Learn ReactDescribing the UIReact is a JavaScript library for rendering user interfaces (UI). UI is built from small units like buttons, text, and images. React lets you combine them into reusable, nestable components. From web sites to phone apps, everything on the screen can be broken down into components. In this chapter, you'll learn to create, customize, and conditionally display React components.
In this chapter
How to write your first React component
When and how to create multi-component files
How to add markup to JavaScript with JSX
How to use curly braces with JSX to access JavaScript functionality from your components
How to configure components with props
How to conditionally render components
How to render multiple components at a time
How to avoid confusing bugs by keeping components pure

Your first component
React applications are built from isolated pieces of UI called components. A React component is a JavaScript function that you can sprinkle with markup. Components can be as small as a button, or as large as an entire page. Here is a Gallery component rendering three Profile components:
App.jsApp.js Download ResetFork991234567891011121314151617181920function Profile() { return ( );}export default function Gallery() { return (<section> <h1>Amazing scientists</h1> <Profile /> <Profile /> <Profile /> </section>);}Show more
Ready to learn this topic?Read Your First Component to learn how to declare and use React components.Read More
Importing and exporting components
You can declare many components in one file, but large files can get difficult to navigate. To solve this, you can export a component into its own file, and then import that component from another file:
Gallery.jsProfile.jsGallery.js ResetForkimport Profile from './Profile.js';

```
export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```

Ready to learn this topic?Read Importing and Exporting Components to learn how to split components into their own files.Read More

Writing markup with JSX

Each React component is a JavaScript function that may contain some markup that React renders into the browser. React components use a syntax extension called JSX to represent that markup. JSX looks a lot like HTML, but it is a bit stricter and can display dynamic information.

If we paste existing HTML markup into a React component, it won't always work:

App.jsApp.js Download ResetForkexport default function TodoList() {
  return (
    // This doesn't quite work!
    <h1>Hedy Lamarr's Todos</h1>
    <img
      src="https://i.imgur.com/yXOvdOSs.jpg"
      alt="Hedy Lamarr"
      class="photo"
    >

      Invent new traffic lights
      Rehearse a movie scene
      Improve spectrum technology

Show more

If you have existing HTML like this, you can fix it using a converter:

App.jsApp.js Download ResetForkexport default function TodoList() {
  return (
    <>
      <h1>Hedy Lamarr's Todos</h1>
      <img
        src="https://i.imgur.com/yXOvdOSs.jpg"
        alt="Hedy Lamarr"
        className="photo"
      />

        Invent new traffic lights
        Rehearse a movie scene
        Improve spectrum technology

    </>
  );
}

Show more

Ready to learn this topic?Read Writing Markup with JSX to learn how to write valid JSX.Read More

JavaScript in JSX with curly braces
JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to “open a window” to JavaScript:
App.jsApp.js Download ResetForkconst person = {
  name: 'Gregorio Y. Zara',
  theme: {
    backgroundColor: 'black',
    color: 'pink'
  }
};

export default function TodoList() {
  return (

      <h1>{person.name}'s Todos</h1>
      <img
        className="avatar"
        src="https://i.imgur.com/7vQD0fPs.jpg"
        alt="Gregorio Y. Zara"
      />

        Improve the videophone
        Prepare aeronautics lectures
        Work on the alcohol-fuelled engine

  );
}

Show more
Ready to learn this topic?Read JavaScript in JSX with Curly Braces to learn how to access JavaScript data from JSX.Read More
Passing props to a component
React components use props to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, functions, and even JSX!
App.jsutils.jsApp.js ResetForkimport { getImageUrl } from './utils.js'

export default function Profile() {
  return (
    <Card>
      <Avatar
        size={100}

```
        person={{
          name: 'Katsuko Saruhashi',
          imageId: 'YfeOqp2'
        }}
      />
    </Card>
  );
}

function Avatar({ person, size }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}

function Card({ children }) {
  return (
    <div className="card">
      {children}
    </div>
  );
}
```

Show more

Ready to learn this topic?Read Passing Props to a Component to learn how to pass and read props.Read More

Conditional rendering

Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, &&, and ? : operators.

In this example, the JavaScript && operator is used to conditionally render a checkmark:

App.jsApp.js Download ResetForkfunction Item({ name, isPacked }) {

```
  return (
    <li className="item">
      {name} {isPacked && ' '}
    </li>
  );
}

export default function PackingList() {
```

```
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Show more

Ready to learn this topic?Read Conditional Rendering to learn the different ways to render content conditionally.Read More

Rendering lists

You will often want to display multiple similar components from a collection of data. You can use JavaScript’s filter() and map() with React to filter and transform your array of data into an array of components.

For each array item, you will need to specify a key. Usually, you will want to use an ID from the database as a key. Keys let React keep track of each item’s place in the list even if the list changes.

App.jsdata.jsutils.jsApp.js ResetForkimport { people } from './data.js';
import { getImageUrl } from './utils.js';

```
export default function List() {
  const listItems = people.map(person =>
    <li key={person.id}>
      <img
        src={getImageUrl(person)}
        alt={person.name}
      />
      <p>
        <b>{person.name}:</b>
        {' ' + person.profession + ' '}
        known for {person.accomplishment}
      </p>
```

```
    </li>
  );
  return (
    <article>
      <h1>Scientists</h1>
      <ul>{listItems}</ul>
    </article>
  );
}
```

Show more
Ready to learn this topic?Read Rendering Lists to learn how to render a list of components, and how to choose a key.Read More
Keeping components pure
Some JavaScript functions are pure. A pure function:

Minds its own business. It does not change any objects or variables that existed before it was called.
Same inputs, same output. Given the same inputs, a pure function should always return the same result.

By strictly only writing your components as pure functions, you can avoid an entire class of baffling bugs and unpredictable behavior as your codebase grows. Here is an example of an impure component:
App.jsApp.js Download ResetForklet guest = 0;

```
function Cup() {
  // Bad: changing a preexisting variable!
  guest = guest + 1;
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup />
      <Cup />
      <Cup />
    </>
  );
}
```

Show more
You can make this component pure by passing a prop instead of modifying a preexisting variable:
App.jsApp.js Download ResetForkfunction Cup({ guest }) {

```
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup guest={1} />
      <Cup guest={2} />
      <Cup guest={3} />
    </>
  );
}
```

Ready to learn this topic?Read Keeping Components Pure to learn how to write components as pure, predictable functions.Read More
What's next?
Head over to Your First Component to start reading this chapter page by page!
Or, if you're already familiar with these topics, why not read about Adding Interactivity?
NextYour First Component
Learn ReactDescribing the UIYour First ComponentComponents are one of the core concepts of React. They are the foundation upon which you build user interfaces (UI), which makes them the perfect place to start your React journey!
You will learn
What a component is
What role components play in a React application
How to write your first React component

Components: UI building blocks
On the Web, HTML lets us create rich structured documents with its built-in set of tags like <h1> and <li>:

```
<article>
  <h1>My First Component</h1>
  <ol>
    <li>Components: UI Building Blocks</li>
    <li>Defining a Component</li>
    <li>Using a Component</li>
  </ol>
</article>
```

This markup represents this article <article>, its heading <h1>, and an (abbreviated) table of contents as an ordered list <ol>. Markup like this, combined with CSS for style, and JavaScript for interactivity, lies behind every sidebar, avatar, modal, dropdown—every piece of UI you see on the Web.
React lets you combine your markup, CSS, and JavaScript into custom "components", reusable UI elements for your app. The table of contents code you saw above could be

turned into a <TableOfContents /> component you could render on every page. Under the hood, it still uses the same HTML tags like <article>, <h1>, etc.
Just like with HTML tags, you can compose, order and nest components to design whole pages. For example, the documentation page you're reading is made out of React components:

```
<PageLayout>
  <NavigationHeader>
    <SearchBar />
    <Link to="/docs">Docs</Link>
  </NavigationHeader>
  <Sidebar />
  <PageContent>
    <TableOfContents />
    <DocumentationText />
  </PageContent>
</PageLayout>
```

As your project grows, you will notice that many of your designs can be composed by reusing components you already wrote, speeding up your development. Our table of contents above could be added to any screen with <TableOfContents />! You can even jumpstart your project with the thousands of components shared by the React open source community like Chakra UI and Material UI.
Defining a component
Traditionally when creating web pages, web developers marked up their content and then added interaction by sprinkling on some JavaScript. This worked great when interaction was a nice-to-have on the web. Now it is expected for many sites and all apps. React puts interactivity first while still using the same technology: a React component is a JavaScript function that you can sprinkle with markup. Here's what that looks like (you can edit the example below):
App.jsApp.js Download ResetForkexport default function Profile() {

```
  return (
    <img
      src="https://i.imgur.com/MK3eW3Am.jpg"
      alt="Katherine Johnson"
    />
  )
}
```

And here's how to build a component:
Step 1: Export the component
The export default prefix is a standard JavaScript syntax (not specific to React). It lets you mark the main function in a file so that you can later import it from other files. (More on importing in Importing and Exporting Components!)
Step 2: Define the function
With function Profile() { } you define a JavaScript function with the name Profile.

PitfallReact components are regular JavaScript functions, but their names must start with a capital letter or they won’t work!
Step 3: Add markup
The component returns an <img /> tag with src and alt attributes. <img /> is written like HTML, but it is actually JavaScript under the hood! This syntax is called JSX, and it lets you embed markup inside JavaScript.
Return statements can be written all on one line, as in this component:
return ;
But if your markup isn’t all on the same line as the return keyword, you must wrap it in a pair of parentheses:
return (   );
PitfallWithout parentheses, any code on the lines after return will be ignored!
Using a component
Now that you’ve defined your Profile component, you can nest it inside other components. For example, you can export a Gallery component that uses multiple Profile components:
App.jsApp.js Download ResetForkfunction Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}

Show more
What the browser sees
Notice the difference in casing:

<section> is lowercase, so React knows we refer to an HTML tag.
<Profile /> starts with a capital P, so React knows that we want to use our component called Profile.

And Profile contains even more HTML: <img />. In the end, this is what the browser sees:

<section> <h1>Amazing scientists</h1>   </section>
Nesting and organizing components
Components are regular JavaScript functions, so you can keep multiple components in the same file. This is convenient when components are relatively small or tightly related to each other. If this file gets crowded, you can always move Profile to a separate file. You will learn how to do this shortly on the page about imports.
Because the Profile components are rendered inside Gallery—even several times!—we can say that Gallery is a parent component, rendering each Profile as a “child”. This is part of the magic of React: you can define a component once, and then use it in as many places and as many times as you like.
PitfallComponents can render other components, but you must never nest their definitions:export default function Gallery() { // Ø=Ý4 Never define a component inside another component! function Profile() {   // ... } // ...}The snippet above is very slow and causes bugs. Instead, define every component at the top level:export default function Gallery() { // ...}// ' Declare components at the top levelfunction Profile() { // ...}When a child component needs some data from a parent, pass it by props instead of nesting definitions.
Deep DiveComponents all the way down Show DetailsYour React application begins at a “root” component. Usually, it is created automatically when you start a new project. For example, if you use CodeSandbox or Create React App, the root component is defined in src/App.js. If you use the framework Next.js, the root component is defined in pages/index.js. In these examples, you’ve been exporting root components.Most React apps use components all the way down. This means that you won’t only use components for reusable pieces like buttons, but also for larger pieces like sidebars, lists, and ultimately, complete pages! Components are a handy way to organize UI code and markup, even if some of them are only used once.React-based frameworks take this a step further. Instead of using an empty HTML file and letting React “take over” managing the page with JavaScript, they also generate the HTML automatically from your React components. This allows your app to show some content before the JavaScript code loads.Still, many websites only use React to add interactivity to existing HTML pages. They have many root components instead of a single one for the entire page. You can use as much—or as little—React as you need.
RecapYou’ve just gotten your first taste of React! Let’s recap some key points.

React lets you create components, reusable UI elements for your app.

In a React app, every piece of UI is a component.

React components are regular JavaScript functions except:

Their names always begin with a capital letter.

They return JSX markup.

Try out some challenges1. Export the component 2. Fix the return statement 3. Spot the mistake 4. Your own component Challenge 1 of 4: Export the component This sandbox doesn’t work because the root component is not exported:App.jsApp.js Download ResetForkfunction Profile() {
  return (
    <img
      src="https://i.imgur.com/lICfvbD.jpg"
      alt="Aklilu Lemma"
    />
  );
}

Try to fix it yourself before looking at the solution! Show solutionNext ChallengePreviousDescribing the UINextImporting and Exporting Components Learn ReactDescribing the UIImporting and Exporting ComponentsThe magic of components lies in their reusability: you can create components that are composed of other components. But as you nest more and more components, it often makes sense to start splitting them into different files. This lets you keep your files easy to scan and reuse components in more places.
You will learn
What a root component file is
How to import and export a component
When to use default and named imports and exports
How to import and export multiple components from one file
How to split components into multiple files

The root component file
In Your First Component, you made a Profile component and a Gallery component that renders it:
App.jsApp.js Download ResetFork991234567891011121314151617181920function Profile() { return (   );}export default function Gallery() { return (<section>   <h1>Amazing scientists</h1>   <Profile />   <Profile />   <Profile />   </section>);}Show more
These currently live in a root component file, named App.js in this example. In Create React App, your app lives in src/App.js. Depending on your setup, your root component could be in another file, though. If you use a framework with file-based routing, such as Next.js, your root component will be different for every page.
Exporting and importing a component
What if you want to change the landing screen in the future and put a list of science books there? Or place all the profiles somewhere else? It makes sense to move Gallery and Profile out of the root component file. This will make them more modular and

reusable in other files. You can move a component in three steps:

Make a new JS file to put the components in.
Export your function component from that file (using either default or named exports).
Import it in the file where you'll use the component (using the corresponding technique for importing default or named exports).

Here both Profile and Gallery have been moved out of App.js into a new file called Gallery.js. Now you can change App.js to import Gallery from Gallery.js:
App.jsGallery.jsApp.js ResetForkimport Gallery from './Gallery.js';

```
export default function App() {
  return (
    <Gallery />
  );
}
```

Notice how this example is broken down into two component files now:

Gallery.js:

Defines the Profile component which is only used within the same file and is not exported.
Exports the Gallery component as a default export.

App.js:

Imports Gallery as a default import from Gallery.js.
Exports the root App component as a default export.

NoteYou may encounter files that leave off the .js file extension like so:import Gallery from './Gallery';Either './Gallery.js' or './Gallery' will work with React, though the former is closer to how native ES Modules work.
Deep DiveDefault vs named exports Show DetailsThere are two primary ways to export values with JavaScript: default exports and named exports. So far, our examples have only used default exports. But you can use one or both of them in the same file. A file can have no more than one default export, but it can have as many named exports as you like.How you export your component dictates how you must import it. You will get an error if you try to import a default export the same way you would a named export! This chart can help you keep track:SyntaxExport statementImport statementDefaultexport default function Button() {}import Button from './Button.js';Namedexport function Button() {}import { Button } from './Button.js';When you

write a default import, you can put any name you want after import. For example, you could write import Banana from './Button.js' instead and it would still provide you with the same default export. In contrast, with named imports, the name has to match on both sides. That's why they are called named imports!People often use default exports if the file exports only one component, and use named exports if it exports multiple components and values. Regardless of which coding style you prefer, always give meaningful names to your component functions and the files that contain them. Components without names, like export default () => {}, are discouraged because they make debugging harder.
Exporting and importing multiple components from the same file
What if you want to show just one Profile instead of a gallery? You can export the Profile component, too. But Gallery.js already has a default export, and you can't have two default exports. You could create a new file with a default export, or you could add a named export for Profile. A file can only have one default export, but it can have numerous named exports!
NoteTo reduce the potential confusion between default and named exports, some teams choose to only stick to one style (default or named), or avoid mixing them in a single file. Do what works best for you!
First, export Profile from Gallery.js using a named export (no default keyword):
export function Profile() { // ...}
Then, import Profile from Gallery.js to App.js using a named import (with the curly braces):
import { Profile } from './Gallery.js';
Finally, render <Profile /> from the App component:
export default function App() { return <Profile />;}
Now Gallery.js contains two exports: a default Gallery export, and a named Profile export. App.js imports both of them. Try editing <Profile /> to <Gallery /> and back in this example:
App.jsGallery.jsApp.js ResetForkimport Gallery from './Gallery.js';
import { Profile } from './Gallery.js';

```
export default function App() {
  return (
    <Profile />
  );
}
```

Now you're using a mix of default and named exports:

Gallery.js:

Exports the Profile component as a named export called Profile.
Exports the Gallery component as a default export.

App.js:

Imports Profile as a named import called Profile from Gallery.js.
Imports Gallery as a default import from Gallery.js.
Exports the root App component as a default export.

RecapOn this page you learned:
What a root component file is
How to import and export a component
When and how to use default and named imports and exports
How to export multiple components from the same file

Try out some challengesChallenge 1 of 1: Split the components further Currently, Gallery.js exports both Profile and Gallery, which is a bit confusing.Move the Profile component to its own Profile.js, and then change the App component to render both <Profile /> and <Gallery /> one after another.You may use either a default or a named export for Profile, but make sure that you use the corresponding import syntax in both App.js and Gallery.js! You can refer to the table from the deep dive above:SyntaxExport statementImport statementDefaultexport default function Button() {}import Button from './Button.js';Namedexport function Button() {}import { Button } from './Button.js';App.jsGallery.jsProfile.jsGallery.js ResetFork// Move me to Profile.js!

```
export function Profile() {
  return (
    <img
      src="https://i.imgur.com/QIrZWGIs.jpg"
      alt="Alan L. Hart"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```

Show moreAfter you get it working with one kind of exports, make it work with the other kind. Show hint Show solutionPreviousYour First ComponentNextWriting Markup with JSX

Learn ReactDescribing the UIWriting Markup with JSXJSX is a syntax extension for JavaScript that lets you write HTML-like markup inside a JavaScript file. Although there are other ways to write components, most React developers prefer the conciseness of JSX, and most codebases use it.
You will learn
Why React mixes markup with rendering logic
How JSX is different from HTML
How to display information with JSX

JSX: Putting markup into JavaScript
The Web has been built on HTML, CSS, and JavaScript. For many years, web developers kept content in HTML, design in CSS, and logic in JavaScript—often in separate files! Content was marked up inside HTML while the page's logic lived separately in JavaScript:
HTMLJavaScript
But as the Web became more interactive, logic increasingly determined content. JavaScript was in charge of the HTML! This is why in React, rendering logic and markup live together in the same place—components.
Sidebar.js React componentForm.js React component
Keeping a button's rendering logic and markup together ensures that they stay in sync with each other on every edit. Conversely, details that are unrelated, such as the button's markup and a sidebar's markup, are isolated from each other, making it safer to change either of them on their own.
Each React component is a JavaScript function that may contain some markup that React renders into the browser. React components use a syntax extension called JSX to represent that markup. JSX looks a lot like HTML, but it is a bit stricter and can display dynamic information. The best way to understand this is to convert some HTML markup to JSX markup.
NoteJSX and React are two separate things. They're often used together, but you can use them independently of each other. JSX is a syntax extension, while React is a JavaScript library.
Converting HTML to JSX
Suppose that you have some (perfectly valid) HTML:
<h1>Hedy Lamarr's Todos</h1><img src="https://i.imgur.com/yXOvdOSs.jpg" alt="Hedy Lamarr" class="photo"> Invent new traffic lights Rehearse a movie scene Improve the spectrum technology
And you want to put it into your component:
export default function TodoList() { return (// ???)}
If you copy and paste it as is, it will not work:
App.jsApp.js Download ResetForkexport default function TodoList() {
```
  return (
    // This doesn't quite work!
    <h1>Hedy Lamarr's Todos</h1>
    <img
      src="https://i.imgur.com/yXOvdOSs.jpg"
      alt="Hedy Lamarr"
```

```
  class="photo"
>
<ul>
  <li>Invent new traffic lights
  <li>Rehearse a movie scene
  <li>Improve the spectrum technology
</ul>
```

Show more

This is because JSX is stricter and has a few more rules than HTML! If you read the error messages above, they'll guide you to fix the markup, or you can follow the guide below.

NoteMost of the time, React's on-screen error messages will help you find where the problem is. Give them a read if you get stuck!

The Rules of JSX

1. Return a single root element

To return multiple elements from a component, wrap them with a single parent tag.

For example, you can use a :

```
<div>  <h1>Hedy Lamarr's Todos</h1>  <img     src="https://i.imgur.com/
yXOvdOSs.jpg"     alt="Hedy Lamarr"     class="photo"  >  <ul>    ...  </ul></div>
```

If you don't want to add an extra to your markup, you can write <> and </> instead:

```
<>  <h1>Hedy Lamarr's Todos</h1>  <img     src="https://i.imgur.com/
yXOvdOSs.jpg"     alt="Hedy Lamarr"     class="photo"  >  <ul>    ...  </ul></>
```

This empty tag is called a Fragment. Fragments let you group things without leaving any trace in the browser HTML tree.

Deep DiveWhy do multiple JSX tags need to be wrapped? Show DetailsJSX looks like HTML, but under the hood it is transformed into plain JavaScript objects. You can't return two objects from a function without wrapping them into an array. This explains why you also can't return two JSX tags without wrapping them into another tag or a Fragment.

2. Close all the tags

JSX requires tags to be explicitly closed: self-closing tags like <img> must become <img />, and wrapping tags like oranges must be written as oranges.

This is how Hedy Lamarr's image and list items look closed:

```
<>  <img     src="https://i.imgur.com/yXOvdOSs.jpg"     alt="Hedy Lamarr"
class="photo"   />  <ul>    <li>Invent new traffic lights</li>    <li>Rehearse a movie
scene</li>    <li>Improve the spectrum technology</li>  </ul></>
```

3. camelCase all most of the things!

JSX turns into JavaScript and attributes written in JSX become keys of JavaScript objects. In your own components, you will often want to read those attributes into variables. But JavaScript has limitations on variable names. For example, their names can't contain dashes or be reserved words like class.

This is why, in React, many HTML and SVG attributes are written in camelCase. For example, instead of stroke-width you use strokeWidth. Since class is a reserved word, in React you write className instead, named after the corresponding DOM property:

You can find all these attributes in the list of DOM component props. If you get one wrong, don’t worry—React will print a message with a possible correction to the browser console.
PitfallFor historical reasons, aria-* and data-* attributes are written as in HTML with dashes.
Pro-tip: Use a JSX Converter
Converting all these attributes in existing markup can be tedious! We recommend using a converter to translate your existing HTML and SVG to JSX. Converters are very useful in practice, but it’s still worth understanding what is going on so that you can comfortably write JSX on your own.
Here is your final result:
App.jsApp.js Download ResetForkexport default function TodoList() {
  return (
    <>
      <h1>Hedy Lamarr's Todos</h1>
      <img
        src="https://i.imgur.com/yXOvdOSs.jpg"
        alt="Hedy Lamarr"
        className="photo"
      />

        Invent new traffic lights
        Rehearse a movie scene
        Improve the spectrum technology

    </>
  );
}

Show more
RecapNow you know why JSX exists and how to use it in components:
React components group rendering logic together with markup because they are related.
JSX is similar to HTML, with a few differences. You can use a converter if you need to.
Error messages will often point you in the right direction to fixing your markup.

Try out some challengesChallenge 1 of 1: Convert some HTML to JSX This HTML was pasted into a component, but it’s not valid JSX. Fix it:App.jsApp.js Download ResetForkexport default function Bio() {
  return (

      <h1>Welcome to my website!</h1>

    <p class="summary">

```
      You can find my thoughts here.
      <br><br>
      <b>And <i>pictures</b></i> of scientists!
    </p>
  );
}
```

Whether to do it by hand or using the converter is up to you! Show solutionPreviousImporting and Exporting ComponentsNextJavaScript in JSX with Curly Braces

Learn ReactDescribing the UIJavaScript in JSX with Curly BracesJSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to open a window to JavaScript.

You will learn

How to pass strings with quotes
How to reference a JavaScript variable inside JSX with curly braces
How to call a JavaScript function inside JSX with curly braces
How to use a JavaScript object inside JSX with curly braces

Passing strings with quotes

When you want to pass a string attribute to JSX, you put it in single or double quotes:

App.jsApp.js Download ResetFork9912345678910export default function Avatar() { return ( );}

Here, "https://i.imgur.com/7vQD0fPs.jpg" and "Gregorio Y. Zara" are being passed as strings.

But what if you want to dynamically specify the src or alt text? You could use a value from JavaScript by replacing " and " with { and }:

App.jsApp.js Download ResetForkexport default function Avatar() {
```
  const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';
  const description = 'Gregorio Y. Zara';
  return (
    <img
      className="avatar"
      src={avatar}
      alt={description}
    />
  );
}
```

Notice the difference between className="avatar", which specifies an "avatar" CSS class name that makes the image round, and src={avatar} that reads the value of the JavaScript variable called avatar. That’s because curly braces let you work with

JavaScript right there in your markup!
Using curly braces: A window into the JavaScript world
JSX is a special way of writing JavaScript. That means it's possible to use JavaScript inside it—with curly braces { }. The example below first declares a name for the scientist, name, then embeds it with curly braces inside the <h1>:
App.jsApp.js Download ResetForkexport default function TodoList() {
  const name = 'Gregorio Y. Zara';
  return (
    <h1>{name}'s To Do List</h1>
  );
}

Try changing the name's value from 'Gregorio Y. Zara' to 'Hedy Lamarr'. See how the list title changes?
Any JavaScript expression will work between curly braces, including function calls like formatDate():
App.jsApp.js Download ResetForkconst today = new Date();

function formatDate(date) {
  return new Intl.DateTimeFormat(
    'en-US',
    { weekday: 'long' }
).format(date);
}

export default function TodoList() {
  return (
    <h1>To Do List for {formatDate(today)}</h1>
  );
}

Where to use curly braces
You can only use curly braces in two ways inside JSX:

As text directly inside a JSX tag: <h1>{name}'s To Do List</h1> works, but <{tag} >Gregorio Y. Zara's To Do List</{tag}> will not.
As attributes immediately following the = sign: src={avatar} will read the avatar variable, but src="{avatar}" will pass the string "{avatar}".

Using "double curlies": CSS and other objects in JSX
In addition to strings, numbers, and other JavaScript expressions, you can even pass objects in JSX. Objects are also denoted with curly braces, like { name: "Hedy Lamarr", inventions: 5 }. Therefore, to pass a JS object in JSX, you must wrap the object in another pair of curly braces: person={{ name: "Hedy Lamarr", inventions: 5 }}.

You may see this with inline CSS styles in JSX. React does not require you to use inline styles (CSS classes work great for most cases). But when you need an inline style, you pass an object to the style attribute:
App.jsApp.js Download ResetForkexport default function TodoList() {
  return (
    <ul style={{
      backgroundColor: 'black',
      color: 'pink'
    }}>
      Improve the videophone
      Prepare aeronautics lectures
      Work on the alcohol-fuelled engine

  );
}

Try changing the values of backgroundColor and color.
You can really see the JavaScript object inside the curly braces when you write it like this:
<ul style={ {   backgroundColor: 'black',   color: 'pink' }}>
The next time you see {{ and }} in JSX, know that it's nothing more than an object inside the JSX curlies!
PitfallInline style properties are written in camelCase. For example, HTML <ul style="background-color: black"> would be written as <ul style={{ backgroundColor: 'black' }}> in your component.
More fun with JavaScript objects and curly braces
You can move several expressions into one object, and reference them in your JSX inside curly braces:
App.jsApp.js Download ResetForkconst person = {
  name: 'Gregorio Y. Zara',
  theme: {
    backgroundColor: 'black',
    color: 'pink'
  }
};

export default function TodoList() {
  return (

      <h1>{person.name}'s Todos</h1>
      <img
        className="avatar"
        src="https://i.imgur.com/7vQD0fPs.jpg"
        alt="Gregorio Y. Zara"
      />

```
      <ul>
        <li>Improve the videophone</li>
        <li>Prepare aeronautics lectures</li>
        <li>Work on the alcohol-fuelled engine</li>
      </ul>
    </div>
  );
}
```

Show more
In this example, the person JavaScript object contains a name string and a theme object:
const person = { name: 'Gregorio Y. Zara', theme: {   backgroundColor: 'black',   color: 'pink' }};
The component can use these values from person like so:
  <h1>{person.name}'s Todos</h1>
JSX is very minimal as a templating language because it lets you organize data and logic using JavaScript.
RecapNow you know almost everything about JSX:
JSX attributes inside quotes are passed as strings.
Curly braces let you bring JavaScript logic and variables into your markup.
They work inside the JSX tag content or immediately after = in attributes.
{{ and }} is not special syntax: it’s a JavaScript object tucked inside JSX curly braces.

Try out some challenges1. Fix the mistake 2. Extract information into an object 3. Write an expression inside JSX curly braces Challenge 1 of 3: Fix the mistake This code crashes with an error saying Objects are not valid as a React child:App.jsApp.js Download ResetForkconst person = {

```
  name: 'Gregorio Y. Zara',
  theme: {
    backgroundColor: 'black',
    color: 'pink'
  }
};

export default function TodoList() {
  return (
    <div style={person.theme}>
      <h1>{person}'s Todos</h1>
      <img
        className="avatar"
        src="https://i.imgur.com/7vQD0fPs.jpg"
        alt="Gregorio Y. Zara"
      />
      <ul>
        <li>Improve the videophone</li>
```

```
        <li>Prepare aeronautics lectures</li>
        <li>Work on the alcohol-fuelled engine</li>
      </ul>
    </div>
  );
}
```

Show moreCan you find the problem? Show hint Show solutionNext ChallengePreviousWriting Markup with JSXNextPassing Props to a Component
Learn ReactDescribing the UIPassing Props to a ComponentReact components use props to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, and functions.
You will learn
How to pass props to a component
How to read props from a component
How to specify default values for props
How to pass some JSX to a component
How props change over time

Familiar props
Props are the information that you pass to a JSX tag. For example, className, src, alt, width, and height are some of the props you can pass to an <img>:
App.jsApp.js Download ResetFork991234567891011121314151617 18function Avatar() { return ( );}export default function Profile() { return (<Avatar />);}Show more
The props you can pass to an <img> tag are predefined (ReactDOM conforms to the HTML standard). But you can pass any props to your own components, such as <Avatar>, to customize them. Here’s how!
Passing props to a component
In this code, the Profile component isn’t passing any props to its child component, Avatar:
export default function Profile() { return (<Avatar />);}
You can give Avatar some props in two steps.
Step 1: Pass props to the child component
First, pass some props to Avatar. For example, let’s pass two props: person (an object), and size (a number):
export default function Profile() { return (<Avatar person={{ name: 'Lin Lanying', imageId: '1bX5QH6' }} size={100} />);}
NoteIf double curly braces after person= confuse you, recall they’re merely an object inside the JSX curlies.
Now you can read these props inside the Avatar component.
Step 2: Read props inside the child component
You can read these props by listing their names person, size separated by the commas

inside ({ and }) directly after function Avatar. This lets you use them inside the Avatar code, like you would with a variable.
function Avatar({ person, size }) { // person and size are available here}
Add some logic to Avatar that uses the person and size props for rendering, and you're done.
Now you can configure Avatar to render in many different ways with different props. Try tweaking the values!
App.jsutils.jsApp.js ResetForkimport { getImageUrl } from './utils.js';

```
function Avatar({ person, size }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}

export default function Profile() {
  return (
    <div>
      <Avatar
        size={100}
        person={{
          name: 'Katsuko Saruhashi',
          imageId: 'YfeOqp2'
        }}
      />
      <Avatar
        size={80}
        person={{
          name: 'Aklilu Lemma',
          imageId: 'OKS67lh'
        }}
      />
      <Avatar
        size={50}
        person={{
          name: 'Lin Lanying',
          imageId: '1bX5QH6'
        }}
      />
    </div>
```

```
  );
}
```

Show more

Props let you think about parent and child components independently. For example, you can change the person or the size props inside Profile without having to think about how Avatar uses them. Similarly, you can change how the Avatar uses these props, without looking at the Profile.

You can think of props like “knobs” that you can adjust. They serve the same role as arguments serve for functions—in fact, props are the only argument to your component! React component functions accept a single argument, a props object:

```
function Avatar(props) {  let person = props.person;  let size = props.size;  // ...}
```

Usually you don’t need the whole props object itself, so you destructure it into individual props.

PitfallDon’t miss the pair of { and } curlies inside of (and) when declaring props:function Avatar({ person, size }) { // ...}This syntax is called “destructuring” and is equivalent to reading properties from a function parameter:function Avatar(props) { let person = props.person; let size = props.size; // ...}

Specifying a default value for a prop

If you want to give a prop a default value to fall back on when no value is specified, you can do it with the destructuring by putting = and the default value right after the parameter:

```
function Avatar({ person, size = 100 }) {  // ...}
```

Now, if <Avatar person={...} /> is rendered with no size prop, the size will be set to 100.

The default value is only used if the size prop is missing or if you pass size={undefined}. But if you pass size={null} or size={0}, the default value will not be used.

Forwarding props with the JSX spread syntax

Sometimes, passing props gets very repetitive:

```
function Profile({ person, size, isSepia, thickBorder }) {  return (    <div
className="card">      <Avatar        person={person}        size={size}
isSepia={isSepia}        thickBorder={thickBorder}      />    </div>  );}
```

There’s nothing wrong with repetitive code—it can be more legible. But at times you may value conciseness. Some components forward all of their props to their children, like how this Profile does with Avatar. Because they don’t use any of their props directly, it can make sense to use a more concise “spread” syntax:

```
function Profile(props) {  return (    <div className="card">      <Avatar {...props} />    </
div>  );}
```

This forwards all of Profile’s props to the Avatar without listing each of their names.

Use spread syntax with restraint. If you’re using it in every other component, something is wrong. Often, it indicates that you should split your components and pass children as JSX. More on that next!

Passing JSX as children

It is common to nest built-in browser tags:

```
<div>  <img /></div>
```

Sometimes you’ll want to nest your own components the same way:

```
<Card>  <Avatar /></Card>
```

When you nest content inside a JSX tag, the parent component will receive that content in a prop called children. For example, the Card component below will receive a children prop set to <Avatar /> and render it in a wrapper div:
App.jsAvatar.jsutils.jsApp.js ResetForkimport Avatar from './Avatar.js';

```
function Card({ children }) {
  return (
    <div className="card">
      {children}
    </div>
  );
}

export default function Profile() {
  return (
    <Card>
      <Avatar
        size={100}
        person={{
          name: 'Katsuko Saruhashi',
          imageId: 'YfeOqp2'
        }}
      />
    </Card>
  );
}
```

Show more
Try replacing the <Avatar> inside <Card> with some text to see how the Card component can wrap any nested content. It doesn’t need to “know” what’s being rendered inside of it. You will see this flexible pattern in many places.
You can think of a component with a children prop as having a “hole” that can be “filled in” by its parent components with arbitrary JSX. You will often use the children prop for visual wrappers: panels, grids, etc.
Illustrated by Rachel Lee Nabors
How props change over time
The Clock component below receives two props from its parent component: color and time. (The parent component’s code is omitted because it uses state, which we won’t dive into just yet.)
Try changing the color in the select box below:
Clock.jsClock.js ResetForkexport default function Clock({ color, time }) {

```
  return (
    <h1 style={{ color: color }}>
      {time}
    </h1>
  );
```

}

This example illustrates that a component may receive different props over time. Props are not always static! Here, the time prop changes every second, and the color prop changes when you select another color. Props reflect a component’s data at any point in time, rather than only in the beginning.
However, props are immutable—a term from computer science meaning “unchangeable”. When a component needs to change its props (for example, in response to a user interaction or new data), it will have to “ask” its parent component to pass it different props—a new object! Its old props will then be cast aside, and eventually the JavaScript engine will reclaim the memory taken by them.
Don’t try to “change props”. When you need to respond to the user input (like changing the selected color), you will need to “set state”, which you can learn about in State: A Component’s Memory.
Recap
To pass props, add them to the JSX, just like you would with HTML attributes.
To read props, use the function Avatar({ person, size }) destructuring syntax.
You can specify a default value like size = 100, which is used for missing and undefined props.
You can forward all props with <Avatar {...props} /> JSX spread syntax, but don’t overuse it!
Nested JSX like <Card><Avatar /></Card> will appear as Card component’s children prop.
Props are read-only snapshots in time: every render receives a new version of props.
You can’t change props. When you need interactivity, you’ll need to set state.

Try out some challenges1. Extract a component 2. Adjust the image size based on a prop 3. Passing JSX in a children prop Challenge 1 of 3: Extract a component This Gallery component contains some very similar markup for two profiles. Extract a Profile component out of it to reduce the duplication. You’ll need to choose what props to pass to it.App.jsutils.jsApp.js ResetForkimport { getImageUrl } from './utils.js';

```
export default function Gallery() {
  return (
    <div>
      <h1>Notable Scientists</h1>
      <section className="profile">
        <h2>Maria Sk &öFðwska-Curie</h2>
        <img
          className="avatar"
          src={getImageUrl('szV5sdG')}
          alt="Maria Sk &öFðwska-Curie"
          width={70}
          height={70}
        />
```

```
        <ul>
          <li>
            <b>Profession: </b>
            physicist and chemist
          </li>
          <li>
            <b>Awards: 4 </b>
            (Nobel Prize in Physics, Nobel Prize in Chemistry, Davy Medal, Matteucci
Medal)
          </li>
          <li>
            <b>Discovered: </b>
            polonium (element)
          </li>
        </ul>
      </section>
      <section className="profile">
        <h2>Katsuko Saruhashi</h2>
        <img
          className="avatar"
          src={getImageUrl('YfeOqp2')}
          alt="Katsuko Saruhashi"
          width={70}
          height={70}
        />
        <ul>
          <li>
            <b>Profession: </b>
            geochemist
          </li>
          <li>
            <b>Awards: 2 </b>
            (Miyake Prize for geochemistry, Tanaka Prize)
          </li>
          <li>
            <b>Discovered: </b>
            a method for measuring carbon dioxide in seawater
          </li>
        </ul>
      </section>
    </div>
  );
}
```

Show more Show hint Show solutionNext ChallengePreviousJavaScript in JSX with Curly BracesNextConditional Rendering

Learn ReactDescribing the UIConditional RenderingYour components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, &&, and ? : operators.
You will learn
How to return different JSX depending on a condition
How to conditionally include or exclude a piece of JSX
Common conditional syntax shortcuts you'll encounter in React codebases

Conditionally returning JSX
Let's say you have a PackingList component rendering several Items, which can be marked as packed or not:
App.jsApp.js Download ResetFork991234567891011121314151617181920212223242526function Item({ name, isPacked }) { return <li className="item">{name};}export default function PackingList() { return (<section> <h1>Sally Ride's Packing List</h1> <Item isPacked={true} name="Space suit" /> <Item isPacked={true} name="Helmet with a golden leaf" /> <Item isPacked={false} name="Photo of Tam" /> </section>);}Show more
Notice that some of the Item components have their isPacked prop set to true instead of false. You want to add a checkmark (') to packed items if isPacked={true}.
You can write this as an if/else statement like so:
if (isPacked) { return <li className="item">{name} ' ;}return <li className="item">{name};
If the isPacked prop is true, this code returns a different JSX tree. With this change, some of the items get a checkmark at the end:
App.jsApp.js Download ResetForkfunction Item({ name, isPacked }) {
```
  if (isPacked) {
    return <li className="item">{name} ' </li>;
  }
  return <li className="item">{name}</li>;
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
```

```
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Show more

Try editing what gets returned in either case, and see how the result changes!

Notice how you’re creating branching logic with JavaScript’s if and return statements. In React, control flow (like conditions) is handled by JavaScript.

Conditionally returning nothing with null

In some situations, you won’t want to render anything at all. For example, say you don’t want to show packed items at all. A component must return something. In this case, you can return null:

```
if (isPacked) {  return null;}return <li className="item">{name}</li>;
```

If isPacked is true, the component will return nothing, null. Otherwise, it will return JSX to render.

App.jsApp.js Download ResetForkfunction Item({ name, isPacked }) {

```
  if (isPacked) {
    return null;
  }
  return <li className="item">{name}</li>;
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
```

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

Show more

In practice, returning null from a component isn’t common because it might surprise a developer trying to render it. More often, you would conditionally include or exclude the component in the parent component’s JSX. Here’s how to do that!

Conditionally including JSX

In the previous example, you controlled which (if any!) JSX tree would be returned by the component. You may already have noticed some duplication in the render output:

<li className="item">{name} '

is very similar to

<li className="item">{name}

Both of the conditional branches return <li className="item">...:

if (isPacked) { return <li className="item">{name} ' ;}return <li className="item">{name};

While this duplication isn’t harmful, it could make your code harder to maintain. What if you want to change the className? You’d have to do it in two places in your code! In such a situation, you could conditionally include a little JSX to make your code more DRY.

Conditional (ternary) operator (? :)

JavaScript has a compact syntax for writing a conditional expression — the conditional operator or “ternary operator”.

Instead of this:

if (isPacked) { return <li className="item">{name} ' ;}return <li className="item">{name};

You can write this:

return (<li className="item">   {isPacked ? name + ' ' ' : name});

You can read it as “if isPacked is true, then (?) render name + ' ' ', otherwise (:) render name”.

Deep DiveAre these two examples fully equivalent? Show DetailsIf you’re coming from an object-oriented programming background, you might assume that the two examples above are subtly different because one of them may create two different “instances” of <li>. But JSX elements aren’t “instances” because they don’t hold any internal state and aren’t real DOM nodes. They’re lightweight descriptions, like blueprints. So these two examples, in fact, are completely equivalent. Preserving and Resetting State goes into detail about how this works.

Now let’s say you want to wrap the completed item’s text into another HTML tag, like <del> to strike it out. You can add even more newlines and parentheses so that it’s easier to nest more JSX in each of the cases:

App.jsApp.js Download ResetForkfunction Item({ name, isPacked }) {

```
  return (
    <li className="item">
      {isPacked ? (
        <del>
```

```
        {name + ' ' '}
      </del>
    ) : (
      name
    )}
  </li>
 );
}

export default function PackingList() {
 return (
  <section>
   <h1>Sally Ride's Packing List</h1>
   <ul>
    <Item
     isPacked={true}
     name="Space suit"
    />
    <Item
     isPacked={true}
     name="Helmet with a golden leaf"
    />
    <Item
     isPacked={false}
     name="Photo of Tam"
    />
   </ul>
  </section>
 );
}
```

Show more

This style works well for simple conditions, but use it in moderation. If your components get messy with too much nested conditional markup, consider extracting child components to clean things up. In React, markup is a part of your code, so you can use tools like variables and functions to tidy up complex expressions.

Logical AND operator (&&)

Another common shortcut you'll encounter is the JavaScript logical AND (&&) operator. Inside React components, it often comes up when you want to render some JSX when the condition is true, or render nothing otherwise. With &&, you could conditionally render the checkmark only if isPacked is true:

```
return (  <li className="item">    {name} {isPacked && '' '}  </li>);
```

You can read this as "if isPacked, then (&&) render the checkmark, otherwise, render nothing".

Here it is in action:

App.jsApp.js Download ResetForkfunction Item({ name, isPacked }) {

```
  return (
    <li className="item">
      {name} {isPacked && ' '}
    </li>
  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Show more

A JavaScript && expression returns the value of its right side (in our case, the checkmark) if the left side (our condition) is true. But if the condition is false, the whole expression becomes false. React considers false as a “hole” in the JSX tree, just like null or undefined, and doesn’t render anything in its place.

PitfallDon’t put numbers on the left side of &&.To test the condition, JavaScript converts the left side to a boolean automatically. However, if the left side is 0, then the whole expression gets that value (0), and React will happily render 0 rather than nothing.For example, a common mistake is to write code like messageCount && New messages . It’s easy to assume that it renders nothing when messageCount is 0, but it really renders the 0 itself!To fix it, make the left side a boolean: messageCount > 0 && New messages .

Conditionally assigning JSX to a variable

When the shortcuts get in the way of writing plain code, try using an if statement and a variable. You can reassign variables defined with let, so start by providing the default content you want to display, the name:

```
let itemContent = name;
```

Use an if statement to reassign a JSX expression to itemContent if isPacked is true:
if (isPacked) { itemContent = name + " ' ";}
Curly braces open the “window into JavaScript”. Embed the variable with curly braces in the returned JSX tree, nesting the previously calculated expression inside of JSX:
<li className="item"> {itemContent}
This style is the most verbose, but it’s also the most flexible. Here it is in action:
App.jsApp.js Download ResetForkfunction Item({ name, isPacked }) {
  let itemContent = name;
  if (isPacked) {
    itemContent = name + " ' ";
  }
  return (
    <li className="item">
      {itemContent}

  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>

        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />

    </section>
  );
}

Show more
Like before, this works not only for text, but for arbitrary JSX too:
App.jsApp.js Download ResetForkfunction Item({ name, isPacked }) {
  let itemContent = name;
  if (isPacked) {
    itemContent = (

```
      <del>
        {name + " ' "}
      </del>
    );
  }
  return (
    <li className="item">
      {itemContent}
    </li>
  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Show more

If you’re not familiar with JavaScript, this variety of styles might seem overwhelming at first. However, learning them will help you read and write any JavaScript code — and not just React components! Pick the one you prefer for a start, and then consult this reference again if you forget how the other ones work.

Recap

In React, you control branching logic with JavaScript.

You can return a JSX expression conditionally with an if statement.

You can conditionally save some JSX to a variable and then include it inside other JSX by using the curly braces.

In JSX, {cond ? <A /> : <B />} means “if cond, render <A />, otherwise <B />”.

In JSX, {cond && <A />} means “if cond, render <A />, otherwise nothing”.

The shortcuts are common, but you don’t have to use them if you prefer plain if.

Try out some challenges1. Show an icon for incomplete items with ? : 2. Show the item importance with && 3. Refactor a series of ? : to if and variables Challenge 1 of 3: Show an icon for incomplete items with ? : Use the conditional operator (cond ? a : b) to render a 'L if isPacked isn’t true.App.jsApp.js Download ResetForkfunction Item({ name, isPacked }) {
  return (
    <li className="item">
      {name} {isPacked && " '}

  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>

        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />

    </section>
  );
}

Show more Show solutionNext ChallengePreviousPassing Props to a ComponentNextRendering Lists
Learn ReactDescribing the UIRendering ListsYou will often want to display multiple similar components from a collection of data. You can use the JavaScript array methods to manipulate an array of data. On this page, you’ll use filter() and map() with React to filter and transform your array of data into an array of components.
You will learn
How to render components from an array using JavaScript’s map()
How to render only specific components using JavaScript’s filter()
When and why to use React keys

Rendering data from arrays
Say that you have a list of content.
 Creola Katherine Johnson: mathematician Mario José Molina-Pasquel Henríquez: chemist Mohammad Abdus Salam: physicist Percy Lavon Julian: chemist Subrahmanyan Chandrasekhar: astrophysicist
The only difference among those list items is their contents, their data. You will often need to show several instances of the same component using different data when building interfaces: from lists of comments to galleries of profile images. In these situations, you can store that data in JavaScript objects and arrays and use methods like map() and filter() to render lists of components from them.
Here's a short example of how to generate a list of items from an array:

Move the data into an array:

```
const people = [  'Creola Katherine Johnson: mathematician',  'Mario José Molina-Pasquel Henríquez: chemist',  'Mohammad Abdus Salam: physicist',  'Percy Lavon Julian: chemist',  'Subrahmanyan Chandrasekhar: astrophysicist'];
```

Map the people members into a new array of JSX nodes, listItems:

```
const listItems = people.map(person => <li>{person}</li>);
```

Return listItems from your component wrapped in a <ul>:

```
return <ul>{listItems}</ul>;
```

Here is the result:
App.jsApp.js Download ResetForkconst people = [

```
  'Creola Katherine Johnson: mathematician',
  'Mario José Molina-Pasquel Henríquez: chemist',
  'Mohammad Abdus Salam: physicist',
  'Percy Lavon Julian: chemist',
  'Subrahmanyan Chandrasekhar: astrophysicist'
];

export default function List() {
  const listItems = people.map(person =>
    <li>{person}</li>
  );
  return <ul>{listItems}</ul>;
}
```

Notice the sandbox above displays a console error:
ConsoleWarning: Each child in a list should have a unique "key" prop.

You'll learn how to fix this error later on this page. Before we get to that, let's add some structure to your data.

Filtering arrays of items

This data can be structured even more.

```
const people = [{  id: 0,  name: 'Creola Katherine Johnson',  profession: 'mathematician',}, {  id: 1,  name: 'Mario José Molina-Pasquel Henríquez',  profession: 'chemist',}, {  id: 2,  name: 'Mohammad Abdus Salam',  profession: 'physicist',}, {  name: 'Percy Lavon Julian',  profession: 'chemist',  }, {  name: 'Subrahmanyan Chandrasekhar',  profession: 'astrophysicist',}];
```

Let's say you want a way to only show people whose profession is 'chemist'. You can use JavaScript's filter() method to return just those people. This method takes an array of items, passes them through a "test" (a function that returns true or false), and returns a new array of only those items that passed the test (returned true).

You only want the items where profession is 'chemist'. The "test" function for this looks like (person) => person.profession === 'chemist'. Here's how to put it together:

Create a new array of just "chemist" people, chemists, by calling filter() on the people filtering by person.profession === 'chemist':

```
const chemists = people.filter(person =>  person.profession === 'chemist');
```

Now map over chemists:

```
const listItems = chemists.map(person =>  <li>     <img src={getImageUrl(person)}       alt={person.name}     />     <p>       <b>{person.name}:</b>       {' ' + person.profession + ' '}       known for {person.accomplishment}     </p>  </li>);
```

Lastly, return the listItems from your component:

```
return <ul>{listItems}</ul>;
```

App.jsdata.jsutils.jsApp.js ResetForkimport { people } from './data.js';
import { getImageUrl } from './utils.js';

```
export default function List() {
  const chemists = people.filter(person =>
    person.profession === 'chemist'
  );
  const listItems = chemists.map(person =>
    <li>
      <img
        src={getImageUrl(person)}
        alt={person.name}
      />
      <p>
        <b>{person.name}:</b>
```

```
        {' ' + person.profession + ' '}
        known for {person.accomplishment}
      </p>
    </li>
  );
  return <ul>{listItems}</ul>;
}
```

Show more

PitfallArrow functions implicitly return the expression right after =>, so you didn’t need a return statement:const listItems = chemists.map(person => ... // Implicit return!);However, you must write return explicitly if your => is followed by a { curly brace! const listItems = chemists.map(person => { // Curly brace return <li>...;});Arrow functions containing => { are said to have a “block body”. They let you write more than a single line of code, but you have to write a return statement yourself. If you forget it, nothing gets returned!

Keeping list items in order with key

Notice that all the sandboxes above show an error in the console:

ConsoleWarning: Each child in a list should have a unique “key” prop.

You need to give each array item a key — a string or a number that uniquely identifies it among other items in that array:

<li key={person.id}>...

NoteJSX elements directly inside a map() call always need keys!

Keys tell React which array item each component corresponds to, so that it can match them up later. This becomes important if your array items can move (e.g. due to sorting), get inserted, or get deleted. A well-chosen key helps React infer what exactly has happened, and make the correct updates to the DOM tree.

Rather than generating keys on the fly, you should include them in your data:

App.jsdata.jsutils.jsdata.js ResetForkexport const people = [{
  id: 0, // Used in JSX as a key
  name: 'Creola Katherine Johnson',
  profession: 'mathematician',
  accomplishment: 'spaceflight calculations',
  imageId: 'MK3eW3A'
}, {
  id: 1, // Used in JSX as a key
  name: 'Mario José Molina-Pasquel Henríquez',
  profession: 'chemist',
  accomplishment: 'discovery of Arctic ozone hole',
  imageId: 'mynHUSa'
}, {
  id: 2, // Used in JSX as a key
  name: 'Mohammad Abdus Salam',
  profession: 'physicist',
  accomplishment: 'electromagnetism theory',
  imageId: 'bE7W1ji'
```
```

```
}, {
  id: 3, // Used in JSX as a key
  name: 'Percy Lavon Julian',
  profession: 'chemist',
  accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
  imageId: 'IOjWm71'
}, {
  id: 4, // Used in JSX as a key
  name: 'Subrahmanyan Chandrasekhar',
  profession: 'astrophysicist',
  accomplishment: 'white dwarf star mass calculations',
  imageId: 'lrWQx8l'
}];
```

Show more
Deep DiveDisplaying several DOM nodes for each list item Show DetailsWhat do you do when each item needs to render not one, but several DOM nodes?The short <>...</> Fragment syntax won't let you pass a key, so you need to either group them into a single , or use the slightly longer and more explicit <Fragment> syntax:import { Fragment } from 'react';// ...const listItems = people.map(person => <Fragment key={person.id}> <h1>{person.name}</h1> {person.bio} </Fragment>);Fragments disappear from the DOM, so this will produce a flat list of <h1>, , <h1>, , and so on.
Where to get your key
Different sources of data provide different sources of keys:

Data from a database: If your data is coming from a database, you can use the database keys/IDs, which are unique by nature.
Locally generated data: If your data is generated and persisted locally (e.g. notes in a note-taking app), use an incrementing counter, crypto.randomUUID() or a package like uuid when creating items.

Rules of keys

Keys must be unique among siblings. However, it's okay to use the same keys for JSX nodes in different arrays.
Keys must not change or that defeats their purpose! Don't generate them while rendering.

Why does React need keys?
Imagine that files on your desktop didn't have names. Instead, you'd refer to them by their order — the first file, the second file, and so on. You could get used to it, but once you delete a file, it would get confusing. The second file would become the first file, the third file would be the second file, and so on.
File names in a folder and JSX keys in an array serve a similar purpose. They let us uniquely identify an item between its siblings. A well-chosen key provides more

information than the position within the array. Even if the position changes due to reordering, the key lets React identify the item throughout its lifetime.
PitfallYou might be tempted to use an item’s index in the array as its key. In fact, that’s what React will use if you don’t specify a key at all. But the order in which you render items will change over time if an item is inserted, deleted, or if the array gets reordered. Index as a key often leads to subtle and confusing bugs.Similarly, do not generate keys on the fly, e.g. with key={Math.random()}. This will cause keys to never match up between renders, leading to all your components and DOM being recreated every time. Not only is this slow, but it will also lose any user input inside the list items. Instead, use a stable ID based on the data.Note that your components won’t receive key as a prop. It’s only used as a hint by React itself. If your component needs an ID, you have to pass it as a separate prop: <Profile key={id} userId={id} />.
RecapOn this page you learned:
How to move data out of components and into data structures like arrays and objects.
How to generate sets of similar components with JavaScript’s map().
How to create arrays of filtered items with JavaScript’s filter().
Why and how to set key on each component in a collection so React can keep track of each of them even if their position or data changes.

Try out some challenges1. Splitting a list in two 2. Nested lists in one component 3. Extracting a list item component 4. List with a separator Challenge 1 of 4: Splitting a list in two This example shows a list of all people.Change it to show two separate lists one after another: Chemists and Everyone Else. Like previously, you can determine whether a person is a chemist by checking if person.profession === 'chemist'.App.jsdata.jsutils.jsApp.js ResetForkimport { people } from './data.js';
import { getImageUrl } from './utils.js';

```
export default function List() {
  const listItems = people.map(person =>
    <li key={person.id}>
      <img
        src={getImageUrl(person)}
        alt={person.name}
      />
      <p>
        <b>{person.name}:</b>
        {' ' + person.profession + ' '}
        known for {person.accomplishment}
      </p>
    </li>
  );
  return (
    <article>
      <h1>Scientists</h1>
      <ul>{listItems}</ul>
    </article>
```

```
  );
}
```

Show more Show solutionNext ChallengePreviousConditional RenderingNextKeeping Components Pure
Learn ReactDescribing the UIKeeping Components PureSome JavaScript functions are pure. Pure functions only perform a calculation and nothing more. By strictly only writing your components as pure functions, you can avoid an entire class of baffling bugs and unpredictable behavior as your codebase grows. To get these benefits, though, there are a few rules you must follow.
You will learn
What purity is and how it helps you avoid bugs
How to keep components pure by keeping changes out of the render phase
How to use Strict Mode to find mistakes in your components

Purity: Components as formulas
In computer science (and especially the world of functional programming), a pure function is a function with the following characteristics:

It minds its own business. It does not change any objects or variables that existed before it was called.
Same inputs, same output. Given the same inputs, a pure function should always return the same result.

You might already be familiar with one example of pure functions: formulas in math.
Consider this math formula: y = 2x.
If x = 2 then y = 4. Always.
If x = 3 then y = 6. Always.
If x = 3, y won’t sometimes be 9 or –1 or 2.5 depending on the time of day or the state of the stock market.
If y = 2x and x = 3, y will always be 6.
If we made this into a JavaScript function, it would look like this:
`function double(number) {  return 2 * number;}`
In the above example, double is a pure function. If you pass it 3, it will return 6. Always.
React is designed around this concept. React assumes that every component you write is a pure function. This means that React components you write must always return the same JSX given the same inputs:
App.jsApp.js Download ResetForkfunction Recipe({ drinkers }) {

```
  return (
    <ol>
      <li>Boil {drinkers} cups of water.</li>
      <li>Add {drinkers} spoons of tea and {0.5 * drinkers} spoons of spice.</li>
      <li>Add {0.5 * drinkers} cups of milk to boil and sugar to taste.</li>
    </ol>
  );
}
```

```
export default function App() {
  return (
    <section>
      <h1>Spiced Chai Recipe</h1>
      <h2>For two</h2>
      <Recipe drinkers={2} />
      <h2>For a gathering</h2>
      <Recipe drinkers={4} />
    </section>
  );
}
```

Show more

When you pass drinkers={2} to Recipe, it will return JSX containing 2 cups of water. Always.

If you pass drinkers={4}, it will return JSX containing 4 cups of water. Always.

Just like a math formula.

You could think of your components as recipes: if you follow them and don't introduce new ingredients during the cooking process, you will get the same dish every time. That "dish" is the JSX that the component serves to React to render.

Illustrated by Rachel Lee Nabors

Side Effects: (un)intended consequences

React's rendering process must always be pure. Components should only return their JSX, and not change any objects or variables that existed before rendering—that would make them impure!

Here is a component that breaks this rule:

App.jsApp.js Download ResetForklet guest = 0;

```
function Cup() {
  // Bad: changing a preexisting variable!
  guest = guest + 1;
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup />
      <Cup />
      <Cup />
    </>
  );
}
```

Show more

This component is reading and writing a guest variable declared outside of it. This means that calling this component multiple times will produce different JSX! And what's more, if other components read guest, they will produce different JSX, too, depending on when they were rendered! That's not predictable.
Going back to our formula y = 2x, now even if x = 2, we cannot trust that y = 4. Our tests could fail, our users would be baffled, planes would fall out of the sky—you can see how this would lead to confusing bugs!
You can fix this component by passing guest as a prop instead:
App.jsApp.js Download ResetForkfunction Cup({ guest }) {
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup guest={1} />
      <Cup guest={2} />
      <Cup guest={3} />
    </>
  );
}

Now your component is pure, as the JSX it returns only depends on the guest prop.
In general, you should not expect your components to be rendered in any particular order. It doesn't matter if you call y = 2x before or after y = 5x: both formulas will resolve independently of each other. In the same way, each component should only "think for itself", and not attempt to coordinate with or depend upon others during rendering. Rendering is like a school exam: each component should calculate JSX on their own!
Deep DiveDetecting impure calculations with StrictMode Show DetailsAlthough you might not have used them all yet, in React there are three kinds of inputs that you can read while rendering: props, state, and context. You should always treat these inputs as read-only.When you want to change something in response to user input, you should set state instead of writing to a variable. You should never change preexisting variables or objects while your component is rendering.React offers a "Strict Mode" in which it calls each component's function twice during development. By calling the component functions twice, Strict Mode helps find components that break these rules.Notice how the original example displayed "Guest #2", "Guest #4", and "Guest #6" instead of "Guest #1", "Guest #2", and "Guest #3". The original function was impure, so calling it twice broke it. But the fixed pure version works even if the function is called twice every time. Pure functions only calculate, so calling them twice won't change anything—just like calling double(2) twice doesn't change what's returned, and solving y = 2x twice doesn't change what y is. Same inputs, same outputs. Always.Strict Mode has no effect in production, so it won't slow down the app for your users. To opt into Strict Mode, you can wrap your root component into <React.StrictMode>. Some frameworks do this by default.

Local mutation: Your component's little secret
In the above example, the problem was that the component changed a preexisting variable while rendering. This is often called a "mutation" to make it sound a bit scarier. Pure functions don't mutate variables outside of the function's scope or objects that were created before the call—that makes them impure!
However, it's completely fine to change variables and objects that you've just created while rendering. In this example, you create an [] array, assign it to a cups variable, and then push a dozen cups into it:
App.jsApp.js Download ResetForkfunction Cup({ guest }) {
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaGathering() {
  let cups = [];
  for (let i = 1; i <= 12; i++) {
    cups.push(<Cup key={i} guest={i} />);
  }
  return cups;
}

If the cups variable or the [] array were created outside the TeaGathering function, this would be a huge problem! You would be changing a preexisting object by pushing items into that array.
However, it's fine because you've created them during the same render, inside TeaGathering. No code outside of TeaGathering will ever know that this happened. This is called "local mutation"—it's like your component's little secret.
Where you can cause side effects
While functional programming relies heavily on purity, at some point, somewhere, something has to change. That's kind of the point of programming! These changes—updating the screen, starting an animation, changing the data—are called side effects. They're things that happen "on the side", not during rendering.
In React, side effects usually belong inside event handlers. Event handlers are functions that React runs when you perform some action—for example, when you click a button. Even though event handlers are defined inside your component, they don't run during rendering! So event handlers don't need to be pure.
If you've exhausted all other options and can't find the right event handler for your side effect, you can still attach it to your returned JSX with a useEffect call in your component. This tells React to execute it later, after rendering, when side effects are allowed. However, this approach should be your last resort.
When possible, try to express your logic with rendering alone. You'll be surprised how far this can take you!
Deep DiveWhy does React care about purity? Show DetailsWriting pure functions takes some habit and discipline. But it also unlocks marvelous opportunities:
Your components could run in a different environment—for example, on the server! Since they return the same result for the same inputs, one component can serve many

user requests.
You can improve performance by skipping rendering components whose inputs have not changed. This is safe because pure functions always return the same results, so they are safe to cache.
If some data changes in the middle of rendering a deep component tree, React can restart rendering without wasting time to finish the outdated render. Purity makes it safe to stop calculating at any time.
Every new React feature we're building takes advantage of purity. From data fetching to animations to performance, keeping components pure unlocks the power of the React paradigm.
Recap
A component must be pure, meaning:

It minds its own business. It should not change any objects or variables that existed before rendering.
Same inputs, same output. Given the same inputs, a component should always return the same JSX.

Rendering can happen at any time, so components should not depend on each others' rendering sequence.
You should not mutate any of the inputs that your components use for rendering. That includes props, state, and context. To update the screen, "set" state instead of mutating preexisting objects.
Strive to express your component's logic in the JSX you return. When you need to "change things", you'll usually want to do it in an event handler. As a last resort, you can useEffect.
Writing pure functions takes a bit of practice, but it unlocks the power of React's paradigm.

Try out some challenges1. Fix a broken clock 2. Fix a broken profile 3. Fix a broken story tray Challenge 1 of 3: Fix a broken clock This component tries to set the <h1>'s CSS class to "night" during the time from midnight to six hours in the morning, and "day" at all other times. However, it doesn't work. Can you fix this component?You can verify whether your solution works by temporarily changing the computer's timezone. When the current time is between midnight and six in the morning, the clock should have inverted colors!Clock.jsClock.js ResetForkexport default function Clock({ time }) {
```
  let hours = time.getHours();
  if (hours >= 0 && hours <= 6) {
    document.getElementById('time').className = 'night';
  } else {
    document.getElementById('time').className = 'day';
  }
  return (
    <h1 id="time">
      {time.toLocaleTimeString()}
```

```
    </h1>
  );
}
```

Show hint Show solutionNext ChallengePreviousRendering ListsNextAdding Interactivity

Learn ReactAdding InteractivitySome things on the screen update in response to user input. For example, clicking an image gallery switches the active image. In React, data that changes over time is called state. You can add state to any component, and update it as needed. In this chapter, you'll learn how to write components that handle interactions, update their state, and display different output over time.

In this chapter

How to handle user-initiated events
How to make components "remember" information with state
How React updates the UI in two phases
Why state doesn't update right after you change it
How to queue multiple state updates
How to update an object in state
How to update an array in state

Responding to events

React lets you add event handlers to your JSX. Event handlers are your own functions that will be triggered in response to user interactions like clicking, hovering, focusing on form inputs, and so on.

Built-in components like <button> only support built-in browser events like onClick. However, you can also create your own components, and give their event handler props any application-specific names that you like.

App.jsApp.js Download ResetFork991234567891011121314151617181920212223242526272829 30export default function App() { return (<Toolbar onPlayMovie={() => alert('Playing!')} onUploadImage={() => alert('Uploading!')} />);}function Toolbar({ onPlayMovie, onUploadImage }) { return (<Button onClick={onPlayMovie}> Play Movie </Button> <Button onClick={onUploadImage}> Upload Image </ Button>);}function Button({ onClick, children }) { return (<button onClick={onClick}> {children} </button>);}Show more

Ready to learn this topic?Read Responding to Events to learn how to add event handlers.Read More

State: a component's memory

Components often need to change what's on the screen as a result of an interaction. Typing into the form should update the input field, clicking "next" on an image carousel should change which image is displayed, clicking "buy" puts a product in the shopping cart. Components need to "remember" things: the current input value, the current image, the shopping cart. In React, this kind of component-specific memory is called state.

You can add state to a component with a useState Hook. Hooks are special functions that let your components use React features (state is one of those features). The

useState Hook lets you declare a state variable. It takes the initial state and returns a pair of values: the current state, and a state setter function that lets you update it.
const [index, setIndex] = useState(0);const [showMore, setShowMore] = useState(false);
Here is how an image gallery uses and updates state on click:
App.jsdata.jsApp.js ResetForkimport { useState } from 'react';
import { sculptureList } from './data.js';

```
export default function Gallery() {
  const [index, setIndex] = useState(0);
  const [showMore, setShowMore] = useState(false);
  const hasNext = index < sculptureList.length - 1;

  function handleNextClick() {
    if (hasNext) {
      setIndex(index + 1);
    } else {
      setIndex(0);
    }
  }

  function handleMoreClick() {
    setShowMore(!showMore);
  }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleNextClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
      </h2>
      <h3>
        ({index + 1} of {sculptureList.length})
      </h3>
      <button onClick={handleMoreClick}>
        {showMore ? 'Hide' : 'Show'} details
      </button>
      {showMore && <p>{sculpture.description}</p>}
      <img
        src={sculpture.url}
        alt={sculpture.alt}
      />
```

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

Show more
Ready to learn this topic?Read State: A Component’s Memory to learn how to remember a value and update it on interaction.Read More
Render and commit
Before your components are displayed on the screen, they must be rendered by React. Understanding the steps in this process will help you think about how your code executes and explain its behavior.
Imagine that your components are cooks in the kitchen, assembling tasty dishes from ingredients. In this scenario, React is the waiter who puts in requests from customers and brings them their orders. This process of requesting and serving UI has three steps:

Triggering a render (delivering the diner’s order to the kitchen)
Rendering the component (preparing the order in the kitchen)
Committing to the DOM (placing the order on the table)

TriggerRenderCommitIllustrated by Rachel Lee Nabors
Ready to learn this topic?Read Render and Commit to learn the lifecycle of a UI update.Read More
State as a snapshot
Unlike regular JavaScript variables, React state behaves more like a snapshot. Setting it does not change the state variable you already have, but instead triggers a re-render. This can be surprising at first!
console.log(count); // 0setCount(count + 1); // Request a re-render with 1console.log(count); // Still 0!
This behavior help you avoid subtle bugs. Here is a little chat app. Try to guess what happens if you press “Send” first and then change the recipient to Bob. Whose name will appear in the alert five seconds later?
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [to, setTo] = useState('Alice');
  const [message, setMessage] = useState('Hello');

  function handleSubmit(e) {
    e.preventDefault();
    setTimeout(() => {
      alert(`You said ${message} to ${to}`);
    }, 5000);
  }

  return (
    <form onSubmit={handleSubmit}>
```

```
      <label>
        To:{' '}
        <select
          value={to}
          onChange={e => setTo(e.target.value)}>
          <option value="Alice">Alice</option>
          <option value="Bob">Bob</option>
        </select>
      </label>
      <textarea
        placeholder="Message"
        value={message}
        onChange={e => setMessage(e.target.value)}
      />
      <button type="submit">Send</button>
    </form>
  );
}
```

Show more

Ready to learn this topic?Read State as a Snapshot to learn why state appears “fixed” and unchanging inside the event handlers.Read More

Queueing a series of state updates

This component is buggy: clicking “+3” increments the score only once.

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [score, setScore] = useState(0);

  function increment() {
    setScore(score + 1);
  }

  return (
    <>
      <button onClick={() => increment()}>+1</button>
      <button onClick={() => {
        increment();
        increment();
        increment();
      }}>+3</button>
      <h1>Score: {score}</h1>
    </>
  )
}
```

Show more
State as a Snapshot explains why this is happening. Setting state requests a new re-render, but does not change it in the already running code. So score continues to be 0 right after you call setScore(score + 1).
console.log(score); // 0setScore(score + 1); // setScore(0 + 1);console.log(score); // 0setScore(score + 1); // setScore(0 + 1);console.log(score); // 0setScore(score + 1); // setScore(0 + 1);console.log(score); // 0
You can fix this by passing an updater function when setting state. Notice how replacing setScore(score + 1) with setScore(s => s + 1) fixes the “+3” button. This lets you queue multiple state updates.
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [score, setScore] = useState(0);

  function increment() {
    setScore(s => s + 1);
  }

  return (
    <>
      <button onClick={() => increment()}>+1</button>
      <button onClick={() => {
        increment();
        increment();
        increment();
      }}>+3</button>
      <h1>Score: {score}</h1>
    </>
  )
}
```

Show more
Ready to learn this topic?Read Queueing a Series of State Updates to learn how to queue a sequence of state updates.Read More
Updating objects in state
State can hold any kind of JavaScript value, including objects. But you shouldn’t change objects and arrays that you hold in the React state directly. Instead, when you want to update an object and array, you need to create a new one (or make a copy of an existing one), and then update the state to use that copy.
Usually, you will use the ... spread syntax to copy objects and arrays that you want to change. For example, updating a nested object could look like this:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [person, setPerson] = useState({
```

```
  name: 'Niki de Saint Phalle',
  artwork: {
    title: 'Blue Nana',
    city: 'Hamburg',
    image: 'https://i.imgur.com/Sd1AgUOm.jpg',
  }
});

function handleNameChange(e) {
  setPerson({
    ...person,
    name: e.target.value
  });
}

function handleTitleChange(e) {
  setPerson({
    ...person,
    artwork: {
      ...person.artwork,
      title: e.target.value
    }
  });
}

function handleCityChange(e) {
  setPerson({
    ...person,
    artwork: {
      ...person.artwork,
      city: e.target.value
    }
  });
}

function handleImageChange(e) {
  setPerson({
    ...person,
    artwork: {
      ...person.artwork,
      image: e.target.value
    }
  });
}

return (
```

```
    <>
      <label>
        Name:
        <input
          value={person.name}
          onChange={handleNameChange}
        />
      </label>
      <label>
        Title:
        <input
          value={person.artwork.title}
          onChange={handleTitleChange}
        />
      </label>
      <label>
        City:
        <input
          value={person.artwork.city}
          onChange={handleCityChange}
        />
      </label>
      <label>
        Image:
        <input
          value={person.artwork.image}
          onChange={handleImageChange}
        />
      </label>
      <p>
        <i>{person.artwork.title}</i>
        {' by '}
        {person.name}
        <br />
        (located in {person.artwork.city})
      </p>
      <img
        src={person.artwork.image}
        alt={person.artwork.title}
      />
    </>
  );
}
```

Show more

If copying objects in code gets tedious, you can use a library like Immer to reduce

repetitive code:
App.jspackage.jsonApp.js ResetForkimport { useImmer } from 'use-immer';

```
export default function Form() {
  const [person, updatePerson] = useImmer({
    name: 'Niki de Saint Phalle',
    artwork: {
      title: 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgUOm.jpg',
    }
  });

  function handleNameChange(e) {
    updatePerson(draft => {
      draft.name = e.target.value;
    });
  }

  function handleTitleChange(e) {
    updatePerson(draft => {
      draft.artwork.title = e.target.value;
    });
  }

  function handleCityChange(e) {
    updatePerson(draft => {
      draft.artwork.city = e.target.value;
    });
  }

  function handleImageChange(e) {
    updatePerson(draft => {
      draft.artwork.image = e.target.value;
    });
  }

  return (
    <>
      <label>
        Name:
        <input
          value={person.name}
          onChange={handleNameChange}
        />
      </label>
```

```
      <label>
        Title:
        <input
          value={person.artwork.title}
          onChange={handleTitleChange}
        />
      </label>
      <label>
        City:
        <input
          value={person.artwork.city}
          onChange={handleCityChange}
        />
      </label>
      <label>
        Image:
        <input
          value={person.artwork.image}
          onChange={handleImageChange}
        />
      </label>
      <p>
        <i>{person.artwork.title}</i>
        {' by '}
        {person.name}
        <br />
        (located in {person.artwork.city})
      </p>
      <img
        src={person.artwork.image}
        alt={person.artwork.title}
      />
    </>
  );
}
```

Show more

Ready to learn this topic?Read Updating Objects in State to learn how to update objects correctly.Read More

Updating arrays in state

Arrays are another type of mutable JavaScript objects you can store in state and should treat as read-only. Just like with objects, when you want to update an array stored in state, you need to create a new one (or make a copy of an existing one), and then set state to use the new array:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let nextId = 3;
const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
  const [list, setList] = useState(
    initialList
  );

  function handleToggle(artworkId, nextSeen) {
    setList(list.map(artwork => {
      if (artwork.id === artworkId) {
        return { ...artwork, seen: nextSeen };
      } else {
        return artwork;
      }
    }));
  }

  return (
    <>
      <h1>Art Bucket List</h1>
      <h2>My list of art to see:</h2>
      <ItemList
        artworks={list}
        onToggle={handleToggle} />
    </>
  );
}

function ItemList({ artworks, onToggle }) {
  return (
    <ul>
      {artworks.map(artwork => (
        <li key={artwork.id}>
          <label>
            <input
              type="checkbox"
              checked={artwork.seen}
              onChange={e => {
                onToggle(
                  artwork.id,
                  e.target.checked
```

```
              );
            }}
          />
          {artwork.title}
        </label>
      </li>
    ))}
  </ul>
 );
}
```

Show more

If copying arrays in code gets tedious, you can use a library like Immer to reduce repetitive code:

App.jspackage.jsonApp.js ResetForkimport { useState } from 'react';
import { useImmer } from 'use-immer';

```
let nextId = 3;
const initialList = [
 { id: 0, title: 'Big Bellies', seen: false },
 { id: 1, title: 'Lunar Landscape', seen: false },
 { id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
 const [list, updateList] = useImmer(initialList);

 function handleToggle(artworkId, nextSeen) {
  updateList(draft => {
   const artwork = draft.find(a =>
    a.id === artworkId
   );
   artwork.seen = nextSeen;
  });
 }

 return (
  <>
   <h1>Art Bucket List</h1>
   <h2>My list of art to see:</h2>
   <ItemList
    artworks={list}
    onToggle={handleToggle} />
  </>
 );
}
```

```
function ItemList({ artworks, onToggle }) {
  return (
    <ul>
      {artworks.map(artwork => (
        <li key={artwork.id}>
          <label>
            <input
              type="checkbox"
              checked={artwork.seen}
              onChange={e => {
                onToggle(
                  artwork.id,
                  e.target.checked
                );
              }}
            />
            {artwork.title}
          </label>
        </li>
      ))}
    </ul>
  );
}
```

Show more
Ready to learn this topic?Read Updating Arrays in State to learn how to update arrays correctly.Read More
What’s next?
Head over to Responding to Events to start reading this chapter page by page!
Or, if you’re already familiar with these topics, why not read about Managing State?
PreviousKeeping Components PureNextResponding to Events
Learn ReactAdding InteractivityResponding to EventsReact lets you add event handlers to your JSX. Event handlers are your own functions that will be triggered in response to interactions like clicking, hovering, focusing form inputs, and so on.
You will learn
Different ways to write an event handler
How to pass event handling logic from a parent component
How events propagate and how to stop them

Adding event handlers
To add an event handler, you will first define a function and then pass it as a prop to the appropriate JSX tag. For example, here is a button that doesn’t do anything yet:
App.jsApp.js Download ResetFork912345678export default function Button() { return (<button>   I don't do anything   </button>);}
You can make it show a message when a user clicks by following these three steps:

Declare a function called handleClick inside your Button component.
Implement the logic inside that function (use alert to show the message).
Add onClick={handleClick} to the <button> JSX.

App.jsApp.js Download ResetForkexport default function Button() {
  function handleClick() {
    alert('You clicked me!');
  }

  return (
    <button onClick={handleClick}>
      Click me
    </button>
  );
}

You defined the handleClick function and then passed it as a prop to <button>.
handleClick is an event handler. Event handler functions:

Are usually defined inside your components.
Have names that start with handle, followed by the name of the event.

By convention, it is common to name event handlers as handle followed by the event name. You'll often see onClick={handleClick}, onMouseEnter={handleMouseEnter}, and so on.
Alternatively, you can define an event handler inline in the JSX:
<button onClick={function handleClick() { alert('You clicked me!');}}>
Or, more concisely, using an arrow function:
<button onClick={() => { alert('You clicked me!');}}>
All of these styles are equivalent. Inline event handlers are convenient for short functions.
PitfallFunctions passed to event handlers must be passed, not called. For example:passing a function (correct)calling a function (incorrect)<button onClick={handleClick}><button onClick={handleClick()}>The difference is subtle. In the first example, the handleClick function is passed as an onClick event handler. This tells React to remember it and only call your function when the user clicks the button.In the second example, the () at the end of handleClick() fires the function immediately during rendering, without any clicks. This is because JavaScript inside the JSX { and } executes right away.When you write code inline, the same pitfall presents itself in a different way:passing a function (correct)calling a function (incorrect)<button onClick={() => alert('...')}><button onClick={alert('...')}>Passing inline code like this won't fire on click—it fires every time the component renders:// This alert fires when the component renders, not when clicked!<button onClick={alert('You clicked me!')}>If you want to define your event handler inline, wrap it in an anonymous function like so:<button

onClick={() => alert('You clicked me!')}>Rather than executing the code inside with every render, this creates a function to be called later.In both cases, what you want to pass is a function:
<button onClick={handleClick}> passes the handleClick function.
<button onClick={() => alert('...')}> passes the () => alert('...') function.
Read more about arrow functions.
Reading props in event handlers
Because event handlers are declared inside of a component, they have access to the component's props. Here is a button that, when clicked, shows an alert with its message prop:
App.jsApp.js Download ResetForkfunction AlertButton({ message, children }) {
  return (
    <button onClick={() => alert(message)}>
      {children}
    </button>
  );
}

export default function Toolbar() {
  return (

      <AlertButton message="Playing!">
        Play Movie
      </AlertButton>
      <AlertButton message="Uploading!">
        Upload Image
      </AlertButton>

  );
}

Show more
This lets these two buttons show different messages. Try changing the messages passed to them.
Passing event handlers as props
Often you'll want the parent component to specify a child's event handler. Consider buttons: depending on where you're using a Button component, you might want to execute a different function—perhaps one plays a movie and another uploads an image. To do this, pass a prop the component receives from its parent as the event handler like so:
App.jsApp.js Download ResetForkfunction Button({ onClick, children }) {
  return (
    <button onClick={onClick}>
      {children}
    </button>
  );

```
}

function PlayButton({ movieName }) {
  function handlePlayClick() {
    alert(`Playing ${movieName}!`);
  }

  return (
    <Button onClick={handlePlayClick}>
      Play "{movieName}"
    </Button>
  );
}

function UploadButton() {
  return (
    <Button onClick={() => alert('Uploading!')}>
      Upload Image
    </Button>
  );
}

export default function Toolbar() {
  return (
    <div>
      <PlayButton movieName="Kiki's Delivery Service" />
      <UploadButton />
    </div>
  );
}
```

Show more

Here, the Toolbar component renders a PlayButton and an UploadButton:

PlayButton passes handlePlayClick as the onClick prop to the Button inside.
UploadButton passes () => alert('Uploading!') as the onClick prop to the Button inside.

Finally, your Button component accepts a prop called onClick. It passes that prop directly to the built-in browser <button> with onClick={onClick}. This tells React to call the passed function on click.

If you use a design system, it's common for components like buttons to contain styling but not specify behavior. Instead, components like PlayButton and UploadButton will pass event handlers down.

Naming event handler props

Built-in components like <button> and only support browser event names like onClick. However, when you're building your own components, you can name their

event handler props any way that you like.
By convention, event handler props should start with on, followed by a capital letter.
For example, the Button component's onClick prop could have been called onSmash:
App.jsApp.js Download ResetForkfunction Button({ onSmash, children }) {
  return (
    <button onClick={onSmash}>
      {children}
    </button>
  );
}

export default function App() {
  return (

      <Button onSmash={() => alert('Playing!')}>
        Play Movie
      </Button>
      <Button onSmash={() => alert('Uploading!')}>
        Upload Image
      </Button>

  );
}

Show more
In this example, <button onClick={onSmash}> shows that the browser <button> (lowercase) still needs a prop called onClick, but the prop name received by your custom Button component is up to you!
When your component supports multiple interactions, you might name event handler props for app-specific concepts. For example, this Toolbar component receives onPlayMovie and onUploadImage event handlers:
App.jsApp.js Download ResetForkexport default function App() {
  return (
    <Toolbar
      onPlayMovie={() => alert('Playing!')}
      onUploadImage={() => alert('Uploading!')}
    />
  );
}

function Toolbar({ onPlayMovie, onUploadImage }) {
  return (

      <Button onClick={onPlayMovie}>
        Play Movie
      </Button>

```
      <Button onClick={onUploadImage}>
        Upload Image
      </Button>
    </div>
  );
}

function Button({ onClick, children }) {
  return (
    <button onClick={onClick}>
      {children}
    </button>
  );
}
```

Show more

Notice how the App component does not need to know what Toolbar will do with onPlayMovie or onUploadImage. That's an implementation detail of the Toolbar. Here, Toolbar passes them down as onClick handlers to its Buttons, but it could later also trigger them on a keyboard shortcut. Naming props after app-specific interactions like onPlayMovie gives you the flexibility to change how they're used later.

NoteMake sure that you use the appropriate HTML tags for your event handlers. For example, to handle clicks, use <button onClick={handleClick}> instead of . Using a real browser <button> enables built-in browser behaviors like keyboard navigation. If you don't like the default browser styling of a button and want to make it look more like a link or a different UI element, you can achieve it with CSS. Learn more about writing accessible markup.

Event propagation

Event handlers will also catch events from any children your component might have. We say that an event "bubbles" or "propagates" up the tree: it starts with where the event happened, and then goes up the tree.

This contains two buttons. Both the and each button have their own onClick handlers. Which handlers do you think will fire when you click a button?

App.jsApp.js Download ResetForkexport default function Toolbar() {

```
  return (
    <div className="Toolbar" onClick={() => {
      alert('You clicked on the toolbar!');
    }}>
      <button onClick={() => alert('Playing!')}>
        Play Movie
      </button>
      <button onClick={() => alert('Uploading!')}>
        Upload Image
      </button>
    </div>
  );
```

```
}
```

If you click on either button, its onClick will run first, followed by the parent ’s onClick. So two messages will appear. If you click the toolbar itself, only the parent ’s onClick will run.
PitfallAll events propagate in React except onScroll, which only works on the JSX tag you attach it to.
Stopping propagation
Event handlers receive an event object as their only argument. By convention, it’s usually called e, which stands for “event”. You can use this object to read information about the event.
That event object also lets you stop the propagation. If you want to prevent an event from reaching parent components, you need to call e.stopPropagation() like this Button component does:
App.jsApp.js Download ResetForkfunction Button({ onClick, children }) {
```
  return (
    <button onClick={e => {
      e.stopPropagation();
      onClick();
    }}>
      {children}
    </button>
  );
}

export default function Toolbar() {
  return (
    <div className="Toolbar" onClick={() => {
      alert('You clicked on the toolbar!');
    }}>
      <Button onClick={() => alert('Playing!')}>
        Play Movie
      </Button>
      <Button onClick={() => alert('Uploading!')}>
        Upload Image
      </Button>
    </div>
  );
}
```

Show more
When you click on a button:

React calls the onClick handler passed to <button>.
That handler, defined in Button, does the following:

Calls e.stopPropagation(), preventing the event from bubbling further.
Calls the onClick function, which is a prop passed from the Toolbar component.

That function, defined in the Toolbar component, displays the button's own alert.
Since the propagation was stopped, the parent 's onClick handler does not run.

As a result of e.stopPropagation(), clicking on the buttons now only shows a single alert (from the <button>) rather than the two of them (from the <button> and the parent toolbar). Clicking a button is not the same thing as clicking the surrounding toolbar, so stopping the propagation makes sense for this UI.
Deep DiveCapture phase events Show DetailsIn rare cases, you might need to catch all events on child elements, even if they stopped propagation. For example, maybe you want to log every click to analytics, regardless of the propagation logic. You can do this by adding Capture at the end of the event name:<div onClickCapture={() => { /* this runs first */ }}> <button onClick={e => e.stopPropagation()} /> <button onClick={e => e.stopPropagation()} /> Each event propagates in three phases:
It travels down, calling all onClickCapture handlers.
It runs the clicked element's onClick handler.
It travels upwards, calling all onClick handlers.
Capture events are useful for code like routers or analytics, but you probably won't use them in app code.
Passing handlers as alternative to propagation
Notice how this click handler runs a line of code and then calls the onClick prop passed by the parent:
function Button({ onClick, children }) { return (<button onClick={e => { e.stopPropagation(); onClick(); }}> {children} </button>);}
You could add more code to this handler before calling the parent onClick event handler, too. This pattern provides an alternative to propagation. It lets the child component handle the event, while also letting the parent component specify some additional behavior. Unlike propagation, it's not automatic. But the benefit of this pattern is that you can clearly follow the whole chain of code that executes as a result of some event.
If you rely on propagation and it's difficult to trace which handlers execute and why, try this approach instead.
Preventing default behavior
Some browser events have default behavior associated with them. For example, a <form> submit event, which happens when a button inside of it is clicked, will reload the whole page by default:
App.jsApp.js Download ResetForkexport default function Signup() {
```
  return (
    <form onSubmit={() => alert('Submitting!')}>
      <input />
      <button>Send</button>
    </form>
```

```
  );
}
```

You can call e.preventDefault() on the event object to stop this from happening:
App.jsApp.js Download ResetForkexport default function Signup() {

```
  return (
    <form onSubmit={e => {
      e.preventDefault();
      alert('Submitting!');
    }}>
      <input />
      <button>Send</button>
    </form>
  );
}
```

Don’t confuse e.stopPropagation() and e.preventDefault(). They are both useful, but are unrelated:

e.stopPropagation() stops the event handlers attached to the tags above from firing.
e.preventDefault() prevents the default browser behavior for the few events that have it.

Can event handlers have side effects?
Absolutely! Event handlers are the best place for side effects.
Unlike rendering functions, event handlers don’t need to be pure, so it’s a great place to change something—for example, change an input’s value in response to typing, or change a list in response to a button press. However, in order to change some information, you first need some way to store it. In React, this is done by using state, a component’s memory. You will learn all about it on the next page.
Recap
You can handle events by passing a function as a prop to an element like <button>.
Event handlers must be passed, not called! onClick={handleClick}, not onClick={handleClick()}.
You can define an event handler function separately or inline.
Event handlers are defined inside a component, so they can access props.
You can declare an event handler in a parent and pass it as a prop to a child.
You can define your own event handler props with application-specific names.
Events propagate upwards. Call e.stopPropagation() on the first argument to prevent that.
Events may have unwanted default browser behavior. Call e.preventDefault() to prevent that.
Explicitly calling an event handler prop from a child handler is a good alternative to propagation.

Try out some challenges1. Fix an event handler 2. Wire up the events Challenge 1 of 2: Fix an event handler Clicking this button is supposed to switch the page background between white and black. However, nothing happens when you click it. Fix the problem. (Don’t worry about the logic inside handleClick—that part is fine.)App.jsApp.js Download ResetForkexport default function LightSwitch() {
```
  function handleClick() {
    let bodyStyle = document.body.style;
    if (bodyStyle.backgroundColor === 'black') {
      bodyStyle.backgroundColor = 'white';
    } else {
      bodyStyle.backgroundColor = 'black';
    }
  }

  return (
    <button onClick={handleClick()}>
      Toggle the lights
    </button>
  );
}
```

Show more Show solutionNext ChallengePreviousAdding InteractivityNextState: A Component's Memory
Learn ReactAdding InteractivityState: A Component's MemoryComponents often need to change what’s on the screen as a result of an interaction. Typing into the form should update the input field, clicking “next” on an image carousel should change which image is displayed, clicking “buy” should put a product in the shopping cart. Components need to “remember” things: the current input value, the current image, the shopping cart. In React, this kind of component-specific memory is called state.
You will learn
How to add a state variable with the useState Hook
What pair of values the useState Hook returns
How to add more than one state variable
Why state is called local

When a regular variable isn’t enough
Here’s a component that renders a sculpture image. Clicking the “Next” button should show the next sculpture by changing the index to 1, then 2, and so on. However, this won’t work (you can try it!):
App.jsdata.jsApp.js ResetFork99123456789101112131415161718192021222324252627282930313233import { sculptureList } from './data.js';export default function Gallery() { let index = 0; function handleClick() { index = index + 1; } let sculpture = sculptureList[index]; return (<> <button onClick={handleClick}> Next </button> <h2> {sculpture.name} by {sculpture.artist} </h2> <h3> ({index + 1} of {sculptureList.length}) </h3> <p> {sculpture.description} </

p> </>);}Show more
The handleClick event handler is updating a local variable, index. But two things prevent that change from being visible:

Local variables don’t persist between renders. When React renders this component a second time, it renders it from scratch—it doesn’t consider any changes to the local variables.
Changes to local variables won’t trigger renders. React doesn’t realize it needs to render the component again with the new data.

To update a component with new data, two things need to happen:

Retain the data between renders.
Trigger React to render the component with new data (re-rendering).

The useState Hook provides those two things:

A state variable to retain the data between renders.
A state setter function to update the variable and trigger React to render the component again.

Adding a state variable
To add a state variable, import useState from React at the top of the file:
import { useState } from 'react';
Then, replace this line:
let index = 0;
with
const [index, setIndex] = useState(0);
index is a state variable and setIndex is the setter function.

The [and] syntax here is called array destructuring and it lets you read values from an array. The array returned by useState always has exactly two items.

This is how they work together in handleClick:
function handleClick() { setIndex(index + 1);}
Now clicking the “Next” button switches the current sculpture:
App.jsdata.jsApp.js ResetForkimport { useState } from 'react';
import { sculptureList } from './data.js';

```
export default function Gallery() {
  const [index, setIndex] = useState(0);

  function handleClick() {
    setIndex(index + 1);
  }
```

```
  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
      </h2>
      <h3>
        ({index + 1} of {sculptureList.length})
      </h3>
      <img
        src={sculpture.url}
        alt={sculpture.alt}
      />
      <p>
        {sculpture.description}
      </p>
    </>
  );
}
```

Show more

Meet your first Hook

In React, useState, as well as any other function starting with ”use”, is called a Hook. Hooks are special functions that are only available while React is rendering (which we’ll get into in more detail on the next page). They let you “hook into” different React features.

State is just one of those features, but you will meet the other Hooks later.

PitfallHooks—functions starting with use—can only be called at the top level of your components or your own Hooks. You can’t call Hooks inside conditions, loops, or other nested functions. Hooks are functions, but it’s helpful to think of them as unconditional declarations about your component’s needs. You “use” React features at the top of your component similar to how you “import” modules at the top of your file.

Anatomy of useState

When you call useState, you are telling React that you want this component to remember something:

`const [index, setIndex] = useState(0);`

In this case, you want React to remember index.

NoteThe convention is to name this pair like const [something, setSomething]. You could name it anything you like, but conventions make things easier to understand across projects.

The only argument to useState is the initial value of your state variable. In this example, the index’s initial value is set to 0 with useState(0).

Every time your component renders, useState gives you an array containing two values:

The state variable (index) with the value you stored.
The state setter function (setIndex) which can update the state variable and trigger React to render the component again.

Here’s how that happens in action:
const [index, setIndex] = useState(0);

Your component renders the first time. Because you passed 0 to useState as the initial value for index, it will return [0, setIndex]. React remembers 0 is the latest state value.
You update the state. When a user clicks the button, it calls setIndex(index + 1). index is 0, so it’s setIndex(1). This tells React to remember index is 1 now and triggers another render.
Your component’s second render. React still sees useState(0), but because React remembers that you set index to 1, it returns [1, setIndex] instead.
And so on!

Giving a component multiple state variables
You can have as many state variables of as many types as you like in one component. This component has two state variables, a number index and a boolean showMore that’s toggled when you click “Show details”:
App.jsdata.jsApp.js ResetForkimport { useState } from 'react';
import { sculptureList } from './data.js';

```
export default function Gallery() {
  const [index, setIndex] = useState(0);
  const [showMore, setShowMore] = useState(false);

  function handleNextClick() {
    setIndex(index + 1);
  }

  function handleMoreClick() {
    setShowMore(!showMore);
  }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleNextClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
```

```
      </h2>
      <h3>
        ({index + 1} of {sculptureList.length})
      </h3>
      <button onClick={handleMoreClick}>
        {showMore ? 'Hide' : 'Show'} details
      </button>
      {showMore && <p>{sculpture.description}</p>}
      <img
        src={sculpture.url}
        alt={sculpture.alt}
      />
    </>
  );
}
```

Show more

It is a good idea to have multiple state variables if their state is unrelated, like index and showMore in this example. But if you find that you often change two state variables together, it might be easier to combine them into one. For example, if you have a form with many fields, it's more convenient to have a single state variable that holds an object than state variable per field. Read Choosing the State Structure for more tips. Deep DiveHow does React know which state to return? Show DetailsYou might have noticed that the useState call does not receive any information about which state variable it refers to. There is no "identifier" that is passed to useState, so how does it know which of the state variables to return? Does it rely on some magic like parsing your functions? The answer is no.Instead, to enable their concise syntax, Hooks rely on a stable call order on every render of the same component. This works well in practice because if you follow the rule above ("only call Hooks at the top level"), Hooks will always be called in the same order. Additionally, a linter plugin catches most mistakes.Internally, React holds an array of state pairs for every component. It also maintains the current pair index, which is set to 0 before rendering. Each time you call useState, React gives you the next state pair and increments the index. You can read more about this mechanism in React Hooks: Not Magic, Just Arrays.This example doesn't use React but it gives you an idea of how useState works internally:index.jsindex.htmlindex.js ResetForklet componentHooks = [];
let currentHookIndex = 0;

```
// How useState works inside React (simplified).
function useState(initialState) {
  let pair = componentHooks[currentHookIndex];
  if (pair) {
    // This is not the first render,
    // so the state pair already exists.
    // Return it and prepare for next Hook call.
    currentHookIndex++;
```

```
    return pair;
  }

  // This is the first time we're rendering,
  // so create a state pair and store it.
  pair = [initialState, setState];

  function setState(nextState) {
    // When the user requests a state change,
    // put the new value into the pair.
    pair[0] = nextState;
    updateDOM();
  }

  // Store the pair for future renders
  // and prepare for the next Hook call.
  componentHooks[currentHookIndex] = pair;
  currentHookIndex++;
  return pair;
}

function Gallery() {
  // Each useState() call will get the next pair.
  const [index, setIndex] = useState(0);
  const [showMore, setShowMore] = useState(false);

  function handleNextClick() {
    setIndex(index + 1);
  }

  function handleMoreClick() {
    setShowMore(!showMore);
  }

  let sculpture = sculptureList[index];
  // This example doesn't use React, so
  // return an output object instead of JSX.
  return {
    onNextClick: handleNextClick,
    onMoreClick: handleMoreClick,
    header: `${sculpture.name} by ${sculpture.artist}`,
    counter: `${index + 1} of ${sculptureList.length}`,
    more: `${showMore ? 'Hide' : 'Show'} details`,
    description: showMore ? sculpture.description : null,
    imageSrc: sculpture.url,
    imageAlt: sculpture.alt
```

```
  };
}

function updateDOM() {
  // Reset the current Hook index
  // before rendering the component.
  currentHookIndex = 0;
  let output = Gallery();

  // Update the DOM to match the output.
  // This is the part React does for you.
  nextButton.onclick = output.onNextClick;
  header.textContent = output.header;
  moreButton.onclick = output.onMoreClick;
  moreButton.textContent = output.more;
  image.src = output.imageSrc;
  image.alt = output.imageAlt;
  if (output.description !== null) {
    description.textContent = output.description;
    description.style.display = '';
  } else {
    description.style.display = 'none';
  }
}

let nextButton = document.getElementById('nextButton');
let header = document.getElementById('header');
let moreButton = document.getElementById('moreButton');
let description = document.getElementById('description');
let image = document.getElementById('image');
let sculptureList = [{
  name: 'Homenaje a la Neurocirugía',
  artist: 'Marta Colvin Andrade',
  description: 'Although Colvin is predominantly known for abstract themes that allude to
pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her
most recognizable public art pieces.',
  url: 'https://i.imgur.com/Mx7dA2Y.jpg',
  alt: 'A bronze statue of two crossed hands delicately holding a human brain in their
fingertips.'
}, {
  name: 'Floralis Genérica',
  artist: 'Eduardo Catalano',
  description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is
designed to move, closing its petals in the evening or when strong winds blow and
opening them in the morning.',
  url: 'https://i.imgur.com/ZF6s192m.jpg',
```

```
  alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'
}, {
  name: 'Eternal Presence',
  artist: 'John Woodrow Wilson',
  description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',
  url: 'https://i.imgur.com/aTtVpES.jpg',
  alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'
}, {
  name: 'Moai',
  artist: 'Unknown Artist',
  description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',
  url: 'https://i.imgur.com/RCwLEoQm.jpg',
  alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'
}, {
  name: 'Blue Nana',
  artist: 'Niki de Saint Phalle',
  description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',
  url: 'https://i.imgur.com/Sd1AgUOm.jpg',
  alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'
}, {
  name: 'Ultimate Form',
  artist: 'Barbara Hepworth',
  description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',
  url: 'https://i.imgur.com/2heNQDcm.jpg',
  alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'
}, {
  name: 'Cavaliere',
  artist: 'Lamidi Olonade Fakeye',
  description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",
  url: 'https://i.imgur.com/wIdGuZwm.png',
```

```
  alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'
}, {
  name: 'Big Bellies',
  artist: 'Alina Szapocznikow',
  description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",
  url: 'https://i.imgur.com/AlHTAdDm.jpg',
  alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'
}, {
  name: 'Terracotta Army',
  artist: 'Unknown Artist',
  description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',
  url: 'https://i.imgur.com/HMFmH6m.jpg',
  alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'
}, {
  name: 'Lunar Landscape',
  artist: 'Louise Nevelson',
  description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',
  url: 'https://i.imgur.com/rN7hY6om.jpg',
  alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'
}, {
  name: 'Aureole',
  artist: 'Ranjani Shettar',
  description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',
  url: 'https://i.imgur.com/okTpbHhm.jpg',
  alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'
}, {
  name: 'Hippos',
  artist: 'Taipei Zoo',
  description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged
```

```
hippos at play.',
  url: 'https://i.imgur.com/6o5Vuyu.jpg',
  alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'
}];

// Make UI match the initial state.
updateDOM();
```

Show moreYou don’t have to understand it to use React, but you might find this a helpful mental model.

State is isolated and private

State is local to a component instance on the screen. In other words, if you render the same component twice, each copy will have completely isolated state! Changing one of them will not affect the other.

In this example, the Gallery component from earlier is rendered twice with no changes to its logic. Try clicking the buttons inside each of the galleries. Notice that their state is independent:

App.jsGallery.jsdata.jsApp.js ResetForkimport Gallery from './Gallery.js';

```
export default function Page() {
  return (
    <div className="Page">
      <Gallery />
      <Gallery />
    </div>
  );
}
```

This is what makes state different from regular variables that you might declare at the top of your module. State is not tied to a particular function call or a place in the code, but it’s “local” to the specific place on the screen. You rendered two <Gallery /> components, so their state is stored separately.

Also notice how the Page component doesn’t “know” anything about the Gallery state or even whether it has any. Unlike props, state is fully private to the component declaring it. The parent component can’t change it. This lets you add state to any component or remove it without impacting the rest of the components.

What if you wanted both galleries to keep their states in sync? The right way to do it in React is to remove state from child components and add it to their closest shared parent. The next few pages will focus on organizing state of a single component, but we will return to this topic in Sharing State Between Components.

Recap

Use a state variable when a component needs to “remember” some information between renders.

State variables are declared by calling the useState Hook.

Hooks are special functions that start with use. They let you “hook into” React features like state.
Hooks might remind you of imports: they need to be called unconditionally. Calling Hooks, including useState, is only valid at the top level of a component or another Hook.
The useState Hook returns a pair of values: the current state and the function to update it.
You can have more than one state variable. Internally, React matches them up by their order.
State is private to the component. If you render it in two places, each copy gets its own state.

Try out some challenges1. Complete the gallery 2. Fix stuck form inputs 3. Fix a crash 4. Remove unnecessary state Challenge 1 of 4: Complete the gallery When you press “Next” on the last sculpture, the code crashes. Fix the logic to prevent the crash. You may do this by adding extra logic to event handler or by disabling the button when the action is not possible.After fixing the crash, add a “Previous” button that shows the previous sculpture. It shouldn’t crash on the first sculpture.App.jsdata.jsApp.js ResetForkimport { useState } from 'react';
import { sculptureList } from './data.js';

```
export default function Gallery() {
  const [index, setIndex] = useState(0);
  const [showMore, setShowMore] = useState(false);

  function handleNextClick() {
    setIndex(index + 1);
  }

  function handleMoreClick() {
    setShowMore(!showMore);
  }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleNextClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
      </h2>
      <h3>
        ({index + 1} of {sculptureList.length})
      </h3>
      <button onClick={handleMoreClick}>
```

```
        {showMore ? 'Hide' : 'Show'} details
      </button>
      {showMore && <p>{sculpture.description}</p>}
      <img
        src={sculpture.url}
        alt={sculpture.alt}
      />
    </>
  );
}
```

Show more Show solutionNext ChallengePreviousResponding to EventsNextRender and Commit
Learn ReactAdding InteractivityRender and CommitBefore your components are displayed on screen, they must be rendered by React. Understanding the steps in this process will help you think about how your code executes and explain its behavior.
You will learn
What rendering means in React
When and why React renders a component
The steps involved in displaying a component on screen
Why rendering does not always produce a DOM update

Imagine that your components are cooks in the kitchen, assembling tasty dishes from ingredients. In this scenario, React is the waiter who puts in requests from customers and brings them their orders. This process of requesting and serving UI has three steps:

Triggering a render (delivering the guest's order to the kitchen)
Rendering the component (preparing the order in the kitchen)
Committing to the DOM (placing the order on the table)

TriggerRenderCommitIllustrated by Rachel Lee Nabors
Step 1: Trigger a render
There are two reasons for a component to render:

It's the component's initial render.
The component's (or one of its ancestors') state has been updated.

Initial render
When your app starts, you need to trigger the initial render. Frameworks and sandboxes sometimes hide this code, but it's done by calling createRoot with the target DOM node, and then calling its render method with your component:
index.jsImage.jsindex.js ResetForkimport Image from './Image.js';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'))
root.render(<Image />);

Try commenting out the root.render() call and see the component disappear!
Re-renders when state updates
Once the component has been initially rendered, you can trigger further renders by updating its state with the set function. Updating your component's state automatically queues a render. (You can imagine these as a restaurant guest ordering tea, dessert, and all sorts of things after putting in their first order, depending on the state of their thirst or hunger.)
State update.....triggers.....render!Illustrated by Rachel Lee Nabors
Step 2: React renders your components
After you trigger a render, React calls your components to figure out what to display on screen. "Rendering" is React calling your components.

On initial render, React will call the root component.
For subsequent renders, React will call the function component whose state update triggered the render.

This process is recursive: if the updated component returns some other component, React will render that component next, and if that component also returns something, it will render that component next, and so on. The process will continue until there are no more nested components and React knows exactly what should be displayed on screen. In the following example, React will call Gallery() and Image() several times:
Gallery.jsindex.jsGallery.js ResetForkexport default function Gallery() {

```
  return (
    <section>
      <h1>Inspiring Sculptures</h1>
      <Image />
      <Image />
      <Image />
    </section>
  );
}

function Image() {
  return (
    <img
      src="https://i.imgur.com/ZF6s192.jpg"
      alt="'Floralis Genérica' by Eduardo Catalano: a gigantic metallic flower sculpture with reflective petals"
    />
  );
}
```

Show more

During the initial render, React will create the DOM nodes for <section>, <h1>, and three <img> tags.
During a re-render, React will calculate which of their properties, if any, have changed since the previous render. It won't do anything with that information until the next step, the commit phase.

PitfallRendering must always be a pure calculation:
Same inputs, same output. Given the same inputs, a component should always return the same JSX. (When someone orders a salad with tomatoes, they should not receive a salad with onions!)
It minds its own business. It should not change any objects or variables that existed before rendering. (One order should not change anyone else's order.)
Otherwise, you can encounter confusing bugs and unpredictable behavior as your codebase grows in complexity. When developing in "Strict Mode", React calls each component's function twice, which can help surface mistakes caused by impure functions.
Deep DiveOptimizing performance Show DetailsThe default behavior of rendering all components nested within the updated component is not optimal for performance if the updated component is very high in the tree. If you run into a performance issue, there are several opt-in ways to solve it described in the Performance section. Don't optimize prematurely!
Step 3: React commits changes to the DOM
After rendering (calling) your components, React will modify the DOM.

For the initial render, React will use the appendChild() DOM API to put all the DOM nodes it has created on screen.
For re-renders, React will apply the minimal necessary operations (calculated while rendering!) to make the DOM match the latest rendering output.

React only changes the DOM nodes if there's a difference between renders. For example, here is a component that re-renders with different props passed from its parent every second. Notice how you can add some text into the <input>, updating its value, but the text doesn't disappear when the component re-renders:
Clock.jsClock.js ResetForkexport default function Clock({ time }) {

```
  return (
    <>
      <h1>{time}</h1>
      <input />
    </>
  );
}
```

This works because during this last step, React only updates the content of <h1> with the new time. It sees that the <input> appears in the JSX in the same place as last time, so React doesn't touch the <input>—or its value!

Epilogue: Browser paint
After rendering is done and React updated the DOM, the browser will repaint the screen. Although this process is known as “browser rendering”, we’ll refer to it as “painting” to avoid confusion throughout the docs.
Illustrated by Rachel Lee Nabors
Recap
Any screen update in a React app happens in three steps:

Trigger
Render
Commit

You can use Strict Mode to find mistakes in your components
React does not touch the DOM if the rendering result is the same as last time
PreviousState: A Component's MemoryNextState as a Snapshot
Learn ReactAdding InteractivityState as a SnapshotState variables might look like regular JavaScript variables that you can read and write to. However, state behaves more like a snapshot. Setting it does not change the state variable you already have, but instead triggers a re-render.
You will learn
How setting state triggers re-renders
When and how state updates
Why state does not update immediately after you set it
How event handlers access a “snapshot” of the state

Setting state triggers renders
You might think of your user interface as changing directly in response to the user event like a click. In React, it works a little differently from this mental model. On the previous page, you saw that setting state requests a re-render from React. This means that for an interface to react to the event, you need to update the state.
In this example, when you press “send”, setIsSent(true) tells React to re-render the UI:
App.jsApp.js Download ResetFork991234567891011121314151617181920212223242526272 8import { useState } from 'react';export default function Form() { const [isSent, setIsSent] = useState(false); const [message, setMessage] = useState('Hi!'); if (isSent) { return <h1>Your message is on its way!</h1> } return (<form onSubmit={(e) => { e.preventDefault(); setIsSent(true); sendMessage(message); }}> <textarea placeholder="Message" value={message} onChange={e => setMessage(e.target.value)} /> <button type="submit">Send</button> </form>);}function sendMessage(message) { // ...}Show more
Here’s what happens when you click the button:

The onSubmit event handler executes.
setIsSent(true) sets isSent to true and queues a new render.
React re-renders the component according to the new isSent value.

Let’s take a closer look at the relationship between state and rendering.
Rendering takes a snapshot in time
“Rendering” means that React is calling your component, which is a function. The JSX you return from that function is like a snapshot of the UI in time. Its props, event handlers, and local variables were all calculated using its state at the time of the render.
Unlike a photograph or a movie frame, the UI “snapshot” you return is interactive. It includes logic like event handlers that specify what happens in response to inputs. React updates the screen to match this snapshot and connects the event handlers. As a result, pressing a button will trigger the click handler from your JSX.
When React re-renders a component:

React calls your function again.
Your function returns a new JSX snapshot.
React then updates the screen to match the snapshot you’ve returned.

React executing the functionCalculating the snapshotUpdating the DOM treeIllustrated by Rachel Lee Nabors
As a component’s memory, state is not like a regular variable that disappears after your function returns. State actually “lives” in React itself—as if on a shelf!—outside of your function. When React calls your component, it gives you a snapshot of the state for that particular render. Your component returns a snapshot of the UI with a fresh set of props and event handlers in its JSX, all calculated using the state values from that render!
You tell React to update the stateReact updates the state valueReact passes a snapshot of the state value into the componentIllustrated by Rachel Lee Nabors
Here’s a little experiment to show you how this works. In this example, you might expect that clicking the “+3” button would increment the counter three times because it calls setNumber(number + 1) three times.
See what happens when you click the “+3” button:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 1);
        setNumber(number + 1);
        setNumber(number + 1);
      }}>+3</button>
    </>
  )
}
```

Show more
Notice that number only increments once per click!
Setting state only changes it for the next render. During the first render, number was 0. This is why, in that render's onClick handler, the value of number is still 0 even after setNumber(number + 1) was called:
<button onClick={() => { setNumber(number + 1); setNumber(number + 1); setNumber(number + 1);}}>+3</button>
Here is what this button's click handler tells React to do:

setNumber(number + 1): number is 0 so setNumber(0 + 1).

React prepares to change number to 1 on the next render.

setNumber(number + 1): number is 0 so setNumber(0 + 1).

React prepares to change number to 1 on the next render.

setNumber(number + 1): number is 0 so setNumber(0 + 1).

React prepares to change number to 1 on the next render.

Even though you called setNumber(number + 1) three times, in this render's event handler number is always 0, so you set the state to 1 three times. This is why, after your event handler finishes, React re-renders the component with number equal to 1 rather than 3.
You can also visualize this by mentally substituting state variables with their values in your code. Since the number state variable is 0 for this render, its event handler looks like this:
<button onClick={() => { setNumber(0 + 1); setNumber(0 + 1); setNumber(0 + 1);}}>+3</button>
For the next render, number is 1, so that render's click handler looks like this:
<button onClick={() => { setNumber(1 + 1); setNumber(1 + 1); setNumber(1 + 1);}}>+3</button>
This is why clicking the button again will set the counter to 2, then to 3 on the next click, and so on.
State over time
Well, that was fun. Try to guess what clicking this button will alert:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [number, setNumber] = useState(0);
```

```
  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
        alert(number);
      }}>+5</button>
    </>
  )
}
```

If you use the substitution method from before, you can guess that the alert shows “0”:
setNumber(0 + 5);alert(0);
But what if you put a timer on the alert, so it only fires after the component re-rendered? Would it say “0” or “5”? Have a guess!
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
        setTimeout(() => {
          alert(number);
        }, 3000);
      }}>+5</button>
    </>
  )
}
```

Show more
Surprised? If you use the substitution method, you can see the “snapshot” of the state passed to the alert.
setNumber(0 + 5);setTimeout(() => { alert(0);}, 3000);
The state stored in React may have changed by the time the alert runs, but it was scheduled using a snapshot of the state at the time the user interacted with it!
A state variable’s value never changes within a render, even if its event handler’s code is asynchronous. Inside that render’s onClick, the value of number continues to be 0 even after setNumber(number + 5) was called. Its value was “fixed” when React “took the snapshot” of the UI by calling your component.
Here is an example of how that makes your event handlers less prone to timing mistakes. Below is a form that sends a message with a five-second delay. Imagine this scenario:

You press the “Send” button, sending “Hello” to Alice.
Before the five-second delay ends, you change the value of the “To” field to “Bob”.

What do you expect the alert to display? Would it display, “You said Hello to Alice”? Or would it display, “You said Hello to Bob”? Make a guess based on what you know, and then try it:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [to, setTo] = useState('Alice');
  const [message, setMessage] = useState('Hello');

  function handleSubmit(e) {
    e.preventDefault();
    setTimeout(() => {
      alert(`You said ${message} to ${to}`);
    }, 5000);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        To:{' '}
        <select
          value={to}
          onChange={e => setTo(e.target.value)}>
          <option value="Alice">Alice</option>
          <option value="Bob">Bob</option>
        </select>
      </label>
      <textarea
        placeholder="Message"
        value={message}
        onChange={e => setMessage(e.target.value)}
      />
      <button type="submit">Send</button>
    </form>
  );
}
```

Show more
React keeps the state values “fixed” within one render’s event handlers. You don’t need to worry whether the state has changed while the code is running.
But what if you wanted to read the latest state before a re-render? You’ll want to use a state updater function, covered on the next page!

Recap
Setting state requests a new render.
React stores state outside of your component, as if on a shelf.
When you call useState, React gives you a snapshot of the state for that render.
Variables and event handlers don’t “survive” re-renders. Every render has its own event handlers.
Every render (and functions inside it) will always “see” the snapshot of the state that React gave to that render.
You can mentally substitute state in event handlers, similarly to how you think about the rendered JSX.
Event handlers created in the past have the state values from the render in which they were created.

Try out some challengesChallenge 1 of 1: Implement a traffic light Here is a crosswalk light component that toggles when the button is pressed:App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function TrafficLight() {
  const [walk, setWalk] = useState(true);

  function handleClick() {
    setWalk(!walk);
  }

  return (
    <>
      <button onClick={handleClick}>
        Change to {walk ? 'Stop' : 'Walk'}
      </button>
      <h1 style={{
        color: walk ? 'darkgreen' : 'darkred'
      }}>
        {walk ? 'Walk' : 'Stop'}
      </h1>
    </>
  );
}
```

Show moreAdd an alert to the click handler. When the light is green and says “Walk”, clicking the button should say “Stop is next”. When the light is red and says “Stop”, clicking the button should say “Walk is next”.Does it make a difference whether you put the alert before or after the setWalk call? Show solutionPreviousRender and CommitNextQueueing a Series of State Updates
Learn ReactAdding InteractivityQueueing a Series of State UpdatesSetting a state variable will queue another render. But sometimes you might want to perform multiple operations on the value before queueing the next render. To do this, it helps to

understand how React batches state updates.
You will learn
What “batching” is and how React uses it to process multiple state updates
How to apply several updates to the same state variable in a row

React batches state updates
You might expect that clicking the “+3” button will increment the counter three times because it calls setNumber(number + 1) three times:
App.jsApp.js Download ResetFork991234567891011121314151617import { useState } from 'react';export default function Counter() { const [number, setNumber] = useState(0); return (<>   <h1>{number}</h1>   <button onClick={() => {   setNumber(number + 1);   setNumber(number + 1);   setNumber(number + 1);   }}>+3</button>   </>)}Show more
However, as you might recall from the previous section, each render’s state values are fixed, so the value of number inside the first render’s event handler is always 0, no matter how many times you call setNumber(1):
setNumber(0 + 1);setNumber(0 + 1);setNumber(0 + 1);
But there is one other factor at play here. React waits until all code in the event handlers has run before processing your state updates. This is why the re-render only happens after all these setNumber() calls.
This might remind you of a waiter taking an order at the restaurant. A waiter doesn’t run to the kitchen at the mention of your first dish! Instead, they let you finish your order, let you make changes to it, and even take orders from other people at the table.
Illustrated by Rachel Lee Nabors
This lets you update multiple state variables—even from multiple components—without triggering too many re-renders. But this also means that the UI won’t be updated until after your event handler, and any code in it, completes. This behavior, also known as batching, makes your React app run much faster. It also avoids dealing with confusing “half-finished” renders where only some of the variables have been updated.
React does not batch across multiple intentional events like clicks—each click is handled separately. Rest assured that React only does batching when it’s generally safe to do. This ensures that, for example, if the first button click disables a form, the second click would not submit it again.
Updating the same state multiple times before the next render
It is an uncommon use case, but if you would like to update the same state variable multiple times before the next render, instead of passing the next state value like setNumber(number + 1), you can pass a function that calculates the next state based on the previous one in the queue, like setNumber(n => n + 1). It is a way to tell React to “do something with the state value” instead of just replacing it.
Try incrementing the counter now:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
```

```
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(n => n + 1);
        setNumber(n => n + 1);
        setNumber(n => n + 1);
      }}>+3</button>
    </>
  )
}
```

Show more
Here, n => n + 1 is called an updater function. When you pass it to a state setter:

React queues this function to be processed after all the other code in the event handler has run.
During the next render, React goes through the queue and gives you the final updated state.

setNumber(n => n + 1);setNumber(n => n + 1);setNumber(n => n + 1);
Here's how React works through these lines of code while executing the event handler:

setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.
setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.
setNumber(n => n + 1): n => n + 1 is a function. React adds it to a queue.

When you call useState during the next render, React goes through the queue. The previous number state was 0, so that's what React passes to the first updater function as the n argument. Then React takes the return value of your previous updater function and passes it to the next updater as n, and so on:
queued updatenreturnsn => n + 100 + 1 = 1n => n + 111 + 1 = 2n => n + 122 + 1 = 3
React stores 3 as the final result and returns it from useState.
This is why clicking "+3" in the above example correctly increments the value by 3.
What happens if you update state after replacing it
What about this event handler? What do you think number will be in the next render?
<button onClick={() => { setNumber(number + 5); setNumber(n => n + 1);}}>
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
```

```
        setNumber(n => n + 1);
      }}>Increase the number</button>
    </>
  )
}
```

Here’s what this event handler tells React to do:

setNumber(number + 5): number is 0, so setNumber(0 + 5). React adds “replace with 5” to its queue.
setNumber(n => n + 1): n => n + 1 is an updater function. React adds that function to its queue.

During the next render, React goes through the state queue:
queued updatenreturns“replace with 5”0 (unused)5n => n + 155 + 1 = 6
React stores 6 as the final result and returns it from useState.
NoteYou may have noticed that setState(5) actually works like setState(n => 5), but n is unused!
What happens if you replace state after updating it
Let’s try one more example. What do you think number will be in the next render?
<button onClick={() => { setNumber(number + 5); setNumber(n => n + 1);
setNumber(42);}}>
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
        setNumber(n => n + 1);
        setNumber(42);
      }}>Increase the number</button>
    </>
  )
}
```

Show more
Here’s how React works through these lines of code while executing this event handler:

setNumber(number + 5): number is 0, so setNumber(0 + 5). React adds “replace with 5” to its queue.
setNumber(n => n + 1): n => n + 1 is an updater function. React adds that function to its queue.

setNumber(42): React adds “replace with 42” to its queue.

During the next render, React goes through the state queue:
queued updatenreturns“replace with 5”0 (unused)5n => n + 155 + 1 = 6“replace with 42”6 (unused)42
Then React stores 42 as the final result and returns it from useState.
To summarize, here’s how you can think of what you’re passing to the setNumber state setter:

An updater function (e.g. n => n + 1) gets added to the queue.
Any other value (e.g. number 5) adds “replace with 5” to the queue, ignoring what’s already queued.

After the event handler completes, React will trigger a re-render. During the re-render, React will process the queue. Updater functions run during rendering, so updater functions must be pure and only return the result. Don’t try to set state from inside of them or run other side effects. In Strict Mode, React will run each updater function twice (but discard the second result) to help you find mistakes.
Naming conventions
It’s common to name the updater function argument by the first letters of the corresponding state variable:
setEnabled(e => !e);setLastName(ln => ln.reverse());setFriendCount(fc => fc * 2);
If you prefer more verbose code, another common convention is to repeat the full state variable name, like setEnabled(enabled => !enabled), or to use a prefix like setEnabled(prevEnabled => !prevEnabled).
Recap
Setting state does not change the variable in the existing render, but it requests a new render.
React processes state updates after event handlers have finished running. This is called batching.
To update some state multiple times in one event, you can use setNumber(n => n + 1) updater function.

Try out some challenges1. Fix a request counter 2. Implement the state queue yourself Challenge 1 of 2: Fix a request counter You’re working on an art marketplace app that lets the user submit multiple orders for an art item at the same time. Each time the user presses the “Buy” button, the “Pending” counter should increase by one. After three seconds, the “Pending” counter should decrease, and the “Completed” counter should increase.However, the “Pending” counter does not behave as intended. When you press “Buy”, it decreases to -1 (which should not be possible!). And if you click fast twice, both counters seem to behave unpredictably.Why does this happen? Fix both counters.App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function RequestTracker() {
  const [pending, setPending] = useState(0);
  const [completed, setCompleted] = useState(0);
```

```
  async function handleClick() {
    setPending(pending + 1);
    await delay(3000);
    setPending(pending - 1);
    setCompleted(completed + 1);
  }

  return (
    <>
      <h3>
        Pending: {pending}
      </h3>
      <h3>
        Completed: {completed}
      </h3>
      <button onClick={handleClick}>
        Buy
      </button>
    </>
  );
}

function delay(ms) {
  return new Promise(resolve => {
    setTimeout(resolve, ms);
  });
}
```

Show more Show solutionNext ChallengePreviousState as a SnapshotNextUpdating Objects in State
Learn ReactAdding InteractivityUpdating Objects in StateState can hold any kind of JavaScript value, including objects. But you shouldn’t change objects that you hold in the React state directly. Instead, when you want to update an object, you need to create a new one (or make a copy of an existing one), and then set the state to use that copy.
You will learn
How to correctly update an object in React state
How to update a nested object without mutating it
What immutability is, and how not to break it
How to make object copying less repetitive with Immer

What’s a mutation?
You can store any kind of JavaScript value in state.
const [x, setX] = useState(0);
So far you’ve been working with numbers, strings, and booleans. These kinds of JavaScript values are “immutable”, meaning unchangeable or “read-only”. You can

trigger a re-render to replace a value:
setX(5);
The x state changed from 0 to 5, but the number 0 itself did not change. It’s not possible to make any changes to the built-in primitive values like numbers, strings, and booleans in JavaScript.
Now consider an object in state:
const [position, setPosition] = useState({ x: 0, y: 0 });
Technically, it is possible to change the contents of the object itself. This is called a mutation:
position.x = 5;
However, although objects in React state are technically mutable, you should treat them as if they were immutable—like numbers, booleans, and strings. Instead of mutating them, you should always replace them.
Treat state as read-only
In other words, you should treat any JavaScript object that you put into state as read-only.
This example holds an object in state to represent the current pointer position. The red dot is supposed to move when you touch or move the cursor over the preview area. But the dot stays in the initial position:
App.jsApp.js Download ResetForkimport { useState } from 'react';
export default function MovingDot() {
  const [position, setPosition] = useState({
    x: 0,
    y: 0
  });
  return (
    <div
      onPointerMove={e => {
        position.x = e.clientX;
        position.y = e.clientY;
      }}
      style={{
        position: 'relative',
        width: '100vw',
        height: '100vh',
      }}>
      <div style={{
        position: 'absolute',
        backgroundColor: 'red',
        borderRadius: '50%',
        transform: `translate(${position.x}px, ${position.y}px)`,
        left: -10,
        top: -10,
        width: 20,
        height: 20,
      }} />

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

Show more
The problem is with this bit of code.
onPointerMove={e => { position.x = e.clientX; position.y = e.clientY;}}
This code modifies the object assigned to position from the previous render. But without using the state setting function, React has no idea that object has changed. So React does not do anything in response. It's like trying to change the order after you've already eaten the meal. While mutating state can work in some cases, we don't recommend it. You should treat the state value you have access to in a render as read-only.
To actually trigger a re-render in this case, create a new object and pass it to the state setting function:
onPointerMove={e => { setPosition({   x: e.clientX,   y: e.clientY });}}
With setPosition, you're telling React:

Replace position with this new object
And render this component again

Notice how the red dot now follows your pointer when you touch or hover over the preview area:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function MovingDot() {
  const [position, setPosition] = useState({
    x: 0,
    y: 0
  });
  return (
    <div
      onPointerMove={e => {
        setPosition({
          x: e.clientX,
          y: e.clientY
        });
      }}
      style={{
        position: 'relative',
        width: '100vw',
        height: '100vh',
      }}>
      <div style={{
        position: 'absolute',
        backgroundColor: 'red',
        borderRadius: '50%',
```

```
        transform: `translate(${position.x}px, ${position.y}px)`,
        left: -10,
        top: -10,
        width: 20,
        height: 20,
      }} />
    </div>
  );
}
```

Show more

Deep DiveLocal mutation is fine Show DetailsCode like this is a problem because it modifies an existing object in state:position.x = e.clientX;position.y = e.clientY;But code like this is absolutely fine because you're mutating a fresh object you have just created:const nextPosition = {};nextPosition.x = e.clientX;nextPosition.y = e.clientY;setPosition(nextPosition);In fact, it is completely equivalent to writing this:setPosition({ x: e.clientX, y: e.clientY});Mutation is only a problem when you change existing objects that are already in state. Mutating an object you've just created is okay because no other code references it yet. Changing it isn't going to accidentally impact something that depends on it. This is called a "local mutation". You can even do local mutation while rendering. Very convenient and completely okay!

Copying objects with the spread syntax

In the previous example, the position object is always created fresh from the current cursor position. But often, you will want to include existing data as a part of the new object you're creating. For example, you may want to update only one field in a form, but keep the previous values for all other fields.

These input fields don't work because the onChange handlers mutate the state:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [person, setPerson] = useState({
    firstName: 'Barbara',
    lastName: 'Hepworth',
    email: 'bhepworth@sculpture.com'
  });

  function handleFirstNameChange(e) {
    person.firstName = e.target.value;
  }

  function handleLastNameChange(e) {
    person.lastName = e.target.value;
  }

  function handleEmailChange(e) {
    person.email = e.target.value;
```

```
  }

  return (
    <>
      <label>
        First name:
        <input
          value={person.firstName}
          onChange={handleFirstNameChange}
        />
      </label>
      <label>
        Last name:
        <input
          value={person.lastName}
          onChange={handleLastNameChange}
        />
      </label>
      <label>
        Email:
        <input
          value={person.email}
          onChange={handleEmailChange}
        />
      </label>
      <p>
        {person.firstName}{' '}
        {person.lastName}{' '}
        ({person.email})
      </p>
    </>
  );
}
```

Show more

For example, this line mutates the state from a past render:

person.firstName = e.target.value;

The reliable way to get the behavior you're looking for is to create a new object and pass it to setPerson. But here, you want to also copy the existing data into it because only one of the fields has changed:

setPerson({ firstName: e.target.value, // New first name from the input lastName: person.lastName, email: person.email});

You can use the ... object spread syntax so that you don't need to copy every property separately.

setPerson({ ...person, // Copy the old fields firstName: e.target.value // But override this one});

Now the form works!
Notice how you didn’t declare a separate state variable for each input field. For large forms, keeping all data grouped in an object is very convenient—as long as you update it correctly!
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [person, setPerson] = useState({
    firstName: 'Barbara',
    lastName: 'Hepworth',
    email: 'bhepworth@sculpture.com'
  });

  function handleFirstNameChange(e) {
    setPerson({
      ...person,
      firstName: e.target.value
    });
  }

  function handleLastNameChange(e) {
    setPerson({
      ...person,
      lastName: e.target.value
    });
  }

  function handleEmailChange(e) {
    setPerson({
      ...person,
      email: e.target.value
    });
  }

  return (
    <>
      <label>
        First name:
        <input
          value={person.firstName}
          onChange={handleFirstNameChange}
        />
      </label>
      <label>
        Last name:
        <input
```

```
          value={person.lastName}
          onChange={handleLastNameChange}
        />
      </label>
      <label>
        Email:
        <input
          value={person.email}
          onChange={handleEmailChange}
        />
      </label>
      <p>
        {person.firstName}{' '}
        {person.lastName}{' '}
        ({person.email})
      </p>
    </>
  );
}
```

Show more

Note that the ... spread syntax is “shallow”—it only copies things one level deep. This makes it fast, but it also means that if you want to update a nested property, you’ll have to use it more than once.

Deep DiveUsing a single event handler for multiple fields Show DetailsYou can also use the [and] braces inside your object definition to specify a property with dynamic name. Here is the same example, but with a single event handler instead of three different ones:App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [person, setPerson] = useState({
    firstName: 'Barbara',
    lastName: 'Hepworth',
    email: 'bhepworth@sculpture.com'
  });

  function handleChange(e) {
    setPerson({
      ...person,
      [e.target.name]: e.target.value
    });
  }

  return (
    <>
      <label>
```

```
        First name:
        <input
          name="firstName"
          value={person.firstName}
          onChange={handleChange}
        />
      </label>
      <label>
        Last name:
        <input
          name="lastName"
          value={person.lastName}
          onChange={handleChange}
        />
      </label>
      <label>
        Email:
        <input
          name="email"
          value={person.email}
          onChange={handleChange}
        />
      </label>
      <p>
        {person.firstName}{' '}
        {person.lastName}{' '}
        ({person.email})
      </p>
    </>
  );
}
```

Show moreHere, e.target.name refers to the name property given to the <input> DOM element.

Updating a nested object

Consider a nested object structure like this:

```
const [person, setPerson] = useState({  name: 'Niki de Saint Phalle',  artwork: {    title: 'Blue Nana',    city: 'Hamburg',    image: 'https://i.imgur.com/Sd1AgUOm.jpg',  }});
```

If you wanted to update person.artwork.city, it’s clear how to do it with mutation:

```
person.artwork.city = 'New Delhi';
```

But in React, you treat state as immutable! In order to change city, you would first need to produce the new artwork object (pre-populated with data from the previous one), and then produce the new person object which points at the new artwork:

```
const nextArtwork = { ...person.artwork, city: 'New Delhi' };const nextPerson = { ...person, artwork: nextArtwork };setPerson(nextPerson);
```

Or, written as a single function call:

setPerson({ ...person, // Copy other fields artwork: { // but replace the artwork   ...person.artwork, // with the same one   city: 'New Delhi' // but in New Delhi! }});
This gets a bit wordy, but it works fine for many cases:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [person, setPerson] = useState({
    name: 'Niki de Saint Phalle',
    artwork: {
      title: 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgUOm.jpg',
    }
  });

  function handleNameChange(e) {
    setPerson({
      ...person,
      name: e.target.value
    });
  }

  function handleTitleChange(e) {
    setPerson({
      ...person,
      artwork: {
        ...person.artwork,
        title: e.target.value
      }
    });
  }

  function handleCityChange(e) {
    setPerson({
      ...person,
      artwork: {
        ...person.artwork,
        city: e.target.value
      }
    });
  }

  function handleImageChange(e) {
    setPerson({
      ...person,
```

```
    artwork: {
      ...person.artwork,
      image: e.target.value
    }
  });
}

return (
  <>
    <label>
      Name:
      <input
        value={person.name}
        onChange={handleNameChange}
      />
    </label>
    <label>
      Title:
      <input
        value={person.artwork.title}
        onChange={handleTitleChange}
      />
    </label>
    <label>
      City:
      <input
        value={person.artwork.city}
        onChange={handleCityChange}
      />
    </label>
    <label>
      Image:
      <input
        value={person.artwork.image}
        onChange={handleImageChange}
      />
    </label>
    <p>
      <i>{person.artwork.title}</i>
      {' by '}
      {person.name}
      <br />
      (located in {person.artwork.city})
    </p>
    <img
      src={person.artwork.image}
```

```
        alt={person.artwork.title}
      />
    </>
  );
}
```

Show more
Deep DiveObjects are not really nested Show DetailsAn object like this appears “nested” in code:let obj = { name: 'Niki de Saint Phalle', artwork: {   title: 'Blue Nana', city: 'Hamburg',   image: 'https://i.imgur.com/Sd1AgUOm.jpg', }};However, “nesting” is an inaccurate way to think about how objects behave. When the code executes, there is no such thing as a “nested” object. You are really looking at two different objects:let obj1 = { title: 'Blue Nana', city: 'Hamburg', image: 'https://i.imgur.com/Sd1AgUOm.jpg',};let obj2 = { name: 'Niki de Saint Phalle', artwork: obj1};The obj1 object is not “inside” obj2. For example, obj3 could “point” at obj1 too:let obj1 = { title: 'Blue Nana', city: 'Hamburg', image: 'https://i.imgur.com/Sd1AgUOm.jpg',};let obj2 = { name: 'Niki de Saint Phalle', artwork: obj1};let obj3 = { name: 'Copycat', artwork: obj1};If you were to mutate obj3.artwork.city, it would affect both obj2.artwork.city and obj1.city. This is because obj3.artwork, obj2.artwork, and obj1 are the same object. This is difficult to see when you think of objects as “nested”. Instead, they are separate objects “pointing” at each other with properties.
Write concise update logic with Immer
If your state is deeply nested, you might want to consider flattening it. But, if you don’t want to change your state structure, you might prefer a shortcut to nested spreads. Immer is a popular library that lets you write using the convenient but mutating syntax and takes care of producing the copies for you. With Immer, the code you write looks like you are “breaking the rules” and mutating an object:
updatePerson(draft => { draft.artwork.city = 'Lagos';});
But unlike a regular mutation, it doesn’t overwrite the past state!
Deep DiveHow does Immer work? Show DetailsThe draft provided by Immer is a special type of object, called a Proxy, that “records” what you do with it. This is why you can mutate it freely as much as you like! Under the hood, Immer figures out which parts of the draft have been changed, and produces a completely new object that contains your edits.
To try Immer:

Run npm install use-immer to add Immer as a dependency
Then replace import { useState } from 'react' with import { useImmer } from 'use-immer'

Here is the above example converted to Immer:
App.jspackage.jsonApp.js ResetForkimport { useImmer } from 'use-immer';

```
export default function Form() {
  const [person, updatePerson] = useImmer({
    name: 'Niki de Saint Phalle',
    artwork: {
```

```
    title: 'Blue Nana',
    city: 'Hamburg',
    image: 'https://i.imgur.com/Sd1AgUOm.jpg',
  }
});

function handleNameChange(e) {
  updatePerson(draft => {
    draft.name = e.target.value;
  });
}

function handleTitleChange(e) {
  updatePerson(draft => {
    draft.artwork.title = e.target.value;
  });
}

function handleCityChange(e) {
  updatePerson(draft => {
    draft.artwork.city = e.target.value;
  });
}

function handleImageChange(e) {
  updatePerson(draft => {
    draft.artwork.image = e.target.value;
  });
}

return (
  <>
    <label>
      Name:
      <input
        value={person.name}
        onChange={handleNameChange}
      />
    </label>
    <label>
      Title:
      <input
        value={person.artwork.title}
        onChange={handleTitleChange}
      />
    </label>
```

```
      <label>
        City:
        <input
          value={person.artwork.city}
          onChange={handleCityChange}
        />
      </label>
      <label>
        Image:
        <input
          value={person.artwork.image}
          onChange={handleImageChange}
        />
      </label>
      <p>
        <i>{person.artwork.title}</i>
        {' by '}
        {person.name}
        <br />
        (located in {person.artwork.city})
      </p>
      <img
        src={person.artwork.image}
        alt={person.artwork.title}
      />
    </>
  );
}
```

Show more

Notice how much more concise the event handlers have become. You can mix and match useState and useImmer in a single component as much as you like. Immer is a great way to keep the update handlers concise, especially if there's nesting in your state, and copying objects leads to repetitive code.

Deep DiveWhy is mutating state not recommended in React? Show DetailsThere are a few reasons:

Debugging: If you use console.log and don't mutate state, your past logs won't get clobbered by the more recent state changes. So you can clearly see how state has changed between renders.

Optimizations: Common React optimization strategies rely on skipping work if previous props or state are the same as the next ones. If you never mutate state, it is very fast to check whether there were any changes. If prevObj === obj, you can be sure that nothing could have changed inside of it.

New Features: The new React features we're building rely on state being treated like a snapshot. If you're mutating past versions of state, that may prevent you from using the new features.

Requirement Changes: Some application features, like implementing Undo/Redo, showing a history of changes, or letting the user reset a form to earlier values, are easier to do when nothing is mutated. This is because you can keep past copies of state in memory, and reuse them when appropriate. If you start with a mutative approach, features like this can be difficult to add later on.
Simpler Implementation: Because React does not rely on mutation, it does not need to do anything special with your objects. It does not need to hijack their properties, always wrap them into Proxies, or do other work at initialization as many “reactive” solutions do. This is also why React lets you put any object into state—no matter how large—without additional performance or correctness pitfalls.
In practice, you can often “get away” with mutating state in React, but we strongly advise you not to do that so that you can use new React features developed with this approach in mind. Future contributors and perhaps even your future self will thank you!
Recap
Treat all state in React as immutable.
When you store objects in state, mutating them will not trigger renders and will change the state in previous render “snapshots”.
Instead of mutating an object, create a new version of it, and trigger a re-render by setting state to it.
You can use the {...obj, something: 'newValue'} object spread syntax to create copies of objects.
Spread syntax is shallow: it only copies one level deep.
To update a nested object, you need to create copies all the way up from the place you’re updating.
To reduce repetitive copying code, use Immer.

Try out some challenges1. Fix incorrect state updates 2. Find and fix the mutation 3. Update an object with Immer Challenge 1 of 3: Fix incorrect state updates This form has a few bugs. Click the button that increases the score a few times. Notice that it does not increase. Then edit the first name, and notice that the score has suddenly “caught up” with your changes. Finally, edit the last name, and notice that the score has disappeared completely.Your task is to fix all of these bugs. As you fix them, explain why each of them happens.App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Scoreboard() {
  const [player, setPlayer] = useState({
    firstName: 'Ranjani',
    lastName: 'Shettar',
    score: 10,
  });

  function handlePlusClick() {
    player.score++;
  }
```

```
  function handleFirstNameChange(e) {
    setPlayer({
      ...player,
      firstName: e.target.value,
    });
  }

  function handleLastNameChange(e) {
    setPlayer({
      lastName: e.target.value
    });
  }

  return (
    <>
      <label>
        Score: <b>{player.score}</b>
        {' '}
        <button onClick={handlePlusClick}>
          +1
        </button>
      </label>
      <label>
        First name:
        <input
          value={player.firstName}
          onChange={handleFirstNameChange}
        />
      </label>
      <label>
        Last name:
        <input
          value={player.lastName}
          onChange={handleLastNameChange}
        />
      </label>
    </>
  );
}
```

Show more Show solutionNext ChallengePreviousQueueing a Series of State UpdatesNextUpdating Arrays in State

Learn ReactAdding InteractivityUpdating Arrays in StateArrays are mutable in JavaScript, but you should treat them as immutable when you store them in state. Just like with objects, when you want to update an array stored in state, you need to create a new one (or make a copy of an existing one), and then set state to use the new array.

You will learn
How to add, remove, or change items in an array in React state
How to update an object inside of an array
How to make array copying less repetitive with Immer

Updating arrays without mutation
In JavaScript, arrays are just another kind of object. Like with objects, you should treat arrays in React state as read-only. This means that you shouldn’t reassign items inside an array like arr[0] = 'bird', and you also shouldn’t use methods that mutate the array, such as push() and pop().
Instead, every time you want to update an array, you’ll want to pass a new array to your state setting function. To do that, you can create a new array from the original array in your state by calling its non-mutating methods like filter() and map(). Then you can set your state to the resulting new array.
Here is a reference table of common array operations. When dealing with arrays inside React state, you will need to avoid the methods in the left column, and instead prefer the methods in the right column:
avoid (mutates the array)prefer (returns a new array)addingpush, unshiftconcat, [...arr] spread syntax (example)removingpop, shift, splicefilter, slice (example)replacingsplice, arr[i] = ... assignmentmap (example)sortingreverse, sortcopy the array first (example)
Alternatively, you can use Immer which lets you use methods from both columns.
PitfallUnfortunately, slice and splice are named similarly but are very different:
slice lets you copy an array or a part of it.
splice mutates the array (to insert or delete items).
In React, you will be using slice (no p!) a lot more often because you don’t want to mutate objects or arrays in state. Updating Objects explains what mutation is and why it’s not recommended for state.
Adding to an array
push() will mutate an array, which you don’t want:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let nextId = 0;

export default function List() {
  const [name, setName] = useState('');
  const [artists, setArtists] = useState([]);

  return (
    <>
      <h1>Inspiring sculptors:</h1>
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
      <button onClick={() => {
        artists.push({
```

```
          id: nextId++,
          name: name,
        });
      }}>Add</button>
      <ul>
        {artists.map(artist => (
          <li key={artist.id}>{artist.name}</li>
        ))}
      </ul>
    </>
  );
}
```

Show more
Instead, create a new array which contains the existing items and a new item at the end. There are multiple ways to do this, but the easiest one is to use the ... array spread syntax:
setArtists(// Replace the state [// with a new array   ...artists, // that contains all the old items   { id: nextId++, name: name } // and one new item at the end]);
Now it works correctly:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let nextId = 0;

export default function List() {
  const [name, setName] = useState('');
  const [artists, setArtists] = useState([]);

  return (
    <>
      <h1>Inspiring sculptors:</h1>
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
      <button onClick={() => {
        setArtists([
          ...artists,
          { id: nextId++, name: name }
        ]);
      }}>Add</button>
      <ul>
        {artists.map(artist => (
          <li key={artist.id}>{artist.name}</li>
        ))}
      </ul>
```

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

Show more

The array spread syntax also lets you prepend an item by placing it before the original ...artists:

```
setArtists([  { id: nextId++, name: name },  ...artists // Put old items at the end]);
```

In this way, spread can do the job of both push() by adding to the end of an array and unshift() by adding to the beginning of an array. Try it in the sandbox above!

Removing from an array

The easiest way to remove an item from an array is to filter it out. In other words, you will produce a new array that will not contain that item. To do this, use the filter method, for example:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let initialArtists = [
  { id: 0, name: 'Marta Colvin Andrade' },
  { id: 1, name: 'Lamidi Olonade Fakeye'},
  { id: 2, name: 'Louise Nevelson'},
];

export default function List() {
  const [artists, setArtists] = useState(
    initialArtists
  );

  return (
    <>
      <h1>Inspiring sculptors:</h1>
      <ul>
        {artists.map(artist => (
          <li key={artist.id}>
            {artist.name}{' '}
            <button onClick={() => {
              setArtists(
                artists.filter(a =>
                  a.id !== artist.id
                )
              );
            }}>
              Delete
            </button>
          </li>
        ))}
      </ul>
```

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

Show more

Click the “Delete” button a few times, and look at its click handler.

```
setArtists(  artists.filter(a => a.id !== artist.id));
```

Here, artists.filter(a => a.id !== artist.id) means “create an array that consists of those artists whose IDs are different from artist.id”. In other words, each artist’s “Delete” button will filter that artist out of the array, and then request a re-render with the resulting array. Note that filter does not modify the original array.

Transforming an array

If you want to change some or all items of the array, you can use map() to create a new array. The function you will pass to map can decide what to do with each item, based on its data or its index (or both).

In this example, an array holds coordinates of two circles and a square. When you press the button, it moves only the circles down by 50 pixels. It does this by producing a new array of data using map():

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let initialShapes = [
  { id: 0, type: 'circle', x: 50, y: 100 },
  { id: 1, type: 'square', x: 150, y: 100 },
  { id: 2, type: 'circle', x: 250, y: 100 },
];

export default function ShapeEditor() {
  const [shapes, setShapes] = useState(
    initialShapes
  );

  function handleClick() {
    const nextShapes = shapes.map(shape => {
      if (shape.type === 'square') {
        // No change
        return shape;
      } else {
        // Return a new circle 50px below
        return {
          ...shape,
          y: shape.y + 50,
        };
      }
    });
    // Re-render with the new array
    setShapes(nextShapes);
```

```
  }

  return (
    <>
      <button onClick={handleClick}>
        Move circles down!
      </button>
      {shapes.map(shape => (
        <div
          key={shape.id}
          style={{
          background: 'purple',
          position: 'absolute',
          left: shape.x,
          top: shape.y,
          borderRadius:
            shape.type === 'circle'
              ? '50%' : '',
          width: 20,
          height: 20,
        }} />
      ))}
    </>
  );
}
```

Show more

Replacing items in an array

It is particularly common to want to replace one or more items in an array. Assignments like arr[0] = 'bird' are mutating the original array, so instead you'll want to use map for this as well.

To replace an item, create a new array with map. Inside your map call, you will receive the item index as the second argument. Use it to decide whether to return the original item (the first argument) or something else:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let initialCounters = [
  0, 0, 0
];

export default function CounterList() {
  const [counters, setCounters] = useState(
    initialCounters
  );

  function handleIncrementClick(index) {
```

```
    const nextCounters = counters.map((c, i) => {
      if (i === index) {
        // Increment the clicked counter
        return c + 1;
      } else {
        // The rest haven't changed
        return c;
      }
    });
    setCounters(nextCounters);
  }

  return (
    <ul>
      {counters.map((counter, i) => (
        <li key={i}>
          {counter}
          <button onClick={() => {
            handleIncrementClick(i);
          }}>+1</button>
        </li>
      ))}
    </ul>
  );
}
```

Show more

Inserting into an array

Sometimes, you may want to insert an item at a particular position that's neither at the beginning nor at the end. To do this, you can use the ... array spread syntax together with the slice() method. The slice() method lets you cut a "slice" of the array. To insert an item, you will create an array that spreads the slice before the insertion point, then the new item, and then the rest of the original array.

In this example, the Insert button always inserts at the index 1:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let nextId = 3;
const initialArtists = [
  { id: 0, name: 'Marta Colvin Andrade' },
  { id: 1, name: 'Lamidi Olonade Fakeye'},
  { id: 2, name: 'Louise Nevelson'},
];

export default function List() {
  const [name, setName] = useState('');
  const [artists, setArtists] = useState(
```

```
    initialArtists
  );

  function handleClick() {
    const insertAt = 1; // Could be any index
    const nextArtists = [
      // Items before the insertion point:
      ...artists.slice(0, insertAt),
      // New item:
      { id: nextId++, name: name },
      // Items after the insertion point:
      ...artists.slice(insertAt)
    ];
    setArtists(nextArtists);
    setName('');
  }

  return (
    <>
      <h1>Inspiring sculptors:</h1>
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
      <button onClick={handleClick}>
        Insert
      </button>
      <ul>
        {artists.map(artist => (
          <li key={artist.id}>{artist.name}</li>
        ))}
      </ul>
    </>
  );
}
```

Show more

Making other changes to an array

There are some things you can’t do with the spread syntax and non-mutating methods like map() and filter() alone. For example, you may want to reverse or sort an array. The JavaScript reverse() and sort() methods are mutating the original array, so you can’t use them directly.

However, you can copy the array first, and then make changes to it.

For example:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let nextId = 3;
const initialList = [
  { id: 0, title: 'Big Bellies' },
  { id: 1, title: 'Lunar Landscape' },
  { id: 2, title: 'Terracotta Army' },
];

export default function List() {
  const [list, setList] = useState(initialList);

  function handleClick() {
    const nextList = [...list];
    nextList.reverse();
    setList(nextList);
  }

  return (
    <>
      <button onClick={handleClick}>
        Reverse
      </button>
      <ul>
        {list.map(artwork => (
          <li key={artwork.id}>{artwork.title}</li>
        ))}
      </ul>
    </>
  );
}
```

Show more

Here, you use the [...list] spread syntax to create a copy of the original array first. Now that you have a copy, you can use mutating methods like nextList.reverse() or nextList.sort(), or even assign individual items with nextList[0] = "something".

However, even if you copy an array, you can’t mutate existing items inside of it directly. This is because copying is shallow—the new array will contain the same items as the original one. So if you modify an object inside the copied array, you are mutating the existing state. For example, code like this is a problem.

const nextList = [...list];nextList[0].seen = true; // Problem: mutates list[0]setList(nextList);

Although nextList and list are two different arrays, nextList[0] and list[0] point to the same object. So by changing nextList[0].seen, you are also changing list[0].seen. This is a state mutation, which you should avoid! You can solve this issue in a similar way to updating nested JavaScript objects—by copying individual items you want to change instead of mutating them. Here’s how.

Updating objects inside arrays

Objects are not really located “inside” arrays. They might appear to be “inside” in code,

but each object in an array is a separate value, to which the array “points”. This is why you need to be careful when changing nested fields like list[0]. Another person’s artwork list may point to the same element of the array!
When updating nested state, you need to create copies from the point where you want to update, and all the way up to the top level. Let’s see how this works.
In this example, two separate artwork lists have the same initial state. They are supposed to be isolated, but because of a mutation, their state is accidentally shared, and checking a box in one list affects the other list:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let nextId = 3;
const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
  const [myList, setMyList] = useState(initialList);
  const [yourList, setYourList] = useState(
    initialList
  );

  function handleToggleMyList(artworkId, nextSeen) {
    const myNextList = [...myList];
    const artwork = myNextList.find(
      a => a.id === artworkId
    );
    artwork.seen = nextSeen;
    setMyList(myNextList);
  }

  function handleToggleYourList(artworkId, nextSeen) {
    const yourNextList = [...yourList];
    const artwork = yourNextList.find(
      a => a.id === artworkId
    );
    artwork.seen = nextSeen;
    setYourList(yourNextList);
  }

  return (
    <>
      <h1>Art Bucket List</h1>
      <h2>My list of art to see:</h2>
      <ItemList
```

```
        artworks={myList}
        onToggle={handleToggleMyList} />
      <h2>Your list of art to see:</h2>
      <ItemList
        artworks={yourList}
        onToggle={handleToggleYourList} />
    </>
  );
}

function ItemList({ artworks, onToggle }) {
  return (
    <ul>
      {artworks.map(artwork => (
        <li key={artwork.id}>
          <label>
            <input
              type="checkbox"
              checked={artwork.seen}
              onChange={e => {
                onToggle(
                  artwork.id,
                  e.target.checked
                );
              }}
            />
            {artwork.title}
          </label>
        </li>
      ))}
    </ul>
  );
}
```

Show more

The problem is in code like this:

```
const myNextList = [...myList];const artwork = myNextList.find(a => a.id === artworkId);artwork.seen = nextSeen; // Problem: mutates an existing itemsetMyList(myNextList);
```

Although the myNextList array itself is new, the items themselves are the same as in the original myList array. So changing artwork.seen changes the original artwork item. That artwork item is also in yourList, which causes the bug. Bugs like this can be difficult to think about, but thankfully they disappear if you avoid mutating state.

You can use map to substitute an old item with its updated version without mutation.

```
setMyList(myList.map(artwork => {  if (artwork.id === artworkId) {    // Create a *new* object with changes    return { ...artwork, seen: nextSeen };  } else {    // No changes
```

```
return artwork;  }}));
```
Here, ... is the object spread syntax used to create a copy of an object.
With this approach, none of the existing state items are being mutated, and the bug is fixed:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
let nextId = 3;
const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
  const [myList, setMyList] = useState(initialList);
  const [yourList, setYourList] = useState(
    initialList
  );

  function handleToggleMyList(artworkId, nextSeen) {
    setMyList(myList.map(artwork => {
      if (artwork.id === artworkId) {
        // Create a *new* object with changes
        return { ...artwork, seen: nextSeen };
      } else {
        // No changes
        return artwork;
      }
    }));
  }

  function handleToggleYourList(artworkId, nextSeen) {
    setYourList(yourList.map(artwork => {
      if (artwork.id === artworkId) {
        // Create a *new* object with changes
        return { ...artwork, seen: nextSeen };
      } else {
        // No changes
        return artwork;
      }
    }));
  }

  return (
    <>
      <h1>Art Bucket List</h1>
```

```
      <h2>My list of art to see:</h2>
      <ItemList
        artworks={myList}
        onToggle={handleToggleMyList} />
      <h2>Your list of art to see:</h2>
      <ItemList
        artworks={yourList}
        onToggle={handleToggleYourList} />
    </>
  );
}

function ItemList({ artworks, onToggle }) {
  return (
    <ul>
      {artworks.map(artwork => (
        <li key={artwork.id}>
          <label>
            <input
              type="checkbox"
              checked={artwork.seen}
              onChange={e => {
                onToggle(
                  artwork.id,
                  e.target.checked
                );
              }}
            />
            {artwork.title}
          </label>
        </li>
      ))}
    </ul>
  );
}
```

Show more

In general, you should only mutate objects that you have just created. If you were inserting a new artwork, you could mutate it, but if you’re dealing with something that’s already in state, you need to make a copy.

Write concise update logic with Immer

Updating nested arrays without mutation can get a little bit repetitive. Just as with objects:

Generally, you shouldn’t need to update state more than a couple of levels deep. If your state objects are very deep, you might want to restructure them differently so that they

are flat.
If you don’t want to change your state structure, you might prefer to use Immer, which lets you write using the convenient but mutating syntax and takes care of producing the copies for you.

Here is the Art Bucket List example rewritten with Immer:
App.jspackage.jsonApp.js ResetForkimport { useState } from 'react';
import { useImmer } from 'use-immer';

```
let nextId = 3;
const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
  const [myList, updateMyList] = useImmer(
    initialList
  );
  const [yourList, updateYourList] = useImmer(
    initialList
  );

  function handleToggleMyList(id, nextSeen) {
    updateMyList(draft => {
      const artwork = draft.find(a =>
        a.id === id
      );
      artwork.seen = nextSeen;
    });
  }

  function handleToggleYourList(artworkId, nextSeen) {
    updateYourList(draft => {
      const artwork = draft.find(a =>
        a.id === artworkId
      );
      artwork.seen = nextSeen;
    });
  }

  return (
    <>
      <h1>Art Bucket List</h1>
      <h2>My list of art to see:</h2>
```

```
      <ItemList
        artworks={myList}
        onToggle={handleToggleMyList} />
      <h2>Your list of art to see:</h2>
      <ItemList
        artworks={yourList}
        onToggle={handleToggleYourList} />
    </>
  );
}

function ItemList({ artworks, onToggle }) {
  return (
    <ul>
      {artworks.map(artwork => (
        <li key={artwork.id}>
          <label>
            <input
              type="checkbox"
              checked={artwork.seen}
              onChange={e => {
                onToggle(
                  artwork.id,
                  e.target.checked
                );
              }}
            />
            {artwork.title}
          </label>
        </li>
      ))}
    </ul>
  );
}
```

Show more

Note how with Immer, mutation like artwork.seen = nextSeen is now okay:

```
updateMyTodos(draft => {  const artwork = draft.find(a => a.id === artworkId);
artwork.seen = nextSeen;});
```

This is because you're not mutating the original state, but you're mutating a special draft object provided by Immer. Similarly, you can apply mutating methods like push() and pop() to the content of the draft.

Behind the scenes, Immer always constructs the next state from scratch according to the changes that you've done to the draft. This keeps your event handlers very concise without ever mutating state.

Recap

You can put arrays into state, but you can’t change them.
Instead of mutating an array, create a new version of it, and update the state to it.
You can use the [...arr, newItem] array spread syntax to create arrays with new items.
You can use filter() and map() to create new arrays with filtered or transformed items.
You can use Immer to keep your code concise.

Try out some challenges1. Update an item in the shopping cart 2. Remove an item from the shopping cart 3. Fix the mutations using non-mutative methods 4. Fix the mutations using Immer Challenge 1 of 4: Update an item in the shopping cart Fill in the handleIncreaseClick logic so that pressing ”+” increases the corresponding number:App.jsApp.js Download ResetForkimport { useState } from 'react';

```
const initialProducts = [{
  id: 0,
  name: 'Baklava',
  count: 1,
}, {
  id: 1,
  name: 'Cheese',
  count: 5,
}, {
  id: 2,
  name: 'Spaghetti',
  count: 2,
}];

export default function ShoppingCart() {
  const [
    products,
    setProducts
  ] = useState(initialProducts)

  function handleIncreaseClick(productId) {

  }

  return (
    <ul>
      {products.map(product => (
        <li key={product.id}>
          {product.name}
          {' '}
          (<b>{product.count}</b>)
          <button onClick={() => {
            handleIncreaseClick(product.id);
          }}>
```

```
          +
        </button>
      </li>
    ))}
  </ul>
 );
}
```

Show more Show solutionNext ChallengePreviousUpdating Objects in StateNextManaging State

Learn ReactManaging StateIntermediateAs your application grows, it helps to be more intentional about how your state is organized and how the data flows between your components. Redundant or duplicate state is a common source of bugs. In this chapter, you'll learn how to structure your state well, how to keep your state update logic maintainable, and how to share state between distant components.

In this chapter

How to think about UI changes as state changes
How to structure state well
How to "lift state up" to share it between components
How to control whether the state gets preserved or reset
How to consolidate complex state logic in a function
How to pass information without "prop drilling"
How to scale state management as your app grows

Reacting to input with state

With React, you won't modify the UI from code directly. For example, you won't write commands like "disable the button", "enable the button", "show the success message", etc. Instead, you will describe the UI you want to see for the different visual states of your component ("initial state", "typing state", "success state"), and then trigger the state changes in response to user input. This is similar to how designers think about UI.

Here is a quiz form built using React. Note how it uses the status state variable to determine whether to enable or disable the submit button, and whether to show the success message instead.

App.jsApp.js Download ResetFork991234567891011121314151617181920212223242526272829303132333435 36import { useState } from 'react';export default function Form() { const [answer, setAnswer] = useState(''); const [error, setError] = useState(null); const [status, setStatus] = useState('typing'); if (status === 'success') {   return <h1>That's right!</h1> } async function handleSubmit(e) {   e.preventDefault();   setStatus('submitting');   try {   await submitForm(answer);   setStatus('success');   } catch (err) {   setStatus('typing');   setError(err);   } } function handleTextareaChange(e) {   setAnswer(e.target.value); } return (<>   <h2>City quiz</h2>   In which city is there a billboard that turns air into drinkable water?   <form onSubmit={handleSubmit}>   <textarea value={answer}Show more

Ready to learn this topic?Read Reacting to Input with State to learn how to approach interactions with a state-driven mindset.Read More

Choosing the state structure
Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs. The most important principle is that state shouldn’t contain redundant or duplicated information. If there’s unnecessary state, it’s easy to forget to update it, and introduce bugs!
For example, this form has a redundant fullName state variable:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');
  const [fullName, setFullName] = useState('');

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
    setFullName(e.target.value + ' ' + lastName);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
    setFullName(firstName + ' ' + e.target.value);
  }

  return (
    <>
      <h2>Let’s check you in</h2>
      <label>
        First name:{' '}
        <input
          value={firstName}
          onChange={handleFirstNameChange}
        />
      </label>
      <label>
        Last name:{' '}
        <input
          value={lastName}
          onChange={handleLastNameChange}
        />
      </label>
      <p>
        Your ticket will be issued to: <b>{fullName}</b>
      </p>
    </>
  );
}
```

Show more
You can remove it and simplify the code by calculating fullName while the component is rendering:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');

  const fullName = firstName + ' ' + lastName;

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
  }

  return (
    <>
      <h2>Let’s check you in</h2>
      <label>
        First name:{' '}
        <input
          value={firstName}
          onChange={handleFirstNameChange}
        />
      </label>
      <label>
        Last name:{' '}
        <input
          value={lastName}
          onChange={handleLastNameChange}
        />
      </label>
      <p>
        Your ticket will be issued to: <b>{fullName}</b>
      </p>
    </>
  );
}
```

Show more
This might seem like a small change, but many bugs in React apps are fixed this way.

Ready to learn this topic?Read Choosing the State Structure to learn how to design the state shape to avoid bugs.Read More

Sharing state between components

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as “lifting state up”, and it’s one of the most common things you will do writing React code.

In this example, only one panel should be active at a time. To achieve this, instead of keeping the active state inside each individual panel, the parent component holds the state and specifies the props for its children.

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Accordion() {
  const [activeIndex, setActiveIndex] = useState(0);
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel
        title="About"
        isActive={activeIndex === 0}
        onShow={() => setActiveIndex(0)}
      >
        With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
      </Panel>
      <Panel
        title="Etymology"
        isActive={activeIndex === 1}
        onShow={() => setActiveIndex(1)}
      >
        The name comes from <span lang="kk-KZ">C ;CÄ0</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
      </Panel>
    </>
  );
}

function Panel({
  title,
  children,
  isActive,
  onShow
}) {
```

```
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={onShow}>
          Show
        </button>
      )}
    </section>
  );
}
```

Show more

Ready to learn this topic?Read Sharing State Between Components to learn how to lift state up and keep components in sync.Read More

Preserving and resetting state

When you re-render a component, React needs to decide which parts of the tree to keep (and update), and which parts to discard or re-create from scratch. In most cases, React's automatic behavior works well enough. By default, React preserves the parts of the tree that "match up" with the previously rendered component tree.

However, sometimes this is not what you want. In this chat app, typing a message and then switching the recipient does not reset the input. This can make the user accidentally send a message to the wrong person:

App.jsContactList.jsChat.jsApp.js ResetForkimport { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

```
export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
      <Chat contact={to} />
    </div>
  )
}

const contacts = [
  { name: 'Taylor', email: 'taylor@mail.com' },
  { name: 'Alice', email: 'alice@mail.com' },
```

```
  { name: 'Bob', email: 'bob@mail.com' }
];
```

Show more

React lets you override the default behavior, and force a component to reset its state by passing it a different key, like <Chat key={email} />. This tells React that if the recipient is different, it should be considered a different Chat component that needs to be re-created from scratch with the new data (and UI like inputs). Now switching between the recipients resets the input field—even though you render the same component.

App.jsContactList.jsChat.jsApp.js ResetForkimport { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

```
export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
      <Chat key={to.email} contact={to} />
    </div>
  )
}

const contacts = [
  { name: 'Taylor', email: 'taylor@mail.com' },
  { name: 'Alice', email: 'alice@mail.com' },
  { name: 'Bob', email: 'bob@mail.com' }
];
```

Show more

Ready to learn this topic?Read Preserving and Resetting State to learn the lifetime of state and how to control it.Read More

Extracting state logic into a reducer

Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called “reducer”. Your event handlers become concise because they only specify the user “actions”. At the bottom of the file, the reducer function specifies how the state should update in response to each action!

App.jsApp.js ResetForkimport { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

```
export default function TaskApp() {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

  function handleAddTask(text) {
    dispatch({
      type: 'added',
      id: nextId++,
      text: text,
    });
  }

  function handleChangeTask(task) {
    dispatch({
      type: 'changed',
      task: task
    });
  }

  function handleDeleteTask(taskId) {
    dispatch({
      type: 'deleted',
      id: taskId
    });
  }

  return (
    <>
      <h1>Prague itinerary</h1>
      <AddTask
        onAddTask={handleAddTask}
      />
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </>
  );
}

function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
```

```
      return [...tasks, {
        id: action.id,
        text: action.text,
        done: false
      }];
    }
    case 'changed': {
      return tasks.map(t => {
        if (t.id === action.task.id) {
          return action.task;
        } else {
          return t;
        }
      });
    }
    case 'deleted': {
      return tasks.filter(t => t.id !== action.id);
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

let nextId = 3;
const initialTasks = [
  { id: 0, text: 'Visit Kafka Museum', done: true },
  { id: 1, text: 'Watch a puppet show', done: false },
  { id: 2, text: 'Lennon Wall pic', done: false }
];
```

Show more

Ready to learn this topic?Read Extracting State Logic into a Reducer to learn how to consolidate logic in the reducer function.Read More

Passing data deeply with context

Usually, you will pass information from a parent component to a child component via props. But passing props can become inconvenient if you need to pass some prop through many components, or if many components need the same information. Context lets the parent component make some information available to any component in the tree below it—no matter how deep it is—without passing it explicitly through props.

Here, the Heading component determines its heading level by “asking” the closest Section for its level. Each Section tracks its own level by asking the parent Section and adding one to it. Every Section provides information to all components below it without passing props—it does that through context.

App.jsSection.jsHeading.jsLevelContext.jsApp.js ResetForkimport Heading from './Heading.js';

```
import Section from './Section.js';

export default function Page() {
  return (
    <Section>
      <Heading>Title</Heading>
      <Section>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Section>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Section>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
          </Section>
        </Section>
      </Section>
    </Section>
  );
}
```

Show more

Ready to learn this topic?Read Passing Data Deeply with Context to learn about using context as an alternative to passing props.Read More

Scaling up with reducer and context

Reducers let you consolidate a component's state update logic. Context lets you pass information deep down to other components. You can combine reducers and context together to manage state of a complex screen.

With this approach, a parent component with complex state manages it with a reducer. Other components anywhere deep in the tree can read its state via context. They can also dispatch actions to update that state.

App.jsTasksContext.jsAddTask.jsTaskList.jsApp.js ResetForkimport AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksProvider } from './TasksContext.js';

```
export default function TaskApp() {
  return (
    <TasksProvider>
      <h1>Day off in Kyoto</h1>
      <AddTask />
      <TaskList />
```

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

Ready to learn this topic?Read Scaling Up with Reducer and Context to learn how state management scales in a growing app.Read More
What's next?
Head over to Reacting to Input with State to start reading this chapter page by page!
Or, if you're already familiar with these topics, why not read about Escape Hatches?
PreviousUpdating Arrays in StateNextReacting to Input with State
Learn ReactManaging StateReacting to Input with StateReact provides a declarative way to manipulate the UI. Instead of manipulating individual pieces of the UI directly, you describe the different states that your component can be in, and switch between them in response to the user input. This is similar to how designers think about the UI.
You will learn
How declarative UI programming differs from imperative UI programming
How to enumerate the different visual states your component can be in
How to trigger the changes between the different visual states from code

How declarative UI compares to imperative
When you design UI interactions, you probably think about how the UI changes in response to user actions. Consider a form that lets the user submit an answer:

When you type something into the form, the "Submit" button becomes enabled.
When you press "Submit", both the form and the button become disabled, and a spinner appears.
If the network request succeeds, the form gets hidden, and the "Thank you" message appears.
If the network request fails, an error message appears, and the form becomes enabled again.

In imperative programming, the above corresponds directly to how you implement interaction. You have to write the exact instructions to manipulate the UI depending on what just happened. Here's another way to think about this: imagine riding next to someone in a car and telling them turn by turn where to go.
Illustrated by Rachel Lee Nabors
They don't know where you want to go, they just follow your commands. (And if you get the directions wrong, you end up in the wrong place!) It's called imperative because you have to "command" each element, from the spinner to the button, telling the computer how to update the UI.
In this example of imperative UI programming, the form is built without React. It only uses the browser DOM:
index.jsindex.htmlindex.js ResetForkasync function handleFormSubmit(e) {
  e.preventDefault();
  disable(textarea);

```
  disable(button);
  show(loadingMessage);
  hide(errorMessage);
  try {
    await submitForm(textarea.value);
    show(successMessage);
    hide(form);
  } catch (err) {
    show(errorMessage);
    errorMessage.textContent = err.message;
  } finally {
    hide(loadingMessage);
    enable(textarea);
    enable(button);
  }
}

function handleTextareaChange() {
  if (textarea.value.length === 0) {
    disable(button);
  } else {
    enable(button);
  }
}

function hide(el) {
  el.style.display = 'none';
}

function show(el) {
  el.style.display = '';
}

function enable(el) {
  el.disabled = false;
}

function disable(el) {
  el.disabled = true;
}

function submitForm(answer) {
  // Pretend it's hitting the network.
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (answer.toLowerCase() == 'istanbul') {
```

```
        resolve();
      } else {
        reject(new Error('Good guess but a wrong answer. Try again!'));
      }
    }, 1500);
  });
}

let form = document.getElementById('form');
let textarea = document.getElementById('textarea');
let button = document.getElementById('button');
let loadingMessage = document.getElementById('loading');
let errorMessage = document.getElementById('error');
let successMessage = document.getElementById('success');
form.onsubmit = handleFormSubmit;
textarea.oninput = handleTextareaChange;
```

Show more

Manipulating the UI imperatively works well enough for isolated examples, but it gets exponentially more difficult to manage in more complex systems. Imagine updating a page full of different forms like this one. Adding a new UI element or a new interaction would require carefully checking all existing code to make sure you haven't introduced a bug (for example, forgetting to show or hide something).

React was built to solve this problem.

In React, you don't directly manipulate the UI—meaning you don't enable, disable, show, or hide components directly. Instead, you declare what you want to show, and React figures out how to update the UI. Think of getting into a taxi and telling the driver where you want to go instead of telling them exactly where to turn. It's the driver's job to get you there, and they might even know some shortcuts you haven't considered!

Illustrated by Rachel Lee Nabors

Thinking about UI declaratively

You've seen how to implement a form imperatively above. To better understand how to think in React, you'll walk through reimplementing this UI in React below:

Identify your component's different visual states
Determine what triggers those state changes
Represent the state in memory using useState
Remove any non-essential state variables
Connect the event handlers to set the state

Step 1: Identify your component's different visual states

In computer science, you may hear about a "state machine" being in one of several "states". If you work with a designer, you may have seen mockups for different "visual states". React stands at the intersection of design and computer science, so both of these ideas are sources of inspiration.

First, you need to visualize all the different "states" of the UI the user might see:

Empty: Form has a disabled “Submit” button.
Typing: Form has an enabled “Submit” button.
Submitting: Form is completely disabled. Spinner is shown.
Success: “Thank you” message is shown instead of a form.
Error: Same as Typing state, but with an extra error message.

Just like a designer, you’ll want to “mock up” or create “mocks” for the different states before you add logic. For example, here is a mock for just the visual part of the form. This mock is controlled by a prop called status with a default value of 'empty':
App.jsApp.js Download ResetForkexport default function Form({
  status = 'empty'
}) {
  if (status === 'success') {
    return <h1>That's right!</h1>
  }
  return (
    <>
      <h2>City quiz</h2>

        In which city is there a billboard that turns air into drinkable water?

      <form>
        <textarea />

        <button>
          Submit
        </button>
      </form>
    </>
)
}

Show more
You could call that prop anything you like, the naming is not important. Try editing status = 'empty' to status = 'success' to see the success message appear. Mocking lets you quickly iterate on the UI before you wire up any logic. Here is a more fleshed out prototype of the same component, still “controlled” by the status prop:
App.jsApp.js Download ResetForkexport default function Form({
  // Try 'submitting', 'error', 'success':
  status = 'empty'
}) {
  if (status === 'success') {
    return <h1>That's right!</h1>
  }
  return (

```
    <>
      <h2>City quiz</h2>
      <p>
        In which city is there a billboard that turns air into drinkable water?
      </p>
      <form>
        <textarea disabled={
          status === 'submitting'
        } />
        <br />
        <button disabled={
          status === 'empty' ||
          status === 'submitting'
        }>
          Submit
        </button>
        {status === 'error' &&
          <p className="Error">
            Good guess but a wrong answer. Try again!
          </p>
        }
      </form>
      </>
  );
}
```

Show more

Deep DiveDisplaying many visual states at once Show DetailsIf a component has a lot of visual states, it can be convenient to show them all on one page:App.jsForm.jsApp.js ResetForkimport Form from './Form.js';

```
let statuses = [
  'empty',
  'typing',
  'submitting',
  'success',
  'error',
];

export default function App() {
  return (
    <>
      {statuses.map(status => (
        <section key={status}>
          <h4>Form ({status}):</h4>
          <Form status={status} />
```

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

Show morePages like this are often called “living styleguides” or “storybooks”.
Step 2: Determine what triggers those state changes
You can trigger state updates in response to two kinds of inputs:

Human inputs, like clicking a button, typing in a field, navigating a link.
Computer inputs, like a network response arriving, a timeout completing, an image loading.

Human inputsComputer inputsIllustrated by Rachel Lee Nabors
In both cases, you must set state variables to update the UI. For the form you’re developing, you will need to change state in response to a few different inputs:

Changing the text input (human) should switch it from the Empty state to the Typing state or back, depending on whether the text box is empty or not.
Clicking the Submit button (human) should switch it to the Submitting state.
Successful network response (computer) should switch it to the Success state.
Failed network response (computer) should switch it to the Error state with the matching error message.

NoteNotice that human inputs often require event handlers!
To help visualize this flow, try drawing each state on paper as a labeled circle, and each change between two states as an arrow. You can sketch out many flows this way and sort out bugs long before implementation.
Form states
Step 3: Represent the state in memory with useState
Next you’ll need to represent the visual states of your component in memory with useState. Simplicity is key: each piece of state is a “moving piece”, and you want as few “moving pieces” as possible. More complexity leads to more bugs!
Start with the state that absolutely must be there. For example, you’ll need to store the answer for the input, and the error (if it exists) to store the last error:
```
const [answer, setAnswer] = useState('');const [error, setError] = useState(null);
```
Then, you’ll need a state variable representing which one of the visual states that you want to display. There’s usually more than a single way to represent that in memory, so you’ll need to experiment with it.
If you struggle to think of the best way immediately, start by adding enough state that you’re definitely sure that all the possible visual states are covered:
```
const [isEmpty, setIsEmpty] = useState(true);const [isTyping, setIsTyping] = useState(false);const [isSubmitting, setIsSubmitting] = useState(false);const [isSuccess, setIsSuccess] = useState(false);const [isError, setIsError] = useState(false);
```
Your first idea likely won’t be the best, but that’s ok—refactoring state is a part of the process!

Step 4: Remove any non-essential state variables
You want to avoid duplication in the state content so you're only tracking what is essential. Spending a little time on refactoring your state structure will make your components easier to understand, reduce duplication, and avoid unintended meanings. Your goal is to prevent the cases where the state in memory doesn't represent any valid UI that you'd want a user to see. (For example, you never want to show an error message and disable the input at the same time, or the user won't be able to correct the error!)
Here are some questions you can ask about your state variables:

Does this state cause a paradox? For example, isTyping and isSubmitting can't both be true. A paradox usually means that the state is not constrained enough. There are four possible combinations of two booleans, but only three correspond to valid states. To remove the "impossible" state, you can combine these into a status that must be one of three values: 'typing', 'submitting', or 'success'.
Is the same information available in another state variable already? Another paradox: isEmpty and isTyping can't be true at the same time. By making them separate state variables, you risk them going out of sync and causing bugs. Fortunately, you can remove isEmpty and instead check answer.length === 0.
Can you get the same information from the inverse of another state variable? isError is not needed because you can check error !== null instead.

After this clean-up, you're left with 3 (down from 7!) essential state variables:
const [answer, setAnswer] = useState('');const [error, setError] = useState(null);const [status, setStatus] = useState('typing'); // 'typing', 'submitting', or 'success'
You know they are essential, because you can't remove any of them without breaking the functionality.
Deep DiveEliminating "impossible" states with a reducer Show DetailsThese three variables are a good enough representation of this form's state. However, there are still some intermediate states that don't fully make sense. For example, a non-null error doesn't make sense when status is 'success'. To model the state more precisely, you can extract it into a reducer. Reducers let you unify multiple state variables into a single object and consolidate all the related logic!
Step 5: Connect the event handlers to set state
Lastly, create event handlers that update the state. Below is the final form, with all event handlers wired up:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [answer, setAnswer] = useState('');
  const [error, setError] = useState(null);
  const [status, setStatus] = useState('typing');

  if (status === 'success') {
    return <h1>That's right!</h1>
  }
```

```
  async function handleSubmit(e) {
    e.preventDefault();
    setStatus('submitting');
    try {
      await submitForm(answer);
      setStatus('success');
    } catch (err) {
      setStatus('typing');
      setError(err);
    }
  }

  function handleTextareaChange(e) {
    setAnswer(e.target.value);
  }

  return (
    <>
      <h2>City quiz</h2>
      <p>
        In which city is there a billboard that turns air into drinkable water?
      </p>
      <form onSubmit={handleSubmit}>
        <textarea
          value={answer}
          onChange={handleTextareaChange}
          disabled={status === 'submitting'}
        />
        <br />
        <button disabled={
          answer.length === 0 ||
          status === 'submitting'
        }>
          Submit
        </button>
        {error !== null &&
          <p className="Error">
            {error.message}
          </p>
        }
      </form>
    </>
  );
}
```

```
function submitForm(answer) {
  // Pretend it's hitting the network.
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let shouldError = answer.toLowerCase() !== 'lima'
      if (shouldError) {
        reject(new Error('Good guess but a wrong answer. Try again!'));
      } else {
        resolve();
      }
    }, 1500);
  });
}
```

Show more

Although this code is longer than the original imperative example, it is much less fragile. Expressing all interactions as state changes lets you later introduce new visual states without breaking existing ones. It also lets you change what should be displayed in each state without changing the logic of the interaction itself.

Recap

Declarative programming means describing the UI for each visual state rather than micromanaging the UI (imperative).

When developing a component:

Identify all its visual states.
Determine the human and computer triggers for state changes.
Model the state with useState.
Remove non-essential state to avoid bugs and paradoxes.
Connect the event handlers to set state.

Try out some challenges1. Add and remove a CSS class 2. Profile editor 3. Refactor the imperative solution without React Challenge 1 of 3: Add and remove a CSS class Make it so that clicking on the picture removes the background--active CSS class from the outer , but adds the picture--active class to the <img>. Clicking the background again should restore the original CSS classes.Visually, you should expect that clicking on the picture removes the purple background and highlights the picture border. Clicking outside the picture highlights the background, but removes the picture border highlight.App.jsApp.js Download ResetForkexport default function Picture() {

```
  return (
    <div className="background background--active">
      <img
        className="picture"
        alt="Rainbow houses in Kampung Pelangi, Indonesia"
        src="https://i.imgur.com/5qwVYb1.jpeg"
```

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

Show solutionNext ChallengePreviousManaging StateNextChoosing the State Structure

Learn ReactManaging StateChoosing the State StructureStructuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs. Here are some tips you should consider when structuring state.

You will learn

When to use a single vs multiple state variables
What to avoid when organizing state
How to fix common issues with the state structure

Principles for structuring state

When you write a component that holds some state, you'll have to make choices about how many state variables to use and what the shape of their data should be. While it's possible to write correct programs even with a suboptimal state structure, there are a few principles that can guide you to make better choices:

Group related state. If you always update two or more state variables at the same time, consider merging them into a single state variable.
Avoid contradictions in state. When the state is structured in a way that several pieces of state may contradict and "disagree" with each other, you leave room for mistakes. Try to avoid this.
Avoid redundant state. If you can calculate some information from the component's props or its existing state variables during rendering, you should not put that information into that component's state.
Avoid duplication in state. When the same data is duplicated between multiple state variables, or within nested objects, it is difficult to keep them in sync. Reduce duplication when you can.
Avoid deeply nested state. Deeply hierarchical state is not very convenient to update. When possible, prefer to structure state in a flat way.

The goal behind these principles is to make state easy to update without introducing mistakes. Removing redundant and duplicate data from state helps ensure that all its pieces stay in sync. This is similar to how a database engineer might want to "normalize" the database structure to reduce the chance of bugs. To paraphrase Albert Einstein, "Make your state as simple as it can be—but no simpler."

Now let's see how these principles apply in action.

Group related state

You might sometimes be unsure between using a single or multiple state variables. Should you do this?

```
const [x, setX] = useState(0);const [y, setY] = useState(0);
```

Or this?
const [position, setPosition] = useState({ x: 0, y: 0 });
Technically, you can use either of these approaches. But if some two state variables always change together, it might be a good idea to unify them into a single state variable. Then you won't forget to always keep them in sync, like in this example where moving the cursor updates both coordinates of the red dot:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function MovingDot() {
  const [position, setPosition] = useState({
    x: 0,
    y: 0
  });
  return (
    <div
      onPointerMove={e => {
        setPosition({
          x: e.clientX,
          y: e.clientY
        });
      }}
      style={{
        position: 'relative',
        width: '100vw',
        height: '100vh',
      }}>
      <div style={{
        position: 'absolute',
        backgroundColor: 'red',
        borderRadius: '50%',
        transform: `translate(${position.x}px, ${position.y}px)`,
        left: -10,
        top: -10,
        width: 20,
        height: 20,
      }} />
    </div>
  )
}
```

Show more
Another case where you'll group data into an object or an array is when you don't know how many pieces of state you'll need. For example, it's helpful when you have a form where the user can add custom fields.
PitfallIf your state variable is an object, remember that you can't update only one field in it without explicitly copying the other fields. For example, you can't do setPosition({ x:

100 }) in the above example because it would not have the y property at all! Instead, if you wanted to set x alone, you would either do setPosition({ ...position, x: 100 }), or split them into two state variables and do setX(100).

Avoid contradictions in state

Here is a hotel feedback form with isSending and isSent state variables:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function FeedbackForm() {
  const [text, setText] = useState('');
  const [isSending, setIsSending] = useState(false);
  const [isSent, setIsSent] = useState(false);

  async function handleSubmit(e) {
    e.preventDefault();
    setIsSending(true);
    await sendMessage(text);
    setIsSending(false);
    setIsSent(true);
  }

  if (isSent) {
    return <h1>Thanks for feedback!</h1>
  }

  return (
    <form onSubmit={handleSubmit}>
      <p>How was your stay at The Prancing Pony?</p>
      <textarea
        disabled={isSending}
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <br />
      <button
        disabled={isSending}
        type="submit"
      >
        Send
      </button>
      {isSending && <p>Sending...</p>}
    </form>
  );
}

// Pretend to send a message.
function sendMessage(text) {
```

```
  return new Promise(resolve => {
    setTimeout(resolve, 2000);
  });
}
```

Show more

While this code works, it leaves the door open for “impossible” states. For example, if you forget to call setIsSent and setIsSending together, you may end up in a situation where both isSending and isSent are true at the same time. The more complex your component is, the harder it is to understand what happened.

Since isSending and isSent should never be true at the same time, it is better to replace them with one status state variable that may take one of three valid states: 'typing' (initial), 'sending', and 'sent':

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function FeedbackForm() {
  const [text, setText] = useState('');
  const [status, setStatus] = useState('typing');

  async function handleSubmit(e) {
    e.preventDefault();
    setStatus('sending');
    await sendMessage(text);
    setStatus('sent');
  }

  const isSending = status === 'sending';
  const isSent = status === 'sent';

  if (isSent) {
    return <h1>Thanks for feedback!</h1>
  }

  return (
    <form onSubmit={handleSubmit}>
      <p>How was your stay at The Prancing Pony?</p>
      <textarea
        disabled={isSending}
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <br />
      <button
        disabled={isSending}
        type="submit"
      >
```

```
        Send
      </button>
      {isSending && <p>Sending...</p>}
    </form>
  );
}

// Pretend to send a message.
function sendMessage(text) {
  return new Promise(resolve => {
    setTimeout(resolve, 2000);
  });
}
```

Show more
You can still declare some constants for readability:
const isSending = status === 'sending';const isSent = status === 'sent';
But they’re not state variables, so you don’t need to worry about them getting out of sync with each other.
Avoid redundant state
If you can calculate some information from the component’s props or its existing state variables during rendering, you should not put that information into that component’s state.
For example, take this form. It works, but can you find any redundant state in it?
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');
  const [fullName, setFullName] = useState('');

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
    setFullName(e.target.value + ' ' + lastName);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
    setFullName(firstName + ' ' + e.target.value);
  }

  return (
    <>
      <h2>Let’s check you in</h2>
      <label>
        First name:{' '}
```

```
        <input
          value={firstName}
          onChange={handleFirstNameChange}
        />
      </label>
      <label>
        Last name:{' '}
        <input
          value={lastName}
          onChange={handleLastNameChange}
        />
      </label>
      <p>
        Your ticket will be issued to: <b>{fullName}</b>
      </p>
    </>
  );
}
```

Show more

This form has three state variables: firstName, lastName, and fullName. However, fullName is redundant. You can always calculate fullName from firstName and lastName during render, so remove it from state.

This is how you can do it:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');

  const fullName = firstName + ' ' + lastName;

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
  }

  return (
    <>
      <h2>Let’s check you in</h2>
      <label>
        First name:{' '}
        <input
```

```
        value={firstName}
        onChange={handleFirstNameChange}
      />
    </label>
    <label>
      Last name:{' '}
      <input
        value={lastName}
        onChange={handleLastNameChange}
      />
    </label>
    <p>
      Your ticket will be issued to: <b>{fullName}</b>
    </p>
  </>
 );
}
```

Show more

Here, fullName is not a state variable. Instead, it's calculated during render:

```
const fullName = firstName + ' ' + lastName;
```

As a result, the change handlers don't need to do anything special to update it. When you call setFirstName or setLastName, you trigger a re-render, and then the next fullName will be calculated from the fresh data.

Deep DiveDon't mirror props in state Show DetailsA common example of redundant state is code like this:function Message({ messageColor }) { const [color, setColor] = useState(messageColor);Here, a color state variable is initialized to the messageColor prop. The problem is that if the parent component passes a different value of messageColor later (for example, 'red' instead of 'blue'), the color state variable would not be updated! The state is only initialized during the first render.This is why "mirroring" some prop in a state variable can lead to confusion. Instead, use the messageColor prop directly in your code. If you want to give it a shorter name, use a constant:function Message({ messageColor }) { const color = messageColor;This way it won't get out of sync with the prop passed from the parent component."Mirroring" props into state only makes sense when you want to ignore all updates for a specific prop. By convention, start the prop name with initial or default to clarify that its new values are ignored:function Message({ initialColor }) { // The `color` state variable holds the *first* value of `initialColor`. // Further changes to the `initialColor` prop are ignored. const [color, setColor] = useState(initialColor);

Avoid duplication in state

This menu list component lets you choose a single travel snack out of several:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
const initialItems = [
  { title: 'pretzels', id: 0 },
  { title: 'crispy seaweed', id: 1 },
```

```
  { title: 'granola bar', id: 2 },
];

export default function Menu() {
  const [items, setItems] = useState(initialItems);
  const [selectedItem, setSelectedItem] = useState(
    items[0]
  );

  return (
    <>
      <h2>What's your travel snack?</h2>
      <ul>
        {items.map(item => (
          <li key={item.id}>
            {item.title}
            {' '}
            <button onClick={() => {
              setSelectedItem(item);
            }}>Choose</button>
          </li>
        ))}
      </ul>
      <p>You picked {selectedItem.title}.</p>
    </>
  );
}
```

Show more

Currently, it stores the selected item as an object in the selectedItem state variable. However, this is not great: the contents of the selectedItem is the same object as one of the items inside the items list. This means that the information about the item itself is duplicated in two places.

Why is this a problem? Let’s make each item editable:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
const initialItems = [
  { title: 'pretzels', id: 0 },
  { title: 'crispy seaweed', id: 1 },
  { title: 'granola bar', id: 2 },
];

export default function Menu() {
  const [items, setItems] = useState(initialItems);
  const [selectedItem, setSelectedItem] = useState(
    items[0]
```

```
  );

  function handleItemChange(id, e) {
    setItems(items.map(item => {
      if (item.id === id) {
        return {
          ...item,
          title: e.target.value,
        };
      } else {
        return item;
      }
    }));
  }

  return (
    <>
      <h2>What's your travel snack?</h2>
      <ul>
        {items.map((item, index) => (
          <li key={item.id}>
            <input
              value={item.title}
              onChange={e => {
                handleItemChange(item.id, e)
              }}
            />
            {' '}
            <button onClick={() => {
              setSelectedItem(item);
            }}>Choose</button>
          </li>
        ))}
      </ul>
      <p>You picked {selectedItem.title}.</p>
    </>
  );
}
```

Show more

Notice how if you first click “Choose” on an item and then edit it, the input updates but the label at the bottom does not reflect the edits. This is because you have duplicated state, and you forgot to update selectedItem.

Although you could update selectedItem too, an easier fix is to remove duplication. In this example, instead of a selectedItem object (which creates a duplication with objects inside items), you hold the selectedId in state, and then get the selectedItem by

searching the items array for an item with that ID:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
const initialItems = [
  { title: 'pretzels', id: 0 },
  { title: 'crispy seaweed', id: 1 },
  { title: 'granola bar', id: 2 },
];

export default function Menu() {
  const [items, setItems] = useState(initialItems);
  const [selectedId, setSelectedId] = useState(0);

  const selectedItem = items.find(item =>
    item.id === selectedId
  );

  function handleItemChange(id, e) {
    setItems(items.map(item => {
      if (item.id === id) {
        return {
          ...item,
          title: e.target.value,
        };
      } else {
        return item;
      }
    }));
  }

  return (
    <>
      <h2>What's your travel snack?</h2>
      <ul>
        {items.map((item, index) => (
          <li key={item.id}>
            <input
              value={item.title}
              onChange={e => {
                handleItemChange(item.id, e)
              }}
            />
            {' '}
            <button onClick={() => {
              setSelectedId(item.id);
            }}>Choose</button>
```

```
          </li>
        ))}
      </ul>
      <p>You picked {selectedItem.title}.</p>
    </>
  );
}
```

Show more
(Alternatively, you may hold the selected index in state.)
The state used to be duplicated like this:

```
items = [{ id: 0, title: 'pretzels'}, ...]
selectedItem = {id: 0, title: 'pretzels'}
```

But after the change it’s like this:

```
items = [{ id: 0, title: 'pretzels'}, ...]
selectedId = 0
```

The duplication is gone, and you only keep the essential state!
Now if you edit the selected item, the message below will update immediately. This is because setItems triggers a re-render, and items.find(...) would find the item with the updated title. You didn’t need to hold the selected item in state, because only the selected ID is essential. The rest could be calculated during render.
Avoid deeply nested state
Imagine a travel plan consisting of planets, continents, and countries. You might be tempted to structure its state using nested objects and arrays, like in this example:
App.jsplaces.jsplaces.js ResetForkexport const initialTravelPlan = {

```
  id: 0,
  title: '(Root)',
  childPlaces: [{
    id: 1,
    title: 'Earth',
    childPlaces: [{
      id: 2,
      title: 'Africa',
      childPlaces: [{
        id: 3,
        title: 'Botswana',
        childPlaces: []
      }, {
        id: 4,
        title: 'Egypt',
        childPlaces: []
      }, {
```

```
      id: 5,
      title: 'Kenya',
      childPlaces: []
    }, {
      id: 6,
      title: 'Madagascar',
      childPlaces: []
    }, {
      id: 7,
      title: 'Morocco',
      childPlaces: []
    }, {
      id: 8,
      title: 'Nigeria',
      childPlaces: []
    }, {
      id: 9,
      title: 'South Africa',
      childPlaces: []
    }]
  }, {
    id: 10,
    title: 'Americas',
    childPlaces: [{
      id: 11,
      title: 'Argentina',
      childPlaces: []
    }, {
      id: 12,
      title: 'Brazil',
      childPlaces: []
    }, {
      id: 13,
      title: 'Barbados',
      childPlaces: []
    }, {
      id: 14,
      title: 'Canada',
      childPlaces: []
    }, {
      id: 15,
      title: 'Jamaica',
      childPlaces: []
    }, {
      id: 16,
      title: 'Mexico',
```

```
    childPlaces: []
  }, {
    id: 17,
    title: 'Trinidad and Tobago',
    childPlaces: []
  }, {
    id: 18,
    title: 'Venezuela',
    childPlaces: []
  }]
}, {
  id: 19,
  title: 'Asia',
  childPlaces: [{
    id: 20,
    title: 'China',
    childPlaces: []
  }, {
    id: 21,
    title: 'Hong Kong',
    childPlaces: []
  }, {
    id: 22,
    title: 'India',
    childPlaces: []
  }, {
    id: 23,
    title: 'Singapore',
    childPlaces: []
  }, {
    id: 24,
    title: 'South Korea',
    childPlaces: []
  }, {
    id: 25,
    title: 'Thailand',
    childPlaces: []
  }, {
    id: 26,
    title: 'Vietnam',
    childPlaces: []
  }]
}, {
  id: 27,
  title: 'Europe',
  childPlaces: [{
```

```
      id: 28,
      title: 'Croatia',
      childPlaces: [],
    }, {
      id: 29,
      title: 'France',
      childPlaces: [],
    }, {
      id: 30,
      title: 'Germany',
      childPlaces: [],
    }, {
      id: 31,
      title: 'Italy',
      childPlaces: [],
    }, {
      id: 32,
      title: 'Portugal',
      childPlaces: [],
    }, {
      id: 33,
      title: 'Spain',
      childPlaces: [],
    }, {
      id: 34,
      title: 'Turkey',
      childPlaces: [],
    }]
  }, {
    id: 35,
    title: 'Oceania',
    childPlaces: [{
      id: 36,
      title: 'Australia',
      childPlaces: [],
    }, {
      id: 37,
      title: 'Bora Bora (French Polynesia)',
      childPlaces: [],
    }, {
      id: 38,
      title: 'Easter Island (Chile)',
      childPlaces: [],
    }, {
      id: 39,
      title: 'Fiji',
```

```
        childPlaces: [],
      }, {
        id: 40,
        title: 'Hawaii (the USA)',
        childPlaces: [],
      }, {
        id: 41,
        title: 'New Zealand',
        childPlaces: [],
      }, {
        id: 42,
        title: 'Vanuatu',
        childPlaces: [],
      }]
    }]
  }, {
    id: 43,
    title: 'Moon',
    childPlaces: [{
      id: 44,
      title: 'Rheita',
      childPlaces: []
    }, {
      id: 45,
      title: 'Piccolomini',
      childPlaces: []
    }, {
      id: 46,
      title: 'Tycho',
      childPlaces: []
    }]
  }, {
    id: 47,
    title: 'Mars',
    childPlaces: [{
      id: 48,
      title: 'Corn Town',
      childPlaces: []
    }, {
      id: 49,
      title: 'Green Hill',
      childPlaces: []
    }]
  }]
};
```

Show more
Now let’s say you want to add a button to delete a place you’ve already visited. How would you go about it? Updating nested state involves making copies of objects all the way up from the part that changed. Deleting a deeply nested place would involve copying its entire parent place chain. Such code can be very verbose.
If the state is too nested to update easily, consider making it “flat”. Here is one way you can restructure this data. Instead of a tree-like structure where each place has an array of its child places, you can have each place hold an array of its child place IDs. Then store a mapping from each place ID to the corresponding place.
This data restructuring might remind you of seeing a database table:
App.jsplaces.jsplaces.js ResetForkexport const initialTravelPlan = {
```
  0: {
    id: 0,
    title: '(Root)',
    childIds: [1, 43, 47],
  },
  1: {
    id: 1,
    title: 'Earth',
    childIds: [2, 10, 19, 27, 35]
  },
  2: {
    id: 2,
    title: 'Africa',
    childIds: [3, 4, 5, 6 , 7, 8, 9]
  },
  3: {
    id: 3,
    title: 'Botswana',
    childIds: []
  },
  4: {
    id: 4,
    title: 'Egypt',
    childIds: []
  },
  5: {
    id: 5,
    title: 'Kenya',
    childIds: []
  },
  6: {
    id: 6,
    title: 'Madagascar',
    childIds: []
  },
```

```
7: {
  id: 7,
  title: 'Morocco',
  childIds: []
},
8: {
  id: 8,
  title: 'Nigeria',
  childIds: []
},
9: {
  id: 9,
  title: 'South Africa',
  childIds: []
},
10: {
  id: 10,
  title: 'Americas',
  childIds: [11, 12, 13, 14, 15, 16, 17, 18],
},
11: {
  id: 11,
  title: 'Argentina',
  childIds: []
},
12: {
  id: 12,
  title: 'Brazil',
  childIds: []
},
13: {
  id: 13,
  title: 'Barbados',
  childIds: []
},
14: {
  id: 14,
  title: 'Canada',
  childIds: []
},
15: {
  id: 15,
  title: 'Jamaica',
  childIds: []
},
16: {
```

```
  id: 16,
  title: 'Mexico',
  childIds: []
},
17: {
  id: 17,
  title: 'Trinidad and Tobago',
  childIds: []
},
18: {
  id: 18,
  title: 'Venezuela',
  childIds: []
},
19: {
  id: 19,
  title: 'Asia',
  childIds: [20, 21, 22, 23, 24, 25, 26],
},
20: {
  id: 20,
  title: 'China',
  childIds: []
},
21: {
  id: 21,
  title: 'Hong Kong',
  childIds: []
},
22: {
  id: 22,
  title: 'India',
  childIds: []
},
23: {
  id: 23,
  title: 'Singapore',
  childIds: []
},
24: {
  id: 24,
  title: 'South Korea',
  childIds: []
},
25: {
  id: 25,
```

```
  title: 'Thailand',
  childIds: []
},
26: {
  id: 26,
  title: 'Vietnam',
  childIds: []
},
27: {
  id: 27,
  title: 'Europe',
  childIds: [28, 29, 30, 31, 32, 33, 34],
},
28: {
  id: 28,
  title: 'Croatia',
  childIds: []
},
29: {
  id: 29,
  title: 'France',
  childIds: []
},
30: {
  id: 30,
  title: 'Germany',
  childIds: []
},
31: {
  id: 31,
  title: 'Italy',
  childIds: []
},
32: {
  id: 32,
  title: 'Portugal',
  childIds: []
},
33: {
  id: 33,
  title: 'Spain',
  childIds: []
},
34: {
  id: 34,
  title: 'Turkey',
```

```
  childIds: []
},
35: {
  id: 35,
  title: 'Oceania',
  childIds: [36, 37, 38, 39, 40, 41, 42],
},
36: {
  id: 36,
  title: 'Australia',
  childIds: []
},
37: {
  id: 37,
  title: 'Bora Bora (French Polynesia)',
  childIds: []
},
38: {
  id: 38,
  title: 'Easter Island (Chile)',
  childIds: []
},
39: {
  id: 39,
  title: 'Fiji',
  childIds: []
},
40: {
  id: 40,
  title: 'Hawaii (the USA)',
  childIds: []
},
41: {
  id: 41,
  title: 'New Zealand',
  childIds: []
},
42: {
  id: 42,
  title: 'Vanuatu',
  childIds: []
},
43: {
  id: 43,
  title: 'Moon',
  childIds: [44, 45, 46]
```

```
  },
  44: {
    id: 44,
    title: 'Rheita',
    childIds: []
  },
  45: {
    id: 45,
    title: 'Piccolomini',
    childIds: []
  },
  46: {
    id: 46,
    title: 'Tycho',
    childIds: []
  },
  47: {
    id: 47,
    title: 'Mars',
    childIds: [48, 49]
  },
  48: {
    id: 48,
    title: 'Corn Town',
    childIds: []
  },
  49: {
    id: 49,
    title: 'Green Hill',
    childIds: []
  }
};
```

Show more

Now that the state is “flat” (also known as “normalized”), updating nested items becomes easier.

In order to remove a place now, you only need to update two levels of state:

The updated version of its parent place should exclude the removed ID from its childIds array.

The updated version of the root “table” object should include the updated version of the parent place.

Here is an example of how you could go about it:

App.jsplaces.jsApp.js ResetForkimport { useState } from 'react';
import { initialTravelPlan } from './places.js';

```
export default function TravelPlan() {
  const [plan, setPlan] = useState(initialTravelPlan);

  function handleComplete(parentId, childId) {
    const parent = plan[parentId];
    // Create a new version of the parent place
    // that doesn't include this child ID.
    const nextParent = {
      ...parent,
      childIds: parent.childIds
        .filter(id => id !== childId)
    };
    // Update the root state object...
    setPlan({
      ...plan,
      // ...so that it has the updated parent.
      [parentId]: nextParent
    });
  }

  const root = plan[0];
  const planetIds = root.childIds;
  return (
    <>
      <h2>Places to visit</h2>
      <ol>
        {planetIds.map(id => (
          <PlaceTree
            key={id}
            id={id}
            parentId={0}
            placesById={plan}
            onComplete={handleComplete}
          />
        ))}
      </ol>
    </>
  );
}

function PlaceTree({ id, parentId, placesById, onComplete }) {
  const place = placesById[id];
  const childIds = place.childIds;
  return (
    <li>
```

```
      {place.title}
      <button onClick={() => {
        onComplete(parentId, id);
      }}>
        Complete
      </button>
      {childIds.length > 0 &&
        <ol>
          {childIds.map(childId => (
            <PlaceTree
              key={childId}
              id={childId}
              parentId={id}
              placesById={placesById}
              onComplete={onComplete}
            />
          ))}
        </ol>
      }
    </li>
  );
}
```

Show more

You can nest state as much as you like, but making it “flat” can solve numerous problems. It makes state easier to update, and it helps ensure you don’t have duplication in different parts of a nested object.

Deep DiveImproving memory usage Show DetailsIdeally, you would also remove the deleted items (and their children!) from the “table” object to improve memory usage. This version does that. It also uses Immer to make the update logic more concise.App.jsplaces.jspackage.jsonApp.js ResetForkimport { useImmer } from 'use-immer';

```
import { initialTravelPlan } from './places.js';

export default function TravelPlan() {
  const [plan, updatePlan] = useImmer(initialTravelPlan);

  function handleComplete(parentId, childId) {
    updatePlan(draft => {
      // Remove from the parent place's child IDs.
      const parent = draft[parentId];
      parent.childIds = parent.childIds
        .filter(id => id !== childId);

      // Forget this place and all its subtree.
      deleteAllChildren(childId);
```

```
      function deleteAllChildren(id) {
        const place = draft[id];
        place.childIds.forEach(deleteAllChildren);
        delete draft[id];
      }
    });
  }

  const root = plan[0];
  const planetIds = root.childIds;
  return (
    <>
      <h2>Places to visit</h2>
      <ol>
        {planetIds.map(id => (
          <PlaceTree
            key={id}
            id={id}
            parentId={0}
            placesById={plan}
            onComplete={handleComplete}
          />
        ))}
      </ol>
    </>
  );
}

function PlaceTree({ id, parentId, placesById, onComplete }) {
  const place = placesById[id];
  const childIds = place.childIds;
  return (
    <li>
      {place.title}
      <button onClick={() => {
        onComplete(parentId, id);
      }}>
        Complete
      </button>
      {childIds.length > 0 &&
        <ol>
          {childIds.map(childId => (
            <PlaceTree
              key={childId}
              id={childId}
              parentId={id}
```

```
            placesById={placesById}
            onComplete={onComplete}
          />
        ))}
      </ol>
    }
  </li>
 );
}
```

Show more
Sometimes, you can also reduce state nesting by moving some of the nested state into the child components. This works well for ephemeral UI state that doesn’t need to be stored, like whether an item is hovered.
Recap
If two state variables always update together, consider merging them into one.
Choose your state variables carefully to avoid creating “impossible” states.
Structure your state in a way that reduces the chances that you’ll make a mistake updating it.
Avoid redundant and duplicate state so that you don’t need to keep it in sync.
Don’t put props into state unless you specifically want to prevent updates.
For UI patterns like selection, keep ID or index in state instead of the object itself.
If updating deeply nested state is complicated, try flattening it.

Try out some challenges1. Fix a component that’s not updating 2. Fix a broken packing list 3. Fix the disappearing selection 4. Implement multiple selection Challenge 1 of 4: Fix a component that’s not updating This Clock component receives two props: color and time. When you select a different color in the select box, the Clock component receives a different color prop from its parent component. However, for some reason, the displayed color doesn’t update. Why? Fix the problem.Clock.jsClock.js ResetForkimport { useState } from 'react';

```
export default function Clock(props) {
  const [color, setColor] = useState(props.color);
  return (
    <h1 style={{ color: color }}>
      {props.time}
    </h1>
  );
}
```

Show solutionNext ChallengePreviousReacting to Input with StateNextSharing State Between Components
Learn ReactManaging StateSharing State Between ComponentsSometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via

props. This is known as lifting state up, and it's one of the most common things you will do writing React code.
You will learn
How to share state between components by lifting it up
What are controlled and uncontrolled components

Lifting state up by example
In this example, a parent Accordion component renders two separate Panels:

Accordion

Panel
Panel

Each Panel component has a boolean isActive state that determines whether its content is visible.
Press the Show button for both panels:
App.jsApp.js Download ResetFork99123456789101112131415161718192021222324252627282930313 2import { useState } from 'react';function Panel({ title, children }) { const [isActive, setIsActive] = useState(false); return (<section className="panel"> <h3>{title}</h3> {isActive ? ({children}) : (<button onClick={() => setIsActive(true)}> Show </button>)} </ section>);}export default function Accordion() { return (<> <h2>Almaty, Kazakhstan</h2> <Panel title="About"> With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city. </ Panel> <Panel title="Etymology"> The name comes from C ;CÄ0, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild Malus sieversii is considered a likely candidate for the ancestor of the modern domestic apple. </Panel> </>);}Show more
Notice how pressing one panel's button does not affect the other panel—they are independent.
Initially, each Panel's isActive state is false, so they both appear collapsedClicking either Panel's button will only update that Panel's isActive state alone
But now let's say you want to change it so that only one panel is expanded at any given time. With that design, expanding the second panel should collapse the first one. How would you do that?
To coordinate these two panels, you need to "lift their state up" to a parent component in three steps:

Remove state from the child components.
Pass hardcoded data from the common parent.
Add state to the common parent and pass it down together with the event handlers.

This will allow the Accordion component to coordinate both Panels and only expand one at a time.
Step 1: Remove state from the child components
You will give control of the Panel's isActive to its parent component. This means that the parent component will pass isActive to Panel as a prop instead. Start by removing this line from the Panel component:
const [isActive, setIsActive] = useState(false);
And instead, add isActive to the Panel's list of props:
function Panel({ title, children, isActive }) {
Now the Panel's parent component can control isActive by passing it down as a prop. Conversely, the Panel component now has no control over the value of isActive—it's now up to the parent component!
Step 2: Pass hardcoded data from the common parent
To lift state up, you must locate the closest common parent component of both of the child components that you want to coordinate:

Accordion (closest common parent)

Panel
Panel

In this example, it's the Accordion component. Since it's above both panels and can control their props, it will become the "source of truth" for which panel is currently active. Make the Accordion component pass a hardcoded value of isActive (for example, true) to both panels:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Accordion() {
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel title="About" isActive={true}>
        With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
      </Panel>
      <Panel title="Etymology" isActive={true}>
        The name comes from <span lang="kk-KZ">C ;CÄ0</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
      </Panel>
    </>
  );
```

```
}

function Panel({ title, children, isActive }) {
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={() => setIsActive(true)}>
          Show
        </button>
      )}
    </section>
  );
}
```

Show more

Try editing the hardcoded isActive values in the Accordion component and see the result on the screen.

Step 3: Add state to the common parent

Lifting state up often changes the nature of what you're storing as state.

In this case, only one panel should be active at a time. This means that the Accordion common parent component needs to keep track of which panel is the active one. Instead of a boolean value, it could use a number as the index of the active Panel for the state variable:

`const [activeIndex, setActiveIndex] = useState(0);`

When the activeIndex is 0, the first panel is active, and when it's 1, it's the second one.

Clicking the "Show" button in either Panel needs to change the active index in Accordion. A Panel can't set the activeIndex state directly because it's defined inside the Accordion. The Accordion component needs to explicitly allow the Panel component to change its state by passing an event handler down as a prop:

`<> <Panel isActive={activeIndex === 0} onShow={() => setActiveIndex(0)} > ... </Panel> <Panel isActive={activeIndex === 1} onShow={() => setActiveIndex(1)} > ... </Panel></>`

The <button> inside the Panel will now use the onShow prop as its click event handler:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Accordion() {
  const [activeIndex, setActiveIndex] = useState(0);
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel
        title="About"
        isActive={activeIndex === 0}
```

```
        onShow={() => setActiveIndex(0)}
      >
        With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
      </Panel>
      <Panel
        title="Etymology"
        isActive={activeIndex === 1}
        onShow={() => setActiveIndex(1)}
      >
        The name comes from <span lang="kk-KZ">C ;CÄ0</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
      </Panel>
    </>
  );
}

function Panel({
  title,
  children,
  isActive,
  onShow
}) {
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={onShow}>
          Show
        </button>
      )}
    </section>
  );
}
```

Show more

This completes lifting state up! Moving state into the common parent component allowed you to coordinate the two panels. Using the active index instead of two “is shown” flags ensured that only one panel is active at a given time. And passing down the event handler to the child allowed the child to change the parent’s state.

Initially, Accordion’s activeIndex is 0, so the first Panel receives isActive = trueWhen

Accordion’s activeIndex state changes to 1, the second Panel receives isActive = true instead
Deep DiveControlled and uncontrolled components Show DetailsIt is common to call a component with some local state “uncontrolled”. For example, the original Panel component with an isActive state variable is uncontrolled because its parent cannot influence whether the panel is active or not.In contrast, you might say a component is “controlled” when the important information in it is driven by props rather than its own local state. This lets the parent component fully specify its behavior. The final Panel component with the isActive prop is controlled by the Accordion component.Uncontrolled components are easier to use within their parents because they require less configuration. But they’re less flexible when you want to coordinate them together. Controlled components are maximally flexible, but they require the parent components to fully configure them with props.In practice, “controlled” and “uncontrolled” aren’t strict technical terms—each component usually has some mix of both local state and props. However, this is a useful way to talk about how components are designed and what capabilities they offer.When writing a component, consider which information in it should be controlled (via props), and which information should be uncontrolled (via state). But you can always change your mind and refactor later.
A single source of truth for each state
In a React application, many components will have their own state. Some state may “live” close to the leaf components (components at the bottom of the tree) like inputs. Other state may “live” closer to the top of the app. For example, even client-side routing libraries are usually implemented by storing the current route in the React state, and passing it down by props!
For each unique piece of state, you will choose the component that “owns” it. This principle is also known as having a “single source of truth”. It doesn’t mean that all state lives in one place—but that for each piece of state, there is a specific component that holds that piece of information. Instead of duplicating shared state between components, lift it up to their common shared parent, and pass it down to the children that need it.
Your app will change as you work on it. It is common that you will move state down or back up while you’re still figuring out where each piece of the state “lives”. This is all part of the process!
To see what this feels like in practice with a few more components, read Thinking in React.
Recap
When you want to coordinate two components, move their state to their common parent.
Then pass the information down through props from their common parent.
Finally, pass the event handlers down so that the children can change the parent’s state.
It’s useful to consider components as “controlled” (driven by props) or “uncontrolled” (driven by state).

Try out some challenges1. Synced inputs 2. Filtering a list Challenge 1 of 2: Synced inputs These two inputs are independent. Make them stay in sync: editing one input should update the other input with the same text, and vice versa.App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function SyncedInputs() {
  return (
    <>
      <Input label="First input" />
      <Input label="Second input" />
    </>
  );
}

function Input({ label }) {
  const [text, setText] = useState('');

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <label>
      {label}
      {' '}
      <input
        value={text}
        onChange={handleChange}
      />
    </label>
  );
}
```

Show more Show hint Show solutionNext ChallengePreviousChoosing the State StructureNextPreserving and Resetting State

Learn ReactManaging StatePreserving and Resetting StateState is isolated between components. React keeps track of which state belongs to which component based on their place in the UI tree. You can control when to preserve state and when to reset it between re-renders.

You will learn

How React “sees” component structures
When React chooses to preserve or reset the state
How to force React to reset component’s state
How keys and types affect whether the state is preserved

The UI tree

Browsers use many tree structures to model UI. The DOM represents HTML elements, the CSSOM does the same for CSS. There’s even an Accessibility tree!

React also uses tree structures to manage and model the UI you make. React makes UI trees from your JSX. Then React DOM updates the browser DOM elements to

match that UI tree. (React Native translates these trees into elements specific to mobile platforms.)
From components, React creates a UI tree which React DOM uses to render the DOM
State is tied to a position in the tree
When you give a component state, you might think the state “lives” inside the component. But the state is actually held inside React. React associates each piece of state it’s holding with the correct component by where that component sits in the UI tree. Here, there is only one <Counter /> JSX tag, but it’s rendered at two different positions:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function App() {
  const counter = <Counter />;
  return (
    <div>
      {counter}
      {counter}
    </div>
  );
}

function Counter() {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Show more
Here’s how these look as a tree:
React tree

These are two separate counters because each is rendered at its own position in the tree. You don’t usually have to think about these positions to use React, but it can be useful to understand how it works.
In React, each component on the screen has fully isolated state. For example, if you render two Counter components side by side, each of them will get its own, independent, score and hover states.
Try clicking both counters and notice they don’t affect each other:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function App() {
  return (
    <div>
      <Counter />
      <Counter />
    </div>
  );
}

function Counter() {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Show more
As you can see, when one counter is updated, only the state for that component is updated:
Updating state
React will keep the state around for as long as you render the same component at the

same position. To see this, increment both counters, then remove the second component by unchecking “Render the second counter” checkbox, and then add it back by ticking it again:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function App() {
  const [showB, setShowB] = useState(true);
  return (
    <div>
      <Counter />
      {showB && <Counter />}
      <label>
        <input
          type="checkbox"
          checked={showB}
          onChange={e => {
            setShowB(e.target.checked)
          }}
        />
        Render the second counter
      </label>
    </div>
  );
}

function Counter() {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
```

```
}
```

Show more

Notice how the moment you stop rendering the second counter, its state disappears completely. That’s because when React removes a component, it destroys its state.

Deleting a component

When you tick “Render the second counter”, a second Counter and its state are initialized from scratch (score = 0) and added to the DOM.

Adding a component

React preserves a component’s state for as long as it’s being rendered at its position in the UI tree. If it gets removed, or a different component gets rendered at the same position, React discards its state.

Same component at the same position preserves state

In this example, there are two different <Counter /> tags:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function App() {
  const [isFancy, setIsFancy] = useState(false);
  return (
    <div>
      {isFancy ? (
        <Counter isFancy={true} />
      ) : (
        <Counter isFancy={false} />
      )}
      <label>
        <input
          type="checkbox"
          checked={isFancy}
          onChange={e => {
            setIsFancy(e.target.checked)
          }}
        />
        Use fancy styling
      </label>
    </div>
  );
}

function Counter({ isFancy }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
```

```
  }
  if (isFancy) {
    className += ' fancy';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Show more

When you tick or clear the checkbox, the counter state does not get reset. Whether isFancy is true or false, you always have a <Counter /> as the first child of the div returned from the root App component:

Updating the App state does not reset the Counter because Counter stays in the same position

It’s the same component at the same position, so from React’s perspective, it’s the same counter.

PitfallRemember that it’s the position in the UI tree—not in the JSX markup—that matters to React! This component has two return clauses with different <Counter /> JSX tags inside and outside the if:App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function App() {
  const [isFancy, setIsFancy] = useState(false);
  if (isFancy) {
    return (
      <div>
        <Counter isFancy={true} />
        <label>
          <input
            type="checkbox"
            checked={isFancy}
            onChange={e => {
              setIsFancy(e.target.checked)
            }}
          />
```

```
          Use fancy styling
        </label>
      </div>
    );
  }
  return (
    <div>
      <Counter isFancy={false} />
      <label>
        <input
          type="checkbox"
          checked={isFancy}
          onChange={e => {
            setIsFancy(e.target.checked)
          }}
        />
        Use fancy styling
      </label>
    </div>
  );
}

function Counter({ isFancy }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }
  if (isFancy) {
    className += ' fancy';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
```

}

Show moreYou might expect the state to reset when you tick checkbox, but it doesn't! This is because both of these <Counter /> tags are rendered at the same position. React doesn't know where you place the conditions in your function. All it "sees" is the tree you return.In both cases, the App component returns a with <Counter /> as a first child. To React, these two counters have the same "address": the first child of the first child of the root. This is how React matches them up between the previous and next renders, regardless of how you structure your logic.
Different components at the same position reset state
In this example, ticking the checkbox will replace <Counter> with a <p>:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function App() {
  const [isPaused, setIsPaused] = useState(false);
  return (
    <div>
      {isPaused ? (
        <p>See you later!</p>
      ) : (
        <Counter />
      )}
      <label>
        <input
          type="checkbox"
          checked={isPaused}
          onChange={e => {
            setIsPaused(e.target.checked)
          }}
        />
        Take a break
      </label>
    </div>
  );
}

function Counter() {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }

  return (
```

```
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Show more

Here, you switch between different component types at the same position. Initially, the first child of the contained a Counter. But when you swapped in a p, React removed the Counter from the UI tree and destroyed its state.

When Counter changes to p, the Counter is deleted and the p is added

When switching back, the p is deleted and the Counter is added

Also, when you render a different component in the same position, it resets the state of its entire subtree. To see how this works, increment the counter and then tick the checkbox:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function App() {
  const [isFancy, setIsFancy] = useState(false);
  return (
    <div>
      {isFancy ? (
        <div>
          <Counter isFancy={true} />
        </div>
      ) : (
        <section>
          <Counter isFancy={false} />
        </section>
      )}
      <label>
        <input
          type="checkbox"
          checked={isFancy}
          onChange={e => {
            setIsFancy(e.target.checked)
          }}
        />
        Use fancy styling
```

```
      </label>
    </div>
  );
}

function Counter({ isFancy }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }
  if (isFancy) {
    className += ' fancy';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Show more

The counter state gets reset when you click the checkbox. Although you render a Counter, the first child of the div changes from a div to a section. When the child div was removed from the DOM, the whole tree below it (including the Counter and its state) was destroyed as well.

When section changes to div, the section is deleted and the new div is added

When switching back, the div is deleted and the new section is added

As a rule of thumb, if you want to preserve the state between re-renders, the structure of your tree needs to “match up” from one render to another. If the structure is different, the state gets destroyed because React destroys state when it removes a component from the tree.

PitfallThis is why you should not nest component function definitions.Here, the MyTextField component function is defined inside MyComponent:App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function MyComponent() {
  const [counter, setCounter] = useState(0);

  function MyTextField() {
    const [text, setText] = useState('');

    return (
      <input
        value={text}
        onChange={e => setText(e.target.value)}
      />
    );
  }

  return (
    <>
      <MyTextField />
      <button onClick={() => {
        setCounter(counter + 1)
      }}>Clicked {counter} times</button>
    </>
  );
}
```

Show moreEvery time you click the button, the input state disappears! This is because a different MyTextField function is created for every render of MyComponent. You're rendering a different component in the same position, so React resets all state below. This leads to bugs and performance problems. To avoid this problem, always declare component functions at the top level, and don't nest their definitions.

Resetting state at the same position

By default, React preserves state of a component while it stays at the same position. Usually, this is exactly what you want, so it makes sense as the default behavior. But sometimes, you may want to reset a component's state. Consider this app that lets two players keep track of their scores during each turn:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Scoreboard() {
  const [isPlayerA, setIsPlayerA] = useState(true);
  return (
    <div>
      {isPlayerA ? (
        <Counter person="Taylor" />
      ) : (
        <Counter person="Sarah" />
      )}
      <button onClick={() => {
```

```
        setIsPlayerA(!isPlayerA);
      }}>
        Next player!
      </button>
    </div>
  );
}

function Counter({ person }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{person}'s score: {score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Show more

Currently, when you change the player, the score is preserved. The two Counters appear in the same position, so React sees them as the same Counter whose person prop has changed.

But conceptually, in this app they should be two separate counters. They might appear in the same place in the UI, but one is a counter for Taylor, and another is a counter for Sarah.

There are two ways to reset state when switching between them:

Render components in different positions

Give each component an explicit identity with key

Option 1: Rendering a component in different positions

If you want these two Counters to be independent, you can render them in two different positions:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Scoreboard() {
  const [isPlayerA, setIsPlayerA] = useState(true);
  return (
    <div>
      {isPlayerA &&
        <Counter person="Taylor" />
      }
      {!isPlayerA &&
        <Counter person="Sarah" />
      }
      <button onClick={() => {
        setIsPlayerA(!isPlayerA);
      }}>
        Next player!
      </button>
    </div>
  );
}

function Counter({ person }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{person}'s score: {score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Show more

Initially, isPlayerA is true. So the first position contains Counter state, and the second one is empty.
When you click the “Next player” button the first position clears but the second one now contains a Counter.

Initial stateClicking “next”Clicking “next” again
Each Counter’s state gets destroyed each time its removed from the DOM. This is why they reset every time you click the button.
This solution is convenient when you only have a few independent components rendered in the same place. In this example, you only have two, so it’s not a hassle to render both separately in the JSX.
Option 2: Resetting state with a key
There is also another, more generic, way to reset a component’s state.
You might have seen keys when rendering lists. Keys aren’t just for lists! You can use keys to make React distinguish between any components. By default, React uses order within the parent (“first counter”, “second counter”) to discern between components. But keys let you tell React that this is not just a first counter, or a second counter, but a specific counter—for example, Taylor’s counter. This way, React will know Taylor’s counter wherever it appears in the tree!
In this example, the two <Counter />s don’t share state even though they appear in the same place in JSX:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Scoreboard() {
  const [isPlayerA, setIsPlayerA] = useState(true);
  return (
    <div>
      {isPlayerA ? (
        <Counter key="Taylor" person="Taylor" />
      ) : (
        <Counter key="Sarah" person="Sarah" />
      )}
      <button onClick={() => {
        setIsPlayerA(!isPlayerA);
      }}>
        Next player!
      </button>
    </div>
  );
}

function Counter({ person }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
```

```
  if (hover) {
    className += ' hover';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{person}'s score: {score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}
```

Show more

Switching between Taylor and Sarah does not preserve the state. This is because you gave them different keys:

{isPlayerA ? (<Counter key="Taylor" person="Taylor" />) : (<Counter key="Sarah" person="Sarah" />)}

Specifying a key tells React to use the key itself as part of the position, instead of their order within the parent. This is why, even though you render them in the same place in JSX, React sees them as two different counters, and so they will never share state. Every time a counter appears on the screen, its state is created. Every time it is removed, its state is destroyed. Toggling between them resets their state over and over.

NoteRemember that keys are not globally unique. They only specify the position within the parent.

Resetting a form with a key

Resetting state with a key is particularly useful when dealing with forms.

In this chat app, the <Chat> component contains the text input state:

App.jsContactList.jsChat.jsApp.js ResetForkimport { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

```
export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
```

```
      <Chat contact={to} />
    </div>
  )
}

const contacts = [
  { id: 0, name: 'Taylor', email: 'taylor@mail.com' },
  { id: 1, name: 'Alice', email: 'alice@mail.com' },
  { id: 2, name: 'Bob', email: 'bob@mail.com' }
];
```

Show more

Try entering something into the input, and then press “Alice” or “Bob” to choose a different recipient. You will notice that the input state is preserved because the <Chat> is rendered at the same position in the tree.

In many apps, this may be the desired behavior, but not in a chat app! You don’t want to let the user send a message they already typed to a wrong person due to an accidental click. To fix it, add a key:

```
<Chat key={to.id} contact={to} />
```

This ensures that when you select a different recipient, the Chat component will be recreated from scratch, including any state in the tree below it. React will also re-create the DOM elements instead of reusing them.

Now switching the recipient always clears the text field:

App.jsContactList.jsChat.jsApp.js ResetForkimport { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

```
export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
      <Chat key={to.id} contact={to} />
    </div>
  )
}

const contacts = [
  { id: 0, name: 'Taylor', email: 'taylor@mail.com' },
  { id: 1, name: 'Alice', email: 'alice@mail.com' },
  { id: 2, name: 'Bob', email: 'bob@mail.com' }
];
```

Show more
Deep DivePreserving state for removed components Show DetailsIn a real chat app, you'd probably want to recover the input state when the user selects the previous recipient again. There are a few ways to keep the state "alive" for a component that's no longer visible:
You could render all chats instead of just the current one, but hide all the others with CSS. The chats would not get removed from the tree, so their local state would be preserved. This solution works great for simple UIs. But it can get very slow if the hidden trees are large and contain a lot of DOM nodes.
You could lift the state up and hold the pending message for each recipient in the parent component. This way, when the child components get removed, it doesn't matter, because it's the parent that keeps the important information. This is the most common solution.
You might also use a different source in addition to React state. For example, you probably want a message draft to persist even if the user accidentally closes the page. To implement this, you could have the Chat component initialize its state by reading from the localStorage, and save the drafts there too.
No matter which strategy you pick, a chat with Alice is conceptually distinct from a chat with Bob, so it makes sense to give a key to the <Chat> tree based on the current recipient.
Recap
React keeps state for as long as the same component is rendered at the same position.
State is not kept in JSX tags. It's associated with the tree position in which you put that JSX.
You can force a subtree to reset its state by giving it a different key.
Don't nest component definitions, or you'll reset state by accident.

Try out some challenges1. Fix disappearing input text 2. Swap two form fields 3. Reset a detail form 4. Clear an image while it's loading 5. Fix misplaced state in the list
Challenge 1 of 5: Fix disappearing input text This example shows a message when you press the button. However, pressing the button also accidentally resets the input. Why does this happen? Fix it so that pressing the button does not reset the input text.App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function App() {
  const [showHint, setShowHint] = useState(false);
  if (showHint) {
    return (
      <div>
        <p><i>Hint: Your favorite city?</i></p>
        <Form />
        <button onClick={() => {
          setShowHint(false);
        }}>Hide hint</button>
      </div>
```

```
    );
  }
  return (
    <div>
      <Form />
      <button onClick={() => {
        setShowHint(true);
      }}>Show hint</button>
    </div>
  );
}

function Form() {
  const [text, setText] = useState('');
  return (
    <textarea
      value={text}
      onChange={e => setText(e.target.value)}
    />
  );
}
```

Show more Show solutionNext ChallengePreviousSharing State Between ComponentsNextExtracting State Logic into a Reducer

Learn ReactManaging StateExtracting State Logic into a ReducerComponents with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called a reducer.

You will learn

What a reducer function is
How to refactor useState to useReducer
When to use a reducer
How to write one well

Consolidate state logic with a reducer

As your components grow in complexity, it can get harder to see at a glance all the different ways in which a component's state gets updated. For example, the TaskApp component below holds an array of tasks in state and uses three different event handlers to add, remove, and edit tasks:

App.jsApp.js ResetFork991234567891011121314151617181920212223242526272829303132333435 36import { useState } from 'react';import AddTask from './AddTask.js';import TaskList from './TaskList.js';export default function TaskApp() { const [tasks, setTasks] = useState(initialTasks); function handleAddTask(text) {   setTasks([...tasks,   {   id: nextId++,   text: text,   done: false,   },   ]); } function handleChangeTask(task) {   setTasks(tasks.map((t) => {   if (t.id === task.id) {   return task;   } else {   return t;   }   })   ); }

function handleDeleteTask(taskId) { setTasks(tasks.filter((t) => t.id !== taskId)); }
return (<>Show more
Each of its event handlers calls setTasks in order to update the state. As this component grows, so does the amount of state logic sprinkled throughout it. To reduce this complexity and keep all your logic in one easy-to-access place, you can move that state logic into a single function outside your component, called a “reducer”.
Reducers are a different way to handle state. You can migrate from useState to useReducer in three steps:

Move from setting state to dispatching actions.
Write a reducer function.
Use the reducer from your component.

Step 1: Move from setting state to dispatching actions
Your event handlers currently specify what to do by setting state:
function handleAddTask(text) { setTasks([...tasks, { id: nextId++, text: text, done: false, },]);}function handleChangeTask(task) { setTasks(tasks.map((t) => { if (t.id === task.id) { return task; } else { return t; } }));}function handleDeleteTask(taskId) { setTasks(tasks.filter((t) => t.id !== taskId));}
Remove all the state setting logic. What you are left with are three event handlers:

handleAddTask(text) is called when the user presses “Add”.
handleChangeTask(task) is called when the user toggles a task or presses “Save”.
handleDeleteTask(taskId) is called when the user presses “Delete”.

Managing state with reducers is slightly different from directly setting state. Instead of telling React “what to do” by setting state, you specify “what the user just did” by dispatching “actions” from your event handlers. (The state update logic will live elsewhere!) So instead of “setting tasks” via an event handler, you’re dispatching an “added/changed/deleted a task” action. This is more descriptive of the user’s intent.
function handleAddTask(text) { dispatch({ type: 'added', id: nextId++, text: text, });}
function handleChangeTask(task) { dispatch({ type: 'changed', task: task, });}
function handleDeleteTask(taskId) { dispatch({ type: 'deleted', id: taskId, });}
The object you pass to dispatch is called an “action”:
function handleDeleteTask(taskId) { dispatch(// "action" object: { type: 'deleted', id: taskId, });}
It is a regular JavaScript object. You decide what to put in it, but generally it should contain the minimal information about what happened. (You will add the dispatch function itself in a later step.)
NoteAn action object can have any shape.By convention, it is common to give it a string type that describes what happened, and pass any additional information in other fields. The type is specific to a component, so in this example either 'added' or 'added_task' would be fine. Choose a name that says what happened!dispatch({ // specific to component type: 'what_happened', // other fields go here});
Step 2: Write a reducer function
A reducer function is where you will put your state logic. It takes two arguments, the

current state and the action object, and it returns the next state:
function yourReducer(state, action) { // return next state for React to set}
React will set the state to what you return from the reducer.
To move your state setting logic from your event handlers to a reducer function in this example, you will:

Declare the current state (tasks) as the first argument.
Declare the action object as the second argument.
Return the next state from the reducer (which React will set the state to).

Here is all the state setting logic migrated to a reducer function:
function tasksReducer(tasks, action) { if (action.type === 'added') {   return [...tasks,   {   id: action.id,   text: action.text,   done: false,   },   ]; } else if (action.type === 'changed') {   return tasks.map((t) => {   if (t.id === action.task.id) {   return action.task;   } else {   return t;   }   }); } else if (action.type === 'deleted') {   return tasks.filter((t) => t.id !== action.id); } else {   throw Error('Unknown action: ' + action.type); }}
Because the reducer function takes state (tasks) as an argument, you can declare it outside of your component. This decreases the indentation level and can make your code easier to read.
NoteThe code above uses if/else statements, but it’s a convention to use switch statements inside reducers. The result is the same, but it can be easier to read switch statements at a glance.We’ll be using them throughout the rest of this documentation like so:function tasksReducer(tasks, action) { switch (action.type) {   case 'added': {   return [...tasks,   {   id: action.id,   text: action.text,   done: false,   },   ];   }   case 'changed': {   return tasks.map((t) => {   if (t.id === action.task.id) {   return action.task;   } else {   return t;   }   });   }   case 'deleted': {   return tasks.filter((t) => t.id !== action.id);   }   default: {   throw Error('Unknown action: ' + action.type);   } }}We recommend wrapping each case block into the { and } curly braces so that variables declared inside of different cases don’t clash with each other. Also, a case should usually end with a return. If you forget to return, the code will “fall through” to the next case, which can lead to mistakes!If you’re not yet comfortable with switch statements, using if/else is completely fine.
Deep DiveWhy are reducers called this way? Show DetailsAlthough reducers can “reduce” the amount of code inside your component, they are actually named after the reduce() operation that you can perform on arrays.The reduce() operation lets you take an array and “accumulate” a single value out of many:const arr = [1, 2, 3, 4, 5];const sum = arr.reduce((result, number) => result + number); // 1 + 2 + 3 + 4 + 5The function you pass to reduce is known as a “reducer”. It takes the result so far and the current item, then it returns the next result. React reducers are an example of the same idea: they take the state so far and the action, and return the next state. In this way, they accumulate actions over time into state.You could even use the reduce() method with an initialState and an array of actions to calculate the final state by passing your reducer function to it:index.jstasksReducer.jsindex.htmlindex.js ResetForkimport tasksReducer from './tasksReducer.js';

```
let initialState = [];
let actions = [
  {type: 'added', id: 1, text: 'Visit Kafka Museum'},
  {type: 'added', id: 2, text: 'Watch a puppet show'},
  {type: 'deleted', id: 1},
  {type: 'added', id: 3, text: 'Lennon Wall pic'},
];

let finalState = actions.reduce(tasksReducer, initialState);

const output = document.getElementById('output');
output.textContent = JSON.stringify(finalState, null, 2);
```

You probably won’t need to do this yourself, but this is similar to what React does!
Step 3: Use the reducer from your component
Finally, you need to hook up the tasksReducer to your component. Import the useReducer Hook from React:

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

Then you can replace useState:

```
const [tasks, setTasks] = useState(initialTasks);
```

with useReducer like so:

```
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
```

The useReducer Hook is similar to useState—you must pass it an initial state and it returns a stateful value and a way to set state (in this case, the dispatch function). But it’s a little different.
The useReducer Hook takes two arguments:

A reducer function
An initial state

And it returns:

A stateful value
A dispatch function (to “dispatch” user actions to the reducer)

Now it’s fully wired up! Here, the reducer is declared at the bottom of the component file:
App.jsApp.js ResetForkimport { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

```
export default function TaskApp() {
  const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);

  function handleAddTask(text) {
    dispatch({
      type: 'added',
```

```
      id: nextId++,
      text: text,
    });
  }

  function handleChangeTask(task) {
    dispatch({
      type: 'changed',
      task: task,
    });
  }

  function handleDeleteTask(taskId) {
    dispatch({
      type: 'deleted',
      id: taskId,
    });
  }

  return (
    <>
      <h1>Prague itinerary</h1>
      <AddTask onAddTask={handleAddTask} />
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </>
  );
}

function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [
        ...tasks,
        {
          id: action.id,
          text: action.text,
          done: false,
        },
      ];
    }
    case 'changed': {
      return tasks.map((t) => {
```

```
        if (t.id === action.task.id) {
          return action.task;
        } else {
          return t;
        }
      });
    }
    case 'deleted': {
      return tasks.filter((t) => t.id !== action.id);
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

let nextId = 3;
const initialTasks = [
  {id: 0, text: 'Visit Kafka Museum', done: true},
  {id: 1, text: 'Watch a puppet show', done: false},
  {id: 2, text: 'Lennon Wall pic', done: false},
];
```

Show more

If you want, you can even move the reducer to a different file:

App.jstasksReducer.jsApp.js ResetForkimport { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import tasksReducer from './tasksReducer.js';

```
export default function TaskApp() {
  const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);

  function handleAddTask(text) {
    dispatch({
      type: 'added',
      id: nextId++,
      text: text,
    });
  }

  function handleChangeTask(task) {
    dispatch({
      type: 'changed',
      task: task,
    });
```

```
  }

  function handleDeleteTask(taskId) {
    dispatch({
      type: 'deleted',
      id: taskId,
    });
  }

  return (
    <>
      <h1>Prague itinerary</h1>
      <AddTask onAddTask={handleAddTask} />
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </>
  );
}

let nextId = 3;
const initialTasks = [
  {id: 0, text: 'Visit Kafka Museum', done: true},
  {id: 1, text: 'Watch a puppet show', done: false},
  {id: 2, text: 'Lennon Wall pic', done: false},
];
```

Show more

Component logic can be easier to read when you separate concerns like this. Now the event handlers only specify what happened by dispatching actions, and the reducer function determines how the state updates in response to them.

Comparing useState and useReducer

Reducers are not without downsides! Here's a few ways you can compare them:

Code size: Generally, with useState you have to write less code upfront. With useReducer, you have to write both a reducer function and dispatch actions. However, useReducer can help cut down on the code if many event handlers modify state in a similar way.

Readability: useState is very easy to read when the state updates are simple. When they get more complex, they can bloat your component's code and make it difficult to scan. In this case, useReducer lets you cleanly separate the how of update logic from the what happened of event handlers.

Debugging: When you have a bug with useState, it can be difficult to tell where the state was set incorrectly, and why. With useReducer, you can add a console log into

your reducer to see every state update, and why it happened (due to which action). If each action is correct, you'll know that the mistake is in the reducer logic itself. However, you have to step through more code than with useState.
Testing: A reducer is a pure function that doesn't depend on your component. This means that you can export and test it separately in isolation. While generally it's best to test components in a more realistic environment, for complex state update logic it can be useful to assert that your reducer returns a particular state for a particular initial state and action.
Personal preference: Some people like reducers, others don't. That's okay. It's a matter of preference. You can always convert between useState and useReducer back and forth: they are equivalent!

We recommend using a reducer if you often encounter bugs due to incorrect state updates in some component, and want to introduce more structure to its code. You don't have to use reducers for everything: feel free to mix and match! You can even useState and useReducer in the same component.
Writing reducers well
Keep these two tips in mind when writing reducers:

Reducers must be pure. Similar to state updater functions, reducers run during rendering! (Actions are queued until the next render.) This means that reducers must be pure—same inputs always result in the same output. They should not send requests, schedule timeouts, or perform any side effects (operations that impact things outside the component). They should update objects and arrays without mutations.
Each action describes a single user interaction, even if that leads to multiple changes in the data. For example, if a user presses "Reset" on a form with five fields managed by a reducer, it makes more sense to dispatch one reset_form action rather than five separate set_field actions. If you log every action in a reducer, that log should be clear enough for you to reconstruct what interactions or responses happened in what order. This helps with debugging!

Writing concise reducers with Immer
Just like with updating objects and arrays in regular state, you can use the Immer library to make reducers more concise. Here, useImmerReducer lets you mutate the state with push or arr[i] = assignment:
App.jspackage.jsonApp.js ResetForkimport { useImmerReducer } from 'use-immer';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

```
function tasksReducer(draft, action) {
  switch (action.type) {
    case 'added': {
      draft.push({
        id: action.id,
        text: action.text,
        done: false,
```

```
      });
      break;
    }
    case 'changed': {
      const index = draft.findIndex((t) => t.id === action.task.id);
      draft[index] = action.task;
      break;
    }
    case 'deleted': {
      return draft.filter((t) => t.id !== action.id);
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

export default function TaskApp() {
  const [tasks, dispatch] = useImmerReducer(tasksReducer, initialTasks);

  function handleAddTask(text) {
    dispatch({
      type: 'added',
      id: nextId++,
      text: text,
    });
  }

  function handleChangeTask(task) {
    dispatch({
      type: 'changed',
      task: task,
    });
  }

  function handleDeleteTask(taskId) {
    dispatch({
      type: 'deleted',
      id: taskId,
    });
  }

  return (
    <>
      <h1>Prague itinerary</h1>
      <AddTask onAddTask={handleAddTask} />
```

```
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </>
  );
}

let nextId = 3;
const initialTasks = [
  {id: 0, text: 'Visit Kafka Museum', done: true},
  {id: 1, text: 'Watch a puppet show', done: false},
  {id: 2, text: 'Lennon Wall pic', done: false},
];
```

Show more
Reducers must be pure, so they shouldn’t mutate state. But Immer provides you with a special draft object which is safe to mutate. Under the hood, Immer will create a copy of your state with the changes you made to the draft. This is why reducers managed by useImmerReducer can mutate their first argument and don’t need to return state.
Recap
To convert from useState to useReducer:

Dispatch actions from event handlers.
Write a reducer function that returns the next state for a given state and action.
Replace useState with useReducer.

Reducers require you to write a bit more code, but they help with debugging and testing.
Reducers must be pure.
Each action describes a single user interaction.
Use Immer if you want to write reducers in a mutating style.

Try out some challenges1. Dispatch actions from event handlers 2. Clear the input on sending a message 3. Restore input values when switching between tabs 4. Implement useReducer from scratch Challenge 1 of 4: Dispatch actions from event handlers Currently, the event handlers in ContactList.js and Chat.js have // TODO comments. This is why typing into the input doesn’t work, and clicking on the buttons doesn’t change the selected recipient.Replace these two // TODOs with the code to dispatch the corresponding actions. To see the expected shape and the type of the actions, check the reducer in messengerReducer.js. The reducer is already written so you won’t need to change it. You only need to dispatch the actions in ContactList.js and Chat.js.App.jsmessengerReducer.jsContactList.jsChat.jsApp.js ResetForkimport { useReducer } from 'react';
import Chat from './Chat.js';

```
import ContactList from './ContactList.js';
import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {
  const [state, dispatch] = useReducer(messengerReducer, initialState);
  const message = state.message;
  const contact = contacts.find((c) => c.id === state.selectedId);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedId={state.selectedId}
        dispatch={dispatch}
      />
      <Chat
        key={contact.id}
        message={message}
        contact={contact}
        dispatch={dispatch}
      />
    </div>
  );
}

const contacts = [
  {id: 0, name: 'Taylor', email: 'taylor@mail.com'},
  {id: 1, name: 'Alice', email: 'alice@mail.com'},
  {id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

Show more Show hint Show solutionNext ChallengePreviousPreserving and Resetting StateNextPassing Data Deeply with Context

Learn ReactManaging StatePassing Data Deeply with ContextUsually, you will pass information from a parent component to a child component via props. But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information. Context lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

You will learn

What “prop drilling” is
How to replace repetitive prop passing with context
Common use cases for context
Common alternatives to context

The problem with passing props

Passing props is a great way to explicitly pipe data through your UI tree to the components that use it.
But passing props can become verbose and inconvenient when you need to pass some prop deeply through the tree, or if many components need the same prop. The nearest common ancestor could be far removed from the components that need data, and lifting state up that high can lead to a situation called “prop drilling”.
Lifting state upProp drilling
Wouldn’t it be great if there were a way to “teleport” data to the components in the tree that need it without passing props? With React’s context feature, there is!
Context: an alternative to passing props
Context lets a parent component provide data to the entire tree below it. There are many uses for context. Here is one example. Consider this Heading component that accepts a level for its size:
App.jsSection.jsHeading.jsApp.js ResetForkimport Heading from './Heading.js';
import Section from './Section.js';

```
export default function Page() {
  return (
    <Section>
      <Heading level={1}>Title</Heading>
      <Heading level={2}>Heading</Heading>
      <Heading level={3}>Sub-heading</Heading>
      <Heading level={4}>Sub-sub-heading</Heading>
      <Heading level={5}>Sub-sub-sub-heading</Heading>
      <Heading level={6}>Sub-sub-sub-sub-heading</Heading>
    </Section>
  );
}
```

Let’s say you want multiple headings within the same Section to always have the same size:
App.jsSection.jsHeading.jsApp.js ResetForkimport Heading from './Heading.js';
import Section from './Section.js';

```
export default function Page() {
  return (
    <Section>
      <Heading level={1}>Title</Heading>
      <Section>
        <Heading level={2}>Heading</Heading>
        <Heading level={2}>Heading</Heading>
        <Heading level={2}>Heading</Heading>
        <Section>
          <Heading level={3}>Sub-heading</Heading>
          <Heading level={3}>Sub-heading</Heading>
```

```
        <Heading level={3}>Sub-heading</Heading>
        <Section>
          <Heading level={4}>Sub-sub-heading</Heading>
          <Heading level={4}>Sub-sub-heading</Heading>
          <Heading level={4}>Sub-sub-heading</Heading>
        </Section>
      </Section>
    </Section>
  </Section>
 );
}
```

Show more
Currently, you pass the level prop to each <Heading> separately:
<Section> <Heading level={3}>About</Heading> <Heading level={3}>Photos</Heading> <Heading level={3}>Videos</Heading></Section>
It would be nice if you could pass the level prop to the <Section> component instead and remove it from the <Heading>. This way you could enforce that all headings in the same section have the same size:
<Section level={3}> <Heading>About</Heading> <Heading>Photos</Heading> <Heading>Videos</Heading></Section>
But how can the <Heading> component know the level of its closest <Section>? That would require some way for a child to “ask” for data from somewhere above in the tree. You can’t do it with props alone. This is where context comes into play. You will do it in three steps:

Create a context. (You can call it LevelContext, since it’s for the heading level.)
Use that context from the component that needs the data. (Heading will use LevelContext.)
Provide that context from the component that specifies the data. (Section will provide LevelContext.)

Context lets a parent—even a distant one!—provide some data to the entire tree inside of it.
Using context in close childrenUsing context in distant children
Step 1: Create the context
First, you need to create the context. You’ll need to export it from a file so that your components can use it:
App.jsSection.jsHeading.jsLevelContext.jsLevelContext.js ResetForkimport { createContext } from 'react';

```
export const LevelContext = createContext(1);
```

The only argument to createContext is the default value. Here, 1 refers to the biggest heading level, but you could pass any kind of value (even an object). You will see the

significance of the default value in the next step.
Step 2: Use the context
Import the useContext Hook from React and your context:
import { useContext } from 'react';import { LevelContext } from './LevelContext.js';
Currently, the Heading component reads level from props:
export default function Heading({ level, children }) { // ...}
Instead, remove the level prop and read the value from the context you just imported, LevelContext:
export default function Heading({ children }) { const level = useContext(LevelContext); // ...}
useContext is a Hook. Just like useState and useReducer, you can only call a Hook immediately inside a React component (not inside loops or conditions). useContext tells React that the Heading component wants to read the LevelContext.
Now that the Heading component doesn’t have a level prop, you don’t need to pass the level prop to Heading in your JSX like this anymore:
<Section> <Heading level={4}>Sub-sub-heading</Heading> <Heading level={4}>Sub-sub-heading</Heading> <Heading level={4}>Sub-sub-heading</Heading></Section>
Update the JSX so that it’s the Section that receives it instead:
<Section level={4}> <Heading>Sub-sub-heading</Heading> <Heading>Sub-sub-heading</Heading> <Heading>Sub-sub-heading</Heading></Section>
As a reminder, this is the markup that you were trying to get working:
App.jsSection.jsHeading.jsLevelContext.jsApp.js ResetForkimport Heading from './Heading.js';
import Section from './Section.js';

```
export default function Page() {
  return (
    <Section level={1}>
      <Heading>Title</Heading>
      <Section level={2}>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Section level={3}>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Section level={4}>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
          </Section>
        </Section>
      </Section>
    </Section>
  );
```

}

Show more

Notice this example doesn’t quite work, yet! All the headings have the same size because even though you’re using the context, you have not provided it yet. React doesn’t know where to get it!

If you don’t provide the context, React will use the default value you’ve specified in the previous step. In this example, you specified 1 as the argument to createContext, so useContext(LevelContext) returns 1, setting all those headings to <h1>. Let’s fix this problem by having each Section provide its own context.

Step 3: Provide the context

The Section component currently renders its children:

export default function Section({ children }) { return (<section className="section"> {children} </section>);}

Wrap them with a context provider to provide the LevelContext to them:

import { LevelContext } from './LevelContext.js';export default function Section({ level, children }) { return (<section className="section"> <LevelContext.Provider value={level}> {children} </LevelContext.Provider> </section>);}

This tells React: “if any component inside this <Section> asks for LevelContext, give them this level.” The component will use the value of the nearest <LevelContext.Provider> in the UI tree above it.

App.jsSection.jsHeading.jsLevelContext.jsApp.js ResetForkimport Heading from './Heading.js';
import Section from './Section.js';

```
export default function Page() {
  return (
    <Section level={1}>
      <Heading>Title</Heading>
      <Section level={2}>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Section level={3}>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Section level={4}>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
          </Section>
        </Section>
      </Section>
    </Section>
  );
```

}

Show more
It’s the same result as the original code, but you did not need to pass the level prop to each Heading component! Instead, it “figures out” its heading level by asking the closest Section above:

You pass a level prop to the <Section>.
Section wraps its children into <LevelContext.Provider value={level}>.
Heading asks the closest value of LevelContext above with useContext(LevelContext).

Using and providing context from the same component
Currently, you still have to specify each section’s level manually:
export default function Page() { return (<Section level={1}> ... <Section level={2}> ... <Section level={3}> ...
Since context lets you read information from a component above, each Section could read the level from the Section above, and pass level + 1 down automatically. Here is how you could do it:
import { useContext } from 'react';import { LevelContext } from './LevelContext.js';export default function Section({ children }) { const level = useContext(LevelContext); return (<section className="section"> <LevelContext.Provider value={level + 1}> {children} </LevelContext.Provider> </section>);}
With this change, you don’t need to pass the level prop either to the <Section> or to the <Heading>:
App.jsSection.jsHeading.jsLevelContext.jsApp.js ResetForkimport Heading from './Heading.js';
import Section from './Section.js';

```
export default function Page() {
  return (
    <Section>
      <Heading>Title</Heading>
      <Section>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Section>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Section>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
            <Heading>Sub-sub-heading</Heading>
          </Section>
        </Section>
```

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

Show more

Now both Heading and Section read the LevelContext to figure out how “deep” they are. And the Section wraps its children into the LevelContext to specify that anything inside of it is at a “deeper” level.

NoteThis example uses heading levels because they show visually how nested components can override context. But context is useful for many other use cases too. You can pass down any information needed by the entire subtree: the current color theme, the currently logged in user, and so on.

Context passes through intermediate components

You can insert as many components as you like between the component that provides context and the one that uses it. This includes both built-in components like and components you might build yourself.

In this example, the same Post component (with a dashed border) is rendered at two different nesting levels. Notice that the <Heading> inside of it gets its level automatically from the closest <Section>:

App.jsSection.jsHeading.jsLevelContext.jsApp.js ResetForkimport Heading from './Heading.js';
import Section from './Section.js';

```
export default function ProfilePage() {
  return (
    <Section>
      <Heading>My Profile</Heading>
      <Post
        title="Hello traveller!"
        body="Read about my adventures."
      />
      <AllPosts />
    </Section>
  );
}

function AllPosts() {
  return (
    <Section>
      <Heading>Posts</Heading>
      <RecentPosts />
    </Section>
  );
}
```

```
function RecentPosts() {
  return (
    <Section>
      <Heading>Recent Posts</Heading>
      <Post
        title="Flavors of Lisbon"
        body="...those pastéis de nata!"
      />
      <Post
        title="Buenos Aires in the rhythm of tango"
        body="I loved it!"
      />
    </Section>
  );
}

function Post({ title, body }) {
  return (
    <Section isFancy={true}>
      <Heading>
        {title}
      </Heading>
      <p><i>{body}</i></p>
    </Section>
  );
}
```

Show more

You didn’t do anything special for this to work. A Section specifies the context for the tree inside it, so you can insert a <Heading> anywhere, and it will have the correct size. Try it in the sandbox above!

Context lets you write components that “adapt to their surroundings” and display themselves differently depending on where (or, in other words, in which context) they are being rendered.

How context works might remind you of CSS property inheritance. In CSS, you can specify color: blue for a , and any DOM node inside of it, no matter how deep, will inherit that color unless some other DOM node in the middle overrides it with color: green. Similarly, in React, the only way to override some context coming from above is to wrap children into a context provider with a different value.

In CSS, different properties like color and background-color don’t override each other. You can set all ’s color to red without impacting background-color. Similarly, different React contexts don’t override each other. Each context that you make with createContext() is completely separate from other ones, and ties together components using and providing that particular context. One component may use or provide many different contexts without a problem.

Before you use context

Context is very tempting to use! However, this also means it’s too easy to overuse it. Just because you need to pass some props several levels deep doesn’t mean you should put that information into context.
Here’s a few alternatives you should consider before using context:

Start by passing props. If your components are not trivial, it’s not unusual to pass a dozen props down through a dozen components. It may feel like a slog, but it makes it very clear which components use which data! The person maintaining your code will be glad you’ve made the data flow explicit with props.
Extract components and pass JSX as children to them. If you pass some data through many layers of intermediate components that don’t use that data (and only pass it further down), this often means that you forgot to extract some components along the way. For example, maybe you pass data props like posts to visual components that don’t use them directly, like <Layout posts={posts} />. Instead, make Layout take children as a prop, and render <Layout><Posts posts={posts} /></Layout>. This reduces the number of layers between the component specifying the data and the one that needs it.

If neither of these approaches works well for you, consider context.
Use cases for context

Theming: If your app lets the user change its appearance (e.g. dark mode), you can put a context provider at the top of your app, and use that context in components that need to adjust their visual look.
Current account: Many components might need to know the currently logged in user. Putting it in context makes it convenient to read it anywhere in the tree. Some apps also let you operate multiple accounts at the same time (e.g. to leave a comment as a different user). In those cases, it can be convenient to wrap a part of the UI into a nested provider with a different current account value.
Routing: Most routing solutions use context internally to hold the current route. This is how every link “knows” whether it’s active or not. If you build your own router, you might want to do it too.
Managing state: As your app grows, you might end up with a lot of state closer to the top of your app. Many distant components below may want to change it. It is common to use a reducer together with context to manage complex state and pass it down to distant components without too much hassle.

Context is not limited to static values. If you pass a different value on the next render, React will update all the components reading it below! This is why context is often used in combination with state.
In general, if some information is needed by distant components in different parts of the tree, it’s a good indication that context will help you.
Recap
Context lets a component provide some information to the entire tree below it.
To pass context:

Create and export it with export const MyContext = createContext(defaultValue).
Pass it to the useContext(MyContext) Hook to read it in any child component, no matter how deep.
Wrap children into <MyContext.Provider value={...}> to provide it from a parent.

Context passes through any components in the middle.
Context lets you write components that “adapt to their surroundings”.
Before you use context, try passing props or passing JSX as children.

Try out some challengesChallenge 1 of 1: Replace prop drilling with context In this example, toggling the checkbox changes the imageSize prop passed to each <PlaceImage>. The checkbox state is held in the top-level App component, but each <PlaceImage> needs to be aware of it.Currently, App passes imageSize to List, which passes it to each Place, which passes it to the PlaceImage. Remove the imageSize prop, and instead pass it from the App component directly to PlaceImage.You can declare context in Context.js.App.jsContext.jsdata.jsutils.jsApp.js ResetForkimport { useState } from 'react';
import { places } from './data.js';
import { getImageUrl } from './utils.js';

```
export default function App() {
  const [isLarge, setIsLarge] = useState(false);
  const imageSize = isLarge ? 150 : 100;
  return (
    <>
      <label>
        <input
          type="checkbox"
          checked={isLarge}
          onChange={e => {
            setIsLarge(e.target.checked);
          }}
        />
        Use large images
      </label>
      <hr />
      <List imageSize={imageSize} />
    </>
  )
}

function List({ imageSize }) {
  const listItems = places.map(place =>
    <li key={place.id}>
      <Place
```

```
        place={place}
        imageSize={imageSize}
      />
    </li>
  );
  return <ul>{listItems}</ul>;
}

function Place({ place, imageSize }) {
  return (
    <>
      <PlaceImage
        place={place}
        imageSize={imageSize}
      />
      <p>
        <b>{place.name}</b>
        {': ' + place.description}
      </p>
    </>
  );
}

function PlaceImage({ place, imageSize }) {
  return (
    <img
      src={getImageUrl(place)}
      alt={place.name}
      width={imageSize}
      height={imageSize}
    />
  );
}
```

Show more Show solutionPreviousExtracting State Logic into a ReducerNextScaling Up with Reducer and Context

Learn ReactManaging StateScaling Up with Reducer and ContextReducers let you consolidate a component's state update logic. Context lets you pass information deep down to other components. You can combine reducers and context together to manage state of a complex screen.

You will learn

How to combine a reducer with context
How to avoid passing state and dispatch through props
How to keep context and state logic in a separate file

Combining a reducer with context

In this example from the introduction to reducers, the state is managed by a reducer. The reducer function contains all of the state update logic and is declared at the bottom of this file:
App.jsAddTask.jsTaskList.jsApp.js ResetFork991234567891011121314151617181920212223242526272829303132333435 36import { useReducer } from 'react';import AddTask from './AddTask.js';import TaskList from './TaskList.js';export default function TaskApp() { const [tasks, dispatch] = useReducer(tasksReducer, initialTasks); function handleAddTask(text) { dispatch({ type: 'added', id: nextId++, text: text, }); } function handleChangeTask(task) { dispatch({ type: 'changed', task: task }); } function handleDeleteTask(taskId) { dispatch({ type: 'deleted', id: taskId }); } return (<> <h1>Day off in Kyoto</h1> <AddTaskShow more
A reducer helps keep the event handlers short and concise. However, as your app grows, you might run into another difficulty. Currently, the tasks state and the dispatch function are only available in the top-level TaskApp component. To let other components read the list of tasks or change it, you have to explicitly pass down the current state and the event handlers that change it as props.
For example, TaskApp passes a list of tasks and the event handlers to TaskList:
<TaskList tasks={tasks} onChangeTask={handleChangeTask} onDeleteTask={handleDeleteTask}/>
And TaskList passes the event handlers to Task:
<Task task={task} onChange={onChangeTask} onDelete={onDeleteTask}/>
In a small example like this, this works well, but if you have tens or hundreds of components in the middle, passing down all state and functions can be quite frustrating! This is why, as an alternative to passing them through props, you might want to put both the tasks state and the dispatch function into context. This way, any component below TaskApp in the tree can read the tasks and dispatch actions without the repetitive “prop drilling”.
Here is how you can combine a reducer with context:

Create the context.
Put state and dispatch into context.
Use context anywhere in the tree.

Step 1: Create the context
The useReducer Hook returns the current tasks and the dispatch function that lets you update them:
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
To pass them down the tree, you will create two separate contexts:

TasksContext provides the current list of tasks.
TasksDispatchContext provides the function that lets components dispatch actions.

Export them from a separate file so that you can later import them from other files:
App.jsTasksContext.jsAddTask.jsTaskList.jsTasksContext.js ResetForkimport { createContext } from 'react';

```
export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);
```

Here, you’re passing null as the default value to both contexts. The actual values will be provided by the TaskApp component.

Step 2: Put state and dispatch into context

Now you can import both contexts in your TaskApp component. Take the tasks and dispatch returned by useReducer() and provide them to the entire tree below:

```
import { TasksContext, TasksDispatchContext } from './TasksContext.js';export default function TaskApp() {  const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);  // ...  return (    <TasksContext.Provider value={tasks}> <TasksDispatchContext.Provider value={dispatch}>        ...      </ TasksDispatchContext.Provider>    </TasksContext.Provider>  );}
```

For now, you pass the information both via props and in context:

App.jsTasksContext.jsAddTask.jsTaskList.jsApp.js ResetForkimport { useReducer } from 'react';

```
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksContext, TasksDispatchContext } from './TasksContext.js';

export default function TaskApp() {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

  function handleAddTask(text) {
    dispatch({
      type: 'added',
      id: nextId++,
      text: text,
    });
  }

  function handleChangeTask(task) {
    dispatch({
      type: 'changed',
      task: task
    });
  }

  function handleDeleteTask(taskId) {
    dispatch({
      type: 'deleted',
      id: taskId
```

```
    });
  }

  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        <h1>Day off in Kyoto</h1>
        <AddTask
          onAddTask={handleAddTask}
        />
        <TaskList
          tasks={tasks}
          onChangeTask={handleChangeTask}
          onDeleteTask={handleDeleteTask}
        />
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}

function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [...tasks, {
        id: action.id,
        text: action.text,
        done: false
      }];
    }
    case 'changed': {
      return tasks.map(t => {
        if (t.id === action.task.id) {
          return action.task;
        } else {
          return t;
        }
      });
    }
    case 'deleted': {
      return tasks.filter(t => t.id !== action.id);
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}
```

```
let nextId = 3;
const initialTasks = [
  { id: 0, text: 'Philosopher’s Path', done: true },
  { id: 1, text: 'Visit the temple', done: false },
  { id: 2, text: 'Drink matcha', done: false }
];
```

Show more

In the next step, you will remove prop passing.

Step 3: Use context anywhere in the tree

Now you don’t need to pass the list of tasks or the event handlers down the tree:

```
<TasksContext.Provider value={tasks}>  <TasksDispatchContext.Provider value={dispatch}>    <h1>Day off in Kyoto</h1>    <AddTask />    <TaskList />  </TasksDispatchContext.Provider></TasksContext.Provider>
```

Instead, any component that needs the task list can read it from the TaskContext:

```
export default function TaskList() {  const tasks = useContext(TasksContext);  // ...
```

To update the task list, any component can read the dispatch function from context and call it:

```
export default function AddTask() {  const [text, setText] = useState('');  const dispatch = useContext(TasksDispatchContext);  // ...  return (    // ...    <button onClick={() => {      setText('');      dispatch({        type: 'added',        id: nextId++,        text: text,      });    }}>Add</button>    // ...
```

The TaskApp component does not pass any event handlers down, and the TaskList does not pass any event handlers to the Task component either. Each component reads the context that it needs:

App.jsTasksContext.jsAddTask.jsTaskList.jsTaskList.js ResetForkimport { useState, useContext } from 'react';
import { TasksContext, TasksDispatchContext } from './TasksContext.js';

```
export default function TaskList() {
  const tasks = useContext(TasksContext);
  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>
          <Task task={task} />
        </li>
      ))}
    </ul>
  );
}

function Task({ task }) {
  const [isEditing, setIsEditing] = useState(false);
  const dispatch = useContext(TasksDispatchContext);
```

```
let taskContent;
if (isEditing) {
  taskContent = (
    <>
      <input
        value={task.text}
        onChange={e => {
          dispatch({
            type: 'changed',
            task: {
              ...task,
              text: e.target.value
            }
          });
        }} />
      <button onClick={() => setIsEditing(false)}>
        Save
      </button>
    </>
  );
} else {
  taskContent = (
    <>
      {task.text}
      <button onClick={() => setIsEditing(true)}>
        Edit
      </button>
    </>
  );
}
return (
  <label>
    <input
      type="checkbox"
      checked={task.done}
      onChange={e => {
        dispatch({
          type: 'changed',
          task: {
            ...task,
            done: e.target.checked
          }
        });
      }}
    />
    {taskContent}
```

```
      <button onClick={() => {
        dispatch({
          type: 'deleted',
          id: task.id
        });
      }}>
        Delete
      </button>
    </label>
  );
}
```

Show more

The state still “lives” in the top-level TaskApp component, managed with useReducer. But its tasks and dispatch are now available to every component below in the tree by importing and using these contexts.

Moving all wiring into a single file

You don’t have to do this, but you could further declutter the components by moving both reducer and context into a single file. Currently, TasksContext.js contains only two context declarations:

```
import { createContext } from 'react';export const TasksContext = createContext(null);export const TasksDispatchContext = createContext(null);
```

This file is about to get crowded! You’ll move the reducer into that same file. Then you’ll declare a new TasksProvider component in the same file. This component will tie all the pieces together:

It will manage the state with a reducer.
It will provide both contexts to components below.
It will take children as a prop so you can pass JSX to it.

```
export function TasksProvider({ children }) {  const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);  return (    <TasksContext.Provider value={tasks}>      <TasksDispatchContext.Provider value={dispatch}>        {children} </TasksDispatchContext.Provider>    </TasksContext.Provider>  );}
```

This removes all the complexity and wiring from your TaskApp component:

App.jsTasksContext.jsAddTask.jsTaskList.jsApp.js ResetForkimport AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksProvider } from './TasksContext.js';

```
export default function TaskApp() {
  return (
    <TasksProvider>
      <h1>Day off in Kyoto</h1>
      <AddTask />
      <TaskList />
```

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

You can also export functions that use the context from TasksContext.js:
export function useTasks() { return useContext(TasksContext);}export function useTasksDispatch() { return useContext(TasksDispatchContext);}
When a component needs to read context, it can do it through these functions:
const tasks = useTasks();const dispatch = useTasksDispatch();
This doesn’t change the behavior in any way, but it lets you later split these contexts further or add some logic to these functions. Now all of the context and reducer wiring is in TasksContext.js. This keeps the components clean and uncluttered, focused on what they display rather than where they get the data:
App.jsTasksContext.jsAddTask.jsTaskList.jsTaskList.js ResetForkimport { useState } from 'react';
import { useTasks, useTasksDispatch } from './TasksContext.js';

```
export default function TaskList() {
  const tasks = useTasks();
  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>
          <Task task={task} />
        </li>
      ))}
    </ul>
  );
}

function Task({ task }) {
  const [isEditing, setIsEditing] = useState(false);
  const dispatch = useTasksDispatch();
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={e => {
            dispatch({
              type: 'changed',
              task: {
                ...task,
                text: e.target.value
```

```
            }
          });
        }} />
        <button onClick={() => setIsEditing(false)}>
          Save
        </button>
      </>
    );
  } else {
    taskContent = (
      <>
        {task.text}
        <button onClick={() => setIsEditing(true)}>
          Edit
        </button>
      </>
    );
  }
  return (
    <label>
      <input
        type="checkbox"
        checked={task.done}
        onChange={e => {
          dispatch({
            type: 'changed',
            task: {
              ...task,
              done: e.target.checked
            }
          });
        }}
      />
      {taskContent}
      <button onClick={() => {
        dispatch({
          type: 'deleted',
          id: task.id
        });
      }}>
        Delete
      </button>
    </label>
  );
}
```

Show more
You can think of TasksProvider as a part of the screen that knows how to deal with tasks, useTasks as a way to read them, and useTasksDispatch as a way to update them from any component below in the tree.
NoteFunctions like useTasks and useTasksDispatch are called Custom Hooks. Your function is considered a custom Hook if its name starts with use. This lets you use other Hooks, like useContext, inside it.
As your app grows, you may have many context-reducer pairs like this. This is a powerful way to scale your app and lift state up without too much work whenever you want to access the data deep in the tree.
Recap
You can combine reducer with context to let any component read and update state above it.
To provide state and the dispatch function to components below:

Create two contexts (for state and for dispatch functions).
Provide both contexts from the component that uses the reducer.
Use either context from components that need to read them.

You can further declutter the components by moving all wiring into one file.

You can export a component like TasksProvider that provides context.
You can also export custom Hooks like useTasks and useTasksDispatch to read it.

You can have many context-reducer pairs like this in your app.
PreviousPassing Data Deeply with ContextNextEscape Hatches
Learn ReactEscape HatchesAdvancedSome of your components may need to control and synchronize with systems outside of React. For example, you might need to focus an input using the browser API, play and pause a video player implemented without React, or connect and listen to messages from a remote server. In this chapter, you'll learn the escape hatches that let you "step outside" React and connect to external systems. Most of your application logic and data flow should not rely on these features.
In this chapter
How to "remember" information without re-rendering
How to access DOM elements managed by React
How to synchronize components with external systems
How to remove unnecessary Effects from your components
How an Effect's lifecycle is different from a component's
How to prevent some values from re-triggering Effects
How to make your Effect re-run less often
How to share logic between components

Referencing values with refs
When you want a component to "remember" some information, but you don't want that

information to trigger new renders, you can use a ref:
const ref = useRef(0);
Like state, refs are retained by React between re-renders. However, setting state re-renders a component. Changing a ref does not! You can access the current value of that ref through the ref.current property.
App.jsApp.js Download ResetForkimport { useRef } from 'react';

```
export default function Counter() {
  let ref = useRef(0);

  function handleClick() {
    ref.current = ref.current + 1;
    alert('You clicked ' + ref.current + ' times!');
  }

  return (
    <button onClick={handleClick}>
      Click me!
    </button>
  );
}
```

Show more
A ref is like a secret pocket of your component that React doesn’t track. For example, you can use refs to store timeout IDs, DOM elements, and other objects that don’t impact the component’s rendering output.
Ready to learn this topic?Read Referencing Values with Refs to learn how to use refs to remember information.Read More
Manipulating the DOM with refs
React automatically updates the DOM to match your render output, so your components won’t often need to manipulate it. However, sometimes you might need access to the DOM elements managed by React—for example, to focus a node, scroll to it, or measure its size and position. There is no built-in way to do those things in React, so you will need a ref to the DOM node. For example, clicking the button will focus the input using a ref:
App.jsApp.js Download ResetForkimport { useRef } from 'react';

```
export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
```

```
      <input ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```

Show more

Ready to learn this topic?Read Manipulating the DOM with Refs to learn how to access DOM elements managed by React.Read More

Synchronizing with Effects

Some components need to synchronize with external systems. For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. Unlike event handlers, which let you handle particular events, Effects let you run some code after rendering. Use them to synchronize your component with a system outside of React.

Press Play/Pause a few times and see how the video player stays synchronized to the isPlaying prop value:

App.jsApp.js Download ResetForkimport { useState, useRef, useEffect } from 'react';

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  }, [isPlaying]);

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  return (
    <>
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
```

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

Show more

Many Effects also “clean up” after themselves. For example, an Effect that sets up a connection to a chat server should return a cleanup function that tells React how to disconnect your component from that server:

App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
export default function ChatRoom() {
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, []);
  return <h1>Welcome to the chat!</h1>;
}
```

In development, React will immediately run and clean up your Effect one extra time. This is why you see '" Connecting..." printed twice. This ensures that you don’t forget to implement the cleanup function.

Ready to learn this topic?Read Synchronizing with Effects to learn how to synchronize components with external systems.Read More

You Might Not Need An Effect

Effects are an escape hatch from the React paradigm. They let you “step outside” of React and synchronize your components with some external system. If there is no external system involved (for example, if you want to update a component’s state when some props or state change), you shouldn’t need an Effect. Removing unnecessary Effects will make your code easier to follow, faster to run, and less error-prone.

There are two common cases in which you don’t need Effects:

You don’t need Effects to transform data for rendering.
You don’t need Effects to handle user events.

For example, you don’t need an Effect to adjust some state based on other state:

function Form() { const [firstName, setFirstName] = useState('Taylor'); const [lastName, setLastName] = useState('Swift'); // Ø=Ý4 Avoid: redundant state and unnecessary Effect const [fullName, setFullName] = useState(''); useEffect(() => {   setFullName(firstName + ' ' + lastName); }, [firstName, lastName]); // ...}

Instead, calculate as much as you can while rendering:

function Form() { const [firstName, setFirstName] = useState('Taylor'); const [lastName, setLastName] = useState('Swift'); // ' Good: calculated during rendering

const fullName = firstName + ' ' + lastName; // ...}
However, you do need Effects to synchronize with external systems.
Ready to learn this topic?Read You Might Not Need an Effect to learn how to remove unnecessary Effects.Read More
Lifecycle of reactive effects
Effects have a different lifecycle from components. Components may mount, update, or unmount. An Effect can only do two things: to start synchronizing something, and later to stop synchronizing it. This cycle can happen multiple times if your Effect depends on props and state that change over time.
This Effect depends on the value of the roomId prop. Props are reactive values, which means they can change on a re-render. Notice that the Effect re-synchronizes (and re-connects to the server) if roomId changes:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);

  return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
```

```
}
```

Show more

React provides a linter rule to check that you've specified your Effect's dependencies correctly. If you forget to specify roomId in the list of dependencies in the above example, the linter will find that bug automatically.

Ready to learn this topic?Read Lifecycle of Reactive Events to learn how an Effect's lifecycle is different from a component's.Read More

Separating events from Effects

Under ConstructionThis section describes an experimental API that has not yet been released in a stable version of React.

Event handlers only re-run when you perform the same interaction again. Unlike event handlers, Effects re-synchronize if any of the values they read, like props or state, are different than during last render. Sometimes, you want a mix of both behaviors: an Effect that re-runs in response to some values but not others.

All code inside Effects is reactive. It will run again if some reactive value it reads has changed due to a re-render. For example, this Effect will re-connect to the chat if either roomId or theme have changed:

App.jschat.jsnotifications.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId, theme }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.on('connected', () => {
      showNotification('Connected!', theme);
    });
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, theme]);

  return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  const [isDark, setIsDark] = useState(false);
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
```

```
        onChange={e => setRoomId(e.target.value)}
      >
        <option value="general">general</option>
        <option value="travel">travel</option>
        <option value="music">music</option>
      </select>
    </label>
    <label>
      <input
        type="checkbox"
        checked={isDark}
        onChange={e => setIsDark(e.target.checked)}
      />
      Use dark theme
    </label>
    <hr />
    <ChatRoom
      roomId={roomId}
      theme={isDark ? 'dark' : 'light'}
    />
  </>
 );
}
```

Show more

This is not ideal. You want to re-connect to the chat only if the roomId has changed. Switching the theme shouldn't re-connect to the chat! Move the code reading theme out of your Effect into an Effect Event:

App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId, theme }) {
  const onConnected = useEffectEvent(() => {
    showNotification('Connected!', theme);
  });

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.on('connected', () => {
      onConnected();
    });
    connection.connect();
```

```
    return () => connection.disconnect();
  }, [roomId]);

  return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  const [isDark, setIsDark] = useState(false);
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <label>
        <input
          type="checkbox"
          checked={isDark}
          onChange={e => setIsDark(e.target.checked)}
        />
        Use dark theme
      </label>
      <hr />
      <ChatRoom
        roomId={roomId}
        theme={isDark ? 'dark' : 'light'}
      />
    </>
  );
}
```

Show more

Code inside Effect Events isn’t reactive, so changing the theme no longer makes your Effect re-connect.

Ready to learn this topic?Read Separating Events from Effects to learn how to prevent some values from re-triggering Effects.Read More

Removing Effect dependencies

When you write an Effect, the linter will verify that you’ve included every reactive value

(like props and state) that the Effect reads in the list of your Effect's dependencies. This ensures that your Effect remains synchronized with the latest props and state of your component. Unnecessary dependencies may cause your Effect to run too often, or even create an infinite loop. The way you remove them depends on the case.
For example, this Effect depends on the options object which gets re-created every time you edit the input:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const options = {
    serverUrl: serverUrl,
    roomId: roomId
  };

  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [options]);

  return (
    <>
      <h1>Welcome to the {roomId} room!</h1>
      <input value={message} onChange={e => setMessage(e.target.value)} />
    </>
  );
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
```

```
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
}
```

Show more

You don’t want the chat to re-connect every time you start typing a message in that chat. To fix this problem, move creation of the options object inside the Effect so that the Effect only depends on the roomId string:

App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
      roomId: roomId
    };
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);

  return (
    <>
      <h1>Welcome to the {roomId} room!</h1>
      <input value={message} onChange={e => setMessage(e.target.value)} />
    </>
  );
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
```

```
        onChange={e => setRoomId(e.target.value)}
      >
        <option value="general">general</option>
        <option value="travel">travel</option>
        <option value="music">music</option>
      </select>
    </label>
    <hr />
    <ChatRoom roomId={roomId} />
  </>
 );
}
```

Show more

Notice that you didn’t start by editing the dependency list to remove the options dependency. That would be wrong. Instead, you changed the surrounding code so that the dependency became unnecessary. Think of the dependency list as a list of all the reactive values used by your Effect’s code. You don’t intentionally choose what to put on that list. The list describes your code. To change the dependency list, change the code.

Ready to learn this topic?Read Removing Effect Dependencies to learn how to make your Effect re-run less often.Read More

Reusing logic with custom Hooks

React comes with built-in Hooks like useState, useContext, and useEffect. Sometimes, you’ll wish that there was a Hook for some more specific purpose: for example, to fetch data, to keep track of whether the user is online, or to connect to a chat room. To do this, you can create your own Hooks for your application’s needs.

In this example, the usePointerPosition custom Hook tracks the cursor position, while useDelayedValue custom Hook returns a value that’s “lagging behind” the value you passed by a certain number of milliseconds. Move the cursor over the sandbox preview area to see a moving trail of dots following the cursor:

App.jsusePointerPosition.jsuseDelayedValue.jsApp.js ResetForkimport
```
{ usePointerPosition } from './usePointerPosition.js';
import { useDelayedValue } from './useDelayedValue.js';

export default function Canvas() {
  const pos1 = usePointerPosition();
  const pos2 = useDelayedValue(pos1, 100);
  const pos3 = useDelayedValue(pos2, 200);
  const pos4 = useDelayedValue(pos3, 100);
  const pos5 = useDelayedValue(pos4, 50);
  return (
    <>
      <Dot position={pos1} opacity={1} />
      <Dot position={pos2} opacity={0.8} />
      <Dot position={pos3} opacity={0.6} />
      <Dot position={pos4} opacity={0.4} />
```

```
      <Dot position={pos5} opacity={0.2} />
    </>
  );
}

function Dot({ position, opacity }) {
  return (
    <div style={{
      position: 'absolute',
      backgroundColor: 'pink',
      borderRadius: '50%',
      opacity,
      transform: `translate(${position.x}px, ${position.y}px)`,
      pointerEvents: 'none',
      left: -20,
      top: -20,
      width: 40,
      height: 40,
    }} />
  );
}
```

Show more

You can create custom Hooks, compose them together, pass data between them, and reuse them between components. As your app grows, you will write fewer Effects by hand because you'll be able to reuse custom Hooks you already wrote. There are also many excellent custom Hooks maintained by the React community.

Ready to learn this topic?Read Reusing Logic with Custom Hooks to learn how to share logic between components.Read More

What's next?

Head over to Referencing Values with Refs to start reading this chapter page by page!

PreviousScaling Up with Reducer and ContextNextReferencing Values with Refs

Learn ReactEscape HatchesReferencing Values with RefsWhen you want a component to "remember" some information, but you don't want that information to trigger new renders, you can use a ref.

You will learn

How to add a ref to your component

How to update a ref's value

How refs are different from state

How to use refs safely

Adding a ref to your component

You can add a ref to your component by importing the useRef Hook from React:

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

Inside your component, call the useRef Hook and pass the initial value that you want to reference as the only argument. For example, here is a ref to the value 0:

const ref = useRef(0);
useRef returns an object like this:
{ current: 0 // The value you passed to useRef}
Illustrated by Rachel Lee Nabors
You can access the current value of that ref through the ref.current property. This value is intentionally mutable, meaning you can both read and write to it. It's like a secret pocket of your component that React doesn't track. (This is what makes it an "escape hatch" from React's one-way data flow—more on that below!)
Here, a button will increment ref.current on every click:
App.jsApp.js Download ResetForkimport { useRef } from 'react';

```
export default function Counter() {
  let ref = useRef(0);

  function handleClick() {
    ref.current = ref.current + 1;
    alert('You clicked ' + ref.current + ' times!');
  }

  return (
    <button onClick={handleClick}>
      Click me!
    </button>
  );
}
```

Show more
The ref points to a number, but, like state, you could point to anything: a string, an object, or even a function. Unlike state, ref is a plain JavaScript object with the current property that you can read and modify.
Note that the component doesn't re-render with every increment. Like state, refs are retained by React between re-renders. However, setting state re-renders a component. Changing a ref does not!
Example: building a stopwatch
You can combine refs and state in a single component. For example, let's make a stopwatch that the user can start or stop by pressing a button. In order to display how much time has passed since the user pressed "Start", you will need to keep track of when the Start button was pressed and what the current time is. This information is used for rendering, so you'll keep it in state:
const [startTime, setStartTime] = useState(null);const [now, setNow] = useState(null);
When the user presses "Start", you'll use setInterval in order to update the time every 10 milliseconds:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Stopwatch() {
  const [startTime, setStartTime] = useState(null);
```

```
  const [now, setNow] = useState(null);

  function handleStart() {
    // Start counting.
    setStartTime(Date.now());
    setNow(Date.now());

    setInterval(() => {
      // Update the current time every 10ms.
      setNow(Date.now());
    }, 10);
  }

  let secondsPassed = 0;
  if (startTime != null && now != null) {
    secondsPassed = (now - startTime) / 1000;
  }

  return (
    <>
      <h1>Time passed: {secondsPassed.toFixed(3)}</h1>
      <button onClick={handleStart}>
        Start
      </button>
    </>
  );
}
```

Show more

When the “Stop” button is pressed, you need to cancel the existing interval so that it stops updating the now state variable. You can do this by calling clearInterval, but you need to give it the interval ID that was previously returned by the setInterval call when the user pressed Start. You need to keep the interval ID somewhere. Since the interval ID is not used for rendering, you can keep it in a ref:

App.jsApp.js Download ResetForkimport { useState, useRef } from 'react';

```
export default function Stopwatch() {
  const [startTime, setStartTime] = useState(null);
  const [now, setNow] = useState(null);
  const intervalRef = useRef(null);

  function handleStart() {
    setStartTime(Date.now());
    setNow(Date.now());

    clearInterval(intervalRef.current);
```

```
    intervalRef.current = setInterval(() => {
      setNow(Date.now());
    }, 10);
  }

  function handleStop() {
    clearInterval(intervalRef.current);
  }

  let secondsPassed = 0;
  if (startTime != null && now != null) {
    secondsPassed = (now - startTime) / 1000;
  }

  return (
    <>
      <h1>Time passed: {secondsPassed.toFixed(3)}</h1>
      <button onClick={handleStart}>
        Start
      </button>
      <button onClick={handleStop}>
        Stop
      </button>
    </>
  );
}
```

Show more

When a piece of information is used for rendering, keep it in state. When a piece of information is only needed by event handlers and changing it doesn't require a re-render, using a ref may be more efficient.

Differences between refs and state

Perhaps you're thinking refs seem less "strict" than state—you can mutate them instead of always having to use a state setting function, for instance. But in most cases, you'll want to use state. Refs are an "escape hatch" you won't need often. Here's how state and refs compare:

refsstateuseRef(initialValue) returns { current: initialValue }useState(initialValue) returns the current value of a state variable and a state setter function ([value, setValue])Doesn't trigger re-render when you change it.Triggers re-render when you change it.Mutable—you can modify and update current's value outside of the rendering process."Immutable"—you must use the state setting function to modify state variables to queue a re-render.You shouldn't read (or write) the current value during rendering.You can read state at any time. However, each render has its own snapshot of state which does not change.

Here is a counter button that's implemented with state:

App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      You clicked {count} times
    </button>
  );
}
```

Because the count value is displayed, it makes sense to use a state value for it. When the counter's value is set with setCount(), React re-renders the component and the screen updates to reflect the new count.
If you tried to implement this with a ref, React would never re-render the component, so you'd never see the count change! See how clicking this button does not update its text: App.jsApp.js Download ResetForkimport { useRef } from 'react';

```
export default function Counter() {
  let countRef = useRef(0);

  function handleClick() {
    // This doesn't re-render the component!
    countRef.current = countRef.current + 1;
  }

  return (
    <button onClick={handleClick}>
      You clicked {countRef.current} times
    </button>
  );
}
```

Show more
This is why reading ref.current during render leads to unreliable code. If you need that, use state instead.
Deep DiveHow does useRef work inside? Show DetailsAlthough both useState and useRef are provided by React, in principle useRef could be implemented on top of useState. You can imagine that inside of React, useRef is implemented like this:// Inside of Reactfunction useRef(initialValue) { const [ref, unused] = useState({ current: initialValue }); return ref;}During the first render, useRef returns { current: initialValue }.

This object is stored by React, so during the next render the same object will be returned. Note how the state setter is unused in this example. It is unnecessary because useRef always needs to return the same object!React provides a built-in version of useRef because it is common enough in practice. But you can think of it as a regular state variable without a setter. If you're familiar with object-oriented programming, refs might remind you of instance fields—but instead of this.something you write somethingRef.current.

When to use refs

Typically, you will use a ref when your component needs to "step outside" React and communicate with external APIs—often a browser API that won't impact the appearance of the component. Here are a few of these rare situations:

Storing timeout IDs
Storing and manipulating DOM elements, which we cover on the next page
Storing other objects that aren't necessary to calculate the JSX.

If your component needs to store some value, but it doesn't impact the rendering logic, choose refs.

Best practices for refs

Following these principles will make your components more predictable:

Treat refs as an escape hatch. Refs are useful when you work with external systems or browser APIs. If much of your application logic and data flow relies on refs, you might want to rethink your approach.
Don't read or write ref.current during rendering. If some information is needed during rendering, use state instead. Since React doesn't know when ref.current changes, even reading it while rendering makes your component's behavior difficult to predict. (The only exception to this is code like if (!ref.current) ref.current = new Thing() which only sets the ref once during the first render.)

Limitations of React state don't apply to refs. For example, state acts like a snapshot for every render and doesn't update synchronously. But when you mutate the current value of a ref, it changes immediately:

ref.current = 5;console.log(ref.current); // 5

This is because the ref itself is a regular JavaScript object, and so it behaves like one. You also don't need to worry about avoiding mutation when you work with a ref. As long as the object you're mutating isn't used for rendering, React doesn't care what you do with the ref or its contents.

Refs and the DOM

You can point a ref to any value. However, the most common use case for a ref is to access a DOM element. For example, this is handy if you want to focus an input programmatically. When you pass a ref to a ref attribute in JSX, like , React will put the corresponding DOM element into myRef.current. You can read more about this in Manipulating the DOM with Refs.

Recap

Refs are an escape hatch to hold onto values that aren't used for rendering. You won't

need them often.
A ref is a plain JavaScript object with a single property called current, which you can read or set.
You can ask React to give you a ref by calling the useRef Hook.
Like state, refs let you retain information between re-renders of a component.
Unlike state, setting the ref’s current value does not trigger a re-render.
Don’t read or write ref.current during rendering. This makes your component hard to predict.

Try out some challenges1. Fix a broken chat input 2. Fix a component failing to re-render 3. Fix debouncing 4. Read the latest state Challenge 1 of 4: Fix a broken chat input Type a message and click “Send”. You will notice there is a three second delay before you see the “Sent!” alert. During this delay, you can see an “Undo” button. Click it. This “Undo” button is supposed to stop the “Sent!” message from appearing. It does this by calling clearTimeout for the timeout ID saved during handleSend. However, even after “Undo” is clicked, the “Sent!” message still appears. Find why it doesn’t work, and fix it.App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Chat() {
  const [text, setText] = useState('');
  const [isSending, setIsSending] = useState(false);
  let timeoutID = null;

  function handleSend() {
    setIsSending(true);
    timeoutID = setTimeout(() => {
      alert('Sent!');
      setIsSending(false);
    }, 3000);
  }

  function handleUndo() {
    setIsSending(false);
    clearTimeout(timeoutID);
  }

  return (
    <>
      <input
        disabled={isSending}
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button
        disabled={isSending}
        onClick={handleSend}>
```

```
        {isSending ? 'Sending...' : 'Send'}
      </button>
      {isSending &&
        <button onClick={handleUndo}>
          Undo
        </button>
      }
    </>
  );
}
```

Show more Show hint Show solutionNext ChallengePreviousEscape HatchesNextManipulating the DOM with Refs
Learn ReactEscape HatchesManipulating the DOM with RefsReact automatically updates the DOM to match your render output, so your components won't often need to manipulate it. However, sometimes you might need access to the DOM elements managed by React—for example, to focus a node, scroll to it, or measure its size and position. There is no built-in way to do those things in React, so you will need a ref to the DOM node.
You will learn
How to access a DOM node managed by React with the ref attribute
How the ref JSX attribute relates to the useRef Hook
How to access another component's DOM node
In which cases it's safe to modify the DOM managed by React

Getting a ref to the node
To access a DOM node managed by React, first, import the useRef Hook:
```
import { useRef } from 'react';
```
Then, use it to declare a ref inside your component:
```
const myRef = useRef(null);
```
Finally, pass your ref as the ref attribute to the JSX tag for which you want to get the DOM node:
```
<div ref={myRef}>
```
The useRef Hook returns an object with a single property called current. Initially, myRef.current will be null. When React creates a DOM node for this , React will put a reference to this node into myRef.current. You can then access this DOM node from your event handlers and use the built-in browser APIs defined on it.
```
// You can use any browser APIs, for example:myRef.current.scrollIntoView();
```
Example: Focusing a text input
In this example, clicking the button will focus the input:
App.jsApp.js Download ResetForkimport { useRef } from 'react';

```
export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
```

```
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```

Show more
To implement this:

Declare inputRef with the useRef Hook.
Pass it as <input ref={inputRef}>. This tells React to put this <input>’s DOM node into inputRef.current.
In the handleClick function, read the input DOM node from inputRef.current and call focus() on it with inputRef.current.focus().
Pass the handleClick event handler to <button> with onClick.

While DOM manipulation is the most common use case for refs, the useRef Hook can be used for storing other things outside React, like timer IDs. Similarly to state, refs remain between renders. Refs are like state variables that don’t trigger re-renders when you set them. Read about refs in Referencing Values with Refs.
Example: Scrolling to an element
You can have more than a single ref in a component. In this example, there is a carousel of three images. Each button centers an image by calling the browser scrollIntoView() method on the corresponding DOM node:
App.jsApp.js Download ResetForkimport { useRef } from 'react';

```
export default function CatFriends() {
  const firstCatRef = useRef(null);
  const secondCatRef = useRef(null);
  const thirdCatRef = useRef(null);

  function handleScrollToFirstCat() {
    firstCatRef.current.scrollIntoView({
      behavior: 'smooth',
      block: 'nearest',
      inline: 'center'
    });
  }
```

```
function handleScrollToSecondCat() {
  secondCatRef.current.scrollIntoView({
    behavior: 'smooth',
    block: 'nearest',
    inline: 'center'
  });
}

function handleScrollToThirdCat() {
  thirdCatRef.current.scrollIntoView({
    behavior: 'smooth',
    block: 'nearest',
    inline: 'center'
  });
}

return (
  <>
    <nav>
      <button onClick={handleScrollToFirstCat}>
        Tom
      </button>
      <button onClick={handleScrollToSecondCat}>
        Maru
      </button>
      <button onClick={handleScrollToThirdCat}>
        Jellylorum
      </button>
    </nav>
    <div>
      <ul>
        <li>
          <img
            src="https://placekitten.com/g/200/200"
            alt="Tom"
            ref={firstCatRef}
          />
        </li>
        <li>
          <img
            src="https://placekitten.com/g/300/200"
            alt="Maru"
            ref={secondCatRef}
          />
        </li>
        <li>
```

```
          <img
            src="https://placekitten.com/g/250/200"
            alt="Jellylorum"
            ref={thirdCatRef}
          />
        </li>
      </ul>
    </div>
  </>
 );
}
```

Show more

Deep DiveHow to manage a list of refs using a ref callback Show DetailsIn the above examples, there is a predefined number of refs. However, sometimes you might need a ref to each item in the list, and you don’t know how many you will have. Something like this wouldn’t work: {items.map((item) => { // Doesn't work! const ref = useRef(null); return <li ref={ref} />; })}
This is because Hooks must only be called at the top-level of your component. You can’t call useRef in a loop, in a condition, or inside a map() call.One possible way around this is to get a single ref to their parent element, and then use DOM manipulation methods like querySelectorAll to “find” the individual child nodes from it. However, this is brittle and can break if your DOM structure changes.Another solution is to pass a function to the ref attribute. This is called a ref callback. React will call your ref callback with the DOM node when it’s time to set the ref, and with null when it’s time to clear it. This lets you maintain your own array or a Map, and access any ref by its index or some kind of ID.This example shows how you can use this approach to scroll to an arbitrary node in a long list:App.jsApp.js Download ResetForkimport { useRef } from 'react';

```
export default function CatFriends() {
  const itemsRef = useRef(null);

  function scrollToId(itemId) {
    const map = getMap();
    const node = map.get(itemId);
    node.scrollIntoView({
      behavior: 'smooth',
      block: 'nearest',
      inline: 'center'
    });
  }

  function getMap() {
    if (!itemsRef.current) {
      // Initialize the Map on first usage.
      itemsRef.current = new Map();
```

```
    }
    return itemsRef.current;
  }

  return (
    <>
      <nav>
        <button onClick={() => scrollToId(0)}>
          Tom
        </button>
        <button onClick={() => scrollToId(5)}>
          Maru
        </button>
        <button onClick={() => scrollToId(9)}>
          Jellylorum
        </button>
      </nav>
      <div>
        <ul>
          {catList.map(cat => (
            <li
              key={cat.id}
              ref={(node) => {
                const map = getMap();
                if (node) {
                  map.set(cat.id, node);
                } else {
                  map.delete(cat.id);
                }
              }}
            >
              <img
                src={cat.imageUrl}
                alt={'Cat #' + cat.id}
              />
            </li>
          ))}
        </ul>
      </div>
    </>
  );
}

const catList = [];
for (let i = 0; i < 10; i++) {
  catList.push({
```

```
    id: i,
    imageUrl: 'https://placekitten.com/250/200?image=' + i
  });
}
```

Show moreIn this example, itemsRef doesn’t hold a single DOM node. Instead, it holds a Map from item ID to a DOM node. (Refs can hold any values!) The ref callback on every list item takes care to update the Map:<li key={cat.id} ref={node => {   const map = getMap();   if (node) {   // Add to the Map   map.set(cat.id, node);   } else {   // Remove from the Map   map.delete(cat.id);   } }}>This lets you read individual DOM nodes from the Map later.

Accessing another component’s DOM nodes

When you put a ref on a built-in component that outputs a browser element like <input />, React will set that ref’s current property to the corresponding DOM node (such as the actual <input /> in the browser).

However, if you try to put a ref on your own component, like <MyInput />, by default you will get null. Here is an example demonstrating it. Notice how clicking the button does not focus the input:

App.jsApp.js Download ResetForkimport { useRef } from 'react';

```
function MyInput(props) {
  return <input {...props} />;
}

export default function MyForm() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <MyInput ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```

Show more

To help you notice the issue, React also prints an error to the console:

ConsoleWarning: Function components cannot be given refs. Attempts to access this ref will fail. Did you mean to use React.forwardRef()?

This happens because by default React does not let a component access the DOM

nodes of other components. Not even for its own children! This is intentional. Refs are an escape hatch that should be used sparingly. Manually manipulating another component's DOM nodes makes your code even more fragile.
Instead, components that want to expose their DOM nodes have to opt in to that behavior. A component can specify that it "forwards" its ref to one of its children. Here's how MyInput can use the forwardRef API:
const MyInput = forwardRef((props, ref) => { return <input {...props} ref={ref} />;});
This is how it works:

<MyInput ref={inputRef} /> tells React to put the corresponding DOM node into inputRef.current. However, it's up to the MyInput component to opt into that—by default, it doesn't.
The MyInput component is declared using forwardRef. This opts it into receiving the inputRef from above as the second ref argument which is declared after props.
MyInput itself passes the ref it received to the <input> inside of it.

Now clicking the button to focus the input works:
App.jsApp.js Download ResetForkimport { forwardRef, useRef } from 'react';

```
const MyInput = forwardRef((props, ref) => {
  return <input {...props} ref={ref} />;
});

export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <MyInput ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```

Show more
In design systems, it is a common pattern for low-level components like buttons, inputs, and so on, to forward their refs to their DOM nodes. On the other hand, high-level components like forms, lists, or page sections usually won't expose their DOM nodes to avoid accidental dependencies on the DOM structure.
Deep DiveExposing a subset of the API with an imperative handle Show DetailsIn the

above example, MyInput exposes the original DOM input element. This lets the parent component call focus() on it. However, this also lets the parent component do something else—for example, change its CSS styles. In uncommon cases, you may want to restrict the exposed functionality. You can do that with useImperativeHandle:App.jsApp.js Download ResetForkimport {
  forwardRef,
  useRef,
  useImperativeHandle
} from 'react';

const MyInput = forwardRef((props, ref) => {
  const realInputRef = useRef(null);
  useImperativeHandle(ref, () => ({
    // Only expose focus and nothing else
    focus() {
      realInputRef.current.focus();
    },
  }));
  return <input {...props} ref={realInputRef} />;
});

export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <MyInput ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}

Show moreHere, realInputRef inside MyInput holds the actual input DOM node. However, useImperativeHandle instructs React to provide your own special object as the value of a ref to the parent component. So inputRef.current inside the Form component will only have the focus method. In this case, the ref “handle” is not the DOM node, but the custom object you create inside useImperativeHandle call.
When React attaches the refs
In React, every update is split in two phases:

During render, React calls your components to figure out what should be on the screen. During commit, React applies changes to the DOM.

In general, you don’t want to access refs during rendering. That goes for refs holding DOM nodes as well. During the first render, the DOM nodes have not yet been created, so ref.current will be null. And during the rendering of updates, the DOM nodes haven’t been updated yet. So it’s too early to read them.
React sets ref.current during the commit. Before updating the DOM, React sets the affected ref.current values to null. After updating the DOM, React immediately sets them to the corresponding DOM nodes.
Usually, you will access refs from event handlers. If you want to do something with a ref, but there is no particular event to do it in, you might need an Effect. We will discuss effects on the next pages.
Deep DiveFlushing state updates synchronously with flushSync Show DetailsConsider code like this, which adds a new todo and scrolls the screen down to the last child of the list. Notice how, for some reason, it always scrolls to the todo that was just before the last added one:App.jsApp.js Download ResetForkimport { useState, useRef } from 'react';

```
export default function TodoList() {
  const listRef = useRef(null);
  const [text, setText] = useState('');
  const [todos, setTodos] = useState(
    initialTodos
  );

  function handleAdd() {
    const newTodo = { id: nextId++, text: text };
    setText('');
    setTodos([ ...todos, newTodo]);
    listRef.current.lastChild.scrollIntoView({
      behavior: 'smooth',
      block: 'nearest'
    });
  }

  return (
    <>
      <button onClick={handleAdd}>
        Add
      </button>
      <input
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <ul ref={listRef}>
```

```
        {todos.map(todo => (
          <li key={todo.id}>{todo.text}</li>
        ))}
      </ul>
    </>
  );
}

let nextId = 0;
let initialTodos = [];
for (let i = 0; i < 20; i++) {
  initialTodos.push({
    id: nextId++,
    text: 'Todo #' + (i + 1)
  });
}
```

Show moreThe issue is with these two lines:setTodos([...todos, newTodo]);listRef.current.lastChild.scrollIntoView();In React, state updates are queued. Usually, this is what you want. However, here it causes a problem because setTodos does not immediately update the DOM. So the time you scroll the list to its last element, the todo has not yet been added. This is why scrolling always “lags behind” by one item.To fix this issue, you can force React to update (“flush”) the DOM synchronously. To do this, import flushSync from react-dom and wrap the state update into a flushSync call:flushSync(() => { setTodos([...todos, newTodo]);});listRef.current.lastChild.scrollIntoView();This will instruct React to update the DOM synchronously right after the code wrapped in flushSync executes. As a result, the last todo will already be in the DOM by the time you try to scroll to it:App.jsApp.js Download ResetForkimport { useState, useRef } from 'react';
import { flushSync } from 'react-dom';

```
export default function TodoList() {
  const listRef = useRef(null);
  const [text, setText] = useState('');
  const [todos, setTodos] = useState(
    initialTodos
  );

  function handleAdd() {
    const newTodo = { id: nextId++, text: text };
    flushSync(() => {
      setText('');
      setTodos([ ...todos, newTodo]);
    });
    listRef.current.lastChild.scrollIntoView({
      behavior: 'smooth',
```

```
      block: 'nearest'
    });
  }

  return (
    <>
      <button onClick={handleAdd}>
        Add
      </button>
      <input
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <ul ref={listRef}>
        {todos.map(todo => (
          <li key={todo.id}>{todo.text}</li>
        ))}
      </ul>
    </>
  );
}

let nextId = 0;
let initialTodos = [];
for (let i = 0; i < 20; i++) {
  initialTodos.push({
    id: nextId++,
    text: 'Todo #' + (i + 1)
  });
}
```

Show more

Best practices for DOM manipulation with refs

Refs are an escape hatch. You should only use them when you have to “step outside React”. Common examples of this include managing focus, scroll position, or calling browser APIs that React does not expose.

If you stick to non-destructive actions like focusing and scrolling, you shouldn’t encounter any problems. However, if you try to modify the DOM manually, you can risk conflicting with the changes React is making.

To illustrate this problem, this example includes a welcome message and two buttons. The first button toggles its presence using conditional rendering and state, as you would usually do in React. The second button uses the remove() DOM API to forcefully remove it from the DOM outside of React’s control.

Try pressing “Toggle with setState” a few times. The message should disappear and appear again. Then press “Remove from the DOM”. This will forcefully remove it. Finally, press “Toggle with setState”:

App.jsApp.js Download ResetForkimport { useState, useRef } from 'react';

```
export default function Counter() {
  const [show, setShow] = useState(true);
  const ref = useRef(null);

  return (
    <div>
      <button
        onClick={() => {
          setShow(!show);
        }}>
        Toggle with setState
      </button>
      <button
        onClick={() => {
          ref.current.remove();
        }}>
        Remove from the DOM
      </button>
      {show && <p ref={ref}>Hello world</p>}
    </div>
  );
}
```

Show more

After you've manually removed the DOM element, trying to use setState to show it again will lead to a crash. This is because you've changed the DOM, and React doesn't know how to continue managing it correctly.

Avoid changing DOM nodes managed by React. Modifying, adding children to, or removing children from elements that are managed by React can lead to inconsistent visual results or crashes like above.

However, this doesn't mean that you can't do it at all. It requires caution. You can safely modify parts of the DOM that React has no reason to update. For example, if some is always empty in the JSX, React won't have a reason to touch its children list. Therefore, it is safe to manually add or remove elements there.

Recap

Refs are a generic concept, but most often you'll use them to hold DOM elements.

You instruct React to put a DOM node into myRef.current by passing .

Usually, you will use refs for non-destructive actions like focusing, scrolling, or measuring DOM elements.

A component doesn't expose its DOM nodes by default. You can opt into exposing a DOM node by using forwardRef and passing the second ref argument down to a specific node.

Avoid changing DOM nodes managed by React.

If you do modify DOM nodes managed by React, modify parts that React has no

reason to update.

Try out some challenges1. Play and pause the video 2. Focus the search field 3. Scrolling an image carousel 4. Focus the search field with separate components Challenge 1 of 4: Play and pause the video In this example, the button toggles a state variable to switch between a playing and a paused state. However, in order to actually play or pause the video, toggling state is not enough. You also need to call play() and pause() on the DOM element for the <video>. Add a ref to it, and make the button work.App.jsApp.js Download ResetForkimport { useState, useRef } from 'react';

```
export default function VideoPlayer() {
  const [isPlaying, setIsPlaying] = useState(false);

  function handleClick() {
    const nextIsPlaying = !isPlaying;
    setIsPlaying(nextIsPlaying);
  }

  return (
    <>
      <button onClick={handleClick}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <video width="250">
        <source
          src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
          type="video/mp4"
        />
      </video>
    </>
  )
}
```

Show moreFor an extra challenge, keep the “Play” button in sync with whether the video is playing even if the user right-clicks the video and plays it using the built-in browser media controls. You might want to listen to onPlay and onPause on the video to do that. Show solutionNext ChallengePreviousReferencing Values with RefsNextSynchronizing with Effects
Learn ReactEscape HatchesSynchronizing with EffectsSome components need to synchronize with external systems. For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. Effects let you run some code after rendering so that you can synchronize your component with some system outside of React.
You will learn
What Effects are

How Effects are different from events
How to declare an Effect in your component
How to skip re-running an Effect unnecessarily
Why Effects run twice in development and how to fix them

What are Effects and how are they different from events?
Before getting to Effects, you need to be familiar with two types of logic inside React components:

Rendering code (introduced in Describing the UI) lives at the top level of your component. This is where you take the props and state, transform them, and return the JSX you want to see on the screen. Rendering code must be pure. Like a math formula, it should only calculate the result, but not do anything else.

Event handlers (introduced in Adding Interactivity) are nested functions inside your components that do things rather than just calculate them. An event handler might update an input field, submit an HTTP POST request to buy a product, or navigate the user to another screen. Event handlers contain “side effects” (they change the program’s state) caused by a specific user action (for example, a button click or typing).

Sometimes this isn’t enough. Consider a ChatRoom component that must connect to the chat server whenever it’s visible on the screen. Connecting to a server is not a pure calculation (it’s a side effect) so it can’t happen during rendering. However, there is no single particular event like a click that causes ChatRoom to be displayed.
Effects let you specify side effects that are caused by rendering itself, rather than by a particular event. Sending a message in the chat is an event because it is directly caused by the user clicking a specific button. However, setting up a server connection is an Effect because it should happen no matter which interaction caused the component to appear. Effects run at the end of a commit after the screen updates. This is a good time to synchronize the React components with some external system (like network or a third-party library).
NoteHere and later in this text, capitalized “Effect” refers to the React-specific definition above, i.e. a side effect caused by rendering. To refer to the broader programming concept, we’ll say “side effect”.
You might not need an Effect
Don’t rush to add Effects to your components. Keep in mind that Effects are typically used to “step out” of your React code and synchronize with some external system. This includes browser APIs, third-party widgets, network, and so on. If your Effect only adjusts some state based on other state, you might not need an Effect.
How to write an Effect
To write an Effect, follow these three steps:

Declare an Effect. By default, your Effect will run after every render.

Specify the Effect dependencies. Most Effects should only re-run when needed rather than after every render. For example, a fade-in animation should only trigger when a component appears. Connecting and disconnecting to a chat room should only happen when the component appears and disappears, or when the chat room changes. You will learn how to control this by specifying dependencies.
Add cleanup if needed. Some Effects need to specify how to stop, undo, or clean up whatever they were doing. For example, “connect” needs “disconnect”, “subscribe” needs “unsubscribe”, and “fetch” needs either “cancel” or “ignore”. You will learn how to do this by returning a cleanup function.

Let’s look at each of these steps in detail.
Step 1: Declare an Effect
To declare an Effect in your component, import the useEffect Hook from React:
import { useEffect } from 'react';
Then, call it at the top level of your component and put some code inside your Effect:
function MyComponent() { useEffect(() => { // Code here will run after *every* render }); return ;}
Every time your component renders, React will update the screen and then run the code inside useEffect. In other words, useEffect “delays” a piece of code from running until that render is reflected on the screen.
Let’s see how you can use an Effect to synchronize with an external system. Consider a <VideoPlayer> React component. It would be nice to control whether it’s playing or paused by passing an isPlaying prop to it:
<VideoPlayer isPlaying={isPlaying} />;
Your custom VideoPlayer component renders the built-in browser <video> tag:
function VideoPlayer({ src, isPlaying }) { // TODO: do something with isPlaying return <video src={src} />;}
However, the browser <video> tag does not have an isPlaying prop. The only way to control it is to manually call the play() and pause() methods on the DOM element. You need to synchronize the value of isPlaying prop, which tells whether the video should currently be playing, with calls like play() and pause().
We’ll need to first get a ref to the <video> DOM node.
You might be tempted to try to call play() or pause() during rendering, but that isn’t correct:
App.jsApp.js Download ResetForkimport { useState, useRef, useEffect } from 'react';

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  if (isPlaying) {
    ref.current.play();  // Calling these while rendering isn't allowed.
  } else {
    ref.current.pause(); // Also, this crashes.
  }

  return <video ref={ref} src={src} loop playsInline />;
```

```
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  return (
    <>
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
      />
    </>
  );
}
```

Show more

The reason this code isn't correct is that it tries to do something with the DOM node during rendering. In React, rendering should be a pure calculation of JSX and should not contain side effects like modifying the DOM.

Moreover, when VideoPlayer is called for the first time, its DOM does not exist yet! There isn't a DOM node yet to call play() or pause() on, because React doesn't know what DOM to create until you return the JSX.

The solution here is to wrap the side effect with useEffect to move it out of the rendering calculation:

```
import { useEffect, useRef } from 'react';function VideoPlayer({ src, isPlaying }) {  const ref = useRef(null);  useEffect(() => {    if (isPlaying) {      ref.current.play();    } else {      ref.current.pause();    }  });  return <video ref={ref} src={src} loop playsInline />;}
```

By wrapping the DOM update in an Effect, you let React update the screen first. Then your Effect runs.

When your VideoPlayer component renders (either the first time or if it re-renders), a few things will happen. First, React will update the screen, ensuring the <video> tag is in the DOM with the right props. Then React will run your Effect. Finally, your Effect will call play() or pause() depending on the value of isPlaying.

Press Play/Pause multiple times and see how the video player stays synchronized to the isPlaying value:

App.jsApp.js Download ResetForkimport { useState, useRef, useEffect } from 'react';

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
```

```
      ref.current.pause();
    }
  });

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  return (
    <>
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
      />
    </>
  );
}
```

Show more

In this example, the “external system” you synchronized to React state was the browser media API. You can use a similar approach to wrap legacy non-React code (like jQuery plugins) into declarative React components.

Note that controlling a video player is much more complex in practice. Calling play() may fail, the user might play or pause using the built-in browser controls, and so on. This example is very simplified and incomplete.

PitfallBy default, Effects run after every render. This is why code like this will produce an infinite loop:const [count, setCount] = useState(0);useEffect(() => { setCount(count + 1);});Effects run as a result of rendering. Setting state triggers rendering. Setting state immediately in an Effect is like plugging a power outlet into itself. The Effect runs, it sets the state, which causes a re-render, which causes the Effect to run, it sets the state again, this causes another re-render, and so on.Effects should usually synchronize your components with an external system. If there’s no external system and you only want to adjust some state based on other state, you might not need an Effect.

Step 2: Specify the Effect dependencies

By default, Effects run after every render. Often, this is not what you want:

Sometimes, it’s slow. Synchronizing with an external system is not always instant, so you might want to skip doing it unless it’s necessary. For example, you don’t want to reconnect to the chat server on every keystroke.

Sometimes, it’s wrong. For example, you don’t want to trigger a component fade-in animation on every keystroke. The animation should only play once when the component appears for the first time.

To demonstrate the issue, here is the previous example with a few console.log calls and a text input that updates the parent component's state. Notice how typing causes the Effect to re-run:
App.jsApp.js Download ResetForkimport { useState, useRef, useEffect } from 'react';

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      console.log('Calling video.play()');
      ref.current.play();
    } else {
      console.log('Calling video.pause()');
      ref.current.pause();
    }
  });

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  const [text, setText] = useState('');
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
      />
    </>
  );
}
```

Show more
You can tell React to skip unnecessarily re-running the Effect by specifying an array of dependencies as the second argument to the useEffect call. Start by adding an empty [] array to the above example on line 14:

```
  useEffect(() => {    // ...  }, []);
```

You should see an error saying React Hook useEffect has a missing dependency: 'isPlaying':

App.jsApp.js Download ResetForkimport { useState, useRef, useEffect } from 'react';

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      console.log('Calling video.play()');
      ref.current.play();
    } else {
      console.log('Calling video.pause()');
      ref.current.pause();
    }
  }, []); // This causes an error

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  const [text, setText] = useState('');
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
      />
    </>
  );
}
```

Show more

The problem is that the code inside of your Effect depends on the isPlaying prop to decide what to do, but this dependency was not explicitly declared. To fix this issue, add isPlaying to the dependency array:

```
  useEffect(() => {    if (isPlaying) { // It's used here...      // ...    } else {      // ...    }  },
[isPlaying]); // ...so it must be declared here!
```

Now all dependencies are declared, so there is no error. Specifying [isPlaying] as the dependency array tells React that it should skip re-running your Effect if isPlaying is the same as it was during the previous render. With this change, typing into the input doesn’t cause the Effect to re-run, but pressing Play/Pause does:

App.jsApp.js Download ResetForkimport { useState, useRef, useEffect } from 'react';

```
function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      console.log('Calling video.play()');
      ref.current.play();
    } else {
      console.log('Calling video.pause()');
      ref.current.pause();
    }
  }, [isPlaying]);

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  const [text, setText] = useState('');
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
      />
    </>
  );
}
```

Show more

The dependency array can contain multiple dependencies. React will only skip re-running the Effect if all of the dependencies you specify have exactly the same values as they had during the previous render. React compares the dependency values using the Object.is comparison. See the useEffect reference for details.

Notice that you can’t “choose” your dependencies. You will get a lint error if the dependencies you specified don’t match what React expects based on the code inside your Effect. This helps catch many bugs in your code. If you don’t want some code to re-run, edit the Effect code itself to not “need” that dependency.

PitfallThe behaviors without the dependency array and with an empty [] dependency array are different:useEffect(() => { // This runs after every render});useEffect(() => { // This runs only on mount (when the component appears)}, []);useEffect(() => { // This

runs on mount *and also* if either a or b have changed since the last render}, [a, b]);We'll take a close look at what "mount" means in the next step.
Deep DiveWhy was the ref omitted from the dependency array? Show DetailsThis Effect uses both ref and isPlaying, but only isPlaying is declared as a dependency:function VideoPlayer({ src, isPlaying }) { const ref = useRef(null); useEffect(() => { if (isPlaying) { ref.current.play(); } else { ref.current.pause(); } }, [isPlaying]);This is because the ref object has a stable identity: React guarantees you'll always get the same object from the same useRef call on every render. It never changes, so it will never by itself cause the Effect to re-run. Therefore, it does not matter whether you include it or not. Including it is fine too:function VideoPlayer({ src, isPlaying }) { const ref = useRef(null); useEffect(() => { if (isPlaying) { ref.current.play(); } else { ref.current.pause(); } }, [isPlaying, ref]);The set functions returned by useState also have stable identity, so you will often see them omitted from the dependencies too. If the linter lets you omit a dependency without errors, it is safe to do.Omitting always-stable dependencies only works when the linter can "see" that the object is stable. For example, if ref was passed from a parent component, you would have to specify it in the dependency array. However, this is good because you can't know whether the parent component always passes the same ref, or passes one of several refs conditionally. So your Effect would depend on which ref is passed.
Step 3: Add cleanup if needed
Consider a different example. You're writing a ChatRoom component that needs to connect to the chat server when it appears. You are given a createConnection() API that returns an object with connect() and disconnect() methods. How do you keep the component connected while it is displayed to the user?
Start by writing the Effect logic:
useEffect(() => { const connection = createConnection(); connection.connect();});
It would be slow to connect to the chat after every re-render, so you add the dependency array:
useEffect(() => { const connection = createConnection(); connection.connect();}, []);
The code inside the Effect does not use any props or state, so your dependency array is [] (empty). This tells React to only run this code when the component "mounts", i.e. appears on the screen for the first time.
Let's try running this code:
App.jschat.jsApp.js ResetForkimport { useEffect } from 'react';
import { createConnection } from './chat.js';

export default function ChatRoom() {
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
  }, []);
  return <h1>Welcome to the chat!</h1>;
}

This Effect only runs on mount, so you might expect "' Connecting..." to be printed once in the console. However, if you check the console, "' Connecting..." gets printed twice. Why does it happen?
Imagine the ChatRoom component is a part of a larger app with many different screens. The user starts their journey on the ChatRoom page. The component mounts and calls connection.connect(). Then imagine the user navigates to another screen—for example, to the Settings page. The ChatRoom component unmounts. Finally, the user clicks Back and ChatRoom mounts again. This would set up a second connection—but the first connection was never destroyed! As the user navigates across the app, the connections would keep piling up.
Bugs like this are easy to miss without extensive manual testing. To help you spot them quickly, in development React remounts every component once immediately after its initial mount.
Seeing the "' Connecting..." log twice helps you notice the real issue: your code doesn't close the connection when the component unmounts.
To fix the issue, return a cleanup function from your Effect:
  useEffect(() => {   const connection = createConnection();   connection.connect();
return () => {   connection.disconnect();   }; }, []);
React will call your cleanup function each time before the Effect runs again, and one final time when the component unmounts (gets removed). Let's see what happens when the cleanup function is implemented:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
export default function ChatRoom() {
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, []);
  return <h1>Welcome to the chat!</h1>;
}
```

Now you get three console logs in development:

"' Connecting..."
"'L Disconnected."
"' Connecting..."

This is the correct behavior in development. By remounting your component, React verifies that navigating away and back would not break your code. Disconnecting and then connecting again is exactly what should happen! When you implement the cleanup well, there should be no user-visible difference between running the Effect once vs running it, cleaning it up, and running it again. There's an extra connect/disconnect call pair because React is probing your code for bugs in development. This

is normal—don’t try to make it go away!
In production, you would only see "' Connecting..." printed once. Remounting components only happens in development to help you find Effects that need cleanup. You can turn off Strict Mode to opt out of the development behavior, but we recommend keeping it on. This lets you find many bugs like the one above.
How to handle the Effect firing twice in development?
React intentionally remounts your components in development to find bugs like in the last example. The right question isn’t “how to run an Effect once”, but “how to fix my Effect so that it works after remounting”.
Usually, the answer is to implement the cleanup function. The cleanup function should stop or undo whatever the Effect was doing. The rule of thumb is that the user shouldn’t be able to distinguish between the Effect running once (as in production) and a setup !’ cleanup !’ setup sequence (as you’d see in development).
Most of the Effects you’ll write will fit into one of the common patterns below.
Controlling non-React widgets
Sometimes you need to add UI widgets that aren’t written to React. For example, let’s say you’re adding a map component to your page. It has a setZoomLevel() method, and you’d like to keep the zoom level in sync with a zoomLevel state variable in your React code. Your Effect would look similar to this:
useEffect(() => { const map = mapRef.current; map.setZoomLevel(zoomLevel);}, [zoomLevel]);
Note that there is no cleanup needed in this case. In development, React will call the Effect twice, but this is not a problem because calling setZoomLevel twice with the same value does not do anything. It may be slightly slower, but this doesn’t matter because it won’t remount needlessly in production.
Some APIs may not allow you to call them twice in a row. For example, the showModal method of the built-in <dialog> element throws if you call it twice. Implement the cleanup function and make it close the dialog:
useEffect(() => { const dialog = dialogRef.current; dialog.showModal(); return () => dialog.close();}, []);
In development, your Effect will call showModal(), then immediately close(), and then showModal() again. This has the same user-visible behavior as calling showModal() once, as you would see in production.
Subscribing to events
If your Effect subscribes to something, the cleanup function should unsubscribe:
useEffect(() => { function handleScroll(e) {   console.log(window.scrollX, window.scrollY); } window.addEventListener('scroll', handleScroll); return () => window.removeEventListener('scroll', handleScroll);}, []);
In development, your Effect will call addEventListener(), then immediately removeEventListener(), and then addEventListener() again with the same handler. So there would be only one active subscription at a time. This has the same user-visible behavior as calling addEventListener() once, as in production.
Triggering animations
If your Effect animates something in, the cleanup function should reset the animation to the initial values:
useEffect(() => { const node = ref.current; node.style.opacity = 1; // Trigger the

animation return () => {   node.style.opacity = 0; // Reset to the initial value };}, []);
In development, opacity will be set to 1, then to 0, and then to 1 again. This should have the same user-visible behavior as setting it to 1 directly, which is what would happen in production. If you use a third-party animation library with support for tweening, your cleanup function should reset the timeline to its initial state.
Fetching data
If your Effect fetches something, the cleanup function should either abort the fetch or ignore its result:
useEffect(() => { let ignore = false; async function startFetching() {   const json = await fetchTodos(userId);   if (!ignore) {   setTodos(json);   } } startFetching(); return () => {   ignore = true; };}, [userId]);
You can't "undo" a network request that already happened, but your cleanup function should ensure that the fetch that's not relevant anymore does not keep affecting your application. If the userId changes from 'Alice' to 'Bob', cleanup ensures that the 'Alice' response is ignored even if it arrives after 'Bob'.
In development, you will see two fetches in the Network tab. There is nothing wrong with that. With the approach above, the first Effect will immediately get cleaned up so its copy of the ignore variable will be set to true. So even though there is an extra request, it won't affect the state thanks to the if (!ignore) check.
In production, there will only be one request. If the second request in development is bothering you, the best approach is to use a solution that deduplicates requests and caches their responses between components:
function TodoList() { const todos = useSomeDataLibrary(`/api/user/${userId}/todos`); // ...
This will not only improve the development experience, but also make your application feel faster. For example, the user pressing the Back button won't have to wait for some data to load again because it will be cached. You can either build such a cache yourself or use one of the many alternatives to manual fetching in Effects.
Deep DiveWhat are good alternatives to data fetching in Effects? Show DetailsWriting fetch calls inside Effects is a popular way to fetch data, especially in fully client-side apps. This is, however, a very manual approach and it has significant downsides:
Effects don't run on the server. This means that the initial server-rendered HTML will only include a loading state with no data. The client computer will have to download all JavaScript and render your app only to discover that now it needs to load the data. This is not very efficient.
Fetching directly in Effects makes it easy to create "network waterfalls". You render the parent component, it fetches some data, renders the child components, and then they start fetching their data. If the network is not very fast, this is significantly slower than fetching all data in parallel.
Fetching directly in Effects usually means you don't preload or cache data. For example, if the component unmounts and then mounts again, it would have to fetch the data again.
It's not very ergonomic. There's quite a bit of boilerplate code involved when writing fetch calls in a way that doesn't suffer from bugs like race conditions.
This list of downsides is not specific to React. It applies to fetching data on mount with any library. Like with routing, data fetching is not trivial to do well, so we recommend the

following approaches:
If you use a framework, use its built-in data fetching mechanism. Modern React frameworks have integrated data fetching mechanisms that are efficient and don’t suffer from the above pitfalls.
Otherwise, consider using or building a client-side cache. Popular open source solutions include React Query, useSWR, and React Router 6.4+. You can build your own solution too, in which case you would use Effects under the hood, but add logic for deduplicating requests, caching responses, and avoiding network waterfalls (by preloading data or hoisting data requirements to routes).
You can continue fetching data directly in Effects if neither of these approaches suit you.
Sending analytics
Consider this code that sends an analytics event on the page visit:
useEffect(() => { logVisit(url); // Sends a POST request}, [url]);
In development, logVisit will be called twice for every URL, so you might be tempted to try to fix that. We recommend keeping this code as is. Like with earlier examples, there is no user-visible behavior difference between running it once and running it twice.
From a practical point of view, logVisit should not do anything in development because you don’t want the logs from the development machines to skew the production metrics. Your component remounts every time you save its file, so it logs extra visits in development anyway.
In production, there will be no duplicate visit logs.
To debug the analytics events you’re sending, you can deploy your app to a staging environment (which runs in production mode) or temporarily opt out of Strict Mode and its development-only remounting checks. You may also send analytics from the route change event handlers instead of Effects. For more precise analytics, intersection observers can help track which components are in the viewport and how long they remain visible.
Not an Effect: Initializing the application
Some logic should only run once when the application starts. You can put it outside your components:
if (typeof window !== 'undefined') { // Check if we're running in the browser.
checkAuthToken(); loadDataFromLocalStorage();}function App() { // ...}
This guarantees that such logic only runs once after the browser loads the page.
Not an Effect: Buying a product
Sometimes, even if you write a cleanup function, there’s no way to prevent user-visible consequences of running the Effect twice. For example, maybe your Effect sends a POST request like buying a product:
useEffect(() => { // Ø=Ý4 Wrong: This Effect fires twice in development, exposing a problem in the code. fetch('/api/buy', { method: 'POST' });}, []);
You wouldn’t want to buy the product twice. However, this is also why you shouldn’t put this logic in an Effect. What if the user goes to another page and then presses Back? Your Effect would run again. You don’t want to buy the product when the user visits a page; you want to buy it when the user clicks the Buy button.
Buying is not caused by rendering; it’s caused by a specific interaction. It should run only when the user presses the button. Delete the Effect and move your /api/buy request into the Buy button event handler:

function handleClick() { // ' Buying is an event because it is caused by a particular interaction. fetch('/api/buy', { method: 'POST' }); }

This illustrates that if remounting breaks the logic of your application, this usually uncovers existing bugs. From the user's perspective, visiting a page shouldn't be different from visiting it, clicking a link, and pressing Back. React verifies that your components abide by this principle by remounting them once in development.

Putting it all together

This playground can help you "get a feel" for how Effects work in practice.

This example uses setTimeout to schedule a console log with the input text to appear three seconds after the Effect runs. The cleanup function cancels the pending timeout. Start by pressing "Mount the component":

App.jsApp.js Download ResetForkimport { useState, useEffect } from 'react';

```
function Playground() {
  const [text, setText] = useState('a');

  useEffect(() => {
    function onTimeout() {
      console.log('#ð ' + text);
    }

    console.log('Ø=Ý5 Schedule "' + text + '" log');
    const timeoutId = setTimeout(onTimeout, 3000);

    return () => {
      console.log('Ø=ßá Cancel "' + text + '" log');
      clearTimeout(timeoutId);
    };
  }, [text]);

  return (
    <>
      <label>
        What to log:{' '}
        <input
          value={text}
          onChange={e => setText(e.target.value)}
        />
      </label>
      <h1>{text}</h1>
    </>
  );
}

export default function App() {
  const [show, setShow] = useState(false);
```

```
  return (
    <>
      <button onClick={() => setShow(!show)}>
        {show ? 'Unmount' : 'Mount'} the component
      </button>
      {show && <hr />}
      {show && <Playground />}
    </>
  );
}
```

Show more

You will see three logs at first: Schedule "a" log, Cancel "a" log, and Schedule "a" log again. Three second later there will also be a log saying a. As you learned earlier, the extra schedule/cancel pair is because React remounts the component once in development to verify that you've implemented cleanup well.

Now edit the input to say abc. If you do it fast enough, you'll see Schedule "ab" log immediately followed by Cancel "ab" log and Schedule "abc" log. React always cleans up the previous render's Effect before the next render's Effect. This is why even if you type into the input fast, there is at most one timeout scheduled at a time. Edit the input a few times and watch the console to get a feel for how Effects get cleaned up.

Type something into the input and then immediately press "Unmount the component". Notice how unmounting cleans up the last render's Effect. Here, it clears the last timeout before it has a chance to fire.

Finally, edit the component above and comment out the cleanup function so that the timeouts don't get cancelled. Try typing abcde fast. What do you expect to happen in three seconds? Will console.log(text) inside the timeout print the latest text and produce five abcde logs? Give it a try to check your intuition!

Three seconds later, you should see a sequence of logs (a, ab, abc, abcd, and abcde) rather than five abcde logs. Each Effect "captures" the text value from its corresponding render. It doesn't matter that the text state changed: an Effect from the render with text = 'ab' will always see 'ab'. In other words, Effects from each render are isolated from each other. If you're curious how this works, you can read about closures.

Deep DiveEach render has its own Effects Show DetailsYou can think of useEffect as "attaching" a piece of behavior to the render output. Consider this Effect:export default function ChatRoom({ roomId }) { useEffect(() => {   const connection = createConnection(roomId);   connection.connect();   return () => connection.disconnect(); }, [roomId]); return <h1>Welcome to {roomId}!</h1>;}Let's see what exactly happens as the user navigates around the app.Initial render The user visits <ChatRoom roomId="general" />. Let's mentally substitute roomId with 'general': // JSX for the first render (roomId = "general") return <h1>Welcome to general!</h1>;The Effect is also a part of the rendering output. The first render's Effect becomes: // Effect for the first render (roomId = "general") () => {   const connection = createConnection('general');   connection.connect();   return () => connection.disconnect(); }, // Dependencies for the first render (roomId = "general") ['general']React runs this Effect, which connects to the 'general' chat room.Re-render

with same dependencies Let’s say <ChatRoom roomId="general" /> re-renders. The JSX output is the same: // JSX for the second render (roomId = "general") return <h1>Welcome to general!</h1>;React sees that the rendering output has not changed, so it doesn’t update the DOM.The Effect from the second render looks like this: // Effect for the second render (roomId = "general") () => { const connection = createConnection('general'); connection.connect(); return () => connection.disconnect(); }, // Dependencies for the second render (roomId = "general") ['general']React compares ['general'] from the second render with ['general'] from the first render. Because all dependencies are the same, React ignores the Effect from the second render. It never gets called.Re-render with different dependencies Then, the user visits <ChatRoom roomId="travel" />. This time, the component returns different JSX: // JSX for the third render (roomId = "travel") return <h1>Welcome to travel!</h1>;React updates the DOM to change "Welcome to general" into "Welcome to travel".The Effect from the third render looks like this: // Effect for the third render (roomId = "travel") () => { const connection = createConnection('travel'); connection.connect(); return () => connection.disconnect(); }, // Dependencies for the third render (roomId = "travel") ['travel']React compares ['travel'] from the third render with ['general'] from the second render. One dependency is different: Object.is('travel', 'general') is false. The Effect can’t be skipped.Before React can apply the Effect from the third render, it needs to clean up the last Effect that did run. The second render’s Effect was skipped, so React needs to clean up the first render’s Effect. If you scroll up to the first render, you’ll see that its cleanup calls disconnect() on the connection that was created with createConnection('general'). This disconnects the app from the 'general' chat room.After that, React runs the third render’s Effect. It connects to the 'travel' chat room.Unmount Finally, let’s say the user navigates away, and the ChatRoom component unmounts. React runs the last Effect’s cleanup function. The last Effect was from the third render. The third render’s cleanup destroys the createConnection('travel') connection. So the app disconnects from the 'travel' room.Development-only behaviors When Strict Mode is on, React remounts every component once after mount (state and DOM are preserved). This helps you find Effects that need cleanup and exposes bugs like race conditions early. Additionally, React will remount the Effects whenever you save a file in development. Both of these behaviors are development-only.

Recap

Unlike events, Effects are caused by rendering itself rather than a particular interaction.
Effects let you synchronize a component with some external system (third-party API, network, etc).
By default, Effects run after every render (including the initial one).
React will skip the Effect if all of its dependencies have the same values as during the last render.
You can’t “choose” your dependencies. They are determined by the code inside the Effect.
Empty dependency array ([]) corresponds to the component “mounting”, i.e. being added to the screen.
In Strict Mode, React mounts components twice (in development only!) to stress-test your Effects.

If your Effect breaks because of remounting, you need to implement a cleanup function. React will call your cleanup function before the Effect runs next time, and during the unmount.

Try out some challenges1. Focus a field on mount 2. Focus a field conditionally 3. Fix an interval that fires twice 4. Fix fetching inside an Effect Challenge 1 of 4: Focus a field on mount In this example, the form renders a <MyInput /> component.Use the input's focus() method to make MyInput automatically focus when it appears on the screen. There is already a commented out implementation, but it doesn't quite work. Figure out why it doesn't work, and fix it. (If you're familiar with the autoFocus attribute, pretend that it does not exist: we are reimplementing the same functionality from scratch.)MyInput.jsMyInput.js ResetForkimport { useEffect, useRef } from 'react';

```
export default function MyInput({ value, onChange }) {
  const ref = useRef(null);

  // TODO: This doesn't quite work. Fix it.
  // ref.current.focus()

  return (
    <input
      ref={ref}
      value={value}
      onChange={onChange}
    />
  );
}
```

Show moreTo verify that your solution works, press "Show form" and verify that the input receives focus (becomes highlighted and the cursor is placed inside). Press "Hide form" and "Show form" again. Verify the input is highlighted again.MyInput should only focus on mount rather than after every render. To verify that the behavior is right, press "Show form" and then repeatedly press the "Make it uppercase" checkbox. Clicking the checkbox should not focus the input above it. Show solutionNext ChallengePreviousManipulating the DOM with RefsNextYou Might Not Need an Effect Learn ReactEscape HatchesYou Might Not Need an EffectEffects are an escape hatch from the React paradigm. They let you "step outside" of React and synchronize your components with some external system like a non-React widget, network, or the browser DOM. If there is no external system involved (for example, if you want to update a component's state when some props or state change), you shouldn't need an Effect. Removing unnecessary Effects will make your code easier to follow, faster to run, and less error-prone.
You will learn
Why and how to remove unnecessary Effects from your components
How to cache expensive computations without Effects
How to reset and adjust component state without Effects

How to share logic between event handlers
Which logic should be moved to event handlers
How to notify parent components about changes

How to remove unnecessary Effects
There are two common cases in which you don’t need Effects:

You don’t need Effects to transform data for rendering. For example, let’s say you want to filter a list before displaying it. You might feel tempted to write an Effect that updates a state variable when the list changes. However, this is inefficient. When you update the state, React will first call your component functions to calculate what should be on the screen. Then React will “commit” these changes to the DOM, updating the screen. Then React will run your Effects. If your Effect also immediately updates the state, this restarts the whole process from scratch! To avoid the unnecessary render passes, transform all the data at the top level of your components. That code will automatically re-run whenever your props or state change.
You don’t need Effects to handle user events. For example, let’s say you want to send an /api/buy POST request and show a notification when the user buys a product. In the Buy button click event handler, you know exactly what happened. By the time an Effect runs, you don’t know what the user did (for example, which button was clicked). This is why you’ll usually handle user events in the corresponding event handlers.

You do need Effects to synchronize with external systems. For example, you can write an Effect that keeps a jQuery widget synchronized with the React state. You can also fetch data with Effects: for example, you can synchronize the search results with the current search query. Keep in mind that modern frameworks provide more efficient built-in data fetching mechanisms than writing Effects directly in your components.
To help you gain the right intuition, let’s look at some common concrete examples!
Updating state based on props or state
Suppose you have a component with two state variables: firstName and lastName. You want to calculate a fullName from them by concatenating them. Moreover, you’d like fullName to update whenever firstName or lastName change. Your first instinct might be to add a fullName state variable and update it in an Effect:
function Form() { const [firstName, setFirstName] = useState('Taylor'); const [lastName, setLastName] = useState('Swift'); // Ø=Ý4 Avoid: redundant state and unnecessary Effect const [fullName, setFullName] = useState(''); useEffect(() => {   setFullName(firstName + ' ' + lastName); }, [firstName, lastName]); // ...}
This is more complicated than necessary. It is inefficient too: it does an entire render pass with a stale value for fullName, then immediately re-renders with the updated value. Remove the state variable and the Effect:
function Form() { const [firstName, setFirstName] = useState('Taylor'); const [lastName, setLastName] = useState('Swift'); // ' Good: calculated during rendering const fullName = firstName + ' ' + lastName; // ...}
When something can be calculated from the existing props or state, don’t put it in state. Instead, calculate it during rendering. This makes your code faster (you avoid the extra “cascading” updates), simpler (you remove some code), and less error-prone (you

avoid bugs caused by different state variables getting out of sync with each other). If this approach feels new to you, Thinking in React explains what should go into state.
Caching expensive calculations
This component computes visibleTodos by taking the todos it receives by props and filtering them according to the filter prop. You might feel tempted to store the result in state and update it from an Effect:
function TodoList({ todos, filter }) { const [newTodo, setNewTodo] = useState(''); // Ø=Ý4 Avoid: redundant state and unnecessary Effect const [visibleTodos, setVisibleTodos] = useState([]); useEffect(() => {   setVisibleTodos(getFilteredTodos(todos, filter)); }, [todos, filter]); // ...}
Like in the earlier example, this is both unnecessary and inefficient. First, remove the state and the Effect:
function TodoList({ todos, filter }) { const [newTodo, setNewTodo] = useState(''); // ' This is fine if getFilteredTodos() is not slow. const visibleTodos = getFilteredTodos(todos, filter); // ...}
Usually, this code is fine! But maybe getFilteredTodos() is slow or you have a lot of todos. In that case you don’t want to recalculate getFilteredTodos() if some unrelated state variable like newTodo has changed.
You can cache (or “memoize”) an expensive calculation by wrapping it in a useMemo Hook:
import { useMemo, useState } from 'react';function TodoList({ todos, filter }) { const [newTodo, setNewTodo] = useState(''); const visibleTodos = useMemo(() => {   // ' Does not re-run unless todos or filter change   return getFilteredTodos(todos, filter); }, [todos, filter]); // ...}
Or, written as a single line:
import { useMemo, useState } from 'react';function TodoList({ todos, filter }) { const [newTodo, setNewTodo] = useState(''); // ' Does not re-run getFilteredTodos() unless todos or filter change const visibleTodos = useMemo(() => getFilteredTodos(todos, filter), [todos, filter]); // ...}
This tells React that you don’t want the inner function to re-run unless either todos or filter have changed. React will remember the return value of getFilteredTodos() during the initial render. During the next renders, it will check if todos or filter are different. If they’re the same as last time, useMemo will return the last result it has stored. But if they are different, React will call the inner function again (and store its result).
The function you wrap in useMemo runs during rendering, so this only works for pure calculations.
Deep DiveHow to tell if a calculation is expensive? Show DetailsIn general, unless you’re creating or looping over thousands of objects, it’s probably not expensive. If you want to get more confidence, you can add a console log to measure the time spent in a piece of code:console.time('filter array');const visibleTodos = getFilteredTodos(todos, filter);console.timeEnd('filter array');Perform the interaction you’re measuring (for example, typing into the input). You will then see logs like filter array: 0.15ms in your console. If the overall logged time adds up to a significant amount (say, 1ms or more), it might make sense to memoize that calculation. As an experiment, you can then wrap the calculation in useMemo to verify whether the total logged time has decreased for that interaction or not:console.time('filter array');const visibleTodos = useMemo(() =>

{ return getFilteredTodos(todos, filter); // Skipped if todos and filter haven't changed}, [todos, filter]);console.timeEnd('filter array');useMemo won’t make the first render faster. It only helps you skip unnecessary work on updates.Keep in mind that your machine is probably faster than your users’ so it’s a good idea to test the performance with an artificial slowdown. For example, Chrome offers a CPU Throttling option for this.Also note that measuring performance in development will not give you the most accurate results. (For example, when Strict Mode is on, you will see each component render twice rather than once.) To get the most accurate timings, build your app for production and test it on a device like your users have.

Resetting all state when a prop changes

This ProfilePage component receives a userId prop. The page contains a comment input, and you use a comment state variable to hold its value. One day, you notice a problem: when you navigate from one profile to another, the comment state does not get reset. As a result, it’s easy to accidentally post a comment on a wrong user’s profile. To fix the issue, you want to clear out the comment state variable whenever the userId changes:

export default function ProfilePage({ userId }) { const [comment, setComment] = useState(''); // Ø=Ý4 Avoid: Resetting state on prop change in an Effect useEffect(() => { setComment(''); }, [userId]); // ...}

This is inefficient because ProfilePage and its children will first render with the stale value, and then render again. It is also complicated because you’d need to do this in every component that has some state inside ProfilePage. For example, if the comment UI is nested, you’d want to clear out nested comment state too.

Instead, you can tell React that each user’s profile is conceptually a different profile by giving it an explicit key. Split your component in two and pass a key attribute from the outer component to the inner one:

export default function ProfilePage({ userId }) { return (<Profile userId={userId} key={userId} />);}function Profile({ userId }) { // ' This and any other state below will reset on key change automatically const [comment, setComment] = useState(''); // ...}

Normally, React preserves the state when the same component is rendered in the same spot. By passing userId as a key to the Profile component, you’re asking React to treat two Profile components with different userId as two different components that should not share any state. Whenever the key (which you’ve set to userId) changes, React will recreate the DOM and reset the state of the Profile component and all of its children. Now the comment field will clear out automatically when navigating between profiles.

Note that in this example, only the outer ProfilePage component is exported and visible to other files in the project. Components rendering ProfilePage don’t need to pass the key to it: they pass userId as a regular prop. The fact ProfilePage passes it as a key to the inner Profile component is an implementation detail.

Adjusting some state when a prop changes

Sometimes, you might want to reset or adjust a part of the state on a prop change, but not all of it.

This List component receives a list of items as a prop, and maintains the selected item in the selection state variable. You want to reset the selection to null whenever the items prop receives a different array:

function List({ items }) { const [isReverse, setIsReverse] = useState(false); const [selection, setSelection] = useState(null); // Ø=Ý4 Avoid: Adjusting state on prop change in an Effect useEffect(() => {   setSelection(null); }, [items]); // ...}

This, too, is not ideal. Every time the items change, the List and its child components will render with a stale selection value at first. Then React will update the DOM and run the Effects. Finally, the setSelection(null) call will cause another re-render of the List and its child components, restarting this whole process again.

Start by deleting the Effect. Instead, adjust the state directly during rendering:

function List({ items }) { const [isReverse, setIsReverse] = useState(false); const [selection, setSelection] = useState(null); // Better: Adjust the state while rendering const [prevItems, setPrevItems] = useState(items); if (items !== prevItems) {   setPrevItems(items);   setSelection(null); } // ...}

Storing information from previous renders like this can be hard to understand, but it’s better than updating the same state in an Effect. In the above example, setSelection is called directly during a render. React will re-render the List immediately after it exits with a return statement. React has not rendered the List children or updated the DOM yet, so this lets the List children skip rendering the stale selection value.

When you update a component during rendering, React throws away the returned JSX and immediately retries rendering. To avoid very slow cascading retries, React only lets you update the same component’s state during a render. If you update another component’s state during a render, you’ll see an error. A condition like items !== prevItems is necessary to avoid loops. You may adjust state like this, but any other side effects (like changing the DOM or setting timeouts) should stay in event handlers or Effects to keep components pure.

Although this pattern is more efficient than an Effect, most components shouldn’t need it either. No matter how you do it, adjusting state based on props or other state makes your data flow more difficult to understand and debug. Always check whether you can reset all state with a key or calculate everything during rendering instead. For example, instead of storing (and resetting) the selected item, you can store the selected item ID:

function List({ items }) { const [isReverse, setIsReverse] = useState(false); const [selectedId, setSelectedId] = useState(null); // ' Best: Calculate everything during rendering const selection = items.find(item => item.id === selectedId) ?? null; // ...}

Now there is no need to “adjust” the state at all. If the item with the selected ID is in the list, it remains selected. If it’s not, the selection calculated during rendering will be null because no matching item was found. This behavior is different, but arguably better because most changes to items preserve the selection.

Sharing logic between event handlers

Let’s say you have a product page with two buttons (Buy and Checkout) that both let you buy that product. You want to show a notification whenever the user puts the product in the cart. Calling showNotification() in both buttons’ click handlers feels repetitive so you might be tempted to place this logic in an Effect:

function ProductPage({ product, addToCart }) { // Ø=Ý4 Avoid: Event-specific logic inside an Effect useEffect(() => {   if (product.isInCart) {   showNotification(`Added ${product.name} to the shopping cart!`);   } }, [product]); function handleBuyClick() {   addToCart(product); } function handleCheckoutClick() {   addToCart(product);   navigateTo('/checkout'); } // ...}

This Effect is unnecessary. It will also most likely cause bugs. For example, let’s say that your app “remembers” the shopping cart between the page reloads. If you add a product to the cart once and refresh the page, the notification will appear again. It will keep appearing every time you refresh that product’s page. This is because product.isInCart will already be true on the page load, so the Effect above will call showNotification().

When you’re not sure whether some code should be in an Effect or in an event handler, ask yourself why this code needs to run. Use Effects only for code that should run because the component was displayed to the user. In this example, the notification should appear because the user pressed the button, not because the page was displayed! Delete the Effect and put the shared logic into a function called from both event handlers:

function ProductPage({ product, addToCart }) { // ' Good: Event-specific logic is called from event handlers function buyProduct() { addToCart(product); showNotification(`Added ${product.name} to the shopping cart!`); } function handleBuyClick() { buyProduct(); } function handleCheckoutClick() { buyProduct(); navigateTo('/checkout'); } // ...}

This both removes the unnecessary Effect and fixes the bug.

Sending a POST request

This Form component sends two kinds of POST requests. It sends an analytics event when it mounts. When you fill in the form and click the Submit button, it will send a POST request to the /api/register endpoint:

function Form() { const [firstName, setFirstName] = useState(''); const [lastName, setLastName] = useState(''); // ' Good: This logic should run because the component was displayed useEffect(() => { post('/analytics/event', { eventName: 'visit_form' }); }, []); // Ø=Ý4 Avoid: Event-specific logic inside an Effect const [jsonToSubmit, setJsonToSubmit] = useState(null); useEffect(() => { if (jsonToSubmit !== null) { post('/api/register', jsonToSubmit); } }, [jsonToSubmit]); function handleSubmit(e) { e.preventDefault(); setJsonToSubmit({ firstName, lastName }); } // ...}

Let’s apply the same criteria as in the example before.

The analytics POST request should remain in an Effect. This is because the reason to send the analytics event is that the form was displayed. (It would fire twice in development, but see here for how to deal with that.)

However, the /api/register POST request is not caused by the form being displayed. You only want to send the request at one specific moment in time: when the user presses the button. It should only ever happen on that particular interaction. Delete the second Effect and move that POST request into the event handler:

function Form() { const [firstName, setFirstName] = useState(''); const [lastName, setLastName] = useState(''); // ' Good: This logic runs because the component was displayed useEffect(() => { post('/analytics/event', { eventName: 'visit_form' }); }, []); function handleSubmit(e) { e.preventDefault(); // ' Good: Event-specific logic is in the event handler post('/api/register', { firstName, lastName }); } // ...}

When you choose whether to put some logic into an event handler or an Effect, the main question you need to answer is what kind of logic it is from the user’s perspective. If this logic is caused by a particular interaction, keep it in the event handler. If it’s caused by the user seeing the component on the screen, keep it in the Effect.

Chains of computations

Sometimes you might feel tempted to chain Effects that each adjust a piece of state based on other state:

```
function Game() {  const [card, setCard] = useState(null);  const [goldCardCount, setGoldCardCount] = useState(0);  const [round, setRound] = useState(1);  const [isGameOver, setIsGameOver] = useState(false);  // Ø=Ý4 Avoid: Chains of Effects that adjust the state solely to trigger each other  useEffect(() => {    if (card !== null && card.gold) {      setGoldCardCount(c => c + 1);    } }, [card]);  useEffect(() => {    if (goldCardCount > 3) {      setRound(r => r + 1)      setGoldCardCount(0);    } }, [goldCardCount]);  useEffect(() => {    if (round > 5) {      setIsGameOver(true);    } }, [round]);  useEffect(() => {    alert('Good game!');  }, [isGameOver]);  function handlePlaceCard(nextCard) {    if (isGameOver) {      throw Error('Game already ended.');    } else {      setCard(nextCard);    } }  // ...
```

There are two problems with this code.

One problem is that it is very inefficient: the component (and its children) have to re-render between each set call in the chain. In the example above, in the worst case (setCard !’ render !’ setGoldCardCount !’ render !’ setRound !’ render !’ setIsGameOver !’ render) there are three unnecessary re-renders of the tree below.

Even if it weren’t slow, as your code evolves, you will run into cases where the “chain” you wrote doesn’t fit the new requirements. Imagine you are adding a way to step through the history of the game moves. You’d do it by updating each state variable to a value from the past. However, setting the card state to a value from the past would trigger the Effect chain again and change the data you’re showing. Such code is often rigid and fragile.

In this case, it’s better to calculate what you can during rendering, and adjust the state in the event handler:

```
function Game() {  const [card, setCard] = useState(null);  const [goldCardCount, setGoldCardCount] = useState(0);  const [round, setRound] = useState(1);  // '  Calculate what you can during rendering  const isGameOver = round > 5;  function handlePlaceCard(nextCard) {    if (isGameOver) {      throw Error('Game already ended.');    }    // '  Calculate all the next state in the event handler    setCard(nextCard);    if (nextCard.gold) {      if (goldCardCount <= 3) {        setGoldCardCount(goldCardCount + 1);      } else {        setGoldCardCount(0);        setRound(round + 1);        if (round === 5) {          alert('Good game!');        }      }    } }  // ...
```

This is a lot more efficient. Also, if you implement a way to view game history, now you will be able to set each state variable to a move from the past without triggering the Effect chain that adjusts every other value. If you need to reuse logic between several event handlers, you can extract a function and call it from those handlers.

Remember that inside event handlers, state behaves like a snapshot. For example, even after you call setRound(round + 1), the round variable will reflect the value at the time the user clicked the button. If you need to use the next value for calculations, define it manually like const nextRound = round + 1.

In some cases, you can’t calculate the next state directly in the event handler. For example, imagine a form with multiple dropdowns where the options of the next dropdown depend on the selected value of the previous dropdown. Then, a chain of Effects is appropriate because you are synchronizing with network.

Initializing the application
Some logic should only run once when the app loads.
You might be tempted to place it in an Effect in the top-level component:
function App() { // Ø=Ý4 Avoid: Effects with logic that should only ever run once useEffect(() => {   loadDataFromLocalStorage();   checkAuthToken(); }, []); // ...}
However, you'll quickly discover that it runs twice in development. This can cause issues—for example, maybe it invalidates the authentication token because the function wasn't designed to be called twice. In general, your components should be resilient to being remounted. This includes your top-level App component.
Although it may not ever get remounted in practice in production, following the same constraints in all components makes it easier to move and reuse code. If some logic must run once per app load rather than once per component mount, add a top-level variable to track whether it has already executed:
let didInit = false;function App() { useEffect(() => {   if (!didInit) {   didInit = true;   // ' Only runs once per app load   loadDataFromLocalStorage();
checkAuthToken();   } }, []); // ...}
You can also run it during module initialization and before the app renders:
if (typeof window !== 'undefined') { // Check if we're running in the browser.   // ' Only runs once per app load checkAuthToken(); loadDataFromLocalStorage();}function App() { // ...}
Code at the top level runs once when your component is imported—even if it doesn't end up being rendered. To avoid slowdown or surprising behavior when importing arbitrary components, don't overuse this pattern. Keep app-wide initialization logic to root component modules like App.js or in your application's entry point.
Notifying parent components about state changes
Let's say you're writing a Toggle component with an internal isOn state which can be either true or false. There are a few different ways to toggle it (by clicking or dragging). You want to notify the parent component whenever the Toggle internal state changes, so you expose an onChange event and call it from an Effect:
function Toggle({ onChange }) { const [isOn, setIsOn] = useState(false); // Ø=Ý4 Avoid: The onChange handler runs too late useEffect(() => {   onChange(isOn); }, [isOn, onChange]) function handleClick() {   setIsOn(!isOn); } function handleDragEnd(e) {   if (isCloserToRightEdge(e)) {   setIsOn(true);   } else {   setIsOn(false);   } } // ...}
Like earlier, this is not ideal. The Toggle updates its state first, and React updates the screen. Then React runs the Effect, which calls the onChange function passed from a parent component. Now the parent component will update its own state, starting another render pass. It would be better to do everything in a single pass.
Delete the Effect and instead update the state of both components within the same event handler:
function Toggle({ onChange }) { const [isOn, setIsOn] = useState(false); function updateToggle(nextIsOn) {   // ' Good: Perform all updates during the event that caused them   setIsOn(nextIsOn);   onChange(nextIsOn); } function handleClick() {   updateToggle(!isOn); } function handleDragEnd(e) {   if (isCloserToRightEdge(e)) {   updateToggle(true);   } else {   updateToggle(false);   } } // ...}
With this approach, both the Toggle component and its parent component update their state during the event. React batches updates from different components together, so

there will only be one render pass.
You might also be able to remove the state altogether, and instead receive isOn from the parent component:
// ' Also good: the component is fully controlled by its parentfunction Toggle({ isOn, onChange }) { function handleClick() { onChange(!isOn); } function handleDragEnd(e) { if (isCloserToRightEdge(e)) { onChange(true); } else { onChange(false); } } // ...}
“Lifting state up” lets the parent component fully control the Toggle by toggling the parent’s own state. This means the parent component will have to contain more logic, but there will be less state overall to worry about. Whenever you try to keep two different state variables synchronized, try lifting state up instead!
Passing data to the parent
This Child component fetches some data and then passes it to the Parent component in an Effect:
function Parent() { const [data, setData] = useState(null); // ... return <Child onFetched={setData} />;}function Child({ onFetched }) { const data = useSomeAPI(); // Ø=Ý4 Avoid: Passing data to the parent in an Effect useEffect(() => { if (data) { onFetched(data); } }, [onFetched, data]); // ...}
In React, data flows from the parent components to their children. When you see something wrong on the screen, you can trace where the information comes from by going up the component chain until you find which component passes the wrong prop or has the wrong state. When child components update the state of their parent components in Effects, the data flow becomes very difficult to trace. Since both the child and the parent need the same data, let the parent component fetch that data, and pass it down to the child instead:
function Parent() { const data = useSomeAPI(); // ... // ' Good: Passing data down to the child return <Child data={data} />;}function Child({ data }) { // ...}
This is simpler and keeps the data flow predictable: the data flows down from the parent to the child.
Subscribing to an external store
Sometimes, your components may need to subscribe to some data outside of the React state. This data could be from a third-party library or a built-in browser API. Since this data can change without React’s knowledge, you need to manually subscribe your components to it. This is often done with an Effect, for example:
function useOnlineStatus() { // Not ideal: Manual store subscription in an Effect const [isOnline, setIsOnline] = useState(true); useEffect(() => { function updateState() { setIsOnline(navigator.onLine); } updateState(); window.addEventListener('online', updateState); window.addEventListener('offline', updateState); return () => { window.removeEventListener('online', updateState); window.removeEventListener('offline', updateState); }; }, []); return isOnline;}function ChatIndicator() { const isOnline = useOnlineStatus(); // ...}
Here, the component subscribes to an external data store (in this case, the browser navigator.onLine API). Since this API does not exist on the server (so it can’t be used for the initial HTML), initially the state is set to true. Whenever the value of that data store changes in the browser, the component updates its state.
Although it’s common to use Effects for this, React has a purpose-built Hook for

subscribing to an external store that is preferred instead. Delete the Effect and replace it with a call to useSyncExternalStore:
function subscribe(callback) { window.addEventListener('online', callback); window.addEventListener('offline', callback); return () => { window.removeEventListener('online', callback); window.removeEventListener('offline', callback); };}function useOnlineStatus() { // ' Good: Subscribing to an external store with a built-in Hook return useSyncExternalStore(subscribe, // React won't resubscribe for as long as you pass the same function () => navigator.onLine, // How to get the value on the client () => true // How to get the value on the server);}function ChatIndicator() { const isOnline = useOnlineStatus(); // ...}
This approach is less error-prone than manually syncing mutable data to React state with an Effect. Typically, you'll write a custom Hook like useOnlineStatus() above so that you don't need to repeat this code in the individual components. Read more about subscribing to external stores from React components.
Fetching data
Many apps use Effects to kick off data fetching. It is quite common to write a data fetching Effect like this:
function SearchResults({ query }) { const [results, setResults] = useState([]); const [page, setPage] = useState(1); useEffect(() => { // Ø=Ý4 Avoid: Fetching without cleanup logic fetchResults(query, page).then(json => { setResults(json); }); }, [query, page]); function handleNextPageClick() { setPage(page + 1); } // ...}
You don't need to move this fetch to an event handler.
This might seem like a contradiction with the earlier examples where you needed to put the logic into the event handlers! However, consider that it's not the typing event that's the main reason to fetch. Search inputs are often prepopulated from the URL, and the user might navigate Back and Forward without touching the input.
It doesn't matter where page and query come from. While this component is visible, you want to keep results synchronized with data from the network for the current page and query. This is why it's an Effect.
However, the code above has a bug. Imagine you type "hello" fast. Then the query will change from "h", to "he", "hel", "hell", and "hello". This will kick off separate fetches, but there is no guarantee about which order the responses will arrive in. For example, the "hell" response may arrive after the "hello" response. Since it will call setResults() last, you will be displaying the wrong search results. This is called a "race condition": two different requests "raced" against each other and came in a different order than you expected.
To fix the race condition, you need to add a cleanup function to ignore stale responses:
function SearchResults({ query }) { const [results, setResults] = useState([]); const [page, setPage] = useState(1); useEffect(() => { let ignore = false; fetchResults(query, page).then(json => { if (!ignore) { setResults(json); } }); return () => { ignore = true; }; }, [query, page]); function handleNextPageClick() { setPage(page + 1); } // ...}
This ensures that when your Effect fetches data, all responses except the last requested one will be ignored.
Handling race conditions is not the only difficulty with implementing data fetching. You

might also want to think about caching responses (so that the user can click Back and see the previous screen instantly), how to fetch data on the server (so that the initial server-rendered HTML contains the fetched content instead of a spinner), and how to avoid network waterfalls (so that a child can fetch data without waiting for every parent). These issues apply to any UI library, not just React. Solving them is not trivial, which is why modern frameworks provide more efficient built-in data fetching mechanisms than fetching data in Effects.
If you don’t use a framework (and don’t want to build your own) but would like to make data fetching from Effects more ergonomic, consider extracting your fetching logic into a custom Hook like in this example:
function SearchResults({ query }) { const [page, setPage] = useState(1); const params = new URLSearchParams({ query, page }); const results = useData(`/api/search?${params}`); function handleNextPageClick() { setPage(page + 1); } // ...}function useData(url) { const [data, setData] = useState(null); useEffect(() => { let ignore = false; fetch(url) .then(response => response.json()) .then(json => { if (!ignore) { setData(json); } }); return () => { ignore = true; }; }, [url]); return data;}
You’ll likely also want to add some logic for error handling and to track whether the content is loading. You can build a Hook like this yourself or use one of the many solutions already available in the React ecosystem. Although this alone won’t be as efficient as using a framework’s built-in data fetching mechanism, moving the data fetching logic into a custom Hook will make it easier to adopt an efficient data fetching strategy later.
In general, whenever you have to resort to writing Effects, keep an eye out for when you can extract a piece of functionality into a custom Hook with a more declarative and purpose-built API like useData above. The fewer raw useEffect calls you have in your components, the easier you will find to maintain your application.
Recap
If you can calculate something during render, you don’t need an Effect.
To cache expensive calculations, add useMemo instead of useEffect.
To reset the state of an entire component tree, pass a different key to it.
To reset a particular bit of state in response to a prop change, set it during rendering.
Code that runs because a component was displayed should be in Effects, the rest should be in events.
If you need to update the state of several components, it’s better to do it during a single event.
Whenever you try to synchronize state variables in different components, consider lifting state up.
You can fetch data with Effects, but you need to implement cleanup to avoid race conditions.

Try out some challenges1. Transform data without Effects 2. Cache a calculation without Effects 3. Reset state without Effects 4. Submit a form without Effects Challenge 1 of 4: Transform data without Effects The TodoList below displays a list of todos. When the “Show only active todos” checkbox is ticked, completed todos are not displayed in the list. Regardless of which todos are visible, the footer displays the count of todos that

are not yet completed.Simplify this component by removing all the unnecessary state and Effects.App.jstodos.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { initialTodos, createTodo } from './todos.js';

```
export default function TodoList() {
  const [todos, setTodos] = useState(initialTodos);
  const [showActive, setShowActive] = useState(false);
  const [activeTodos, setActiveTodos] = useState([]);
  const [visibleTodos, setVisibleTodos] = useState([]);
  const [footer, setFooter] = useState(null);

  useEffect(() => {
    setActiveTodos(todos.filter(todo => !todo.completed));
  }, [todos]);

  useEffect(() => {
    setVisibleTodos(showActive ? activeTodos : todos);
  }, [showActive, todos, activeTodos]);

  useEffect(() => {
    setFooter(
      <footer>
        {activeTodos.length} todos left
      </footer>
    );
  }, [activeTodos]);

  return (
    <>
      <label>
        <input
          type="checkbox"
          checked={showActive}
          onChange={e => setShowActive(e.target.checked)}
        />
        Show only active todos
      </label>
      <NewTodo onAdd={newTodo => setTodos([...todos, newTodo])} />
      <ul>
        {visibleTodos.map(todo => (
          <li key={todo.id}>
            {todo.completed ? <s>{todo.text}</s> : todo.text}
          </li>
        ))}
      </ul>
      {footer}
```

```
    </>
  );
}

function NewTodo({ onAdd }) {
  const [text, setText] = useState('');

  function handleAddClick() {
    setText('');
    onAdd(createTodo(text));
  }

  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <button onClick={handleAddClick}>
        Add
      </button>
    </>
  );
}
```

Show more Show hint Show solutionNext ChallengePreviousSynchronizing with EffectsNextLifecycle of Reactive Effects

Learn ReactEscape HatchesLifecycle of Reactive EffectsEffects have a different lifecycle from components. Components may mount, update, or unmount. An Effect can only do two things: to start synchronizing something, and later to stop synchronizing it. This cycle can happen multiple times if your Effect depends on props and state that change over time. React provides a linter rule to check that you've specified your Effect's dependencies correctly. This keeps your Effect synchronized to the latest props and state.

You will learn

How an Effect's lifecycle is different from a component's lifecycle
How to think about each individual Effect in isolation
When your Effect needs to re-synchronize, and why
How your Effect's dependencies are determined
What it means for a value to be reactive
What an empty dependency array means
How React verifies your dependencies are correct with a linter
What to do when you disagree with the linter

The lifecycle of an Effect

Every React component goes through the same lifecycle:

A component mounts when it's added to the screen.
A component updates when it receives new props or state, usually in response to an interaction.

A component unmounts when it’s removed from the screen.

It’s a good way to think about components, but not about Effects. Instead, try to think about each Effect independently from your component’s lifecycle. An Effect describes how to synchronize an external system to the current props and state. As your code changes, synchronization will need to happen more or less often.

To illustrate this point, consider this Effect connecting your component to a chat server:

```
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) {  useEffect(() => {    const connection = createConnection(serverUrl, roomId);
connection.connect();    return () => {      connection.disconnect();    };  }, [roomId]);  // ...}
```

Your Effect’s body specifies how to start synchronizing:

```
    // ...    const connection = createConnection(serverUrl, roomId);
connection.connect();    return () => {      connection.disconnect();    };    // ...
```

The cleanup function returned by your Effect specifies how to stop synchronizing:

```
    // ...    const connection = createConnection(serverUrl, roomId);
connection.connect();    return () => {      connection.disconnect();    };    // ...
```

Intuitively, you might think that React would start synchronizing when your component mounts and stop synchronizing when your component unmounts. However, this is not the end of the story! Sometimes, it may also be necessary to start and stop synchronizing multiple times while the component remains mounted.

Let’s look at why this is necessary, when it happens, and how you can control this behavior.

NoteSome Effects don’t return a cleanup function at all. More often than not, you’ll want to return one—but if you don’t, React will behave as if you returned an empty cleanup function.

Why synchronization may need to happen more than once

Imagine this ChatRoom component receives a roomId prop that the user picks in a dropdown. Let’s say that initially the user picks the "general" room as the roomId. Your app displays the "general" chat room:

```
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId /* "general" */ }) {  // ...  return <h1>Welcome to the {roomId} room!</h1>;}
```

After the UI is displayed, React will run your Effect to start synchronizing. It connects to the "general" room:

```
function ChatRoom({ roomId /* "general" */ }) {  useEffect(() => {    const connection = createConnection(serverUrl, roomId); // Connects to the "general" room
connection.connect();    return () => {      connection.disconnect(); // Disconnects from the "general" room    };  }, [roomId]);  // ...
```

So far, so good.

Later, the user picks a different room in the dropdown (for example, "travel"). First, React will update the UI:

```
function ChatRoom({ roomId /* "travel" */ }) {  // ...  return <h1>Welcome to the {roomId} room!</h1>;}
```

Think about what should happen next. The user sees that "travel" is the selected chat room in the UI. However, the Effect that ran the last time is still connected to the "general" room. The roomId prop has changed, so what your Effect did back then (connecting to the "general" room) no longer matches the UI.

At this point, you want React to do two things:

Stop synchronizing with the old roomId (disconnect from the "general" room)
Start synchronizing with the new roomId (connect to the "travel" room)

Luckily, you've already taught React how to do both of these things! Your Effect's body specifies how to start synchronizing, and your cleanup function specifies how to stop synchronizing. All that React needs to do now is to call them in the correct order and with the correct props and state. Let's see how exactly that happens.

How React re-synchronizes your Effect

Recall that your ChatRoom component has received a new value for its roomId prop. It used to be "general", and now it is "travel". React needs to re-synchronize your Effect to re-connect you to a different room.

To stop synchronizing, React will call the cleanup function that your Effect returned after connecting to the "general" room. Since roomId was "general", the cleanup function disconnects from the "general" room:

```
function ChatRoom({ roomId /* "general" */ }) {  useEffect(() => {    const connection = createConnection(serverUrl, roomId); // Connects to the "general" room    connection.connect();    return () => {      connection.disconnect(); // Disconnects from the "general" room    };    // ...
```

Then React will run the Effect that you've provided during this render. This time, roomId is "travel" so it will start synchronizing to the "travel" chat room (until its cleanup function is eventually called too):

```
function ChatRoom({ roomId /* "travel" */ }) {  useEffect(() => {    const connection = createConnection(serverUrl, roomId); // Connects to the "travel" room    connection.connect();    // ...
```

Thanks to this, you're now connected to the same room that the user chose in the UI. Disaster averted!

Every time after your component re-renders with a different roomId, your Effect will re-synchronize. For example, let's say the user changes roomId from "travel" to "music". React will again stop synchronizing your Effect by calling its cleanup function (disconnecting you from the "travel" room). Then it will start synchronizing again by running its body with the new roomId prop (connecting you to the "music" room).

Finally, when the user goes to a different screen, ChatRoom unmounts. Now there is no need to stay connected at all. React will stop synchronizing your Effect one last time and disconnect you from the "music" chat room.

Thinking from the Effect's perspective

Let's recap everything that's happened from the ChatRoom component's perspective:

ChatRoom mounted with roomId set to "general"
ChatRoom updated with roomId set to "travel"
ChatRoom updated with roomId set to "music"
ChatRoom unmounted

During each of these points in the component's lifecycle, your Effect did different things:

Your Effect connected to the "general" room
Your Effect disconnected from the "general" room and connected to the "travel" room
Your Effect disconnected from the "travel" room and connected to the "music" room
Your Effect disconnected from the "music" room

Now let’s think about what happened from the perspective of the Effect itself:
useEffect(() => { // Your Effect connected to the room specified with roomId... const connection = createConnection(serverUrl, roomId); connection.connect(); return () => { // ...until it disconnected connection.disconnect(); }; }, [roomId]);
This code’s structure might inspire you to see what happened as a sequence of non-overlapping time periods:

Your Effect connected to the "general" room (until it disconnected)
Your Effect connected to the "travel" room (until it disconnected)
Your Effect connected to the "music" room (until it disconnected)

Previously, you were thinking from the component’s perspective. When you looked from the component’s perspective, it was tempting to think of Effects as “callbacks” or “lifecycle events” that fire at a specific time like “after a render” or “before unmount”. This way of thinking gets complicated very fast, so it’s best to avoid.
Instead, always focus on a single start/stop cycle at a time. It shouldn’t matter whether a component is mounting, updating, or unmounting. All you need to do is to describe how to start synchronization and how to stop it. If you do it well, your Effect will be resilient to being started and stopped as many times as it’s needed.
This might remind you how you don’t think whether a component is mounting or updating when you write the rendering logic that creates JSX. You describe what should be on the screen, and React figures out the rest.
How React verifies that your Effect can re-synchronize
Here is a live example that you can play with. Press “Open chat” to mount the ChatRoom component:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);
  return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
```

```
  const [show, setShow] = useState(false);
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <button onClick={() => setShow(!show)}>
        {show ? 'Close chat' : 'Open chat'}
      </button>
      {show && <hr />}
      {show && <ChatRoom roomId={roomId} />}
    </>
  );
}
```

Show more

Notice that when the component mounts for the first time, you see three logs:

```
'  Connecting to "general" room at https://localhost:1234... (development-only)
'L Disconnected from "general" room at https://localhost:1234. (development-only)
'  Connecting to "general" room at https://localhost:1234...
```

The first two logs are development-only. In development, React always remounts each component once.

React verifies that your Effect can re-synchronize by forcing it to do that immediately in development. This might remind you of opening a door and closing it an extra time to check if the door lock works. React starts and stops your Effect one extra time in development to check you've implemented its cleanup well.

The main reason your Effect will re-synchronize in practice is if some data it uses has changed. In the sandbox above, change the selected chat room. Notice how, when the roomId changes, your Effect re-synchronizes.

However, there are also more unusual cases in which re-synchronization is necessary. For example, try editing the serverUrl in the sandbox above while the chat is open. Notice how the Effect re-synchronizes in response to your edits to the code. In the future, React may add more features that rely on re-synchronization.

How React knows that it needs to re-synchronize the Effect

You might be wondering how React knew that your Effect needed to re-synchronize after roomId changes. It's because you told React that its code depends on roomId by

including it in the list of dependencies:
function ChatRoom({ roomId }) { // The roomId prop may change over time useEffect(() => { const connection = createConnection(serverUrl, roomId); // This Effect reads roomId connection.connect(); return () => { connection.disconnect(); }; }, [roomId]); // So you tell React that this Effect "depends on" roomId // ...
Here's how this works:

You knew roomId is a prop, which means it can change over time.
You knew that your Effect reads roomId (so its logic depends on a value that may change later).
This is why you specified it as your Effect's dependency (so that it re-synchronizes when roomId changes).

Every time after your component re-renders, React will look at the array of dependencies that you have passed. If any of the values in the array is different from the value at the same spot that you passed during the previous render, React will re-synchronize your Effect.
For example, if you passed ["general"] during the initial render, and later you passed ["travel"] during the next render, React will compare "general" and "travel". These are different values (compared with Object.is), so React will re-synchronize your Effect. On the other hand, if your component re-renders but roomId has not changed, your Effect will remain connected to the same room.
Each Effect represents a separate synchronization process
Resist adding unrelated logic to your Effect only because this logic needs to run at the same time as an Effect you already wrote. For example, let's say you want to send an analytics event when the user visits the room. You already have an Effect that depends on roomId, so you might feel tempted to add the analytics call there:
function ChatRoom({ roomId }) { useEffect(() => { logVisit(roomId); const connection = createConnection(serverUrl, roomId); connection.connect(); return () => { connection.disconnect(); }; }, [roomId]); // ...}
But imagine you later add another dependency to this Effect that needs to re-establish the connection. If this Effect re-synchronizes, it will also call logVisit(roomId) for the same room, which you did not intend. Logging the visit is a separate process from connecting. Write them as two separate Effects:
function ChatRoom({ roomId }) { useEffect(() => { logVisit(roomId); }, [roomId]); useEffect(() => { const connection = createConnection(serverUrl, roomId); // ... }, [roomId]); // ...}
Each Effect in your code should represent a separate and independent synchronization process.
In the above example, deleting one Effect wouldn't break the other Effect's logic. This is a good indication that they synchronize different things, and so it made sense to split them up. On the other hand, if you split up a cohesive piece of logic into separate Effects, the code may look "cleaner" but will be more difficult to maintain. This is why you should think whether the processes are same or separate, not whether the code looks cleaner.
Effects "react" to reactive values

Your Effect reads two variables (serverUrl and roomId), but you only specified roomId as a dependency:
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) { useEffect(() => {   const connection = createConnection(serverUrl, roomId);
connection.connect();   return () => {   connection.disconnect();   }; }, [roomId]); // ...}
Why doesn’t serverUrl need to be a dependency?
This is because the serverUrl never changes due to a re-render. It’s always the same no matter how many times the component re-renders and why. Since serverUrl never changes, it wouldn’t make sense to specify it as a dependency. After all, dependencies only do something when they change over time!
On the other hand, roomId may be different on a re-render. Props, state, and other values declared inside the component are reactive because they’re calculated during rendering and participate in the React data flow.
If serverUrl was a state variable, it would be reactive. Reactive values must be included in dependencies:
function ChatRoom({ roomId }) { // Props change over time const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // State may change over time
useEffect(() => {   const connection = createConnection(serverUrl, roomId); // Your Effect reads props and state   connection.connect();   return () => {   connection.disconnect();   }; }, [roomId, serverUrl]); // So you tell React that this Effect "depends on" on props and state // ...}
By including serverUrl as a dependency, you ensure that the Effect re-synchronizes after it changes.
Try changing the selected chat room or edit the server URL in this sandbox:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]);

  return (
    <>
      <label>
        Server URL:{' '}
        <input
          value={serverUrl}
          onChange={e => setServerUrl(e.target.value)}
        />
      </label>
      <h1>Welcome to the {roomId} room!</h1>
```

```
    </>
  );
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
}
```

Show more

Whenever you change a reactive value like roomId or serverUrl, the Effect re-connects to the chat server.

What an Effect with empty dependencies means

What happens if you move both serverUrl and roomId outside the component?

const serverUrl = 'https://localhost:1234';const roomId = 'general';function ChatRoom() { useEffect(() => {   const connection = createConnection(serverUrl, roomId); connection.connect();   return () => {   connection.disconnect();   }; }, []); // ' All dependencies declared // ...}

Now your Effect's code does not use any reactive values, so its dependencies can be empty ([]).

Thinking from the component's perspective, the empty [] dependency array means this Effect connects to the chat room only when the component mounts, and disconnects only when the component unmounts. (Keep in mind that React would still re-synchronize it an extra time in development to stress-test your logic.)

App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react'; import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';
const roomId = 'general';
```

```
function ChatRoom() {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []);
  return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
  const [show, setShow] = useState(false);
  return (
    <>
      <button onClick={() => setShow(!show)}>
        {show ? 'Close chat' : 'Open chat'}
      </button>
      {show && <hr />}
      {show && <ChatRoom />}
    </>
  );
}
```

Show more

However, if you think from the Effect's perspective, you don't need to think about mounting and unmounting at all. What's important is you've specified what your Effect does to start and stop synchronizing. Today, it has no reactive dependencies. But if you ever want the user to change roomId or serverUrl over time (and they would become reactive), your Effect's code won't change. You will only need to add them to the dependencies.

All variables declared in the component body are reactive

Props and state aren't the only reactive values. Values that you calculate from them are also reactive. If the props or state change, your component will re-render, and the values calculated from them will also change. This is why all variables from the component body used by the Effect should be in the Effect dependency list.

Let's say that the user can pick a chat server in the dropdown, but they can also configure a default server in settings. Suppose you've already put the settings state in a context so you read the settings from that context. Now you calculate the serverUrl based on the selected server from props and the default server:

function ChatRoom({ roomId, selectedServerUrl }) { // roomId is reactive const settings = useContext(SettingsContext); // settings is reactive const serverUrl = selectedServerUrl ?? settings.defaultServerUrl; // serverUrl is reactive useEffect(() => { const connection = createConnection(serverUrl, roomId); // Your Effect reads roomId and serverUrl connection.connect(); return () => { connection.disconnect(); }; }, [roomId, serverUrl]); // So it needs to re-synchronize when either of them changes! // ...}

In this example, serverUrl is not a prop or a state variable. It's a regular variable that you calculate during rendering. But it's calculated during rendering, so it can change

due to a re-render. This is why it’s reactive.
All values inside the component (including props, state, and variables in your component’s body) are reactive. Any reactive value can change on a re-render, so you need to include reactive values as Effect’s dependencies.
In other words, Effects “react” to all values from the component body.
Deep DiveCan global or mutable values be dependencies? Show DetailsMutable values (including global variables) aren’t reactive.A mutable value like location.pathname can’t be a dependency. It’s mutable, so it can change at any time completely outside of the React rendering data flow. Changing it wouldn’t trigger a re-render of your component. Therefore, even if you specified it in the dependencies, React wouldn’t know to re-synchronize the Effect when it changes. This also breaks the rules of React because reading mutable data during rendering (which is when you calculate the dependencies) breaks purity of rendering. Instead, you should read and subscribe to an external mutable value with useSyncExternalStore.A mutable value like ref.current or things you read from it also can’t be a dependency. The ref object returned by useRef itself can be a dependency, but its current property is intentionally mutable. It lets you keep track of something without triggering a re-render. But since changing it doesn’t trigger a re-render, it’s not a reactive value, and React won’t know to re-run your Effect when it changes.As you’ll learn below on this page, a linter will check for these issues automatically.
React verifies that you specified every reactive value as a dependency
If your linter is configured for React, it will check that every reactive value used by your Effect’s code is declared as its dependency. For example, this is a lint error because both roomId and serverUrl are reactive:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) { // roomId is reactive
  const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // serverUrl is reactive

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // <-- Something's wrong here!

  return (
    <>
      <label>
        Server URL:{' '}
        <input
          value={serverUrl}
          onChange={e => setServerUrl(e.target.value)}
        />
      </label>

```
      <h1>Welcome to the {roomId} room!</h1>
    </>
  );
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
}
```

Show more

This may look like a React error, but really React is pointing out a bug in your code. Both roomId and serverUrl may change over time, but you're forgetting to re-synchronize your Effect when they change. You will remain connected to the initial roomId and serverUrl even after the user picks different values in the UI.

To fix the bug, follow the linter's suggestion to specify roomId and serverUrl as dependencies of your Effect:

function ChatRoom({ roomId }) { // roomId is reactive const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // serverUrl is reactive useEffect(() => {   const connection = createConnection(serverUrl, roomId);   connection.connect();   return () => {   connection.disconnect();   }; }, [serverUrl, roomId]); // ' All dependencies declared // ...}

Try this fix in the sandbox above. Verify that the linter error is gone, and the chat re-connects when needed.

NoteIn some cases, React knows that a value never changes even though it's declared inside the component. For example, the set function returned from useState and the ref object returned by useRef are stable—they are guaranteed to not change on a re-render. Stable values aren't reactive, so you may omit them from the list. Including them is allowed: they won't change, so it doesn't matter.

What to do when you don't want to re-synchronize

In the previous example, you've fixed the lint error by listing roomId and serverUrl as dependencies.
However, you could instead "prove" to the linter that these values aren't reactive values, i.e. that they can't change as a result of a re-render. For example, if serverUrl and roomId don't depend on rendering and always have the same values, you can move them outside the component. Now they don't need to be dependencies:
const serverUrl = 'https://localhost:1234'; // serverUrl is not reactiveconst roomId = 'general'; // roomId is not reactivefunction ChatRoom() { useEffect(() => {   const connection = createConnection(serverUrl, roomId);   connection.connect();   return () => {   connection.disconnect();   }; }, []); // ' All dependencies declared // ...}
You can also move them inside the Effect. They aren't calculated during rendering, so they're not reactive:
function ChatRoom() { useEffect(() => {   const serverUrl = 'https://localhost:1234'; // serverUrl is not reactive   const roomId = 'general'; // roomId is not reactive   const connection = createConnection(serverUrl, roomId);   connection.connect();   return () => {   connection.disconnect();   }; }, []); // ' All dependencies declared // ...}
Effects are reactive blocks of code. They re-synchronize when the values you read inside of them change. Unlike event handlers, which only run once per interaction, Effects run whenever synchronization is necessary.
You can't "choose" your dependencies. Your dependencies must include every reactive value you read in the Effect. The linter enforces this. Sometimes this may lead to problems like infinite loops and to your Effect re-synchronizing too often. Don't fix these problems by suppressing the linter! Here's what to try instead:

Check that your Effect represents an independent synchronization process. If your Effect doesn't synchronize anything, it might be unnecessary. If it synchronizes several independent things, split it up.

If you want to read the latest value of props or state without "reacting" to it and re-synchronizing the Effect, you can split your Effect into a reactive part (which you'll keep in the Effect) and a non-reactive part (which you'll extract into something called an Effect Event). Read about separating Events from Effects.

Avoid relying on objects and functions as dependencies. If you create objects and functions during rendering and then read them from an Effect, they will be different on every render. This will cause your Effect to re-synchronize every time. Read more about removing unnecessary dependencies from Effects.

PitfallThe linter is your friend, but its powers are limited. The linter only knows when the dependencies are wrong. It doesn't know the best way to solve each case. If the linter suggests a dependency, but adding it causes a loop, it doesn't mean the linter should be ignored. You need to change the code inside (or outside) the Effect so that that value

isn’t reactive and doesn’t need to be a dependency.If you have an existing codebase, you might have some Effects that suppress the linter like this:useEffect(() => { // ... // Ø=Ý4 Avoid suppressing the linter like this: // eslint-ignore-next-line react-hooks/exhaustive-deps}, []);On the next pages, you’ll learn how to fix this code without breaking the rules. It’s always worth fixing!

Recap

Components can mount, update, and unmount.
Each Effect has a separate lifecycle from the surrounding component.
Each Effect describes a separate synchronization process that can start and stop.
When you write and read Effects, think from each individual Effect’s perspective (how to start and stop synchronization) rather than from the component’s perspective (how it mounts, updates, or unmounts).
Values declared inside the component body are “reactive”.
Reactive values should re-synchronize the Effect because they can change over time.
The linter verifies that all reactive values used inside the Effect are specified as dependencies.
All errors flagged by the linter are legitimate. There’s always a way to fix the code to not break the rules.

Try out some challenges1. Fix reconnecting on every keystroke 2. Switch synchronization on and off 3. Investigate a stale value bug 4. Fix a connection switch 5. Populate a chain of select boxes Challenge 1 of 5: Fix reconnecting on every keystroke In this example, the ChatRoom component connects to the chat room when the component mounts, disconnects when it unmounts, and reconnects when you select a different chat room. This behavior is correct, so you need to keep it working.However, there is a problem. Whenever you type into the message box input at the bottom, ChatRoom also reconnects to the chat. (You can notice this by clearing the console and typing into the input.) Fix the issue so that this doesn’t happen.App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  });

  return (
    <>
      <h1>Welcome to the {roomId} room!</h1>
      <input
```

```
        value={message}
        onChange={e => setMessage(e.target.value)}
      />
    </>
  );
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
}
```

Show more Show hint Show solutionNext ChallengePreviousYou Might Not Need an EffectNextSeparating Events from Effects

Learn ReactEscape HatchesSeparating Events from EffectsEvent handlers only re-run when you perform the same interaction again. Unlike event handlers, Effects re-synchronize if some value they read, like a prop or a state variable, is different from what it was during the last render. Sometimes, you also want a mix of both behaviors: an Effect that re-runs in response to some values but not others. This page will teach you how to do that.

You will learn

How to choose between an event handler and an Effect
Why Effects are reactive, and event handlers are not
What to do when you want a part of your Effect's code to not be reactive
What Effect Events are, and how to extract them from your Effects
How to read the latest props and state from Effects using Effect Events

Choosing between event handlers and Effects

First, let's recap the difference between event handlers and Effects.

Imagine you're implementing a chat room component. Your requirements look like this:

Your component should automatically connect to the selected chat room.
When you click the “Send” button, it should send a message to the chat.

Let’s say you’ve already implemented the code for them, but you’re not sure where to put it. Should you use event handlers or Effects? Every time you need to answer this question, consider why the code needs to run.
Event handlers run in response to specific interactions
From the user’s perspective, sending a message should happen because the particular “Send” button was clicked. The user will get rather upset if you send their message at any other time or for any other reason. This is why sending a message should be an event handler. Event handlers let you handle specific interactions:
function ChatRoom({ roomId }) { const [message, setMessage] = useState(''); // ... function handleSendClick() { sendMessage(message); } // ... return (<> <input value={message} onChange={e => setMessage(e.target.value)} /> <button onClick={handleSendClick}>Send</button>; </>);}
With an event handler, you can be sure that sendMessage(message) will only run if the user presses the button.
Effects run whenever synchronization is needed
Recall that you also need to keep the component connected to the chat room. Where does that code go?
The reason to run this code is not some particular interaction. It doesn’t matter why or how the user navigated to the chat room screen. Now that they’re looking at it and could interact with it, the component needs to stay connected to the selected chat server. Even if the chat room component was the initial screen of your app, and the user has not performed any interactions at all, you would still need to connect. This is why it’s an Effect:
function ChatRoom({ roomId }) { // ... useEffect(() => { const connection = createConnection(serverUrl, roomId); connection.connect(); return () => { connection.disconnect(); }; }, [roomId]); // ...}
With this code, you can be sure that there is always an active connection to the currently selected chat server, regardless of the specific interactions performed by the user. Whether the user has only opened your app, selected a different room, or navigated to another screen and back, your Effect ensures that the component will remain synchronized with the currently selected room, and will re-connect whenever it’s necessary.
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection, sendMessage } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
```

```
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);

  function handleSendClick() {
    sendMessage(message);
  }

  return (
    <>
      <h1>Welcome to the {roomId} room!</h1>
      <input value={message} onChange={e => setMessage(e.target.value)} />
      <button onClick={handleSendClick}>Send</button>
    </>
  );
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  const [show, setShow] = useState(false);
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <button onClick={() => setShow(!show)}>
        {show ? 'Close chat' : 'Open chat'}
      </button>
      {show && <hr />}
      {show && <ChatRoom roomId={roomId} />}
    </>
  );
}
```

Show more

Reactive values and reactive logic

Intuitively, you could say that event handlers are always triggered “manually”, for example by clicking a button. Effects, on the other hand, are “automatic”: they run and

re-run as often as it’s needed to stay synchronized.
There is a more precise way to think about this.
Props, state, and variables declared inside your component’s body are called reactive values. In this example, serverUrl is not a reactive value, but roomId and message are. They participate in the rendering data flow:
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) { const [message, setMessage] = useState(''); // ...}
Reactive values like these can change due to a re-render. For example, the user may edit the message or choose a different roomId in a dropdown. Event handlers and Effects respond to changes differently:

Logic inside event handlers is not reactive. It will not run again unless the user performs the same interaction (e.g. a click) again. Event handlers can read reactive values without “reacting” to their changes.
Logic inside Effects is reactive. If your Effect reads a reactive value, you have to specify it as a dependency. Then, if a re-render causes that value to change, React will re-run your Effect’s logic with the new value.

Let’s revisit the previous example to illustrate this difference.
Logic inside event handlers is not reactive
Take a look at this line of code. Should this logic be reactive or not?
    // ...   sendMessage(message);   // ...
From the user’s perspective, a change to the message does not mean that they want to send a message. It only means that the user is typing. In other words, the logic that sends a message should not be reactive. It should not run again only because the reactive value has changed. That’s why it belongs in the event handler:
  function handleSendClick() {   sendMessage(message); }
Event handlers aren’t reactive, so sendMessage(message) will only run when the user clicks the Send button.
Logic inside Effects is reactive
Now let’s return to these lines:
    // ...   const connection = createConnection(serverUrl, roomId);
connection.connect();   // ...
From the user’s perspective, a change to the roomId does mean that they want to connect to a different room. In other words, the logic for connecting to the room should be reactive. You want these lines of code to “keep up” with the reactive value, and to run again if that value is different. That’s why it belongs in an Effect:
  useEffect(() => {   const connection = createConnection(serverUrl, roomId);
connection.connect();   return () => {   connection.disconnect()   }; }, [roomId]);
Effects are reactive, so createConnection(serverUrl, roomId) and connection.connect() will run for every distinct value of roomId. Your Effect keeps the chat connection synchronized to the currently selected room.
Extracting non-reactive logic out of Effects
Things get more tricky when you want to mix reactive logic with non-reactive logic.
For example, imagine that you want to show a notification when the user connects to the chat. You read the current theme (dark or light) from the props so that you can show

the notification in the correct color:
function ChatRoom({ roomId, theme }) { useEffect(() => {   const connection = createConnection(serverUrl, roomId);   connection.on('connected', () => {   showNotification('Connected!', theme);   });   connection.connect();   // ...
However, theme is a reactive value (it can change as a result of re-rendering), and every reactive value read by an Effect must be declared as its dependency. Now you have to specify theme as a dependency of your Effect:
function ChatRoom({ roomId, theme }) { useEffect(() => {   const connection = createConnection(serverUrl, roomId);   connection.on('connected', () => {   showNotification('Connected!', theme);   });   connection.connect();   return () => {   connection.disconnect()   }; }, [roomId, theme]); // ' All dependencies declared // ...
Play with this example and see if you can spot the problem with this user experience:
App.jschat.jsnotifications.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId, theme }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.on('connected', () => {
      showNotification('Connected!', theme);
    });
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, theme]);

  return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  const [isDark, setIsDark] = useState(false);
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>

```
      </label>
      <label>
        <input
          type="checkbox"
          checked={isDark}
          onChange={e => setIsDark(e.target.checked)}
        />
        Use dark theme
      </label>
      <hr />
      <ChatRoom
        roomId={roomId}
        theme={isDark ? 'dark' : 'light'}
      />
    </>
  );
}
```

Show more

When the roomId changes, the chat re-connects as you would expect. But since theme is also a dependency, the chat also re-connects every time you switch between the dark and the light theme. That's not great!

In other words, you don't want this line to be reactive, even though it is inside an Effect (which is reactive):

```
      // ...      showNotification('Connected!', theme);      // ...
```

You need a way to separate this non-reactive logic from the reactive Effect around it.

Declaring an Effect Event

Under ConstructionThis section describes an experimental API that has not yet been released in a stable version of React.

Use a special Hook called useEffectEvent to extract this non-reactive logic out of your Effect:

```
import { useEffect, useEffectEvent } from 'react';function ChatRoom({ roomId, theme }) { const onConnected = useEffectEvent(() => {    showNotification('Connected!', theme);  });  // ...
```

Here, onConnected is called an Effect Event. It's a part of your Effect logic, but it behaves a lot more like an event handler. The logic inside it is not reactive, and it always "sees" the latest values of your props and state.

Now you can call the onConnected Effect Event from inside your Effect:

```
function ChatRoom({ roomId, theme }) {  const onConnected = useEffectEvent(() => {    showNotification('Connected!', theme);  });  useEffect(() => {    const connection = createConnection(serverUrl, roomId);    connection.on('connected', () => {      onConnected();    });    connection.connect();    return () => connection.disconnect();  }, [roomId]); // '  All dependencies declared  // ...
```

This solves the problem. Note that you had to remove onConnected from the list of your Effect's dependencies. Effect Events are not reactive and must be omitted from dependencies.

Verify that the new behavior works as you would expect:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId, theme }) {
  const onConnected = useEffectEvent(() => {
    showNotification('Connected!', theme);
  });

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.on('connected', () => {
      onConnected();
    });
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);

  return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  const [isDark, setIsDark] = useState(false);
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <label>
        <input
          type="checkbox"
          checked={isDark}
          onChange={e => setIsDark(e.target.checked)}
```

```
        />
        Use dark theme
      </label>
      <hr />
      <ChatRoom
        roomId={roomId}
        theme={isDark ? 'dark' : 'light'}
      />
    </>
  );
}
```

Show more

You can think of Effect Events as being very similar to event handlers. The main difference is that event handlers run in response to a user interactions, whereas Effect Events are triggered by you from Effects. Effect Events let you “break the chain” between the reactivity of Effects and code that should not be reactive.

Reading latest props and state with Effect Events

Under ConstructionThis section describes an experimental API that has not yet been released in a stable version of React.

Effect Events let you fix many patterns where you might be tempted to suppress the dependency linter.

For example, say you have an Effect to log the page visits:

```
function Page() {  useEffect(() => {    logVisit();  }, []);  // ...}
```

Later, you add multiple routes to your site. Now your Page component receives a url prop with the current path. You want to pass the url as a part of your logVisit call, but the dependency linter complains:

```
function Page({ url }) {  useEffect(() => {    logVisit(url);  }, []); // Ø=Ý4 React Hook useEffect has a missing dependency: 'url'  // ...}
```

Think about what you want the code to do. You want to log a separate visit for different URLs since each URL represents a different page. In other words, this logVisit call should be reactive with respect to the url. This is why, in this case, it makes sense to follow the dependency linter, and add url as a dependency:

```
function Page({ url }) {  useEffect(() => {    logVisit(url);  }, [url]); // '  All dependencies declared  // ...}
```

Now let’s say you want to include the number of items in the shopping cart together with every page visit:

```
function Page({ url }) {  const { items } = useContext(ShoppingCartContext);  const numberOfItems = items.length;  useEffect(() => {    logVisit(url, numberOfItems);  }, [url]); // Ø=Ý4 React Hook useEffect has a missing dependency: 'numberOfItems'  // ...}
```

You used numberOfItems inside the Effect, so the linter asks you to add it as a dependency. However, you don’t want the logVisit call to be reactive with respect to numberOfItems. If the user puts something into the shopping cart, and the numberOfItems changes, this does not mean that the user visited the page again. In other words, visiting the page is, in some sense, an “event”. It happens at a precise moment in time.

Split the code in two parts:
function Page({ url }) { const { items } = useContext(ShoppingCartContext); const numberOfItems = items.length; const onVisit = useEffectEvent(visitedUrl => {   logVisit(visitedUrl, numberOfItems); }); useEffect(() => {   onVisit(url); }, [url]); // ' All dependencies declared // ...}
Here, onVisit is an Effect Event. The code inside it isn't reactive. This is why you can use numberOfItems (or any other reactive value!) without worrying that it will cause the surrounding code to re-execute on changes.
On the other hand, the Effect itself remains reactive. Code inside the Effect uses the url prop, so the Effect will re-run after every re-render with a different url. This, in turn, will call the onVisit Effect Event.
As a result, you will call logVisit for every change to the url, and always read the latest numberOfItems. However, if numberOfItems changes on its own, this will not cause any of the code to re-run.
NoteYou might be wondering if you could call onVisit() with no arguments, and read the url inside it: const onVisit = useEffectEvent(() => {   logVisit(url, numberOfItems); }); useEffect(() => {   onVisit(); }, [url]);This would work, but it's better to pass this url to the Effect Event explicitly. By passing url as an argument to your Effect Event, you are saying that visiting a page with a different url constitutes a separate "event" from the user's perspective. The visitedUrl is a part of the "event" that happened: const onVisit = useEffectEvent(visitedUrl => {   logVisit(visitedUrl, numberOfItems); }); useEffect(() => {   onVisit(url); }, [url]);Since your Effect Event explicitly "asks" for the visitedUrl, now you can't accidentally remove url from the Effect's dependencies. If you remove the url dependency (causing distinct page visits to be counted as one), the linter will warn you about it. You want onVisit to be reactive with regards to the url, so instead of reading the url inside (where it wouldn't be reactive), you pass it from your Effect.This becomes especially important if there is some asynchronous logic inside the Effect: const onVisit = useEffectEvent(visitedUrl => {   logVisit(visitedUrl, numberOfItems); }); useEffect(() => {   setTimeout(() => {   onVisit(url);   }, 5000); // Delay logging visits }, [url]);Here, url inside onVisit corresponds to the latest url (which could have already changed), but visitedUrl corresponds to the url that originally caused this Effect (and this onVisit call) to run.
Deep DiveIs it okay to suppress the dependency linter instead? Show DetailsIn the existing codebases, you may sometimes see the lint rule suppressed like this:function Page({ url }) { const { items } = useContext(ShoppingCartContext); const numberOfItems = items.length; useEffect(() => {   logVisit(url, numberOfItems);   // Ø=Ý4 Avoid suppressing the linter like this: // eslint-disable-next-line react-hooks/exhaustive-deps }, [url]); // ...}After useEffectEvent becomes a stable part of React, we recommend never suppressing the linter.The first downside of suppressing the rule is that React will no longer warn you when your Effect needs to "react" to a new reactive dependency you've introduced to your code. In the earlier example, you added url to the dependencies because React reminded you to do it. You will no longer get such reminders for any future edits to that Effect if you disable the linter. This leads to bugs.Here is an example of a confusing bug caused by suppressing the linter. In this example, the handleMove function is supposed to read the current canMove state variable value in order to decide whether the dot should follow the cursor. However,

canMove is always true inside handleMove.Can you see why?App.jsApp.js Download ResetForkimport { useState, useEffect } from 'react';

```
export default function App() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const [canMove, setCanMove] = useState(true);

  function handleMove(e) {
    if (canMove) {
      setPosition({ x: e.clientX, y: e.clientY });
    }
  }

  useEffect(() => {
    window.addEventListener('pointermove', handleMove);
    return () => window.removeEventListener('pointermove', handleMove);
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []);

  return (
    <>
      <label>
        <input type="checkbox"
          checked={canMove}
          onChange={e => setCanMove(e.target.checked)}
        />
        The dot is allowed to move
      </label>
      <hr />
      <div style={{
        position: 'absolute',
        backgroundColor: 'pink',
        borderRadius: '50%',
        opacity: 0.6,
        transform: `translate(${position.x}px, ${position.y}px)`,
        pointerEvents: 'none',
        left: -20,
        top: -20,
        width: 40,
        height: 40,
      }} />
    </>
  );
}
```

Show moreThe problem with this code is in suppressing the dependency linter. If you

remove the suppression, you'll see that this Effect should depend on the handleMove function. This makes sense: handleMove is declared inside the component body, which makes it a reactive value. Every reactive value must be specified as a dependency, or it can potentially get stale over time!The author of the original code has "lied" to React by saying that the Effect does not depend ([]) on any reactive values. This is why React did not re-synchronize the Effect after canMove has changed (and handleMove with it). Because React did not re-synchronize the Effect, the handleMove attached as a listener is the handleMove function created during the initial render. During the initial render, canMove was true, which is why handleMove from the initial render will forever see that value.If you never suppress the linter, you will never see problems with stale values.With useEffectEvent, there is no need to "lie" to the linter, and the code works as you would expect:App.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

```
export default function App() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const [canMove, setCanMove] = useState(true);

  const onMove = useEffectEvent(e => {
    if (canMove) {
      setPosition({ x: e.clientX, y: e.clientY });
    }
  });

  useEffect(() => {
    window.addEventListener('pointermove', onMove);
    return () => window.removeEventListener('pointermove', onMove);
  }, []);

  return (
    <>
      <label>
        <input type="checkbox"
          checked={canMove}
          onChange={e => setCanMove(e.target.checked)}
        />
        The dot is allowed to move
      </label>
      <hr />
      <div style={{
        position: 'absolute',
        backgroundColor: 'pink',
        borderRadius: '50%',
        opacity: 0.6,
        transform: `translate(${position.x}px, ${position.y}px)`,
        pointerEvents: 'none',
```

```
        left: -20,
        top: -20,
        width: 40,
        height: 40,
      }} />
    </>
  );
}
```

Show moreThis doesn’t mean that useEffectEvent is always the correct solution. You should only apply it to the lines of code that you don’t want to be reactive. In the above sandbox, you didn’t want the Effect’s code to be reactive with regards to canMove. That’s why it made sense to extract an Effect Event.Read Removing Effect Dependencies for other correct alternatives to suppressing the linter.

Limitations of Effect Events

Under ConstructionThis section describes an experimental API that has not yet been released in a stable version of React.

Effect Events are very limited in how you can use them:

Only call them from inside Effects.
Never pass them to other components or Hooks.

For example, don’t declare and pass an Effect Event like this:

```
function Timer() {  const [count, setCount] = useState(0);  const onTick =
useEffectEvent(() => {    setCount(count + 1);  });  useTimer(onTick, 1000); // Ø=Ý4 Avoid:
Passing Effect Events  return <h1>{count}</h1>}function useTimer(callback, delay)
{  useEffect(() => {    const id = setInterval(() => {      callback();    }, delay);    return () =>
{      clearInterval(id);    };  }, [delay, callback]); // Need to specify "callback" in
dependencies}
```

Instead, always declare Effect Events directly next to the Effects that use them:

```
function Timer() {  const [count, setCount] = useState(0);  useTimer(() =>
{    setCount(count + 1);  }, 1000);  return <h1>{count}</h1>}function useTimer(callback,
delay) {  const onTick = useEffectEvent(() => {    callback();  });  useEffect(() => {    const
id = setInterval(() => {      onTick(); // '  Good: Only called locally inside an Effect    },
delay);    return () => {      clearInterval(id);    };  }, [delay]); // No need to specify
"onTick" (an Effect Event) as a dependency}
```

Effect Events are non-reactive “pieces” of your Effect code. They should be next to the Effect using them.

Recap

Event handlers run in response to specific interactions.
Effects run whenever synchronization is needed.
Logic inside event handlers is not reactive.
Logic inside Effects is reactive.
You can move non-reactive logic from Effects into Effect Events.
Only call Effect Events from inside Effects.
Don’t pass Effect Events to other components or Hooks.

Try out some challenges1. Fix a variable that doesn’t update 2. Fix a freezing counter 3. Fix a non-adjustable delay 4. Fix a delayed notification Challenge 1 of 4: Fix a variable that doesn’t update This Timer component keeps a count state variable which increases every second. The value by which it’s increasing is stored in the increment state variable. You can control the increment variable with the plus and minus buttons.However, no matter how many times you click the plus button, the counter is still incremented by one every second. What’s wrong with this code? Why is increment always equal to 1 inside the Effect’s code? Find the mistake and fix it.App.jsApp.js Download ResetForkimport { useState, useEffect } from 'react';

```
export default function Timer() {
  const [count, setCount] = useState(0);
  const [increment, setIncrement] = useState(1);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + increment);
    }, 1000);
    return () => {
      clearInterval(id);
    };
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []);

  return (
    <>
      <h1>
        Counter: {count}
        <button onClick={() => setCount(0)}>Reset</button>
      </h1>
      <hr />
      <p>
        Every second, increment by:
        <button disabled={increment === 0} onClick={() => {
          setIncrement(i => i - 1);
        }}>–</button>
        <b>{increment}</b>
        <button onClick={() => {
          setIncrement(i => i + 1);
        }}>+</button>
      </p>
    </>
  );
}
```

Show more Show hint Show solutionNext ChallengePreviousLifecycle of Reactive EffectsNextRemoving Effect Dependencies
Learn ReactEscape HatchesRemoving Effect DependenciesWhen you write an Effect, the linter will verify that you've included every reactive value (like props and state) that the Effect reads in the list of your Effect's dependencies. This ensures that your Effect remains synchronized with the latest props and state of your component. Unnecessary dependencies may cause your Effect to run too often, or even create an infinite loop. Follow this guide to review and remove unnecessary dependencies from your Effects.
You will learn
How to fix infinite Effect dependency loops
What to do when you want to remove a dependency
How to read a value from your Effect without "reacting" to it
How and why to avoid object and function dependencies
Why suppressing the dependency linter is dangerous, and what to do instead

Dependencies should match the code
When you write an Effect, you first specify how to start and stop whatever you want your Effect to be doing:
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) { useEffect(() => {   const connection = createConnection(serverUrl, roomId);
connection.connect();   return () => connection.disconnect(); 'òò ââçÐ
Then, if you leave the Effect dependencies empty ([]), the linter will suggest the correct dependencies:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // <-- Fix the mistake here!
  return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}

```
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
}
```

Show more

Fill them in according to what the linter says:

function ChatRoom({ roomId }) { useEffect(() => {   const connection = createConnection(serverUrl, roomId);   connection.connect();   return () => connection.disconnect(); }, [roomId]); // ' All dependencies declared // ...}

Effects “react” to reactive values. Since roomId is a reactive value (it can change due to a re-render), the linter verifies that you’ve specified it as a dependency. If roomId receives a different value, React will re-synchronize your Effect. This ensures that the chat stays connected to the selected room and “reacts” to the dropdown:

App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);
  return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
```

```
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
}
```

Show more

To remove a dependency, prove that it’s not a dependency

Notice that you can’t “choose” the dependencies of your Effect. Every reactive value used by your Effect’s code must be declared in your dependency list. The dependency list is determined by the surrounding code:

```
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) { // This is a reactive value  useEffect(() => {    const connection = createConnection(serverUrl, roomId); // This Effect reads that reactive value    connection.connect();    return () => connection.disconnect();  }, [roomId]); // '  So you must specify that reactive value as a dependency of your Effect  // ...}
```

Reactive values include props and all variables and functions declared directly inside of your component. Since roomId is a reactive value, you can’t remove it from the dependency list. The linter wouldn’t allow it:

```
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) {  useEffect(() => {    const connection = createConnection(serverUrl, roomId); connection.connect();    return () => connection.disconnect();  }, []); // Ø=Ý4 React Hook useEffect has a missing dependency: 'roomId'  // ...}
```

And the linter would be right! Since roomId may change over time, this would introduce a bug in your code.

To remove a dependency, “prove” to the linter that it doesn’t need to be a dependency. For example, you can move roomId out of your component to prove that it’s not reactive and won’t change on re-renders:

```
const serverUrl = 'https://localhost:1234';const roomId = 'music'; // Not a reactive value anymorefunction ChatRoom() {  useEffect(() => {    const connection = createConnection(serverUrl, roomId);    connection.connect();    return () => connection.disconnect();  }, []); // '  All dependencies declared  // ...}
```

Now that roomId is not a reactive value (and can’t change on a re-render), it doesn’t need to be a dependency:

App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';
const roomId = 'music';

export default function ChatRoom() {
  useEffect(() => {
```

```
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []);
  return <h1>Welcome to the {roomId} room!</h1>;
}
```

This is why you could now specify an empty ([]) dependency list. Your Effect really doesn't depend on any reactive value anymore, so it really doesn't need to re-run when any of the component's props or state change.
To change the dependencies, change the code
You might have noticed a pattern in your workflow:

First, you change the code of your Effect or how your reactive values are declared.
Then, you follow the linter and adjust the dependencies to match the code you have changed.
If you're not happy with the list of dependencies, you go back to the first step (and change the code again).

The last part is important. If you want to change the dependencies, change the surrounding code first. You can think of the dependency list as a list of all the reactive values used by your Effect's code. You don't choose what to put on that list. The list describes your code. To change the dependency list, change the code.
This might feel like solving an equation. You might start with a goal (for example, to remove a dependency), and you need to "find" the code matching that goal. Not everyone finds solving equations fun, and the same thing could be said about writing Effects! Luckily, there is a list of common recipes that you can try below.
PitfallIf you have an existing codebase, you might have some Effects that suppress the linter like this:useEffect(() => { // ... // Ø=Ý4 Avoid suppressing the linter like this: // eslint-ignore-next-line react-hooks/exhaustive-deps}, []);When dependencies don't match the code, there is a very high risk of introducing bugs. By suppressing the linter, you "lie" to React about the values your Effect depends on.Instead, use the techniques below.
Deep DiveWhy is suppressing the dependency linter so dangerous? Show DetailsSuppressing the linter leads to very unintuitive bugs that are hard to find and fix. Here's one example:App.jsApp.js Download ResetForkimport { useState, useEffect } from 'react';

```
export default function Timer() {
  const [count, setCount] = useState(0);
  const [increment, setIncrement] = useState(1);

  function onTick() {
—6WD6÷VçB†6÷VçB ² –æ7&VÖVçB“°
  }
```

```
  useEffect(() => {
    const id = setInterval(onTick, 1000);
    return () => clearInterval(id);
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, []);

  return (
    <>
      <h1>
        Counter: {count}
        <button onClick={() => setCount(0)}>Reset</button>
      </h1>
      <hr />
      <p>
        Every second, increment by:
        <button disabled={increment === 0} onClick={() => {
          setIncrement(i => i - 1);
        }}>–</button>
        <b>{increment}</b>
        <button onClick={() => {
          setIncrement(i => i + 1);
        }}>+</button>
      </p>
    </>
  );
}
```

Show moreLet's say that you wanted to run the Effect "only on mount". You've read that empty ([]) dependencies do that, so you've decided to ignore the linter, and forcefully specified [] as the dependencies.This counter was supposed to increment every second by the amount configurable with the two buttons. However, since you "lied" to React that this Effect doesn't depend on anything, React forever keeps using the onTick function from the initial render. During that render, count was 0 and increment was 1. This is why onTick from that render always calls setCount(0 + 1) every second, and you always see 1. Bugs like this are harder to fix when they're spread across multiple components.There's always a better solution than ignoring the linter! To fix this code, you need to add onTick to the dependency list. (To ensure the interval is only setup once, make onTick an Effect Event.)We recommend treating the dependency lint error as a compilation error. If you don't suppress it, you will never see bugs like this. The rest of this page documents the alternatives for this and other cases.

Removing unnecessary dependencies

Every time you adjust the Effect's dependencies to reflect the code, look at the dependency list. Does it make sense for the Effect to re-run when any of these dependencies change? Sometimes, the answer is "no":

You might want to re-execute different parts of your Effect under different conditions.

You might want to only read the latest value of some dependency instead of “reacting” to its changes.
A dependency may change too often unintentionally because it’s an object or a function.

To find the right solution, you’ll need to answer a few questions about your Effect. Let’s walk through them.
Should this code move to an event handler?
The first thing you should think about is whether this code should be an Effect at all.
Imagine a form. On submit, you set the submitted state variable to true. You need to send a POST request and show a notification. You’ve put this logic inside an Effect that “reacts” to submitted being true:
function Form() { const [submitted, setSubmitted] = useState(false); useEffect(() => { if (submitted) { // Ø=Ý4 Avoid: Event-specific logic inside an Effect post('/api/register'); showNotification('Successfully registered!'); } }, [submitted]); function handleSubmit() { setSubmitted(true); } // ...}
Later, you want to style the notification message according to the current theme, so you read the current theme. Since theme is declared in the component body, it is a reactive value, so you add it as a dependency:
function Form() { const [submitted, setSubmitted] = useState(false); const theme = useContext(ThemeContext); useEffect(() => { if (submitted) { // Ø=Ý4 Avoid: Event-specific logic inside an Effect post('/api/register'); showNotification('Successfully registered!', theme); } }, [submitted, theme]); // ' All dependencies declared function handleSubmit() { setSubmitted(true); } // ...}
By doing this, you’ve introduced a bug. Imagine you submit the form first and then switch between Dark and Light themes. The theme will change, the Effect will re-run, and so it will display the same notification again!
The problem here is that this shouldn’t be an Effect in the first place. You want to send this POST request and show the notification in response to submitting the form, which is a particular interaction. To run some code in response to particular interaction, put that logic directly into the corresponding event handler:
function Form() { const theme = useContext(ThemeContext); function handleSubmit() { // ' Good: Event-specific logic is called from event handlers post('/api/register'); showNotification('Successfully registered!', theme); } // ...}
Now that the code is in an event handler, it’s not reactive—so it will only run when the user submits the form. Read more about choosing between event handlers and Effects and how to delete unnecessary Effects.
Is your Effect doing several unrelated things?
The next question you should ask yourself is whether your Effect is doing several unrelated things.
Imagine you’re creating a shipping form where the user needs to choose their city and area. You fetch the list of cities from the server according to the selected country to show them in a dropdown:
function ShippingForm({ country }) { const [cities, setCities] = useState(null); const [city, setCity] = useState(null); useEffect(() => { let ignore = false; fetch(`/api/cities?country=${country}`) .then(response => response.json()) .then(json => { if (!ignore) { setCities(json); } }); return () => { ignore = true; }; },

[country]); // ' All dependencies declared // ...
This is a good example of fetching data in an Effect. You are synchronizing the cities state with the network according to the country prop. You can't do this in an event handler because you need to fetch as soon as ShippingForm is displayed and whenever the country changes (no matter which interaction causes it).
Now let's say you're adding a second select box for city areas, which should fetch the areas for the currently selected city. You might start by adding a second fetch call for the list of areas inside the same Effect:
function ShippingForm({ country }) { const [cities, setCities] = useState(null); const [city, setCity] = useState(null); const [areas, setAreas] = useState(null); useEffect(() => { let ignore = false; fetch(`/api/cities?country=${country}`) .then(response => response.json()) .then(json => { if (!ignore) { setCities(json); } }); // Ø=Ý4 Avoid: A single Effect synchronizes two independent processes if (city) { fetch(`/api/areas?city=${city}`) .then(response => response.json()) .then(json => { if (!ignore) { setAreas(json); } }); } return () => { ignore = true; }; }, [country, city]); // ' All dependencies declared // ...
However, since the Effect now uses the city state variable, you've had to add city to the list of dependencies. That, in turn, introduced a problem: when the user selects a different city, the Effect will re-run and call fetchCities(country). As a result, you will be unnecessarily refetching the list of cities many times.
The problem with this code is that you're synchronizing two different unrelated things:

You want to synchronize the cities state to the network based on the country prop.
You want to synchronize the areas state to the network based on the city state.

Split the logic into two Effects, each of which reacts to the prop that it needs to synchronize with:
function ShippingForm({ country }) { const [cities, setCities] = useState(null); useEffect(() => { let ignore = false; fetch(`/api/cities?country=${country}`) .then(response => response.json()) .then(json => { if (!ignore) { setCities(json); } }); return () => { ignore = true; }; }, [country]); // ' All dependencies declared const [city, setCity] = useState(null); const [areas, setAreas] = useState(null); useEffect(() => { if (city) { let ignore = false; fetch(`/api/areas?city=${city}`) .then(response => response.json()) .then(json => { if (!ignore) { setAreas(json); } }); return () => { ignore = true; }; } }, [city]); // ' All dependencies declared // ...
Now the first Effect only re-runs if the country changes, while the second Effect re-runs when the city changes. You've separated them by purpose: two different things are synchronized by two separate Effects. Two separate Effects have two separate dependency lists, so they won't trigger each other unintentionally.
The final code is longer than the original, but splitting these Effects is still correct. Each Effect should represent an independent synchronization process. In this example, deleting one Effect doesn't break the other Effect's logic. This means they synchronize different things, and it's good to split them up. If you're concerned about duplication, you can improve this code by extracting repetitive logic into a custom Hook.

Are you reading some state to calculate the next state?
This Effect updates the messages state variable with a newly created array every time a new message arrives:
function ChatRoom({ roomId }) { const [messages, setMessages] = useState([]); useEffect(() => {   const connection = createConnection();   connection.connect(); connection.on('message', (receivedMessage) => {   setMessages([...messages, receivedMessage]);   });   // ...
It uses the messages variable to create a new array starting with all the existing messages and adds the new message at the end. However, since messages is a reactive value read by an Effect, it must be a dependency:
function ChatRoom({ roomId }) { const [messages, setMessages] = useState([]); useEffect(() => {   const connection = createConnection();   connection.connect(); connection.on('message', (receivedMessage) => {   setMessages([...messages, receivedMessage]);   });   return () => connection.disconnect(); }, [roomId, messages]); // ' All dependencies declared // ...
And making messages a dependency introduces a problem.
Every time you receive a message, setMessages() causes the component to re-render with a new messages array that includes the received message. However, since this Effect now depends on messages, this will also re-synchronize the Effect. So every new message will make the chat re-connect. The user would not like that!
To fix the issue, don’t read messages inside the Effect. Instead, pass an updater function to setMessages:
function ChatRoom({ roomId }) { const [messages, setMessages] = useState([]); useEffect(() => {   const connection = createConnection();   connection.connect(); connection.on('message', (receivedMessage) => {   setMessages(msgs => [...msgs, receivedMessage]);   });   return () => connection.disconnect(); }, [roomId]); // ' All dependencies declared // ...
Notice how your Effect does not read the messages variable at all now. You only need to pass an updater function like msgs => [...msgs, receivedMessage]. React puts your updater function in a queue and will provide the msgs argument to it during the next render. This is why the Effect itself doesn’t need to depend on messages anymore. As a result of this fix, receiving a chat message will no longer make the chat re-connect.
Do you want to read a value without “reacting” to its changes?
Under ConstructionThis section describes an experimental API that has not yet been released in a stable version of React.
Suppose that you want to play a sound when the user receives a new message unless isMuted is true:
function ChatRoom({ roomId }) { const [messages, setMessages] = useState([]); const [isMuted, setIsMuted] = useState(false); useEffect(() => {   const connection = createConnection();   connection.connect();   connection.on('message', (receivedMessage) => {   setMessages(msgs => [...msgs, receivedMessage]);   if (!isMuted) {   playSound();   }   });   // ...
Since your Effect now uses isMuted in its code, you have to add it to the dependencies:
function ChatRoom({ roomId }) { const [messages, setMessages] = useState([]); const [isMuted, setIsMuted] = useState(false); useEffect(() => {   const connection = createConnection();   connection.connect();   connection.on('message',

(receivedMessage) => { setMessages(msgs => [...msgs, receivedMessage]); if (!isMuted) { playSound(); } }); return () => connection.disconnect(); }, [roomId, isMuted]); // ' All dependencies declared // ...

The problem is that every time isMuted changes (for example, when the user presses the “Muted” toggle), the Effect will re-synchronize, and reconnect to the chat. This is not the desired user experience! (In this example, even disabling the linter would not work—if you do that, isMuted would get “stuck” with its old value.)

To solve this problem, you need to extract the logic that shouldn’t be reactive out of the Effect. You don’t want this Effect to “react” to the changes in isMuted. Move this non-reactive piece of logic into an Effect Event:

import { useState, useEffect, useEffectEvent } from 'react';function ChatRoom({ roomId }) { const [messages, setMessages] = useState([]); const [isMuted, setIsMuted] = useState(false); const onMessage = useEffectEvent(receivedMessage => { setMessages(msgs => [...msgs, receivedMessage]); if (!isMuted) { playSound(); } }); useEffect(() => { const connection = createConnection(); connection.connect(); connection.on('message', (receivedMessage) => { onMessage(receivedMessage); }); return () => connection.disconnect(); }, [roomId]); // ' All dependencies declared // ...

Effect Events let you split an Effect into reactive parts (which should “react” to reactive values like roomId and their changes) and non-reactive parts (which only read their latest values, like onMessage reads isMuted). Now that you read isMuted inside an Effect Event, it doesn’t need to be a dependency of your Effect. As a result, the chat won’t re-connect when you toggle the “Muted” setting on and off, solving the original issue!

Wrapping an event handler from the props

You might run into a similar problem when your component receives an event handler as a prop:

function ChatRoom({ roomId, onReceiveMessage }) { const [messages, setMessages] = useState([]); useEffect(() => { const connection = createConnection(); connection.connect(); connection.on('message', (receivedMessage) => { onReceiveMessage(receivedMessage); }); return () => connection.disconnect(); }, [roomId, onReceiveMessage]); // ' All dependencies declared // ...

Suppose that the parent component passes a different onReceiveMessage function on every render:

<ChatRoom roomId={roomId} onReceiveMessage={receivedMessage => { // ... }}/>

Since onReceiveMessage is a dependency, it would cause the Effect to re-synchronize after every parent re-render. This would make it re-connect to the chat. To solve this, wrap the call in an Effect Event:

function ChatRoom({ roomId, onReceiveMessage }) { const [messages, setMessages] = useState([]); const onMessage = useEffectEvent(receivedMessage => { onReceiveMessage(receivedMessage); }); useEffect(() => { const connection = createConnection(); connection.connect(); connection.on('message', (receivedMessage) => { onMessage(receivedMessage); }); return () => connection.disconnect(); }, [roomId]); // ' All dependencies declared // ...

Effect Events aren’t reactive, so you don’t need to specify them as dependencies. As a

result, the chat will no longer re-connect even if the parent component passes a function that’s different on every re-render.

Separating reactive and non-reactive code

In this example, you want to log a visit every time roomId changes. You want to include the current notificationCount with every log, but you don’t want a change to notificationCount to trigger a log event.

The solution is again to split out the non-reactive code into an Effect Event:

function Chat({ roomId, notificationCount }) { const onVisit = useEffectEvent(visitedRoomId => { logVisit(visitedRoomId, notificationCount); }); useEffect(() => { onVisit(roomId); }, [roomId]); // ' All dependencies declared // ...}

You want your logic to be reactive with regards to roomId, so you read roomId inside of your Effect. However, you don’t want a change to notificationCount to log an extra visit, so you read notificationCount inside of the Effect Event. Learn more about reading the latest props and state from Effects using Effect Events.

Does some reactive value change unintentionally?

Sometimes, you do want your Effect to “react” to a certain value, but that value changes more often than you’d like—and might not reflect any actual change from the user’s perspective. For example, let’s say that you create an options object in the body of your component, and then read that object from inside of your Effect:

function ChatRoom({ roomId }) { // ... const options = { serverUrl: serverUrl, roomId: roomId }; useEffect(() => { const connection = createConnection(options); connection.connect(); // ...

This object is declared in the component body, so it’s a reactive value. When you read a reactive value like this inside an Effect, you declare it as a dependency. This ensures your Effect “reacts” to its changes:

// ... useEffect(() => { const connection = createConnection(options); connection.connect(); return () => connection.disconnect(); }, [options]); // ' All dependencies declared // ...

It is important to declare it as a dependency! This ensures, for example, that if the roomId changes, your Effect will re-connect to the chat with the new options. However, there is also a problem with the code above. To see it, try typing into the input in the sandbox below, and watch what happens in the console:

App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  // Temporarily disable the linter to demonstrate the problem
  // eslint-disable-next-line react-hooks/exhaustive-deps
  const options = {
    serverUrl: serverUrl,
    roomId: roomId
  };
```

```
  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [options]);

  return (
    <>
      <h1>Welcome to the {roomId} room!</h1>
      <input value={message} onChange={e => setMessage(e.target.value)} />
    </>
  );
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
}
```

Show more

In the sandbox above, the input only updates the message state variable. From the user's perspective, this should not affect the chat connection. However, every time you update the message, your component re-renders. When your component re-renders, the code inside of it runs again from scratch.

A new options object is created from scratch on every re-render of the ChatRoom component. React sees that the options object is a different object from the options object created during the last render. This is why it re-synchronizes your Effect (which depends on options), and the chat re-connects as you type.

This problem only affects objects and functions. In JavaScript, each newly created

object and function is considered distinct from all the others. It doesn’t matter that the contents inside of them may be the same!
// During the first renderconst options1 = { serverUrl: 'https://localhost:1234', roomId: 'music' };// During the next renderconst options2 = { serverUrl: 'https://localhost:1234', roomId: 'music' };// These are two different objects!console.log(Object.is(options1, options2)); // false
Object and function dependencies can make your Effect re-synchronize more often than you need.
This is why, whenever possible, you should try to avoid objects and functions as your Effect’s dependencies. Instead, try moving them outside the component, inside the Effect, or extracting primitive values out of them.
Move static objects and functions outside your component
If the object does not depend on any props and state, you can move that object outside your component:
const options = { serverUrl: 'https://localhost:1234', roomId: 'music'};function ChatRoom() { const [message, setMessage] = useState(''); useEffect(() => {   const connection = createConnection(options);   connection.connect();   return () => connection.disconnect(); }, []); // ' All dependencies declared // ...
This way, you prove to the linter that it’s not reactive. It can’t change as a result of a re-render, so it doesn’t need to be a dependency. Now re-rendering ChatRoom won’t cause your Effect to re-synchronize.
This works for functions too:
function createOptions() { return {   serverUrl: 'https://localhost:1234',   roomId: 'music' };}function ChatRoom() { const [message, setMessage] = useState(''); useEffect(() => {   const options = createOptions();   const connection = createConnection();   connection.connect();   return () => connection.disconnect(); }, []); // ' All dependencies declared // ...
Since createOptions is declared outside your component, it’s not a reactive value. This is why it doesn’t need to be specified in your Effect’s dependencies, and why it won’t ever cause your Effect to re-synchronize.
Move dynamic objects and functions inside your Effect
If your object depends on some reactive value that may change as a result of a re-render, like a roomId prop, you can’t pull it outside your component. You can, however, move its creation inside of your Effect’s code:
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) { const [message, setMessage] = useState(''); useEffect(() => {   const options = {   serverUrl: serverUrl,   roomId: roomId   };   const connection = createConnection(options);   connection.connect();   return () => connection.disconnect(); }, [roomId]); // ' All dependencies declared // ...
Now that options is declared inside of your Effect, it is no longer a dependency of your Effect. Instead, the only reactive value used by your Effect is roomId. Since roomId is not an object or function, you can be sure that it won’t be unintentionally different. In JavaScript, numbers and strings are compared by their content:
// During the first renderconst roomId1 = 'music';// During the next renderconst roomId2 = 'music';// These two strings are the same!console.log(Object.is(roomId1, roomId2)); // true

Thanks to this fix, the chat no longer re-connects if you edit the input:
App.jschat.jsApp.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
      roomId: roomId
    };
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);

  return (
    <>
      <h1>Welcome to the {roomId} room!</h1>
      <input value={message} onChange={e => setMessage(e.target.value)} />
    </>
  );
}

export default function App() {
  const [roomId, setRoomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => setRoomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom roomId={roomId} />
    </>
  );
```

}

Show more
However, it does re-connect when you change the roomId dropdown, as you would expect.
This works for functions, too:
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) { const [message, setMessage] = useState(''); useEffect(() => { function createOptions() { return { serverUrl: serverUrl, roomId: roomId }; } const options = createOptions(); const connection = createConnection(options); connection.connect(); return () => connection.disconnect(); }, [roomId]); // ' All dependencies declared // ...
You can write your own functions to group pieces of logic inside your Effect. As long as you also declare them inside your Effect, they're not reactive values, and so they don't need to be dependencies of your Effect.
Read primitive values from objects
Sometimes, you may receive an object from props:
function ChatRoom({ options }) { const [message, setMessage] = useState(''); useEffect(() => { const connection = createConnection(options); connection.connect(); return () => connection.disconnect(); }, [options]); // ' All dependencies declared // ...
The risk here is that the parent component will create the object during rendering:
<ChatRoom roomId={roomId} options={{ serverUrl: serverUrl, roomId: roomId }}/>
This would cause your Effect to re-connect every time the parent component re-renders. To fix this, read information from the object outside the Effect, and avoid having object and function dependencies:
function ChatRoom({ options }) { const [message, setMessage] = useState(''); const { roomId, serverUrl } = options; useEffect(() => { const connection = createConnection({ roomId: roomId, serverUrl: serverUrl }); connection.connect(); return () => connection.disconnect(); }, [roomId, serverUrl]); // ' All dependencies declared // ...
The logic gets a little repetitive (you read some values from an object outside an Effect, and then create an object with the same values inside the Effect). But it makes it very explicit what information your Effect actually depends on. If an object is re-created unintentionally by the parent component, the chat would not re-connect. However, if options.roomId or options.serverUrl really are different, the chat would re-connect.
Calculate primitive values from functions
The same approach can work for functions. For example, suppose the parent component passes a function:
<ChatRoom roomId={roomId} getOptions={() => { return { serverUrl: serverUrl, roomId: roomId }; }}/>
To avoid making it a dependency (and causing it to re-connect on re-renders), call it outside the Effect. This gives you the roomId and serverUrl values that aren't objects, and that you can read from inside your Effect:
function ChatRoom({ getOptions }) { const [message, setMessage] = useState(''); const { roomId, serverUrl } = getOptions(); useEffect(() => { const connection =

createConnection({ roomId: roomId, serverUrl: serverUrl }); connection.connect(); return () => connection.disconnect(); }, [roomId, serverUrl]); // ' All dependencies declared // ...

This only works for pure functions because they are safe to call during rendering. If your function is an event handler, but you don’t want its changes to re-synchronize your Effect, wrap it into an Effect Event instead.

Recap

Dependencies should always match the code.

When you’re not happy with your dependencies, what you need to edit is the code.

Suppressing the linter leads to very confusing bugs, and you should always avoid it.

To remove a dependency, you need to “prove” to the linter that it’s not necessary.

If some code should run in response to a specific interaction, move that code to an event handler.

If different parts of your Effect should re-run for different reasons, split it into several Effects.

If you want to update some state based on the previous state, pass an updater function.

If you want to read the latest value without “reacting” it, extract an Effect Event from your Effect.

In JavaScript, objects and functions are considered different if they were created at different times.

Try to avoid object and function dependencies. Move them outside the component or inside the Effect.

Try out some challenges1. Fix a resetting interval 2. Fix a retriggering animation 3. Fix a reconnecting chat 4. Fix a reconnecting chat, again Challenge 1 of 4: Fix a resetting interval This Effect sets up an interval that ticks every second. You’ve noticed something strange happening: it seems like the interval gets destroyed and re-created every time it ticks. Fix the code so that the interval doesn’t get constantly re-created.App.jsApp.js Download ResetForkimport { useState, useEffect } from 'react';

```
export default function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('' Creating an interval');
    const id = setInterval(() => {
      console.log('#ð Interval tick');
      setCount(count + 1);
    }, 1000);
    return () => {
      console.log('L Clearing an interval');
      clearInterval(id);
    };
  }, [count]);

  return <h1>Counter: {count}</h1>
```

}

Show more Show hint Show solutionNext ChallengePreviousSeparating Events from EffectsNextReusing Logic with Custom Hooks
Learn ReactEscape HatchesReusing Logic with Custom HooksReact comes with several built-in Hooks like useState, useContext, and useEffect. Sometimes, you'll wish that there was a Hook for some more specific purpose: for example, to fetch data, to keep track of whether the user is online, or to connect to a chat room. You might not find these Hooks in React, but you can create your own Hooks for your application's needs.
You will learn
What custom Hooks are, and how to write your own
How to reuse logic between components
How to name and structure your custom Hooks
When and why to extract custom Hooks

Custom Hooks: Sharing logic between components
Imagine you're developing an app that heavily relies on the network (as most apps do). You want to warn the user if their network connection has accidentally gone off while they were using your app. How would you go about it? It seems like you'll need two things in your component:

A piece of state that tracks whether the network is online.
An Effect that subscribes to the global online and offline events, and updates that state.

This will keep your component synchronized with the network status. You might start with something like this:
App.jsApp.js Download ResetFork991234567891011121314151617181920212 2import { useState, useEffect } from 'react';export default function StatusBar() { const [isOnline, setIsOnline] = useState(true); useEffect(() => { function handleOnline() { setIsOnline(true); } function handleOffline() { setIsOnline(false); } window.addEventListener('online', handleOnline); window.addEventListener('offline', handleOffline); return () => { window.removeEventListener('online', handleOnline); window.removeEventListener('offline', handleOffline); }; }, []); return <h1>{isOnline ? '' Online' : 'L Disconnected'}</h1>;}Show more
Try turning your network on and off, and notice how this StatusBar updates in response to your actions.
Now imagine you also want to use the same logic in a different component. You want to implement a Save button that will become disabled and show "Reconnecting…" instead of "Save" while the network is off.
To start, you can copy and paste the isOnline state and the Effect into SaveButton:
App.jsApp.js Download ResetForkimport { useState, useEffect } from 'react';

```
export default function SaveButton() {
  const [isOnline, setIsOnline] = useState(true);
  useEffect(() => {
```

```
    function handleOnline() {
      setIsOnline(true);
    }
    function handleOffline() {
      setIsOnline(false);
    }
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);
    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);

  function handleSaveClick() {
    console.log('  Progress saved');
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? 'Save progress' : 'Reconnecting...'}
    </button>
  );
}
```

Show more

Verify that, if you turn off the network, the button will change its appearance.

These two components work fine, but the duplication in logic between them is unfortunate. It seems like even though they have different visual appearance, you want to reuse the logic between them.

Extracting your own custom Hook from a component

Imagine for a moment that, similar to useState and useEffect, there was a built-in useOnlineStatus Hook. Then both of these components could be simplified and you could remove the duplication between them:

function StatusBar() { const isOnline = useOnlineStatus(); return <h1>{isOnline ? '' Online' : ''L Disconnected'}</h1>;}function SaveButton() { const isOnline = useOnlineStatus(); function handleSaveClick() {   console.log(' Progress saved'); } return (<button disabled={!isOnline} onClick={handleSaveClick}>   {isOnline ? 'Save progress' : 'Reconnecting...'}   </button>);}

Although there is no such built-in Hook, you can write it yourself. Declare a function called useOnlineStatus and move all the duplicated code into it from the components you wrote earlier:

function useOnlineStatus() { const [isOnline, setIsOnline] = useState(true); useEffect(() => {   function handleOnline() {   setIsOnline(true);   }   function handleOffline() {   setIsOnline(false);   }   window.addEventListener('online', handleOnline);   window.addEventListener('offline', handleOffline);   return () =>

```
{      window.removeEventListener('online', handleOnline);
window.removeEventListener('offline', handleOffline);    };  }, []);  return isOnline;}
```
At the end of the function, return isOnline. This lets your components read that value:
App.jsuseOnlineStatus.jsApp.js ResetForkimport { useOnlineStatus } from './
useOnlineStatus.js';

```
function StatusBar() {
  const isOnline = useOnlineStatus();
  return <h1>{isOnline ? ''  Online' : ''L Disconnected'}</h1>;
}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log(''  Progress saved');
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? 'Save progress' : 'Reconnecting...'}
    </button>
  );
}

export default function App() {
  return (
    <>
      <SaveButton />
      <StatusBar />
    </>
  );
}
```

Show more
Verify that switching the network on and off updates both components.
Now your components don’t have as much repetitive logic. More importantly, the code inside them describes what they want to do (use the online status!) rather than how to do it (by subscribing to the browser events).
When you extract logic into custom Hooks, you can hide the gnarly details of how you deal with some external system or a browser API. The code of your components expresses your intent, not the implementation.
Hook names always start with use
React applications are built from components. Components are built from Hooks, whether built-in or custom. You’ll likely often use custom Hooks created by others, but occasionally you might write one yourself!

You must follow these naming conventions:

React component names must start with a capital letter, like StatusBar and SaveButton. React components also need to return something that React knows how to display, like a piece of JSX.
Hook names must start with use followed by a capital letter, like useState (built-in) or useOnlineStatus (custom, like earlier on the page). Hooks may return arbitrary values.

This convention guarantees that you can always look at a component and know where its state, Effects, and other React features might “hide”. For example, if you see a getColor() function call inside your component, you can be sure that it can’t possibly contain React state inside because its name doesn’t start with use. However, a function call like useOnlineStatus() will most likely contain calls to other Hooks inside!
NoteIf your linter is configured for React, it will enforce this naming convention. Scroll up to the sandbox above and rename useOnlineStatus to getOnlineStatus. Notice that the linter won’t allow you to call useState or useEffect inside of it anymore. Only Hooks and components can call other Hooks!
Deep DiveShould all functions called during rendering start with the use prefix? Show DetailsNo. Functions that don’t call Hooks don’t need to be Hooks.If your function doesn’t call any Hooks, avoid the use prefix. Instead, write it as a regular function without the use prefix. For example, useSorted below doesn’t call Hooks, so call it getSorted instead:// Ø=Ý4 Avoid: A Hook that doesn't use Hooksfunction useSorted(items) { return items.slice().sort();}// ' Good: A regular function that doesn't use Hooksfunction getSorted(items) { return items.slice().sort();}This ensures that your code can call this regular function anywhere, including conditions:function List({ items, shouldSort }) { let displayedItems = items; if (shouldSort) {   // ' It's ok to call getSorted() conditionally because it's not a Hook   displayedItems = getSorted(items); } // ...}You should give use prefix to a function (and thus make it a Hook) if it uses at least one Hook inside of it:// ' Good: A Hook that uses other Hooksfunction useAuth() { return useContext(Auth);}
Technically, this isn’t enforced by React. In principle, you could make a Hook that doesn’t call other Hooks. This is often confusing and limiting so it’s best to avoid that pattern. However, there may be rare cases where it is helpful. For example, maybe your function doesn’t use any Hooks right now, but you plan to add some Hook calls to it in the future. Then it makes sense to name it with the use prefix:// ' Good: A Hook that will likely use some other Hooks laterfunction useAuth() { // TODO: Replace with this line when authentication is implemented: // return useContext(Auth); return TEST_USER;}
Then components won’t be able to call it conditionally. This will become important when you actually add Hook calls inside. If you don’t plan to use Hooks inside it (now or later), don’t make it a Hook.
Custom Hooks let you share stateful logic, not state itself
In the earlier example, when you turned the network on and off, both components updated together. However, it’s wrong to think that a single isOnline state variable is shared between them. Look at this code:
function StatusBar() { const isOnline = useOnlineStatus(); // ...}function SaveButton() { const isOnline = useOnlineStatus(); // ...}
It works the same way as before you extracted the duplication:

function StatusBar() { const [isOnline, setIsOnline] = useState(true); useEffect(() => {   // ... }, []); // ...}function SaveButton() { const [isOnline, setIsOnline] = useState(true); useEffect(() => {   // ... }, []); // ...}
These are two completely independent state variables and Effects! They happened to have the same value at the same time because you synchronized them with the same external value (whether the network is on).
To better illustrate this, we'll need a different example. Consider this Form component:
App.jsApp.js Download ResetForkimport { useState } from 'react';

```
export default function Form() {
  const [firstName, setFirstName] = useState('Mary');
  const [lastName, setLastName] = useState('Poppins');

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
  }

  return (
    <>
      <label>
        First name:
        <input value={firstName} onChange={handleFirstNameChange} />
      </label>
      <label>
        Last name:
        <input value={lastName} onChange={handleLastNameChange} />
      </label>
      <p><b>Good morning, {firstName} {lastName}.</b></p>
    </>
  );
}
```

Show more
There's some repetitive logic for each form field:

There's a piece of state (firstName and lastName).
There's a change handler (handleFirstNameChange and handleLastNameChange).
There's a piece of JSX that specifies the value and onChange attributes for that input.

You can extract the repetitive logic into this useFormInput custom Hook:
App.jsuseFormInput.jsuseFormInput.js ResetForkimport { useState } from 'react';

```
export function useFormInput(initialValue) {
  const [value, setValue] = useState(initialValue);

  function handleChange(e) {
    setValue(e.target.value);
  }

  const inputProps = {
    value: value,
    onChange: handleChange
  };

  return inputProps;
}
```

Show more

Notice that it only declares one state variable called value.

However, the Form component calls useFormInput two times:

function Form() { const firstNameProps = useFormInput('Mary'); const lastNameProps = useFormInput('Poppins'); // ...

This is why it works like declaring two separate state variables!

Custom Hooks let you share stateful logic but not state itself. Each call to a Hook is completely independent from every other call to the same Hook. This is why the two sandboxes above are completely equivalent. If you'd like, scroll back up and compare them. The behavior before and after extracting a custom Hook is identical.

When you need to share the state itself between multiple components, lift it up and pass it down instead.

Passing reactive values between Hooks

The code inside your custom Hooks will re-run during every re-render of your component. This is why, like components, custom Hooks need to be pure. Think of custom Hooks' code as part of your component's body!

Because custom Hooks re-render together with your component, they always receive the latest props and state. To see what this means, consider this chat room example. Change the server URL or the chat room:

App.jsChatRoom.jschat.jsnotifications.jsChatRoom.js ResetForkimport { useState, useEffect } from 'react';
import { createConnection } from './chat.js';
import { showNotification } from './notifications.js';

```
export default function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
      roomId: roomId
```

```
    };
    const connection = createConnection(options);
    connection.on('message', (msg) => {
      showNotification('New message: ' + msg);
    });
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]);

  return (
    <>
      <label>
        Server URL:
        <input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />
      </label>
      <h1>Welcome to the {roomId} room!</h1>
    </>
  );
}
```

Show more

When you change serverUrl or roomId, the Effect “reacts” to your changes and re-synchronizes. You can tell by the console messages that the chat re-connects every time that you change your Effect’s dependencies.

Now move the Effect’s code into a custom Hook:

```
export function useChatRoom({ serverUrl, roomId }) {  useEffect(() => {    const options = {      serverUrl: serverUrl,      roomId: roomId    };    const connection = createConnection(options);    connection.connect();    connection.on('message', (msg) => {      showNotification('New message: ' + msg);    });    return () => connection.disconnect();  }, [roomId, serverUrl]);}
```

This lets your ChatRoom component call your custom Hook without worrying about how it works inside:

```
export default function ChatRoom({ roomId }) {  const [serverUrl, setServerUrl] = useState('https://localhost:1234');  useChatRoom({    roomId: roomId,    serverUrl: serverUrl  });  return (    <>      <label>        Server URL:        <input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />      </label>      <h1>Welcome to the {roomId} room!</h1>    </>  );}
```

This looks much simpler! (But it does the same thing.)

Notice that the logic still responds to prop and state changes. Try editing the server URL or the selected room:

App.jsChatRoom.jsuseChatRoom.jschat.jsnotifications.jsChatRoom.js ResetForkimport { useState } from 'react';
import { useChatRoom } from './useChatRoom.js';

```
export default function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');
```

```
  useChatRoom({
    roomId: roomId,
    serverUrl: serverUrl
  });

  return (
    <>
      <label>
        Server URL:
        <input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />
      </label>
      <h1>Welcome to the {roomId} room!</h1>
    </>
  );
}
```

Show more

Notice how you're taking the return value of one Hook:

```
export default function ChatRoom({ roomId }) {  const [serverUrl, setServerUrl] = useState('https://localhost:1234');  useChatRoom({    roomId: roomId,    serverUrl: serverUrl  });  // ...
```

and pass it as an input to another Hook:

```
export default function ChatRoom({ roomId }) {  const [serverUrl, setServerUrl] = useState('https://localhost:1234');  useChatRoom({    roomId: roomId,    serverUrl: serverUrl  });  // ...
```

Every time your ChatRoom component re-renders, it passes the latest roomId and serverUrl to your Hook. This is why your Effect re-connects to the chat whenever their values are different after a re-render. (If you ever worked with audio or video processing software, chaining Hooks like this might remind you of chaining visual or audio effects. It's as if the output of useState "feeds into" the input of the useChatRoom.)

Passing event handlers to custom Hooks

Under ConstructionThis section describes an experimental API that has not yet been released in a stable version of React.

As you start using useChatRoom in more components, you might want to let components customize its behavior. For example, currently, the logic for what to do when a message arrives is hardcoded inside the Hook:

```
export function useChatRoom({ serverUrl, roomId }) {  useEffect(() => {    const options = {      serverUrl: serverUrl,      roomId: roomId    };    const connection = createConnection(options);    connection.connect();    connection.on('message', (msg) => {      showNotification('New message: ' + msg);    });    return () => connection.disconnect();  }, [roomId, serverUrl]);}
```

Let's say you want to move this logic back to your component:

```
export default function ChatRoom({ roomId }) {  const [serverUrl, setServerUrl] = useState('https://localhost:1234');  useChatRoom({    roomId: roomId,    serverUrl: serverUrl,    onReceiveMessage(msg) {      showNotification('New message: ' + msg);    }  });  // ...
```

To make this work, change your custom Hook to take onReceiveMessage as one of its named options:
export function useChatRoom({ serverUrl, roomId, onReceiveMessage }) { useEffect(() => {   const options = {   serverUrl: serverUrl,   roomId: roomId   };   const connection = createConnection(options);   connection.connect(); connection.on('message', (msg) => {   onReceiveMessage(msg);   });   return () => connection.disconnect(); }, [roomId, serverUrl, onReceiveMessage]); // ' All dependencies declared}
This will work, but there's one more improvement you can do when your custom Hook accepts event handlers.
Adding a dependency on onReceiveMessage is not ideal because it will cause the chat to re-connect every time the component re-renders. Wrap this event handler into an Effect Event to remove it from the dependencies:
import { useEffect, useEffectEvent } from 'react';// ...export function useChatRoom({ serverUrl, roomId, onReceiveMessage }) { const onMessage = useEffectEvent(onReceiveMessage); useEffect(() => {   const options = {   serverUrl: serverUrl,   roomId: roomId   };   const connection = createConnection(options); connection.connect();   connection.on('message', (msg) => {   onMessage(msg);   });   return () => connection.disconnect(); }, [roomId, serverUrl]); // ' All dependencies declared}
Now the chat won't re-connect every time that the ChatRoom component re-renders. Here is a fully working demo of passing an event handler to a custom Hook that you can play with:
App.jsChatRoom.jsuseChatRoom.jschat.jsnotifications.jsChatRoom.js ResetForkimport { useState } from 'react';
import { useChatRoom } from './useChatRoom.js';
import { showNotification } from './notifications.js';

```
export default function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useChatRoom({
    roomId: roomId,
    serverUrl: serverUrl,
    onReceiveMessage(msg) {
      showNotification('New message: ' + msg);
    }
  });

  return (
    <>
      <label>
        Server URL:
        <input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />
      </label>
      <h1>Welcome to the {roomId} room!</h1>
```

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

Show more

Notice how you no longer need to know how useChatRoom works in order to use it. You could add it to any other component, pass any other options, and it would work the same way. That's the power of custom Hooks.

When to use custom Hooks

You don't need to extract a custom Hook for every little duplicated bit of code. Some duplication is fine. For example, extracting a useFormInput Hook to wrap a single useState call like earlier is probably unnecessary.

However, whenever you write an Effect, consider whether it would be clearer to also wrap it in a custom Hook. You shouldn't need Effects very often, so if you're writing one, it means that you need to "step outside React" to synchronize with some external system or to do something that React doesn't have a built-in API for. Wrapping it into a custom Hook lets you precisely communicate your intent and how the data flows through it.

For example, consider a ShippingForm component that displays two dropdowns: one shows the list of cities, and another shows the list of areas in the selected city. You might start with some code that looks like this:

```
function ShippingForm({ country }) {  const [cities, setCities] = useState(null);  // This Effect fetches cities for a country  useEffect(() => {    let ignore = false;    fetch(`/api/cities?country=${country}`)      .then(response => response.json())      .then(json => {        if (!ignore) {          setCities(json);        }      });    return () => {      ignore = true;    };  }, [country]);  const [city, setCity] = useState(null);  const [areas, setAreas] = useState(null);  // This Effect fetches areas for the selected city  useEffect(() => {    if (city) {      let ignore = false;      fetch(`/api/areas?city=${city}`)        .then(response => response.json())        .then(json => {          if (!ignore) {            setAreas(json);          }        });      return () => {        ignore = true;      };    }  }, [city]);  // ...
```

Although this code is quite repetitive, it's correct to keep these Effects separate from each other. They synchronize two different things, so you shouldn't merge them into one Effect. Instead, you can simplify the ShippingForm component above by extracting the common logic between them into your own useData Hook:

```
function useData(url) {  const [data, setData] = useState(null);  useEffect(() => {    if (url) {      let ignore = false;      fetch(url)        .then(response => response.json())        .then(json => {          if (!ignore) {            setData(json);          }        });      return () => {        ignore = true;      };    }  }, [url]);  return data;}
```

Now you can replace both Effects in the ShippingForm components with calls to useData:

```
function ShippingForm({ country }) {  const cities = useData(`/api/cities?country=${country}`);  const [city, setCity] = useState(null);  const areas = useData(city ? `/api/areas?city=${city}` : null);  // ...
```

Extracting a custom Hook makes the data flow explicit. You feed the url in and you get

the data out. By “hiding” your Effect inside useData, you also prevent someone working on the ShippingForm component from adding unnecessary dependencies to it. With time, most of your app’s Effects will be in custom Hooks.
Deep DiveKeep your custom Hooks focused on concrete high-level use cases Show DetailsStart by choosing your custom Hook’s name. If you struggle to pick a clear name, it might mean that your Effect is too coupled to the rest of your component’s logic, and is not yet ready to be extracted.Ideally, your custom Hook’s name should be clear enough that even a person who doesn’t write code often could have a good guess about what your custom Hook does, what it takes, and what it returns:
' useData(url)
' useImpressionLog(eventName, extraData)
' useChatRoom(options)
When you synchronize with an external system, your custom Hook name may be more technical and use jargon specific to that system. It’s good as long as it would be clear to a person familiar with that system:
' useMediaQuery(query)
' useSocket(url)
' useIntersectionObserver(ref, options)
Keep custom Hooks focused on concrete high-level use cases. Avoid creating and using custom “lifecycle” Hooks that act as alternatives and convenience wrappers for the useEffect API itself:
Ø=Ý4 useMount(fn)
Ø=Ý4 useEffectOnce(fn)
Ø=Ý4 useUpdateEffect(fn)
For example, this useMount Hook tries to ensure some code only runs “on mount”:function ChatRoom({ roomId }) { const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // Ø=Ý4 Avoid: using custom "lifecycle" Hooks useMount(() => { const connection = createConnection({ roomId, serverUrl }); connection.connect(); post('/analytics/event', { eventName: 'visit_chat' }); }); // ...}// Ø=Ý4 Avoid: creating custom "lifecycle" Hooksfunction useMount(fn) { useEffect(() => { fn(); }, []); // Ø=Ý4 React Hook useEffect has a missing dependency: 'fn'}Custom “lifecycle” Hooks like useMount don’t fit well into the React paradigm. For example, this code example has a mistake (it doesn’t “react” to roomId or serverUrl changes), but the linter won’t warn you about it because the linter only checks direct useEffect calls. It won’t know about your Hook.If you’re writing an Effect, start by using the React API directly:function ChatRoom({ roomId }) { const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // ' Good: two raw Effects separated by purpose useEffect(() => { const connection = createConnection({ serverUrl, roomId }); connection.connect(); return () => connection.disconnect(); }, [serverUrl, roomId]); useEffect(() => { post('/analytics/event', { eventName: 'visit_chat', roomId }); }, [roomId]); // ...}Then, you can (but don’t have to) extract custom Hooks for different high-level use cases:function ChatRoom({ roomId }) { const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // ' Great: custom Hooks named after their purpose useChatRoom({ serverUrl, roomId }); useImpressionLog('visit_chat', { roomId }); // ...}A good custom Hook makes the calling code more declarative by constraining what it does. For example, useChatRoom(options) can only connect to the chat room, while

useImpressionLog(eventName, extraData) can only send an impression log to the analytics. If your custom Hook API doesn’t constrain the use cases and is very abstract, in the long run it’s likely to introduce more problems than it solves.

Custom Hooks help you migrate to better patterns

Effects are an “escape hatch”: you use them when you need to “step outside React” and when there is no better built-in solution for your use case. With time, the React team’s goal is to reduce the number of the Effects in your app to the minimum by providing more specific solutions to more specific problems. Wrapping your Effects in custom Hooks makes it easier to upgrade your code when these solutions become available.

Let’s return to this example:

App.jsuseOnlineStatus.jsuseOnlineStatus.js ResetForkimport { useState, useEffect } from 'react';

```
export function useOnlineStatus() {
  const [isOnline, setIsOnline] = useState(true);
  useEffect(() => {
    function handleOnline() {
      setIsOnline(true);
    }
    function handleOffline() {
      setIsOnline(false);
    }
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);
    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);
  return isOnline;
}
```

Show more

In the above example, useOnlineStatus is implemented with a pair of useState and useEffect. However, this isn’t the best possible solution. There is a number of edge cases it doesn’t consider. For example, it assumes that when the component mounts, isOnline is already true, but this may be wrong if the network already went offline. You can use the browser navigator.onLine API to check for that, but using it directly would not work on the server for generating the initial HTML. In short, this code could be improved.

Luckily, React 18 includes a dedicated API called useSyncExternalStore which takes care of all of these problems for you. Here is how your useOnlineStatus Hook, rewritten to take advantage of this new API:

App.jsuseOnlineStatus.jsuseOnlineStatus.js ResetForkimport { useSyncExternalStore } from 'react';

```
function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);
  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
}

export function useOnlineStatus() {
  return useSyncExternalStore(
    subscribe,
    () => navigator.onLine, // How to get the value on the client
    () => true // How to get the value on the server
  );
}
```

Show more

Notice how you didn’t need to change any of the components to make this migration: function StatusBar() { const isOnline = useOnlineStatus(); // ...}function SaveButton() { const isOnline = useOnlineStatus(); // ...}

This is another reason for why wrapping Effects in custom Hooks is often beneficial:

You make the data flow to and from your Effects very explicit.

You let your components focus on the intent rather than on the exact implementation of your Effects.

When React adds new features, you can remove those Effects without changing any of your components.

Similar to a design system, you might find it helpful to start extracting common idioms from your app’s components into custom Hooks. This will keep your components’ code focused on the intent, and let you avoid writing raw Effects very often. Many excellent custom Hooks are maintained by the React community.

Deep DiveWill React provide any built-in solution for data fetching? Show DetailsWe’re still working out the details, but we expect that in the future, you’ll write data fetching like this:import { use } from 'react'; // Not available yet!function ShippingForm({ country }) { const cities = use(fetch(`/api/cities?country=${country}`)); const [city, setCity] = useState(null); const areas = city ? use(fetch(`/api/areas?city=${city}`)) : null; // ...If you use custom Hooks like useData above in your app, it will require fewer changes to migrate to the eventually recommended approach than if you write raw Effects in every component manually. However, the old approach will still work fine, so if you feel happy writing raw Effects, you can continue to do that.

There is more than one way to do it

Let’s say you want to implement a fade-in animation from scratch using the browser requestAnimationFrame API. You might start with an Effect that sets up an animation

loop. During each frame of the animation, you could change the opacity of the DOM node you hold in a ref until it reaches 1. Your code might start like this:
App.jsApp.js Download ResetForkimport { useState, useEffect, useRef } from 'react';

```
function Welcome() {
  const ref = useRef(null);

  useEffect(() => {
    const duration = 1000;
    const node = ref.current;

    let startTime = performance.now();
    let frameId = null;

    function onFrame(now) {
      const timePassed = now - startTime;
      const progress = Math.min(timePassed / duration, 1);
      onProgress(progress);
      if (progress < 1) {
        // We still have more frames to paint
        frameId = requestAnimationFrame(onFrame);
      }
    }

    function onProgress(progress) {
      node.style.opacity = progress;
    }

    function start() {
      onProgress(0);
      startTime = performance.now();
      frameId = requestAnimationFrame(onFrame);
    }

    function stop() {
      cancelAnimationFrame(frameId);
      startTime = null;
      frameId = null;
    }

    start();
    return () => stop();
  }, []);

  return (
    <h1 className="welcome" ref={ref}>
```

```
      Welcome
    </h1>
  );
}

export default function App() {
  const [show, setShow] = useState(false);
  return (
    <>
      <button onClick={() => setShow(!show)}>
        {show ? 'Remove' : 'Show'}
      </button>
      <hr />
      {show && <Welcome />}
    </>
  );
}
```

Show more

To make the component more readable, you might extract the logic into a useFadeIn custom Hook:

App.jsuseFadeIn.jsApp.js ResetForkimport { useState, useEffect, useRef } from 'react';
import { useFadeIn } from './useFadeIn.js';

```
function Welcome() {
  const ref = useRef(null);

  useFadeIn(ref, 1000);

  return (
    <h1 className="welcome" ref={ref}>
      Welcome
    </h1>
  );
}

export default function App() {
  const [show, setShow] = useState(false);
  return (
    <>
      <button onClick={() => setShow(!show)}>
        {show ? 'Remove' : 'Show'}
      </button>
      <hr />
      {show && <Welcome />}
    </>
```

```
  );
}
```

Show more

You could keep the useFadeIn code as is, but you could also refactor it more. For example, you could extract the logic for setting up the animation loop out of useFadeIn into a custom useAnimationLoop Hook:

App.jsuseFadeIn.jsuseFadeIn.js ResetForkimport { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

```
export function useFadeIn(ref, duration) {
  const [isRunning, setIsRunning] = useState(true);

  useAnimationLoop(isRunning, (timePassed) => {
    const progress = Math.min(timePassed / duration, 1);
    ref.current.style.opacity = progress;
    if (progress === 1) {
      setIsRunning(false);
    }
  });
}

function useAnimationLoop(isRunning, drawFrame) {
  const onFrame = useEffectEvent(drawFrame);

  useEffect(() => {
    if (!isRunning) {
      return;
    }

    const startTime = performance.now();
    let frameId = null;

    function tick(now) {
      const timePassed = now - startTime;
      onFrame(timePassed);
      frameId = requestAnimationFrame(tick);
    }

    tick();
    return () => cancelAnimationFrame(frameId);
  }, [isRunning]);
}
```

Show more

However, you didn’t have to do that. As with regular functions, ultimately you decide

where to draw the boundaries between different parts of your code. You could also take a very different approach. Instead of keeping the logic in the Effect, you could move most of the imperative logic inside a JavaScript class:
App.jsuseFadeIn.jsanimation.jsuseFadeIn.js ResetForkimport { useState, useEffect } from 'react';
import { FadeInAnimation } from './animation.js';

```
export function useFadeIn(ref, duration) {
  useEffect(() => {
    const animation = new FadeInAnimation(ref.current);
    animation.start(duration);
    return () => {
      animation.stop();
    };
  }, [ref, duration]);
}
```

Effects let you connect React to external systems. The more coordination between Effects is needed (for example, to chain multiple animations), the more it makes sense to extract that logic out of Effects and Hooks completely like in the sandbox above. Then, the code you extracted becomes the “external system”. This lets your Effects stay simple because they only need to send messages to the system you’ve moved outside React.
The examples above assume that the fade-in logic needs to be written in JavaScript. However, this particular fade-in animation is both simpler and much more efficient to implement with a plain CSS Animation:
App.jswelcome.csswelcome.css ResetFork.

```
welcome {
  color: white;
  padding: 50px;
  text-align: center;
  font-size: 50px;
  background-image: radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%);

  animation: fadeIn 1000ms;
}

@keyframes fadeIn {
  0% { opacity: 0; }
  100% { opacity: 1; }
}
```

Sometimes, you don’t even need a Hook!
Recap

Custom Hooks let you share logic between components.
Custom Hooks must be named starting with use followed by a capital letter.
Custom Hooks only share stateful logic, not state itself.
You can pass reactive values from one Hook to another, and they stay up-to-date.
All Hooks re-run every time your component re-renders.
The code of your custom Hooks should be pure, like your component's code.
Wrap event handlers received by custom Hooks into Effect Events.
Don't create custom Hooks like useMount. Keep their purpose specific.
It's up to you how and where to choose the boundaries of your code.

Try out some challenges1. Extract a useCounter Hook 2. Make the counter delay configurable 3. Extract useInterval out of useCounter 4. Fix a resetting interval 5. Implement a staggering movement Challenge 1 of 5: Extract a useCounter Hook This component uses a state variable and an Effect to display a number that increments every second. Extract this logic into a custom Hook called useCounter. Your goal is to make the Counter component implementation look exactly like this:export default function Counter() { const count = useCounter(); return <h1>Seconds passed: {count} </h1>;}You'll need to write your custom Hook in useCounter.js and import it into the Counter.js file.App.jsuseCounter.jsApp.js ResetForkimport { useState, useEffect } from 'react';

```
export default function Counter() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return <h1>Seconds passed: {count}</h1>;
}
```

Show solutionNext ChallengePreviousRemoving Effect Dependencies