.user.address !== null;
    const disabled = !this.props.canCheckout;
    return hasCreditCard ? (    #A
      <Fragment>
        <button disabled={disabled}>Checkout</button>
        {hasAddress && <button disabled={disabled}>One-click buy</button>}    #B
      </Fragment>
    ) : (
      <button>Add credit card</button>
    );
  }
}
class GuestButtons extends Component {
  render() {
    return (
      <Fragment>
        <button>Login</button>
        <button disabled={!this.props.canCheckout}>
          Checkout as guest
        </button>
      </Fragment>
    );
  }
}
class ShoppingCart extends Component {
  render() {
    const hasItems = this.props.items.length > 0;
    const isLoggedIn = this.props.user !== null;
    const isAvailable = this.props.items.every(item => !item.outOfStock);
    const canCheckout = hasItems && isAvailable;
    return isLoggedIn ? (    #A
      <UserButtons user={this.props.user} canCheckout={canCheckout} />
    ) : (
      <GuestButtons canCheckout={canCheckout}( />
    );
  }
}
```

#A Ternary operators
#B Logical and for optional rendering

```
rq03-cart-multi
```

This works exactly the same as before and has exactly the same complexity as before. But each component is much simpler than before and you can easily understand each component on its own by reading through the code. You could even take it an extra step and split the `<UserButtons>` component into two as well for the "has credit card" and "doesn't have credit card" situations.

We must of course acknowledge that more components means more code and more code means more memory and CPU usage (in general), so this latter example is slightly more resource intensive than the former. In most applications, this difference is negligible though and code quality often trumps such minor optimizations.

3.2.7 Comments in JSX

Because JSX is written inside of JavaScript, you can use regular JavaScript comments outside the JSX elements as normal:

```
// This is the page title
const title = <h1>Hello world!</h1>;
```

However, if you have very long segments of JSX code, you might want to add comments inline inside the JSX. If you want to do that, you can't always use a regular JavaScript comment directly.

Comments in JSX work similar to comments in regular JavaScript. To add JSX comments between tags, you can wrap standard JavaScript comments using `/**/` or `//` in `{}`, like this:

```
const content = (
  <div>
    {/* Just like a JS comment */}
    {/* It can also span
        multiple lines */}
    {// Single line comments are possible too
    }
  </div>
);
```

Or, you can use JavaScript comments directly using either `/**/` or `//` inside tags:

```
const content = (
  <div>
    <input
      /* This element is
         rendered because... */
      name={this.props.name} // Some important comment here
    />
  </div>
);
```

Note that when you use a regular single-line comment between tags inside curly brackets, you need to have a newline before you end the curly brackets. The following code would fail:

```
const content = (
  <div>
    {// This does NOT work! }
  </div>
);
```

The above would result in a compiler error, because the ending curly brackets is considered part of the comment, so the opening curly bracket does not have a matching ending brackets, which causes a parsing error.

3.2.8 Lists of JSX objects

A common tactic in React elements is to map an array of elements to an array of JSX objects to be returned in a component.

Let's say we want to create a component to render a drop down. We want to pass the list of options in the dropdown as an array of strings to a new `<Select />` component.

Essentially we want to be able to do this in our application:

```
class App extends Component {
  render() {
    const items = ['apples', 'pears', 'playstations'];
    return <Select items={items} />;
  }
}
```

And then our `Select` component should correctly render a `<select>` with `<option>`'s in HTML. How would we go about doing that?

The best way to do it, is to simply map the elements from strings to JSX objects using declarative programming:

```
class Select extends Component {
  render() {
    return (
      <select>
        {this.props.items.map(item => <option>{item}</option>)}
      </select>
    )
  }
}
```

This is actually a pretty decent attempt at solving this. However, if we run this in the browser, we will get a warning:

```
Warning: Each child in a list should have a unique "key" prop.
```

The application works, but we get this warning about a missing `key` property. The usage of the key property is a bit advanced at this stage in our React learning, but in short it's used by React to track if the same element moves around in the rendered DOM. If the same element moves around, React will reuse the same element, but if React doesn't know about whether it's the same element or not, React will have to delete all the old elements and re-create completely new elements every time the list renders.

For the purposes of this example, we can just use the `item` value as the key property on the root element returned inside the mapped array. This results in the code in listing 3.10.

Listing 3.10 Correct implementation of select

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    const items = ['apples', 'pears', 'playstations'];
    return <Select items={items} />;
  }
}
class Select extends Component {
  render() {
    return (
      <select>
        {this.props.items.map(item => (
          <option key={item}>    #A
            {item}
          </option>
        ))}
      </select>
    )
  }
}
export default App;
```

#A We have added a key property to the <option> element

```
rq03-correct-select
```

This key property is an internal React property that will never be rendered to the DOM. It is recommended that the key property is some unique identifier for the element in question and not just the index of the element in the array (if elements move around in the array, the indexes change even though the elements don't, so proper element reuse is circumvented).

Keys must be unique. If you render a list with non-unique keys, you will get a different warning in the console about duplicate keys.

> **NOTE** **Keys are local to the individual array, so they only have to be unique within each array, not between all arrays in your application or even your component. Different arrays of JSX objects can have duplicate keys between them as long as no single array has duplicate keys inside it.**

As mentioned, this is a fairly complicated feature of React to understand at this point, so for now just be aware that if you get a warning in the console about a missing key property or duplicate keys, this is the reason.

3.2.9 Fragments in JSX

We already covered JSX fragments a few times. They're used to export multiple elements at the same level in a situation, where only a single element is allowed.

We have previously done something like this to include both a heading and a link:

```
import { Fragment } from 'react';
...
return (
  <Fragment>
    <h1>Hello and welcome</h1>
    <a href="/blog">Go to the blog</a>
  </Fragment>
);
```

However, from React 16.2 (and Babel 7) and forward, a shorter syntax is also allowed. Now you don't even have to import the `Fragment` component:

```
return (
  <>
    <h1>Hello and welcome</h1>
    <a href="/blog">Go to the blog</a>
  </>
);
```

This new shorthand syntax uses a seemingly empty tag to render fragments.

This syntax with `<></>` cannot take any attributes or properties however. The only property you might want to apply to this would be a key, because you are rendering a list of elements, where each element has more than a single JSX element at the root.

A classic scenario for this is a definition list. It's defined in HTML like this:

```
<dl>
  <dt>Term A</dt>
  <dd>Description of Term A.</dd>
  <dt>Term B</dt>
  <dd>Description of Term B.</dd>
</dl>
```

As you can see, each entry requires two sibling elements in the list to render (`<dt>` and `<dd>`).

If we create an application to render three dog breeds with a little description about each, we need to map our dog breed names and definitions to two elements. We do that by wrapping them in a fragment, but because we need the fragment to have a key property, we have to use the literal `Fragment` component and we unfortunately can't use the shorthand syntax mentioned before. Let's implement this in listing 3.11.

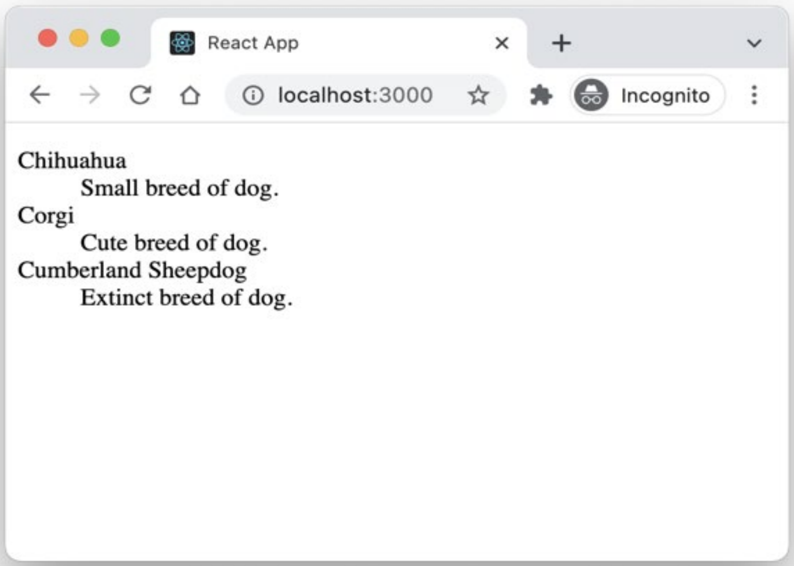
Listing 3.11 Definition list of dog breeds

```
import React, { Component, Fragment } from 'react';
class App extends Component {
  render() {
    const list = [
     {breed: 'Chihuahua', description: 'Small breed of dog.'},
     {breed: 'Corgi', description: 'Cute breed of dog.'},
     {breed: 'Cumberland Sheepdog', description: 'Extinct breed of dog.'},
    ];
    return <Breeds list={list} />;
  }
}
class Breeds extends Component {
  render() {
    return (
      <dl>
        {this.props.list.map(({ breed, description }) => (    #A
          <Fragment key={breed}>    #B
            <dt>{breed}</dt>
            <dd>{description}</dd>
          </Fragment>
        ))}
      </dl>
    )
  }
}
export default App;
```

#A Here we use destructuring to easily access the properties of the list item.
#B Because we need a key prop, we have to use the proper Fragment component. Note that we just use the breed as the key, as that uniquely identifies each element in the array.

```
rq03-dog-breeds
```

If we run this in the browser, we get a nice definition list exactly as we wanted to as you can see in figure 3.4



```
Chihuahua
        Small breed of dog.
Corgi
        Cute breed of dog.
Cumberland Sheepdog
        Extinct breed of dog.
```

Figure 3.4 A definition list of a curated subset of dog breeds and their primary description.

This situation where we use fragments with keys is not as edge-case as it might seem, so this is very useful to know already at this stage.

You've now had a taste of JSX and its benefits. The rest of this chapter is dedicated to JSX tools and potential traps to avoid. That's right: tools and gotchas.

Because before we can continue, you must understand that for any JSX project to function properly, JSX needs to be compiled. Browsers can't run JSX directly—they can run only JavaScript, so you need to take the JSX and transpile it to normal JS (refer back to figure 3.1). It's fortunately a lot less complicated than it sounds.

3.3 How to transpile JSX

For all the projects and examples in this book, you don't need to set up your own transpiler (or most other things in your technology stack). For most projects you will encounter out in the wild, you also don't need to worry about that yourself, as existing projects already have a working pipeline and new projects can be based on the best-practice documentation provided by whatever framework you desire to work with.

However, if you really wanted to set a React project up from scratch, you could do so. That would include setting up a JSX transpiler.

The most popular tool out there to do JSX transpiling is Babel, but there are alternatives that you might want to take a look at as well. Some of the alternatives can be part of a larger package of build tools that make your whole setup easier to maintain or simply be drop-in replacements for Babel. Alternatives include SWC, Sucrase, and ESBuild. Please consult their various documentation for details on how to use them.

We strongly recommend that you don't spend too much time creating a JSX transpiler setup yourself. The good folks at React have created CRA for us, that will serve almost all our needs in this book, and for the few examples and projects that go beyond CRA, there will still be a full transpiling build setup included.

3.4 React and JSX gotchas

This section covers some edge cases and oddities that you should be aware of when you use JSX:

- Self-closing tags are required for leaf nodes
- Special characters are written literally
- String conversion is a bit peculiar
- The style attribute is an object
- Some attributes have reserved names and must be renamed
- Multi-word attributes are camel-cased
- Boolean attributes are handled differently than in HTML
- Some white-space is collapsed (but not all)
- You can add `data-` attributes where desired

3.4.1 Self-closing elements

JSX requires you to have a closing slash (`/`) either in the closing tag or, if you don't have any children, at the end of that single tag. For example, this is correct:

```
<a href="//reactjs.org">React</a>
<img src="/logo.png" alt="Logo" />
```

The following is *not* correct, because both nodes are missing an end tag:

```
<a href="//reactjs.org">React
<img src="/logo.png" alt="Logo">
```

You might know that HTML is more fault-tolerant. Browsers will ignore the missing slash or end element and render the element just fine without it. Go ahead: try to create an HTML file with just `<button>Press me` and see for yourself that this renders the button just fine!

3.4.2 Special characters

HTML entities are codes that display special characters such as copyright symbols, em dashes, quotation marks, and so on. Here are some examples:

```
&copy;
&mdash;
&ldquo;
```

You can render those codes as any string in text content inside a node or in the as an attribute to a node. For example, this is static JSX (text defined in code without variables or properties):

```
<span>&copy;&mdash;&ldquo;</span>
<input value="&copy;&mdash;&ldquo;"/>
```

But if you want to dynamically output HTML entities (from a variable or a property), all you'll get is the direct output (that is, literally this text: `&copy;&mdash;&ldquo;`), not the special characters. Thus, the following code won't work:

```
// Anti-pattern. Will NOT work!
const specialChars = '&copy;&mdash;&ldquo;'
<span>{specialChars}</span>
<input value={specialChars}/>
```

React/JSX will auto-escape the dangerous HTML, which is convenient in terms of security (security by default rocks!). To output special characters, you need to use one of these approaches:

- Copy the special character directly into your source code. Just make sure you use a UTF-8 character set. This is the recommended method to deal with special characters.
- Escape the special character with `\u`, and use its Unicode number (use a website such as fileformat.info to look it up).
- Convert from a character code to a character number with `String .fromCharCode(charCodeNumber)`.
- Use the special property `dangerouslySetInnerHTML` to (dangerously) set the inner HTML (not recommended).

To illustrate the last approach (as a last resort—when all else fails on the Titanic, run for the boats!), look at this code:

```
const specialChars = '&copy;&mdash;&ldquo;';
<span dangerouslySetInnerHTML={{__html: specialChars}}/>
```

Obviously, the React team has a sense of humor to name a property `dangerouslySetInnerHTML`.

3.4.3 String conversion

When Reacts outputs the value of your variable (or your expression in general), what does it actually render it as? It can only render as one of two things. It's either a string, that becomes the string content between elements, or it's an element, that then just becomes an element as if it was rendered directly.

But how are "things" converted to a string, if they're not an element? Well, React is a bit peculiar here as it depends on the type of the expression that you're rendering.

Take a look at table 3.2 to understand the different possibilities for the different primitive values in JavaScript.

Table 3.2. React rendering different types

Type	Output
`"string"`	`"string"`
`""`	`""`
`3.4`	`"3.4"`
`0`	`"0"`
`NaN`	`"NaN"`
`Number.POSITIVE_INFINITY`	`"Infinity"`
`Number.NEGATIVE_INFINITY`	`"-Infinity"`
`true`	`"true"`
`false`	`""`
`undefined`	`""`
`null`	`""`

There's a few surprises there. Most importantly, *false* becomes the empty string, but *true* becomes `"true"`. So, 4 of the *falsy* values (empty string, `false`, `null`, and `undefined`) all become the empty string.

But what about 0, which is also *falsy*? Well, it becomes 0. It would be weird if you could not render a 0 in your components, so that's kind of necessary. And finally, *NaN* is also just `"NaN"`, and not the empty string. This is generally to help you debug your calculations better - if you see a *NaN*, you know you made an error somewhere, but if you just saw nothing, you might not find it as quickly.

This fact that false renders nothing, but 0 renders something especially matters when using *logical and* to render optional elements as we discussed earlier. You might be used to doing things like this in JavaScript:

```
if (items.length) {
  hasItems = true;
}
```

Here we just use `items.length` as the condition for our if statement because we know that 0 is *falsy* anyway, so we don't have to actually say `items.length > 0` — the truthiness of the statement is the same.

You shouldn't do that in JSX though. Let's say, that you want to render a *"Checkout"* button in your shopping cart if you have at least one item in it, but just nothing in case of no items:

```
class ShoppingCart extends Component {
  render() {
    return (
      <aside>
        <h1>Shopping cart</h1>
        <CartItems items={this.props.items} />
        {this.props.items.length && <button>Checkout</button>}    #A
      </aside>
    );
  }
}
```

#A Don't do this - using array length as a condition directly in a logical and expression leads to problems

This works as long as there are more than 0 items in the cart. But what happens where there are 0 items? The *logical and* expression highlight in annotation #A above short-circuits and returns the first *falsy* value as it is. And because the length of the array is 0, the resulting value of the expression is suddenly 0. And 0 renders as "0" in the document as you can see in Table 3.2.

So if you did the above, your empty shopping cart would suddenly display a "0" in the bottom of the component to the utter confusion of everyone.

To implement this correctly, always compare the length of the array to be greater than 0 to ensure the type is boolean. Or even better, store that comparison in another variable making the code even simpler to read:

```
class ShoppingCart extends Component {
  render() {
    const hasItems = this.props.items.length > 0;    #A
    return (
      <aside>
        <h1>Shopping cart</h1>
        <CartItems items={this.props.items} />
        {hasItems && <button>Checkout</button>}    #B
      </aside>
    );
  }
}
```

#A Store the comparison in variable guaranteed to be of type boolean
#B Then use that variable to conditionally render your optional element

It's actually not that uncommon to forget this when you're developing, so spotting rogue 0's throughout your application is definitely possible. And they are almost always the result of this type of expression.

3.4.4 style attribute

The style attribute in JSX works differently than in plain HTML. With JSX, instead of a string, you need to pass a JavaScript object, and CSS properties need to be in *camelCase*. For example:

- `background-image` becomes `backgroundImage`.
- `font-size` becomes `fontSize`.
- `font-family` becomes `fontFamily`.

You can save the JavaScript object in a variable or render it inline with double curly braces (`{{...}}`). The double braces are needed because one set is for JSX and the other is for the JavaScript object literal.

Suppose you have an object with this font size:

```
const smallFontSize = { fontSize: '10pt' };
```

In your JSX, you can use the `smallFontSize` object:

```
<input style={smallFontSize} />
```

Or you can settle for a larger font (30 point) by passing the values directly without an extra variable:

```
<input style={{ fontSize: '30pt' }} />
```

Let's look at another example of passing styles directly. This time, you're setting a red border on a `<span>`:

```
<span style={{
  borderWidth: '1px',
  borderStyle: 'solid',
  borderColor: 'red',
}}>Red velvet cake is delicious</span>
```

Alternatively, the following border value will also work and do the exact same thing:

```
<span style={{border: '1px red solid'}}>Hey</span>
```

The main reason styles are not opaque strings but JavaScript objects is so that React can work with them more quickly when it applies changes to views.

3.4.5 Reserved names: class and for

React (and JSX) accepts any attribute that's a standard HTML attribute, except `class` and `for`. Those names are reserved words in JavaScript/ECMAScript (for creating classes and for-loops respectively), and JSX is converted into regular JavaScript. So just like you can't create a variable named `for` or any other reserved word, you can't create attributes with these names (not directly anyway).

Use `className` and `htmlFor` instead respectively. For example, if you want to apply a class name of "hidden" to an element, you have to do use the attribute `className`:

```
<p className="hidden">...</p>
```

If you need to create a label for a form element, use `htmlFor`:

```
<input type="checkbox" id={this.props.id} value="hasCorgi" />
<label htmlFor={this.props.id}>Corgi?</label>
```

Both of these are pretty easy to remember, as you will get compiler errors if you forget.

3.4.6 Multi-word attributes

In the same vein as the two reserved names mentioned in the previous section, some other HTML attributes are renamed in React. Some of them make sense, others less so.

Any attribute with more than one English word in it, is renamed to *camelCase* style naming. This makes sense for SVG attributes using a dash such as `clip-path` or `fill-opacity`. We can't use dashed attributes directly in JSX, so these are renamed to `clipPath` and `fillOpacity` respectively.

But the same goes for HTML attributes, that don't use a dash, but are all lowercase normally, which can be quite confusing. If you type the following in JSX, it doesn't work:

```
return <video autoplay>...</video>;
```

That's because while the attribute is called `autoplay` (and can be all lowercase in HTML), in React you have to use camel case and call it `autoPlay`. This can be a bit frustrating. This goes for a huge number of properties that you often use in HTML.

React will not warn you about skipping these properties, but merely filter them out silently. So you might never know that you typed it wrong until you realize, that your video isn't auto-playing (because of `autoPlay` rather than `autoplay`), your iframe doesn't allow fullscreen (because of `allowFullscreen` rather than `allowfullscreen`), or your input field doesn't have a maximum of characters allowed (because of `maxLength` rather than `maxlength`).

Appendix A has a full list of all attributes that have been renamed from HTML and SVG to their React equivalent.

3.4.7 Boolean attribute values

Some attributes (such as `disabled`, `required`, `checked`, `autoFocus`, and `readOnly`) are specific only to form elements. The most important thing to remember here is that the attribute value must be set as a JavaScript expression (that is, inside `{}`) and not set as a string.

For example, use `{false}` to enable the input:

```
<input disabled={false} />
```

But don't use a `"false"` value, because it'll pass the truthy check (a non-empty string is truthy in JavaScript as you hopefully remember from section 3.2.6). This is because the string `"false"` is not any of the 6 falsy values. It is in fact a non-empty string, which is truthy and results in the value true. React will render the input as disabled (`disabled` will be set to true):

```
<input disabled="false" /> // Don't do this!
```

If you omit a value after a property, React will set the value to true:

```
<input required />
```

This is equivalent to manually setting the value to true, so just do the above rather than use `required={true}`.

For many of these attributes in HTML, completely excluding the value means setting the value to false, so if you want to set a value specifically to true or false, simply include it without a value or omit it.

If you want to set a value based on the contents of a variable, use an expression:

```
<input readOnly={!isEditable} />
```

> **NOTE** **Notice the multi-word problem mentioned before. This boolean attribute in React is called** `readOnly` **and not** `readonly` **as you know it from HTML.**

CUSTOM COMPONENT WITH BOOLEAN PROPERTIES

The same thing is the case when you create your own components. If you have a custom component and want to accept a boolean property, you can just use a property from this.props as if it was a boolean, and React will make sure to set it to true, if specified when used.

We can for example create an alert component that will display an alert message to the user. This message is either an error or just a warning. To control the level of the alert, we add a boolean flag, isError, and if true, we include a warning sign emoji around the message.

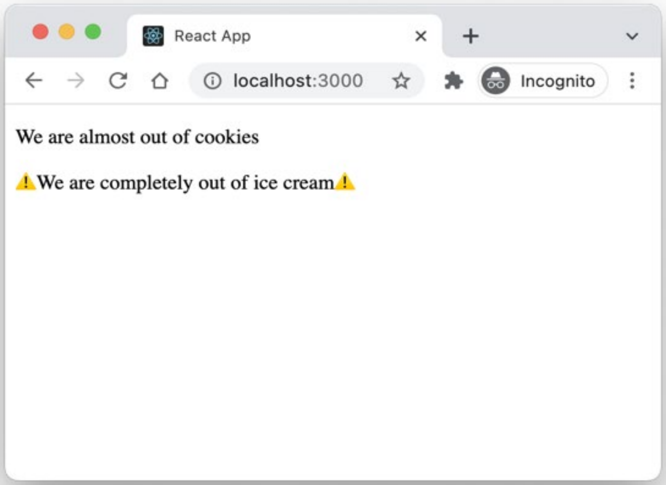
We will then use this component to display two different alerts in our application - one as an error, the other just as a warning. Let's do this in listing 3.12.

Listing 3.12 Passing and accepting boolean properties in JSX

```
import { Component } from 'react';
class Alert extends Component {
  render() {
    return (
      <p>
        {this.props.isError && '⚠'}
        {this.props.children}
        {this.props.isError && '⚠'}
      </p>
    );
  }
}
class App extends Component {
  render() {
    return (
      <main>
        <Alert>We are almost out of cookies</Alert>
        <Alert isError>We are completely out of ice cream</Alert>
      </main>
    );
  }
}
```

#A Some plain text

If we run this in the browser, we see how the two messages are correctly displayed as in figure 3.5.



We are almost out of cookies

⚠We are completely out of ice cream⚠

Figure 3.5 The first message is just a warning, but the second is definitely an error.

3.4.8 White space

If you want to add white space between components—e.g. if you're adding a bold word inside a sentence—you have to be very careful about how you place your newlines.

Let's say you want to write a headline with an emphasized word in the middle, for example, "All corgis are awesome" but "corgis" must be in italics. You could do it in JSX in this way as in listing 3.13.

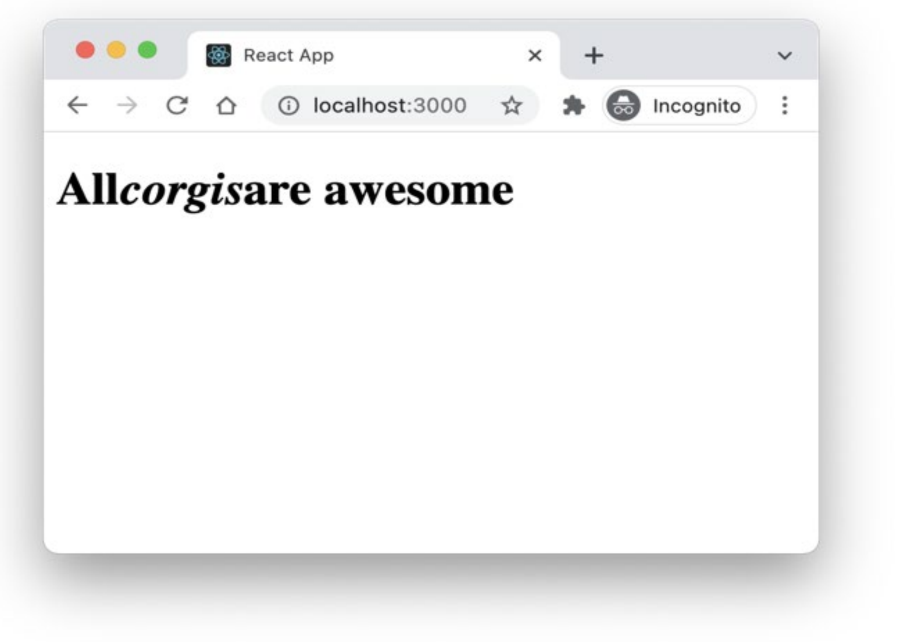
Listing 3.13 Naïve implementation of a partially emphasized message

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <h1>
         All    #A
         <em>corgis</em>    #B
         are awesome    #C
       </h1>
    );
  }
}
export default App;
```

#A Some plain text
#B Then another JSX node
#C Then some more plain text

```
rq03-bad-whitespace
```

This seems pretty reasonable, no? Let's run this app with CRA and watch it in effect in the browser – see the result in figure 3.6.



All *corgis* are awesome

Figure 3.6 Emphasized message with improper whitespace

That's clearly wrong. The space around the word *"corgis"* just collapsed. What happened? Normally newlines and tabs are ignored as white space in JSX. When we earlier in this chapter had this JSX:

```
return (
  <main>
    <h1>Hello world</h1>
  </main>
);
```

We did not actually want spaces between the elements `<main>` and `<h1>`. We just formatted it on multiple lines because it looks pretty - not because we want a lot of extra white space rendered in the browser. So if there is white space between elements in JSX that include newlines, all the white space is collapsed. It does not matter if you have an extra normal space at the end of the line marked #A in listing 3.8. If there is a newline between the elements, all white space is collapsed.

So, how could we do this correctly? There are two ways to do it:

1. Either don't use newlines at all between the elements, or
2. Add spaces as expressions in the code.

The latter sounds a bit complex and doesn't look all that good, but can be necessary. Let's see the first solution in practice - no newlines - in listing 3.14.

Listing 3.14 Partially emphasized message without newlines

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <h1>    #A
        All <em>corgis</em> are awesome    #B
      </h1>    #A
    );
  }
}
export default App;
```

#A We have new lines before and after the heading
#B But no newlines inside the heading

Note that we can have newlines before and after the message (because the white space here can be collapsed–we don't care about it). We just don't want newlines in places, where we want actual space characters to be inserted.

Now let's look at space expressions in listing 3.15.

Listing 3.15 Partially emphasized message with space expressions

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <h1>
         All
         {' '}    #A
         <em>corgis</em>
         {' '}    #A
         are awesome
       </h1>
    );
  }
}
export default App;
```

#A Space inserted as an expressions

Here we add a space using curly brackets. This will force the JSX engine to include the spaces as actual elements and not treat them as part of the negligible white space that normally exists between elements. You will often see developers append such space-as-expressions at the end of the line before the newline as in listing 3.16.

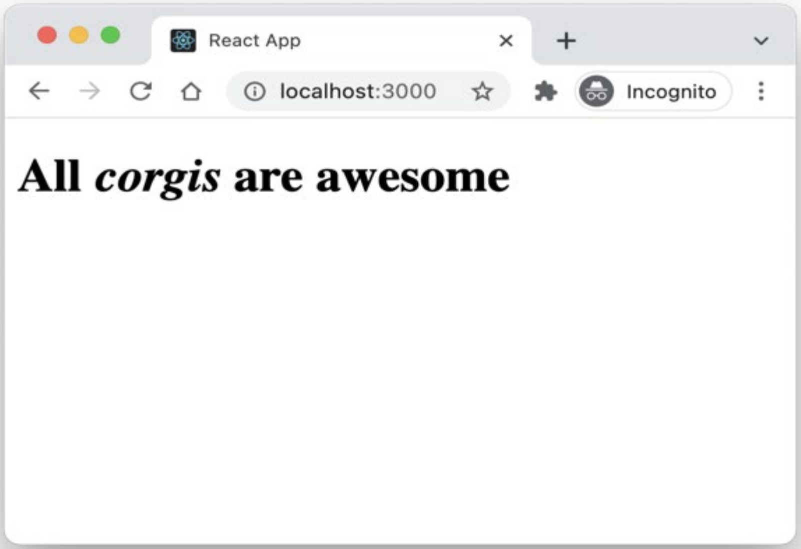
Listing 3.16 Partially emphasized message with fewer lines

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <h1>
         All{' '}    #A
         <em>corgis</em>{' '}    #A
         are awesome
       </h1>
    );
  }
}
export default App;
```

#A Space expression appended at the end of the line

```
rq03-good-whitespace
```

Both of the above will render our message correctly in the browser—a message that nobody can contest as can be seen in figure 3.7.



All *corgis* are awesome

Figure 3.7 All corgis are now correctly rendered as awesome

3.4.9 data- attributes

Sometimes, you want to pass additional data using DOM nodes. While you should not use your DOM as a database or local storage, sometimes you need to do it to pass variables to third-party libraries. If you need to create custom attributes and get them rendered, use the `data-` prefix.

For example, this is a valid custom `data-object-id` attribute that React will render in the view (HTML will be the same as this JSX):

```
<li data-object-id={object.id}>...</li>
```

If the input is the following JSX element, React won't render `object-id`, because it's not a standard HTML attribute (HTML will miss `object-id`, unlike this JSX):

```
<li object-id={object.id}>...</li>
```

3.5 Quiz

1. To output a JavaScript variable in JSX, which of the following do you use? `=`, `<%= %>`, `{}`, or `<?= ?>`
2. The class attribute isn't allowed in JSX. True or false?
3. The default value for an attribute without a value is *false*. True or false?
4. The inline style attribute in JSX is a JavaScript object and not a string like other attributes. True or false?
5. If you need to have if/else logic in JSX, you can use it inside `{}`. For example, `class={if (!this.props.isAdmin) return 'hide'}` is valid JSX code. True or false?

3.6 Summary

- JSX is just syntactic sugar for React methods like `React.createElement`.
- You should use `className` and `htmlFor` instead of the standard HTML class and for attributes.
- The style attribute takes a JavaScript object, not a string like normal HTML.
- Ternary operators and IIFE are the best ways to implement if/else statements.
- Outputting variables, comments, and HTML entities are very easy and straightforward.
- Several multi-word HTML and SVG attributes are renamed in React so pay attention to these special attributes and remember to check if your attributes correctly make it into the HTML document.
- JSX needs to be transpiled into JavaScript before it can run in the browser, but you rarely have to worry about that. If you do have to, there are a number of tools out there including Babel as the most popular one at time of writing.

3.7 Quiz answers

1. You use `{}` for variables and expressions.
2. True. `class` is a reserved JavaScript statement. For this reason, you use `className` in JSX.
3. False. The default value for a property with no value specified is *true*.
4. True. `style` is an object for performance reasons.
5. False. First, `class` isn't the proper attribute name, it's `className`. Then, instead of `if return` (which isn't valid JavaScript anyway in this context), you should use a ternary operator or logical expressions. You could do it like this: `className={this.props.isAdmin || 'hide'}`.

4 Functional components

This chapter covers

- **Introducing functional components**
- **Comparing functional components to class-based components**
- **Choosing between the two types of component definitions**
- **Converting a class-based component to a functional**

React was based on class-based components for a long time in the early years, but at some point along the way, an alternative came along for the simplest of components. Functional components are a more succinct and in some regards simpler way of writing React components, but nowadays with the same feature set as their class-based cousins.

The term functional component is not meant as a contrast to a non-functional component--nobody has any use for those. Rather, the functional part refers to the component definition itself being a JavaScript function rather than a JavaScript class.

In the beginning, functional components were less powerful than class-based components, but when React Hooks came in React 16.8, functional components were suddenly as powerful, if not more, than their class-based siblings. Today, many React developers exclusively use functional components, as they are the primary method recommended by the React team.

Class-based components are still fully supported in React and they're probably not going anywhere anytime soon. You will also find them very common "in the wild", for several reasons:

- Not all older codebases have been refactored away from class-based components and must still be maintained.
- Some older libraries still only document how they interface with class-based components and thus require your code to use them to interface with the library

correctly.
- Some long-time React developers started using class-based components and feel more comfortable with them, so they prefer to stick to them when possible.
- The mental model of a component lifecycle changed quite a bit when going from class-based to functional components and in some instances, the re-render lifecycle can be easier to maintain when using the old class-based approach.
- Finally, a tiny subset of the core functionality in React is only possible using class-based components (Error Boundary in particular).

Not only are functional components here to stay, but they're also going to take over the world. Or at least the world of React. All indicators point to functional components being the main way to write React going forward. Writing functional components makes your life as a developer significantly easier with no downsides.

In this chapter, we'll go over what functional components are, how they differ (and how they don't) from class-based components, how to choose which component type to use in your own projects, as well as how you can convert a class-based component to a functional one.

> **NOTE** The source code for the examples in this chapter is available at https://github.com/rq2e/rq2e/tree/main/ch04. But as you learned in chapter 2, you can instantiate all the examples directly from the command line using a single command.

4.1 The shorter way to write React components

In this section, we will introduce functional components and slowly add some extra utilities on top of that. These utilities are merely "syntactical sugar" and it's often enabled by modern JavaScript features rather than React-specific functionality. However, we will introduce these techniques in this chapter, because we will be using all of them in later chapters. They are all very standard in the industry, so you will see them in React codebases all the time.

These utilities are all about simplifying how you write and interact with components:

- Simplifying access to properties using destructuring
- Simplifying the component API with default values
- Simplifying the component signature using pass-through properties

Together this will give you a good foundation to write simple presentational React components using concise component definitions.

4.1.1 An example application

Let's create a simple React application: a menu with a list of links all built with plain HTML. This is a very simple HTML fragment, but it is the building block of every web application.

We will use this example to illustrate that when components get even a tiny bit complex, the three utilities mentioned previously will help us keep our components simple both on the outside and the inside. See the component tree in figure 4.1.

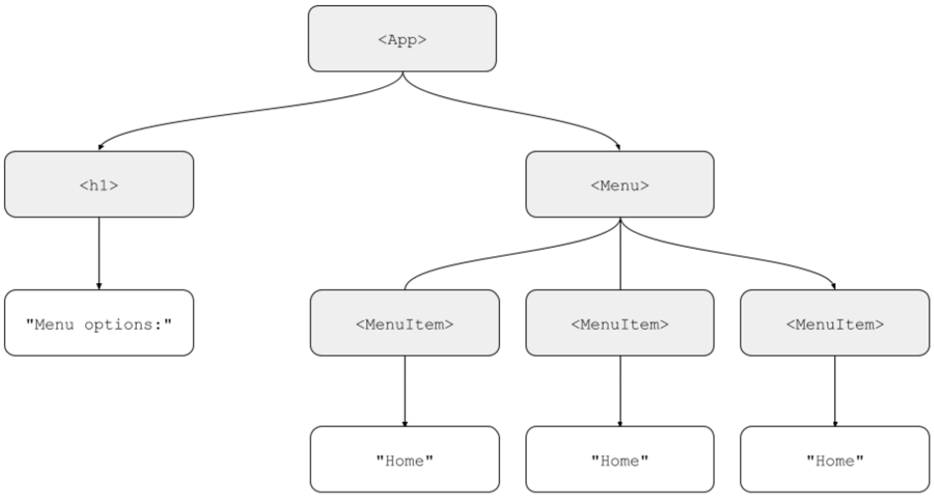
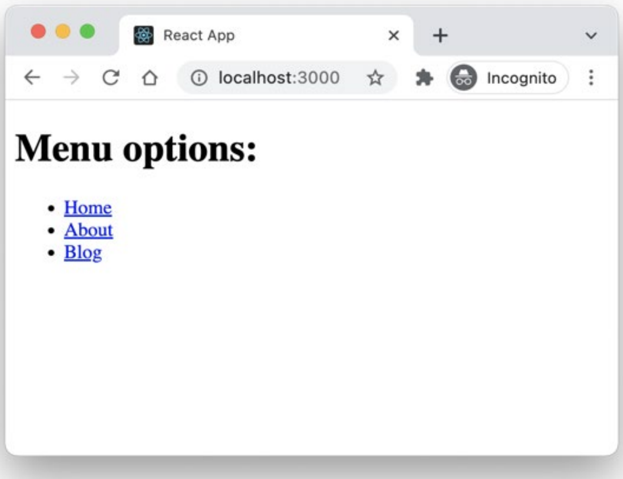


Figure 4.1: A diagram of our menu application.

The output of this application will look like this in the browser:



Menu options:

- Home
- About
- Blog

Figure 4.2: Our menu application as seen in the browser. It's just plain HTML with a list of links and no styling.

First, we will create it as we have seen previously using class-based components, and then secondly we will create those same components using functions.

When we get there, we will have a short discussion about which way is better. Do note, that this is a subjective discussion - there is no right answer and you should feel free to use whatever method you feel works best for you.

IMPLEMENTATION USING CLASSES

This application includes three components. Overall we have the <App/>, and inside that a <Menu /> and its <MenuItem />'s. Let's just for now put everything in the same file - this can all go inside the app.js file in a new minimal project:

Listing 4.1 Menu application using classes

```
import { Component } from 'react';
class App extends Component {    #A
  render() {
    return (
      <main>    #B
        <h1>Menu options:</h1>    #B
        <Menu />    #C
      </main>
    );
  }
}
class Menu extends Component {    #A
  render() {
    return (
      <ul>    #B
        <MenuItem label="Home" href="/" />    #D
        <MenuItem label="About" href="/about/" />    #D
        <MenuItem label="Blog" href="/blog" />    #D
      </ul>
    );
  }
}
class MenuItem extends Component {    #A
  render() {
    return (
      <li>    #B
        <a    #B
           href={this.props.href}    #E
           title={this.props.label}    #E
        >
          {this.props.label}    #E
        </a>
      </li>
    );
  }
}
```

#A Defining a new component
#B Using a standard HTML tag
#C Making an instance of another custom component
#D Passing properties to a custom component
#E Using properties passed to a custom component

```
rq04-menu-class
```

This application only uses things we already know - we can nest components, we can use both built-in HTML components and our own custom components, we can pass properties to child components, and we can access properties passed to our custom components.

This uses the render function only, just as we've seen previously.

IMPLEMENTATION USING FUNCTIONS

For this example, we're just going to throw ourselves in at the deep end. Let's see what this same application looks like using functional components:

Listing 4.2 Menu application using functions

```
function App() {    #A
  return (
    <main>
      <h1>Menu options:</h1>
      <Menu />
    </main>
  );
}
function Menu() {    #A
  return (
    <ul>
      <MenuItem label="Home" href="/" />
      <MenuItem label="About" href="/about/" />
      <MenuItem label="Blog" href="/blog" />
    </ul>
  );
}
function MenuItem(props) {    #B
  return (
    <li>
      <a href={props.href} title={props.label}>
        {props.label}
      </a>
    </li>
  );
}
```

#A Two functional components that do not take any arguments
#B A functional component that takes a single argument

```
rq04-menu-function
```

This almost looks too good to be true, but that's it. To create a functional component, we simply create a function and return JSX, that is it.

If we need to access properties passed to the component, we can do that through the single argument passed to the function, which is a frozen object of properties. This works similarly to `this.props` in a class-based component.

4.1.2 Any function will do

As you saw in listing 4.2 in the annotations, some functional components accept a `props` argument, but some do not. Also, we used the statement version (i.e. `function name() {}`) of the function definition, but we didn't actually have to. Any value that can be executed as a function, that returns JSX, can be used as a component.

You can even define them inline in another component. This is not generally considered a good practice, but sometimes it might be useful.

Listing 4.3 Even shorter application definition

```
const App = function() {                     #A
  const Menu = () => { return <ul /> };      #B
  return (
    <main>
      <h1>Menu options:</h1>
      <Menu />
    </main>
  );
}
```

#A Functional expression using the "function" keyword
#B Functional expression using arrow notation

Here we define one component, `App`, using a function expression, and another component, `Menu`, right in the function body using arrow-notation.

The latter function can even be shortened further using implicit return:

```
const Menu = () => <ul />;
```

Yes, this is a fully valid React component. It's a very simple component, for sure, and it doesn't do much (yet), but for all intents and purposes, this is a React component.

We will get back to how this feature of "any function is a component" can be useful in later chapters. For now, just remember that we write our components in a certain way out of convention rather than framework constraints.

4.1.3 Destructuring properties

In the previous `MenuItem` example, we received our properties in the functional component as a props object and accessed the properties on the object later using e.g. props.label.

A more common approach used by many React developers is to destructure the properties directly in the function signature.

Destructuring in JavaScript generally takes this form:

```
const someObject = { a: 1, b: 2, c: 3 };
const { a, b } = someObject;
```

This expression assigns the value of `a` from inside `someObject` to the variable `a`. And similarly the value of `b` inside the object to the variable `b`. The value of `c` inside the object is ignored, as we don't destructure that in our expression.

We can also use destructuring when accepting object arguments to a function:

```
function log({ message, level }) {
  console.log(level.toUpperCase(), " Message:", message);
}
log({ message: "Unknown product", level: "error" });
```

This would result in the console output:

```
ERROR Message: Unknown product
```

For such a simple example, you might question why we don't make it two different arguments instead, such as:

```
function log(message, level) { ...
```

But as functions get more complex and more arguments are added, using just a single object makes it a lot easier to call the function with a variable number of arguments instead of having to remember that `level` is the fifth argument, etc.

In a functional React component, properties are always given as the first (and only) argument to our defining functions. We can use the method of destructuring the argument object to make the component definition even cleaner as in listing 4.4.

Listing 4.4 MenuItem with argument destructuring

```
function MenuItem({ href, label }) {    #A
  return (
    <li>
      <a href={href} title={label}>    #B
        {label}    #B
      </a>
    </li>
  );
}
```

#A Here we destruct the argument in the function definition
#B This allows us to use the properties without going through the props object

```
 rq04-menu-destruct
```

This is of course completely identical to doing destructuring in a line of its own inside the function definition as in listing 4.5

Listing 4.5 MenuItem with explicit destructuring

```
function MenuItem(props) {    #A
  const { href, label } = props;    #B
  return (
    <li>
      <a href={href} title={label}>    #C
        {label}    #C
      </a>
    </li>
  );
}
```

#A Here we still access a props argument with destructuring it
#B However, we destruct it as a separate statement in the very first line in the component
#C We can then proceed to use the properties as before

In this book, we will use the approach shown in listing 4.4 with argument destructuring directly in the component definition. You will also often see this in the wild, as many React developers use this convention. But to reiterate, it is merely a convention, so other variants are possible.

4.1.4 Default values

An added benefit of using destructured properties is that we can also introduce default values.

Let's say that our menu link to the blog should be opened in a new browser window (or tab), but the other links should just open regularly in the same session.

We can do that by adding a new property, target, that the menu has to specify as in listing 4.6.

Listing 4.6 Menu items with targets

```
function Menu() {
  return (
    <ul>
      <MenuItem label="Home" href="/" target="_self" />    #A
      <MenuItem label="About" href="/about/" target="_self" />    #A
      <MenuItem label="Blog" href="/blog" target="_blank" />    #A
    </ul>
  );
}
function MenuItem({ label, href, target }) {    #B
  return (
    <li>
      <a href={href} title={label} target={target}>    #C
        {label}
      </a>
    </li>
  );
}
```

#A We add a new property to every instance of the menu item component
#B Inside the component, we accept the new property in the destructuring

#C And we assign the property to the relevant JSX element as an attribute

However, one could argue that it makes sense that opening a link in the same session is the default behavior and the menu shouldn't have to specify that. We can implement this using default values in the function definition. Let's go ahead and do that in listing 4.7.

Listing 4.7 Menu items with a default target

```
function Menu() {
  return (
    <ul>
      <MenuItem label="Home" href="/"/>
      <MenuItem label="About" href="/about/" />
      <MenuItem label="Blog" href="/blog" target="_blank" />
    </ul>
  );
}

function MenuItem({ label, href, target="_self" }) {
  return (
    <li>
      <a href={href} title={label} target={target}>
        {label}
      </a>
    </li>
  );
}
 rq04-menu-default
```

Do note, that this is not React-specific functionality, but just normal JavaScript functionality. In a later section in this chapter, we'll talk more about default properties, because you can also specify default properties using React and you need to use this other syntax for class-based components. For functional components, you can even do both at the same time for added confusion. But we will get back.

ORDERING PROPERTIES

You can specify your component properties in any order you feel like. It is however common JavaScript practice to specify properties with defaults at the end of the definition, but nothing is preventing you from doing it differently.

This means that the following line is generally not recommended, but still completely valid:

```
function MenuItem({ label, target="_self", href }) {
```

And this is the recommended order:

```
function MenuItem({ label, href, target="_self" }) {
```

This only refers to the order of non-default properties versus default properties - the internal order of either list of properties has no general ordering and it's up to you or your team to set any such recommendations.

4.1.5 Pass-through properties

Let's make our example even more hypothetical and say, that we need various extra properties on the different elements:

- The home link needs a class of "logo" (remember to use className).
- The about link needs an ID of "about-link"
- The blog link needs an ID of "blog-link"

Let's implement this using what we know so far using default values for unspecified values in listing 4.8.

Listing 4.8 Menu items with many default values

```
function Menu() {
  return (
    <ul>
      <MenuItem label="Home" href="/" className="logo" />
      <MenuItem label="About" href="/about/" id="about-link" />
      <MenuItem label="Blog" href="/blog" target="_blank" id="blog-link" />
    </ul>
  );
}
function MenuItem({ label, href, target="_self", id=null, className=null }) {
  return (
    <li>
      <a href={href} title={label} target={target} id={id} className={className}>
        {label}
      </a>
    </li>
  );
}
```

This is beginning to look a bit repetitive. We are accepting a bunch of arguments only to just pass them straight through to a single element - even with the same name and everything else intact.

THE REST SYNTAX

What if we could just say that some select arguments we care about and want to handle specially, but all other arguments should just be passed straight through to the target element? We can do that too - using another modern JavaScript concept known as the rest syntax.

When destructuring an object, you can use the rest syntax, denoted by three periods, to specify an object, that will be assigned all the "left-over" properties not already assigned:

```
const someObject = { a: 1, b: 2, c: 3, d: 4 };
const { a, b, ...otherAttrs } = someObject;
```

The two properties, c and d, that we didn't already reference in the destructuring statement are transferred as properties to a new object with the given name, otherAttrs.

That means that this above code snippet is equivalent to the following code snippet:

```
const a = 1;
const b = 2
const otherAttrs = { c: 3, d: 4 };
```

We can use this in a component function definition like this:

```
function MyComponent({ a, b, ...rest }) {
  // a = 1, b = 2, rest = { c: 3 }
}
// Later:
<MyComponent a="1" b="2" c="3" />
```

We can capture all the remaining properties in an object, often called rest. Now we just need to use this object and apply all the properties inside it to an element in the output.

We've already previously seen how to assign properties to a JSX element from an object, but to reiterate, we do this using the spread operator:

```
const extraProps = { target: "_blank", id: "link" }
return <a href="/blog/" {...extraProps} />
```

Remember to wrap the spread inside brackets, otherwise, it will not work.

REST IN PRACTICE

Let's go back to our example. We want to capture the label and href properties passed to our <MenuItem /> component, but we don't really care about the rest. If any other properties are passed to the component, we just want to pass them straight through to our target element.

Putting this all together our component becomes listing 4.9.

Listing 4.9 Menu items with rest and spread

```
function Menu() {
  return (
    <ul>
      <MenuItem label="Home" href="/" className="logo" />
      <MenuItem label="About" href="/about/" id="about-link" />
      <MenuItem label="Blog" href="/blog" target="_blank" id="blog-link" />
    </ul>
  );
}

function MenuItem({ label, href, ...rest }) {      #A
  return (
    <li>
      <a href={href} title={label} {...rest}>      #B
        {label}
      </a>
    </li>
  );
}
```

#A In this line, "..." is the rest syntax
#B In this line, "..." is the spread operator

```
rq04-menu-rest
```

This looks a lot nicer now. We don't have to specify all those extra properties that we don't even really care about. We allow any other component to pass whatever they want except that we do something special for a few properties - here label and href.

A few things to note here: As you can see, the rest syntax and the spread operator are identical. Both are three periods before a variable name. However, they are used very differently, as one is used for destructuring and the other for assigning. They have a similar nature, which is why they look the same, but they are different operators altogether.

Using the variable name rest for the extra parameters is a common convention, but is by no means a requirement anywhere. You will see many developers use it, but feel free to change it to something that makes sense to you.

Also, this again is not React-specific functionality, but merely a useful artifact of the JavaScript language that you will see many React developers use. We will be using it in future chapters as well.

REST AND PROPERTY ORDERING

The rest syntax has to be the very last element of the object destructuring, so you have to specify it at the end of the property list.

When combined with default properties, which is of course still possible, the ordering goes:

1. Properties without defaults
2. Properties with defaults
3. Rest

An example of all three types of properties could be:

```
function MenuItem({ label, href, target="_self", ...rest }) {
```

4.2 A comparison of component types

At this stage in your React edification, the differences between functional components and class-based components might seem small or even insignificant. Boiled down to its bare minimum, it is the difference between writing the following as a class-based component:

```
class Menu extends Component {
  render() {
    return <nav />;
  }
}
```

Versus writing the following as a functional component:

```
function Menu() {
  return <nav />;
}
```

When we in later chapters get to more complex components, especially when using callbacks and state, things get more complicated, and the difference between functional components and their class-based siblings become larger and larger. And when we get to composition of components and reuse of generalized functionality, very different patterns emerge in the two worlds - almost completely different.

The choice between component types is quite fundamental in your React journey, but quite frankly, it's not really a choice anymore. You will probably be using functional components unless there's a strong reason not to for your particular project or development team.

Nevertheless, we will in this section go over the benefits and disadvantages of functional components, as well as go over some factors that are in fact not factors in this choice.

4.2.1 Benefits of functional components

The following is a non-exhaustive list of some subjective benefits to using functional components:

Compactness: Functional components are most often more compact in terms of lines of code and pure template code overhead than class-based components. You simply have to type fewer characters when implementing functional components.

Readability: It can be much harder to track down the origin of some property in a class-based component going through layers of composed higher-order components than to do the same thing in a functional component using hooks. Generally, functional components are much easier to read and understand even at a glimpse.

Purity: The purity of a function is easier to determine and the side-effects of impure functions are easier to deduce due to the existence of hooks. The purity or lack thereof of a class-based component is generally harder to deduce and that can make debugging and understanding a lot harder.

Simplicity: Functions are a fundamental part of any programming language and even in mathematics. The theoretical tools used to describe, work with, compose, and explain functions are far greater than those we have to do the same for classes. Classes are of course also fundamental in many programming languages, but they are still a significantly higher-level abstraction than simple functions.

Testability: Due to the ability to break off bits of functionality into independent hooks, functional components are often much easier to unit test, as you can break

Popularity: The preference of functional components is a benefit in and of itself. Most other React developers will by now be more at home using functional components, most new development happens in the ecosystem of functional components, and the vast majority of new content about React (videos, tutorials, books, etc) refers exclusively to functional components.

Note that all of these benefits are about developer experience. The actual end product - the final web application available to end-users - is not improved or hindered by the choice of component type. It's almost exclusively about making it easier for developers to write, maintain, and debug components, where the syntax of functional components really shines.

In general, using functional components is more elegant, more succinct, and most importantly, far easier to understand. This is of course partially a subjective opinion on behalf of the authors, but it is a common opinion found prevalent among React developers as can be seen in public codebases on Github and similar repositories.

4.2.2 Disadvantages of functional components

In and of itself, there are no direct disadvantages to using functional components. There are a few select scenarios, where you can't (or shouldn't) use functional components, but those are very rare and far between.

We will cover these in the next section.

For any feature that you can create in both a functional component and a class-based component, there are no disadvantages to creating said functionality in a functional one.

4.2.3 Non-differences between component types

Some factors that are important to developers, development teams, and business units alike, are not a factor at all in the choice of component types.

These non-factors include:

Speed: There is no inherent speed difference in running a simple component as a functional one versus a class-based one. The tools to make every component, and thus your entire application speedy and responsive are slightly different in the two types of components. Most would probably argue that the tools are a bit more transparent and easier to understand in functional components, but similar tools exist for class-based components, so any component can be made fast or behave sluggishly if not optimized properly.

Composability: Albeit the design patterns used are very different, code reuse and composability of functionality are just as good and well-supported in both types of components.

Usability: For the end-user visiting your web application, there is no difference whatsoever if you're using one type of the other. User experience has no impact on this decision.

Accessibility: Making React components accessible is a skill of its own, but that applies regardless of whether you're writing them one way or the other.

Reliability: Components are just as easy or difficult to make reliable or correct regardless of the choice of component type. Reliability is a property of good software development, not the choice of tooling.

Maintainability: At least for now, there are no indications that class-based components are being deprecated, so either component type is expected to be fully supported by React in all future versions.

While all the above are important aspects of software development, they're not directly influenced by the choice of component types, but rather by the competence and vision of the developer or development team wielding the keyboard.

4.2.4 Choosing component type

If the question is "what component type should I choose for my project?" the short answer is simply: Use functional components. The slightly longer answer adds the following postfix: unless there's a very strong reason not to.

In our most informed opinion, you should always use the latest stable version of any technology, and for React, that is most definitely functional components over class-based components. Functional components have been around for quite a while by now, most new development happens in functional components and their environment (hooks in particular), and most other developers will be using functional components as well.

However, there might be scenarios, where even we would consider using class-based components, and we will cover those in the very next section.

4.3 When not to use a functional component

As mentioned previously, anything you can do with a class-based component, you can do with a functional. With a single exception, error boundary. There are a few other instances where you might want to choose to use a class-based component anyway, even if you don't have to for technical reasons.

In this section we'll discuss the following cases where you might want to avoid using functional components:

- You want to set up an error boundary to handle errors occurring further down the render tree.
- You are working in a codebase primarily composed of class-based components and want to make something that fits in.
- You are using a library that is tailored to class-based components only.
- You're specifically tasked to use the built-in React functionality of getSnapshotBeforeUpdate.

The above items are written in prioritized order of their likeliness to occur in your everyday work. And given that the very first item is an extremely specialized case necessary in only the largest and most complex codebases, you're not likely to come across any of these exceptions at all.

We'll cover each of the exceptions in the following subsections.

4.3.1 Error boundary

Establishing an error boundary is a valid concern for a mature React codebase once it gets to a certain complexity level, so this is something you're likely to come across if you're working on a large codebase.

Currently, at the time of writing (with React 18 on the horizon but still not out yet), there is still no way to solve this without using a class-based component. There aren't even plans to convert the error boundary functionality to a hook or similar, that would allow it to be possible in a functional component.

An error boundary is a way of establishing a fallback in case a child component throws a JavaScript error. You should of course always strive to never have unhandled errors, but it might happen as things get complex, input changes, APIs evolve and your codebase gets more complex and harder to properly cover by tests, errors will occur. An error boundary is your way of making sure that when such an error does occur, at least the end-user is presented with a nicely formatted error message along with your sincere apologies. You should probably also log the error to your analytics tool of choice.

There are two methods in the React API that deal with errors occurring in child components. One is getDerivedStateFromError, where you can set an internal flag that this component should render differently because an error occurred somewhere, and the other is componentDidCatch, where you get the actual error that occurred along with its stack trace and other info. This latter part allows you to log it for debugging purposes.

We will not go into detail about how these work in this book, as it is outside the scope, but if you do find yourself needing them, the React documentation on both methods is pretty substantial.

If you find yourself needing to catch errors in a component tree, you have to use a class-based component for at least this one component. You can still keep 99% of your components functional, despite having a single class-based error boundary or two.

4.3.2 Codebase is class-based

Imagine that you're hired to a development position in a company that has an old React codebase that they're still actively working on. It's a huge application, maybe with hundreds or even thousands of components, and an extensive set of complex functionality.

You are asked to add some new functionality to just a tiny part of this application. While there is no problem mixing class-based and functional components, it might seem very odd to other developers, that some components are written in one style, and others in a different style.

Refactoring the entire codebase to class-based components would be a huge undertaking, but is hopefully the goal for the engineering team in the long term. However, it is very likely that there will be a transition period, where only certain parts of the codebase have been converted, and you will be asked to keep using classes in some parts while using functional components in others.

As React ages and the ghost of class-based components is a relic of a further and further past, this scenario becomes less and less likely.

If you find yourself in such a scenario, we recommend using the wisdom of the team and going with the flow. Don't force a conversion before the team is ready as a whole, and don't go against the agreed-upon coding conventions of your team.

4.3.3 Library requires class-based components

This scenario is somewhat hypothetical, as we can't actually find a library with this property, but that's not to say that it doesn't exist.

There might be a circumstance where you're interacting with third-party functionality, that requires you to use class-based components.

The most likely case where this would happen is that you want to use an old library that hasn't been updated since before React Hooks came out and their examples and guides still use class-based components. That does not mean that you can't use the library with hooks, it just means that you're on your own and can't use the library documentation to help you out if things don't work.

While outdated documentation is the most likely culprit of the library instructing you to use class-based components, we can't rule out that there might be a library that doesn't work with hooks at all.

If either of the above scenarios occurs, your best bet is to look around for a more modern library. Many things have changed in the 3 years since hooks came out - not just notation - and you will probably find that the library in question is behind the curve on many things if it's been unmaintained for that long.

4.3.4 Snapshot before updating

There is another built-in function in the React API, that doesn't exist in a hooks-only React world: `getSnapshotBeforeUpdate`.

It's an extremely specific piece of functionality, that has just the narrowest use-case, the details of which we will not go over here. You will be able to work around it easily with hooks if you just structure your components slightly differently.

However, if you are specifically tasked to use this functionality, there's no way around it (also, who gave you this weird task?). If you're just tasked with solving a problem, where `getSnapshotBeforeUpdate` would be a solution in a class-based component, you would be able to find a similar solution using functional components.

This method is only mentioned here for completion, not because it's a method frequently used at all. A quick search of Github reveals only 7 repositories mentioning the method. 2 are lists of React functions, 2 are examples of how to use this specific method and the last 3 are old unmaintained demos. So this whole method is a candidate for functionality that will more likely disappear from the React API completely rather than be upgraded to a functional equivalent.

4.4 Conversion from a class-based to a functional component

We've already seen a simple component converted from a class-based component to a functional one — we saw that between code listings 4.1 and 4.2. In this section, we'll dig more into this conversion, iron out some gotchas, and prepare you for the journey ahead, as we will keep coming back to this conversion as we add more and more complicated functionality to our components in the next chapters.

For the purpose of this conversion exercise, we'll create another simple web application: a gallery with images and titles for each. It's a very simple visual application, that has no interaction again (as we have learned how to add that) but highlights different features of component internals, so we have to use some different tricks to convert the components.

The output of this application will look like figure 4.3 in the browser.

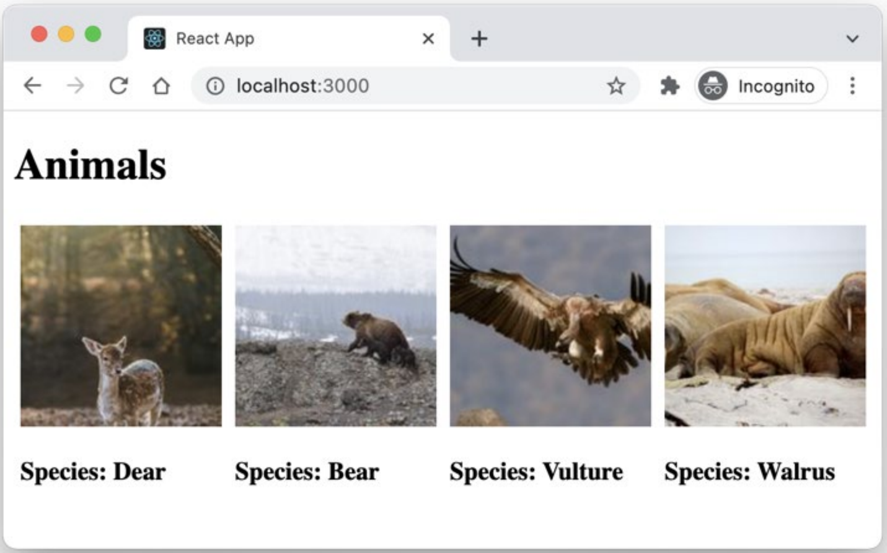


Figure 4.3: Our gallery application as seen in the browser. Simple figures with captions.

We will create 4 versions of this component, which iterate as follows:

- Version 1: using only the render method
- Version 2: using a secondary method as utility only
- Version 3: using a secondary method with class access
- Version 4: using the constructor to initialize a calculation

The reason why we go through these iterations is to see how to convert class-based components to functional components when the classes use more and more advanced patterns that require slightly different

Finally, we will discuss how 1-to-1 conversion gets more complex bordering on impossible as components get more complicated.

4.4.1 Version 1: Render only

In this first iteration, we're going to implement this with classes completely similar to our menu earlier in this chapter.

ORIGINAL

We use three components, and each just uses their render method to return JSX:

Listing 4.10 Gallery v1 using classes

```
class App extends Component {
  render() {    #A
    return (
      <main>
        <h1>Animals</h1>
        <Gallery />
      </main>
    );
  }
}
class Gallery extends Component {
  render() {    #A
    return (
      <section style={{display: 'flex'}}>
        <Image index="1003" title="Dear" />
        <Image index="1020" title="Bear" />
        <Image index="1024" title="Vulture" />
        <Image index="1084" title="Walrus" />
      </section>
    );
  }
}
class Image extends Component {
  render() {    #A
    return (
      <figure style={{margin: '5px'}}>
        <img
          src={`//picsum.photos/id/${this.props.index}/150/150/`}
          alt={this.props.title}
        />
        <figcaption>
          <h3>Species: {this.props.title}</h3>
        </figcaption>
      </figure>
    );
  }
}
```

#A Our 3 class components only have a render function and no other methods

```
rq04-gallery-class-v1
```

CONVERSION

When converting such a simple component to a functional component, we simply directly convert the render method of the class to a function with the same name as the class. Thus it follows this simple template. If you have this class:

```
class MyComponent extends Component {
  render() {
    ...
  }
}
```

We end up with this:

```
function MyComponent() {
  ..
}
```

The only other thing we also have to do is to make sure to directly destructure the props in the component definition rather than access them through this.props as we have seen before. This leads us to the following result in listing 4.11.

Listing 4.11 Gallery v1 using functions

```
function App() {    #A
  return (
    <main>
      <h1>Animals</h1>
      <Gallery />
    </main>
  );
}
function Gallery() {    #A
  return (
    <section style={{display: 'flex'}}>
      <Image index="1003" title="Dear" />
      <Image index="1020" title="Bear" />
      <Image index="1024" title="Vulture" />
      <Image index="1084" title="Walrus" />
    </section>
  );
}
function Image({ index, title }) {    #B
  return (
    <figure style={{margin: '5px'}}>
      <img
        src={`//picsum.photos/id/${this.props.index}/150/150/`}    #C
        alt={title}    #C
      />
      <figcaption>
        <h3>Species: {title}</h3>    #C
      </figcaption>
    </figure>
  );
}
```

#A Component definition changed to a function
#B This component definition also takes destructured properties
#C Property reference changed to a direct variable rather than an object property

```
rq04-gallery-function-v1
```

Nothing surprising here. We used all the tricks we learned so far in this chapter:

- A functional component is simply a function returning JSX
- If we need to accept properties, we destruct them in the function definition
- When we need to access properties, we can do so directly using the destructured variables

In the next couple of subsections, we will iterate the definition of the image component, so we'll see different versions of only that component. For brevity, only the image component will be shown in the sample code listings.

4.4.2 Version 2: Class method as utility

In this version of the implementation, we examine what we would do if the image class had another method that served as a utility function aiding the rendering.

The argument for this is, that the src property of the <img /> element is a bit long and windy and the JSX would look a lot simpler if we had a utility method to render this URL.

ORIGINAL

In this iteration, we will imagine that the developer of the gallery wants to reduce the visual clutter of these lines:

```
<img
  src={`//lorempixel.com/200/100/animals/${this.props.index}/`}
  alt={this.props.title}
/>
```

To something that looks simpler like this:

```
<img
  src={this.getImageSource(this.props.index)}
  alt={this.props.title}
/>
```

This requires us to define a method on the class, `getImageSource`, that takes an argument, index, and returns a string with the URL. This method looks like this:

```
getImageSource(index) {
  return `//lorempixel.com/200/100/animals/${index}/`;
}
```

Putting all this together, the resulting image component looks like listing 4.12.

Listing 4.12 Gallery v2 using classes

```
class Image extends Component {
  getImageSource(index) {    #A
    return `//picsum.photos/id/${index}/150/150/`;
  }
  render() {
    return (
      <figure style={{margin: '5px'}}>
        <img
          src={this.getImageSource(this.props.index)}    #B
          alt={this.props.title}
        />
        <figcaption>
          <h3>Species: {title}</h3>
        </figcaption>
      </figure>
    );
  }
}
```

#A We define a new method on the class
#B Here the new method is invoked with a property as the argument

```
rq04-gallery-class-v2
```

Note that this listing only shows the image component. The app and gallery components are the same as before. We will not bother repeating these above nor in the conversion that follows.

The task now becomes to convert this new class-based component, using multiple methods, to a functional component.

CONVERSION

The key to converting this function here is to recognize that the class method is practically not, in the object-oriented sense of the word, a method of the class, but merely a utility function. In fact, you could move the function completely outside the class and get the exact same result.

Imagine that the code listing above instead looked like listing 4.13.

Listing 4.13 Gallery v2 using classes and a function

```
function getImageSource(index) {    #A
  return `//picsum.photos/id/${index}/150/150/`;
}
class Image extends Component {
  render() {
    return (
      <figure style={{margin: '5px'}}>
        <img
          src={getImageSource(this.props.index)}    #B
          alt={this.props.title}
        />
        <figcaption>
          <h3>Species: {title}</h3>
        </figcaption>
      </figure>
    );
  }
}
```

#A We have moved the method out as a separate and independent function outside the class
#B We know invoke the function as any other function, and not as a method on the class

This works exactly in the same way because the `getImageSource` method did in fact not use any knowledge only available inside the class. In other words, the function was pure and only relied on its input and no other outside information, nor did it have any outside consequences.

Converting this new class-based component using a utility function now becomes just as simple as before. We leave the utility function as is and just convert the component itself in listing 4.14.

Listing 4.14 Gallery v2 using functions

```
function getImageSource(index) {    #A
   return `//lorempixel.com/200/100/animals/${index}/`;
}

function Image({ index, title }) {
  return (
    <figure>
      <img src={getImageSource(index)} alt={title} />    #B
      <figcaption>
        Species: {title}
      </figcaption>
    </figure>
  );
}
```

#A We keep the utility function outside the component definition
#B And we just invoke that function as any other function

```
 rq04-gallery-function-v2
```

This does seem a lot simpler and more compact than the previous iteration in code listing 4.10. The `<img />` tag is much simpler to read and the details of the actual URL generation have been moved to a function dedicated to that only.

Note that we here used the knowledge that the method was pure - i.e. the method did not use any outside information but relied on its arguments only. What if this was not the case? We'll get to that in the next chapter.

4.4.3 Version 3: Real class method

Now let's take a new look at the class method we had in the previous example. Let's instead imagine that the developer implementing this component wanted to use the fact that the method is part of the class and thus has direct access to the properties of the component.

ORIGINAL

Using this information, the method does not need to rely on an argument delivering the index, but can retrieve the index directly from the component properties using `this.props`:

```
getImageSource() {
  return `//lorempixel.com/200/100/animals/${this.props.index}/`;
}
```

Now when we use this method, we don't have to provide an argument, but can just call the method. This results in the component definition in listing 4.15.

Listing 4.15 Gallery v3 using classes

```
class Image extends Component {
  getImageSource() {
    return `//picsum.photos/id/${this.props.index}/150/150/`;    #A
  }
  render() {
    return (
      <figure style={{margin: '5px'}}>
        <img src={this.getImageSource()} alt={this.props.title} />    #B
        <figcaption>
          <h3>Species: {this.props.title}</h3>
        </figcaption>
      </figure>
    );
  }
}
```

#A This time the class method uses the props object directly
#B We can now call the method without passing an argument to it

```
 rq04-gallery-class-v3
```

Now the class method is indeed a method of the class and relies on outside information. What do we do now?

The short answer: There is no direct equivalent of this in a functional component. There are similar ways to achieve the same result, but no direct equivalent.

There are two primary approaches to converting this class-based component to a functional component, each with its advantages and drawbacks:

1. Convert the method to a pure function and move it outside the component
2. Create a local function inside the component

We'll cover both of those approaches and compare them.

CONVERSION USING A PURE FUNCTION

Option 1 is to remember the previous version of the gallery image and try to reverse this advancement of complexity and interconnectedness.

For this method, it's quite simple - the goal is to remove any direct access to component properties or other component-local information and instead pass it as arguments to the function. This would lead us to the same version of `getImageSource` that we saw in version 2, where it took an argument and returned a string.

All in all, implementing this would look exactly like we saw in code listing 4.14.

However, imagine that the method was more complex, and used a lot of properties. Let's say we had this method:

```
getImageSource() {
  const { width, height, index } = this.props;
  return `//picsum.photos/id/${index}/${width}/${height}/`;
}
```

When we use this method in our class-based component render, it looks quite nice:

```
return (
  ...
  <img src={this.getImageSource()} alt={this.props.title} />
  ...
);
```

The usage of this function is quite compact and all the complexity of accessing the different properties is moved to the method only.

If we convert it to a pure function, we suddenly have to pass a ton of arguments to it and the complexity would grow. In our functional component with a pure function, we would have to pass all the properties to the function, and it would suddenly look like this:

```
return (
  ...
  <img
    src={getImageSource(width, height, index)}
    alt={title}
  />
  ...
);
```

This is not as nice and isolated as before. But it would work and it would be a valid conversion.

CONVERSION USING A LOCAL FUNCTION

Option 2 is to convert the class method to a local function inside our functional component. That would look like listing 4.16.

Listing 4.16 Gallery v3 using functions

```
function Image({ index, title }) {
  const getImageSource = () =>
    `//picsum.photos/id/${index}/150/150/`;    #A
  return (
    <figure style={{margin: '5px'}}>
      <img src={getImageSource()} alt={title}/>    #B
      <figcaption>
        <h3>Species: {title}</h3>
      </figcaption>
    </figure>
  );
}
```

#A We now define a local function inside the component, that has access to properties
#B And we just invoke that function as any other function

```
rq04-gallery-function-v3
```

Now because this definition of `getImageSource` is a local function inside our component, it has access to the properties passed to the component, and we don't have to pass all the properties to the helper function.

The downside to this approach is, that every time we create a new component, we create a new local function. This doesn't matter a lot in this example with only 4 components, but imagine a huge complex application with thousands or even millions of instances of some components. If we had millions of instances of our original class-based component as defined in listing 4.15, we would still only have a single definition of the `getImageSource` method and it would not occupy a lot of memory.

However, with our functional component as defined in listing 4.16, every instance of our component would have a locally defined function and each would occupy a slot in the program memory. This is not normally a worry, but it is a slight difference between the two implementations.

When you are converting a class-based component with extra class methods, you can use either option as outlined above. Just be aware of the advantages and disadvantages of both. In the concrete example, both options are fully valid solutions, but sometimes one option will be more appropriate than the other, all depending on the exact circumstances.

4.4.4 Version 4: Constructor

As we have mentioned previously, you can also add a constructor method to your class-based component.

Generally, the constructor is used for initializing attributes that will remain the same in the component's entire lifetime, regardless of the properties passed. This is because the constructor is only executed once, the first time the component is created, and not every time the component properties update or the component re-renders for other reasons.

We will get into a lot more details about component re-renders in future chapters. For now, just know that the constructor is only called once in the component's lifetime, so you should not put any functionality there, that depends on properties that might change in the future.

In this example, we're going to add a constructor to our image component that generates a random ID to be applied to our element. The reason for doing this could be because we wanted to attach it to some external library or we wanted to be able to reference the element using ARIA properties for accessibility.

If we created an ID in the render method, the ID would regenerate every time the component renders, so we create it in the constructor, to make sure that the ID stays the same in the component's lifetime:

Listing 4.16 Gallery v4 using classes

```
class Image extends Component {
  constructor(props) {
    super(props);    #A
    this.id = 'image-'+Math.floor(Math.random()*1000000);    #B
  }
  render() {
    return (
      <figure style={{margin: '5px'}} id={this.id}>    #C
        <img
          src={`//picsum.photos/id/${this.props.index}/150/150/`}
          alt={this.props.title}
        />
        <figcaption>
          <h3>Species: {this.props.title}</h3>
        </figcaption>
      </figure>
    );
  }
}
```

#A Remember to call super(props) in the constructor!
#B We create a class variable with the generated unique ID.
#C We then use this ID in the render method by retrieving it from the class

```
 rq04-gallery-class-v4
```

Note that we simply store the ID as a property directly on the class instance itself using `this.id`. We do not put it in `this.props` for two reasons: a) because we can't (it's a frozen object), and b) because it's not a property passed to our component - it's something we calculated ourselves.

So, how do we convert this to a functional component with the knowledge we have so far? We can't! You will learn the tools to do this in a future chapter, using hooks (`useMemo` in particular), but for now, we do not have the features to do this.

The problem is, that unlike class-based components, that have a constructor, which runs only once when the component is created the first time, and a separate render method, which runs every time the component renders (and re-renders), a functional component only has a single method, that runs every time the component renders, including the first time. In a functional component, there is no real difference between the first render and subsequent renders.

We haven't actually seen a component yet that re-renders, but for now, just trust us that almost all components that you will be writing in React will need to render more than once. If a component only ever renders once, it's most likely a very simple component with no internal logic or state. For example, your web application logo might be defined in a simple component that never changes. We will talk a lot more about component lifecycles and rendering in chapter 6.

To give you a sneak-peak of what this ID generation would look like using functional components, check out listing 4.17, where we use the hook, `useMemo`, to generate a unique ID the first time the component renders, and then reuse this same calculated result on every subsequent render:

Listing 4.17 Gallery v4 using functions

```
import { useMemo } from 'react';    #A
function Image({ index, title }) {
  const id = useMemo(    #B
    () => 'image-'+Math.floor(Math.random()*1000000),
    [],    #C
  );
  return (
    <figure style={{margin: '5px'}} id={id}>
      <img
        src={`//picsum.photos/id/${index}/150/150/`}
        alt={title}
      />
      <figcaption>
        <h3>Species: {title}</h3>
      </figcaption>
    </figure>
  );
}
```

#A Importing the hook from the React package
#B Applying the hook
#C The secret magic sauce that makes the hook run only once - yes, it's just an empty array. It's a so-called dependency array, which we will get back to in chapter 6.

```
rq04-gallery-function-v4
```

We will not go into more detail on how this works right now, but this would be the logical equivalent of initializing a variable in the constructor. You can however look forward to learning more about hooks, rendering, and memoization in chapter 6.

This technique does require some rewriting of the component and you have to think a bit differently, but all the examples seen so far can be converted to functional components without too much work.

4.4.5 More complexity = harder conversion

All the examples seen so far are very simple. We do not have any interaction and we don't have any state. What if you can filter which animals you want to see? What if you can click to expand the information about each animal? All these features require more complex React components, which in turn require more complex logic to convert to a functional one if that's the task you have.

When we introduce more complex features of functional React components in future chapters, we're also briefly going to discuss what this would look like in a class-based component, and how you would handle the conversion of a component using these features, to the equivalent functionality in a functional component.

For now just know that while all functionality (barring the very few exceptions we saw in the previous chapter) can be converted from a class-based component to a functional one, it might not always be simple. Just as we had to do some judgment calls when converting components using class methods, other degrees of complexity are going to introduce several different approaches, some of which will be more applicable to a given situation than others. On top of that, as the original developer starts to combine all these features, you might end up with a very complex component that needs to be completely reworked to make sense in a functional world.

4.5 Quiz

1. Are functional components *less powerful*, *as powerful*, or *more powerful* than class-based components in terms of React functionality, and which applications you can build?
2. How many arguments are passed to a functional component?
3. Which one of these statements is **not** a benefit of using functional components:
4. *Functional components are more compact than class-based components*
   a) Functional components are faster than class-based components
   b) Functional components are easier to understand than class-based components
5. If you are starting a brand new React application, should you use *functional components* or *class-based components*, all else being equal?
6. Converting a class-based component to a functional component is always trivial; *true* or *false*?

4.6 Summary

- Functional components are another way to write React components as an alternative to class-based components.
- Any JavaScript function returning JSX is a functional component, but for the sake of convention, we tend to write functional components in a certain style.
- Certain JavaScript tricks are often used to aid the definition of functional components including destructuring, default values, the rest syntax, and the object spread operator.
- Functional components are at least as capable as class-based components in every respect.
- In certain aspects, functional components are more beneficial to the developer experience, but these benefits do not extend to the final product - that is independent of the choice of component type.
- If given the choice, functional components are the recommended way to write React components.
- Class-based components can generally be converted to functional components, but it might require a lot of work and refactoring of existing functionality.

4.7 Quiz answers

1. Functional components are exactly **as powerful** as class-based components. Any application you can build using one type can also be built using the other.
2. Functional components receive **one** argument: A frozen object of properties.
3. The correct answer is B: **Functional components are not faster than class-based components**. While there might be a slight difference in speed for any naïve implementation, both types of components can when optimized properly, be lightning-fast, or, when ill-composed, be dragging your whole application down. The choice of component type is not in itself an indicator of application speed.
4. All else being equal, you should start any new project using **functional components**.
5. **False**. Converting a simple class-based component to a functional one can often be trivial, but as component complexity grows, the conversion gets more and more complex.

5 Making React interactive with states

This chapter covers

- **The role of state in a component**
- **Using state in functional components**
- **Converting stateful class-based components to functional components**

All the components we have created so far, take some properties and render some HTML based on those properties. We can pass a label property to a button, so the button is displayed with that exact button text, for instance. But we cannot make the button text change, when something happens, such as changing between "Turn on" and "Turn off" when toggled. That's because we lack both the ability to react to something that happens, as well as the ability to store the single piece of information that something *has* happened.

The output of the components we have created so far depends on nothing but their properties. In other words, the components are *"pure"* in functional programming terms. The components have no other inputs and no side effects. If you give the same component the same properties, you will always get the same result and nothing but that result.

All of that is a good thing, and it's exactly what we want. It's also kind of boring. Such components are vital for presenting data but are useless if we want to create an interactive application. If we want to update something when a button is pressed or an input is filled, we need to store that somewhere and pass that information to some other component to react to it. Imagine a login form. When you enter your email and password, we need to store that information somewhere, in order to display error messages if filled incorrectly. When you press the send button we need to send the data to a remote server.

Another word for components that depend only on their properties and have no internal logic beyond that, is a *stateless* component. The alternative is a *stateful* component. By state we are here talking about the ability to change over time by using internal variables. The same component can at one point have one internal state which might result in one JSX output, and later have a different state, which might result in a different output. Imagine a push button that can toggle between being pressed and not pressed. Whether it is pressed or not is the state of the button, and a component that holds state, is *stateful*.

In this chapter, we're going to cover exactly what a stateful application is and what a stateful component does in such an application. We are then going to make this more concrete by seeing how you set, update, and use component state inside a functional component. Despite the very simple API which consists of a single function, `useState`, there's a lot of information to cover.

At the end of this chapter, we will briefly discuss how setting, updating, and using state in a class-based component works. This is done in a related but different way, so there are some important things to be aware of.

When discussing class-based components, we will also introduce how to convert stateful class-based components to stateful functional components. This will come in handy, if you are tasked with working on an older codebase that is still using class-based components, but want to upgrade it to a functional codebase. You would want to do that so you can use the latest and greatest technology available. It can also come in handy if you find examples and guides online that teach you how to do something in a class-based component. There are still thousands of older but useful tutorials out there, and in order to use the advice presented in a modern codebase, you have to convert some of the concepts.

NOTE: The source code for the examples in this chapter is available at https://github.com/rq2e/rq2e/tree/main/ch05. But as you learned in chapter 2, you can instantiate all the examples directly from the command line using a single command.

5.1 Why is React state relevant?

State is essential for making any kind of interactive application. If your application does not have any state, it means that your application is completely static. It cannot change at all once opened in the browser. This might be fine for a blog post or a recipe, but if you want users to login, update, click, or in any other way interact with your application to influence what is being shown, you need your application to be stateful.

React components are individually stateful. Keeping state in a component is what makes your React application as a whole stateful.

Note that while almost all React applications are stateful, *not all components are stateful*. You might have only very few stateful components in your application, but those few components can control state for your entire application and will make sure to update all the stateless components when necessary. While it is extremely hard to generalize about this, a rough estimate would be that probably no more than a third of your components are stateful in your final application, and as applications grow larger and more complex, that ratio is probably going further down.

Imagine a fictional component tree for a fictional application as in figure 5.1 – only the dark components are *stateful* whereas the light ones are *stateless*.

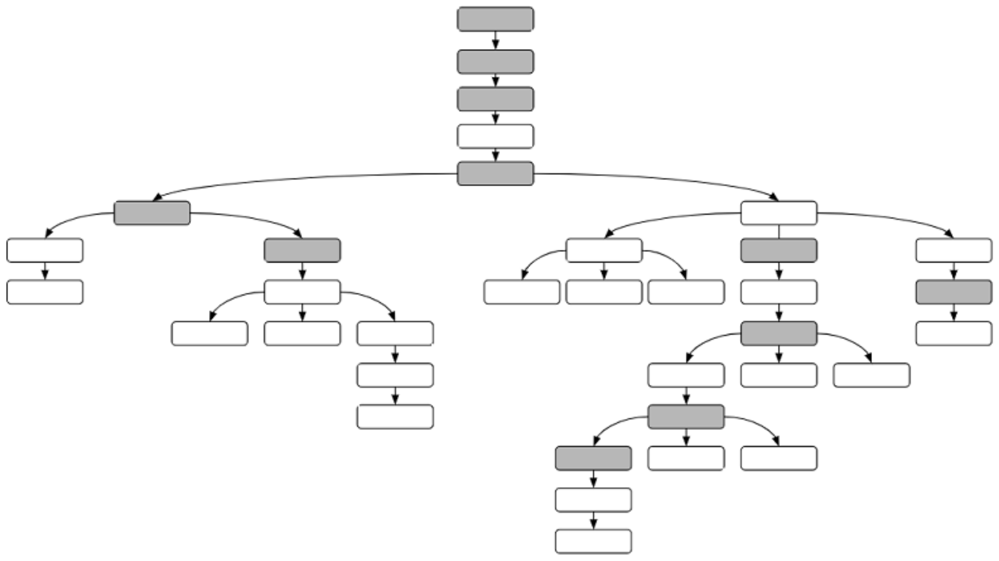


Figure 5.1 The dark components are stateful, the light ones are stateless. Note how stateful components often reside towards the top of your components tree, where stateless components are more prevalent towards the leaves.

React does not have tools to make your application as a whole stateful. A React application is only defined as the sum of its components, so to make your application stateful, you have to make your components stateful.

5.1.1 React component state

Component state is what makes a component *stateful* rather than *stateless*:

- A stateful component is independent of its content and has the ability to update itself based on internal triggers.
- A stateless component can only change or update when its parent component provides it with new properties.

React component state is the mechanism that allows you to store values inside your component that can change over time.

Imagine the difference between a clock component that can display some time of day based on a property passed to it, versus a clock component that is able to update itself every second and continually display the current time. In order to do the latter, the component needs to have a way to store what the current time of day is (and we also need a way to advance that value).

See figure 5.2 for an illustration of the difference between these two approaches.

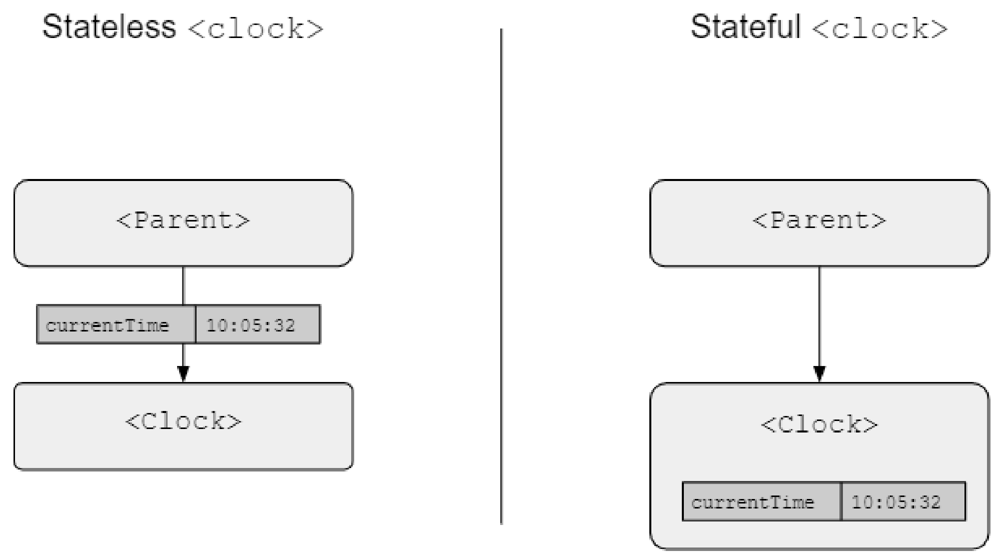


Figure 5.2 The stateless clock needs its parent component to update it every second in order to display the time, whereas the stateful clock can update itself and the parent need not worry about it.

Note that in order for the stateless clock in figure 5.2 to actually work, the parent has to be stateful, because we still need to keep the current time in a state *somewhere*. Of course, we could make the parent component stateless as well, but then we would have to push the state up yet another component.

5.1.2 Where should I put state?

Okay, so we want our application to be stateful. Now where do we put the state? Normally you would try to put the state as close to the components that need it.

Let's say you have an application that contains a top menu with a (live and functional) clock in it, but also a main section with many different pages that can update as you navigate around the page and a footer with some static links.

You need the state for the clock to exist somewhere either inside the clock component itself or in any component above it in the tree.

If we design our application as in figure 5.3 you have a choice of components for where to put your clock state.

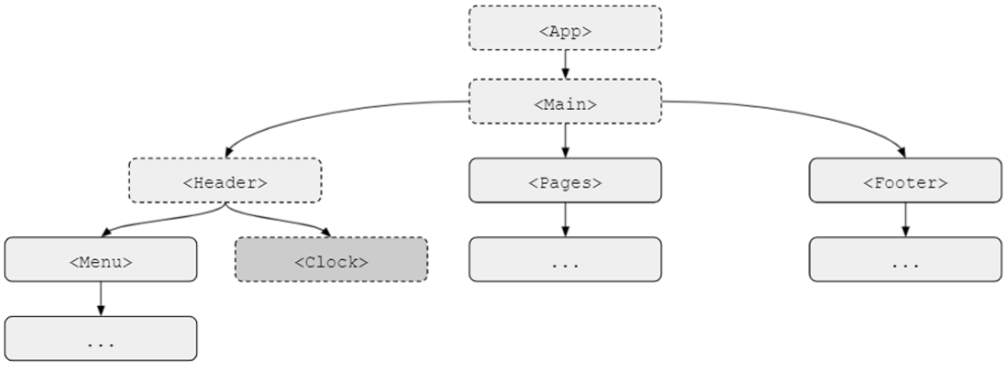


Figure 5.3 Clock state is only required in the component marked in a darker gray. You can put your clock state in any of the components with a dashed border. These are the clock component and its ancestors.

In this example, it makes sense to put the clock state inside the clock component itself. No other component has a need to know about the current time, so we just have the state localized to the component that needs it.

On the other hand, let's say we also need to keep the state of which page is currently displayed in the application. This information is required both in the Pages component, because it actually needs to display the active page, but also in the Menu component, because it needs to display the link to the current page with a highlighted background.

If we examine the document tree in figure 5.4, we see that we can put the information in either Main or App, as those are the only two components that contain both the components that require the state we care about.

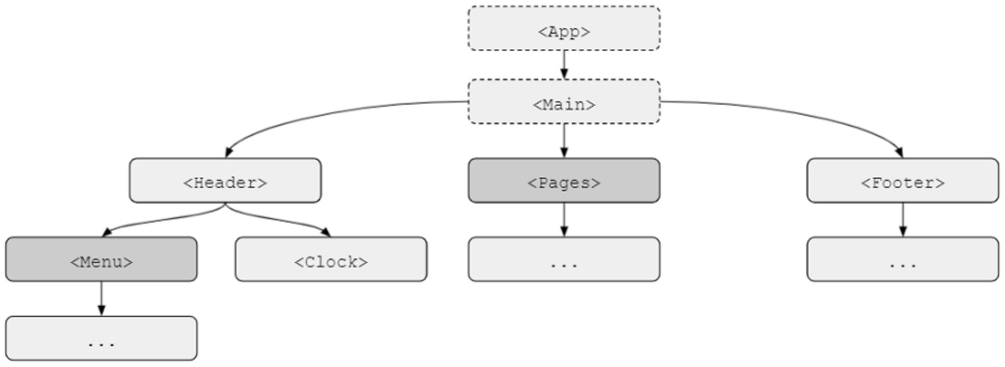


Figure 5.4 Current page state is required in the two components marked in a darker gray. You can then put your current page state in the two components marked with a dashed border. These are the common ancestors of two target components.

Whether you actually decide to put your current page state in Main or App is up to you. It's probably a matter of taste or personal preference. There's a practical argument as to keep state as "low" in the document tree as possible (which means to put it in Main), but that component might already have a ton of other responsibilities, so it might make sense to put this information in the parent App for organizational purposes.

5.1.3 What kind of information do you store in component state?

In general, any state used in a web application belongs to one of three categories:

- Application data,
- UI state,
- Form data.

This is not a rule or an artifact of React in particular, but is a consequence of how stateful applications operate.

Different types of data are stored in different ways. We will cover each of these to talk about how to store and use the data appropriately.

There might be other categories of component state, but most of them fall within one of the three above categories.

APPLICATION DATA

Application data is the data the user is working on, updating, or reading. If you are building a web application, where users can log in, the information about the user is application data. If the user can log in and see available classes in the gym, book a class, etc, all of that data is application data as well.

Application data is most often stored on a global level in your component. If you have a component that displays gym classes, then it would be possible to store the list of available classes locally in this component, but that would also mean that all the information about the available classes would be lost when the component unmounts, and they would have to be re-loaded from the server when the component later re-mounts.

A better solution is often to create a data store in a component that is persistent in your application so that when data is loaded once, it remains through the application. We will see different ways of doing this in the future both involving built-in React functionality (using Context) as well as with external libraries (using Redux and react-query).

UI STATE

UI state refers to the current state of UI components. In general, this is intermittent data that is not persisted but just helps the web application render the correct elements in the correct way. It can be things like which tab is currently active, whether a panel is collapsed or not, is the menu open or not, etc.

UI state values are most often kept as local as possible. The information about whether the menu is open or not is only of relevance inside the menu component, so you can easily store this as local state inside this component only.

FORM DATA

As you will see in chapter 8, form data is another very common use-case for component state.

While the user is interacting with a form, entering data, moving from one form field to the next, the current state of the form is often kept in local state in the component that covers all the form fields.

5.1.4 What not to store in state

A number of things should never be stored in state.

This includes:

- Values that don't change. This is not just constants like magic numbers, but also configuration values loaded in at application start. If it can't change, don't make it variable.
- Copies of other state values. You should try to keep a single source of truth. If you have some data in a global state in your application, it will get messy if you also keep it in a local state in a different component (unless you're locally allowing the user to update the data there in a form).
- Duplicates of the same data. If you have two versions of the same data in state, you might want to consolidate that data. For instance, if you have both first name, last name, and full name in state, you will have to update at least two of these values every time one of the changes. It would be a lot better to only keep the source values, first name and last name, in the state and calculate full name as needed based on the state.

There are of course many more things you should never put in state (for instance, your car keys), but that list is too long to write out. The above are common pitfalls that you might think about doing, but probably shouldn't.

5.2 Adding state to a functional component

We've so far discussed why, where and what to keep in component state, but we still don't know how to actually do it. Keeping state in a functional component has a surprisingly simple API which is both the main attraction, but sometimes also a headache. It's a very low-level API, so you might have to add some functionality to get a smooth developer experience, but on the other hand, it allows you to make simple cases of stateful components very, *very*, easily.

Let's jump right in and see it in action. Let's create the simplest possible stateful component, a click counter component. What we need is a way to initialize our counter, display the current value, and increment the counter every time we click a button. However, there is one very important last step. We can't just simply update any old variable and hope that the component renders correctly. We need to let React know that a value has been updated, and to do that, we need to go through the React-specific API.

Refer to figure 5.5 for a simple flow chart.

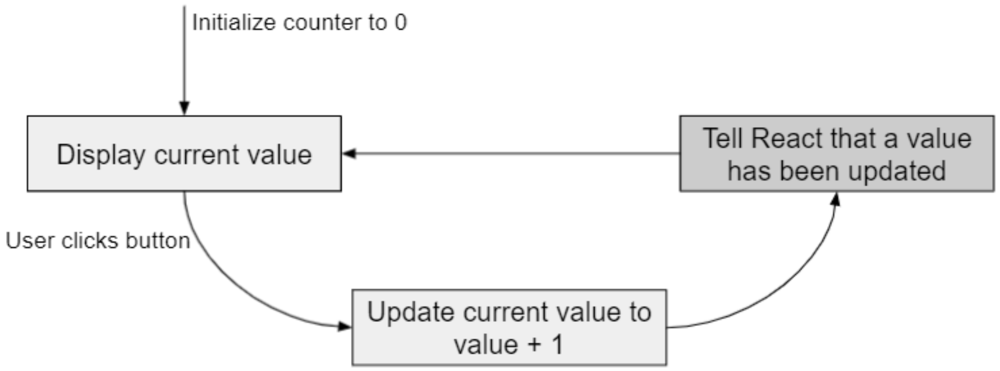
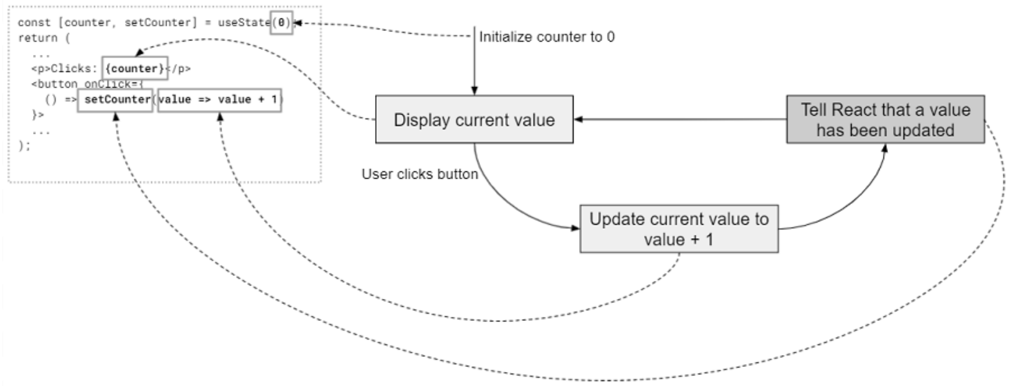

Initialize counter to 0
Display current value
Tell React that a value has been updated
User clicks button
Update current value to value + 1

Figure 5.5 The flow of state in our counter component. We initialize the variable, display it, and on button click, we increment the value and make sure React knows to update the component to display the new value.

In order to do this in a functional component, we need to use a function from the React package named useState. It takes an initial value and returns the current state and an update function. Let's add in the relevant bits of React specific API we need to do these things in the diagram as in figure 5.6.



```
const [counter, setCounter] = useState(0);
return (
  ...
  <p>Clicks: {counter}</p>
  <button onClick={
    () => setCounter(value => value + 1)
  }>
  ...
);
```

Initialize counter to 0
Display current value
Tell React that a value has been updated
User clicks button
Update current value to value + 1

Figure 5.6 The flow of state and the lines of code, that refers to each action in the chart. The dashes arrows translate the concept to the actual piece of code implementing the given goal.

Let's see the code in the full listing in listing 5.1.

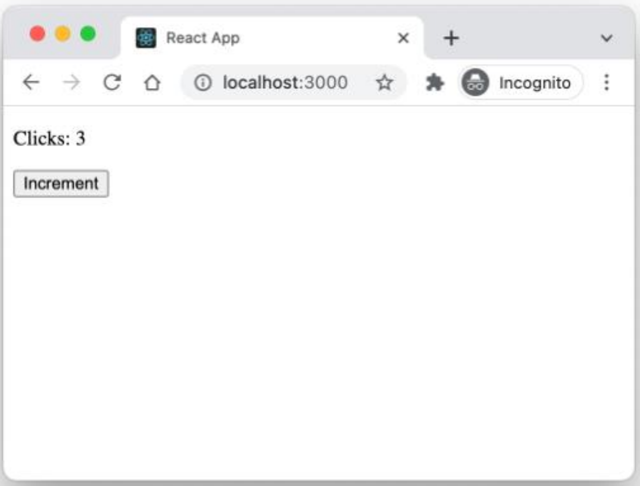
Listing 5.1 A fully functional counter.

```
import { useState } from 'react';    #A
function Counter() {
  const [counter, setCounter] = useState(0);    #B
  return (
    <main>
      <p>Clicks: {counter}</p>    #C
      <button onClick={() => setCounter(value => value + 1)}>    #D
        Increment
      </button>
    </main>
  );
}
```

#A We have to import the function useState from the React package
#B Here we init a new state with an initial value and get the current value and a setter function back
#C We display the value through the current state
#D We update the value through the setter function

rq05-functional-counter

Let's go ahead and run this in the browser right away and get clicking as we have in figure 5.7.



Clicks: 3

Figure 5.7 The counter in action after only 3 clicks. But feel free to keep going—the sky's the limit. Or actually 9,007,199,254,740,991 is the limit, but you probably won't get that far.

There's a lot to cover here, so let's go over these stepsone by one:

- Import the function `useState` from the React package
- Call `useState` in the functional component and supply an initial value
- Destructure the response from calling `useState` as two array elements
    - The first element is the current value
    - The second element is a setter function
- Then use the current value however you see fit
- When you want to update the state, simply call the setter with either a function or a plain value

We'll cover each of these steps one at a time in the next subsections. We'll also discover how you can use multiple `useState`'s to create more complex components.

Oh, and did we mention: `useState` is a hook. The first and simplest of the new React hooks that came in React 16.8 and changed everything. Hooks are special functions that you can't treat like any other function. We'll cover some of that in this section, but will go even deeper into the topic of hooks in the next chapter.

5.2.1 Importing and using a hook

`useState` is a hook. Hooks are an umbrella term for a new kind of special function that exists in React 16.8 and forward. React comes with a number of built-in hooks, and they are hooks because React says so. They don't do the same things nor provide overlapping functionality, but are all "hooks" into the React core functionality and require special attention to work correctly.

The fact, that `useState` is a hook, is actually very easy to see, because the function starts with the word `use*`. In modern React it's now a convention, that any function that starts with the word `use*` is a hook and no non-hooks should ever start with that word.

So, what's so special about hooks? Hooks are named like that, because they are hooks *from* your component *to* the "insides" of the React machinery. You can do some magic things that aren't possible without having this extra access. A functional component is just a function, so it can't really do much beyond controlling a single render if we don't have this deeper access.

React comes with 10 hooks (at time of writing), that are low-level units, that can be combined to create all sorts of advanced components. New built-in hooks can be added to the React API over time, so by the time you're reading this, there might be more than 10.

You can create your own custom hooks on top of the React hooks. If you do, you should name your custom hooks `use*` as well. E.g. we could have created a `useCounter` hook for the above component. We'll cover custom hooks in chapter 9.

RULES OF HOOKS

When you use a hook in a component, you must always use that hook. And you must use the exact same hooks in the exact same order *every time* you render the component.

This might seem weird, but it's required by React to be able to make your function work correctly.

Imagine a variant of the counter component where we pass a property to the component to indicate whether it should be visible at all. You might think that we would be able to do something like figure 5.8.

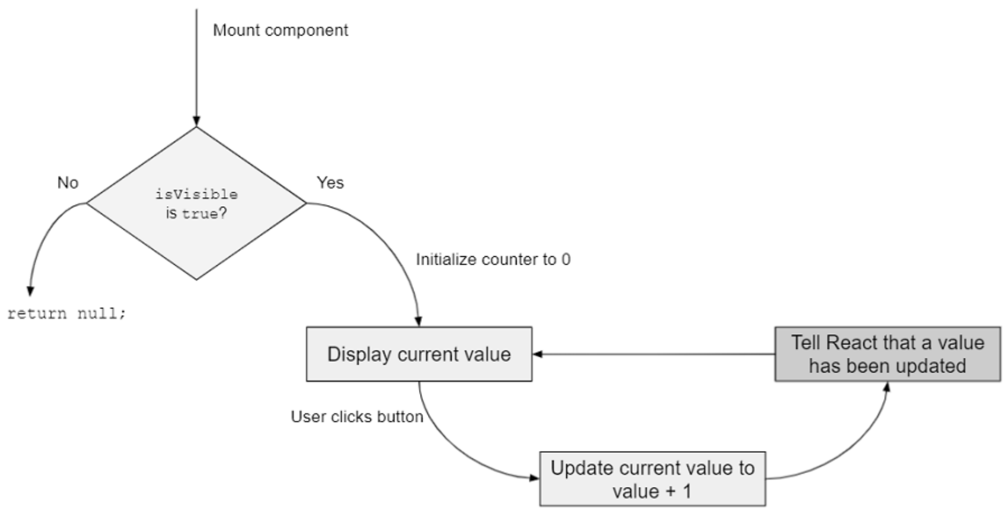


Figure 5.8 Can we check the property first and if false, simply ignore initializing the state altogether?

We can go ahead and implement this:

```
function Counter({ isVisible }) {
  if (!isVisible) {
   return null;    #A
  }
  const [counter, setCounter] = useState(0);    #B
  return (
    ...
  );
}
```

#A If isVisible is false, we just return null from the very start
#B Only if isVisible is not false, do we actually initialize the state using the useState hook. This is wrong!

We'll, that's not allowed! What's wrong here? The hook is not called every time. If the property `isVisible` is set to true in one render, the hook *will* be called, but if it's set to false in the next render, the hook *will not* be called. And that's not just bad, it will completely break your React application. React will throw an error message something along the lines of:

```
React Hook "useState" is called conditionally. React Hooks must be called in the exact same
       order in every component render. Did you accidentally call a React Hook after an
       early return?
```

That's a built-in restriction in the React API, that we simply must obey. Hooks must always be called in the exact same order in every render.

For this reason, you will need to sometimes make what looks like sub-optimal code. You need to put all your hooks before any attempt to return anything in the component as in figure 5.9.

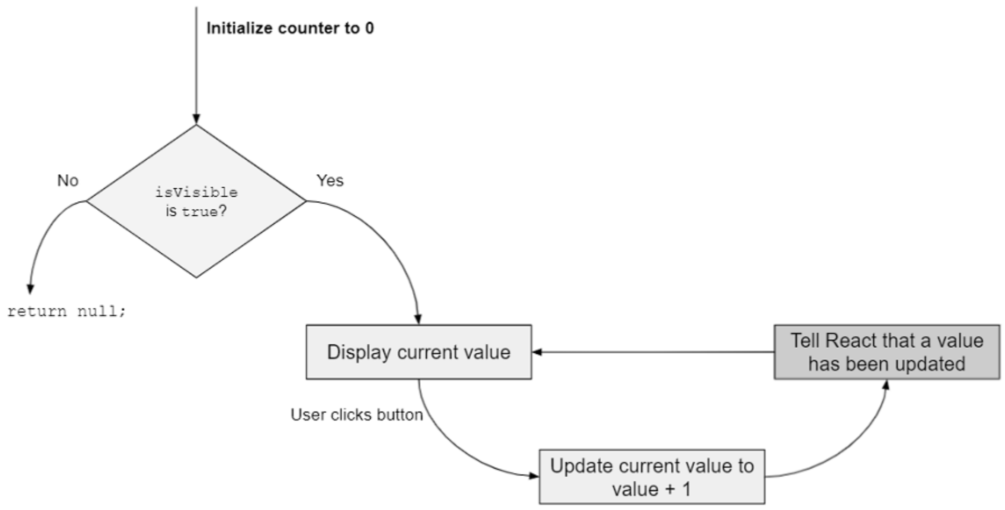


Figure 5.9 We have to initialize the state before optionally aborting rendering even if we don't need the state at all.

Let's implement this:

```
function Counter({ isVisible }) {
  const [counter, setCounter] = useState(0);                    #A
  if (!isVisible) {
   return null;                                                 #B
  }
  return (
    ...
  );
}
```

#A We initialize two variables here, that we might never need
#B Only after all our hooks have been executed, can we return something

This also means that you can never conditionally run a hook (e.g. inside an `if` block), you can never run a hook a in a loop (because that would mean you might have a varying number of hook calls), and you can never call a hook inside a callback or event handler (it has to run directly in the component body when called).

We will see some examples of some of these restrictions in the next sections and how you can work around those restrictions to still achieve the desired goal.

We will also cover a lot more about hooks in the next chapter, where we go a lot more under the hood of hooks and how they must be used.

5.2.2 Initializing the state

When you call `useState`, you must supply an initial value. If you do not pass an initial value, the initial value is assumed to be `undefined`. Only the value that you pass to `useState` the very first time around for each component instance matters. When you hook re-renders for whatever reason, the initial value is ignored.

The most obvious use case for this is just setting up a baseline in your component. When it mounts the first time, what should the state be? If it should be some dynamic value passed in as a property, use that property. If it should be any static value, write that. In 99% of the cases, you will set your initial value to either a static value (including null very often) or a property. We will cover some examples of initialization in the rest of this section.

INITIAL VALUE

Every state has an initial value. Our counter had an initial value of 0, but it didn't need to be 0 of course. We could have initialized it to 10 or 100 or even some dynamic value.

Let's say we want to create a variant of the counter, where we can initialize the value to some property that we pass in. We will then create an application with three different instances of this counter initialized with different starting values. The resulting component tree will look like figure 5.10.

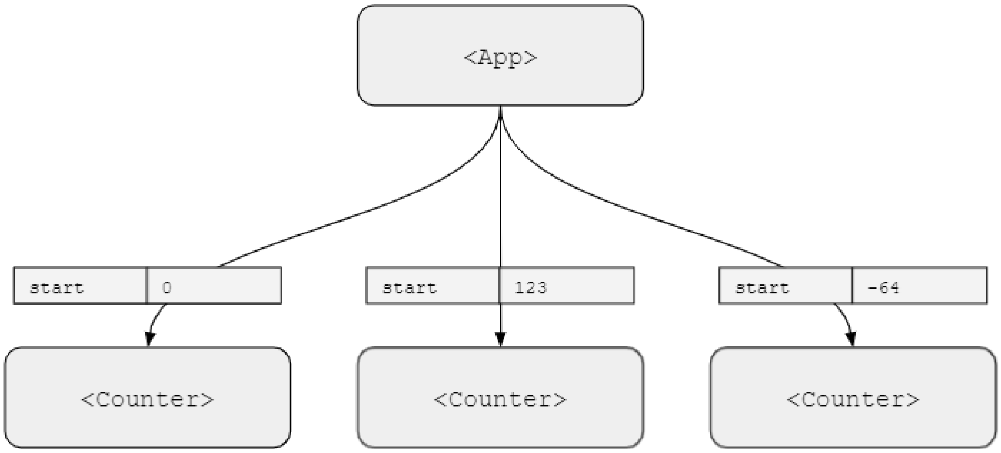


Figure 5.10 We now want to have three counters initialized to different starting values because it looks cool.

We can go ahead and implement this in listing 5.2.

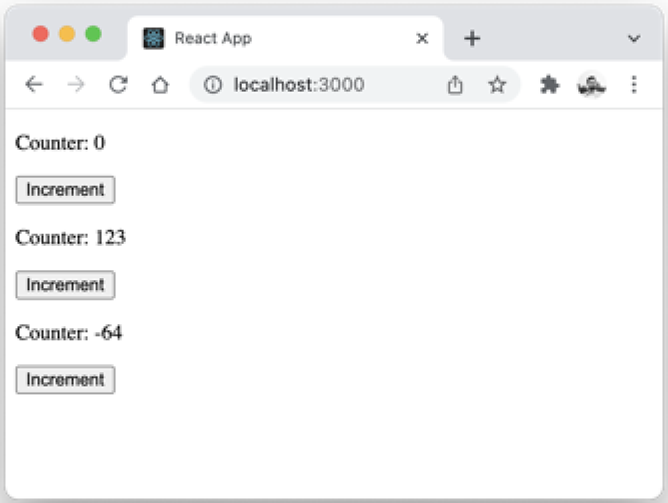
Listing 5.2 Triple counters.

```
import { useState } from 'react';
function Counter({ start }) {    #A
  const [counter, setCounter] = useState(start);    #B
  return (
    <main>
      <p>Counter: {counter}</p>
      <button
        onClick={() => setCounter(value => value + 1)}
      >
        Increment
      </button>
    </main>
  );
}
function App() {
  return (
    <>
      <Counter start={0} />    #C
      <Counter start={123} />    #C
      <Counter start={-64} />    #C
    </>
  );
}
```

#A The property passed to this component is named start
#B We use that property to initialize our state
#C Three instances of the counter with three different start values

`rq05-triple-counter`

The result in a browser will look like figure 5.11.



Counter: 0

Increment

Counter: 123

Increment

Counter: -64

Increment

Figure 5.11 Our three counters before we start clicking any of them.

Other common static initial values besides numbers include:

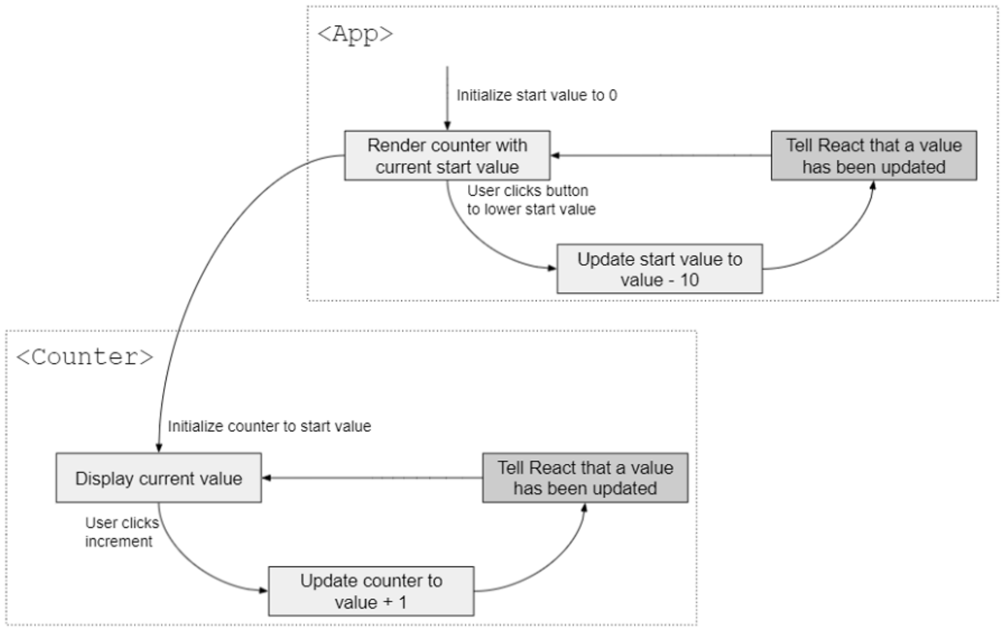
- `true` or `false` for booleans—if your menu is hidden until a button is clicked, the `isMenuVisible` state is initialized to `false`).
- the empty string, `""`—if you have an input for a login email address, you will initialize your state to the empty value so the input is empty until the user starts typing.
- `null`—if you have a complex value that hasn't been set to anything yet, `null` is the perfect placeholder value indicating that no value exists yet.

The most common dynamic initial value is to use a property. We did that with the counter above. Similarly you would use a property if you had a component to change your name. You would pass in the current name as a property and initialize your state based on that.

You could also initialize your state to a value from a cookie or similar local storage. For a login form, you could initialize the email address state with the last known email address used in this same form as stored in a cookie.

ONLY THE FIRST INITIAL VALUE MATTERS

Let's make a new variant of the counter with a variable start value. This time we want to add a new button outside the counter, that will change the start value of the counter. So instead of just initializing the value to 0 every time, we have a button that if pressed will lower the start value of the counter by 10. Let's try to illustrate this in a flowchart in figure 5.12.



<App>
Initialize start value to 0
Render counter with current start value
Tell React that a value has been updated
User clicks button to lower start value
Update start value to value - 10
<Counter>
Initialize counter to start value
Display current value
Tell React that a value has been updated
User clicks increment
Update counter to value + 1

Figure 5.12 We now have state in both the app and the counter – and we want to use the app state to initialize the counter state.

We are actually creating a stateful component on top of our stateful component.

Rather than just blindly go ahead and implement this, let's think through some scenarios. What happens in this scenario:

1. At first the counter will be initialized with a start value of 0
2. We then press the button to lower the start value, so the start value should now be -10
3. Does the counter then update to -10?

Let's expand with another scenario:

1. At first the counter will be initialized with a start value of 0
2. We then press increment in the counter to increase the counter value to 1
3. We then press the button to lower the start value (which was 0), so the start value should now be -10
4. Does the counter then update to -10? -9?

In fact, both of these scenarios are meaningless. That is because, as we have briefly mentioned, only the very first value passed to `useState` is used as the value for the state. If the initial value changes in a subsequent render, the state never updates. This is both good and bad. It's good because otherwise our counter would always be the same value as we keep passing the same value in every render.

In this instance, it wasn't actually clear what we wanted to happen if we started lowering the start value after we had begun counting. It is important to figure out exactly what we want to happen before we try to implement it in code.

It is possible to have a state value update based on a property, but that requires other hooks that we will only introduce in the next chapter. In particular, the `useState` hook.

INITIALIZER FUNCTION

There are times where you want to set the initial value to the result of some calculation. Let's say that you have a password input that you want to initialize to a good, strong password, but once the user starts typing, you just use whatever the user types. We have an expensive function somewhere else in our codebase, `generatePassword`, that we will use to create this initial password. Let's go ahead and diagram this as earlier in figure 5.x.

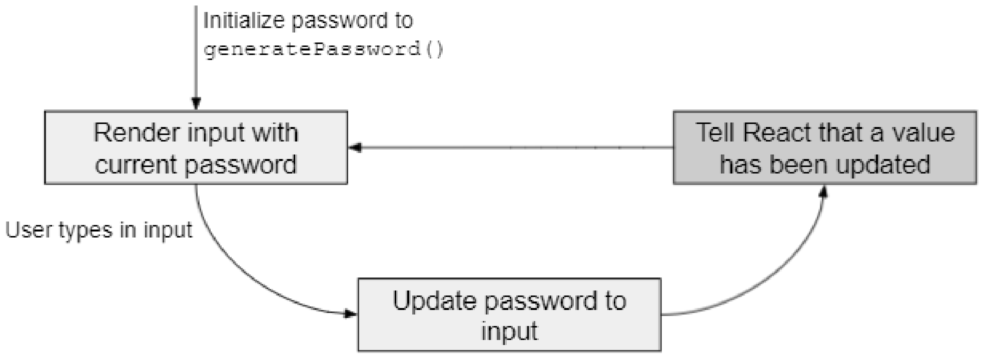


Figure 5.13 The state flow when we use a function to generate the initial value

If we implement this using the initial value, we get something like:

```
function Password() {
  const [password, setPassword] = useState(generatePassword());
  ...
}
```

However this function `generatePassword()` will actually be called on every render (because it is executed on every render), while the return value will be ignored on every render but the very first, as just explained. It might be a very complex function that runs a lot of expensive algorithms, so we should avoid running it if we don't need the returned value.

For this purpose, the initial value can instead be a function that returns the initial value. In such a setup, the initial value function will only be invoked the first time around and it will be ignored for future renders. This would be as in figure 5.14.

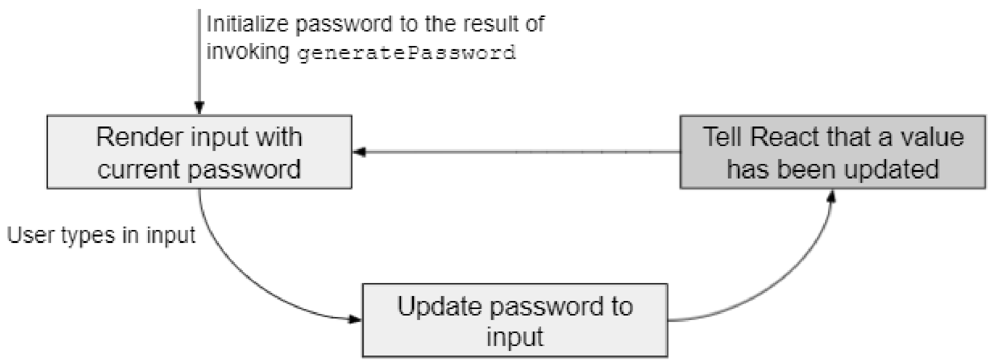

Initialize password to the result of invoking generatePassword
Render input with current password
Tell React that a value has been updated
User types in input
Update password to input

Figure 5.14 We don't actually call generatePassword in this instance. We rather instruct the hook to call the function itself only when it needs to (which is only the first time around).

We can do that generally as:

```
const [password, setPassword] = useState(() => generatePassword());
```

Or in this instance much simpler as just:

```
const [password, setPassword] = useState(generatePassword);
```

Because `generatePassword` is already a function, we can just pass it as is. However, if the function took an argument, perhaps the length of the generated password, we would have to use the former form:

```
const [password, setPassword] = useState(() => generatePassword(12));
```

INITIALIZING TO A FUNCTION

What if your state is a function? If we pass a function to the initial value, it will be called, so how can we store a function as the initial value? We make another function that returns the first one. It sounds a bit weird, but it actually can make sense.

Let's say that we have a calculator component and we can do some mathematical operation on two values entered in two input fields (e.g. addition, subtraction, and multiplication). This calculation is a function that takes two values and returns a single response. We can implement this as an enum-like type as follows:

```
const OPERATORS = {
  ADDITION: (a, b) => a + b,
  SUBTRACTION: (a, b) => a - b,
  PRODUCT: (a, b) => a * b,
};
function Calculator() {
  const [operator, setOperator] = useState(OPERATORS.ADDITION);
  ...
}
```

This looks pretty good, but it doesn't work. What we have done is described in figure 5.15.

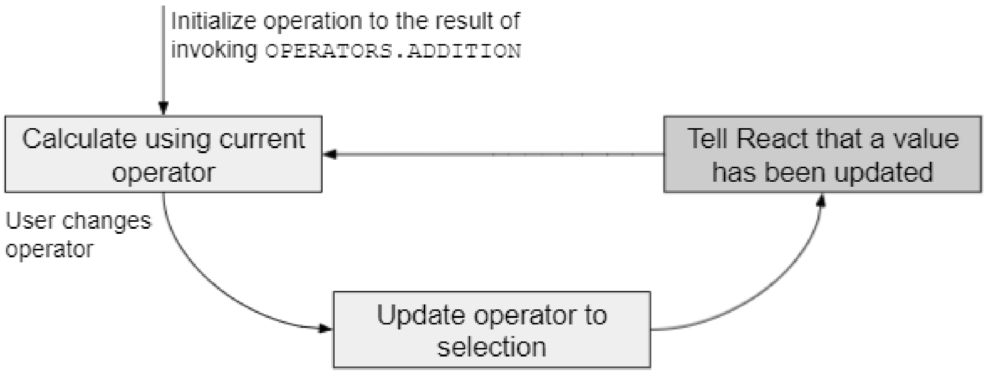

Initialize operation to the result of invoking OPERATORS.ADDITION
Calculate using current operator
Tell React that a value has been updated
User changes operator
Update operator to selection

Figure 5.15 Because we pass in a function as the initial value, React treats it as an initializer function and invokes it—just like before.

If you type the above into a component you would have the operator initialize to the value `NaN`. And that's of course because `useState` is invoked with a function as initializer, so it calls the function, but that function doesn't know what to do without arguments, so it just returns `NaN`.

What we need is a function that returns the operator as in figure 5.16.

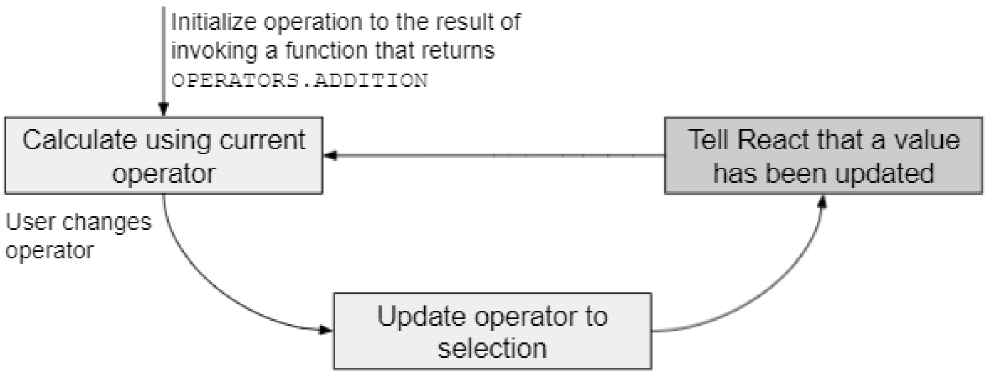

Initialize operation to the result of invoking a function that returns OPERATORS.ADDITION
Calculate using current operator
Tell React that a value has been updated
User changes operator
Update operator to selection

Figure 5.16 This time we will still pass in a function as the initializer value, but that function will then in turn return our desired value (which happens to be function).

We can go ahead and implement that as:

```
function Calculator() {
  const [operator, setOperator] = useState(() => OPERATORS.ADDITION);
  ...
}
```

This works and is a perfectly fine construction. You will see this construction repeated when we talk about the setter function in a bit.

5.2.3 Destructuring the state value and setter

When we need a stateful component, we use the useState hook. This hooks returns a value, that we have destructured into a state value and a setter function like this:

```
const [value, setter] = useState(initial);
```

This is as close to mandatory as it gets. There are other ways to do it, but everyone uses the useState hook in this manner. If you do the same thing, your code will make sense to other developers. This is simply a convention that is necessary when using this hook. Other hooks work in similar ways, and you just have to get used to this notation.

THE USESTATE RETURN VALUE

The `useState` hook return value is a bit cryptic. The hook returns an array with two elements. The first element is the current value of the state and the second value is the setter function. There are many ways we could "accept" this return value and change it to our use.

We could just store the returned array in a variable and address the two items as `value[0]` and `value[1]` respectively or copy those elements to two other variables. But the recommended and most common way is to destructure the array directly in the assignment of the return value to a variable and name the two returned values as we see fit:

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

At this point, for most React developers this is just instinctive. This is how you use the `useState` hook and after a while you don't really think about it. The only thing to think about here is the naming of the two destructured variables.

The common approach is to name the state value after what we store in it, and name the setter function the same, but with a `set*` prefix. This is what we do in the above with `counter` and `setCounter` respectively.

Teams will often come up with their own naming standards or apply those from others, but the above is a safe default. The only potential deviation is when it comes to boolean state values. You might have a state value called `isCollapsed`. The setter function would then be called `setIsCollapsed`, which just sounds like terrible English, so some might just call it `setCollapsed` and skip the boolean prefix of is* or has* that boolean variables often have.

The next subsection explains why this works and why we do it like this. This is only necessary for you to read, if you really want to know. If not, just accept that this is the convention. You can always return to this part, if you get curious later.

THAT'S WEIRD THOUGH

Okay, we get that. But.. why? Why does `useState` return an array with two unrelated values? It's clearly not a list of something!

Imagine that you are a React core developer and you are creating the `useState` hook. The `useState` function needs to return two values. One value is the current state and it can be any type. The second value is the setter function, and it's a function that either takes any value or even an updater function.

JavaScript doesn't have tuples or structures, where you can structurally combine different types in a "nice" way. Some might think that we could return an object with the two properties, and we would just have to agree on their naming - it could be e.g. `obj.value` and `obj.set`. These would also destructure well as simply:

```
const { value, set } = useState(0); // This doesn't actually work!
```

But as you're inclined to have multiple states in the same component, you would have to rename them often. Even if you only have a single state, you might still want to have a more descriptive name. And destructuring the named properties in an object to different local variables is just more verbose than doing it for an array:

```
const {
  value: counter,
  set: setCounter,
} = useState(0); // This still doesn't work
```

That's a lot of extra typing and unnecessary overhead. So rather than returning a more well-defined object with the two named properties, the React developers went with the array for ease of use.

Because this `useState` function is so well-known to React developers and because you use it many times in your daily workflow, the unusual syntax just becomes muscle memory and you don't really think about it. But we do agree, it is actually a bit weird.

5.2.4 Using the state value

Imagine our counter from earlier. What would happen if we changed the increment button to rather than increment the value, instead just set the value to the string `"hi there"`. So, it's not a number anymore, but a string. This would look like figure 5.17.

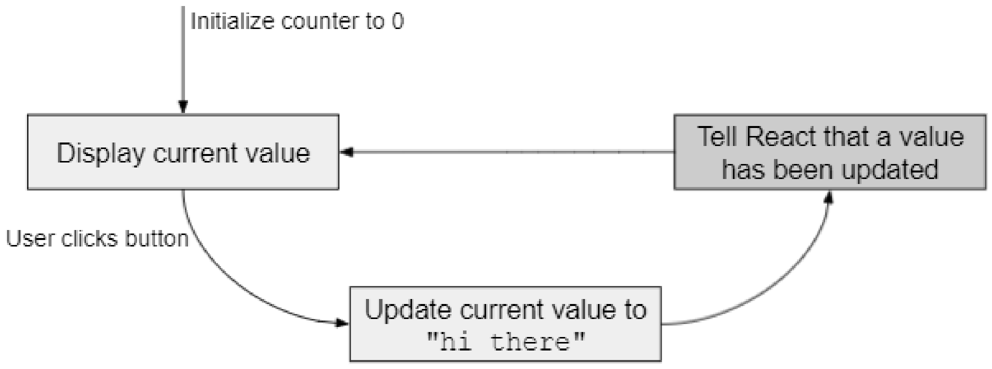

Initialize counter to 0
Display current value
Tell React that a value has been updated
User clicks button
Update current value to "hi there"

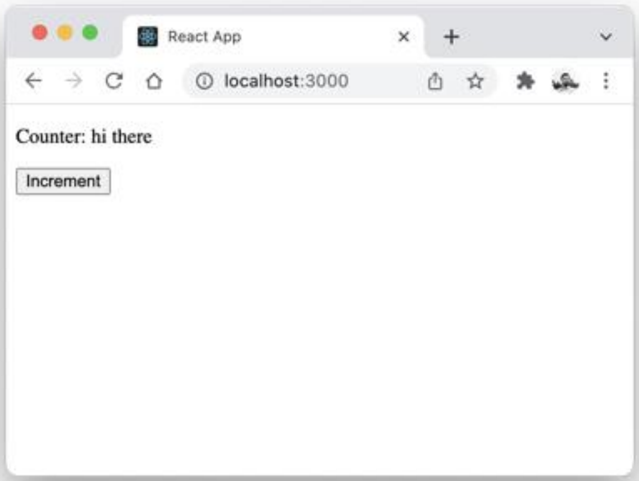
Figure 5.17 We set the counter value to a string, when we click the button.

Let's try to implement that:

```
import { useState } from 'react';
function Counter() {
  const [counter, setCounter] = useState(0);    #A
  return (
    <main>
      <p>Clicks: {counter}</p>    #B
      <button onClick={() => setCounter('hi there')}>    #C
        Increment
      </button>
    </main>
  );
}
```

#A We initialize the counter to a number
#B Here we display whatever the counter currently is
#C On click we change the value of the counter to a string

This actually works. If we click the button, the result will look like figure 5.18.



Counter: hi there

Figure 5.18 Our "counter" value is now a string, but it is still displayed because we don't actually check if it's a number.

It's pretty nonsensical to change the type of a state, just like it is nonsensical to change the type of any other variable. We definitely don't recommend doing the above!

The state value returned by the `useState` hook is whatever you set it to. You can change the type, the complexity, etc. You have full control over the value. The value will start at whatever you pass in as the initial value and from then on, it will be whatever you pass to the setter function.

Most of the time your state type should not change, however. Just as any other variable in your codebase, keeping the type consistent is a huge help, even though JavaScript does not by itself put any such constraints on you.

You can for instance initialize a value to `null` and later set it to a number, where `null` would represent that you don't know what the number would be yet, so initializing to 0 would be misleading. This could be the case with e.g. an age input. Just because you haven't typed your age yet, doesn't mean that it's 0. This would be a change of type, where the type is initially null but later changes to a number.

You can of course have object literals in the state. That might make sense for related values, that you either always update together or use together.

You might for instance have a loader component that displays loading progress of a file in both percent and in text with bytes loaded out of bytes total:

```
function Loader() {
  const [progress, setProgress] = useState(null);
  const someCallback() {
    ...
    setProgress({ loaded, total });
  };
  if (!progress) {
    return null;
  }
  const { loaded, total } = progress;
  return (
    <h2>{Math.floor(100 * loaded / total)}%</h2>
    <p>Loaded { loaded } out of {total}.</p>
  );
}
```

This is a partial example only, as we don't actually load anything here, so we would need more logic to actually fetch something and check the values. But it is an example of related values stored in a single state value.

In a few sections, we will discuss how you can use multiple states rather than cram all your states into a single value. You should only put multiple values into a single state, when the values are tightly related as in the above Loader example.

5.2.5 Setting the state

Setting the state is fairly straightforward in that it works exactly like setting the initial value with all the same quirks and workarounds.

SETTING TO A STATIC VALUE

Let's create a simple accordion component where you can expand and collapse the contents. We have a headline with two buttons with a plus and minus respectively. Clicking the plus button will expand the accordion and show the contents and clicking the minus button will collapse the accordion and hide the contents. This can be illustrated in a diagram as in figure 5.19.

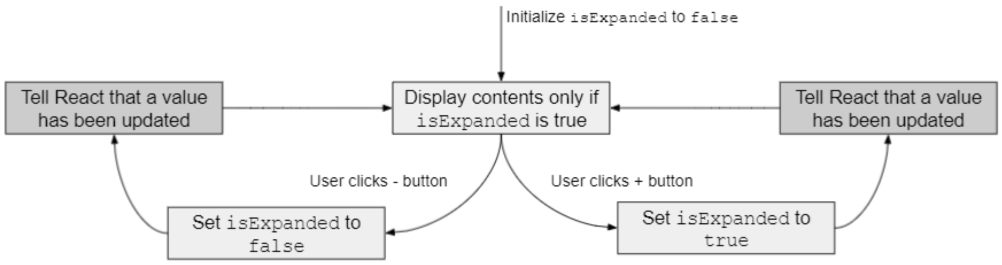

Initialize isExpanded to false
Tell React that a value has been updated
Display contents only if isExpanded is true
Tell React that a value has been updated
User clicks - button
User clicks + button
Set isExpanded to false
Set isExpanded to true

Figure 5.19 The flow chart for an accordion. The boolean is set to true or false depending on which button is pressed.

We can go ahead and implement this in listing 5.3.

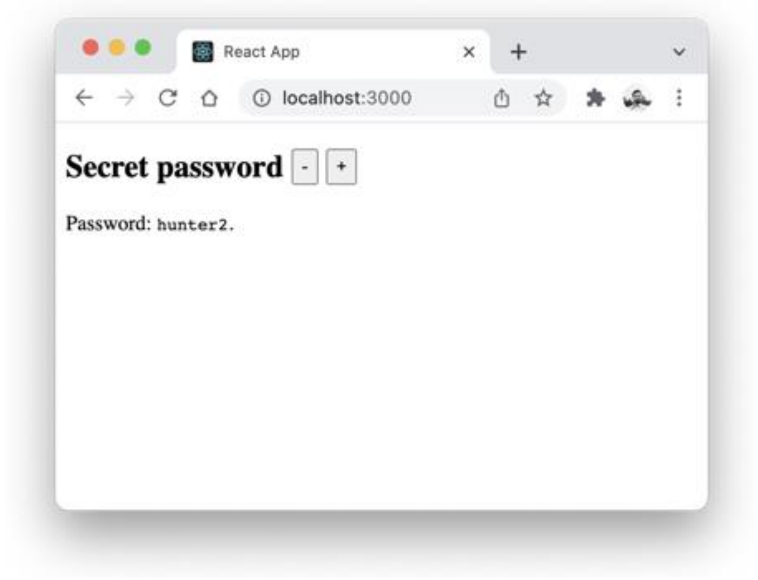
Listing 5.3 A simple accordion.

```
import { useState } from 'react';
function Accordion() {
  const [isExpanded, setExpanded] = useState(false);    #A
  return (
    <main>
      <h2 style={{display: 'flex', gap: '6px'}}>
        Secret password
        <button onClick={() => setExpanded(false)}>-</button>    #B
        <button onClick={() => setExpanded(true)}>+</button>    #B
      </h2>
      {isExpanded && <p>Password: <code>hunter2</code>.</p>}    #C
    </main>
  );
}
```

#A We initialize the state to false
#B When a button is pressed, we invoke the setter with either true or false
#C If the boolean is true, we display the secret accordion contents

`rq05-accordion`

The result in a browser will look like figure 5.20.



Secret password - +

Password: hunter2.

Figure 5.20 After clicking the plus, the secret accordion contents have been revealed.

This component is an example of using the setter with a static value. The minus button always sets the state value to false no matter how many times you press it. Because we set it to a fixed value, we don't need to look at the current value.

Setting using an updater

You can set the value as either a direct value as above or with an updater function that returns the new value. If you use an updater function it will be passed the current state as an argument.

We've already seen examples of using an updater function:

```
const [counter, setCounter] = useState(0);
…
<button onClick={() => setCounter(value => value + 1)}>
```

This updates the value in the state by using a simple increment function that takes an argument and returns the argument + 1.

Setting to a function

If we need to set the state value to a function, we have to use the same workaround as with the initial value. We need a function that returns our operator function.

So if we expand our calculator example from earlier with buttons to change the operator, we would be implementing a full application. Let's first try to diagram the state flow in figure 5.21.

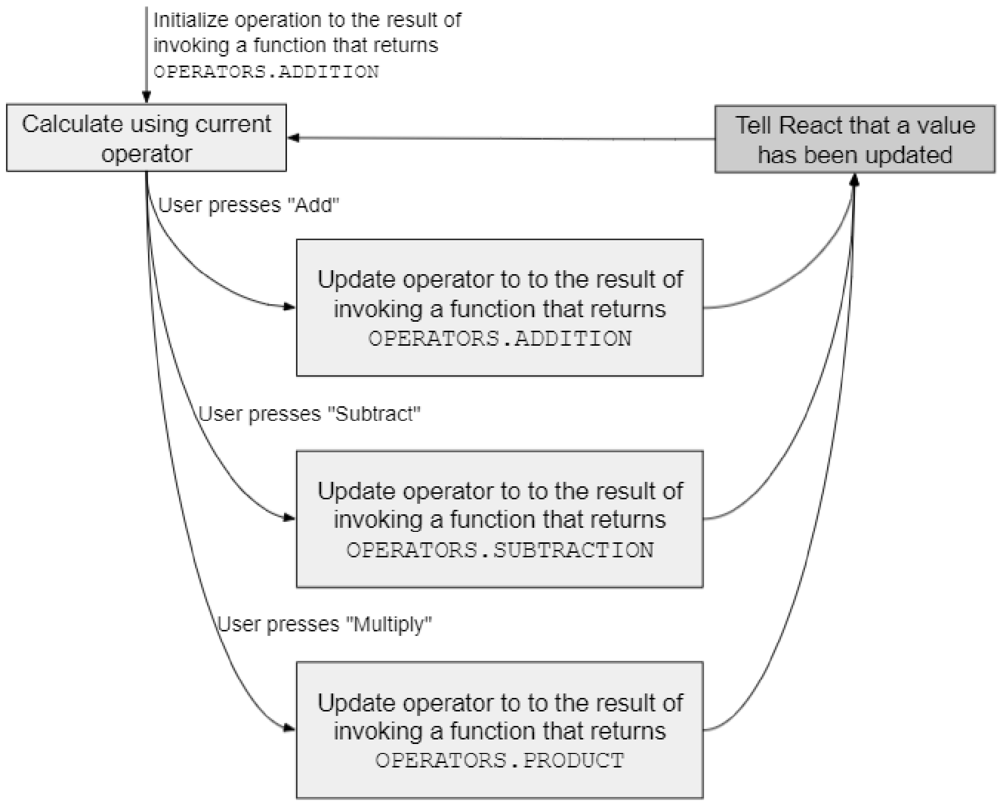

Initialize operation to the result of invoking a function that returns OPERATORS.ADDITION
Calculate using current operator
Tell React that a value has been updated
User presses "Add"
Update operator to to the result of invoking a function that returns OPERATORS.ADDITION
User presses "Subtract"
Update operator to to the result of invoking a function that returns OPERATORS.SUBTRACTION
User presses "Multiply"
Update operator to to the result of invoking a function that returns OPERATORS.PRODUCT

Figure 5.21 The expanded calculator example now has three buttons to change the operator.

Implementing this it would become listing 5.4.

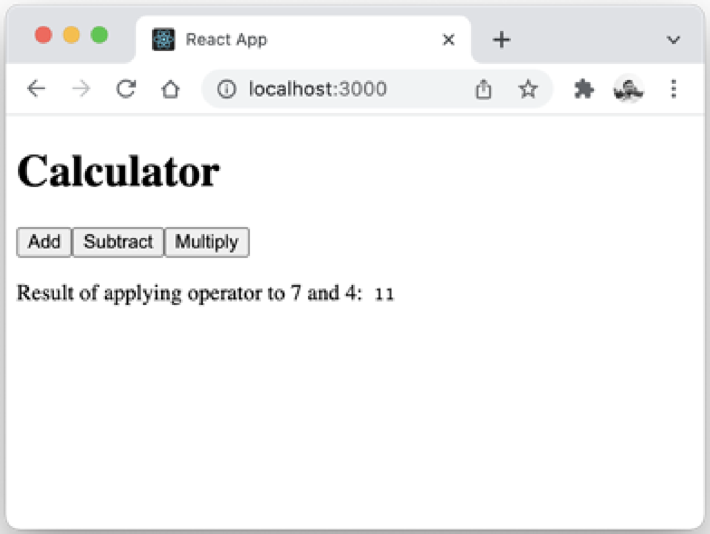
Listing 5.4 Simple calculator.

```
import { useState } from 'react';
const OPERATORS = {
  ADDITION: (a, b) => a + b,
  SUBTRACTION: (a, b) => a - b,
  PRODUCT: (a, b) => a * b,
};
function Calculator({a, b}) {
  const [operator, setOperator] =
    useState(() => OPERATORS.ADDITION);    #A
  return (
    <main>
      <h1>Calculator</h1>
      <button
        onClick={() => setOperator(() => OPERATORS.ADDITION)}    #B
      >
        Add
      </button>
      <button
        onClick={() => setOperator(() => OPERATORS.SUBTRACTION)}    #B
      >
        Subtract
      </button>
      <button
        onClick={() => setOperator(() => OPERATORS.PRODUCT)}    #B
      >
        Multiply
      </button>
      <p>
        Result of applying operator to {a} and {b}:
        <code> {operator(a, b)}</code>    #C
      </p>
    </main>
  )
}
function App() {
  return <Calculator a={7} b={4}  />;
}
export default App;
```

#A We initialize the state with a function returning the default operator function
#B We also update the state with a function returning the clicked operator function
#C We can now call the state value as a function, because we've made sure it is always a function

rq05-calculator

See this fancy (but a bit simple) calculator in action in figure 5.22.



Calculator

Result of applying operator to 7 and 4: 11

Figure 5.22 Our calculator with the default operator, addition. We don't actually display anywhere what the operator is, we just display the result of the calculation of applying that operator to the two operands.

SETTING AND RENDERING

What happens if we keep clicking the add button in the above calculator? Does the component re-render every time, doing the calculation every time? We set the state to the exact same value every time, so why should it?

Actually, it will not re-render. React includes built-in optimization, so `useState` will wait until the end of the current cycle to update the component, and it will check if the value actually changed, and only if it did change, will it re-render the component. That means that there can be situations where you call `useState`, but no re-render happens (because no re-render should be necessary – nothing has changed).

Let's expand our counter from earlier and add a reset button that resets the counter to 0. The state flowchart would now look like figure 5.23.

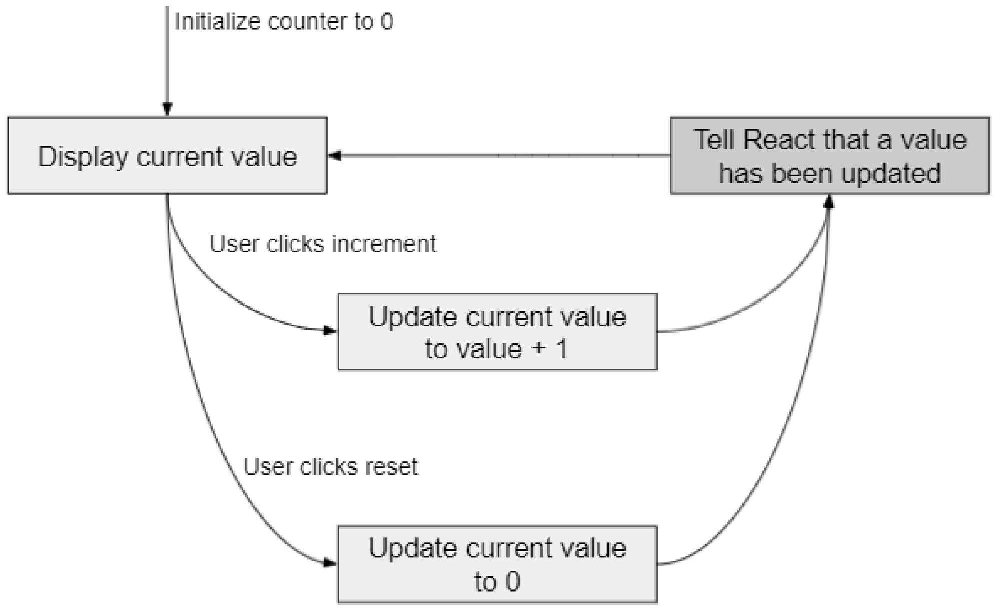

Initialize counter to 0
Display current value
Tell React that a value has been updated
User clicks increment
Update current value to value + 1
User clicks reset
Update current value to 0

Figure 5.23 The new reset button sets the counter to 0 regardless of the old value.

If we implement this, we get Listing 5.5.

Listing 5.5 A Resettable counter.

```
import { useState } from 'react';
function Counter() {
  const [counter, setCounter] = useState(0);
  return (
    <main>
      <p>Counter: {counter}</p>
      <button onClick={() => setCounter(value => value + 1)}>
        Increment
      </button>
      <button onClick={() => setCounter(0)}>                              #A
        Reset
      </button>
    </main>
  );
}
```

#A When you click on reset, the counter is set to 0.

`rq05-reset-counter`

See this new resettable counter in figure 5.24.

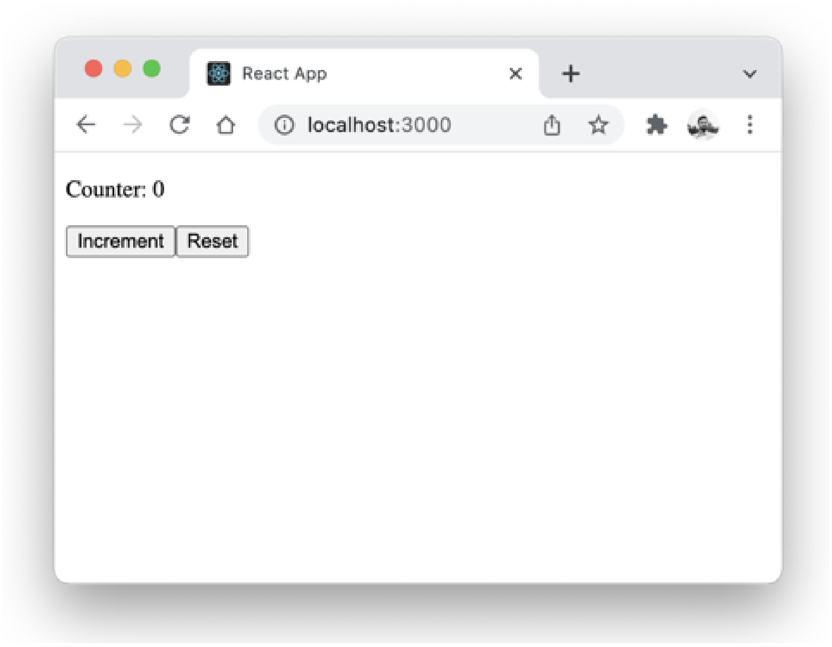


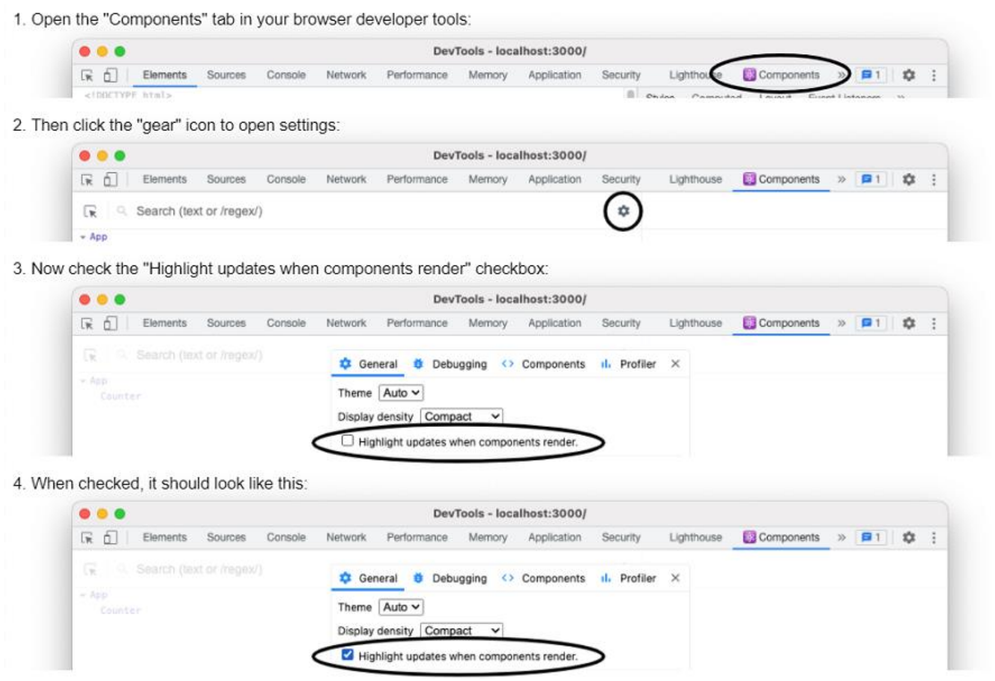
Figure 5.24 A resettable counter that we have just set back to 0.

Clicking reset works. It does reset the value to 0. If we click the button again, nothing happens. But how do we know if the component re-rendered or not? How can you tell?

We're going to use a very useful plugin that is available for Chrome, Firefox and modern versions of Edge, namely the *React Developer Tools*. It is available for download from their respective stores:

- Chrome and Edge: https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi
- Firefox: https://addons.mozilla.org/en-US/firefox/addon/react-devtools/

With this tool, we are able to see when any component renders. Please see instructions in figure 5.25.



1. Open the "Components" tab in your browser developer tools:
2. Then click the "gear" icon to open settings:
3. Now check the "Highlight updates when components render" checkbox:
4. When checked, it should look like this:

Theme Auto
Display density Compact
☐ Highlight updates when components render.
☑ Highlight updates when components render.

Figure 5.25 Go ahead and open up the Components panel for React Developer Tools, open the little "gear" menu and check the "Highlight updates when components render" checkbox.

When we do this and then go back to the resettable counter application and press the increment counter, we can see a blue outline around the entire component flash briefly every time the counter increases. It should look like figure 5.26.

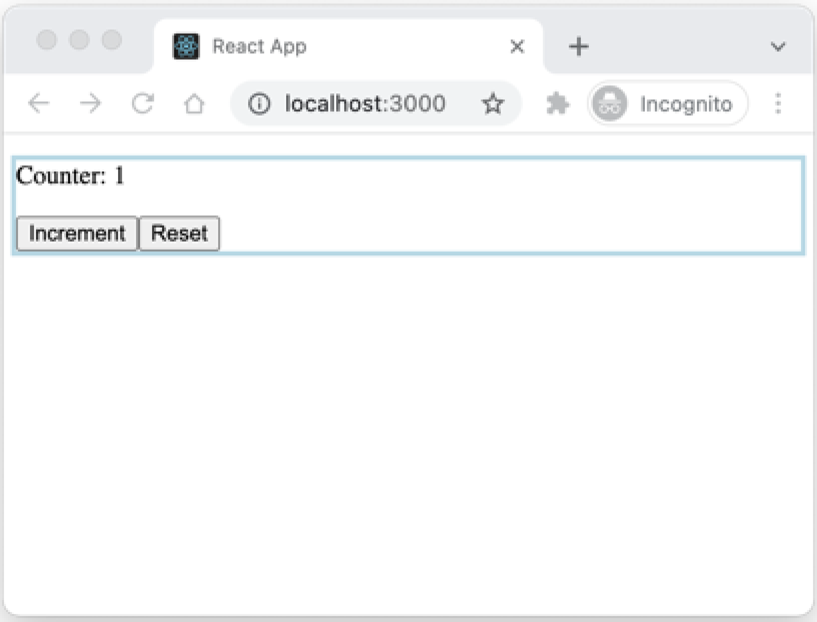


Figure 5.26 The blue outline around the entire component indicates that it has just rendered due to the state changing.

If you press the reset button when the counter is not at 0, you will see the blue outline flash, because the component renders. But if you click the reset button when the counter is already at 0, no blue outline appears. React is smart enough to know that if the state is unchanged, the component output is (or at least should be) unchanged.

SETTING TO THE SAME OBJECT

This above condition on the re-render however also means that if you set the state value to the same object that it already is, even if you changed the object "on the inside", nothing would happen, because there would not be a re-render.

This can for example occur if you have an array in your state. If you manipulate the array in-place and set the same array as the state value again, the component will not render, because nothing has changed (from a referential equality perspective at least).

Let's see this in action and how we can fix this. For that purpose, we're going to build a simple todo-application. We have a list of items we can tick off the list, and as we tick off an item, we remove it from the array and then render the list again.

The wrong way to do this, is to set the state to the same array every time. It does not matter if we edit the array before setting it as state again, because React does not look inside our state value, but only at the reference. We can sketch this wrong approach in figure 5.27.

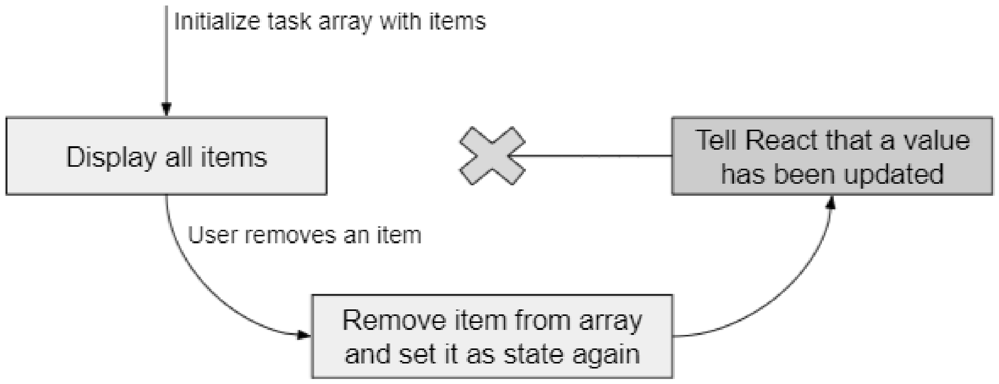

Initialize task array with items
Display all items
Tell React that a value has been updated
User removes an item
Remove item from array and set it as state again

Figure 5.27 The wrong way to use an array as a state value is to set the state to the same array every time. The problem here is, that React will actually not see that anything changed, and it will not cause our component to re-render.

While we now understand that this is wrong, let's try to implement it anyway, so we can also see that it actually does not work for real. We can do that in listing 5.6

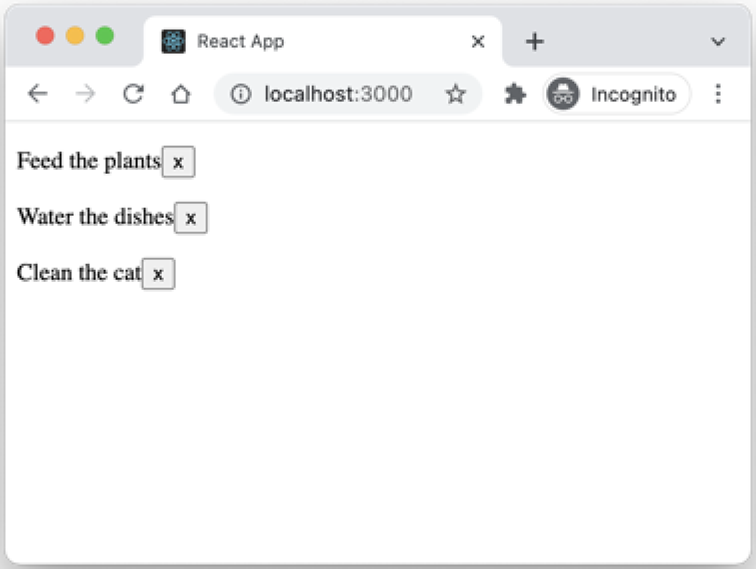
Listing 5.6 A broken todo-list application.

```
import { useState } from 'react';
function TodoApplication({initialList}) {
  const [todos, setTodos] = useState(initialList);
  return (
    <main>
      {todos.map((todo, index) => (
        <p key={todo}>
         {todo}
         <button onClick={() => {
           todos.splice(index, 1);    #A
           setTodos(todos);    #B
         }}>x</button>
        </p>
      ))}
    </main>
  );
}
function App() {
  const items = ['Feed the plants', 'Water the dishes', 'Clean the cat'];
  return <TodoApplication initialList={items} />;
}
```

#A We modify the array in place
#B Add then we try to update the state to the same value it already has (which we changed though, right?)

rq05-bad-todo

Let's try this out and click those delete buttons in figure 5.28. Nothing happens when you click. If you try enabling update outlines in the React developer tools, you will see that the component doesn't actually re-render because the state value is identical by reference, even though it might have been "updated".



Feed the plants x

Water the dishes x

Clean the cat x

Figure 5.28: Our functional todo-list application looks right, but doesn't work–nothing happens when we click the buttons.

For this reason, it is not merely recommended that you don't mutate state directly, it is absolutely necessary. Doing this correctly requires setting the state value to a new array, which is a duplicate of the old one but without the spliced item. One way of doing it could be like this using the spread operator on a slice of the array before and after the item to be deleted. We can fix the model so it becomes figure 5.29 by creating a new array and setting *that* as the new state.

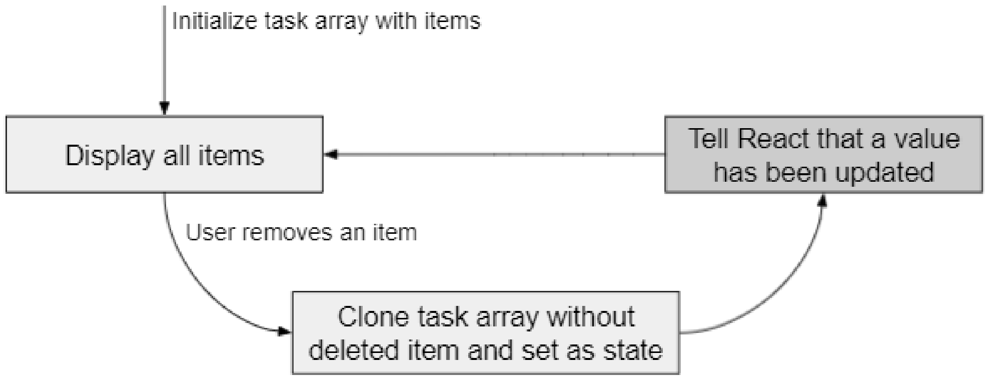

Initialize task array with items
Display all items
Tell React that a value has been updated
User removes an item
Clone task array without deleted item and set as state

Figure 5.29 We now pass a new array to the setter function every time an item is removed, and React correctly identifies that state as updated and will re-render the component.

Let's implement this in listing 5.7.

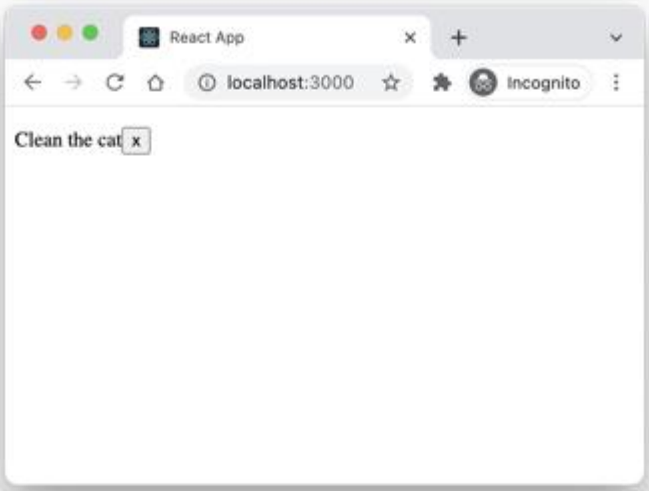
Listing 5.7 A proper todo-list application.

```
import { useState } from 'react';
function TodoApplication({initialList}) {
  const [todos, setTodos] = useState(initialList);
  return (
    <main>
      {todos.map((todo, index) => (
        <p key={todo}>
         {todo}
         <button onClick={() => {
           setTodos([    #A
             ...todos.slice(0, index),    #B
             ...todos.slice(index + 1),    #C
           ]);
         }}>x</button>
        </p>
      ))}
    </main>
  );
}
function App() {
  const items = ['Feed the plants', 'Water the dishes', 'Clean the cat'];
  return <TodoApplication initialList={items} />;
}
export default App;
```

#A We set the state to a new array, which is the concatenation of two things
#B the old array sliced from the start to just before the deleted element, plus
#C the old array sliced just after the deleted element to the end

rq05-proper-todo

This looks the same as before, but now we can actually delete items from the list, as you can see in figure 5.30.



Clean the cat x

Figure 5.30: Our functional todo-list application now actually works as here we have completed two of the items on the task list for the day. Now go run a bath, this is gonna be messy!

5.2.6 Using multiple states

We've hinted it a few times, but to confirm it we can now directly say: **Yes, you can have multiple** `useState` **hooks in the same component.** And you often will have.

Let's expand our new todo-list application. Let's stop deleting items from the array when we complete them. Instead we're just going to mark them as completed. Completed items will be rendered in the list with a strike-through. On top of that, we're also going to add a new filter at the top, where you can decide if you want to see all items or only uncompleted items.

To be able to filter the list, we need to remember whether we should filter out completed items or not. And the perfect way to do this is to add another state value that holds this filter flag. We can illustrate this in figure 5.31.

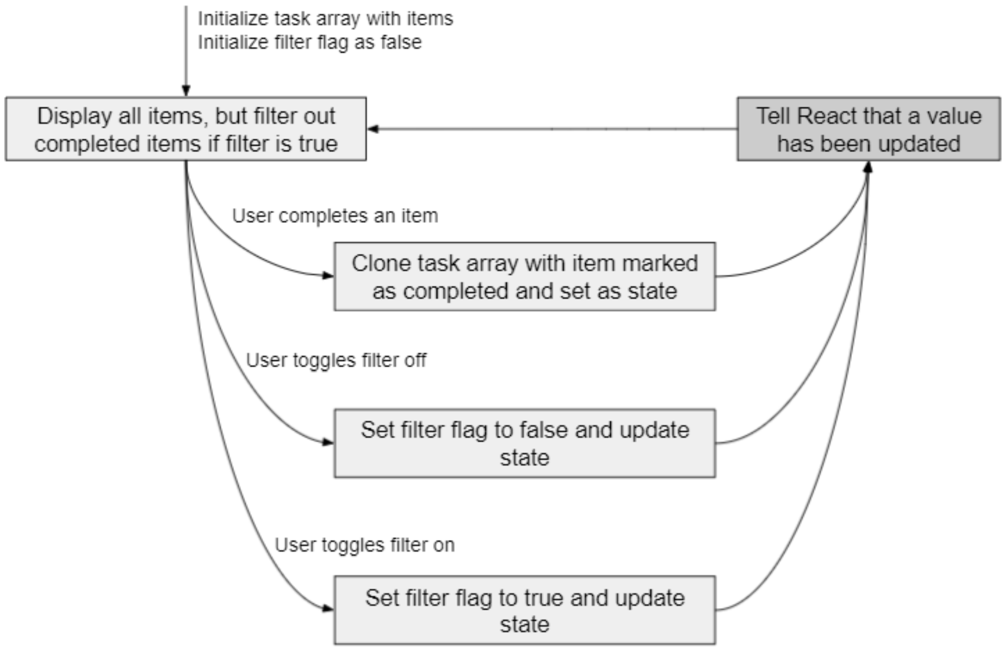

Initialize task array with items
Initialize filter flag as false
Display all items, but filter out completed items if filter is true
Tell React that a value has been updated
User completes an item
Clone task array with item marked as completed and set as state
User toggles filter off
Set filter flag to false and update state
User toggles filter on
Set filter flag to true and update state

Figure 5.31 We can now update state in three different ways. If an item is marked as completed, we still have to remember to create a new array, but with the item in question marked as completed. If we toggle the filter flag, we simply set the relevant state flag.

The implementation of this becomes listing 5.8.

Listing 5.8 Todo-app with a filter.

```
import { useState } from 'react';
function markDone(list, index) {    #A
  return list.map(
    (item, i) =>
      i === index
        ? { ...item, done: true }
        : item
  )
}
function TodoApplication({initialList}) {
  const [todos, setTodos] = useState(initialList);    #B
  const [hideDone, setHideDone] = useState(false);    #C
  const filteredTodos = hideDone    #D
    ? todos.filter(({done}) => !done)
    : todos;
  return (
    <main>
      <div style={{display: 'flex'}}>
        <button onClick={() => setHideDone(false)}>    #E
          Show all
        </button>
        <button onClick={() => setHideDone(true)}>    #E
          Hide done
        </button>
      </div>
      {filteredTodos.map((todo, index) => (    #F
        <p key={todo.task}>
        {todo.done ? (
          <strike>{todo.task}</strike>    #G
        ) : (
          <>
            {todo.task}
            <button onClick={() =>
              setTodos(todos => markDone(todos, index))    #H
            }>x</button>
          </>
        )}
        </p>
      ))}
    </main>
  );
}
function App() {
  const items = [
    { task: 'Feed the plants', done: false },    #I
    { task: 'Water the dishes', done: false },    #I
    { task: 'Clean the cat', done: false },    #I
  ];
  return <TodoApplication initialList={items} />;
}
```

#A We have created a little utility function, that will take an array of task objects and return a new array of the same objects, except one of them will be marked as done as indicated by the second argument
#B We still initialize the task list using the useState hook
#C But now we have a second instance of the useState hook for the new filter flag, which we default to false
#D We use the filter flag to optionally filter the list of todo items to display
#E The two filter buttons call the filter setter function with either true or false

#F Now we must remember to use the new (optionally filtered) list
#G We can now render the task with a strikethrough if it has been completed
#H If not completed, we render a button, that will call our utility function and update the task list state
#I Finally we have to create the list of initial items as a list of objects, each marked as not done yet

`rq05-filter-todo`

Let's see this in action in figure 5.32 and try to use the various buttons.

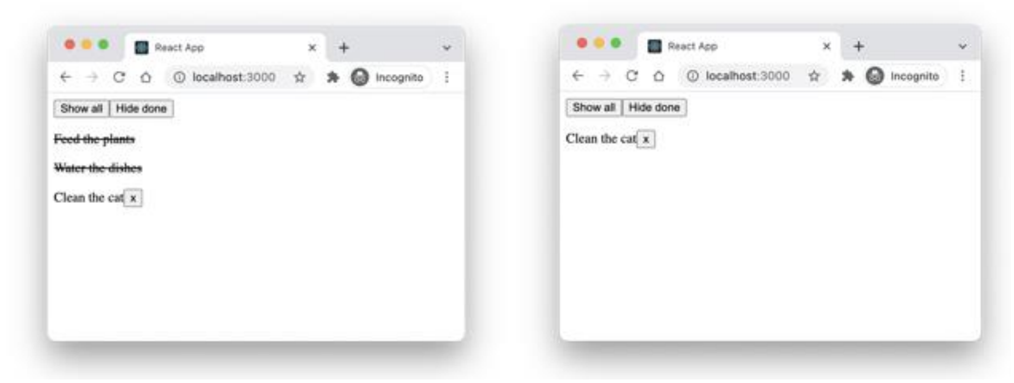


Figure 5.32 After we've completed the two easy items, we can decide if we want to see the full list and bask in our 67% completed progress or only see the remaining items and get a bit overwhelmed because of the single daunting task.

And you're of course not limited to two state values in a single component. You can use as many as you want, though it might get a bit hard to follow if the number exceeds ten. We suggest using either context providers, reducers, or custom hooks—or all three—if states get more complex. We'll get back to how these more advanced techniques work in chapter 9.

5.2.7 State scope

In all the components we created above, we only used the state in the component that we created. But what if we want to have state that spans multiple components? What if we want to access the value in one component, but update it in another? We hinted at this at the beginning of the chapter, when we talked about the number of stateful components in the entire application component tree, but we have actually used it yet.

To do this, we can use properties to pass state values and state setters to the relevant components. The state flowchart is the same as before. The difference is the component tree. Where we just had a single component handle everything, we're now going to introduce a number of components. The TodoApplication component is still our stateful component holding the two state values. To aid this one, we add a FilterButton and a Task, that takes care of rendering the top filter buttons and individual tasks in the list respectively. We can see this new component tree as well as all the properties passed in figure 5.33.

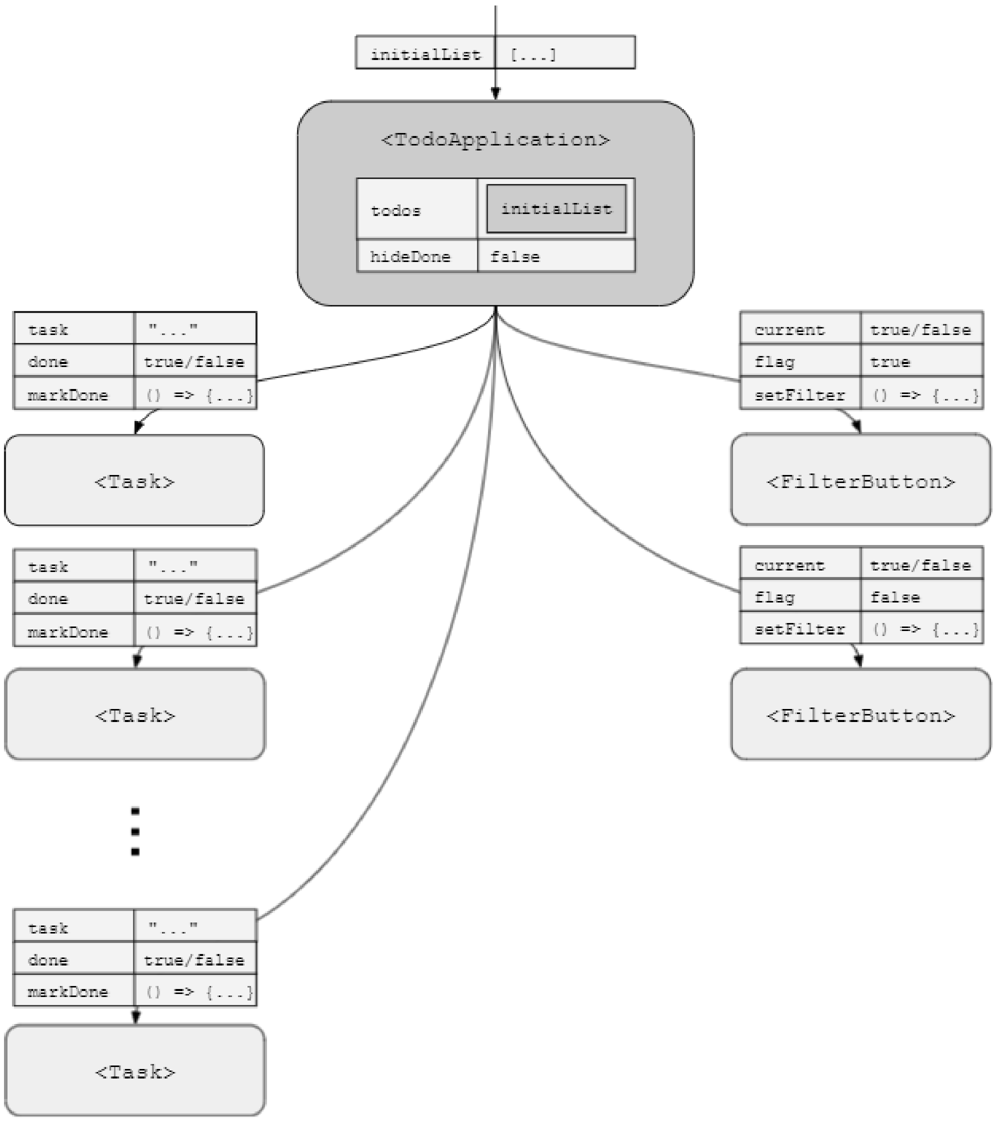


Figure 5.33 The component tree of our multi-component todo-application. We render a variable number of Task instances, one for every item in the list, and always exactly two filter buttons.

Let's now put this all together in a single application in listing 5.9 and while we're at it, let's also make things look a bit nicer with some styles.

Listing 5.15 Advanced multi-component todo-application.

```
import { useState } from 'react';
function markDone(list, index) {
  return list.map(
    (item, i) =>
      i === index
        ? { ...item, done: true }
        : item
  )
}
function FilterButton({ current, flag, setFilter, children }) {    #A
  const style = {
    border: '1px solid dimgray',
    background: current === flag ? 'dimgray' : 'transparent',
    color: current === flag ? 'white' : 'dimgray',
    padding: '4px 10px',
  };
  return (
    <button style={style} onClick={() => setFilter(flag)}>    #B
      {children}
    </button>
  );
}
function Task({ task, done, markDone }) {    #C
  const paragraphStyle = {
    color: done ? 'gray' : 'black',
    borderLeft: '2px solid',
  };
  const buttonStyle = {
    border: 'none',
    background: 'transparent',
    display: 'inline',
    color: 'inherit',
  };
  return (
    <p style={paragraphStyle}>
      <button style={buttonStyle} onClick={done ? null : markDone}>    #D
        {done ? '✓ ' : '◯ '}
      </button>
      {task}
    </p>
  );
}
function TodoApplication({initialList}) {
  const [todos, setTodos] = useState(initialList);
  const [hideDone, setHideDone] = useState(false);
  const filteredTodos = hideDone
    ? todos.filter(({done}) => !done)
    : todos;
  return (
    <main>
      <div style={{display: 'flex'}}>
        <FilterButton    #E
          current={hideDone}
          flag={false}
          setFilter={setHideDone}
        >Show all</FilterButton>
        <FilterButton    #E
```

```
          current={hideDone}
          flag={true}
          setFilter={setHideDone}
        >Hide done</FilterButton>
      </div>
      {filteredTodos.map((todo, index) => (
        <Task    #F
          key={todo.task}
          task={todo.task}
          done={todo.done}
          markDone={() => setTodos(todos => markDone(todos, index))}    #G
        />
      ))}
    </main>
  );
}
function App() {
  const items = [
    { task: 'Feed the plants', done: false },
    { task: 'Water the dishes', done: false },
    { task: 'Clean the cat', done: false },
  ];
  return <TodoApplication initialList={items} />;
}
```

#A The FilterButton takes four properties and render a nice button based on these
#B In particular, the onClick property on the button calls the passer setter function with the passed value
#C Similarly, the Task component takes a number of properties including a callback
#D This time we just invoke the passed callback when we click the button, because it does the required work - but only if the item was not already done
#E We have two filter buttons in the final component with almost identical properties
#F For each task item we create a Task component instance
#G And we set the markDone to the same calculation as before

rq05-nice-todo

And there we have it - our first complete, useful, and well-architected application in React! It looks just like before, except it looks a lot nicer as you can see in figure 5.34.

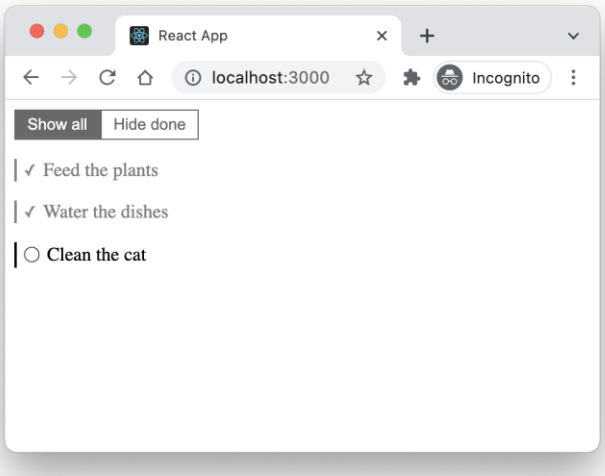


Figure 5.34: Our fully developed todo-application with a lovely UI even! Currently the filter is set to show all, but if we toggle it to "hide done", only the last item would be displayed – just like before.

The ideas used in this application are the same, that fuel any application. We store state in one level and pass it around to other components where applicable to render the result we need. In our latest todo-application, state is stored "globally" in the TodoApplication component and not just locally inside each of the child components.

If we were to add another component to this that existed next to the task list but would need access to the same state values as the task list does, we would need to lift the states from the TodoApplication component up to the App component and then pass the values and the setters down to the TodoApplication component. All of this work of passing state values and setters around can get a bit complex, but we will see how to solve that in a better way in chapter 9 using React context.

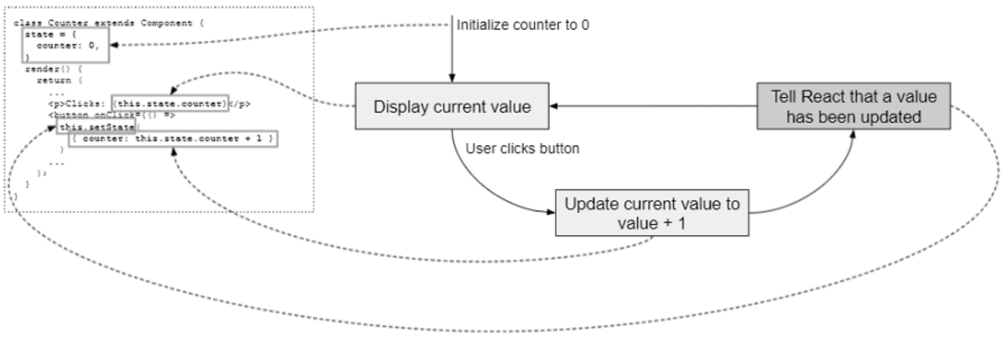
5.3 Stateful class-based components

So far we've covered how to add component state to functional components. But stateful components existed before the emergence of hooks. In fact, state was a primary feature built into the functionality of class-based components.

In class-based components state works the exact same way and has the exact same four steps:

1. Initialize state
2. Display current value
3. Update state
4. Inform React that state has been updated

We've seen a ton of examples of how we do these four steps in a functional component. Now let's look at how to do the same in a class-based component. The API is similar, but the syntax is a bit different and behavior also varies slightly. The basic concept is the same, however. Please take a look at figure 5.35 for a quick overview.



```
class Counter extends Component {
  state = {
    counter: 0,
  }
  render() {
    return (
      ...
      <p>Clicks: {this.state.counter}</p>
      <button onClick={() =>
        this.setState(
          { counter: this.state.counter + 1 }
        )
      ...
    );
  }
}
```

Initialize counter to 0
Display current value
Tell React that a value has been updated
User clicks button
Update current value to value + 1

Figure 5.35 The flowchart of data in our click counter with code added. The dashed arrows connect the state in the flowchart to the corresponding bit of code responsible for that particular action.

Rather than store values as local variables as we do in a functional component, we store the state values on the class itself on the class member called `this.state`.

In this section we'll first cover how this is similar, but with a slightly different syntax, then proceed to cover how this is also fundamentally different due to three larger changes in behavior, and finally we'll briefly cover how to convert a stateful class-based component to a functional component.

Note that we're not going to provide examples in this section, merely explain the differences.

5.3.1 Similarities with useState hook

Everything we've done so far in functional components could just as well have been done in class-based components. Initializing, updating, and displaying state is the name of the game, and we can do that with a slightly different syntax. See table 5.1 for an overview of how the syntax deviates.

Table 5.1. State in functional components and class-based components respectively

Functional component	Class-based component
`const [counter, setCounter] =` `useState(0);` This is the case if state is initialized to a static value.	`state = {` `counter: 0,` `}` When we initialize state to a static value here, we can use a class member.
`const [counter, setCounter] =` `useState(initialValue);` This is the case if state is initialized to a dynamic value from a property.	`constructor(props) {` `this.state = {` `counter:` `props.initialValue,` `};` `}` Here we have to access the property in the constructor and initialize the state using `this.state`.
`<p>` `Counter: {counter}` `</p>`	`<p>` `Counter: {this.state.counter}` `</p>`
`onClick={() =>` `setCounter(0)` `}` If we're setting the state to a fixed value, we can use the non-functional variant of the setter.	`onClick={() =>` `this.setState({ counter: 0 })` `}` We do the same thing here, except we need to make it an object.
`onClick={() =>` `setCounter(` `value => value + 1` `)` `}` If we use an updater function, we simply use the old value and return a new one using whatever type we have in the state.	`onClick={() =>` `this.setState(` `({ counter }) =>` `({ counter: counter + 1 })` `)` `}` If we're setting the state to a dynamic value based on the current state, we can use the functional updater variant of the setter but we have to return an object based on the old state object.

5.3.2 Differences from useState hook

There are differences though. And those are quite significant and will impact how you use state in class-based components as opposed to functional components.

The main differences are:

- You can only have **one state object** and it is **always an object**.
- Components **always re-render if updated**, even if nothing changed.
- Objects are merged when you update the state, so **partial updates are possible**.

We will go over each of these differences with a short example in the following subsections.

ONLY ONE STATE OBJECT

As you can see in table 5.1, class-based component state lives inside a state object. Even if you only have a single value, e.g. a counter, you have to create it on the state object, update it on the state object, and display it from the state object.

The upside is that moving from a single to multiple state values is very smooth. You simply add a second property to the state object, and you are good to go. As soon as you have introduced state into a class-based component, you can support one or multiple state values without an issue.

COMPONENTS ALWAYS RENDER WHEN STATE IS UPDATED

We mentioned in section 5.2.5 that the `useState` hook will only cause the component to re-render, if the state actually updates. If you set a state value to 0 when it is already 0, it will not update the component. React assumes that our components are pure and thus assumes that the component will render the same, if the state has not updated.

It was different in the old days, and some applications actually depended on this back then. In a class-based component, you can call `setState` with the same values or even no values, and React will re-render your component.

STATE OBJECTS ARE MERGED

Because the state in a class-based component is a single big object with potentially tens of state values, it would be annoying if you have to remember to set all of them.

Imagine that you reset a counter with the following snippet:

```
setState({ counter: 0 });
```

If you had a number of other state values in this same component, imagine that this would reset or even delete all these other state values, because you did not include them in the object that you passed to `setState`. That would be quite annoying. If that happened, you would have to do something like this every time to copy all the existing values into the new object:

```
this.setState(
  oldState => ({ ...oldState, counter: 0 })
);
```

Fortunately, you don't have to do that. React automatically does this for class-based components. When you pass a new object (or an update function that returns an object) to the `setState` method, React will automatically merge this new object onto the existing state object. It will in fact do exactly the above, so you don't have to remember to do that every time.

5.3.3 Conversion to a functional component

Imagine that you are tasked with converting a stateful class-based component to a practically and effectively equivalent functional component. This could happen if you work with an older codebase that has to be updated. Or if you are using an older tutorial, but want to update it to hooks to use other more recent features.

We've already seen how to convert the output of the render method to an equivalent functional component.

As components get more complex, the architectural and conceptual differences between class-based and functional components will get more and more significant and at some stage, you will often have to re-think the desired functionality to refactor it to the functional domain.

Sometimes you will even have to scrap the whole thing and start over from scratch because that can be easier than trying to cram the square class-based components into a triangular functional component architecture.

If you are so lucky, that the component you're converting only has a single value in state, conversion is often pretty straightforward. Refer back to table 5.1 to see the syntax conversion between class-based components and functional components.

MULTIPLE STATE VALUES

If you have multiple state values in your class-based component, you need to consider whether you want to convert them to a single state value with an object in your functional component or multiple independent state values. You could even use a mix of the two approaches, with some values going together in objects, and others being independent single values.

It's hard to give any good tips or hints for this process. It depends entirely on the component and application in question. So you will need to use your own development skills to deduce what's the best approach in the given situation.

If you do convert your state to objects with multiple values in a functional component, there are a few pitfalls to note:

- Objects are not merged in `useState` setter functions as they are in `setState`.
- Re-render only happens when a `useState` setter function is invoked with an actual new value, whereas `setState` would always trigger a re-render regardless of value reference equality.

Refer back to section 5.3.2 for details on these differences in behavior.

5.4 Quiz

1. Which of the following would you store in component state:
   a) Dynamic application data
   b) Component properties
   c) Constant values
2. Which of the following is the correct way to initialize a simple numeric state in a functional component:
   a) `const { value, setter } = useState(0);`
   b) `const [ value, setter ] = useState(0);`
   c) `const { value, setter } = useState({ value: 0 });`
   d) `const [ value, setter ] = useState({ value: 0 });`
3. You can only have a single `useState` hook in each functional component, *true* or false?
4. When updating a component state value through a `useState` setter function, the component will always re-render, *true* or *false*?
5. Which of the following would you use to read a single numerical value from state in a class-based component:
   a) `<p>Value: {this.state}.</p>`
   b) `<p>Value: {this.counter}.</p>`
   c) `<p>Value: {this.state.counter}.</p>`

5.5 Summary

- Component state is used to make your application interactive. You will get just about nowhere in your application development if you don't have stateful components.
- You can have state in both class-based components and functional components.
- State in functional components are initialized as separate distinct calls to `useState` and have a separate setter per state value.
- State in class-based components is initialized as a single object and updated using the `setState` method.
- Conversion from a stateful class-based component to a stateful functional component might require a bigger refactor as the two approaches are significantly different.

5.6 Quiz answers

1. The correct answer is *a*. You should definitely store dynamic application data in state, but never properties (you already have them in the properties object), and neither should you store unchanging values in state.
2. The correct answer is *b*. You provide the initial value to `useState` as a simple value and you destruct the returned value as an array, not an object.
3. This is *false*. You can have as many `useState` hooks in each component as you desire.
4. This is *false*. The component will only render if the new value passed to the setter function is different from the existing value. The comparison is done using referential equality, so even if an object has updated internally, if it is still the same object, it will not cause a re-render.
5. The correct answer is *c*. Remember that state in a class-based component is always an object, and your state values are properties of that object.

6 Effects and the React component lifecycle

This chapter covers

- **Running effects inside components**
- **A complete guide to the React component lifecycle**
- **Mounting, unmounting, and rendering components**
- **Introducing lifecycle methods for class-based components**

React components use JSX to send information to the user in the form of HTML. But components need to do a lot more than just that in order to be useful in an application. In React, everything that happens, happens in some component, so if your application wants to set a cookie, load some data, handle form input, display the user's camera, start or stop a timer, or a myriad of other dynamic capabilities, you need more than just JSX.

If you want your component to load some data from a server, you want the effect to run as soon as the component loads, but then you don't actually need the effect to run again even if your component re-renders. On the other hand, if you want to set a cookie with the last username entered into the login field, you want that effect to run every time the user types in the input field. And if you want to display a timer inside your component, you want the timer to start ticking as your component loads, but you also want the timer to stop ticking as your component later unloads to avoid unnecessarily clogging up resources.

What you need are *effects*. Effects are functions that run inside a component under certain circumstances. To run an effect, you have to specify under which circumstances the effect should run. In order to fully understand this, we have to dive into the topic of the React component lifecycle.

We will properly define some terminology that we have already used previously, but not properly explained. This includes mounting, unmounting, and re-rendering. The latter is especially important. When and why do components re-render and how can you hook into this process to either control it or react to it?

We'll finally give a brief introduction to how lifecycle methods work in class-based components and what they compare to in functional components. This difference is even more complicated than earlier because lifecycle methods were extraordinarily complex compared to how much simpler the effects are.

With all that to cover, let's get started!

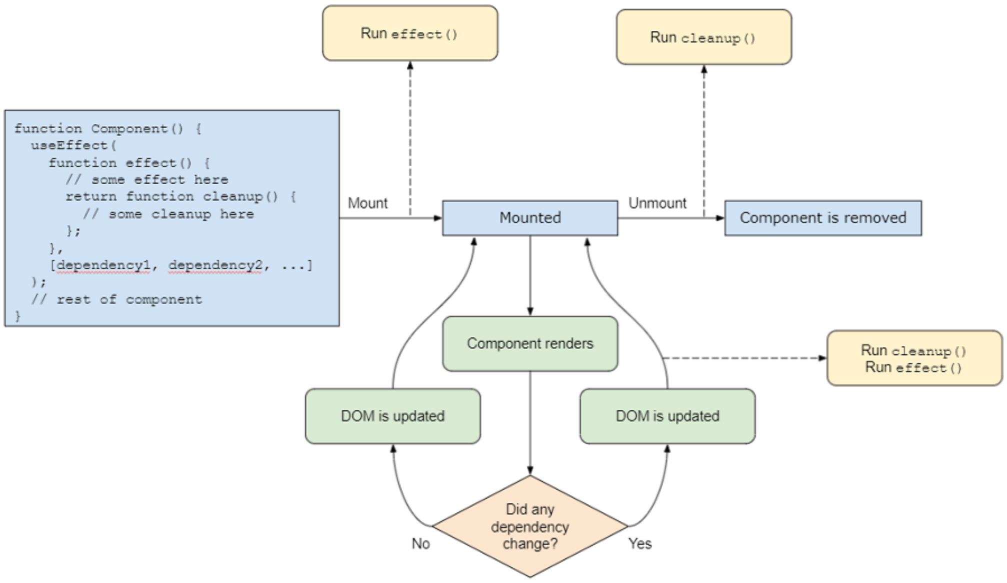
NOTE: The source code for the examples in this chapter is available at https://github.com/rq2e/rq2e/tree/main/ch06. But as you learned in chapter 2, you can instantiate all the examples directly from the command line using a single command.

6.1 Running effects in components

Let's say that you have a Timer component and you want it to display the number of seconds it has been mounted. The first thing that comes to mind is to create an interval with `setInterval` inside the function body which increments a counter state value every second. But when you change the state value, the whole component re-renders, which would start another interval, which would render your component twice every second, which would start another two intervals rendering it four times every second, etc. That's clearly not the way to do it.

Another idea could be to just use a timeout with `setTimeout`. 1 second after the component renders, we increment the counter state value, which in turn causes a re-render, which would start a new timeout. This seems like a reasonable approach. But what if your component re-renders for other reasons? A component can re-render because a property changes. Or it could have multiple state values that could change independently of the counter. And if your component unmounts because it isn't needed anymore, the timeout would still be running and after a second, it would try to update a component that no longer exists. That's unfortunately neither a good way to do it.

To solve this problem, React introduced an *effect hook*, called `useEffect` (notice the important `use*` prefix used on all hooks). An effect in a `useEffect` hook is triggered when either value in a set of dependencies changes. Furthermore, when an effect in `useEffect` runs, it can define a cleanup function that should run in one of two cases: before the effect is triggered again or if the component unmounts. Please refer to figure 6.1 to see a description of this.



```
function Component() {
  useEffect(
    function effect() {
      // some effect here
      return function cleanup() {
        // some cleanup here
      };
    },
    [dependency1, dependency2, ...]
  );
  // rest of component
}
```

Figure 6.1 The useEffect hook here is displayed both as a code snippet and a flow chart. The hook contains an optional effect as well as an optional cleanup function. The effect runs on mount and the cleanup runs on unmount - if they're defined of course. Furthermore, if the effect has a dependency array, the cleanup and effect will also run, every time any value reference in the dependency array changes.

We get that this diagram is pretty complex, but we will take you through it one step at a time by introducing functions that do just a few things at a time.

The trick to this diagram is, you can set your useEffect call up in such a way, that you can define just the effect, just the cleanup function, or both when it suits your needs. Furthermore, by carefully crafting the dependency array with the right values, you can trigger your effect and cleanup to run at exactly the desired instances.

There are 5 likely scenarios that you want your effect and cleanup function to run under. We will go through all 5 scenarios with examples of each:

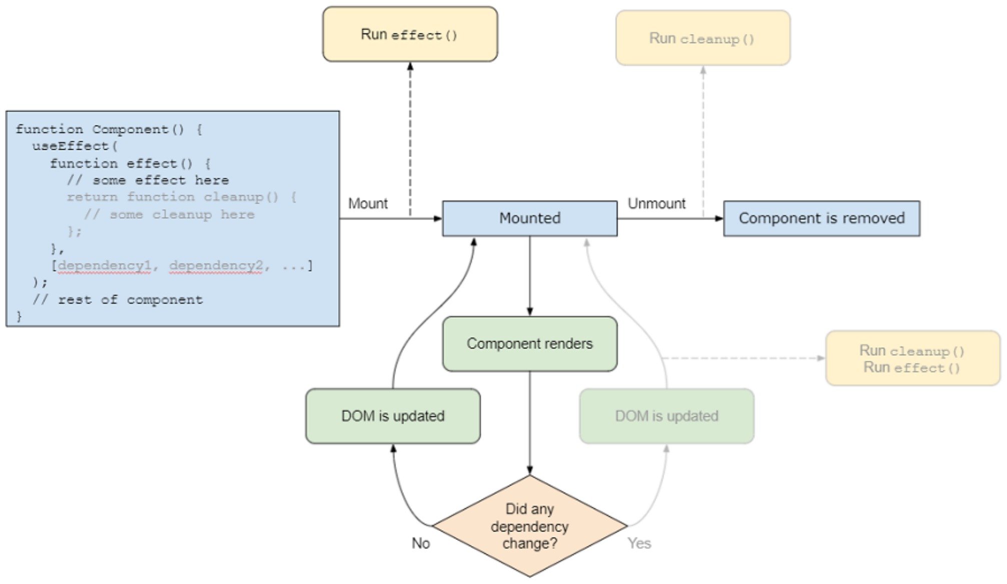
- Imagine you are loading some external data in a component. You want to run such an effect *as your component mounts*.
- Imagine you are creating a timer using an interval. You want to run such an effect *as your component mounts*, but also clean it up again *as your component later unmounts*.
- Imagine you want to track when a dialog is closed regardless of how it is closed. You want to run such an effect only as your component unmounts.
- Imagine you want to update the browser window (or tab) title with the title of the page currently displayed. You want to run such an effect *every time the title property changes*, but not when any other property changes as long as the title remains the same.
- Imagine you want to run a timer but only if the timer is active as denoted by an `isActive` flag. You want to run *such an effect and its cleanup every time the* `isActive`

flag changes, but not if other properties or values change as long as the `isActive` flag remains the same.

6.1.1 Run an effect on mount

Let's say we want to create a dropdown component that loads data from an external server to be displayed in the dropdown. We need to load this data as an effect, that runs on mount and then it shouldn't ever run again (because we already have the data).

In this scenario, only the part of the diagram highlighted in figure 6.2 is relevant.



```
function Component() {
  useEffect(
    function effect() {
      // some effect here
      return function cleanup() {
        // some cleanup here
      };
    },
    [dependency1, dependency2, ...]
  );
  // rest of component
}
```

Figure 6.2 This shows the execution of an effect hook when the effect is only desired as the component mounts. Note how the dependency array is kept empty and there's no cleanup function. This means that the effect is only ever executed on mount and never as the component re-renders.

You can see the code in listing 6.1 and the result in figure 6.3.

Listing 6.1 Dropdown loading options from remote

```
import { useState, useEffect } from 'react';
function RemoteDropdown() {
  const [options, setOptions] = useState([]);    #A
  useEffect(
    () =>
      fetch('//www.swapi.tech/api/people')    #B
        .then(res => res.json())
        .then(data => setOptions(data.results.map(({ name }) => name))),    #C
    [],    #D
  );
  return (
    <select>
      {options.map(option => (
        <option key={option}>{option}</option>
      ))}
    </select>
  );
}
```

#A We need a state to have a place to hold the values once the options have been fetched
#B In our effect hook, we load this URL (which is a list of characters in Star Wars)
#C As the result is parsed, we set our state value with an array of character names
#D Finally we make sure to pass an empty dependency array, so this effect only runs on mount and never again

`rq06-remote-dropdown`

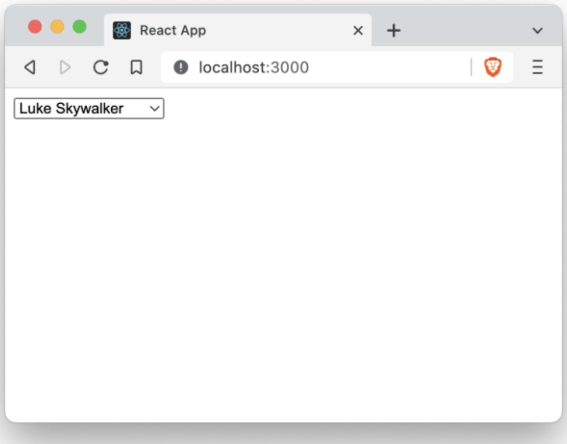


Figure 6.3 Our Star Wars character dropdown in action. May the source be with you!

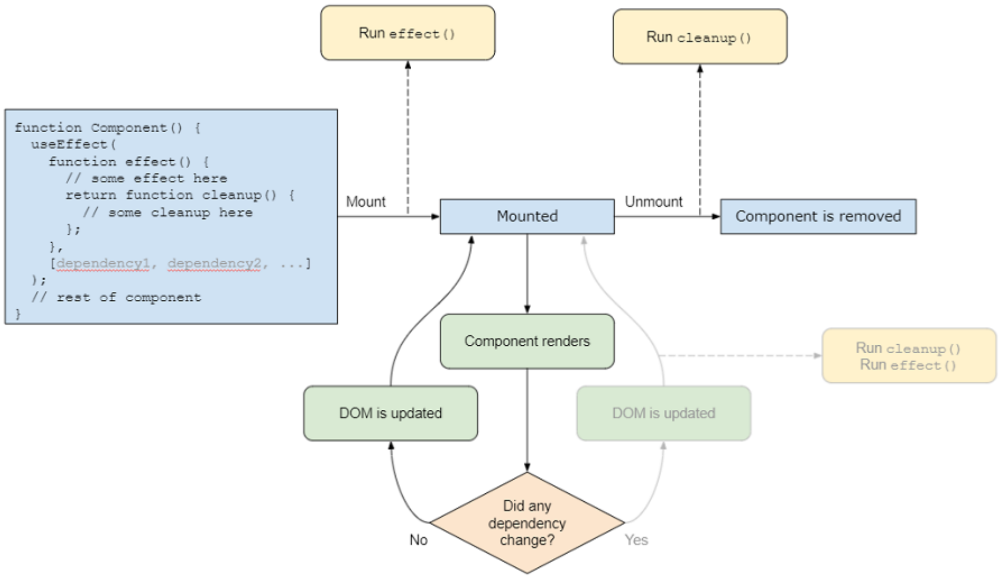
This is a pretty classic setup that you will often see in web apps loading data relevant only inside a small part of the overall application.

It does have a small problem though. What happens if the component for some reason unmounts before the response comes back from the server? We'll have to deal with that in a cleanup function. Cue next section.

6.1.2 Run an effect on mount and cleanup on unmount

We have been tasked with creating a Stopwatch component. It should start an interval as soon as the component mounts, that just keep ticking up as time passes, but if the component is ever unmounted in the future (because the user closes it for example), we must make sure to stop the interval again. This requires an effect that runs on mount, but also runs a cleanup function on unmount.

In this scenario, only the part of the diagram highlighted in figure 6.4 is relevant.



```
function Component() {
  useEffect(
    function effect() {
      // some effect here
      return function cleanup() {
        // some cleanup here
      };
    },
    [dependency1, dependency2, ...]
  );
  // rest of component
}
```

Figure 6.4 The effect hook when an effect and cleanup only on mount and unmount respectively require an empty dependency array to make sure that the effect and cleanup never runs just because the component is re-rendering.

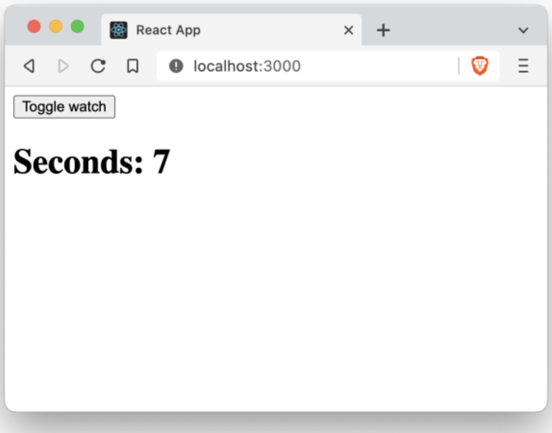
You can see the code for such a component in listing 6.2 and a screenshot of it in action in figure 6.5.

Listing 6.2 Stopwatch

```
import { useState, useEffect } from 'react';
function Stopwatch() {
  const [seconds, setSeconds] = useState(0);
  useEffect(
    () => {    #A
    const interval = setInterval(
      () => setSeconds(seconds => seconds + 1),
      1000,
    );
    return () => clearInterval(interval);    #B
  }, []);
  return <h1>Seconds: {seconds}</h1>;
}
function App() {
  const [showWatch, setShowWatch] = useState(false);
  return (
    <>
      <button onClick={() => setShowWatch(b => !b)}>Toggle watch</button>
      {showWatch && <Stopwatch />}    #C
    </>
  );
}
export default App;
```

#A In our effect function, we start an interval to be run every second, that increments the counter
#B We make sure to cancel the interval in the cleanup function
#C We only conditionally render the stopwatch to see the cleanup function actually do its job

rq06- stopwatch



Seconds: 7

Figure 6.5 Our stopwatch is ticking away

You might note that even though we're using the variable `setSeconds` inside the effect, we don't list it as a dependency. Why is that? We'll get back to just that in section 6.3.3 about dependency arrays in general.

EVENTS

Another common example of using an effect only for mount and unmount is listening for events. It could be that you want your component to update itself when the whole web page resizes (listening to the resize event) or when a certain element scrolls (listening to the scroll event).

We will see a number of examples of this happening in chapter 7, that's dedicated to events.

CANCEL ACTION IF UNMOUNTED

The third type of use case for mount and unmount is actually an extension of the example in the previous section. Our `RemoteDropdown` loads data when mounted, but what would happen if the data transfer was slow and the user somehow navigated away from the part of the application with the dropdown, before the response came in? We would be trying to update state on a component that no longer existed!

This can be mitigated in one of two ways. You can either cancel the request in a cleanup function (which in JavaScript can be done with an `AbortController`) or you can have a local flag that remembers whether the component is still mounted and only updates the component state if the flag is true. If not, the component just ignores the returned response.

Canceling the request using an `AbortController` on unmount would be done something like this:

```
useEffect(
  () => {
    const controller = new AbortController()    #A
    fetch(url, { controller })    #B
      .then(data => {
        // handle the data
      });
    () => {
      controller.abort();    #C
    };
  },
  [],
);
```

#A We create an abort controller inside the effect
#B Make sure to pass the abort controller to the fetch function
#C In the cleanup function, we ask the abort controller to do its job. If the request already went through, nothing happes if we try to abort anyway

The former option of aborting is of course the better option, just cancel the request, but that might not always be possible. If we can't cancel the request for some reason, we can keep track of whether the component is still mounted as follows:

```
useEffect(
  () => {
    let mounted = true;    #A
    fetch(url)
      .then(data => {
        if (!mounted) {    #B
          return;
        }
        // handle the data
      });
    () => {
      mounted = false;    #C
    };
  },
  [],
);
```

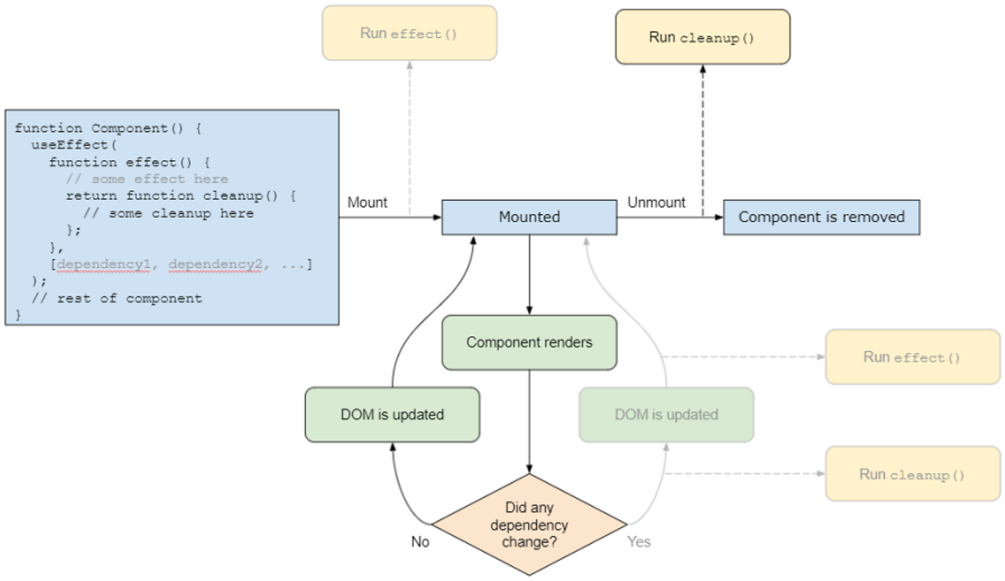
#A We keep a local variable in our effect function that is initially set to true and reflects that fact, that as far as we know, the component is currently mounted
#B Once the data comes in, we will first check whether the component is still mounted. If not, just about now
#C We flip the boolean flag in a cleanup function, which will only be invoked if our component unmounts

This setup works for any type of delayed callback running in an effect hook. It could be a promise that resolves, a timeout that executes, or anything like that. You set a local variable inside the effect to false when the component unmounts and then make sure to just abort the callback when triggered.

6.1.3 Run cleanup on unmount

Imagine we are working on a large application with a dialog component that is displayed when some kind of alert has to be presented to the user. This dialog can be closed in a number of ways including pressing the little x in the corner, pressing escape on the keyboard, pressing the OK button in the bottom, etc. You are tasked with adding an analytics call as the dialog closes. You could manually add this little piece of code to all the different ways the dialog can be closed, but you know that you can run an effect as a component unmounts, so you want to do just that.

In this scenario, only the part of the diagram highlighted in figure 6.6 is relevant.



```
function Component() {
  useEffect(
    function effect() {
      // some effect here
      return function cleanup() {
        // some cleanup here
      };
    },
    [dependency1, dependency2, ...]
  );
  // rest of component
}
```

Figure 6.6 If only the cleanup function is relevant, there's no need to specify any effect code, only return a function from the effect function. And with an empty dependency array once again, this code will never run just because the component re-renders.

We can do this in our dialog as follows:

```
function Dialog() {
  useEffect(
    () => () => trackEvent('dialog_dismissed'),    #A
    [],
  );
  // rest of component goes here
};
```

#A Double arrow notation is required, as we want our effect function to return a function when executed

Note that this is only a partial example, as it assumes our dialog is part of a larger application with a lot more functionality.

Another example, which coincidentally also involves a dialog, is focus management. When you use a keyboard to tab your way to a button and press enter to open a dialog, if you then later dismiss the dialog, you want the keyboard focus returned to *that same button*, so you can keep tabbing from there to other buttons in the user interface.

When the dialog opens, we want the keyboard focus to move inside the dialog, but once unmounted, we should make sure to reset the keyboard focus to whatever element had focus before the dialog was opened. This could be done in a `useEffect` hook with only a cleanup function.

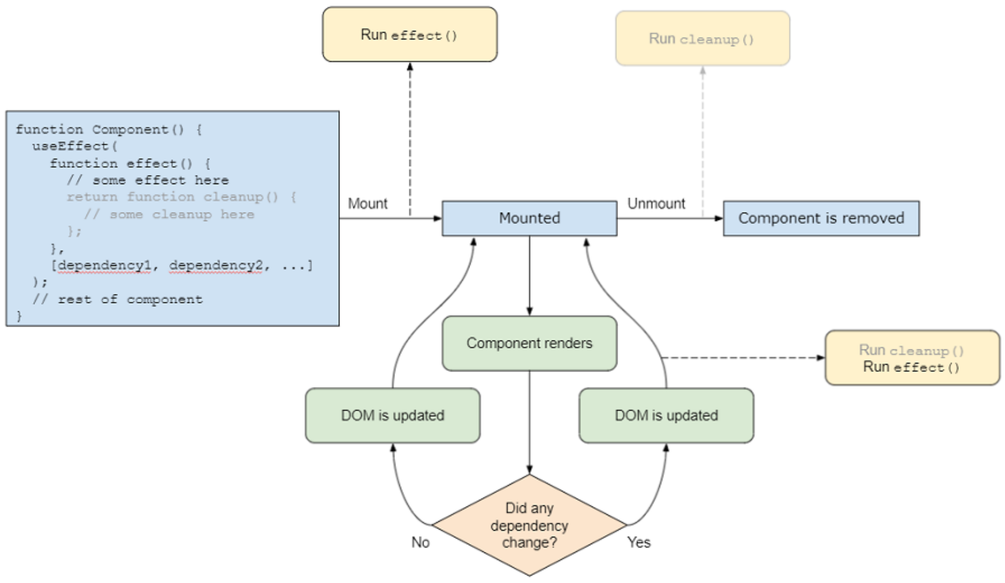
You might note that both of the above examples are a bit far-fetched or at least very specific to some narrow use-cases. That's because this flow of only using a `useEffect` hook for its cleanup function on unmount is a bit unusual and doesn't happen that often in components in the real world.

A much more common use-case for the cleanup function is to do exactly what it's named for: clean up after a `useEffect`, that leaves some sort of functionality in place while mounted, which we need to clean up as the component unmounts, in order to not misuse resources or have memory leaks in our application. We saw an example of that in the previous subsection and we'll see a lot more examples in the future.

6.1.4 Run an effect on some renders

Wouldn't it be wonderful, if the title of the tab in the browser updates as the user navigates around on our blog? We happen to have created the whole blog website in React and it has a component that can dynamically display any blog post. We now want to change the document title in an effect that runs every time the blog title changes, but it does not need to run if any other property changes.

In this scenario, only the part of the diagram highlighted in figure 6.7 is relevant.



```
function Component() {
  useEffect(
    function effect() {
      // some effect here
      return function cleanup() {
        // some cleanup here
      };
    },
    [dependency1, dependency2, ...]
  );
  // rest of component
}
```

Figure 6.7 This time we're going to utilize the dependency array. We want our effect to run on mount, but also every time a certain property changes. We however don't want it to run just because other properties change, so we are careful about including only the relevant variables in the dependency array.

See this component in listing 6.3.

Listing 6.3 Side-effect executed in hook

```
import { useEffect} from 'react';
function BlogPost({ title, body }) {
  useEffect(
    () => {
      document.title = title;    #A
    },
    [title],    #B
  );
  return (
    <article>
      <h1>{title}</h1>
      {body}
    </article>
  );
}
```

#A Our effect inside useEffect sets the document title to the value of the title property
#B Putting only the title in the dependency array ensures, that the document title is updated only when the post title is

rq06-blog-title

This is probably the perfect textbook example of what `useEffect` is meant for - namely to execute side effects of a component. You can't update the document title through the DOM, so it has to be a side effect and for that, `useEffect` is the perfect solution.

UPDATE FROM PROPERTY

Another common use case is to update a state value based on a property. You might remember from the last chapter that if we initialize a state in `useState` to the value of a property, it only gets that property when the component renders the first time around on mounting. If the component later updates with a new property value, the state will not automatically update to that value.

We can fix that using an effect that depends on the value of the property and updates the state value based on it. Let's build a very simple email input component, where the user can input their email address. However, we will allow the email address to be prefilled from the "outside" from a parent component using a property. We do this in listing 6.4.

Listing 6.4 State updated from property

```
import { useEffect, useState } from 'react';
function EmailInput({ value }) {
  const [email, setEmail] = useState();    #A
  useEffect(
    () => setEmail(value),    #B
    [value],    #C
  );
  return (
    <label>
      Email address
      <input
        type="email"
        value={email}    #D
        onChange={(evt) => setEmail(evt.target.value)} />    #E
    </label>
  );
}
```

#A We create a new state value, but we don't initialize it to anything
#B That's because on every render where the property value changes, we will (re)set the email state value to the property
#C We remember to add a dependency array, which is only the property
#D We update the email input field in a new way in this component. We will discuss this more in chapter 8
#E Finally we update the state value every time the input changes

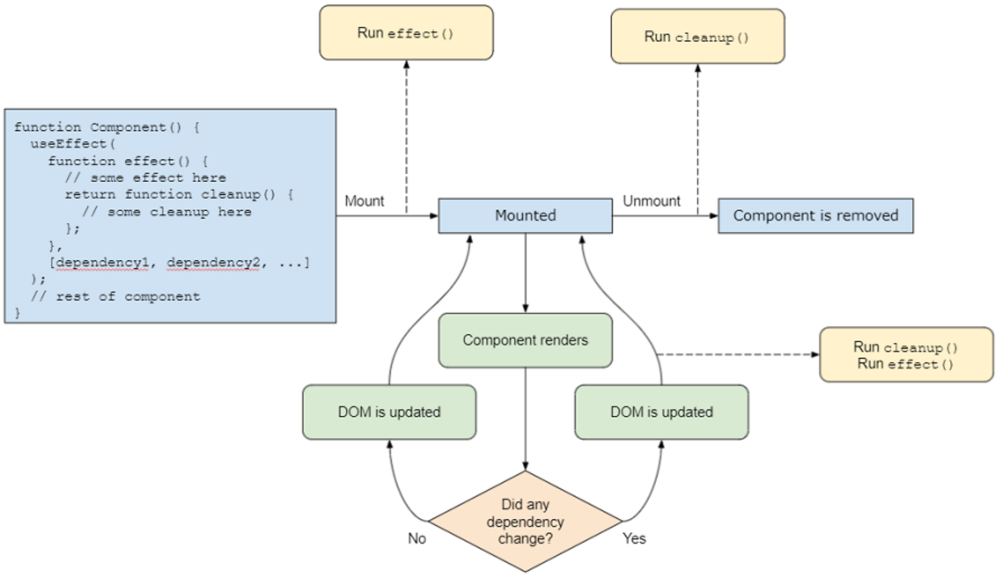
`rq06-email-input`

The use-case here can maybe be a little hard to understand, but it is actually a pretty common pattern in controlled input components.

6.1.5 Run an effect and cleanup on some renders

This time, rather than a stopwatch that counts up, we're now going to create a countdown component that – you guessed it – counts down. This countdown can now be paused and resumed. In order to do that, we still need to run an interval in an effect, but we need to stop and start this interval every time the countdown is paused and resumed respectively. To do this, we need to create an effect (with a cleanup function), that has a dependency.

In this scenario, everything in the diagram figure 6.8 is relevant.



```
function Component() {
  useEffect(
    function effect() {
      // some effect here
      return function cleanup() {
        // some cleanup here
      };
    },
    [dependency1, dependency2, ...]
  );
  // rest of component
}
```

Figure 6.8 Everything in the effect hook is relevant, when effect and cleanup on some renders are the goal

An example of a component, where we want to run cleanup on unmount would be a Countdown component. This component is different from the Stopwatch from earlier in that you can pause, start, and stop the clock whenever you feel like without unmounting and remounting the component (which was the only way to stop the Stopwatch from earlier).

The countdown component will be initialized with the starting time of the counter. It also has a reset button that will reset the counter to the initial value at any point. Furthermore, there's a pause/resume button that will toggle whether the counter is running or not. And finally, there's the actual countdown decreasing every second, pausing the counter once it reaches 0. And to make sure we can't start the counter again at 0, the pause/resume button is disabled if the countdown is over.

This sounds complicated, but please see the state flowchart for this component in figure 6.9.

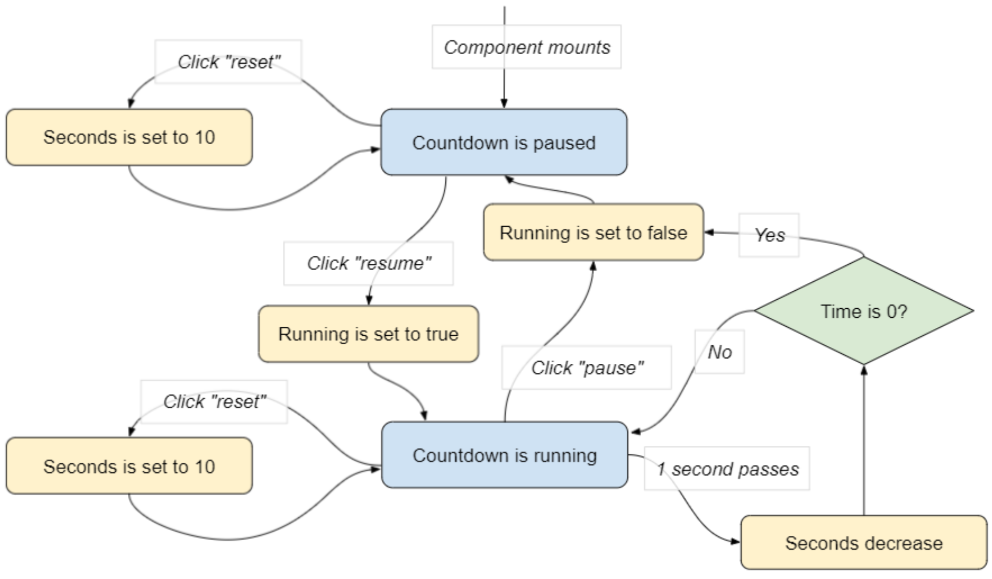


Figure 6.9 This describes the flow of state in the countdown component, as time passes and the user interacts with the component. Notice in particular how pressing reset does not stop or start the countdown, but just leaves it running or not. Also note how the time stops when time runs out.

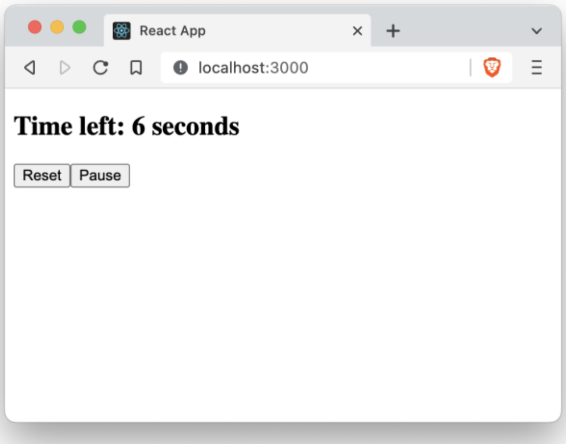
You can see this implemented in listing 6.5 and the result in figure 6.10.

Listing 6.5 An interactive countdown

```
import { useEffect, useState } from 'react';
function Countdown({ from }) {
  const [seconds, setSeconds] = useState(from);    #A
  const [isRunning, setRunning] = useState(false);    #B
  useEffect(
    () => {
      if (!isRunning) {    #C
        return;
      }
      const interval = setInterval(() => setSeconds(value => {    #D
        if (value <= 1) {
          setRunning(false);    #E
        }
        return value - 1;    #F
      }, 1000);
      return () => clearInterval(interval);    #G
    },
    [isRunning],    #H
  );
  return (
    <section>
      <h2>Time left: {seconds} seconds</h2>
      <button onClick={() => setSeconds(from)}>Reset</button>    #I
      <button
        onClick={() => setRunning(v => !v)}    #J
        disabled={seconds === 0}    #K
      >
        {isRunning ? 'Pause' : 'Resume'}    #L
      </button>
    </label>
  );
}
```

#A We initialize the seconds to the value of the initial property
#B We initialize the isRunning flag to false
#C The first thing we check in the effect is whether the countdown is running at all. If not, we just abort silently (and return nothing - nothing to cleanup)
#D If the countdown is running, we define an interval that updates the state value every second
#E When we update the state value, we check if the value was 1 (or less), if so, make sure to stop the countdown
#F Then we return one less that the current value of the counter
#G We also make sure, that our effect returns a cleanup function, that cancels the interval completely
#H We make our effect depend on the value of the isRunning state value. Whenever this value changes our effect runs (and the cleanup of the last effect runs just before it)
#I In our component JSX we have a button, that resets the counter and only that (is does not change the value of the run flag)
#J We have another button, that flips the value of the run flag, but doesn't change the counter
#K This button however is disabled, if the counter is at zero
#L And finally, we vary the text on the toggle button depending on the current state of the run flag

`rq06-countdown`



Time left: 6 seconds

Figure 6.10 The countdown component while running

There's quite a lot happening in this component and we use the three hooks in a clever combination to achieve the desired goals. One thing you might notice is that when our counter reaches zero, we don't directly stop the interval. In line #E above, we just toggle the running flag to false. Doing so will force our component to re-render, and as it re-renders the effect will re-run, because `isRunning` is listed as a dependency for the effect. And as the effect reruns, the cleanup function will make sure to stop the interval. So setting the run flag to false will indirectly stop the interval, but only through the magic of the hook.

This is a pretty advanced component, so it's okay if you don't understand it at first. We strongly recommend that you download the code for the app above and play around with it. Try changing parts of the code to see what makes it tick and how it works the way it does.

6.1.6 Running an effect synchronously

Now we're going to talk about an even more hypothetical situation than we normally do. Imagine that you are creating a component that has a bunch of text in it and you want to count how many letters are in total and display that. The text is all static, so you could just go ahead and count them all by hand before creating the component, but you want to make sure the component automatically updates the count if you later change the text.

One way to create this would be to add a state value to the component, that will contain the letter count and initialize this to zero. Then we add an effect to the component, that runs after the component has rendered, counts all the letters, updates the state, and when the component re-renders, it will display the correct count.

This would combine the data flows of an effect hook with that of a state hook. We've illustrated that in figure 6.11.

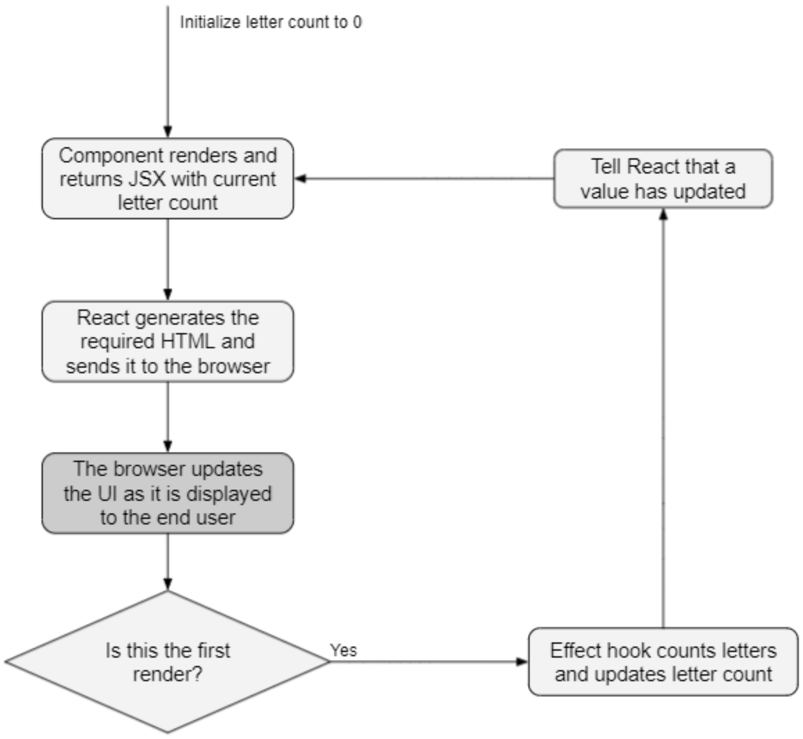

Initialize letter count to 0
Component renders and returns JSX with current letter count
React generates the required HTML and sends it to the browser
The browser updates the UI as it is displayed to the end user
Is this the first render?
Yes
Effect hook counts letters and updates letter count
Tell React that a value has updated

Figure 6.11 The state flow as we update state and run an effect. The problem here is the darker box. As we update the UI, the user will see the initial 0 displayed, before the component quickly re-renders and displays the correct number of letters.

The problem with the flow as described and displayed in figure 6.11 is that the browser updates the UI and displays it to the user before the effect hook runs. This means that the user will actually see the component render a 0 briefly, before the component re-renders and displays the correct number of letters.

What if we instead could run an effect after React generates the required HTML, but before the browser updates the UI and displays it to the user? Well, surprise, surprise, we can do exactly that. We can run a *layout effect hook* instead, which does two things differently than the regular effect hook. First off, it runs before the browser updates the UI, but just as important, it runs instantly as the DOM is generated. If React detects a state update from a layout effect, it will immediately re-render the component with the updated state. We can see that in figure 6.12.

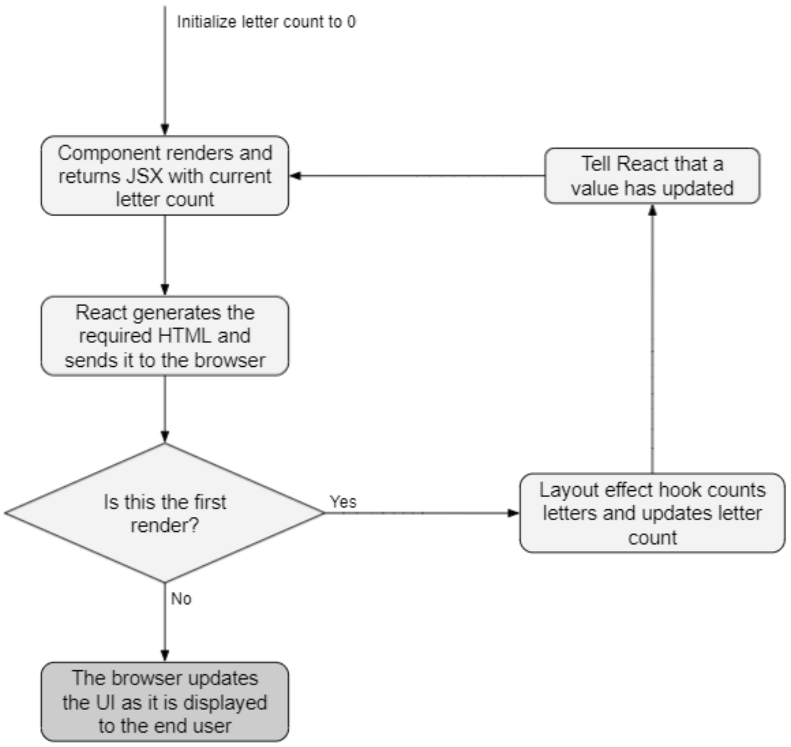

Initialize letter count to 0
Component renders and returns JSX with current letter count
Tell React that a value has updated
React generates the required HTML and sends it to the browser
Is this the first render?
Yes
Layout effect hook counts letters and updates letter count
No
The browser updates the UI as it is displayed to the end user

Figure 6.12 Now that we're using a layout effect, it will run before the browser UI is updated and the new state will render immediately as the effect updates it, which makes the UI correct the very first time around.

By replacing `useEffect` with a `useLayoutEffect` in this very special instance, we can avoid a brief flash of the wrong content.

Do note that `useEffect` is the correct hook to use in almost all the use-cases. `useLayoutEffect` is only relevant in very few specialized instances, so always try `useEffect` first, and only if that doesn't work for your purpose, see if `useLayoutEffect` might be the right choice instead.

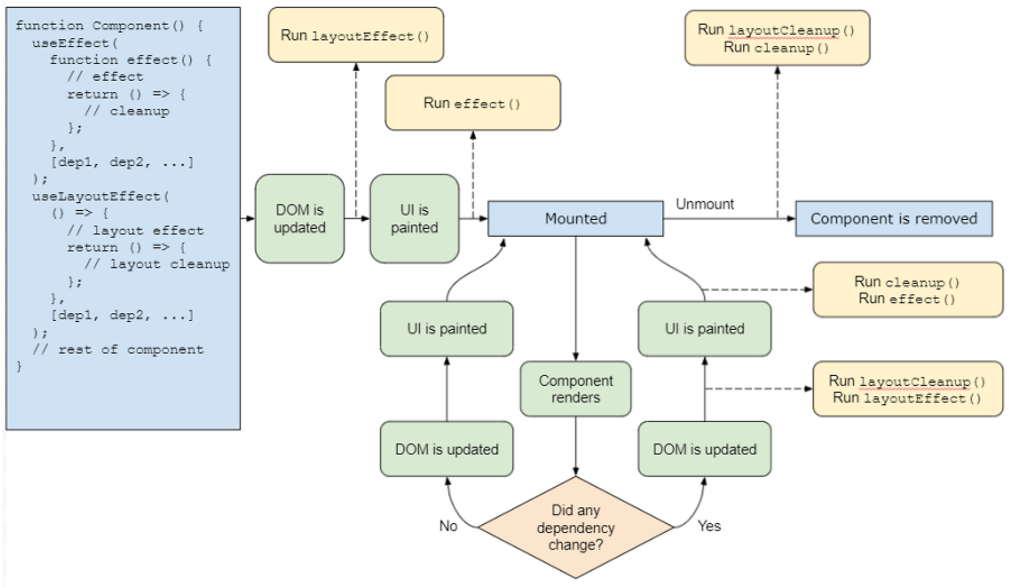
TECHNICAL DETAILS OF A LAYOUT EFFECT

This hook is a variant of `useEffect`. It is completely identical to `useEffect` in every way except *when* it is called. That means, that similarly to `useEffect`, the hook `useLayoutEffect` also does the following:

- takes a function and a dependency array as arguments,
- when any dependency changes,
- the cleanup function if any of the previous effect is run,
- then the effect is run for this instance
- while capturing any potential returned cleanup function resulting from the effect.

The difference between `useEffect` and `useLayoutEffect` is a bit technical but boils down to timing. `useLayoutEffect` is called *synchronously* in the same execution cycle as when the components are rendered into the DOM (but before the browser has had a chance to *paint* the DOM to the browser window), while `useEffect` is invoked *asynchronously* on the next execution cycle where the DOM will have been painted to the window and all CSS will have taken effect and be calculated.

The timing of the two events can be seen in figure 6.13.



```
function Component() {
  useEffect(
    function effect() {
      // effect
      return () => {
        // cleanup
      };
    },
    [dep1, dep2, ...]
  );
  useLayoutEffect(
    () => {
      // layout effect
      return () => {
        // layout cleanup
      };
    },
    [dep1, dep2, ...]
  );
  // rest of component
}
```

Figure 6.13 This displays the timing of useLayoutEffect versus useEffect. Note how the layout effect is run just after the DOM is updated, but before the browser has had a chance to layout the elements using CSS.

As you can see in figure 6.13, we actually hid some details from the previous diagrams of `useEffect` execution. Note that here we have a `useEffect` and a `useLayoutEffect` with the exact same dependencies. These dependencies can of course vary, which would make the flow run differently for different renders, as some would only run layout effects and cleanups, while others would run regular effects and cleanups and some might still run both.

If you don't fully understand the difference, you shouldn't worry too much about it. In 99% of cases, you ought to use the `useEffect` hook. Only in the very rare instances, where you need to update the DOM in an effect before it is painted to the window after the component has rendered, do you need to use the `useLayoutEffect` hook.

6.2 Understanding rendering

In the previous section, we have talked about components re-rendering many times. In this section, we will go into some more technical details about what it means for a component to (re-)render. Note that this is not directly practically useful, but it is very important background information in order to understand what is going on in your application.

A functional component will render for one of 3 reasons:

- The component has just been mounted
- The parent component re-rendered
- The component uses stateful hooks, that have updated

That's it. If *none* of the above happens, your component *will not* re-render, and that's a guarantee. If *either* of the above happens, your component *will* re-render for sure. However, React might batch rendering after several of these things happening, so if both a state value changes and the parent component re-renders, the component might only re-render once, or it might re-render twice. That is controlled by React and depends on subtle timing details.

You can take steps to minimize some of those re-renders, which we will cover in the next chapter, but for now, let's cover each of the above three reasons for re-render.

We will give detailed examples of all of the above, discuss how you can see these things happen, and what you can do when they happen.

Do note that we are here talking about your component re-rendering *as a whole*. You might have functions or callbacks in your component that render some part of your output, and they can re-render for any of a myriad of reasons depending on your usage. This is particularly the case if you're using so-called *render props*, which are often used in older codebases and in the non-hook variant of the React Context API. You might still see render props in modern and more complex codebases, as they can be used to render partial content in a generic component. We will discuss this at the end of this section.

6.2.1 Rendering on mount

Imagine that we have a component that loads some external data. We can use the example of our remote dropdown from earlier. When it mounts, it loads data from a remote server and stores it locally in the component. When it unmounts, the data is forgotten.

The render of a component that happens on mount, is the most trivial and obvious. Note, that if a component is included conditionally in a parent component, it will mount and unmount depending on that condition. This is not always what you want.

If we conditionally render the previously mentioned RemoteDropdown component in a parent component, thus toggling it on and off many times, we will many times load the external data and throw it away, wasting time and bandwidth on the same request. While network caching will help somewhat, we can mitigate this in two ways. We can either move the data storage and fetching up to a higher-level component, that is always included in the application, or we can conditionally render the component differently. You will see this sometimes in real components.

Normally we conditionally render a component like this:

```
return (
  <main>
    {hasDropdown && (<RemoteDropdown />)}                           #A
  </main>
);
```

#A If the boolean is true, we mount the component. If later set to false, the component is unmounted.

We can instead do it like this:

```
return (
  <main>
    <RemoteDropdown isVisible={hasDropdown} />    #A
  </main>
);
```

#A Here we always mount the component and simply toggle a flag as a property

We would need to modify the component to use this flag as an indicator about whether to render anything at all:

```
function RemoteDropdown({ isVisible }) {
  const [options, setOptions] = useState([]);    #A
  useEffect(() => {    #A
    // Loading happens here
  }, []);
  if (!isVisible) {    #B
    return null;
  }
  // Rest of component goes here
);
```

#A We first include all the hooks, that we need in the component
#B Only after all hooks have been evaluated, can we check if we need to render anything at all

While this approach to conditional rendering is generally not recommended (and the former approach is), this can be a handy tool in a situation, where you don't want your component to mount and unmount again and again, but want to keep it in the document all the time, but only sometimes actually render anything.

6.2.2 Rendering on parent render

This might come partially as a surprise, but every child component also renders, when their parent component renders.

Let's create this simple example of an icon inside a push button:

```
function Icon() {    #A
  return <img src="/arrow.png" alt="" />
);
function Button() {    #B
  const [enabled, setEnabled] = useState(false);
  const style = { border: `1px solid ${enabled ? 'red' : 'black'}`;
  return (
    <button style={style} onClick={() => setEnabled(b => !b)}>
      <Icon /> Toggle
    </button>
  );
}
```

#A The icon component is extremely simple - it never changes nor updates based on anything
#B The button component has internal state and renders every time the state changes

If we test this out in the browser, what do you think will happen? Every time we press the button, the enabled flag flips and the button renders again. But what about the icon? Will it render again (as in, will the function definition `Icon` be executed again)? Yes, it does. React does not assume that components are "pure", and they might not be. For that reason, React will render the component every time that the parent renders. If the component takes properties, React will render the component every time, regardless of whether these properties change or not.

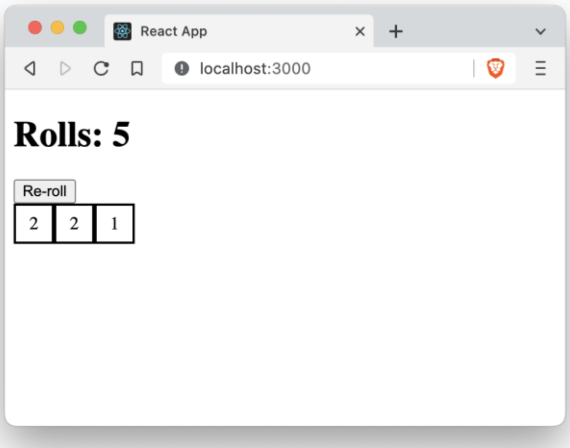
Let's imagine a different scenario, where we are actively utilizing this behavior. We can create a dice roller, where we can roll three dice. See the code in listing 6.6 and the output in figure 6.14.

Listing 6.6 A dice roller

```
import { useState } from 'react';
function Die() {
  const style = {
    border: '2px solid black',
    display: 'inline-block',
    width: 2em;
    height: 2em;
    text-align: center;
    line-height: 2;
  };
  const value = Math.floor(6*Math.random());    #A
  return <span style={style}>{value}</span>
}
function DiceRoller() {
  Const [rolls, setRolls] = useState(1);    #B
  return (
    <main>
      <h1>Rolls: {rolls}
      <button onClick={() => setRolls(r => r+1)}>Re-roll</button>    #C
      <div>
        <Die />
        <Die />
        <Die />
      </div>
    </main>
  );
}
```

#A Even though our Die component appears to be pure, it actually has an external source of information (Math.random) and (potentially) returns something new on every render
#B Our DiceRoller component is stateful
#C When we click the button, we increase the roll count, which forces a complete render of the component, which causes all the child components to render, giving us new dice values in turn.

`rq06-dice-roller`



Rolls: 5

```
2 2 1
```

Figure 6.14 Our dice roller after 5 rolls

On a general level, you should always put such variable content (such as the value of a die roll) inside component state, and not depend on it just magically updating on every render as in the above example. Please don't do this at home. This makes the above a terrible React design pattern. For example, we can't display the sum of the dice in the parent component, because it doesn't know the values of the child components.

A much better structure would be for the parent component to generate three random numbers and pass them to the dice as properties. But for demonstration purposes, this does highlight how even seemingly pure components render when their parent component does.

This behavior of child components always rendering is not necessarily desired though. See section 7.x for how to avoid this.

6.2.3 Rendering on state update

When you update the state inside a stateful hook (see the next chapter for more details about stateful hooks) your component using that hook will render. That's the whole purpose of updating the state, so this flow is pretty obvious and even desired.

But you can also render too often if your state contains data that updates often. You really want to avoid components rendering all the time, as it can be very CPU and/or memory intensive for the browser. This in turn will be annoying for the user.

A potential source of frequently updated information is something like the mouse position. A user can move the mouse around a lot - many times per second. And if you have multiple components that store the mouse position in state, you will have multiple components rendering many times per second, which will slow down your computer.

We will look at two different instances where you have such a setup and how you can minimize renders on state updates.

STORE HIGHER-LEVEL INFORMATION

Imagine a component that has a yellow background, when the cursor is on the left side and a blue background when the cursor is on the right side. Let's say the component is 200px wide, so if the cursor is more than 100px from the left edge, the cursor is on the right, otherwise, it's on the left.

The first way to implement this would be to save the cursor offset from the left side in a state value and check on each render, which cursor to display:

```
function BlinkingBackground() {
  const [offsetLeft, setOffsetLeft] = useState(0);    #A
  const onMouseMove = (evt) => setOffsetLeft(evt.offsetX);    #B
  const style = {
    backgroundColor: offsetLeft < 100 ? 'blue' : 'red',    #C
  };
  return <div style={style} onMouseOver={onMouseMove} />;
);
```

#A The state holds the mouse position
#B When the mouse moves, we record the mouse position in the state
#C Every time the component renders, we determine which color to use

This however wastes a ton of render cycles, because all the time you're moving the mouse around on only one side of the component, it'll render again and again for every position.

A much smarter approach is to just store in state, whether the cursor is on the left or right side and nothing else. Then our component only renders, when the cursor changes sides:

```
function BlinkingBackground() {
  const [isLeft, setLeft] = useState(true);    #A
  const onMouseMove = (evt) => setLeft(evt.offsetX < 100);    #B
  const style = { backgroundColor: isLeft ? 'blue' : 'red' };    #C
  return <div style={style} onMouseOver={onMouseMove} />;
);
```

#A The state holds only a boolean flag
#B When the mouse moves, we record the mouse position in the state
#C Every time the component renders, we determine which color to use

In this example, we utilize the fact that React only renders a component on a state update, if the state actually changed value. We call the setter function just as often as in the previous example, but because we only store a boolean value, which only changes from true to false as the cursor moves over the center of the component, most calls to the setter are simply ignored, as they don't actually alter the component state.

This new example results in a much cleaner data flow and now our component only renders, when something happens, that impacts the output.

MANIPULATE DOM ELEMENTS DIRECTLY

In this example, we have a component that moves an element in sync with the cursor at all times. It seems like we're doomed now. How can we update the style of an element in a component without storing it in component state?

For such a case, you might want to look at circumventing React and directly updating the DOM instead. To do this, we need a reference to the element in question (which requires another hook, `useRef`, that we will also introduce in chapter 7), and on mouse move, we will update the style of the element directly:

```
function PhantomCursor() {
  const element = useRef();    #A
  const onMouseMove = (evt) => {
    element.current.style.left = `${evt.offsetX}px`;    #B
    element.current.style.top = `${evt.offsetY}px`;    #B
  }
  return (
    <div style={{position: 'relative'}} onMouseMove={onMouseMove}>
      <img
        style={{position: 'absolute'}}
        ref={element}    #C
        src="/images/fake_cursor.png"
        alt=""
      />
    </div>
  );
);
```

#A We create a reference, that will point to our DOM element
#B Whenever the mouse moves, we directly update the DOM element through the reference
#C Remember to put the ref on the element, that we want to be able to manipulate

This component actually never re-renders. It renders as it is mounted and then it just stays around. The mouse event will change the appearance of the component, but that is outside the control of React. As seen from the perspective of React, this is a static component that doesn't ever change.

The problem here of course is if we need to use the mouse position for other things in our application - maybe we want to do some math on it, check collisions, etc. If so, we will have to store it in state anyway, but if at all possible, we want to avoid updating the state often.

6.2.4 Rendering inside functions

A component doesn't have to render directly inside another component. It can also render inside a function for example. If that's the case, the component will render every time the function runs. Sometimes such a function only runs when the parent component renders, so the result is the same, but other times you might have a function that can run at other times, causing render to happen at different times.

Imagine a button component that allows the parent to specify an icon for the component. The button is a push-button, so it has a state of either pressed or not pressed. Sometimes you want the icon to be different for those two states. You can make the component accept two different properties to use for the two states, or you can instead accept a function that will receive the state of the button as an argument, and then return the proper icon. Please see listing 6.7.

Listing 6.7 A push-button with an icon function

```
import { useState } from 'react';
function Icon({ type }) {    #A
  return <img src={`/images/${type}.png`} alt="" />;
}
function Button({ label, getIcon }) {
  const [pressed, setPressed] = useState(false);
  return (
    <button onClick={() => setPressed(p => !p)}>
      {getIcon(pressed)}    #B
      {label}
    </button>
  );
}
function LockButton() {
  const getIcon = pressed => pressed ?    #C
    <Icon type="lock" /> :
    <Icon type="unlock" />;
  return <Button label="Lock" getIcon={getIcon} />;
}
```

#A We have a general icon component, that embeds an image loaded from the right folder
#B Our button calls the getIcon function with its current state on every render
#C We define getIcon to return one of two icons.

In this setup, we render the icon component inside a function, and not directly inside our component. However, the function is only called inside the button component directly when it renders, so it is as if we include an icon conditionally directly in the render—we just do it through a function.

This does however change some of the things we know about components, and it can be a bit hard to optimize the above or even figure out exactly what's going on.

But we can also do the above in a much more familiar way. Take another look at that `getIcon` function. It's a function that returns JSX based on some arguments. Does that sound familiar? That's actually exactly what a functional component does. So we can alter this slightly to instead make the `getIcon` function into a custom component. Let's do that in listing 6.8.

Listing 6.8 A push-button with an icon component

```
import { useState } from 'react';
function Icon({ type }) {
  return <img src={`/images/${type}.png`} alt="" />;
}
function Button({ label, Icon }) {    #A
  const [pressed, setPressed] = useState(false);
  return (
    <button onClick={() => setPressed(p => !p)}>
      <Icon pressed={pressed} />    #B
      {label}
    </button>
  );
}
function LockIcon({ pressed }) {    #C
  return pressed ? <Icon type="lock" /> : <Icon type="unlock" />;
}
function LockButton() {
  return <Button label="Lock" Icon={LockIcon} />;    #D
}
```

#A The button component now expects a (capitalized) Icon property rather than a getIcon function as before
#B Because it's a component we expect, we can render it as such directly in the body
#C getIcon is now not just a function, but a fully-fledged functional component (by accepting properties rather than a simple argument)
#D Finally, we just supply LockIcon as a property–that's completely legal to do even though we haven't done it before

This works great—and looks so much cleaner! We can of course further optimize this (we can, for instance, move the ternary conditional to the property that changes inside the LockIcon component), but that is beside the point of this example.

The concept of providing functions that render JSX is called *render props* and was a pretty common approach in older React codebases. However, with functional components, almost all such cases are better solved by converting the argument to a full component as we do above. It makes the whole flow of data much easier to understand. This will solve ~95% of the cases of a function (that is *not* a functional component) rendering JSX.

REACT CONTEXT

One of the remaining reasons for rendering JSX in functions is if you use the non-hook version of the React Context API with a `MyContext.Consumer` component. This component takes a function as a child component. But that's quite a special case and not one that you are likely to encounter in a modern React codebase with functional components. If that does happen, you should check the online React documentation for how to use the React Context API. Or even better, convert the component to a functional component and use the `useContext` hook if possible – please see chapter 7 and again chapter 10 for more details on how to use this hook.

6.3 The lifecycle of a class-based component

When a class-based component mounts, renders, and unmounts, rather than using hooks, you can use lifecycle methods to react to the different stages in the component lifecycle.

The methods are named after what they do and where they fit into the lifecycle, so they're most of the time fairly self-evident.

Some lifecycle methods are executed in multiple events. And other lifecycle methods allow you to interfere with React's regular scheduling of component updates if you have inside knowledge that React doesn't have.

React used to have more lifecycle methods, but they've been deprecated in newer versions of React because of their troublesome behavior.

You ought to be using functional components, but if you do come across a class-based component, and you want to refactor it to a functional one, there are some general tips for how to make this conversion. Note that this is not an exact science, and rewriting or completely rethinking the feature might be required.

6.3.1 Lifecycle methods

When a component mounts, these class methods are called (in this order):

1. `constructor()`
2. `static getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

When a class-based component updates (for any of the previously mentioned reasons), the following methods are invoked in this order:

1. `static getDerivedStateFromProps()`
2. `shouldComponentUpdate()`
3. `render()`
4. `getSnapshotBeforeUpdate()`
5. `componentDidUpdate()`

Well, that's actually not completely true. `shouldComponentUpdate()` is special here, in that if defined, you can halt the render loop if you return `false`. It seems like a great way to minimize renders, but it can be very tricky to do and can if used incorrectly lead to components that are out of sync with their actual DOM representation.

When a component unmounts, the following method is invoked:

1. `componentDidUnmount()`

6.3.2 Legacy lifecycle methods

A number of lifecycle methods existed previously, that you might still see in some legacy codebases, as these were quite popular to use, but came with a lot of problems, hence their deprecation.

The methods have been renamed for now, but still exist in the React codebase - even in React 18. They will at some point be removed and not work anymore, but for now, they still work. Albeit, if you use them, you will be aware of how fragile they are based on their current naming.

The methods *were* called:

- `componentWillMount()`
- `componentWillUpdate()`
- `componentWillReceiveProps()`

All three have been renamed and you now have to create the following class methods to be able to use the functionality:

- `UNSAFE_componentWillMount()`
- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

Literally typing `UNSAFE` will get most developers to realize that they probably shouldn't be using this method or at least have a plan for how to get rid of it fairly soon.

We will not cover their functionality, as they are strongly discouraged. If you find them in a codebase, please check the online documentation for their functionality, so you're able to recreate the features without these methods.

6.3.3 Converting lifecycle methods to hooks

Converting a class-based component can be tricky. We've already seen how to deal with some of the tasks involved, which got a bit complicated as we introduced stateful components. Now that we add lifecycle methods, it gets even more daunting.

We will briefly cover each method below and describe how you can implement similar functionality using hooks.

- `constructor()`: This method can be implemented either using a `useEffect()` with no dependencies or if used for pre-calculating expensive values in useMemo() with no dependencies.
- `getDerivedStateFromProps()`: This can be implemented with a `useEffect()` hook with the relevant properties as dependencies.
- `render()`: The entire functional component is the render function.
- `componentDidMount()`: This method is mostly used for exactly what a `useEffect()` hook with no dependencies achieves. And it's often used together with `componentDidUnmount()` which is then the equivalent cleanup function for the hook. Note that to be technically correct `componentDidMount` runs *synchronously*, whereas `useEffect` runs *asynchronously*, so to achieve the *exact same effect*, you might have to use `useLayoutEffect`, but most of the time, `useEffect` will do just fine as the synchronous aspect is rarely a factor relevant for this lifecycle method.
- `shouldComponentUpdate()`: This method has no hooks equivalent, but is also not necessary when using hooks. If you want to minimize the renders of a functional component, use the approaches described in the next chapter.
- `getSnapshotBeforeUpdate()`: This is quite a weird method that's very seldomly used. It is actually almost exclusively used for a single specific purpose, which is to record the scroll position of some part of a component *before* the component updates, so you can restore that position *after* the component updates with new data. This specific behavior can be emulated in a functional component by wrapping the state setter in a custom function that records the old scroll position in a reference before updating the

component and causing a new render.

- `componentDidUpdate()`: This can be emulated with a `useEffect` hook with dependencies set to the relevant values that have changed and caused whatever changed behavior that you want to react to.
- `componentDidUnmount()`: Functionality in this method can be moved to a cleanup function in a `useEffect` (or `useLayoutEffect`) hook with no dependencies. This is often used to cancel subscriptions or intervals set on mount, so it goes together with the effect in the same hook.

6.4 Quiz

1. It is not possible to run side-effects inside functional components, only class-based components can do that. *True* or *false*?
2. When can you run an effect using an effect hook?
   a) As the component *mounts*
   b) As the component *unmounts*
   c) As the component *updates*
   d) All of the above
3. If you want to load data in a component as soon as it is displayed, but then not reload the data even if the component updates, your dependency array should:
   a) Be skipped
   b) Be empty
   c) Contain only the URL of the data
4. When a parent component renders, the child components only re-render if their properties update. *True* or *false*?
5. What is the correct syntax for an effect hook that only runs as the component unmounts:
   a) `useEffect(() => runOnUnmount(), []);`
   b) `useEffect(() => () => runOnUnmount(), []);`
   c) `useEffect(() => runOnUnmount());`
   d) `useEffect(() => () => runOnUnmount());`

6.5 Summary

- A React component has a lifecycle, that's individual for each instance of the component.
- The `useEffect` hook is the primary way to trigger side-effects as a component mounts, renders, and unmounts, that are relevant for the particular component.
- By carefully crafting the dependency array, you can trigger an effect hook to run at exactly the times you need it to run, which is how you can make smart components that interact with the browser, network, and user in many different ways.
- Components render whenever React determines that they need to under three main circumstances: when the component mounts, when the state of the component updates, and when the parent component updates.
- Class-based components cannot use hooks but rely on lifecycle methods for similar behavior. These can be converted to hooks, but the conversion is not always straightforward.

6.6 Quiz answers

1. *False*. Via the `useEffect` (and alternatively `useLayoutEffect`) hook, you can run side-effects inside functional components too.
2. The correct answer is *d*. You can run an effect on any particular render of the component and even as it unmounts.
3. The correct answer is *b*. If you want to run an effect only as a component mounts, you should supply an empty dependency array.
4. *False*. Any time a component renders, all the child components of that component will render too, regardless of whether their properties change or not.
5. The correct answer is *b*. An unmount (also known as a cleanup) effect has to be returned by the effect function, so double function notation is required. Also, the dependency array has to be empty, not skipped.

7 Hooks to fuel your web applications

This chapter covers

- **Creating stateful components in a larger perspective**
- **Optimizing performance using memoization**
- **Introducing advanced topics solvable by hooks**
- **Rules to observe when using hooks in general**

Hooks are what make modern React applications tick. They're a pretty small part of the overall React API, but very significant nonetheless. Hooks are also quite tricky to use.

In this chapter, we're going to discuss all the hooks and what they do, cover some important things about using hooks in general, and will at the end talk a bit about the dependency array that several hooks accept and what it means in practice.

Hooks are a special kind of creature in the React biosphere. From the outside, they seem completely unrelated in functionality, but when examined closer, they have some common traits and behaviors that we need to account for when using them. You could say that they stem from a common ancestor somewhere in the evolutionary tree, even though they have advanced to become very different beings.

We have dedicated this chapter to all the hooks for this very reason. So while we're going to be covering some wildly different topics, all of them are concerned with using hooks, and we'll try to tie a bow on it at the end by explaining how all of these hooks are in fact related despite their seemingly divergent purposes.

You've seen three different hooks up until now: `useState` (in chapter 5) plus `useEffect` and `useLayoutEffect` (in chapter 6). There are at this time 15 built-in hooks in React (as of React 18) and we're going to briefly cover all of them grouped by their functionality.

- **The stateful hooks**. These are functions that are concerned with making components and applications stateful on several different layers and levels of complexity: `useState`, `useReducer`, `useRef`, `useContext`, `useDeferredValue`, and `useTransition`.
- **The effect hooks**. These are functions that are concerned with running effects inside a component at different stages of the overall component lifecycle as well as during each individual render cycle: `useEffect` and `useLayoutEffect`.
- **The memoization hooks**. These are functions that are used for performance optimization via avoiding re-calculating values, if their constituent parts haven't changed: `useMemo`, `useCallback`, `useId`.
- **The library hooks**. These are advanced functions almost exclusively used in larger component libraries that are created to be shared either with the community or internally in a larger organization. These functions are rarely used in smaller or medium-size applications: `useDebugValue`, `useImperativeHandle`, `useInsertionEffect`, and `useSyncExternalStore`.

These 15 hooks are the built-in "base" hooks that React comes with. You can build more hooks on top of that, but you cannot build your own "base" hooks. You can only build hooks that utilize one or more of the existing hooks. We will be discussing custom hooks in future chapters a couple of times.

Do note that React might be extended with more built-in hooks in future releases. React 18.0 came with 5 new hooks, and incremental releases after React 18 might come with even more.

> **NOTE:** The source code for the examples in this chapter is available at https://github.com/rq2e/rq2e/tree/main/ch07. But as you learned in chapter 2, you can instantiate all the examples directly from the command line using a single command.

7.1 Stateful components

We've covered the topic in general in chapter 5, but we'll gladly re-iterate it here. Stateful components and in turn stateful applications are essential for web applications to actually be interesting to use.

An application without state is completely static. It will be identical for the entire time you have it open in the browser and it will be identical for every user using it. If you need login, if you need sessions, if you need interaction, if you need variability and changes over time, you need your application to be stateful.

However, stateful components are not all the same, just like not all states are the same. Some state is only kept briefly, some state is hyperlocal to an individual component, some state is application-wide, and state can be a single variable or a huge complex web of interdependent variables that have to update in unison.

In this section, we'll cover some different use cases for stateful components and applications, and discuss how to solve the given challenge via the proper hooks.

7.1.1 Simple state values with useState

`useState` is the bread-and-butter of stateful applications. You will probably find yourself using this hook for state the majority of times that you need it, so it is definitely an important one.

If you have a menu that can open and close, you keep the state of it in a local useState inside your menu component. It's a single simple value, it's only used inside this component and it's unrelated to any other state values in the application.

We discussed all the ins and outs of useState in chapter 5, so we're not going to go into further detail on this hook here. However, in the rest of this section, we're going to introduce some more complex scenarios, where `useState` just isn't enough or is sub-optimal.

7.1.2 Create complex state with useReducer

Imagine that you have a loader component, where you want to know whether loading succeeded or failed, you want to know what the error message is in case of failure, and you want to know what the data is in case of success.

The value of the error message is only relevant, if the loading failed. If loading succeeds, the error message is completely irrelevant and shouldn't even be set – and vice versa for the result data. This is an example of interdependent state. The individual values in the state depend on each other and you will often be updating multiple values at the same time.

`useReducer` is a stateful hook for exactly this purpose. It is an advanced version of `useState`, where you can alter your state in a more complex and controlled way (almost like a state machine, but not really) if you have a setup that's more complex than a single state value can reasonably represent.

Using a reducer is a way to generate a new state ("reduce") solely based on the current state and some action that takes some payload. The concept of *reducing state* is known from other frameworks such as Redux (hence the name), so it is already familiar to many React developers.

Do note, that `useReducer` is never strictly necessary - anything you can do with a reducer can also be done with a combination of simpler `useState`'s. There are many cases where you would likely want to use a reducer rather than settling for multiple disparate states to ensure a more strict data flow and better control.

We will present some examples of a reducer in chapter 10 when we move to more complex application architecture. A reducer is only really relevant for rather complex data flows, so it is not something we will use a lot in the simple applications that we're building throughout this book.

7.1.3 Remember a value without re-rendering with useRef

Imagine that you want to create a button that only works on double-clicking within a certain number of milliseconds. In order to create this, we need to remember how much time has passed between successive clicks. Remembering data inside a component is exactly what we have state for. We have a value, that we want to persist between renders, but we don't actually use it for rendering. The button does not change when we click the first time. We just need to remember a value inside a component instance for some amount of time, but not use the value to determine the output of the component.

`useRef` is both one of the simplest hooks in React and also one of the least understood hooks. It's a hook with a *passive state*. By that we mean the hook can contain state, but setting or updating the state does not cause a re-render.

`useRef` is used for a number of purposes including remembering values between renders and serving as a reference to DOM elements used in the render. The latter is a very important use case (and the reason for the name, `useRef`), as it is the best and simplest way to address DOM elements through script in your components.

PASSIVE STATE VALUES

You can use the `useRef` hook to remember some value that is relevant between renders of the component, but doesn't directly impact the outcome of the component.

That sounds a bit complex and even rare. When would you have such a value in a component? Let's as an example re-create our counter component once again, but this time with the added functionality, that the increment button only works if double-pressed.

We need to store the time of the last click event somewhere in our component, and it needs to be in a place that persists between renders. We already know that we can store such a value in a state provided by the `useState` hook. Let's try to sketch this scenario in figure 7.1.

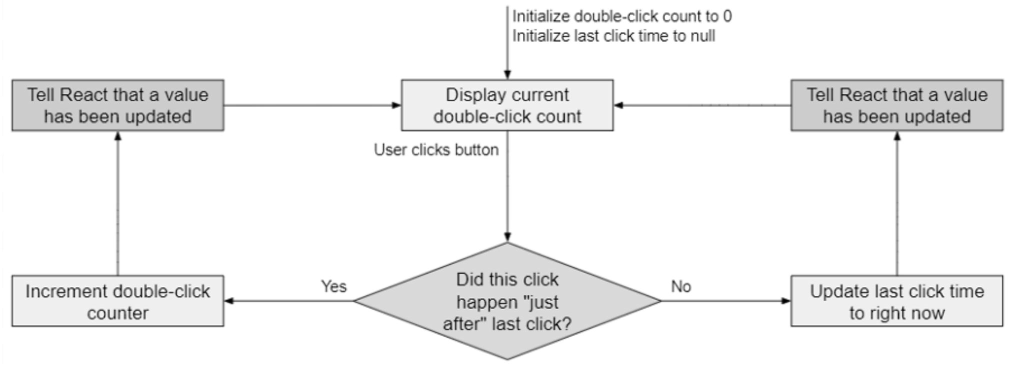

Initialize double-click count to 0
Initialize last click time to null
Tell React that a value has been updated
Display current double-click count
Tell React that a value has been updated
User clicks button
Increment double-click counter
Yes
Did this click happen "just after" last click?
No
Update last click time to right now

Figure 7.1 As the user clicks the button, the execution bifurcates based on whether this click happens within a very short time of the last recorded click.

Let's go ahead and implement this in listing 7.1.

Listing 7.1 A double-click counter with useState

```
import { useState } from 'react';
const THRESHOLD = 300;
function DoubleClickCounter({ from }) {
  const [counter, setCounter] = useState(0);
  const [lastClickTime, setLastClickTime] = useState(null);    #A
  const onClick = () => {
    const isDoubleClick = Date.now() - lastClickTime < THRESHOLD;    #B
    if (isDoubleClick) {
      setCounter(value => value + 1);    #C
    } else {
      setLastClickTime(Date.now());     #D
    }
  };
  return (
    <main>
      <p>Counter: {counter}</p>
      <button onClick={onClick}>Increment</button>
    </main>
  );
}
```

#A We remember the time of the last click in a state value
#B If the time since the last click is less than 300 ms, it's a double click
#C Only if it's a double-click do we actually increment the counter
#D Otherwise we just remember the time of the current click

This is however not necessary and will cause needless extra rerenders. When we call `setLastClickTime`, React will rerender the component because a state value changes. However, nothing actually updated in the component and the exact same DOM output will be rendered to the screen. The code in listing 7.1 works, but it is less than optimal.

Because we only need the state value internally in the component and we don't need the component to rerender just because the value is updated, we can instead use a reference via the `useRef` hook.

To instantiate a `useRef` hook, you simply call the hook and store it in a variable. You can optionally pass an initial value to the hook as well. To read or update the current value of a useRef hook, you access the `.current` property on the hook return value. Let's sketch this in figure 7.2 and compare this with 7.1. We skip an entire cycle of rendering!

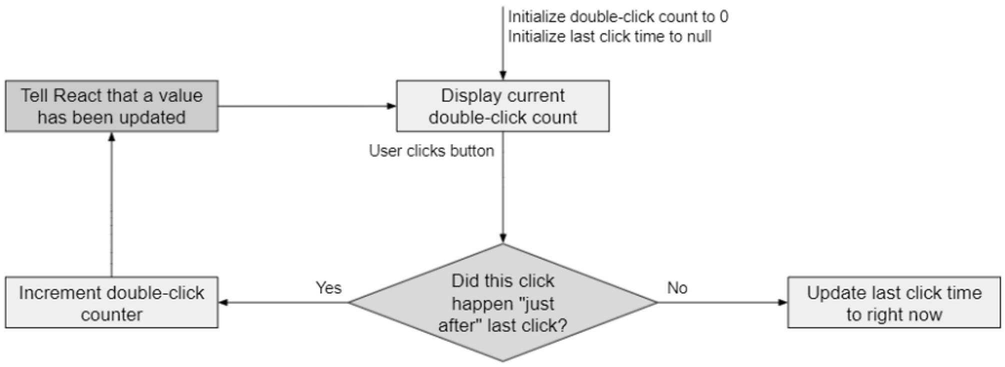

Initialize double-click count to 0
Initialize last click time to null
Display current double-click count
User clicks button
Did this click happen "just after" last click?
Yes
Increment double-click counter
Tell React that a value has been updated
No
Update last click time to right now

Figure 7.2 This time, if the user clicks the button the first time, the last click time is recorded, but it does not cause a re-render, because it is not an active state value, but a passive one. This means that we can still access the value later (on this or future renders), but it does not cause a new render by itself.

See this implemented in listing 7.2.

Listing 7.2 A double-click counter with useRef

```
import { useState, useRef } from 'react';
const THRESHOLD = 300;
function DoubleClickCounter({ from }) {
  const [counter, setCounter] = useState(0);
  const lastClickTime = useRef(null);    #A
  const onClick = () => {
    const isDoubleClick = Date.now() - lastClickTime.current < THRESHOLD;    #B
    if (isDoubleClick) {
      setCounter(value => value + 1);
    } else {
      lastClickTime.current = Date.now();    #C
    }
  };
  return (
    <main>
      <p>Counter: {counter}</p>
      <button onClick={onClick}>Increment</button>
    </main>
  );
}
```

#A We remember the time of the last click in a useRef value
#B We do the same check as before, except now we access the value through the .current property
#C We also update the current value of the state through the .current property

rq07-double-counter

This is a much better version of our component, because we don't have needless rerenders. We persist the state properly in a way that works, even if the component does rerender for some other reason.

REFERENCES TO DOM ELEMENTS

The other use case for `useRef` is as mentioned to get references to DOM elements. You will see this used many times throughout the remainder of this book. We use it to have a reference to the actual DOM element that is rendered in the document as a consequence of the JSX element that we have created. The syntax is very simple as follows:

```
function Component() {
  const ref = useRef();          #A
  return <div ref={ref} />;      #B
}
```

#A We create a reference using the hook
#B We "assign" the reference to the DOM node, which actually when the component is rendered assigns a reference to the DOM node inside the ref object

This doesn't actually use the reference for anything, but merely creates it. You can use the reference in an effect hook to e.g. invoke methods on the element, that isn't directly possible through properties of the DOM element.

Let's for instance auto-focus an input field when a component is mounted:

```
function AutoFocusInput() {
  const ref = useRef();
  useEffect(() => ref.current.focus(), []);    #A
  return <input ref={ref} />;
}
```

#A We create an effect that runs on mount only (empty dependency array) and focuses the element through the ref object. Note that it uses the ref.current syntax as mentioned previously.

This latter use of `useRef` is the more common use (and the originally intended purpose). There's quite a bit more to say about how both `useRef` and the JSX `ref` property works, but we will leave it here for now. We will expand a bit on this in future chapters where relevant.

7.1.4 Easier multi-component state with useContext

`useContext` is a stateful hook, meaning that it works similarly to `useState`. But rather than load and update values in a local store, `useContext` works in a store in a parent component somewhere up the component tree. This is the hook-version of the React Context API, and we will see a lot more about how this works in practice in chapter 10.

We can reveal that it is one of the more powerful hooks when it comes to building good architecture.

7.1.5 Low priority state updates with useDeferredValue and useTransition

Note: This topic is both fairly advanced and also brand-new. These are features only just added to React, so there's still a lot of discoveries around best-practices yet to be made. It is also an advanced topic that probably isn't relevant for every-day applications.

Imagine that you have created an online application like Google Docs. It is an online document editor, and you have added many features to this application. You are now adding spell-checking to the application and you want to do this by adding a button that can be clicked to enable spell-checking and clicking again to disable it. The button has a different background color if enabled.

When a user clicks this button, it should react instantly. The button should change display within a few milliseconds for the user to feel like the button works. We can implement this using state - we just toggle an enabled property inside the button using a `useState` setter.

But our button is of course going to do more than that. When spell-check is enabled, all the typos in the document should be highlighted with a red line underneath them. If the user is working on a large document, finding and highlighting all the errors might be an expensive operation. You have to run a lot of operations on all the words in the document to find errors and maybe even find suggestions for correct spellings for each. This sounds like something that can easily take tenths of a second or even whole seconds to execute.

If we update both the internal enabled state flag inside the button as well as the global flag that triggers the spell-check calculation with the same priority at the same time in React, React's internals will treat both updates as happening at the same time and React will not render anything until both updates have been fully calculated.

This is not ideal, because it will make the spell-check button seem non-functional. The user presses the button, nothing happens and they press it again. Then all of a sudden the button actually works once the calculation is completed, but because the user pressed it again, the user disables the functionality again. That's a horrible user experience.

What if we could inform React, that updating the button state is high-priority and should happen right away, whereas highlighting all the spelling errors in the document is lower priority and we don't mind if that trails the button click by several render cycles?

React 18 introduced a brand-new concept, Concurrent Mode, and it allows exactly that. The two hooks `useDeferredValue` and `useTransition` are two different ways of specifying low-priority state updates from two different angles.

Given the complexity of these hooks, we're not going to cover them further in this book, but please refer to the following online materials for further information:

- Article: https://academind.com/tutorials/react-usetransition-vs-usedeferredvalue
- Video: https://www.youtube.com/watch?v=lDukIAymutM

7.2 Component effects

This section is going to be short, so try not to blink or you might miss the whole action!

Component effects are a group of hooks dedicated to running side-effects from within hooks to either influence the outside based on the component state, update the component state based on something from the outside, or even do both at the same time.

We've seen two such hooks already presented in the previous chapter, `useEffect` and `useLayoutEffect`. There is only one more such hook, `useInsertionEffect`, but it is "reserved" for advanced use by specific libraries, so it is not recommended to be used by "regular" developers.

We're not going to add any information on the `useEffect` and `useLayoutEffect` in this chapter, as we covered everything there is to know about it in the previous one. With a single exception though. We only briefly touched upon the concept of dependency arrays in the previous chapter and we will expand that coverage in this chapter, but we are saving that for section 7.6, as we first have to cover some other hooks that also use dependency arrays.

The last effect hook, `useInsertionEffect`, will be briefly covered in section 7.4.

7.3 Optimizing performance by minimizing re-rendering

We have already several times mentioned the importance of minimizing unnecessary renders. But just how important is it and what are the tools we can use?

JavaScript tries to run at about 60 frames per second in most browsers. That is only 16ms per frame! And each time React renders one or more components, it will count as one frame. So if your entire React render takes more than 16ms, your browser will start to treat your script as "slow" and start dropping frames. For some applications, this won't matter, but if you have a lot of animations and other moving elements, it will matter and users will notice.

Secondarily, response time matters. Research shows that a user interface should update within 0.1 second for reaction to seem *instantaneous* to the user. If you're slower than that, your users will notice and it might result in users double-clicking buttons, because the first click doesn't work, or users will simply leave in annoyance, because the app feels slow.

In React applications, the new state of a component will often only render on the next frame, so you're already losing 16ms from click to reaction. You really should do your utmost to make the best use of the remaining time, and updating every component in the application every time the user clicks anything is definitely not it.

There are many ways to optimize JavaScript applications in general and React applications in particular, and we're not going to cover all of them, only the ones particularly relevant to React. One method that we already discussed was minimizing state updates, so we only update state, so a state update actually means a different render in the component.

Another very common tool in the React toolbox is *memoization*. We already mentioned it a few times, but in this section we're going to see a lot more examples of this in use.

What is memoization?

Memoization is the concept of remembering the last input and output of a *pure* function, and if the function is invoked with those exact same inputs the next time around, return the already calculated value rather than call the function again.

Note that this only makes sense for pure functions whose return value depends solely on their inputs and never any outside information nor any randomness.

This can also sometimes be considered caching, but where caching often remembers many different values for many different inputs, memoization normally only remembers the latest call to a given function and checks whether the next call is equivalent and then reuses the previous answer.

React has a function that can memoize any function. It is simply called `memo()` and can be imported from the React package:

```
import { memo } from 'react';
const rawAddition = (a,  b) => a + b;
const addition = memo(add);
```

In the above example, if we invoked `rawAddition()` with the same values again and again, the calculation would be performed over and over. But if we invoked `addition()` with the same values multiple times, the calculation would only be performed the first time and the response would be cached. The cached response would be returned for the subsequent invocations as long as the inputs remained the same.

We can memoize bits of React applications in three ways:

- We can memoize an entire component
- We can memoize a bit of JSX
- We can memoize a property to be passed to a component

We will discuss all of these approaches with examples in the next subsections. After going through these examples of where memoization is required, we're going to discuss the hooks that we use to achieve this in some more technical detail at the end of this section.

7.3.1 Memoize a component

We already mentioned something that you might find a bit weird: When a component renders, all child components also render, regardless of whether they have actually changed or not.

This includes components that are completely self-contained and don't even take properties, but just render a static bit of JSX. Also component accepting properties will render, even if given the exact same properties again.

We can use the `memo()` function from the React module to memoize an entire component. This means that if it is invoked again with the exact same properties (or lack thereof), it will not render again, but just use the exact same response already calculated once.

React will in such a case optimize the reconciliation of your component into the browser DOM and realize that no new information has been created, so the DOM does not even need to be compared to the JSX. React knows that because the JSX is not just similar, but exactly the same as before, the DOM will already be correct. This can save a lot of time!

Let's create a todo-list application, which allows users to add new todos to the list. While typing in the input field, we will update the internal state of the title of the todo to be added. This is a common approach for a *controlled input field* (much more about that in chapter 8), but it does also cause a lot of renders. In our first instance, we will do this without memoization in listing 7.4.

Listing 7.4 A todo-list without memoization

```
import { useState } from 'react';
function Items({ items }) {    #A
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map(todo => <li key={todo}>{todo}</li>)}
      </ul>
    </>
  );
}
function Todo() {
  const [items, setItems] = useState(['Clean gutter', 'Do dishes']);
  const [newItem, setNewItem] = useState('');
  const onSubmit = (evt) => {
    setItems(items => items.concat([newItem]));
    setNewItem('');
    evt.preventDefault();
  };
  const onChange = (evt) => setNewItem(evt.target.value);    #B
  return (
    <main>
      <Items items={items} />    #C
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
```

#A Our Items component just render the items it receives - it does not have any state itself
#B The Todo component does have state, and it is updated every time the user types in the input field
#C Because the state updates on every key entered, the JSX returned in the todo list is also re-generated every time causing the Items component to render every time

`rq07-todo-naive`

If we spin this up in the browser, we will see that it works. It is a decent attempt at a simple todo application. But if we open up the performance tools in our browser of choice to see what happens every time you type in the input field, you will see that the browser can spend up to 5ms handling the keypress event. That might not sound like a lot, but remember we only have 16ms per frame, and we're already spending about a third of that just handling a single input field. If we have other things happening in the application, we're quickly running behind!

Now let's try to do just one thing. Let's memoize the items component using the `memo()` function imported from the React module in listing 7.5.

Listing 7.5 A todo-list with memoization

```
import { memo, useState } from 'react';    #A
const Items = memo(function Items({ items }) {    #B
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map(todo => <li key={todo}>{todo}</li>)}
      </ul>
    </>
  );
});
function Todo() {
  const [items, setItems] = useState(['Clean gutter', 'Do dishes']);
  const [newItem, setNewItem] = useState('');
  const onSubmit = (evt) => {
    setItems(items => items.concat([newItem]));
    setNewItem('');
    evt.preventDefault();
  };
  const onChange = (evt) => setNewItem(evt.target.value);
  return (
    <main>
      <Items items={items} />
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
```

#A Remember to import memo()
#B Then apply it to the whole component

`rq07-todo-memo`

With just this minor optimization, our render for every keypress now drops to about 2ms - simply because we don't need to render the whole list every time, looping over every entry, creating a new JSX element for every todo item already in the list.

That's a very significant time save with a minimal amount of work. Extra good job, us!

7.3.2 Memoize part of a component

In the above scenario, the items were rendered in a different component, so we had the luxury option of just memoizing the entire component. But that's not always the case. Sometimes the relevant part of the JSX covers multiple components. Imagine for example, that we did not have an Items component, but instead rendered the list items directly in the Todo component. What can we do then?

There are two things we can do. We can move the section of the component, that is often unchanged, to a new separate component and memoize that component, which would take us directly back to listing 7.5. The other option is to use the `useMemo` hook to memoize the JSX directly in the parent component. Let's do that in listing 7.6.

Listing 7.6 A todo-list with memo hook

```
import { useMemo, useState } from 'react';    #A
function Todo() {
  const [items, setItems] = useState(['Clean gutter', 'Do dishes']);
  const [newItem, setNewItem] = useState('');
  const onSubmit = (evt) => {
    setItems(items => items.concat([newItem]));
    setNewItem('');
    evt.preventDefault();
  };
  const itemsRendered = useMemo(    #B
    () => (
      <>
        <h2>Todo items</h2>
        <ul>
          {items.map(todo => <li key={todo}>{todo}</li>)}
        </ul>
      </>
    ),
    [items],    #C
  );
  const onChange = (evt) => setNewItem(evt.target.value);
  return (
    <main>
      {itemsRendered}
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
```

#A We import useMemo() rather than memo()
#B We render the items JSX inside the useMemo hook
#C Remember to add items as a dependency of the hook. If not, the list never updates even as you add items!

`rq07-todo-usememo`

This once again runs in 2ms while typing. If you remove the `useMemo` hook and just render the items directly inline in the component, runtime jumps back up to 3-4ms per event (less than before, because we use less components, but still a lot more).

We can probably argue about which version of the todo application has the *cleanest* code. We would probably personally prefer the version using a memoized component in listing 7.5, but others would prefer the one in listing 7.6. The two different approaches allow us to choose either option as we see fit.

7.3.3 Memoize properties to memoized components

Let's go back to our todo application example in listing 7.x, but now we add a new requirement. We always want to display a todo item of *"Complete todo list"* at the top of the list, regardless of what is in the list of items. Let's do that in listing 7.7.

Listing 7.7 A todo-list with a fixed item

```
import { memo, useState } from 'react';
const Items = memo(function Items({ items }) {
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map(todo => <li key={todo}>{todo}</li>)}
      </ul>
    </>
  );
});
function Todo() {
  const [items, setItems] = useState(['Clean gutter', 'Do dishes']);
  const [newItem, setNewItem] = useState('');
  const onSubmit = (evt) => {
    setItems(items => items.concat([newItem]));
    setNewItem('');
    evt.preventDefault();
  };
  const onChange = (evt) => setNewItem(evt.target.value);
  return (
    <main>
      <Items items={['Complete todo list', ...items]} />    #A
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
```

#A The only change from 7.5 is this line. Imn stead of just passing items as the property, we create a new array with a fixed item at the start and then the rest of the items as before

If you spin this up in the browser and check the runtime per keypress event, you will see that it now jumped back up to around 5ms again. What happened?

The problem is, that even if the state value is identical (even *referentially identical*) on every render while we're typing, we create a new array inline on every render, which has the extra item in the front. We then pass *that newly created array* to the memoized component, but because the passed value isn't referentially identical every time, the component has to do a full render.

The good thing here is though that we already know how to fix this! We need to create a value in the component that will only change, when the state value changes. That's what we have the `useMemo` hook for. Let's apply that in listing 7.8.

Listing 7.8 A todo-list with a memoized fixed item

```
import { memo, useMemo, useState } from 'react';
const Items = memo(function Items({ items }) {
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map(todo => <li key={todo}>{todo}</li>)}
      </ul>
    </>
  );
});
function Todo() {
  const [items, setItems] = useState(['Clean gutter', 'Do dishes']);
  const [newItem, setNewItem] = useState('');
  const onSubmit = (evt) => {
    setItems(items => items.concat([newItem]));
    setNewItem('');
    evt.preventDefault();
  };
  const onChange = (evt) => setNewItem(evt.target.value);
  const allItems = useMemo(
    () => ['Complete todo list', ...items],
    [items],    #A
  );
  return (
    <main>
      <Items items={allItems} />    #B
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
```

#A We now memoize the inline created array and save that in a variable allItems. This hook depends on the items array, so only when that array changes do we actually need to update the allItems value.
#B We then pass the memoized property to the items component, which will now correctly memoize and only render, when the list is updated with new content.

rq07-todo-fixed

That's it. We did it! We fixed the render and the performance is now back down to about 2ms per render while typing.

To make this a better todo list, we should be able to delete items from the list once completed. Let's add a callback to our items component that will be invoked on click with the relevant item to remove. We will for simplicity go back to the todo list without the fixed item in the front, but you could easily do both things at the same time.

If we just do this the naive way, we will be back at the problem from before:

```
<Items
  items={items}
  onDelete={(item) => setItems(ls => ls.filter(i => i === item))}
/>
```

Because this is a function defined inline in JavaScript, we essentially create a completely new function every time, even though they appear to be identical. That's a no go when we're using memoization. We need our values to be referentially identical if memoization is supposed to kick in. We need to memoize this callback. And what better way to do that, than using the `useCallback` hook made for just this purpose. Let's do this in listing 7.9.

Listing 7.9 A todo-list with deletable items

```
import { memo, useCallback, useState } from 'react';
const Items = memo(function Items({ items, onDelete }) {
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map(todo => (
          <li key={todo}>
            {todo}
            <button onClick={() => onDelete(todo)}>X</button>
          </li>
        ))}
      </ul>
    </>
  );
});
function Todo() {
  const [items, setItems] = useState(['Clean gutter', 'Do dishes']);
  const [newItem, setNewItem] = useState('');
  const onSubmit = (evt) => {
    setItems(items => items.concat([newItem]));
    setNewItem('');
    evt.preventDefault();
  };
  const onChange = (evt) => setNewItem(evt.target.value);
  const onDelete = useCallback(    #A
    (item) => setItems(list => list.filter(i => i !== item)),
    [],    #B
  );
  return (
    <main>
      <Items items={items} onDelete={onDelete} />
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
```

#A We memoize the callback in a hook. We could also use the useMemo hook, but this is simpler
#B We even pass an empty dependency array, because our only dependency is a state setter, which is a known stable value

`rq07-todo-delete`

We're now back where we wanted to be. Our component renders swiftly even while typing, because the "expensive" part of the component, that doesn't actually change while typing, is properly memoized.

You will often see that it is necessary to memoize properties, once you start memoizing components. This is relevant for objects and arrays created inline, but even more so for functions. This is the primary reason for the existence of the useCallback hook and why it is so often used in React.

Memoization is not just a job left for the end of a project when you start to notice that your application is running a bit sluggish. Memoization is something you do all the time when developing, to ensure a smooth and optimal user experience. With the tools presented in this section, you should be able to apply this to your own projects.

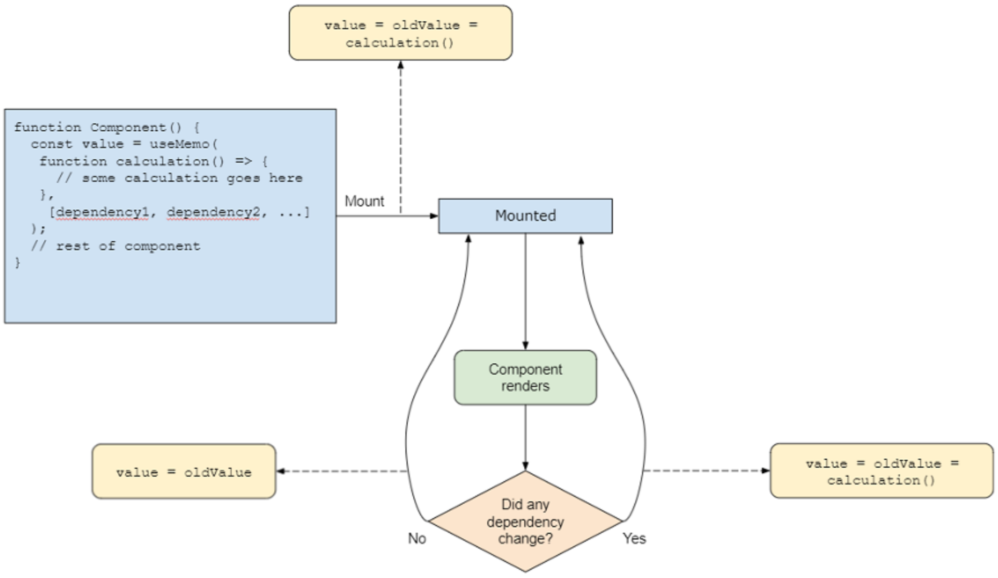
7.3.4 Memoize any value with useMemo

This hook is used to memoize values between renders. It can be used for two different purposes (or both at once):

- Avoid expensive recalculations
- Maintain *referential equality*

While the latter concept is pretty easy to grasp, the former is a lot more complex. We will get back to why *referential equality* matters in section 7.3, where we talk more about dependency arrays, and we will revisit this again in section 7.4, where we talk about minimizing re-rendering.

`useMemo` takes a function and a dependency array. If any value in the dependency array has changed since this component was last rendered, the function will be executed and the return value of said function will be the return value of the call to `useMemo`. If no value in the dependency array has changed since the last render, the value returned in the last render will be returned again. This is illustrated in figure 7.3.



```
function Component() {
  const value = useMemo(
   function calculation() => {
     // some calculation goes here
   },
    [dependency1, dependency2, ...]
  );
  // rest of component
}
```

value = oldValue = calculation()
Mount
Mounted
Component renders
value = oldValue
Did any dependency change?
No
Yes
value = oldValue = calculation()

Figure 7.3: The useMemo flow. Note that oldValue doesn't actually exist in the component. It is an internal value only accessible to the React API, not directly visible in the component.

Using this hook to avoid expensive recalculations is relatively obvious to grasp.

Let's for example say, that we have a list of employees and we have some filters above the list, where we can filter which employees to see in the list. For this example, let's just allow the user to either see all employees or temporary workers only.

We could perform this filtering of the employee list every time the component renders, but that could be expensive, e.g. if the list contains 1000+ records. Instead, we will use the `useMemo` hook to only do the filtering, when either the source array or the filter changes value–see listing 7.3.

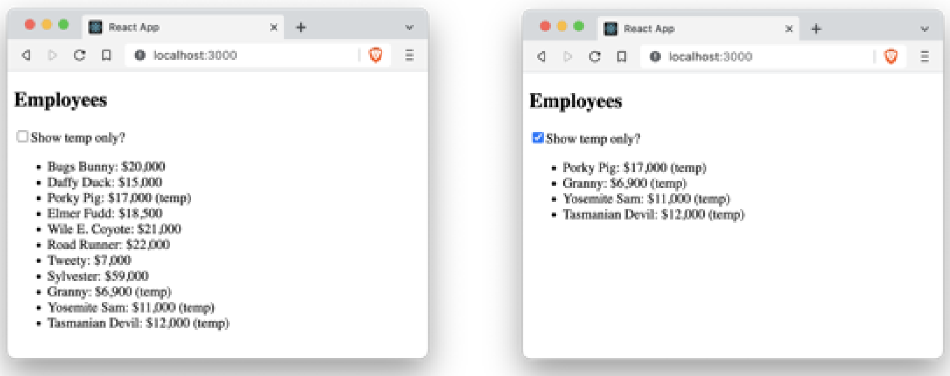
Listing 7.3 Memoized filtering

```
import { useMemo, useState } from 'react';    #A
function Employees({ employeeList }) {
  Const [showTempOnly, setShowTempOnly] = useState(false);
  const filteredList = useMemo(    #B
    () => employeeList    #C
      .filter(
        ({ isTemporary }) =>
          showTempOnly ? isTemporary : true
      ),
    [employeeList, showTempOnly],    #D
  );
  return (
    <section>
      <label>
        <input type="checkbox" onChange={() => setShowTempOnly(f => !f)} />
        Show temp only?
      </label>
      <ul>
        {filteredList.map(({id, name, salary, isTemporary}) => (
          <li key={id}>
            {name}: {salary} {isTemporary && '(temp)'}
          </li>
        ))}
      </ul>
    </section>
  );
}
```

#A We import the useMemo hook (along with useState) from the React package
#B We then invoke useMemo with the two arguments it takes
#C First argument is a function, that performs a filtering on the current array
#D Second argument is the dependencies of our calculation—it should be re-performed only if either the list or the boolean changes

`rq07-employee-list`

You can see this application in action in figure 7.4 with the boolean flag turned on and off.



Employees

☐ Show temp only?

- Bugs Bunny: $20,000
- Daffy Duck: $15,000
- Porky Pig: $17,000 (temp)
- Elmer Fudd: $18,500
- Wile E. Coyote: $21,000
- Road Runner: $22,000
- Tweety: $7,000
- Sylvester: $59,000
- Granny: $6,900 (temp)
- Yosemite Sam: $11,000 (temp)
- Tasmanian Devil: $12,000 (temp)

Employees

☑ Show temp only?

- Porky Pig: $17,000 (temp)
- Granny: $6,900 (temp)
- Yosemite Sam: $11,000 (temp)
- Tasmanian Devil: $12,000 (temp)

Figure 7.4 This shows the list of employees both with and without the filter enabled.

Actually, this is not a good example—*if* that's all there is. Why is that? The only things that can change in this component are the list and the boolean, so every time the component renders, one of those two will (probably) have changed. This means that we need to do the calculation on every render because the dependencies always change. And if we need to do the calculation on every render, using memoization is actually more expensive than not (because of the overhead of calling extra functions).

However, if our component had a bunch of other properties and state values, that could change independently of the above state and property values, then our optimization starts to help out a lot.

Imagine the component having some other state, that did not refer to the filtering of the list, but for example the sorting of the list. If this was the case, our filtering calculation would not be performed simply because we sorted the list differently. The filtering would only be (re)applied when we updated the boolean filter flag or if the source array changed, which is what we want in this case.

The second use-case, and probably the more common use-case of `useMemo`, is for *referential equality*. We'll discuss this further in section 7.6.

7.3.5 Memoize functions with useCallback

`useCallback` is actually one of the least necessary hooks built-in to React, as it's a very simple extension of `useMemo`. It does the same thing as `useMemo` if `useMemo` is used to memoize a function.

`useCallback` could be defined in terms of `useMemo` simply like this:

```
const useCallback = (fn, deps) => useMemo(() => fn, deps);
```

This is even stated directly in the React documentation.

So why do both hooks exist? `useCallback` is used to make memoized callbacks where *referential equality* is desired (and never to avoid expensive calculations). They are most often defined inline in the component body like so:

```
const handleClick = useCallback(
  (evt) => {
    // handle evt and do stuff
  },
  [],
);
```

If we wanted to define this same memoized function using `useMemo`, we would do it like this:

```
const handleClick = useMemo(
  () => (evt) => {    #A
    // handle evt and do stuff
  },
  [],
);
```

#A This notation looks a little weird

That double arrow notation at `#A` is very easy to forget—and forgetting that completely changes the meaning of the code. Rather than assigning a function to `handleClick` as desired, if we forgot the double arrow, we would instead assign the result of invoking said callback, which is most likely `undefined`, as we rarely return anything from event handlers.

Despite this hook only being able to do a subset of what `useMemo` can do, we will be using `useCallback` more than `useMemo` in the remainder of this book. That's because we memoize callbacks more often than other types of values.

7.3.6 Create stable DOM identifiers with useId

> **NOTE:** **This is a pretty advanced topic only relevant for server-generated React. Feel free to skip this subsection, as it requires quite a built-up of knowledge to be able to understand the circumstances for this hook that has an extremely narrow usage.**

When creating production-ready web applications, it is often a goal to have the server generate the HTML before it reaches the client in order for search engines and sharing bots to be able to read the content. If you share a link on Twitter or LinkedIn, or if you search for information using Google, Bing or even Siri, these services need to be able to crawl your site contents in order to display what's on the page.

However, many such services will not run JavaScript on the page before determining the contents, so you will have to make sure that the HTML output from the server contains all the relevant information that is needed to share this information.

Imagine that you have an accordion on your page with frequently asked questions about your product which exists somewhere on your website built completely in React. One of the questions is about which age your product is adequate for. If someone asks Siri *"what age is* *product X designed for?"*, you want to make sure that Siri can read this answer from your page.

In order to generate the HTML on your server, you need to run your React application on your server as well as in the browser. This is a whole topic all on its own (and many books have been written about it). We will briefly cover the topic of server-generated React in chapter 18.

React has a magic trick, where it can *rehydrate* a React application based on a previously rendered HTML output. If you render your React application (in its very initial state) on the server, then React can run in the browser and "pick up" where the server left off by re-using all the already exported HTML – as long as that HTML is *identical* to what it would be as rendered by the React application in the browser.

For the purpose of this section, we're going to cover a single tiny bit of the problems of server-generated React. When you create a semantic accordion on a webpage, you should make sure to associate your buttons with the panels that they expand. You create this association using IDs and aria-properties in the HTML output. A common tactic in React is to create an Accordion component and generate a random ID for each button and panel combination and just use that variable for the two components.

However, note how we just mentioned, that when React rehydrates in the browser, it expects the server-generated HTML to be identical to the browser-generated one, but we're using randomly-generated IDs in the HTML to link elements, which will of course be generated differently on different renders.

This is the very narrow use-case that the `useId` hook solves. It makes sure that for two completely identical component trees, if any component inside this tree calls `useId`, it will get the exact same ID returned regardless of which platform the hook is run on.

You can read more about this hook and this use-case in the following article: https://blog.saeloun.com/2021/12/09/react-18-useid-api.

7.4 Create complex component libraries

This section is only included for completion, so we cover all the hooks in React. The 4 hooks described in the next subsections are all very advanced and only rarely used. They are meant for reusable packages such as component libraries or open-source modules.

The last two hooks mentioned in this section are introduced in React 18 as a consequence of the new Concurrent Mode. Some libraries have to be updated to correctly render in Concurrent Mode to avoid calculating logic that isn't actually required or is premature because of concurrency.

Feel free to skip this section and go on to section 7.5 if you want to get on with the more practical stuff.

7.4.1 Create component APIs with useImperativeHandle

This hook is used for advanced component libraries where you want to expose an API to parent components, that it's either custom for your particular component or mimicks a built-in DOM element for ease of use. It is almost exclusively used with `forwardRef`, which allows you to create your own components that accept refs, but either pass them on to other elements or allows you to make a custom reference.

A quick example of this would be a generalized custom input component, where you want the parent component to be able to focus the input. Maybe you have an error message saying "missing field" and if the user clicks the error message, the correct field is focused. However, inside your component, the input can be many different types of elements (input, text area, or select) and can maybe even have multiple input fields (imagine a phone form field, which would consist of both a country prefix field and another phone number field).

In order to generalize this and make a unified API for all of these cases, you can use the hook `useImperativeHandle` to expose a `focus()` method for your component, that can be used in imperative code (rather than declarative code through props only), which will make sure to focus the proper element when invoked.

We will not go into details about how this hook nor how `forwardRef` works in this book, as that is an advanced chapter beyond this scope, but do know that this hook exists if you want to create an advanced custom component that exposes a custom API through a reference.

7.4.2 Better debugging of hooks with useDebugValue

This is a hook only meant for developer experience (DX). It does not change nor improve the user experience (UX) of your application regardless of how it is used.

The `useDebugValue` hook allows you as a React library developer to display a custom message when other developers are inspecting your custom hooks in their React application using the React DevTools plugin in their browser.

Normally a custom hook would display all its internal states in the React DevTools explorer, but that might be confusing to someone who doesn't really care about the internals of your custom hook. With the `useDebugValue` hook you can expose only what the developer using your hook cares about.

7.4.3 Synchronize non-React data with useSyncExternalStore

In Concurrent Mode it can happen that React is updating a state value with low priority and while calculating the consequences of said update, an urgent update comes in that has to be calculated irrespectively of the incomplete update. Because React is running concurrently, it means that React will actually have several completely separate instances of the application running and can thus spin off a new calculation based on a former state when an urgent update comes in.

If an application uses an external library to keep state updated, this external library has to be able to support this kind of concurrent state logic, so it too can keep multiple instances of the state running at the same time. React 18 introduces the `useSyncExternalStore` for that exact purpose.

7.4.4 Run effect before rendering with useInsertionEffect

If you have a library, that as a side-effect of component rendering creates stylesheets or similar HTML nodes in the document, your library now needs to be aware of Concurrent Mode in order for your library to correctly render the correct nodes at the correct time. For that specific purpose, React 18 introduces the `useInsertionEffect` hook.

While this is and looks like an effect hook in the same vein as `useEffect` and `useLayoutEffect`, the `useInsertionEffect` hook is never applicable to regular components and it's only created as a consequence of how some general-purpose libraries have to be updated to account for the consequences of concurrency.

7.5 The two key principles of hooks

There are only two rules about React hooks that you need to obey.

1. Only call hooks at the top-level of functional components
2. Only call hooks inside functional components

The first rule we already discussed: you can only use hooks directly in your components and you must always include the exact same number of hooks. That means that you can never call hooks inside a function (including inside a function used in a hook), a nested block (either a conditional or loop), or have early returns in your component before you've rendered all your hooks.

The second rule is kind of obvious, but maybe kind of *not* obvious: You can only use hooks inside functional components. You can't create some helper function or callback, that calls a hook. Nor can you use them inside class-based components.

The only exception to this rule is that you can use hooks inside other hooks, so-called custom hooks. And you can again use custom hooks inside other custom hooks, etc. But you can only use those custom hooks either in other custom hooks or in your components, so you can't actually circumvent this rule, you can just hide it a layer (or multiple layers) down.

We'll cover using custom hooks in chapter 10.

7.6 Understanding dependency arrays

We have used dependency arrays a few times already to restrict when various hooks are triggered. We use dependency arrays for effect hooks. An empty array in an effect hook indicates, it only runs on mount, whereas a non-empty array indicates, that the hook should run both on mount, as well as everytime the mentioned dependencies update. We also use dependency arrays for memoization hooks, where we for example can create stable values by using empty dependency arrays.

But how do these dependency arrays work in practice? How do you specify the right elements in a dependency array and how do you make sure that the dependencies don't update "too" often?

First let's reiterate which hooks use these dependency arrays to define when the hooks should take effect: `useEffect`, `useCallback`, `useMemo`, and `useLayoutEffect`. These four, and only these, use dependency arrays to conditionally trigger their effect and/or execution.

There are 3 general classes of dependency arrays: You either don't specify an array at all, you specify an empty array, or you specify a non-empty array.

If you don't specify a dependency array at all, it means that the hook should be triggered on every render regardless of which values update in the hook (if any at all). If you have an empty array, your hook will only ever trigger on mount and never again (except for cleanup functions, which will trigger on unmount, but that's not your hook, that's a side-effect of running your hook).

If you specify a non-empty array, your hook will trigger when any of the values in the array change on a render. Any single change will trigger the hook and React uses *referential equality* to determine whether a value has changed or not.

Referential equality

In JavaScript (and in many other languages), values come in two types: Either they're primitives or complex objects. JavaScript has 7 primitive types (number, bigint, boolean, string, symbol, null, and undefined) and 1 complex type (object). You might be thinking: what about classes, regular expressions, arrays, and functions? They are however also considered objects (though classes and functions in some sense are considered to be of type function, which is a subtype of type object).

Values can be compared to see if they're strictly equal to each other with the triple equals operator: ===. The regular double equals operator will do type conversion, so "1" == 1, but strict equality requires that types are also identical.

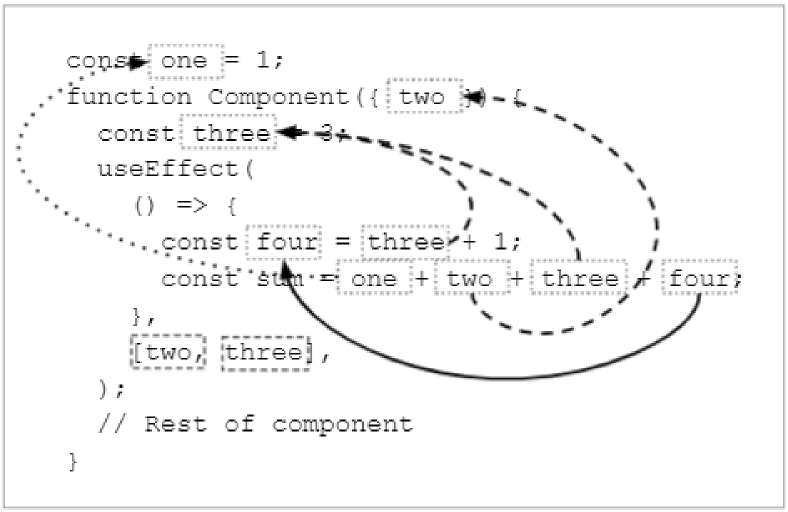
When you're comparing two different primitive values to each other, they will be considered strictly equal to each other if their values represent the same data, even if they are two different variables. You can e.g. compare 2 === 1+1, and it would be true.

For complex types however, strict equality means referential equality. For objects, they are only considered identical if they literally are the same object, where updating one would also update the other. That means that {} !== {} and [] !== [], even though we're comparing two empty objects and arrays respectively. They are not considered strictly equal, because they're not the exact same object (or array), they are just similar data structures.

When we refer to React using referential equality, we mean that React will be comparing values using the strict equality operator and will thus only consider objects or arrays unchanged, if they're references to the exact same object and not merely two different values containing similar data.

7.6.1 What are dependencies?

Dependencies of a hook are a subset of all the variables and references that you use in the hook. It is any local variable that exists locally in the component scope, but not any variable that also exists outside the component scope. Please refer to figure 7.5 for examples.



```
const one = 1;
function Component({ two }) {
  const three = 3;
  useEffect(
    () => {
      const four = three + 1;
      const sum = one + two + three + four;
    },
    [two, three],
  );
  // Rest of component
}
```

Figure 7.5 This hook uses four variables inside named one, two, three and four. Variable one comes from outside the component as noted by the dotted arrow. Variables two and three come from inside the component, but outside the effect as noted by the dashed arrows. Variable four comes from inside the effect itself. Only variables two and three and relevant to add as dependencies as can be seen in the dashed boxes at the bottom.

That means, that this includes any variable defined as a `const`, `let`, or `var` in the component, any functions defined inside the component, and any argument passed to the component (mostly properties, but potentially also forwarded references). It also means that any function or variable defined outside the component or imported from other files is not relevant as a dependency for a hook. And finally, any variable defined *inside* the hook is not a dependency, as it does not exist in the outside component.

Let's create a fictional example, with a bunch of variables used inside a hook, in this case, a `useEffect` hook. We will only specify the relevant dependencies in the array, to illustrate which go there, and which don't. Please see figure 7.6 for this visually annotated example, where you can see that variables from the component body as well as properties are added as dependencies, but global variables from outside the component and local variables inside the effect are not.

7.6.2 Run on every render by skipping the dependency array

Imagine that you want your effect to run on every render, regardless of why a component re-renders. Maybe you want to actually track all the renders for tracking or statistical purposes. You could add a dependency array listing every single property and state value that exists, and that would work if any of those values changed.

But remember that your component also re-renders, because its parent component re-renders, and that might not come with any change of any property or state value, so regardless of how many values you put in your dependency array, your effect will never run on every single possible render.

There is a simple solution: You can just skip the dependency array completely. Don't supply any dependency array at all and your effect will run on every render, regardless of why the render happens:

```
function Component({ ... }) {
  useEffect(() => track('Component rendered'));
  ...
}
```

Note how we only supply a single argument to the `useEffect` function. We simply ignore passing a second argument all together.

You might ask why then run the effect in a hook at all - why not just run the code inline like so:

```
function Component({ ... }) {
  track('Component rendered');
  ...
}
```

Executing the tracking function inside an effect hook is recommended for optimization. The track function above might be a bit slow and the responsiveness of this function call should not block your component from rendering. So by running it in an effect, you decouple the execution of the effect from the rendering of the component.

MEMOIZATION WITHOUT DEPENDENCIES IS MEANINGLESS

Would you ever skip the dependency array for a memoization hook then? If the memoization hook body runs for *every* render, then the overhead of memoizing the value doesn't actually do anything. So no, you would never do that, as it would just be useless code.

If you do this:

```
const value = useMemo(() => someCalculation());
```

You might as well just do this, as it is much more efficient and does the exact same thing anyway:

```
const value = someCalculation();
```

NO DEPENDENCY ARRAY IS DIFFERENT FROM EMPTY ARRAY

Do be aware that a missing dependency array is very different from an empty dependency array. They are actually each other's polar opposites.

A hook with an empty dependency array only runs once on the initial mounting render of the component and never again, whereas a hook without a dependency array runs on every single render of the component regardless of why it renders.

7.6.3 Skip stable variables from dependencies

If you have been extra attentive, you might now realize that we have cheated. We did not actually follow the above best practice of always specifying all the variables used in a hook in the dependency array. We skipped that one time. When? We did that in listing 6.2 previously in this chapter. You get an extra gold star for noticing! We actually did point it out at the time, so maybe you only get a silver star.

Here's the relevant section of the component in listing 6.2:

```
const [seconds, setSeconds] = useState(0);
useEffect(
  () => {
  const interval = setInterval(
    () => setSeconds(seconds => seconds + 1),    #A
    1000,
  );
  return () => clearInterval(interval);
}, []);    #B
```

#A Here we use the variable setSeconds, which is clearly defined outside the effect, but inside the component
#B But we still specify an empty dependency array? That's not allowed, is it?

So what's going on here? Are we allowed to skip listing variables as dependencies? Yes, if it is a *stable variable*. The concept of a *stable variable* is actually quite the oxymoron because it's a *variable* that doesn't *vary*. If the variable always has the same value for every render of your component, it's irrelevant to put it in the dependency array, because we know it never changes. That's partially the reason why values from outside our component don't go in the dependency array. If we define a constant outside our component or if we import a constant from another file, we know that it's always the same constant for every render of our component, so even if we depend on it, we don't need to consider it as a value that can change.

In the same way, we can have variables inside our component that we know are stable. Variables that never change. When it comes to functions and objects, that's extra important, because even if a function has the same body every time, it does not mean that it *is* the exact same value.

When it comes to hooks, React actually defines and specifically lists some return values as stable. If you compare the returned values from certain hooks, you will see that it is not just similar functions or objects, it is the *exact same* function or object that is returned. We can ignore adding these values as dependencies to make our components and hooks easier to read and understand.

This is the case for the setter function returned by `useState`. The value returned can change for every render, but the setter function is always the exact same function reference. That is the reason why we don't need to include it in dependency arrays.

This is also the case for the object returned by `useRef`. It is always the exact same object, but the value of the current property changes and is dynamic. The object however stays the same.

If you use a `useRef` reference or a `useState` setter inside an effect hook (or a memoized hook), you *can* specify it in the dependency array, but you don't *have to*. Both you and React know that both the reference and the setter are stable, so they never change, and will thus never cause the execution of the hook to change. Specifying them as dependencies is optional. Development teams often include in their coding standards whether to *always* or *never* do so in their codebase.

You can make your own stable variables as well to make your components easier to read and understand for you as a developer – and for the rest of the team. If you memoize a value in your component using a hook, and you include an empty dependency array, the returned value is stable. A memoizing hook with an empty dependency array will always return the exact same value, thus it can be considered stable.

For example, imagine this code for an incomplete component, where we *do* specify all dependencies even if they're known to be stable. The code becomes more verbose sacrificing simplicity and understanding:

```
function Panel() {
  const [isOpen, setOpen] = useState(false);
  const toggleOpen = useCallback(
    () => setOpen(open => !open),    #A
    [setOpen],    #B
  );
  useEffect(
    () => {
      // Some effect here
      toggleOpen();    #C
    },
    [toggleOpen],    #D
  );
  ...
}
```

#A We use a component-scoped variable
#B So we specify it as a dependency
#C We use another component-scoped variable
#D So we specify it too as a dependency

Here we specify `setOpen` as a dependency, even though as we just discussed, we can actually skip it because it is known to be stable. It will never change. If you are examining the code in the above component it is however not obvious, that the effect only runs on mount, because there is a dependency array. You have to track that dependency to check its origin, which might force you to track another list of dependencies, etc.

If we go ahead and skip the `setOpen` dependency from the `useCallback` hook, we will then see that `toggleOpen` now is a stable value, because it is defined in a memoizing hook with an empty dependency array. This value will (also) never change in the lifetime of the component. That means that we can then skip the `toggleOpen` value from the dependency array of the effect hook.

We can greatly simplify this component as just the following:

```
function Panel() {
  const [isOpen, setOpen] = useState(false);
  const toggleOpen = useCallback(() => setOpen(open => !open), []);    #A
  useEffect(() => toggleOpen(), []);    #A
  ...
}
```

#A We can supply an empty dependency array for both hooks, because we know both only use values known to be stable

This version is much easier to read and understand, because you instantly know that both hooks only run a single time due to their empty lists of dependencies.

7.7 Quiz

1. React has always had and will always have 15 hooks. *True* or *false*?
2. Which of these are considered *stateful* hooks:

   a) `useState`
   b) `useValue`
   c) `useId`
   d) `useReducer`

3. All components should be memoized as a standard. *True* or *false*?
4. You cannot call a hook inside a function unless it is a functional component or a custom hook. *True* or *false*?
5. Which of the following constructions are not allowed:

   a)

```
function Component({ isVisible }) {
  if (!isVisible) return false;
  useEffect(() => { ... }, []);
  ...
}
```

   b)

```
function Component({ hasEffect }) {
  if (hasEffect) {
    useEffect(() => { ... }, []);
  }
  ...
}
```

   c)

```
function Component({ shouldRender }) {
  useEffect(() => { ... }, []);
  if (!shouldRender) return false;
  ...
}
```

7.8 Summary

- React has 15 different built-in hooks, but several of them are rarely used, leaving about 10 as the core API on which all React applications are built.
- Hooks are used for a variety of purposes that make components smart and able to interact with the webpage as a whole, and even though all the hooks vary wildly in their purposes, they all have some common features.
- Stateful hooks are required to make applications stateful. You can use several different hooks depending on the complexity of your application and the values in your state. With React 18 you can even make lower-priority and high-priority state updates to help React make your UI as responsive as possible.
- Effect hooks are used to run side-effects inside components as we learned in chapter 6. By using the dependency array, you can trigger your effect to run at the desired time(s).
- Memoization hooks are used for optimization of rendering in React. Memoization can be done without hooks, but sometimes you need hooks to memoize complex values – callbacks in particular are a common target of memoization.
- Library hooks are meant for more complex codebases only and probably not relevant for your everyday applications.
- If you use a hook in a component, you must obey the two laws of hooks: only call hooks at the top level of a component (so no conditional hooks or loops of hooks) and only use hooks inside functional components (so no hooks outside a component, in a helper function, or even in a class-based component).
- Dependency arrays are tricky and a common source of confusion. By special attention to what you put in there to make sure your effects and memoization triggers correctly.

7.9 Quiz answers

1. *False*. React 16.8 introduced the first 10 hooks, and React 18.0 added another 5. More will definitely come in future releases.
2. The correct answers are `useState` and `useReducer`. `useValue` is not a built-in hook (but you could make a custom hook named this if you wanted) and `useId` is not a stateful hook, but used for a rather specific memoization purpose.
3. *False*. You should always start out not memoizing any components at all, and only if you later find that you have performance issues with your application should you start to add memoization, and even then only where it actually makes a difference. Memoizing without merit will only hurt performance.
4. *True*. You should not attempt to call hooks inside functions, that are not themselves custom hooks. Even though it might seem to work at first, it will only lead to problems down the line, if you suddenly start calling one of these functions outside of a functional component. Obey the principles of hooks!
5. The illegal constructions are *a and b*. Only version c is a valid component. Versions a and b both use conditional rendering of hooks, which are not allowed.

8 Handling events in React

This chapter covers

- **Reacting to user input using events**
- **Handling event capturing and bubbling**
- **Managing default event actions**
- **Attaching event listeners directly to the HTML**

Events are the way that users interact with a JavaScript web application. Events can be caused by mouse movement or clicking, touch interface clicks and drags, keyboard button presses, scrolling, copying and pasting, as well as indirect interactions such as focusing and unfocusing elements or the entire application.

So far we have created React applications with very little user interaction. We have handled clicking a button here and there, but not really talked in depth about how the click event works, and how we as developers handle it. We are going to change that in this chapter, which is dedicated to event handling.

You can think of events as the way to handle inputs from a user. Our web application creates JSX which is converted to HTML. The user then interacts with that HTML, and the result of those interactions are events dispatched from the HTML elements to our React application. This simple flow of information is illustrated in figure 8.1.

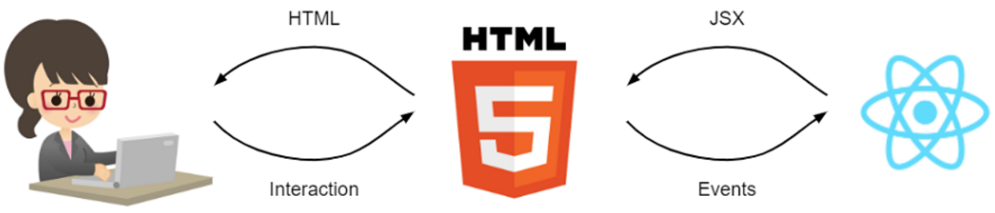


Figure 8.1 Information flow between React and the user goes through the HTML. Imagine the user visiting a login page. The user inputs the email and password, the browser forwards those interactions as events to React, the application then renders the JSX required to display a green checkmark next to each input as it is filled, and the browser renders the corresponding HTML to display to the user.

Events are also used internally in the browser to signify when things change between elements. It can be when a video is playing/pausing/buffering, an animation is completed, a DOM element is mutated, data is loaded (or failed to load), etc. There are hundreds of possible events, and any interactive web application will be utilizing a big amount of them.

There are two ways to handle events in React:

- You can use React to manage your event listener, or
- You can manually add and remove your event listener directly on a DOM object

Relying on React to handle listeners saves a bunch of tedious headaches (and potential memory leaks), but it comes with a minor loss of flexibility. Directly adding event listeners allows you to listen for all kinds of events and assign listeners to whichever nodes you feel like when you need to, but comes with the cost of having to manage listeners (and remembering to remove them again) as well as dealing with native events that might differ between browsers.

In this chapter we will show you both approaches, and discuss when best to apply one or the other. Do note that handling events *in React* is both a whole lot easier as well as recommended. Therefore, this part will be covered in a lot more detail in this chapter.

As we cover how you can listen to events using React's interface for it, we will discuss a number of topics about how React handles events and how you can work with the React API to listen to the specific events that you need. We will answer all of the following questions:

- Which events are supported?
- How do you create the event handler function?
- What are the event objects that you receive?
- How do event phases and propagation work?
- How do you handle events in the capture phase of the event dispatch?
- What are default handlers and how do you prevent them?
- When should you persist an event?
- Can you use properties as event handlers?
- Should you memoize an event handler?
- What are event handler generators?

We will then proceed to situations, where this just isn't enough, and you need to handle events manually in the DOM. And we will of course give you all the insights into how to do this best.

All of this will lead to the next chapter, where we will be utilizing our newfound understanding of event handling to create interactive form inputs and forms in general, which is a cornerstone of many web applications.

NOTE: The source code for the examples in this chapter is available at https://github.com/rq2e/rq2e/tree/main/ch08. But as you learned in chapter 2, you can instantiate all the examples directly from the command line using a single command.

8.1 Handling DOM events in React

Events are an essential way of communicating in the browser between the user and the script as well as between different elements in the application. Because of this, proper event handling is a first-class citizen in React, meaning that React has dedicated a big part of its core API to this exact purpose.

The API is very simple. If you define a property on a JSX element, that references an HTML node, and that property matches a known event from React's list of supported events, React will treat the property not as a DOM attribute, but rather as an event listener. React will then make sure to correctly add and remove the event listener, as the component mounts and unmounts.

8.1.1 Basic event handling in React

The most important event of all in almost any web application is the *click event*. And contrary to its name, it is not actually only used to accept clicks from a mouse. The click event in HTML is also invoked when a touch-screen user taps on a button or when a keyboard user activates a button using the enter key.

Let's go back to our trusted counter component and take a closer look at how we handle the click event. If you remember, this application had a button and we did something as a response to the user pressing said button. First, let's repeat the code for this simple application in listing 8.1.

Listing 8.1 Counter component

```
import { useState } from 'react';
function Counter() {
  const [counter, setCounter] = useState(0);
  const onClick = () => setCounter(c => c + 1);    #A
  return (
    <>
      <h1>Value: {counter}</h1>
      <button onClick={onClick}>Increment</button>    #B
    </>
  );
}
```

#A We create a local variable, which is a function, that when invoked, will increment the state value
#B Then we assign that local variable to the onClick property on our button.

In this example, we handle a click event on an HTML object, which is a `<button>`. Any HTML element will actually dispatch a click event if clicked, so we could have changed this element to a `<div>` or any other type of element.

Another event, that we can listen for on all objects, are mouse (or pointer) events. Any element can dispatch e.g. a `mousemove` event, when a mouse moves inside that element's boundary. We can listen for such an event in the same way.

Let's create a component that shows a checkmark if the mouse is moving around inside the element, but changes to a cross if the mouse has stopped moving for half a second or if the mouse moved outside the element.

To do that, we need to listen for the `mousemove` event. In React, that means that we assign a function as the `onMouseMove` property on our target element. In this case, we will just use a `<section>` element and display our result in a heading inside of that. See this implemented in listing 8.2 and see the result in figure 8.2.

Listing 8.2 Is the mouse moving?

```
import { useState } from 'react';
function MouseStatus() {
  const [isMoving, setMoving] = useState(false);
  const onMouseMove = () => setMoving(true);    #A
  useEffect(() => {
    if (!isMoving) return;
    const timeout = setTimeout(() => setMoving(false), 500);
    return () => clearTimeout(timeout);
  }, [isMoving]);
  return (
    <section onMouseMove={onMouseMove}>    #B
      <h2>
        The mouse is {!isMoving && 'not'} moving: {isMoving ? '✓' : '✗'}
      </h2>
    </section>
  );
}
```

#A Again we create a local variable, which is a function, that when invoked, will set the moving flag to true
#B And again we assign that local variable to the relevant property on our element, this time the onMouseMove property on our section element

rq08-mouse-status

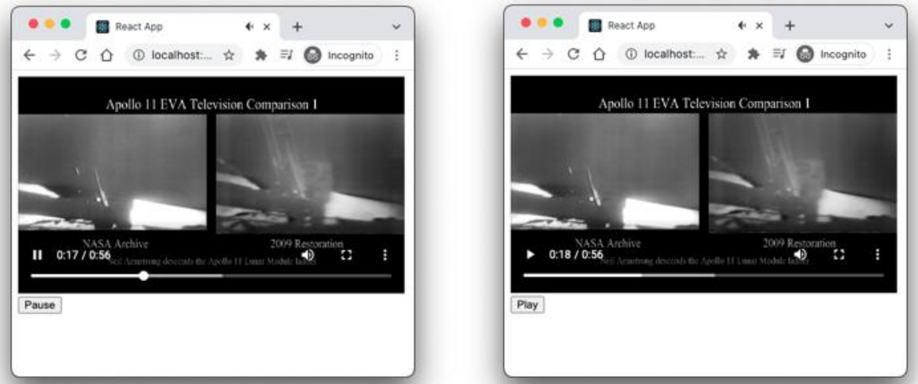


Figure 8.2 Our mouse status component when the mouse is moving or not moving respectively.

Not all events are dispatched by all types of elements though. Video (and audio) elements dispatch a play event, once the video (or audio) starts playing. Buttons don't dispatch that event, because they aren't videos (or audios) that play.

Let's create an application that displays a play/pause button next to a video. When the video is playing, the button is a pause button. When the video is paused, the button is a play button.

For this we need a total of four listeners. We need to listen to the play and pause events on the video object, and we need to listen for the click event on our button, but with two different event listeners depending on whether the video is playing or not. We will implement this in listing 8.3.

Listing 8.3 A very simple video player

```
import { useState, useRef } from 'react';
const VIDEO_SRC = '//images-assets.nasa.gov/' +
  'video/One Small Step/One Small Step~orig.mp4';
function VideoPlayer() {
  const [isPlaying, setPlaying] = useState(false);
  const onPlay = () => setPlaying(true);    #A
  const onPause = () => setPlaying(false);    #B
  const onClickPlay = () => video.current.play();    #C
  const onClickPause = () => video.current.pause();    #D
  const video = useRef();
  return (
    <section>
      <video
        ref={video}
        src={VIDEO_SRC}
        controls
        width="480"
        onPlay={onPlay}    #E
        onPause={onPause}    #E
      />
      <button
        onClick={isPlaying ? onClickPause : onClickPlay}    #F
      >
        {isPlaying ? 'Pause' : 'Play'}
      </button>
    </section>
  );
}
```

#A When the video starts playing, we toggle the state flag to true
#B Vice versa when it pauses - we set the flag to false
#C When the button is clicked to play the video, we invoke play on the reference to the video DOM element
#D And when the button is clicked while the video is already playing, we pause the video
#E We assign the two video event listeners to the video element using the appropriate properties
#F We assign one of the button click event listeners to the onClick property depending on the flag

`rq08-video-player`

If you run this application, you should see something similar to figure 8.3 in your browser.

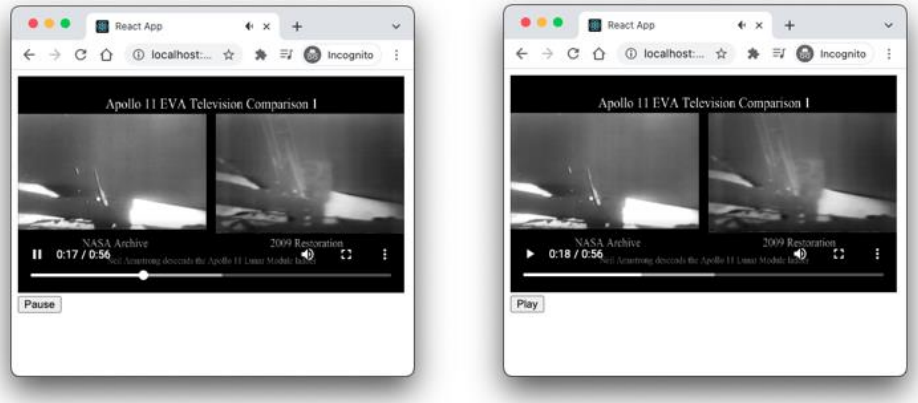


Figure 8.3 The video player interface when the video is playing and paused respectively.

Listening to events in React is really only about three things:

- Knowing what event to listen for
- Knowing which element to listen on
- Assigning a listening function to the correct property on the correct element.

That's really all there is to it.

YOU CAN USE THESE EVENTS

You can only use React to listen for events that are actually supported by React. Normally this is not something you have to worry about, because almost all DOM events are supported by React. You can see a full list of all the supported events in table 8.1.

There are some JavaScript events that are not supported in React. Mostly because these events are dispatched from objects that aren't in the DOM, but are objects that you create in JavaScript only. This includes events from objects such as socket connections, request objects.

Some other non-supported DOM events are events that are only sent to the window or document nodes. These are not supported by React, because these two nodes are never inside your React application. React only lives somewhere inside the document element, never above it.

Note that if you set a property on a JSX element that matches a known event type listed below, React will convert that property to a listener on that element, regardless of whether that element can actually dispatch that event at all. You can for instance assign an `onPlay` event listener to an `<h1 />` element, though that event will only ever be dispatched from `<video />` and `<audio />` elements.

Table 8.1. List of events directly supported in React

Clipboard events	`onCopy onCut onPaste`
Composition events	`onCompositionEnd onCompositionStart onCompositionUpdate`
Keyboard events	`onKeyDown onKeyPress onKeyUp`
Focus events	`onFocus onBlur`
Form events	`onChange onInput onInvalid onReset onSubmit`
Generic events	`onError onLoad`
Mouse events	`onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp`
Pointer events	`onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut`
Selection events	`onSelect`
Touch events	`onTouchCancel onTouchEnd onTouchMove onTouchStart`
UI events	`onScroll`
Wheel events	`onWheel`

Media events	`onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend onTimeUpdate onVolumeChange onWaiting`
Image events	`onLoad onError`
Animation events	`onAnimationStart onAnimationEnd onAnimationIteration`
Transition events	`onTransitionEnd`
Other events	`onToggle`

8.2 Event handlers

To handle an event, simply assign any function to the value of the event on an object that might dispatch such an event. Your function does not have to behave in any particular way or accept any particular argument. The event handler function will be called with a single argument, which is the event object, but you don't have to accept it.

Because there are no restrictions nor any defined best practices from an "official" side on how to define an event handler function, you will see people do it in many different ways. In this section, we will cover some of the different options and some conventions that we have seen used in larger codebases.

8.2.1 Definition of event handlers

You can define the event function any way you feel like. If it is a valid function, it is valid as an event handler. Common options include:

- Define the function as a local variable using an arrow function
- Define the function as a local variable using a function expression
- Define the function as an inline function using an arrow function directly assigned to the property

Here's an example of our counter component once again with a local variable using an arrow function:

```
function Counter() {
  const [counter, setCounter] = useState(0);
  const onClick = () => setCounter(c => c + 1);    #A
  return (
    <>
      <h1>Value: {counter}</h1>
      <button onClick={onClick}>Increment</button>    #B
    </>
  );
}
```

#A We create a variable using const and assign a function using the arrow function notation
#B We then assign that variable to the onClick property

Here's the very same component, but with the handler function defined using a function expression:

```
function Counter() {
  const [counter, setCounter] = useState(0);
  function onClick() {    #A
    setCounter(c => c + 1);
  }
  return (
    <>
      <h1>Value: {counter}</h1>
      <button onClick={onClick}>Increment</button>    #B
    </>
  );
}
```

#A Here we create the function using a function expression, which will scope the variable as a local variable
#B We again assign that variable to the onClick property

And finally, here's the same component with the handler defined inline using an arrow function:

```
function Counter() {
  const [counter, setCounter] = useState(0);  return (
    <>
      <h1>Value: {counter}</h1>
      <button onClick={() => setCounter(c => c + 1)}>    #A
        Increment
      </button>
    </>
  );
}
```

#A This time we create the event handler inline and directly assign it to the relevant property on the HTML element.

The middle approach above with a function expression inside your component is a bit unusual, albeit fully valid. We will not be using that syntax, and we haven't seen it much in the wild.

Whether you define your event handler in a variable or inline in the JSX is up to you. Many will mix and match the two options, and so will we throughout this book. Your team will most likely find a convention that works for them, and if you're working alone, find what works for you.

A common convention is to define single-line event handlers inline, but multi-line event handlers in a separate variable.

So while there is nothing stopping you from doing this:

```
return (
  <button onClick={() => {
    setCounter(count => count + 1);
    toggleState();
  }}>Button</button>
);
```

Some developers will find it a bit messy and will prefer to have such multi-line event handlers defined separately in a variable before the JSX is returned:

```
const onClick = () => {
  setCounter(count => count + 1);
  toggleState();
};
return <button onClick={onClick}>Button</button>;
```

8.2.2 Event objects

When an event handler is invoked because the event has occurred, the event handler is invoked with a single argument—the event object. This happens both in regular HTML and JavaScript as well as in React.

React event objects are a bit special, but we will get to that in the next subsection. For now we will showcase a few things that regular JAvaScript event objects and React event objects have in common.

Let's try to build our counter component with both increment and decrement buttons again, but this time we will use the same event handler function to handle the click event on both buttons. We do this to display an alternative way of structuring the code. It is not faster or better in terms of code performance, but some will prefer this style over the previous one.

To do this, we need to know what button was pressed inside the event handler. And we can do that by looking at the event object passed. It will have a property, .target, that points to the HTML node that was clicked. In order to compare this target property with the actual node, we need a reference to one of the nodes in our component. Let's implement this in listing 8.4.

Listing 8.4 Increment and decrement with a single event handler .

```
import { useState, useRef } from 'react';
function Counter({ title, body }) {
  const [counter, setCounter] = useState(0);
  const increment = useRef();    #A
  const onClick = (evt) => {
    const delta = evt.target === increment.current ? 1 : -1;    #B
    setCounter(value => value + delta);    #C
  };
  return (
    <section>
      <h1>Value: {counter}</h1>
      <button ref={increment} onClick={onClick}>Increment</button>    #D
      <button onClick={onClick}>Decrement</button>    #D
    </section>
  );
}
```

#A First we need a ref, so we can access the HTML node
#B Then in our single event handler, we compare the event target with the increment node. If it's not that button, it must be the other one
#C We then added the delta to the currently stored value
#D We assign the same event handler to both buttons but only a ref to the increment button

Is this better than having two separate event handlers? That's a subjective question. Both solutions are fine. Sometimes one seems more appropriate, other times the other one. The choice mostly comes down to personal preference. Do you feel that having a single event handler makes the code more readable or do you prefer having separate handlers? There's no difference in performance, so it's completely up to your preferred style.

Event objects always have a *target property*, that refers to the target of said event. Another property that all events have is the *type property*. The value of this property is the type of event invoked. Imagine that we assigned the same event handler to both the `onMouseEnter` and the `onFocus` property of an input field. Then our event handler would fire if the user either moved their mouse over the field or used the keyboard to tab into the field. We could tell which event occurred by looking at the `evt.type` property.

Some event objects have extra properties that are specific for the event types that dispatched them. For instance, mouse event objects always have the properties `.clientX` and `.clientY` which indicate where in the document the mouse event occurred as well as `.ctrlKey` and `.shiftKey` that indicate whether either of those buttons were pressed while the mouse event occurred. Mouse event objects have many other properties than these though.

8.2.3 React event objects

React event handlers are not the same as "true" DOM event handlers. A DOM event handler is added to a DOM object and passed a DOM event object when invoked. A React event handler is not directly added to any DOM object and will be invoked by React with a React event object when React detects that an event of the given type happened on that object.

Compare and contrast the two approaches below in figure 8.4.

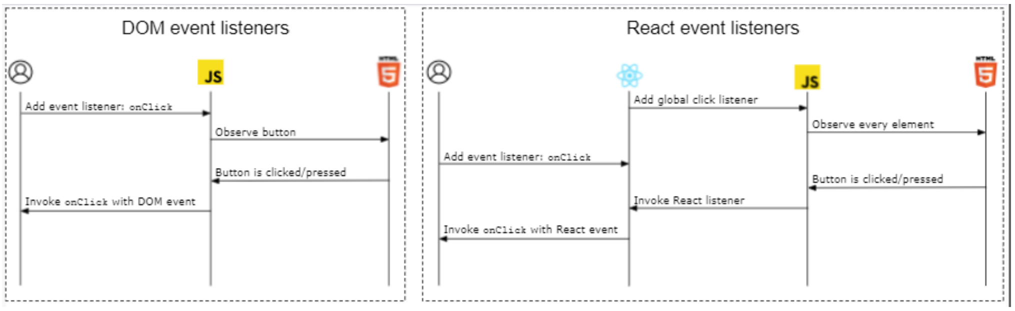

DOM event listeners
Add event listener: onClick
Observe button
Button is clicked/pressed
Invoke onClick with DOM event
React event listeners
Add global click listener
Observe every element
Add event listener: onClick
Button is clicked/pressed
Invoke React listener
Invoke onClick with React event

Figure 8.4 React doesn't actually add a listener on the individual object, but listens to any event on any object unlike native DOM listeners.

Notice in figure 8.4, that when you add a listener to an object, React doesn't add a new listener anywhere. Instead, it just remembers that you want to be informed about this specific type of event of this specific type of object. React already listens to all events on all objects, so when an event of the specified type occurs, React will check if the target matches the one you asked about, and if so, invoke your event listener with a custom React event object.

The reasoning for React implementing this new event system on top of the already existing native browser event system is two-fold: performance and consistency.

PERFORMANCE

As we just mentioned, React does not add listeners to individual elements. React adds a single listener of every event type to the document. This is done for performance reasons.

And this performance gain is real. If you add a thousand buttons and assign a click event listener to each element in pure JavaScript, it requires quite a lot of memory. But if you use React to do the same, React will only ever create a single click event listener on the document as a whole, and when invoked, check if the target matches any that you asked for. This significantly reduces memory usage.

For this reason, you don't have to worry about adding "too many" event listeners in React. If you were implementing a web app in plain JavaScript, you might have to create some workarounds to reduce how many listeners you have. When using React this is all taken care of for you, so you can just add listeners as you feel like and know that you still have great performance.

CONSISTENCY

Despite browsers being more and more standards-compliant, there are still older browsers out there, and they might do things a bit differently. This is particularly relevant when it comes to the event API. A lot of this is about browsers that are 5+ years old (older versions of Firefox and especially Internet Explorer 9 and earlier), so it does not seem extremely important today, but these browsers might still exist in the wild.

Another big reason for consistency might be slightly surprising. Some events are not standardized but still implemented by every browser. This includes events such as the mouse wheel event. There is no standard for this event, nor is any such on the way, but all browsers still support it, so React does as well. Because there is no standard, browsers handle it slightly differently when it comes to naming. The scroll wheel change in the x-direction is stored in a property called either `.deltaX` or `.wheelDeltaX` in different browsers. React's synthetic mouse wheel event takes care of this and unifies this naming as `.deltaX` always. Similar unification happens for some other non-standard properties on this and other event types.

For this reason, you don't have to worry about browser differences at all when using React events. You can rely on the React documentation only and trust that React will take care of all the underlying details for you.

Due to browser differences disappearing with older browsers becoming less and less frequent out there, it is likely that this feature of the React synthetic event system will disappear at some point in the future and be replaced with exclusively browser native events.

THE SYNTHETIC EVENT API

React's synthetic events have an API that's based on the standard API model as defined in the HTML specification. This means that you can use all the properties and methods on it that you expect from an event.

All synthetic events share a set of common properties and methods, and more specialized events have extra properties specific to the specific events. For instance, all events have a `.type` property, as well as a `.target` property. And they all have a `.preventDefault()` and a `.stopPropagation()` method. We'll get back to how these work later.

Individual event types have extra properties as needed for specific events. This e.g. includes properties like `.pageX` and `.pageY` on mouse and pointer events, which include the coordinates of the clicks on the page.

For details on the specific properties and methods, please see the React Synthetic Event API documentation: https://reactjs.org/docs/events.html.

ACCESS TO NATIVE EVENTS

If you for some reason need access to the underlying native event, maybe because you're doing something for a specific browser, that can include some extra information that is useful for your particular application, you can access it via the `.nativeEvent` property.

This property is of course non-standard and a React-only extension of the event API.

8.2.4 Synthetic event object persistence

Events need no longer be persisted. That's it, next section.

Wait, what? This might seem odd, but event persistence was a thing you had to do in earlier versions of React up until the release of React 17, after which it was no longer needed.

However, because it was a commonly used "feature" that only recently became obsolete, we will still cover it here in case you stumble upon it in the wild. This will be found in a codebase that hasn't been fully updated when moving to newer versions of React or even in tutorials and guides about React that aren't completely up to date.

Back in the day, for performance reasons, React's synthetic events were pooled in order to not create too many objects all the time. React before version 17 did not create new event objects every time an event was dispatched. React instead held an internal pool (an array, basically) of events, and when it needed to send an event, React would get one from the pool, and then immediately after dispatching the event, return the event object to the pool. As the event object returned to the pool, the event was "cleared out" meaning that all properties were reset to have no value.

What this meant for you as a developer was that if you received an event in an event handler, you had to consume the event right away. You couldn't save it or otherwise access it in a delayed manner.

Let's say that you want to create a counter, where you can increase the counter by a value selected from a dropdown. We have created a ton of counters, but this is a new variant. The goal is to display the current counter value (starting at 0) and also a dropdown with values from 1 to 5. When you select one of the values, the counter will increase by that amount. When you then select a new value, the dropdown will again increase by that new amount, etc. Let's implement this in listing 8.5.

Listing 8.5 Dropdown counter

```
import { useState } from 'react';
function DropdownCounter({ title, body }) {
  const [counter, setCounter] = useState(0);
  const onChange = (evt) =>
    setCounter(value => value + parseInt(evt.target.value));    #A
  const values = [1,2,3,4,5];
  return (
    <section>
      <h1>Counter: {counter}</h1>
      <select onChange={onChange}>    #B
        {values.map(value => (
          <option key={value} value={value}>{value}</option>
        ))}
      </select>
    </section>
  );
}
```

#A In our change event handler, we add the selected option to the current counter value by using an updater function
#B Here we assign the event handler to the select element

`rq08-persistence`

This works and all is well. However, if you were to create this in React 16.8 (when React hooks were introduced) through React 16.14 (the latest React 16 version before React 17 was introduced), this would not work. What would happen is that `evt.target.value` would throw an error in the console, because `evt.target` is undefined. The reason for this is, that we pass an updater function to the state setter, and that updater function is invoked asynchronously. By the time the function is invoked, React would already have returned the event object to the pool and reset it – including clearing `evt.target`. We could have solved this in React 16 in one of two ways:

- We could make a local copy of the value from the event object we needed right away and use that value asynchronously in our updater.
- We could persist the event, meaning that React would know not to return this particular event object to the pool, and instead just discard it as a "one-time event object" and create another brand new event object to return to the pool instead.

The former approach of copying the values we need would look like this:

```
const onChange = (evt) => {
  const delta = parseInt(evt.target);    #A
  setCounter(value => value + delta);    #B
};
```

#A First we copy the value from the event object we need to access later
#B And then we use that value

While the latter approach, where we persist the event object, would look like this:

```
const onChange = (evt) => {
  evt.persist()    #A
  setCounter(value => value + parseInt(evt.target.value));    #B
};
```

#A We instruct React to not reuse this event object, but persist it for our use indefinitely
#B And then we can freely use the event object even in asynchronous code.

This whole mess of having to remember to persist events if used asynchronously was pretty annoying. It was not something that happened very frequently and it was a very common source of confusion and errors, even for experienced developers,, which is partially why it was abandoned. The other reason was that the performance gained by pooling events grew ever smaller as devices grew faster, so it became an unnecessary optimization.

8.3 Event phases and propagation

Events are not only sent to the target object. When you click a link, the link dispatches a click event. But what if the link had a bold text inside it. Then you actually click the bold text element. Does the link then not dispatch a click event? Of course it does, because you also click the link element. You actually "click" on all the parent elements of the bold text element. This is called event propagation.

To introduce the concept of event propagation, let's consider a new example. We want to build a contact form that contains two different sections (field sets). The first section is information about the user (name and email) and the second is about why they are sending this contact request (subject and body).

Because we want the form to look nice and user friendly, we will highlight which section the user is currently inputting data in. We want the result to look like what you see in figure 8.5.

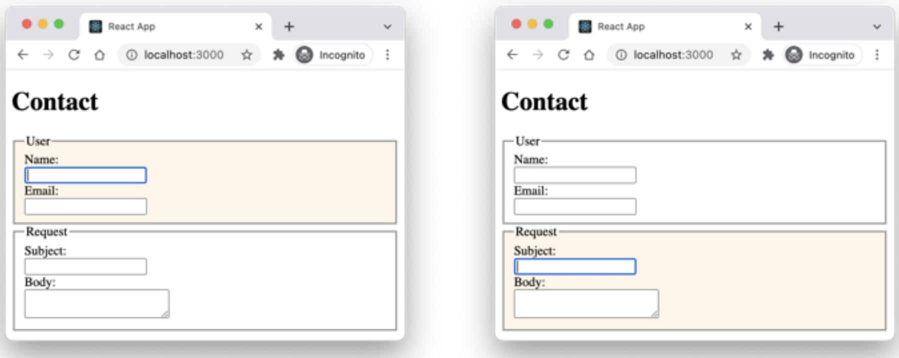


Figure 8.5 Our finished form when the user is inputting data into either section respectively.

In order to achieve this goal, we need to listen to the focus and blur events on the input fields. When an input receives focus, store its section as the focused one. When an input field loses focus, remember that no section has focus. With this approach, we need to put two event listeners on every input in both sections. In this example we only have two inputs in each section so that's a total of eight listeners, but what if we had a lot more inputs? We would have to duplicate the same two listeners to every input field. That seems like a terrible way to do this.

And it is. You should avoid repeating yourself if avoidable. In this instance it is very much avoidable because events *bubble*!

Every event in React *bubbles* up through all the nodes in the document tree above it. The only thing we need to do in order to know which section has focus is to listen for when anything inside a section receives focus. And the only thing we need to do to know that an element loses focus is to know whenever any element inside the form blurs. We can use this trick to place our focus listeners on the two sections and the blur listener on the form itself. Then we only need a total of three event listeners to achieve this result rather than eight different listeners of which most are identical anyway.

Let's look at the resulting JSX structure and where we want to put our listeners in figure 8.6.

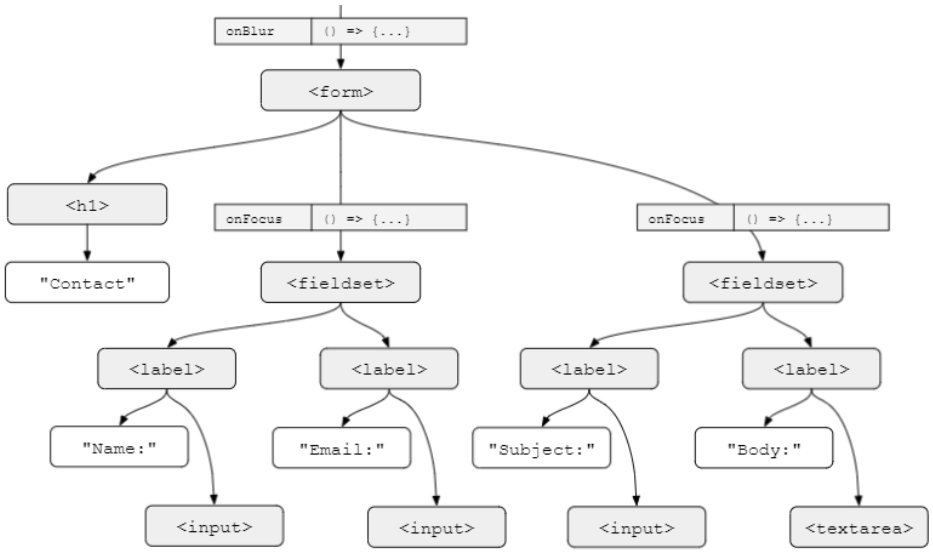


Figure 8.6 We add a blur listener to the entire form and focus listeners on both fieldsets. When an event occurs on any input (in the bottom row), the event will travel up the tree and be handled by the proper event handler.

When you focus any input field, the event will first dispatch on the input field itself, but after that, the same event will dispatch on every ancestor to the target element in order from the parent all the way up to the root node of the React application. When the bubbling reaches the fieldset, React will invoke our `onFocus` listener placed there. Similarly when an input blurs, React will invoke the `onBlur` listener placed on the form element.

Now we know what we want to achieve and what the resulting JSX is going to look like. All there's left is to put it all together into a single component. Let's do that in listing 8.6.

Listing 8.6 Dropdown counter

```
import { useState } from 'react';
const FOCUS_NONE = 0;
const FOCUS_USER = 1;
const FOCUS_REQUEST = 2;
function getStyle(isActive) {    #A
  return {
    display: 'flex',
    flexDirection: 'column',
    backgroundColor: isActive ? 'oldlace' : 'transparent',
  };
}
function Contact() {
  const [focus, setFocus] = useState(FOCUS_NONE);    #B
  const onUserFocus = () => setFocus(FOCUS_USER);    #C
  const onRequestFocus = () => setFocus(FOCUS_REQUEST);    #C
  const onBlur = () => setFocus(FOCUS_NONE);    #C
  return (
    <form onBlur={onBlur}>    #D
      <h1>Contact</h1>
      <fieldset
        onFocus={onUserFocus}    #D
        style={getStyle(focus === FOCUS_USER)}    #E
      >
        <legend>User</legend>
        <label>Name:<br/><input /></label>
        <label>Email:<br/><input type="email" /></label>
      </fieldset>
      <fieldset
        onFocus={onRequestFocus}    #D
        style={getStyle(focus === FOCUS_REQUEST)}    #E
      >
        <legend>Request</legend>
        <label>Subject:<br/><input /></label>
        <label>Body:<br/><textarea /></label>
      </fieldset>
    </form>
  );
}
```

#A First we add a helper function to generate the style for a section depending on whether it is the active section or not
#B Secondly, we need to remember what section has focus right now (at the start, none of them do)
#C Then we create three different listeners that we need to use – they are all very simple
#D We assign the listeners where we need them
#E Finally we assign the correct style to each section depending on whether it has focus or not

rq08-contact

That's it! We now have a fancy styled contact form with some pretty clever focus listeners. If you run this in a browser, you will get exactly the desired result that we saw in figure 8.5.

In the rest of this section, we will discuss in a lot more detail how event propagation works from a technical perspective. First we will discuss events in HTML and JavaScript in general, and later events in React specifically.

8.3.1 How phases and propagation work in the browser

React events bubble as just mentioned. HTML events also bubble. When you click on a button, every ancestor of that button will dispatch an event. They will actually dispatch two events – one *before* the target element itself, and again one *after* the target element itself.

> **NOTE:** **This subsection 8.3.1 is about events in HTML in general, and not React specifically. We need to cover this topic first in order for you to better understand how events in React work. In the next subsections, we will discuss event phases in React specifically, which are slightly different.**

Above, we discussed events *bubbling*, which is what happens when the ancestors dispatch an event *after* the event has already dispatched to the target element. However, all events also *capture*, which is what happens *before* the event is dispatched to the target element.

The three stages of event dispatches are:

- The *capture phase*, which happens in descending order starting with the window element going through every ancestor ending at the parent of the target element.
- The *target phase*, which is just sent to the target element itself,
- The *bubbling phase*, which happens in ascending order starting at the parent of the target element and moving up through the ancestors until the window element.

See figure 8.7 for an illustration of this.

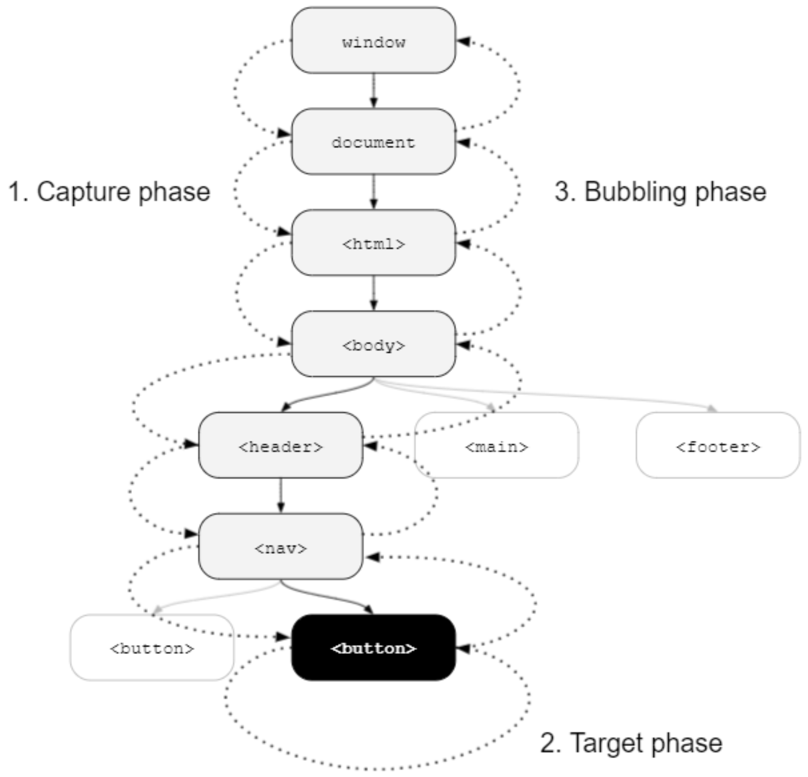


Figure 8.7 When you click the black button, the browser will start propagating events throughout the nodes in the document starting at the window, moving down through the document tree in the capture phase until the target, and then move back up the tree until the window in the bubbling phase.

This entire concept is called *event propagation*. An event propagates first in the capture phase from the window object "down" to the target element and then proceeds to propagate back "up" to the window object in the bubble phase.

When you want to listen for an event on a particular element, you can specify in which phase you are listening to the event. The default is to listen for an event in the bubbling and target phase, but you can add an argument to listen for events in the capture phase specifically.

In JavaScript, you add a (bubble and target) listener by simply calling addEventListener with the event and callback function:

```
element.addEventListener("click", onClick);
```

If you wanted to a capture listener instead, you would have to add a third argument with an object as follows:

```
element.addEventListener("click", onClick, { capture: true });
```

When you receive an event, you can check the `.eventPhase` property of the event object to see which event phase it belongs to. The possible values are:

- `Event.CAPTURING_PHASE (1)` for capture
- `Event.AT_TARGET (2)` for target
- `Event.BUBBLING_PHASE (3)` for bubble

In the example in figure 8.7, a total of 14 potential events will be sent in this exact order:

1. Capture phase:
   a) Capture event dispatched on `window`
   b) Capture event dispatched on `document`
   c) Capture event dispatched on the `<html>` element
   d) Capture event dispatched on the `<body>` element
   e) Capture event dispatched on the `<header>` element
   f) Capture event dispatched on the `<nav>` element
2. Target phase:
   a) Target event (registered as capture listener) dispatched on the `<button>` element
   b) Target event (registered as bubble listener) dispatched on the `<button>` element
3. Bubbling phase
   a) Bubble event dispatched on the `<nav>` element
   b) Bubble event dispatched on the `<header>` element
   c) Bubble event dispatched on the `<body>` element
   d) Bubble event dispatched on the `<html>` element
   e) Bubble event dispatched on `document`
   f) Bubble event dispatched on `window`

Events 2.a and 2.b might seem similar, but they will be grouped into first all dispatches to listeners defined as capture listeners, then dispatches to listeners defined as bubble listeners.

You can of course have multiple listeners listening to the same event on the same target. If that happens, events will be dispatched in order of assignment of the listeners.

Let's make a simplified view of the figure from before with only three elements in descending order in figure 8.8.

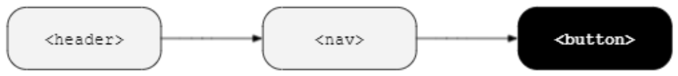


Figure 8.8 Three elements in ascending order

Let's say that we add a number of listeners to the different elements in this order:

1. Add a capture listener to `<nav>` element
   a) Add a bubble listener to `<button>` element
   b) Add a capture listener to `<button>` element
   c) Add a capture listener to `<header>` element
   d) Add a capture listener to `<nav>` element (again)
   e) Add a capture listener to `<button>` element (again)
   f) Add a bubble listener to `<nav>` element

These 8 listeners (A through G) will be invoked in this order:

1. Capture listeners on <header>: D (eventPhase=CAPTURING_PHASE)
2. Capture listeners on `<nav>`: A, E (`eventPhase=CAPTURING_PHASE`)
3. Capture listeners on `<button>`: C, F (`eventPhase=AT_TARGET`)
4. Bubble listeners on <button>: B (eventPhase=AT_TARGET)
5. Bubble listeners on `<nav>` element: G (`eventPhase: BUBBLING_PHASE`)
6. Bubble listeners on `<header>` element: *None*

Note that even though the listeners on the target element itself will be dispatched in the order of capture listeners first, then bubble listeners second, they will all be invoked with an event phase of `AT_TARGET`, rather than capturing and bubbling phases respectively.

8.3.2 Handling event phases in React

Events in React are not added by adding a listener to an object using a method. Events in React are added by assigning a property to the JSX object that represents that element.

Because of that, you cannot add an argument to say which phase you are listening to an event in. In React, as in JavaScript, the default is to add events as bubble listeners. When you write:

```
<main onClick={onClickHandler}>
  ...
</main>
```

This `onClickHandler` would be added as a bubble phase listener. If you want to add a capture event listener, you have to prefix the event with `*Capture`. For a click handler, that would be `onClickCapture`.

So if you have this bit of code:

```
<main onClickCapture={handler1} onClick={handler4}>
  <button onClickCapture={handler2} onClick={handler3} />
</main>
```

These click handlers would be invoked as `handler1`, `handler2`, `handler3`, and then `handler4`.

Capture handlers are pretty rare. It is not uncommon that you will never use them or maybe use them only once or twice in a huge application. They're a great tool to have available for when you really need them, but it is not a tool you're going to use very often. They're the julienne peelers of React–rarely used, but when needed they're going to do the job perfectly!

8.3.3 Unusual event propagation

Four event types have a very unusual event propagation flow in React.

This concerns the pairs `mouseEnter`/`mouseLeave` and `pointerEnter`/`pointerLeave`. These pairs of events are related, as the mouse or pointer will enter one element as it leaves another. The propagation of these events bubble from the element being left to the element being entered and they do not capture.

Please see the following article for details on this flow if you ever need it. It would only come up in some very specialized cases though, so this is probably not something you need to worry about: barklund.dev/mouseevents

8.3.4 Non-bubbling DOM events

In the document object model, some events don't bubble at all but still capture. This only happens for blur and focus events.

However, in React, for ease of use, both of these events still bubble as normal like other events. So let's say you have this structure in React:

```
<label onFocusCapture={handler1} onFocus={handler3}>
  <input onFocus={handler2} />
</label>
```

If you put the cursor inside the input field, the three event handlers would fire in this order: `handler1`, `handler2`, `handler3`. If you implemented the same thing without React and added the event listeners using JavaScript, `handler3` would never fire, because it is an event in the bubbling phase on an event type that doesn't bubble.

There is a technical reason for these events not bubbling in HTML, but because it's pretty confusing for developers (and very easy to forget), React simply bubbles these events as well. As a React developer you don't need to worry about this and can just use the focus and blur events as normal events. Which we actually already did in the beginning of this section.

Why focus and blur events don't bubble in HTML

When the window loses focus by the user (because the user switches tabs in the browser or even switches to a different program), a blur event will be dispatched on the window object. Similarly, when the window regains focus by a user returning to the same window/tab in the browser, a focus event will be dispatched on the window object.

If a focus or blur event on an input field or button bubbled, it would have to bubble all the way up to the window object. Then you as a developer could be confusing the event for the window losing/gaining focus by the user. You would actually be able to tell the two occurrences apart by examining the `.target` property of the event, but for historical reasons these events don't bubble simply to make sure this confusion doesn't happen.

In React, that's not a problem, because you cannot assign a React event listener to the window object. You can only assign React event listeners to actual HTML elements (and the window object is not an HTML element), and only those inside your application (which goes somewhere inside the <body> element). For this reason, focus and blur events do bubble in React.

8.4 Default actions and how to prevent them

Browsers have default actions as the consequence of some events. Most of the time, you as a developer want these default actions to occur, but sometimes you don't. In this next example, we will see an example of a default action that you don't want the browser to do, and we will see how we can prevent it from happening.

Let's say we want to create an administrator login form in React. We want to have a password field and a login button, and when the user presses the button, we want our code to check if the password matches the secret string `"platypus"`. If it does, reveal whatever secret information we have inside our application to the clearly legit administrator.

Let's start by creating this in listing 8.7.

Listing 8.7 Admin form (potentially broken?)

```
import { useState } from 'react';
function Admin() {
  const [password, setPassword] = useState('');    #A
  const [isAdmin, setAdmin] = useState(false);    #B
  const onClick = () => {
    if (password === 'platypus') {    #C
      setAdmin(true);
    }
  };
  return isAdmin ? (    #D
    <h1>Bacon is delicious!</h1>
  ) : (
    <form>
      <input type="password" onChange={evt => setPassword(evt.target.value)} />    #E
      <button onClick={onClick}>Login</button>    #F
    </form>
  );
}
```

#A We store the entered password in a state value
#B We store whether the user is approved as an admin user in another state value
#C When the user clicks the button, we check if the entered password matches the expectation and if so, update the state
#D We display different JSX depending on whether the user is approved as an admin user or not
#E Our input field will update the state password when changed
#F Our button will call the event handler when clicked

If you spin this up in a browser, enter something into the input field, and press the button, something quite unexpected happens. The whole page reloads and the input field is cleared. That's not at all what we wanted here, that just seems like a completely arbitrary result. Why did this happen?

8.4.1 The default event action

If you create an HTML form with a button on a webpage and press the button, the page will reload. This is the default behavior in HTML.

Let's say you put this HTML (note that we're talking about plain HTML at this point, not JSX) into a file and open it in a browser:

```
<form>
  <button>Click me</button>
</form>
```

Pressing this button reloads the page. That's because a button inside a form causes the form to submit, and when a form submits, the variables inside the form will be sent to the target URL of the form. This even happens if the form doesn't have any inputs and even if the form doesn't have an explicit target URL (the default target URL is the page itself).

Knowing this information, we now see what we did wrong before. Our button inside our application would submit the form, and submitting a form causes the page to reload by default.

8.4.2 Preventing default

With our newfound knowledge, we will do two things in our form to make it work correctly.

First, we will move the event handler from clicking the button to submitting the form. It's the exact same handler, we just assign it to the `onSubmit` property of the form rather than the `onClick` property of the button.

Secondly, we need to tell the form not to do the default action that it normally does when submitting. We do that by invoking `event.preventDefault()` on the event object passed to the event handler.

Let's implement this in listing 8.8.

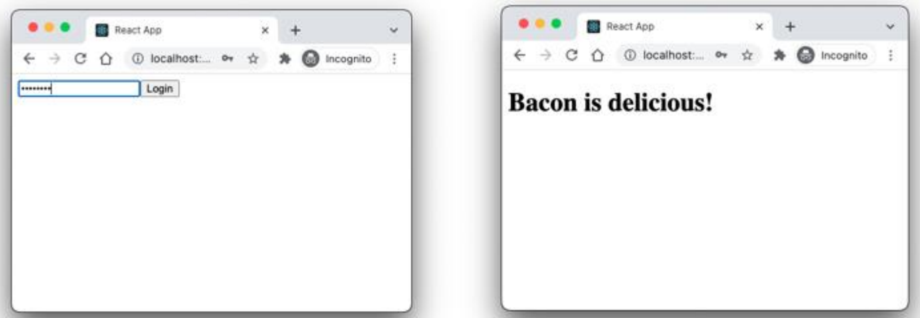
Listing 8.8 Admin form (potentially fixed?)

```
import { useState } from 'react';
function Admin() {
  const [password, setPassword] = useState('');
  const [isAdmin, setAdmin] = useState(false);
  const onSubmit = (evt) => {    #A
    evt.preventDefault();    #B
    if (password === 'platypus') {
      setAdmin(true);
    }
  };
  return isAdmin ? (
    <h1>Bacon is delicious!</h1>
  ) : (
    <form onSubmit={onSubmit}>    #C
      <input type="password" onChange={evt => setPassword(evt.target.value)} />
      <button>Login</button>
    </form>
  );
}
```

#A We store the entered password in a state value
#B We store whether the user is approved as an admin user in another state value
#C We then just have to connect the event handler to the form element

`rq08-admin`

And there we go. Our admin form works as intended! We prevented the default event from happening in our form, so the browser native event handler did not kick in. See the result in figure 8.9.



Bacon is delicious!

Figure 8.9 The admin login form works and when we enter the correct password, the secrets of the universe are revealed to us (and apparently those secrets are delicious).

Bonus info: If you press enter while focusing an input field inside a form which also has a submit button, the form will be submitted. If we just put our event handler on the button as `onClick`, submitting the form by clicking enter with focus in the input field would not work as intended, and it would still reload the page because of form submission. By moving our handler to the form's submit event we handle both ways of submitting a form.

> **NOTE:** This example is of course not in any way proper web security. Anything that happens in React is readable by any visitor on your web page and the above security would be compromised in seconds. Please use proper web architecture for creating secure logins.

8.4.3 Other default events

Form submit events have a default action, where the form actually submits to the target URL with all the values entered into the form. This is one of the default events used in the browser, but definitely not the only one.

Clicking a link will create a click event on the link element. The default action for this event is to actually follow the link and go to the new URL as indicated by the `href` property. Again here you can prevent this default behavior by invoking the `.preventDefault()` method on the click event object received in an event handler. This would mean that the browser would not go to the target URL and effectively nothing would happen.

You can check if an event is cancellable by checking the `.cancellable` property. If true, `.preventDefault()` can be invoked to stop whatever the browser's default action would have been. If false, invoking `.preventDefault()` is still possible, but it just doesn't do anything.

Here's a non-exhaustive list of cancellable events:

- Scroll events can be canceled, which would cause the scroll not to occur and the scroll offset to remain unchanged
- Key down, key press, and key up events are cancellable and will all cause the character to not be inserted (if invoked on an input field or text area) or cause whatever the browser would do in case of the given key not to happen (e.g. make the browser not scroll the page in case of canceling the press of "page up"). Input events are on the other hand not cancellable as they are dispatched after the fact (e.g. after the user typed something or pasted something).
- Drag start and drag enter events are cancellable (respectively causing the drag not happen at all or causing the drag effect to remain unchanged), however the drag end and drag leave events are not cancellable.

React follows the exact same procedures for default actions and preventable actions as HTML, so please see any online HTML guide on which events are cancellable and what the default action is.

8.5 React event objects in summary

We've seen a number of different ways to use the event object that React sends to an event handler. Table 8.2 lists a subset of properties that all event objects have in common. A lot of these properties have already been explained in detail in this chapter. These are not all the properties available on all event objects, but in our opinion the most important ones.

Table 8.2. Important properties common for all event objects in HTML

Property	Purpose
`bubbles`	A boolean value indicating whether or not the event bubbles
`cancelable`	A boolean value indicating whether or not the event can be canceled
`eventPhase`	A numerical value indicating which phase in the event propagation this event belongs to
`preventDefault`	A method to prevent the browser from handling the event with its default action
`stopPropagation`	A method to prevent the event from propagated any further
`target`	The target node that this event was assigned to
`timestamp`	The time at which the event was created in milliseconds
`type`	The type of event that caused this event object to be dispatched

8.6 Event handler functions from properties

When you are creating reusable UI elements, a vital part is to create generalized interface elements that you can then use in other locations without having to style them every time.

So for this purpose, let's now create a styled generalized button component, that can be reused over and over. We will use this generalized button component to create a counter with an increment and a decrement button - but styled.

We want to create something that looks like figure 8.10 – look at those stylish buttons.

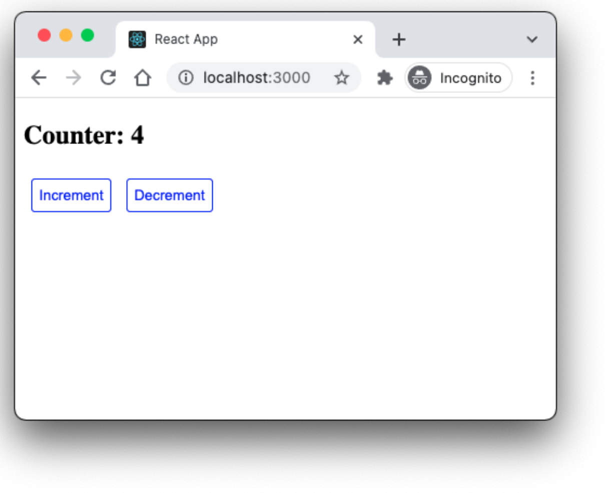


Figure 8.10 The final application with the counter already increased a few times. Don't these buttons look just a bit nicer than the default ones we're used to seeing?

We are going to structure the application as you can see in the JSX diagram in figure 8.11.

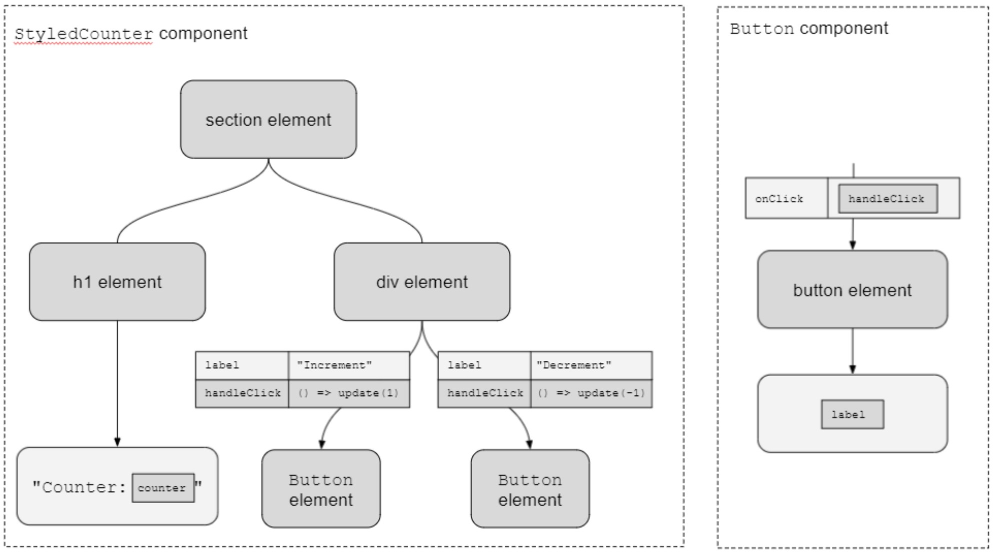


Figure 8.11 Our styled counter application will include two instances of our Button component with slightly different properties.

As you can see, we are going to pass a function, `handleClick`, to each of the button component instances, that should internally be assigned to the button as the click handler. Let's implement this in listing 8.9.

Listing 8.9 Styled counter

```
import { useState } from 'react';
function Button({ handleClick, label }) {
  const buttonStyle = {
    color: 'blue',
    border: '1px solid',
    background: 'transparent',
    borderRadius: '.25em',
    padding: '.5em',
    margin: '.5em',
  };
  return (
    <button style={buttonStyle} onClick={handleClick}>    #A
      {label}
    </button>
  );
}
function StyledCounter() {
  const [counter, setCounter] = useState(0);
  const update = (d) => setCounter(v => v + d);
  return (
    <section>
      <h1>Counter: {counter}</h1>
      <div>
        <Button handleClick={() => update(1) } label="Increment" />    #B
        <Button handleClick={() => update(-1)} label="Decrement" />    #B
      </div>
    </section>
  );
}
```

#A Inside the button component we directly assign the received handleClick property as the onClick event handler
#B When we use our buttons, we set the handleClick property to a function updating the state.

`rq08-styled-counter`

This looks pretty good and is a nice, compact, well-defined application.

However, there is a slightly weird thing. When we use our button components, we assign a function that should be invoked when the button is clicked. However, in the above, because we assign the function directly as the `onClick` property to the button, the function will be invoked with an event object as the first and only argument. Sometimes you might find this to be a good solution, but other times you might think that this is not ideal. The outside component should not have access to this event object. It is an internal implementation detail of the button component.

If you want to remove this event from the function invocation, we would have to create another function as the event handler, and when invoked, call the `handleClick` property (without any arguments).

That would look something like this:

```
function Button({ handleClick, label }) {    #A
  const buttonStyle = {...};
  const onClick = () => handleClick();    #B
  return (
    <button style={buttonStyle} onClick={onClick}>    #C
      {label}
    </button>
  );
}
```

#A We still receive a handleClick property as before
#B We now create a local function, that when called will invoke the passed property
#C And we assign this local function as the event handler, not the passed property

Note that we have named the event listener as a property `handle*`. That is a fairly common practice, when passed functions as properties to elements, that aren't directly event listeners themselves, but rather just callbacks that will be invoked by event listeners or effects as needed. We could also have named this property `onClick`, but that would make it seem like an event listener, and users would expect it to act as an event listener (and we would definitely have to send the event object to it as an argument).

You will see many examples of function properties invoked as callbacks (either directly as event listeners or inside event listeners) in real-life codebases. It is a very common way to design reusable UI component libraries. We will also be using this structure in many future chapters. We will be using `on*` naming for direct event handlers (that receive an event object) and `handle*` naming for callbacks (the either don't take any arguments or take some custom argument).

8.7 Memoization of event handler functions

Event handlers attached *directly* to JSX elements representing HTML elements need never be memoized. There is never any benefit to doing this.

Let's take an example and repeat our simplest version of the counter component:

```
function Counter() {
  const [counter, setCounter] = useState(0);
  const onClick = () => setCounter(c => c + 1);    #A
  return (
    <>
      <h1>Value: {counter}</h1>
      <button onClick={onClick}>Increment</button>    #B
    </>
  );
}
```

#A We create a local function, that is not memoized to set as the event handler function
#B We assign this event handler directly to an HTML element in JSX (the element is "button")

In the above, you might want to wrap the `onClick` definition in a memoizing hook such as `useCallback`:

```
const onClick = useCallback(
  () => setCounter(c => c + 1),
  [],
);
```

We *can* do this, but we should not. There is nothing gained from doing this. React does not automatically memoize HTML elements generated by JSX even if all the properties are identical. And besides, React does not actually add the event listener as a callback directly on the element, but merely remembers internally that you are listening for this particular event on this particular object (remember that React listens to all events on all objects always).

The only possible reason there could be for memoizing event listeners, would be similar to the arguments for memoizing values in general: If the calculation arriving at the value is complex, you can save CPU time or memory by memoizing the result. If you e.g. have a function that *generates* your event handler through some complex logic, you can avoid doing this calculation multiple times by memoizing the output. This is however a very unusual situation. Normally complex logic would occur *inside* your event handler, not in the *definition* of your event handler.

8.7.1 Memoizing event handler properties

There is a related scenario, where memoizing event handlers (or callbacks) are very common and definitely a benefit to your application: Memoizing event handlers passed as properties to other custom components.

Let's go back to our styled buttons from earlier. This time, let's memoize our button component, so it renders much quicker if nothing changes. We will implement this in listing 8.10.

Listing 8.10 Styled memoized counter

```
import { memo, useState } from 'react';
const Button = memo(function Button({ handleClick, label }) {    #A
  const buttonStyle = {
    color: 'blue',
    border: '1px solid',
    background: 'transparent',
    borderRadius: '.25em',
    padding: '.5em',
    margin: '.5em',
  };
  return (
    <button style={buttonStyle} onClick={handleClick}>    #B
      {label}
    </button>
  );
});
function StyledCounter() {
  const [counter, setCounter] = useState(0);
  const update = (d) => setCounter(v => v + d);
  return (
    <section>
      <h1>Counter: {counter}</h1>
      <div>
        <Button handleClick={() => update(1) } label="Increment" />    #C
        <Button handleClick={() => update(-1)} label="Decrement" />    #C
      </div>
    </section>
  );
}
```

#A The only change since listing 8.9 is that we added memoization to the entire Button component
#B We still assign our event listener directly from the property passed to the component
#C And we still define the event listener property directly on our components

In this instance, our memoization actually hurts more than it helps. That is because we define our event handlers without memoization (in lines #C in listing 8.10), so they are redefined every time the `StyledCounter` component renders. And that means that the `Button` component will receive new properties on every render. The `handleClick` property will be unique every time, even though it performs the same action every time.

In order to actually get a performance benefit from our memoized button, we need to memoize all the complex properties passed to our memoized component, so we need to create our two `handleClick` properties as proper variables outside the returned JSX and we need to define them using `useCallback`.

However, remember that both of these callbacks depend on our utility function `update()`, so we also need to make this value stable. We have to define this function using `useCallback` as well, and this will only depend on `setCounter`, which is a setter function returned by `useState` which we know to be stable. Let's implement all this in listing 8.11.

Listing 8.11 Styled memoized counter

```
import { memo, useState, useCallback } from 'react';
const Button = memo(function Button({ handleClick, label }) {
  const buttonStyle = {
    color: 'blue',
    border: '1px solid',
    background: 'transparent',
    borderRadius: '.25em',
    padding: '.5em',
    margin: '.5em',
  };
  return (
    <button style={buttonStyle} onClick={handleClick}>
      {label}
    </button>
  );
});
function StyledCounter() {
  const [counter, setCounter] = useState(0);
  const update = useCallback((d) => setCounter(v => v + d), [setCounter]);    #A
  const handleIncrement = useCallback(() => update(1), [update]);    #B
  const handleDecrement = useCallback(() => update(-1), [update]);    #B
  return (
    <section>
      <h1>Counter: {counter}</h1>
      <div>
        <Button handleClick={handleIncrement} label="Increment" />    #C
        <Button handleClick={handleDecrement} label="Decrement" />    #C
      </div>
    </section>
  );
}
```

#A We want to memoize any variable, that our callback depend upon
#B We then need to memoize our callbacks
#C And finally we need to pass these memoized callbacks to our memoized components

rq08-styled-memo-counter

Now in this latest iteration in listing 8.11, the memoization actually works. Our buttons are rendered only once each. As we press them and the values update, the button components *never* rerender because all the properties are identical and the function definitions are memoized.

8.8 Event handler generators

If you have many event handler functions that only vary slightly, you might want to generalize them into an *event handler generator*.

Let's take our earlier example of a counter with both an increment and a decrement button. We generalized these two different functions into a single function, that updates the value based on an argument, and then in the click event handler on the two buttons we call *that* function with different arguments:

```
function Counter() {
  const [counter, setCounter] = useState(0);
  const update = (delta) => setCounter(c => c + delta);    #A
  return (
    <>
      <h1>Value: {counter}</h1>
      <button onClick={() => update(1)}>Increment</button>    #B
      <button onClick={() => update(-1)}>Decrement</button>    #B
    </>
  );
}
```

#A A generic function for updating the counter value with a delta
#B In the event handlers, we invoke update with two different values

We can actually take this concept one step further. Note that in both event handlers, we're still defining a function that then calls update (both have an arrow definition like `() => update`). We can actually move that function definition inside the update function with a curried function.

This turns the `update` function into an event handler *generator*, which when invoked returns an event handler. So it's a function, that returns another function:

```
function Counter() {
  const [counter, setCounter] = useState(0);
  const update = (delta) => () => setCounter(c => c + delta);    #A
  return (
    <>
      <h1>Value: {counter}</h1>
      <button onClick={update(1)}>Increment</button>    #B
      <button onClick={update(-1)}>Decrement</button>    #B
    </>
  );
}
```

#A A generic event handler generator for updating the counter value with a delta
#B In the event handlers, we invoke generate an event handler with a specific delta

This might look a bit esoteric, and it's not essential that you fully understand the logic here. Just note that this is a fairly common approach used by many developers, so you might see it in your everyday work. We will revisit this approach of using event handler generators in the next chapter on event handling in forms, so you will get some more experience with the concept there.

8.9 Listening to DOM events manually

Sometimes you want to be able to listen for events on elements not directly controlled by React. Other times you want to manually control whether to listen for events at all. For both of these purposes, you need to listen for events directly on the DOM elements in regular JavaScript circumventing React's event listener setup.

Here are some example situations where you might want to manually manage event listeners:

- You want to listen for events on the window or document object.
- You want to listen for events on HTML elements not directly included inside the React application. This could be on e.g. body, which can never be inside your React application, but could also just be some element outside of the control of the React application.
- You want to listen for events on non-DOM objects, such as a request, socket or any other JavaScript object
- You want to listen for a single event on a particular object, but do not care about more than one instance of the event occurring.
- You want to conditionally listen for an event on an object.

The first three examples above are only possible by listening directly on the objects, but the two latter are still possible using React. However both would require extra work that might not be necessary.

In the following subsection, we're going to see how you achieve each of the items above by manually listening to DOM events by going outside of the React architecture.

8.9.1 Listening for window and document events

Let's say that you want to display the size of the browser window in your application. We can display the size of the browser window when the component renders the first time by just looking at `window.innerWidth` and `window.innerHeight`. But if the user resizes the window while our component is mounted, it will not automatically rerender, and we won't update our displayed value.

In order to make sure our component updates when the window resizes, we need to listen for the resize event on the window object. Since this is an event not managed by React directly, we need to attach our listener directly on the window object using `window.addEventListener`. But we also need to make sure to remove our event listener again if our component unmounts by calling `window.removeEventListener`.

If you remember back to the chapter 6 on component lifecycles, this seems like a perfect candidate for a `useEffect` hook. And you're right, it is! We will combine this with a `useState` hook to achieve a component that works something like the flow chart outlined in figure 8.12.

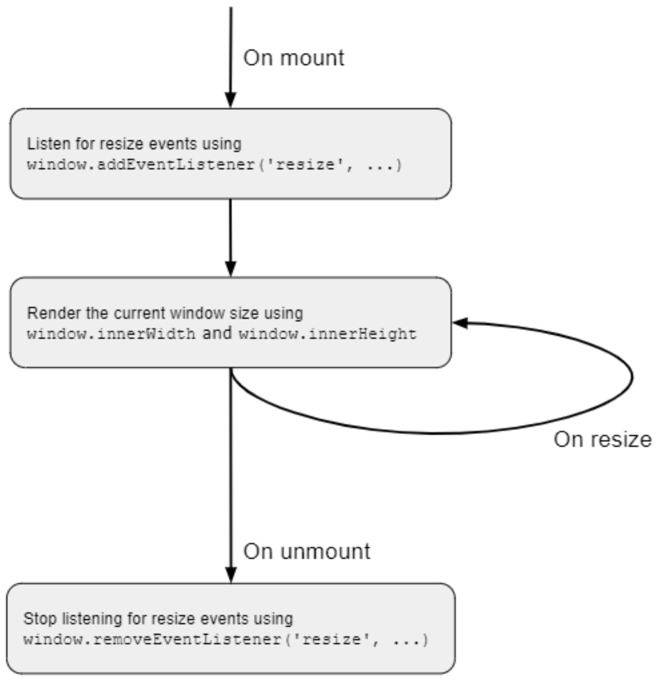

On mount
Listen for resize events using
window.addEventListener('resize', ...)
Render the current window size using
window.innerWidth and window.innerHeight
On resize
On unmount
Stop listening for resize events using
window.removeEventListener('resize', ...)

Figure 8.12 The WindowSize component must add a listener on mount and remove it again as the component unmounts. While mounted, the browser will invoke our callback if the browser window ever resizes.

Let's go ahead and implement this in listing 8.12.

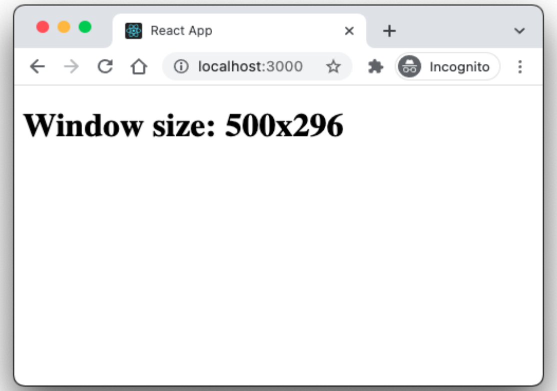
Listing 8.12 Window size display

```
import { useState, useEffect } from 'react';
function getWindowSize() {    #A
  return `${window.innerWidth}x${window.innerHeight}`;
}
function WindowSize() {
  const [size, setSize] = useState(getWindowSize());    #B
  useEffect(() => {    #C
    const onResize = () => setSize(getWindowSize());    #D
    window.addEventListener('resize', onResize);    #E
    return () => window.removeEventListener('resize', onResize);    #F
  }, [setSize]);    #G
  return <h1>Window size: {size}</h1>;    #H
}
```

#A First a little utility function to get a nice display value for the size of the browser window
#B We use that utility function to initialize our state value
#C We then setup an effect hook
#D Inside this hook, we define a function to be called when the window resizes
#E We assign this function as an event listener directly on the window object
#F We furthermore make sure that our effect hook returns a cleanup function, that removes the listener again
#G Because it's an effect hook, we need to set up our dependencies. They only contain the setSize function, which we know to be stable, but we can include it anyway for transparency
#H Finally we just need to render the actual window size in the returned JSX

`rq08-window-size`

If you run this app in a browser, you will see something like figure 8.13.



Window size: 500x296

Figure 8.13 The window size app in action with a small window

This is a very basic example on how to listen for events on permanent objects such as window or document. This is a pretty common approach for a number of events that only occur on those two objects or because you want to catch all events of some type that bubble all the way up here.

Note the clever use of a cleanup function in our useEffect. Because we define our listener function inside the effect, add the listener in the effect and in case we need to clean up, remove the listener again, this structure works regardless of our dependencies and regardless of whether our function mounts and remounts several times.

However, also note that because we're not using React's clever trick of only listening to events once and just manually remembering who listens for what, we are adding a listener to the window object for every instance of our component. If this was an element in a long list of similar elements, we would be adding a new listener for every element that we added. That definitely seems pointless, if we could just be adding a single listener.

When adding events directly on JavaScript objects, you might have to pay extra attention to how to optimize these.

8.9.2 Dealing with unsupported HTML events

Now let's look at how to listen for DOM events that are not supported by React. While we mentioned that most DOM events are supported by React, there are a few that aren't. One such example is the transition events. These events are dispatched by CSS actually, when a CSS transition is assigned, started, ended, and canceled. Of these four events, only the ended event is supported directly in React using the onTransitionEnd property.

Let's create a component with an element with a transition. We want to display a text in a transition from red to blue and back again. We will trigger this transition with two different buttons that set the color directly on the HTML node using the node's style list object. We then want to display in the headline whether the transition is running or not.

While we can listen for the transitionend event in React using the onTransitionEnd property, we cannot listen to the transitionstart event in the same way. So for ease of use, we will listen for both events using a regular DOM listener.

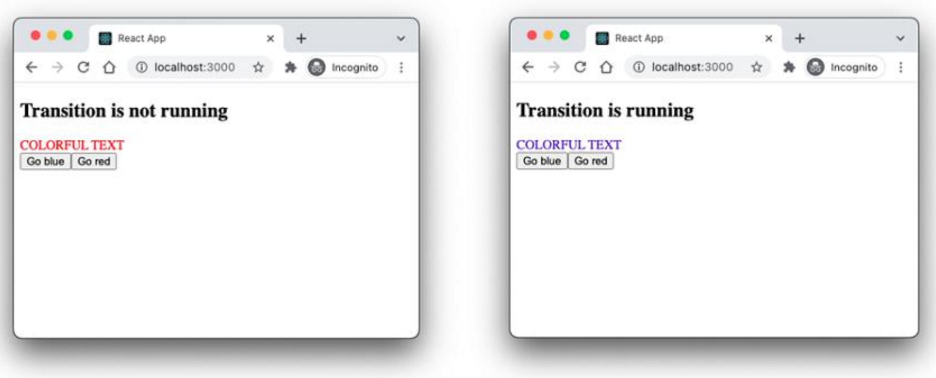
Let's combine all this in listing 8.13 and you can see the result in figure 8.14.

Listing 8.13 Transition events

```
import { useState, useRef, useEffect } from 'react';
function Transition() {
  const [isRunning, setRunning] = useState(false);
  const div = useRef();    #A
  useEffect(() => {
    const onStart = () => setRunning(true);    #B
    const onEnd = () => setRunning(false);    #B
    const node = div.current;    #C
    node.addEventListener('transitionstart', onStart);    #D
    node.addEventListener('transitionend', onEnd);    #D
    return () => {
      node.removeEventListener('transitionstart', onStart);    #E
      node.removeEventListener('transitionend', onEnd);    #E
    }
  }, [setRunning]);
  return (
    <section>
      <h1>Transition is {!isRunning && 'not'} running</h1>
      <div
        style={{ color: 'red', transition: 'color 1s linear'}}
        ref={div}    #F
      >COLORFUL TEXT</div>
      <button onClick={() => div.current.style.color = 'blue'}>Go blue</button>
      <button onClick={() => div.current.style.color = 'red'}>Go red</button>
    </section>
  );
}
```

#A Because we need to reference an HTML element, we need to use the useRef hook
#B Inside an effect hook we create two callbacks that we want to use as listeners
#C We also need a local variable that points to the DOM element - we need this to be able to access it in the cleanup function
#D We add the listeners in the effect hook directly on the DOM element
#E On cleanup we remove the same listeners from the same object
#F The only last thing we need to remember is to set the ref property on our target element

rq08-transition



Transition is not running

COLORFUL TEXT

Transition is running

COLORFUL TEXT

Figure 8.14 If you click the two buttons, you will see the text change color from red to blue to red, and the headline will reflect whether the animation is running or not. As a bonus, notice that if you press the red button while the text is already red, the transition never starts, so the headline never changes.

This is an example of one of those rare cases, where you need to listen for one of the few events not directly supported in React. And that's because it is a pretty rare event to use in an application, but that of course doesn't mean that it doesn't have its use cases.

A more likely scenario for using direct DOM listeners on HTML elements in your application are more complex scenarios, where listeners can change based on other criteria, and it just makes more sense to manage the event listeners manually rather than relying on JSX and React to add and remove listeners for us. We have an example of that coming up in the very next section and again in section 8.3.

8.9.3 Combining React and DOM event handling

In this example, we will use a combination of React's event listeners with manual DOM event listeners.

Let's create a menu that pops up when we click a button, and then closes again when we click outside the menu. We will create this application in two iterations. First we will implement it in a slightly naive way but as we find a bug, we will fix that bug and implement the component correctly.

For now, let's consider the flow of events. We need to listen for clicks on the button that opens the menu. We know how to do that using `onClick` in React. But then we also need to listen for clicks anywhere when the menu is opened. To do that, we need to listen for any click on the window object and we need to assign this handler in an effect hook.

Let's illustrate this flow of events in a diagram in figure 8.15.

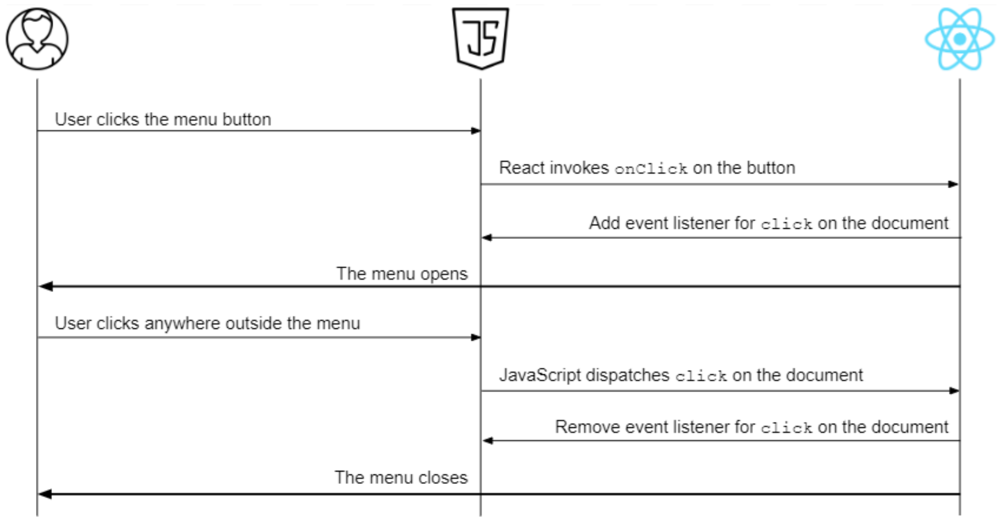

User clicks the menu button
React invokes onClick on the button
Add event listener for click on the document
The menu opens
User clicks anywhere outside the menu
JavaScript dispatches click on the document
Remove event listener for click on the document
The menu closes

Figure 8.15 The flowchart that governs our menu component

Let's go ahead and implement this in listing 8.14.

Listing 8.14 An expandable menu (naive version)

```
import { useState, useEffect } from 'react';
function Menu() {
  const [isExpanded, setExpanded] = useState(false);    #A
  useEffect(() => {
    if (!isExpanded) {    #B
      return false;
    }
    const onWindowClick = () => setExpanded(false);    #C
    window.addEventListener('click', onWindowClick);    #D
    return () => window.removeEventListener('click', onWindowClick);    #E
  }, [isExpanded]);    #F
  return (
    <main>
      <button onClick={() => setExpanded(true)}>Show menu</button>    #G
      {isExpanded && (    #H
        <aside style={{border: '1px solid black', padding: '1em'}}>
          This is the menu
        </aside>
      )}

    </main>
  );
}
```

#A We store whether the menu is expanded or not in a state value (default false)
#B Inside our effect hook, we abort early if the menu is not expanded (nothing to do in this case)
#C If the menu is expanded, we create a listener that will collapse the menu again to be invoked when anything is clicked inside the window
#D We then add the listener to the document object
#E And we remove the listener again on cleanup
#F Because we have isExpanded in the dependency array, this hook will rerun every time the menu changes state from expanded to collapsed and vice versa
#G Our menu button will simply toggle the expanded flag to true
#H And if the expanded flag is true, we render our menu

`rq08-naive-menu`

We can run this application in the browser and see what you see in figure 8.16.

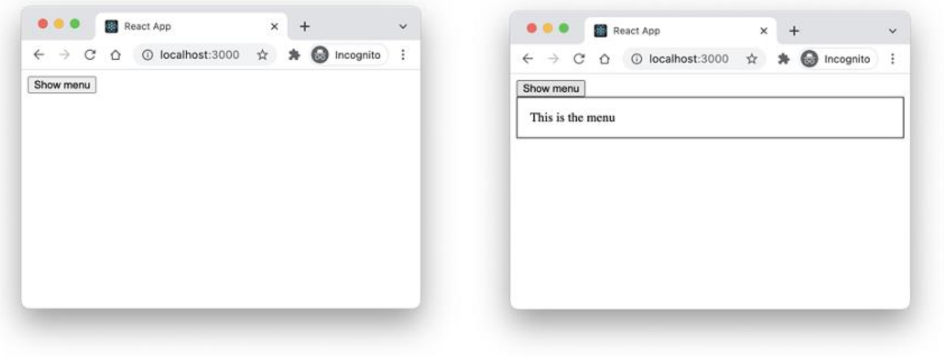


Figure 8.16 The menu app when collapsed and expanded respectively.

However, there is a slight problem. If you try this out, and click outside the menu when it is expanded, the menu does correctly close. However, if you click inside the menu, it also closes. That's not good. We probably want the user to be able to interact with our menu—we will probably have some buttons or links in there at a later point.

What we want is to close the menu, only when the user clicks outside the menu, not when they click inside it. The tricky part is to do something, when clicking "anywhere" except in a specific location.

To do this, we will use three techniques that we have learned so far:

- When we expand the menu, we will add a listener on the window object for any click that happens on the window. When invoked, we will collapse the menu just like before.
- This time, we will also add an event listener on the menu itself, that will block click events inside of it from bubbling to the window object. We do this by stopping the propagation of those events.
- Because we're going to need a reference to our menu HTML element, we need to use a `useRef` hook.

By combining these three things, we will ensure that any click inside the window (even on elements completely outside the control of React) will cause our menu to collapse, but any clicks inside the menu will not cause our menu to collapse, because we make sure that these events do not bubble to the window object. We have captured this flow of events in figure 8.17.

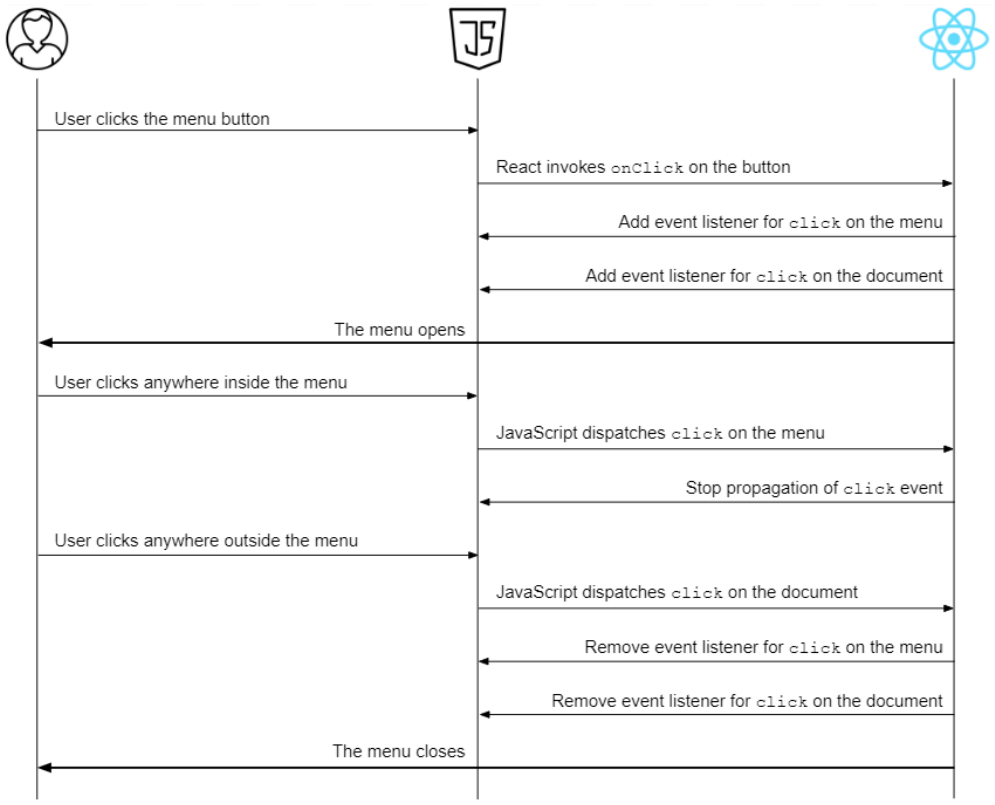

User clicks the menu button
React invokes onClick on the button
Add event listener for click on the menu
Add event listener for click on the document
The menu opens
User clicks anywhere inside the menu
JavaScript dispatches click on the menu
Stop propagation of click event
User clicks anywhere outside the menu
JavaScript dispatches click on the document
Remove event listener for click on the menu
Remove event listener for click on the document
The menu closes

Figure 8.17 The flowchart that governs our menu component

Let's go ahead and implement this in listing 8.15.

Listing 8.15 An expandable menu

```
import { useRef, useState, useEffect } from 'react';
function Menu() {
  const [isExpanded, setExpanded] = useState(false);
  useEffect(() => {
    if (!isExpanded) {
      return false;
    }
    const onWindowClick = () => setExpanded(false);
    const onMenuClick = evt => evt.stopPropagation();    #A
    const menu = menuRef.current;    #B
    window.addEventListener('click', onWindowClick);
    menu.addEventListener('click', onMenuClick);    #C
    return () => {
      window.removeEventListener('click', onWindowClick);    #D
      menu.removeEventListener('click', onMenuClick);    #D
    };
  }, [isExpanded]);
  const menuRef = useRef();    #E
  return (
    <main>
      <button onClick={() => setExpanded(true)}>Show menu</button>
      {isExpanded && (
        <aside
          ref={menuRef}    #F
          style={{border: '1px solid black', padding: '1em'}}
        >
          This is the menu
        </aside>
      )}
    </main>
  );
}
```

#A This timer, to stop clicks inside the menu itself to close the menu, we will stop any click events from propagating beyond the menu element
#B Before we assign a listener to the menu element, we need to actually capture a reference to said element via the ref
#C We now also add a listener to the menu element
#D And we still remove these listeners on cleanup
#E We need a useRef to store our reference to the menu element
#F Finally we need to assign our reference to the proper JSX element

rq08-menu

If you run this app in a browser, you will see the exact same thing as before in figure 8.6. Observe that when you expand the menu, you can collapse the menu by clicking anywhere except on the menu itself (i.e. except when inside the big box with a black border).

Notice how we use a variety of hooks and even combine React event listeners with DOM event listeners to achieve this result. All of these low-level elements go together nicely in a simple component that does exactly what we want it to do.

8.10 Quiz

1. What is the correct way to add a click listener to a JSX button?
   a) `<button click={onClick}>Click me</click>`
   b) `<button click="onClick">Click me</click>`
   c) `<button onClick={onClick}>Click me</click>`
   d) `<button onClick="onClick">Click me</click>`
2. React event handlers can also be assigned by calling `addEventListener` on the JSX element, *true* or *false*?
3. Event bubbling is rare and only happens for a few types of events, *true* or *false*?
4. If you don't want a form to reload the page when submitted, what do you do?
   a) You assign the listener as a capture listener
   b) You invoke `evt.preventDefault()` on the event object
   c) You have to assign the listener manually on the HTML element
   d) You invoke `evt.stopPropagation()` on the event object.
5. You cannot listen to events on HTML elements not inside the React application, *true* or *false*?

8.11 Summary

- Events are essential to creating interactive web applications. Events are the way an application reacts to user input.
- Events are also used to communicate between HTML elements and the React application, e.g. when a resource has loaded or a video has finished playing.
- React event listeners are assigned to JSX elements using a property. A click listener is assigned using `onClick`, a paste listener is assigned using `onPaste`, etc.
- Event listeners are invoked with an event object, which can be used to tell which event occurred, what element caused the event to happen, which phase in the event propagation is currently going, and several other properties relevant for the specific event.
- Event objects are also used to interrupt the normal progression of event handling by either preventing the browser's default action, stopping further propagation of the event to other event listeners, or even a combination of both.
- Events propagate from the window object down to the target element and back up to the window again. You can assign listeners to listen for events as they go up or down the tree in order to e.g. interrupt the regular flow or listen for events on multiple targets.
- You can still assign regular event listeners to JavaScript objects and HTML elements using regular JavaScript. You even have to do this sometimes, as not all event types are supported in React, nor are all HTML elements accessible through React.

8.12 Quiz answers

1. The correct answer is c: <button onClick={onClick}>Click me</click>
2. *False*. React event handlers can only be assigned using a property, e.g. `onClick`. They cannot be assigned using `addEventListener`.
3. *False*. All events bubble in React—even some events that don't bubble in HTML.
4. The correct answer is *b*. An HTML form would cause the page to reload on submission as the default action. If you want to cancel the default action, you must invoke `evt.preventDefault()` on the event object.
5. *False*. You can use manual DOM event listeners to listen for events on any HTML element as long as you have a reference to it.