

Quick Start – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?Learn ReactQuick StartWelcome to the React documentation! This page will give you an introduction to the 80% of React concepts that you will use on a daily basis.

You will learn

How to create and nest components

How to add markup and styles

How to display data

How to render conditions and lists

How to respond to events and update the screen

How to share data between components

Creating and nesting components

React apps are made out of components. A component is a piece of the UI (user interface) that has its own logic and appearance. A component can be as small as a button, or as large as an entire page.

React components are JavaScript functions that return markup:

```
function MyButton() { return ( <button>I'm a button</button> );}
```

Now that you've declared MyButton, you can nest it into another component:

```
export default function MyApp() { return ( <div> <h1>Welcome to my app</h1> <MyButton /> </div> );}
```

Notice that `<MyButton />` starts with a capital letter. That's how you know it's a React component. React component names must always start with a capital letter, while HTML tags must be lowercase.

Have a look at the result:

```
App.jsApp.js ResetForkfunction MyButton() {
```

```
    return (
```

```
        <button>
```

```
            I'm a button
```

```
        </button>
```

```
    );
```

```
}
```

```
export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton />
    </div>
  );
}
```

Show more

The `export default` keywords specify the main component in the file. If you're not familiar with some piece of JavaScript syntax, MDN and [javascript.info](#) have great references.

Writing markup with JSX

The markup syntax you've seen above is called JSX. It is optional, but most React projects use JSX for its convenience. All of the tools we recommend for local development support JSX out of the box.

JSX is stricter than HTML. You have to close tags like `
`. Your component also can't return multiple JSX tags. You have to wrap them into a shared parent, like a `<div>...</div>` or an empty `<>...</>` wrapper:

```
function AboutPage() { return (  <>    <h1>About</h1>    <p>Hello there.<br />How do you do?</p>  </> );}
```

If you have a lot of HTML to port to JSX, you can use an online converter.

Adding styles

In React, you specify a CSS class with `className`. It works the same way as the HTML `class` attribute:

```
<img className="avatar" />
```

Then you write the CSS rules for it in a separate CSS file:

```
/* In your CSS */
.avatar { border-radius: 50%;}
```

React does not prescribe how you add CSS files. In the simplest case, you'll add a `<link>` tag to your HTML. If you use a build tool or a framework, consult its documentation to learn how to add a CSS file to your project.

Displaying data

JSX lets you put markup into JavaScript. Curly braces let you "escape back" into JavaScript so that you can embed some variable from your code and display it to the user. For example, this will display `user.name`:

```
return ( <h1> {user.name} </h1>);
```

You can also "escape into JavaScript" from JSX attributes, but you have to use curly braces instead of quotes. For example, `className="avatar"` passes the "avatar" string as the CSS class, but `src={user.imageUrl}` reads the JavaScript `user.imageUrl` variable value, and then passes that value as the `src` attribute:

```
return ( <img className="avatar" src={user.imageUrl} />);
```

You can put more complex expressions inside the JSX curly braces too, for example, string concatenation:

```
App.js
App.js
Reset
Fork
const user = {
```

```
name: 'Hedy Lamarr',
imageUrl: 'https://i.imgur.com/yXOvdOSs.jpg',
imageSize: 90,
};

export default function Profile() {
  return (
    <>
    <h1>{user.name}</h1>
    <img
      className="avatar"
      src={user.imageUrl}
      alt={'Photo of ' + user.name}
      style={{
        width: user.imageSize,
        height: user.imageSize
      }}
    />
    </>
  );
}


```

Show more

In the above example, `style={{}}` is not a special syntax, but a regular {} object inside the `style={ }` JSX curly braces. You can use the `style` attribute when your styles depend on JavaScript variables.

Conditional rendering

In React, there is no special syntax for writing conditions. Instead, you'll use the same techniques as you use when writing regular JavaScript code. For example, you can use an if statement to conditionally include JSX:

```
let content;if (isLoggedIn) { content = <AdminPanel /> } else { content = <LoginForm /> }return ( <div> {content} </div> );
```

If you prefer more compact code, you can use the conditional ? operator. Unlike if, it works inside JSX:

```
<div> {isLoggedIn ? ( <AdminPanel /> ) : ( <LoginForm /> )}</div>
```

When you don't need the else branch, you can also use a shorter logical && syntax:

```
<div> {isLoggedIn && <AdminPanel />}</div>
```

All of these approaches also work for conditionally specifying attributes. If you're unfamiliar with some of this JavaScript syntax, you can start by always using if...else.

Rendering lists

You will rely on JavaScript features like for loop and the array map() function to render lists of components.

For example, let's say you have an array of products:

```
const products = [ { title: 'Cabbage', id: 1 }, { title: 'Garlic', id: 2 }, { title: 'Apple', id: 3 },];
```

Inside your component, use the map() function to transform an array of products into an array of items:

```
const listItems = products.map(product => <li key={product.id}> {product.title} </li>);return (<ul>{listItems}</ul>);
```

Notice how has a key attribute. For each item in a list, you should pass a string or a number that uniquely identifies that item among its siblings. Usually, a key should be coming from your data, such as a database ID. React uses your keys to know what happened if you later insert, delete, or reorder the items.

```
App.jsApp.js ResetForkconst products = [
```

```
  { title: 'Cabbage', isFruit: false, id: 1 },
```

```
  { title: 'Garlic', isFruit: false, id: 2 },
```

```
  { title: 'Apple', isFruit: true, id: 3 },
```

```
];
```

```
export default function ShoppingList() {
```

```
  const listItems = products.map(product =>
```

```
    <li
```

```
      key={product.id}
```

```
      style={{
```

```
        color: product.isFruit ? 'magenta' : 'darkgreen'
```

```
      }}
```

```
    >
```

```
      {product.title}
```

```
    </li>
```

```
  );
```

```
  return (
```

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

```
  );
```

```
}
```

Show more

Responding to events

You can respond to events by declaring event handler functions inside your components:

```
function MyButton() { function handleClick() { alert('You clicked me!'); } return ( <button onClick={handleClick}> Click me </button> );}
```

Notice how `onClick={handleClick}` has no parentheses at the end! Do not call the event handler function: you only need to pass it down. React will call your event handler when the user clicks the button.

Updating the screen

Often, you'll want your component to "remember" some information and display it. For example, maybe you want to count the number of times a button is clicked. To do this, add state to your component.

First, import `useState` from React:

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

Now you can declare a state variable inside your component:

```
function MyButton() { const [count, setCount] = useState(0); // ...
```

You'll get two things from `useState`: the current state (`count`), and the function that lets you update it (`setCount`). You can give them any names, but the convention is to write `[something, setSomething]`.

The first time the button is displayed, `count` will be 0 because you passed 0 to `useState()`. When you want to change state, call `setCount()` and pass the new value to it. Clicking this button will increment the counter:

```
function MyButton() { const [count, setCount] = useState(0); function handleClick() { setCount(count + 1); } return ( <button onClick={handleClick}> Clicked {count} times </button> );}
```

React will call your component function again. This time, `count` will be 1. Then it will be 2. And so on.

If you render the same component multiple times, each will get its own state. Click each button separately:

```
App.jsApp.js ResetForKimport { useState } from 'react';
```

```
export default function MyApp() {
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}
```

```
function MyButton() {
  const [count, setCount] = useState(0);

  function handleClick() {
```

```

    setCount(count + 1);

}

return (
  <button onClick={handleClick}>
    Clicked {count} times
  </button>
);
}

```

[Show more](#)

Notice how each button “remembers” its own count state and doesn’t affect other buttons.

Using Hooks

Functions starting with `use` are called Hooks. `useState` is a built-in Hook provided by React. You can find other built-in Hooks in the API reference. You can also write your own Hooks by combining the existing ones.

Hooks are more restrictive than other functions. You can only call Hooks at the top of your components (or other Hooks). If you want to use `useState` in a condition or a loop, extract a new component and put it there.

Sharing data between components

In the previous example, each `MyButton` had its own independent count, and when each button was clicked, only the count for the button clicked changed:

Initially, each `MyButton`’s count state is 0The first `MyButton` updates its count to 1

However, often you’ll need components to share data and always update together.

To make both `MyButton` components display the same count and update together, you need to move the state from the individual buttons “upwards” to the closest component containing all of them.

In this example, it is `MyApp`:

Initially, `MyApp`’s count state is 0 and is passed down to both childrenOn click, `MyApp` updates its count state to 1 and passes it down to both children

Now when you click either button, the count in `MyApp` will change, which will change both of the counts in `MyButton`. Here’s how you can express this in code.

First, move the state up from `MyButton` into `MyApp`:

```

export default function MyApp() {
  const [count, setCount] = useState(0);
  function handleClick() {
    setCount(count + 1);
  }
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

function MyButton() {
  // ... we're moving code from here ...
}

```

Then, pass the state down from `MyApp` to each `MyButton`, together with the shared click handler. You can pass information to `MyButton` using the JSX curly braces, just like you previously did with built-in tags like ``:

```

export default function MyApp() {
  const [count, setCount] = useState(0);
  function handleClick() {
    setCount(count + 1);
  }
  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}

```

The information you pass down like this is called props. Now the MyApp component contains the count state and the handleClick event handler, and passes both of them down as props to each of the buttons.

Finally, change MyButton to read the props you have passed from its parent component:

```
function MyButton({ count, onClick }) { return ( <button onClick={onClick}> Clicked {count} times </button> );}
```

When you click the button, the onClick handler fires. Each button's onClick prop was set to the handleClick function inside MyApp, so the code inside of it runs. That code calls setCount(count + 1), incrementing the count state variable. The new count value is passed as a prop to each button, so they all show the new value. This is called "lifting state up". By moving state up, you've shared it between components.

```
App.js
```

```
App.js
```

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

```
export default function MyApp() {
```

```
  const [count, setCount] = useState(0);
```

```
  function handleClick() {
```

```
    setCount(count + 1);
```

```
  }
```

```
  return (
```

```
    <div>
```

```
      <h1>Counters that update together</h1>
```

```
      <MyButton count={count} onClick={handleClick} />
```

```
      <MyButton count={count} onClick={handleClick} />
```

```
    </div>
```

```
  );
```

```
}
```

```
function MyButton({ count, onClick }) {
```

```
  return (
```

```
    <button onClick={onClick}>
```

```
      Clicked {count} times
```

```
    </button>
```

```
  );
```

```
}
```

Show more

Next Steps

By now, you know the basics of how to write React code!

Check out the Tutorial to put them into practice and build your first mini-app with React.

NextTutorial: Tic-Tac-Toe

©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewCreating and nesting components Writing markup with JSX Adding styles Displaying data Conditional rendering Rendering lists Responding to events Updating the screen Using Hooks Sharing data between components Next Steps Tutorial: Tic-Tac-Toe – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET

STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?Learn ReactQuick StartTutorial: Tic-Tac-ToeYou will build a small tic-tac-toe game during this tutorial. This tutorial does not assume any existing React knowledge. The techniques you'll learn in the tutorial are fundamental to building any React app, and fully understanding it will give you a deep understanding of React.

NoteThis tutorial is designed for people who prefer to learn by doing and want to quickly try making something tangible. If you prefer learning each concept step by step, start with Describing the UI.

The tutorial is divided into several sections:

Setup for the tutorial will give you a starting point to follow the tutorial.

Overview will teach you the fundamentals of React: components, props, and state.

Completing the game will teach you the most common techniques in React development.

Adding time travel will give you a deeper insight into the unique strengths of React.

What are you building?

In this tutorial, you'll build an interactive tic-tac-toe game with React.

You can see what it will look like when you're finished here:

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

```
function Square({ value, onSquareClick }) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}
```

```
}
```

```
function Board({ xIsNext, squares, onPlay }) {  
  function handleClick(i) {  
    if (calculateWinner(squares) || squares[i]) {  
      return;  
    }  
    const nextSquares = squares.slice();  
    if (xIsNext) {  
      nextSquares[i] = 'X';  
    } else {  
      nextSquares[i] = 'O';  
    }  
    onPlay(nextSquares);  
  }  
  
  const winner = calculateWinner(squares);  
  let status;  
  if (winner) {  
    status = 'Winner: ' + winner;  
  } else {  
    status = 'Next player: ' + (xIsNext ? 'X' : 'O');  
  }  
  
  return (  
    <>  
    <div className="status">{status}</div>  
    <div className="board-row">  
      <Square value={squares[0]} onClick={() => handleClick(0)} />  
      <Square value={squares[1]} onClick={() => handleClick(1)} />  
      <Square value={squares[2]} onClick={() => handleClick(2)} />  
    </div>  
    <div className="board-row">  
      <Square value={squares[3]} onClick={() => handleClick(3)} />
```

```
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
  <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
  <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
  <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}
```

```
export default function Game() {
```

```
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const xIsNext = currentMove % 2 === 0;
  const currentSquares = history[currentMove];
```

```
  function handlePlay(nextSquares) {
```

```
    const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
    setHistory(nextHistory);
    setCurrentMove(nextHistory.length - 1);
  }
```

```
  function jumpTo(nextMove) {
```

```
    setCurrentMove(nextMove);
  }
```

```
  const moves = history.map((squares, move) => {
    let description;
    if (move > 0) {
      description = 'Go to move #' + move;
    } else {
      description = 'Go to game start';
    }
  });
}
```

```
    }

    return (
      <li key={move}>
        <button onClick={() => jumpTo(move)}>{description}</button>
      </li>
    );
  });

return (
  <div className="game">
    <div className="game-board">
      <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
    </div>
    <div className="game-info">
      <ol>{moves}</ol>
    </div>
  </div>
);

}

function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
```

```
    return squares[a];  
  }  
}  
return null;  
}
```

Show more

If the code doesn't make sense to you yet, or if you are unfamiliar with the code's syntax, don't worry! The goal of this tutorial is to help you understand React and its syntax.

We recommend that you check out the tic-tac-toe game above before continuing with the tutorial. One of the features that you'll notice is that there is a numbered list to the right of the game's board. This list gives you a history of all of the moves that have occurred in the game, and it is updated as the game progresses.

Once you've played around with the finished tic-tac-toe game, keep scrolling. You'll start with a simpler template in this tutorial. Our next step is to set you up so that you can start building the game.

Setup for the tutorial

In the live code editor below, click Fork in the top-right corner to open the editor in a new tab using the website CodeSandbox. CodeSandbox lets you write code in your browser and preview how your users will see the app you've created. The new tab should display an empty square and the starter code for this tutorial.

```
App.jsApp.js ResetForkexport default function Square() {  
  return <button className="square">X</button>;  
}
```

Note You can also follow this tutorial using your local development environment. To do this, you need to:

Install Node.js

In the CodeSandbox tab you opened earlier, press the top-left corner button to open the menu, and then choose Download Sandbox in that menu to download an archive of the files locally

Unzip the archive, then open a terminal and cd to the directory you unzipped

Install the dependencies with npm install

Run npm start to start a local server and follow the prompts to view the code running in a browser

If you get stuck, don't let this stop you! Follow along online instead and try a local setup again later.

Overview

Now that you're set up, let's get an overview of React!

Inspecting the starter code

In CodeSandbox you'll see three main sections:

The Files section with a list of files like App.js, index.js, styles.css and a folder called public

The code editor where you'll see the source code of your selected file

The browser section where you'll see how the code you've written will be displayed

The App.js file should be selected in the Files section. The contents of that file in the code editor should be:

```
export default function Square() { return <button className="square">X</button>;}
```

The browser section should be displaying a square with a X in it like this:

Now let's have a look at the files in the starter code.

App.js

The code in App.js creates a component. In React, a component is a piece of reusable code that represents a part of a user interface. Components are used to render, manage, and update the UI elements in your application. Let's look at the component line by line to see what's going on:

```
export default function Square() { return <button className="square">X</button>;}
```

The first line defines a function called Square. The export JavaScript keyword makes this function accessible outside of this file. The default keyword tells other files using your code that it's the main function in your file.

```
export default function Square() { return <button className="square">X</button>;}
```

The second line returns a button. The return JavaScript keyword means whatever comes after is returned as a value to the caller of the function. <button> is a JSX element. A JSX element is a combination of JavaScript code and HTML tags that describes what you'd like to display. className="square" is a button property or prop that tells CSS how to style the button. X is the text displayed inside of the button and </button> closes the JSX element to indicate that any following content shouldn't be placed inside the button.

styles.css

Click on the file labeled styles.css in the Files section of CodeSandbox. This file defines the styles for your React app. The first two CSS selectors (* and body) define the style of large parts of your app while the .square selector defines the style of any component where the className property is set to square. In your code, that would match the button from your Square component in the App.js file.

index.js

Click on the file labeled index.js in the Files section of CodeSandbox. You won't be editing this file during the tutorial but it is the bridge between the component you created in the App.js file and the web browser.

```
import { StrictMode } from 'react';import { createRoot } from 'react-dom/client';import './styles.css';import App from './App';
```

Lines 1-5 bring all the necessary pieces together:

React

React's library to talk to web browsers (React DOM)

the styles for your components

the component you created in App.js.

The remainder of the file brings all the pieces together and injects the final product into index.html in the public folder.

Building the board

Let's get back to App.js. This is where you'll spend the rest of the tutorial.

Currently the board is only a single square, but you need nine! If you just try and copy paste your square to make two squares like this:

```
export default function Square() { return <button className="square">X</button><button  
className="square">X</button>;}
```

You'll get this error:

Console/src/App.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX Fragment
<>...</>?

React components need to return a single JSX element and not multiple adjacent JSX elements like two buttons. To fix this you can use Fragments (<> and </>) to wrap multiple adjacent JSX elements like this:

```
export default function Square() { return ( <> <button className="square">X</button> <button  
className="square">X</button> </> );}
```

Now you should see:

Great! Now you just need to copy-paste a few times to add nine squares and...

Oh no! The squares are all in a single line, not in a grid like you need for our board. To fix this you'll need to group your squares into rows with divs and add some CSS classes. While you're at it, you'll give each square a number to make sure you know where each square is displayed.

In the App.js file, update the Square component to look like this:

```
export default function Square() { return ( <> <div className="board-row"> <button  
className="square">1</button> <button className="square">2</button> <button  
className="square">3</button> </div> <div className="board-row"> <button  
className="square">4</button> <button className="square">5</button> <button  
className="square">6</button> </div> <div className="board-row"> <button  
className="square">7</button> <button className="square">8</button> <button  
className="square">9</button> </div> </> );}
```

The CSS defined in styles.css styles the divs with the className of board-row. Now that you've grouped your components into rows with the styled divs you have your tic-tac-toe board:

But you now have a problem. Your component named Square, really isn't a square anymore. Let's fix that by changing the name to Board:

```
export default function Board() { //...}
```

At this point your code should look something like this:

```
App.jsApp.js ResetForkexport default function Board() {  
  
return (
```

```

<>

<div className="board-row">
  <button className="square">1</button>
  <button className="square">2</button>
  <button className="square">3</button>
</div>

<div className="board-row">
  <button className="square">4</button>
  <button className="square">5</button>
  <button className="square">6</button>
</div>

<div className="board-row">
  <button className="square">7</button>
  <button className="square">8</button>
  <button className="square">9</button>
</div>

</>
);

}

```

Show more

NotePsssst... That's a lot to type! It's okay to copy and paste code from this page. However, if you're up for a little challenge, we recommend only copying code that you've manually typed at least once yourself.

Passing data through props

Next, you'll want to change the value of a square from empty to "X" when the user clicks on the square. With how you've built the board so far you would need to copy-paste the code that updates the square nine times (once for each square you have)! Instead of copy-pasting, React's component architecture allows you to create a reusable component to avoid messy, duplicated code.

First, you are going to copy the line defining your first square (`<button className="square">1</button>`) from your Board component into a new Square component:

```
function Square() { return <button className="square">1</button> } export default function Board() { // ... }
```

Then you'll update the Board component to render that Square component using JSX syntax:

```
// ...export default function Board() { return ( <>   <div className="board-row">     <Square />     <Square />
<Square />   </div>   <div className="board-row">     <Square />     <Square />     <Square />   </div>
<div className="board-row">     <Square />     <Square />     <Square />   </div> ); }
```

Note how unlike the browser divs, your own components Board and Square must start with a capital letter.

Let's take a look:

Oh no! You lost the numbered squares you had before. Now each square says “1”. To fix this, you will use props to pass the value each square should have from the parent component (Board) to its child (Square).

Update the Square component to read the value prop that you’ll pass from the Board:

```
function Square({ value }) { return <button className="square">1</button>;}
```

function Square({ value }) indicates the Square component can be passed a prop called value.

Now you want to display that value instead of 1 inside every square. Try doing it like this:

```
function Square({ value }) { return <button className="square">value</button>;}
```

Oops, this is not what you wanted:

You wanted to render the JavaScript variable called value from your component, not the word “value”. To “escape into JavaScript” from JSX, you need curly braces. Add curly braces around value in JSX like so:

```
function Square({ value }) { return <button className="square">{value}</button>;}
```

For now, you should see an empty board:

This is because the Board component hasn’t passed the value prop to each Square component it renders yet. To fix it you’ll add the value prop to each Square component rendered by the Board component:

```
export default function Board() { return ( <>    <div className="board-row">      <Square value="1" />
<Square value="2" />      <Square value="3" />      </div>    <div className="board-row">      <Square value="4" />
<Square value="5" />      <Square value="6" />      </div>    <div className="board-row">      <Square value="7" />
<Square value="8" />      <Square value="9" />      </div>  );}
```

Now you should see a grid of numbers again:

Your updated code should look like this:

```
App.js
Reset
function Square({ value }) {
  return <button className="square">{value}</button>;
}
```

```
export default function Board() {
```

```
  return (
    <>
    <div className="board-row">
      <Square value="1" />
      <Square value="2" />
      <Square value="3" />
    </div>
```

```

<div className="board-row">
  <Square value="4" />
  <Square value="5" />
  <Square value="6" />
</div>

<div className="board-row">
  <Square value="7" />
  <Square value="8" />
  <Square value="9" />
</div>
</>
);

}

```

Show more

Making an interactive component

Let's fill the Square component with an X when you click it. Declare a function called handleClick inside of the Square. Then, add onClick to the props of the button JSX element returned from the Square:

```
function Square({ value }) { function handleClick() { console.log('clicked!'); } return ( <button
  className="square"    onClick={handleClick} >    {value}  </button> );}
```

If you click on a square now, you should see a log saying "clicked!" in the Console tab at the bottom of the Browser section in CodeSandbox. Clicking the square more than once will log "clicked!" again. Repeated console logs with the same message will not create more lines in the console. Instead, you will see an incrementing counter next to your first "clicked!" log.

NoteIf you are following this tutorial using your local development environment, you need to open your browser's Console. For example, if you use the Chrome browser, you can view the Console with the keyboard shortcut Shift + Ctrl + J (on Windows/Linux) or Option + ⌘ + J (on macOS).

As a next step, you want the Square component to “remember” that it got clicked, and fill it with an “X” mark. To “remember” things, components use state.

React provides a special function called useState that you can call from your component to let it “remember” things. Let's store the current value of the Square in state, and change it when the Square is clicked.

Import useState at the top of the file. Remove the value prop from the Square component. Instead, add a new line at the start of the Square that calls useState. Have it return a state variable called value:

```
import { useState } from 'react';function Square() { const [value, setValue] = useState(null); function handleClick() {
//...}}
```

value stores the value and setValue is a function that can be used to change the value. The null passed to useState is used as the initial value for this state variable, so value here starts off equal to null.

Since the Square component no longer accepts props anymore, you'll remove the value prop from all nine of the Square components created by the Board component:

```
// ...export default function Board() { return ( <>   <div className="board-row">     <Square />     <Square />
<Square />   </div>   <div className="board-row">     <Square />     <Square />     <Square />   </div>
<div className="board-row">     <Square />     <Square />     <Square />   </div> </> );}
```

Now you'll change Square to display an "X" when clicked. Replace the `console.log("clicked!");` event handler with `setValue('X');`. Now your Square component looks like this:

```
function Square() { const [value, setValue] = useState(null); function handleClick() { setValue('X'); } return (
<button    className="square"    onClick={handleClick} >    {value}  </button> );}
```

By calling this set function from an `onClick` handler, you're telling React to re-render that Square whenever its `<button>` is clicked. After the update, the Square's value will be 'X', so you'll see the "X" on the game board. Click on any Square, and "X" should show up:

Each Square has its own state: the value stored in each Square is completely independent of the others. When you call a set function in a component, React automatically updates the child components inside too.

After you've made the above changes, your code will look like this:

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

```
function Square() {
  const [value, setValue] = useState(null);
```

```
  function handleClick() {
```

```
    setValue('X');
```

```
}
```

```
  return (
```

```
    <button
```

```
      className="square"
```

```
      onClick={handleClick}
```

```
>
```

```
    {value}
```

```
  </button>
```

```
);
```

```
}
```

```
export default function Board() {
```

```
  return (
```

```
    <>
```

```
    <div className="board-row">
```

```
<Square />
<Square />
<Square />
</div>
<div className="board-row">
  <Square />
  <Square />
  <Square />
</div>
<div className="board-row">
  <Square />
  <Square />
  <Square />
</div>
</>
);
```

}

Show more

React Developer Tools

React DevTools let you check the props and the state of your React components. You can find the React DevTools tab at the bottom of the browser section in CodeSandbox:

To inspect a particular component on the screen, use the button in the top left corner of React DevTools:

NoteFor local development, React DevTools is available as a Chrome, Firefox, and Edge browser extension. Install it, and the Components tab will appear in your browser Developer Tools for sites using React.

Completing the game

By this point, you have all the basic building blocks for your tic-tac-toe game. To have a complete game, you now need to alternate placing “X”s and “O”s on the board, and you need a way to determine a winner.

Lifting state up

Currently, each Square component maintains a part of the game’s state. To check for a winner in a tic-tac-toe game, the Board would need to somehow know the state of each of the 9 Square components.

How would you approach that? At first, you might guess that the Board needs to “ask” each Square for that Square’s state. Although this approach is technically possible in React, we discourage it because the code becomes difficult to understand, susceptible to bugs, and hard to refactor. Instead, the best approach is to store the game’s state in the

parent Board component instead of in each Square. The Board component can tell each Square what to display by passing a prop, like you did when you passed a number to each Square.

To collect data from multiple children, or to have two child components communicate with each other, declare the shared state in their parent component instead. The parent component can pass that state back down to the children via props. This keeps the child components in sync with each other and with their parent.

Lifting state into a parent component is common when React components are refactored.

Let's take this opportunity to try it out. Edit the Board component so that it declares a state variable named squares that defaults to an array of 9 nulls corresponding to the 9 squares:

```
// ...export default function Board() { const [squares, setSquares] = useState(Array(9).fill(null)); return ( // ... );}
```

Array(9).fill(null) creates an array with nine elements and sets each of them to null. The useState() call around it declares a squares state variable that's initially set to that array. Each entry in the array corresponds to the value of a square. When you fill the board in later, the squares array will look like this:

```
['O', null, 'X', 'X', 'X', 'O', 'O', null, null]
```

Now your Board component needs to pass the value prop down to each Square that it renders:

```
export default function Board() { const [squares, setSquares] = useState(Array(9).fill(null)); return ( <> <div className="board-row"> <Square value={squares[0]} /> <Square value={squares[1]} /> <Square value={squares[2]} /> </div> <div className="board-row"> <Square value={squares[3]} /> <Square value={squares[4]} /> <Square value={squares[5]} /> </div> <div className="board-row"> <Square value={squares[6]} /> <Square value={squares[7]} /> <Square value={squares[8]} /> </div> </> );}
```

Next, you'll edit the Square component to receive the value prop from the Board component. This will require removing the Square component's own stateful tracking of value and the button's onClick prop:

```
function Square({value}) { return <button className="square">{value}</button>;}
```

At this point you should see an empty tic-tac-toe board:

And your code should look like this:

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

```
function Square({ value }) {
  return <button className="square">{value}</button>;
}
```

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));
  return (
    <>
    <div className="board-row">
      <Square value={squares[0]} />
      <Square value={squares[1]} />
```

```

<Square value={squares[2]} />
</div>
<div className="board-row">
  <Square value={squares[3]} />
  <Square value={squares[4]} />
  <Square value={squares[5]} />
</div>
<div className="board-row">
  <Square value={squares[6]} />
  <Square value={squares[7]} />
  <Square value={squares[8]} />
</div>
</>
);
}

```

Show more

Each Square will now receive a value prop that will either be 'X', 'O', or null for empty squares.

Next, you need to change what happens when a Square is clicked. The Board component now maintains which squares are filled. You'll need to create a way for the Square to update the Board's state. Since state is private to a component that defines it, you cannot update the Board's state directly from Square.

Instead, you'll pass down a function from the Board component to the Square component, and you'll have Square call that function when a square is clicked. You'll start with the function that the Square component will call when it is clicked. You'll call that function `onSquareClick`:

```
function Square({ value }) { return ( <button className="square" onClick={onSquareClick}> {value} </button> ); }
```

Next, you'll add the `onSquareClick` function to the Square component's props:

```
function Square({ value, onSquareClick }) { return ( <button className="square" onClick={onSquareClick}> {value} </button> ); }
```

Now you'll connect the `onSquareClick` prop to a function in the Board component that you'll name `handleClick`. To connect `onSquareClick` to `handleClick` you'll pass a function to the `onSquareClick` prop of the first Square component:

```
export default function Board() { const [squares, setSquares] = useState(Array(9).fill(null)); return ( <> <div className="board-row"> <Square value={squares[0]} onSquareClick={handleClick} /> //... ); }
```

Lastly, you will define the `handleClick` function inside the Board component to update the squares array holding your board's state:

```
export default function Board() { const [squares, setSquares] = useState(Array(9).fill(null)); function handleClick() { const nextSquares = squares.slice(); nextSquares[0] = "X"; setSquares(nextSquares); } return ( // ... ) }
```

The handleClick function creates a copy of the squares array (nextSquares) with the JavaScript slice() Array method. Then, handleClick updates the nextSquares array to add X to the first ([0] index) square.

Calling the setSquares function lets React know the state of the component has changed. This will trigger a re-render of the components that use the squares state (Board) as well as its child components (the Square components that make up the board).

Note JavaScript supports closures which means an inner function (e.g. handleClick) has access to variables and functions defined in a outer function (e.g. Board). The handleClick function can read the squares state and call the setSquares method because they are both defined inside of the Board function.

Now you can add X's to the board... but only to the upper left square. Your handleClick function is hardcoded to update the index for the upper left square (0). Let's update handleClick to be able to update any square. Add an argument i to the handleClick function that takes the index of the square to update:

```
export default function Board() { const [squares, setSquares] = useState(Array(9).fill(null)); function handleClick(i) { const nextSquares = squares.slice(); nextSquares[i] = "X"; setSquares(nextSquares); } return ( // ... )}
```

Next, you will need to pass that i to handleClick. You could try to set the onSquareClick prop of square to be handleClick(0) directly in the JSX like this, but it won't work:

```
<Square value={squares[0]} onSquareClick={handleClick(0)} />
```

Here is why this doesn't work. The handleClick(0) call will be a part of rendering the board component. Because handleClick(0) alters the state of the board component by calling setSquares, your entire board component will be re-rendered again. But this runs handleClick(0) again, leading to an infinite loop:

Console Too many re-renders. React limits the number of renders to prevent an infinite loop.

Why didn't this problem happen earlier?

When you were passing onSquareClick={handleClick}, you were passing the handleClick function down as a prop. You were not calling it! But now you are calling that function right away—notice the parentheses in handleClick(0)—and that's why it runs too early. You don't want to call handleClick until the user clicks!

You could fix this by creating a function like handleFirstSquareClick that calls handleClick(0), a function like handleSecondSquareClick that calls handleClick(1), and so on. You would pass (rather than call) these functions down as props like onSquareClick={handleFirstSquareClick}. This would solve the infinite loop.

However, defining nine different functions and giving each of them a name is too verbose. Instead, let's do this:

```
export default function Board() { // ... return ( <> <div className="board-row"> <Square value={squares[0]} onSquareClick={() => handleClick(0)} /> // ... );}
```

Notice the new () => syntax. Here, () => handleClick(0) is an arrow function, which is a shorter way to define functions. When the square is clicked, the code after the => “arrow” will run, calling handleClick(0).

Now you need to update the other eight squares to call handleClick from the arrow functions you pass. Make sure that the argument for each call of the handleClick corresponds to the index of the correct square:

```
export default function Board() { // ... return ( <> <div className="board-row"> <Square value={squares[0]} onSquareClick={() => handleClick(0)} /> <Square value={squares[1]} onSquareClick={() => handleClick(1)} /> <Square value={squares[2]} onSquareClick={() => handleClick(2)} /> </div> <div className="board-row"> <Square value={squares[3]} onSquareClick={() => handleClick(3)} /> <Square value={squares[4]} onSquareClick={() => handleClick(4)} /> <Square value={squares[5]} onSquareClick={() => handleClick(5)} /> </div> <div className="board-row"> <Square value={squares[6]} onSquareClick={() => handleClick(6)} /> <Square value={squares[7]} onSquareClick={() => handleClick(7)} /> <Square value={squares[8]} onSquareClick={() => handleClick(8)} /> </div> </> );};
```

Now you can again add X's to any square on the board by clicking on them:

But this time all the state management is handled by the Board component!

This is what your code should look like:

```
App.js
import { useState } from 'react';

function Square({ value, onSquareClick }) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}

export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick(i) {
    const nextSquares = squares.slice();
    nextSquares[i] = 'X';
    setSquares(nextSquares);
  }

  return (
    <>
    <div className="board-row">
      <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
      <Square value={squares[1]} onSquareClick={() => handleClick(1)} />
      <Square value={squares[2]} onSquareClick={() => handleClick(2)} />
    </div>
    <div className="board-row">
      <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
      <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
      <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
    </div>
  )
}
```

```

<div className="board-row">
  <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
  <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
  <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}

```

Show more

Now that your state handling is in the Board component, the parent Board component passes props to the child Square components so that they can be displayed correctly. When clicking on a Square, the child Square component now asks the parent Board component to update the state of the board. When the Board's state changes, both the Board component and every child Square re-renders automatically. Keeping the state of all squares in the Board component will allow it to determine the winner in the future.

Let's recap what happens when a user clicks the top left square on your board to add an X to it:

Clicking on the upper left square runs the function that the button received as its onClick prop from the Square. The Square component received that function as its onSquareClick prop from the Board. The Board component defined that function directly in the JSX. It calls handleClick with an argument of 0.

handleClick uses the argument (0) to update the first element of the squares array from null to X.

The squares state of the Board component was updated, so the Board and all of its children re-render. This causes the value prop of the Square component with index 0 to change from null to X.

In the end the user sees that the upper left square has changed from empty to having a X after clicking it.

NoteThe DOM <button> element's onClick attribute has a special meaning to React because it is a built-in component. For custom components like Square, the naming is up to you. You could give any name to the Square's onSquareClick prop or Board's handleClick function, and the code would work the same. In React, it's conventional to use onSomething names for props which represent events and handleSomething for the function definitions which handle those events.

Why immutability is important

Note how in handleClick, you call .slice() to create a copy of the squares array instead of modifying the existing array. To explain why, we need to discuss immutability and why immutability is important to learn.

There are generally two approaches to changing data. The first approach is to mutate the data by directly changing the data's values. The second approach is to replace the data with a new copy which has the desired changes. Here is what it would look like if you mutated the squares array:

```
const squares = [null, null, null, null, null, null, null, null, null]; squares[0] = 'X'; // Now `squares` is ["X", null, null, null, null, null, null, null, null];
```

And here is what it would look like if you changed data without mutating the squares array:

```
const squares = [null, null, null, null, null, null, null, null, null];const nextSquares = ['X', null, null, null, null, null, null, null, null];// Now `squares` is unchanged, but `nextSquares` first element is 'X' rather than 'null'
```

The result is the same but by not mutating (changing the underlying data) directly, you gain several benefits.

Immutability makes complex features much easier to implement. Later in this tutorial, you will implement a “time travel” feature that lets you review the game’s history and “jump back” to past moves. This functionality isn’t specific to games—an ability to undo and redo certain actions is a common requirement for apps. Avoiding direct data mutation lets you keep previous versions of the data intact, and reuse them later.

There is also another benefit of immutability. By default, all child components re-render automatically when the state of a parent component changes. This includes even the child components that weren’t affected by the change. Although re-rendering is not by itself noticeable to the user (you shouldn’t actively try to avoid it!), you might want to skip re-rendering a part of the tree that clearly wasn’t affected by it for performance reasons. Immutability makes it very cheap for components to compare whether their data has changed or not. You can learn more about how React chooses when to re-render a component in the memo API reference.

Taking turns

It’s now time to fix a major defect in this tic-tac-toe game: the “O”s cannot be marked on the board.

You’ll set the first move to be “X” by default. Let’s keep track of this by adding another piece of state to the Board component:

```
function Board() { const [xIsNext, setXIsNext] = useState(true); const [squares, setSquares] = useState(Array(9).fill(null)); // ...}
```

Each time a player moves, `xIsNext` (a boolean) will be flipped to determine which player goes next and the game’s state will be saved. You’ll update the Board’s `handleClick` function to flip the value of `xIsNext`:

```
export default function Board() { const [xIsNext, setXIsNext] = useState(true); const [squares, setSquares] = useState(Array(9).fill(null)); function handleClick(i) { const nextSquares = squares.slice(); if (xIsNext) { nextSquares[i] = "X"; } else { nextSquares[i] = "O"; } setSquares(nextSquares); setXIsNext(!xIsNext); } return ( //... );}
```

Now, as you click on different squares, they will alternate between X and O, as they should!

But wait, there’s a problem. Try clicking on the same square multiple times:

The X is overwritten by an O! While this would add a very interesting twist to the game, we’re going to stick to the original rules for now.

When you mark a square with a X or an O you aren’t first checking to see if the square already has a X or O value. You can fix this by returning early. You’ll check to see if the square already has a X or an O. If the square is already filled, you will return in the `handleClick` function early—before it tries to update the board state.

```
function handleClick(i) { if (squares[i]) { return; } const nextSquares = squares.slice(); //...}
```

Now you can only add X’s or O’s to empty squares! Here is what your code should look like at this point:

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

```
function Square({value, onSquareClick}) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}
```

```
</button>
);

}

export default function Board() {
  const [xIsNext, setXIsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick(i) {
    if (squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
    setSquares(nextSquares);
    setXIsNext(!xIsNext);
  }

  return (
    <>
    <div className="board-row">
      <Square value={squares[0]} onClick={() => handleClick(0)} />
      <Square value={squares[1]} onClick={() => handleClick(1)} />
      <Square value={squares[2]} onClick={() => handleClick(2)} />
    </div>
    <div className="board-row">
      <Square value={squares[3]} onClick={() => handleClick(3)} />
      <Square value={squares[4]} onClick={() => handleClick(4)} />
      <Square value={squares[5]} onClick={() => handleClick(5)} />
    </div>
  )
}
```

```

<div className="board-row">
  <Square value={squares[6]} onClick={() => handleClick(6)} />
  <Square value={squares[7]} onClick={() => handleClick(7)} />
  <Square value={squares[8]} onClick={() => handleClick(8)} />
</div>
</>
);
}

```

Show more

Declaring a winner

Now that the players can take turns, you'll want to show when the game is won and there are no more turns to make. To do this you'll add a helper function called calculateWinner that takes an array of 9 squares, checks for a winner and returns 'X', 'O', or null as appropriate. Don't worry too much about the calculateWinner function; it's not specific to React:

```
export default function Board() { //...}function calculateWinner(squares) { const lines = [ [0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7], [2, 5, 8], [0, 4, 8], [2, 4, 6] ]; for (let i = 0; i < lines.length; i++) { const [a, b, c] = lines[i]; if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) { return squares[a]; } } return null;}
```

Note it does not matter whether you define calculateWinner before or after the Board. Let's put it at the end so that you don't have to scroll past it every time you edit your components.

You will call calculateWinner(squares) in the Board component's handleClick function to check if a player has won. You can perform this check at the same time you check if a user has clicked a square that already has a X or and O. We'd like to return early in both cases:

```
function handleClick(i) { if (squares[i] || calculateWinner(squares)) { return; } const nextSquares = squares.slice();
//...}
```

To let the players know when the game is over, you can display text such as "Winner: X" or "Winner: O". To do that you'll add a status section to the Board component. The status will display the winner if the game is over and if the game is ongoing you'll display which player's turn is next:

```
export default function Board() { // ... const winner = calculateWinner(squares); let status; if (winner) { status =
"Winner: " + winner; } else { status = "Next player: " + (xIsNext ? "X" : "O"); } return ( <> <div
className="status">{status}</div> <div className="board-row"> // ... )}
```

Congratulations! You now have a working tic-tac-toe game. And you've just learned the basics of React too. So you are the real winner here. Here is what the code should look like:

```
App.jsApp.js ResetFor import { useState } from 'react';
```

```
function Square({value, onClick}) {
  return (
    <button className="square" onClick={onClick}>
      {value}
    </button>
  );
}
```

```
</button>
);

}

export default function Board() {
  const [xIsNext, setXIsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
    setSquares(nextSquares);
    setXIsNext(!xIsNext);
  }

  const winner = calculateWinner(squares);
  let status;
  if (winner) {
    status = `Winner: ${winner}`;
  } else {
    status = `Next player: ${xIsNext ? 'X' : 'O'}`;
  }

  return (
    <>
    <div className="status">{status}</div>
    <div className="board-row">
```

```

<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>

<div className="board-row">
  <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
  <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
  <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>

<div className="board-row">
  <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
  <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
  <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>

</>
);

}

```

```

function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
}

```

```
}

return null;

}
```

Show more

Adding time travel

As a final exercise, let's make it possible to "go back in time" to the previous moves in the game.

Storing a history of moves

If you mutated the squares array, implementing time travel would be very difficult.

However, you used slice() to create a new copy of the squares array after every move, and treated it as immutable. This will allow you to store every past version of the squares array, and navigate between the turns that have already happened.

You'll store the past squares arrays in another array called history, which you'll store as a new state variable. The history array represents all board states, from the first to the last move, and has a shape like this:

```
[ // Before first move [null, null, null, null, null, null, null, null, null], // After first move [null, null, null, null, 'X', null, null, null, null], // After second move [null, null, null, null, 'X', null, null, null, 'O'], // ...]
```

Lifting state up, again

You will now write a new top-level component called Game to display a list of past moves. That's where you will place the history state that contains the entire game history.

Placing the history state into the Game component will let you remove the squares state from its child Board component. Just like you "lifted state up" from the Square component into the Board component, you will now lift it up from the Board into the top-level Game component. This gives the Game component full control over the Board's data and lets it instruct the Board to render previous turns from the history.

First, add a Game component with export default. Have it render the Board component and some markup:

```
function Board() { // ...}export default function Game() { return (  <div className="game">    <div className="game-board">      <Board />    </div>    <div className="game-info">      <ol>{/*TODO*!}</ol>    </div>  </div> );}
```

Note that you are removing the export default keywords before the function Board() { declaration and adding them before the function Game() { declaration. This tells your index.js file to use the Game component as the top-level component instead of your Board component. The additional divs returned by the Game component are making room for the game information you'll add to the board later.

Add some state to the Game component to track which player is next and the history of moves:

```
export default function Game() { const [xIsNext, setXIsNext] = useState(true); const [history, setHistory] = useState([Array(9).fill(null)]); // ...
```

Notice how [Array(9).fill(null)] is an array with a single item, which itself is an array of 9 nulls.

To render the squares for the current move, you'll want to read the last squares array from the history. You don't need useState for this—you already have enough information to calculate it during rendering:

```
export default function Game() { const [xIsNext, setXIsNext] = useState(true); const [history, setHistory] = useState([Array(9).fill(null)]); const currentSquares = history[history.length - 1]; // ...
```

Next, create a handlePlay function inside the Game component that will be called by the Board component to update the game. Pass xIsNext, currentSquares and handlePlay as props to the Board component:

```
export default function Game() { const [xIsNext, setXIsNext] = useState(true); const [history, setHistory] = useState([Array(9).fill(null)]); const currentSquares = history[history.length - 1]; function handlePlay(nextSquares) { // TODO } return ( <div className="game">   <div className="game-board">     <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />   //... )}
```

Let's make the Board component fully controlled by the props it receives. Change the Board component to take three props: xIsNext, squares, and a new onPlay function that Board can call with the updated squares array when a player makes a move. Next, remove the first two lines of the Board function that call useState:

```
function Board({ xIsNext, squares, onPlay }) { function handleClick(i) { //... } // ...}
```

Now replace the setSquares and setXIsNext calls in handleClick in the Board component with a single call to your new onPlay function so the Game component can update the Board when the user clicks a square:

```
function Board({ xIsNext, squares, onPlay }) { function handleClick(i) { if (calculateWinner(squares) || squares[i]) { return; } const nextSquares = squares.slice(); if (xIsNext) { nextSquares[i] = "X"; } else { nextSquares[i] = "O"; } onPlay(nextSquares); } //...}
```

The Board component is fully controlled by the props passed to it by the Game component. You need to implement the handlePlay function in the Game component to get the game working again.

What should handlePlay do when called? Remember that Board used to call setSquares with an updated array; now it passes the updated squares array to onPlay.

The handlePlay function needs to update Game's state to trigger a re-render, but you don't have a setSquares function that you can call any more—you're now using the history state variable to store this information. You'll want to update history by appending the updated squares array as a new history entry. You also want to toggle xIsNext, just as Board used to do:

```
export default function Game() { //... function handlePlay(nextSquares) { setHistory([...history, nextSquares]); setXIsNext(!xIsNext); } //...}
```

Here, [...history, nextSquares] creates a new array that contains all the items in history, followed by nextSquares. (You can read the ...history spread syntax as “enumerate all the items in history”.)

For example, if history is [[null,null,null], ["X",null,null]] and nextSquares is ["X",null,"O"], then the new [...history, nextSquares] array will be [[null,null,null], ["X",null,null], ["X",null,"O"]].

At this point, you've moved the state to live in the Game component, and the UI should be fully working, just as it was before the refactor. Here is what the code should look like at this point:

```
App.js
import { useState } from 'react';
```

```
function Square({ value, onSquareClick }) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}
```

```
function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
    onPlay(nextSquares);
  }

  const winner = calculateWinner(squares);
  let status;
  if (winner) {
    status = `Winner: ${winner}`;
  } else {
    status = `Next player: ${xIsNext ? 'X' : 'O'}`;
  }

  return (
    <>
    <div className="status">{status}</div>
    <div className="board-row">
      <Square value={squares[0]} onClick={() => handleClick(0)} />
      <Square value={squares[1]} onClick={() => handleClick(1)} />
      <Square value={squares[2]} onClick={() => handleClick(2)} />
    </div>
    <div className="board-row">
      <Square value={squares[3]} onClick={() => handleClick(3)} />
      <Square value={squares[4]} onClick={() => handleClick(4)} />
      <Square value={squares[5]} onClick={() => handleClick(5)} />
    </div>
  )
}
```

```
</div>

<div className="board-row">
  <Square value={squares[6]} onClick={() => handleClick(6)} />
  <Square value={squares[7]} onClick={() => handleClick(7)} />
  <Square value={squares[8]} onClick={() => handleClick(8)} />
</div>
</>
);

}

}

export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];

  function handlePlay(nextSquares) {
    setHistory([...history, nextSquares]);
    setXIsNext(!xIsNext);
  }

  return (
    <div className="game">
      <div className="game-board">
        <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
      </div>
      <div className="game-info">
        <ol>{/*TODO*!}</ol>
      </div>
    </div>
  );
}

}

function calculateWinner(squares) {
  const lines = [

```

```

[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[0, 3, 6],
[1, 4, 7],
[2, 5, 8],
[0, 4, 8],
[2, 4, 6],
};

for (let i = 0; i < lines.length; i++) {
  const [a, b, c] = lines[i];
  if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
    return squares[a];
  }
}
return null;
}

```

Show more

Showing the past moves

Since you are recording the tic-tac-toe game's history, you can now display a list of past moves to the player.

React elements like `<button>` are regular JavaScript objects; you can pass them around in your application. To render multiple items in React, you can use an array of React elements.

You already have an array of history moves in state, so now you need to transform it to an array of React elements. In JavaScript, to transform one array into another, you can use the array map method:

```
[1, 2, 3].map((x) => x * 2) // [2, 4, 6]
```

You'll use map to transform your history of moves into React elements representing buttons on the screen, and display a list of buttons to "jump" to past moves. Let's map over the history in the Game component:

```
export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];
  function handlePlay(nextSquares) {
    setHistory([...history, nextSquares]);
    setXIsNext(!xIsNext);
  }
  function jumpTo(nextMove) {
    // TODO
  }
  const moves = history.map((squares, move) => {
    let description;
    if (move > 0) {
      description = `Go to move #${move}`;
    } else {
      description = 'Go to game start';
    }
    return (
      <li>
        <button onClick={() => jumpTo(move)}>{description}</button>
      </li>
    );
  });
  return (
    <div className="game">
      <div className="game-board">
        <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
      </div>
      <div className="game-info">
        <ol>{moves}</ol>
      </div>
    </div>
  );
}
```

You can see what your code should look like below. Note that you should see an error in the developer tools console that says:

ConsoleWarning: Each child in an array or iterator should have a unique “key” prop. Check the render method of `Game`.

You'll fix this error in the next section.

```
App.js
```

```
App.js ResetFor
import { useState } from 'react';
```

```
function Square({ value, onSquareClick }) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}
```

```
function Board({ xIsNext, squares, onPlay }) {
```

```
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
      nextSquares[i] = 'X';
    } else {
      nextSquares[i] = 'O';
    }
    onPlay(nextSquares);
  }
}
```

```
  const winner = calculateWinner(squares);
```

```
  let status;
  if (winner) {
    status = 'Winner: ' + winner;
  } else {
    status = 'Next player: ' + (xIsNext ? 'X' : 'O');
  }
}
```

```

return (
  <>
  <div className="status">{status}</div>
  <div className="board-row">
    <Square value={squares[0]} onClick={() => handleClick(0)} />
    <Square value={squares[1]} onClick={() => handleClick(1)} />
    <Square value={squares[2]} onClick={() => handleClick(2)} />
  </div>
  <div className="board-row">
    <Square value={squares[3]} onClick={() => handleClick(3)} />
    <Square value={squares[4]} onClick={() => handleClick(4)} />
    <Square value={squares[5]} onClick={() => handleClick(5)} />
  </div>
  <div className="board-row">
    <Square value={squares[6]} onClick={() => handleClick(6)} />
    <Square value={squares[7]} onClick={() => handleClick(7)} />
    <Square value={squares[8]} onClick={() => handleClick(8)} />
  </div>
</>
);
}

```

```

export default function Game() {
  const [xIsNext, setXIsNext] = useState(true);
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const currentSquares = history[history.length - 1];

```

```

function handlePlay(nextSquares) {
  setHistory([...history, nextSquares]);
  setXIsNext(!xIsNext);
}

```

```

function jumpTo(nextMove) {

```

```
// TODO
}

const moves = history.map((squares, move) => {
  let description;
  if (move > 0) {
    description = 'Go to move #' + move;
  } else {
    description = 'Go to game start';
  }
  return (
    <li>
      <button onClick={() => jumpTo(move)}>{description}</button>
    </li>
  );
});

return (
  <div className="game">
    <div className="game-board">
      <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
    </div>
    <div className="game-info">
      <ol>{moves}</ol>
    </div>
  </div>
);

function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
  ];
```

```

[0, 3, 6],
[1, 4, 7],
[2, 5, 8],
[0, 4, 8],
[2, 4, 6],
];
for (let i = 0; i < lines.length; i++) {
  const [a, b, c] = lines[i];
  if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
    return squares[a];
  }
}
return null;
}

```

Show more

As you iterate through history array inside the function you passed to map, the squares argument goes through each element of history, and the move argument goes through each array index: 0, 1, 2, (In most cases, you'd need the actual array elements, but to render a list of moves you will only need indexes.)

For each move in the tic-tac-toe game's history, you create a list item `` which contains a button `<button>`. The button has an `onClick` handler which calls a function called `jumpTo` (that you haven't implemented yet).

For now, you should see a list of the moves that occurred in the game and an error in the developer tools console. Let's discuss what the "key" error means.

Picking a key

When you render a list, React stores some information about each rendered list item. When you update a list, React needs to determine what has changed. You could have added, removed, re-arranged, or updated the list's items.

Imagine transitioning from

`Alexa: 7 tasks leftBen: 5 tasks left`

to

`Ben: 9 tasks leftClaudia: 8 tasks leftAlexa: 5 tasks left`

In addition to the updated counts, a human reading this would probably say that you swapped Alexa and Ben's ordering and inserted Claudia between Alexa and Ben. However, React is a computer program and does not know what you intended, so you need to specify a `key` property for each list item to differentiate each list item from its siblings. If your data was from a database, Alexa, Ben, and Claudia's database IDs could be used as keys.

`<li key={user.id}> {user.name}: {user.taskCount} tasks left`

When a list is re-rendered, React takes each list item's key and searches the previous list's items for a matching key. If the current list has a key that didn't exist before, React creates a component. If the current list is missing a key that

existed in the previous list, React destroys the previous component. If two keys match, the corresponding component is moved.

Keys tell React about the identity of each component, which allows React to maintain state between re-renders. If a component's key changes, the component will be destroyed and re-created with a new state.

key is a special and reserved property in React. When an element is created, React extracts the key property and stores the key directly on the returned element. Even though key may look like it is passed as props, React automatically uses key to decide which components to update. There's no way for a component to ask what key its parent specified.

It's strongly recommended that you assign proper keys whenever you build dynamic lists. If you don't have an appropriate key, you may want to consider restructuring your data so that you do.

If no key is specified, React will report an error and use the array index as a key by default. Using the array index as a key is problematic when trying to re-order a list's items or inserting/removing list items. Explicitly passing key={i} silences the error but has the same problems as array indices and is not recommended in most cases.

Keys do not need to be globally unique; they only need to be unique between components and their siblings.

Implementing time travel

In the tic-tac-toe game's history, each past move has a unique ID associated with it: it's the sequential number of the move. Moves will never be re-ordered, deleted, or inserted in the middle, so it's safe to use the move index as a key.

In the Game function, you can add the key as <li key={move}>, and if you reload the rendered game, React's "key" error should disappear:

```
const moves = history.map((squares, move) => { //... return ( <li key={move}> <button onClick={()=> jumpTo(move)}>{description}</button> </li> );});
```

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

```
function Square({ value, onSquareClick }) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}
```

```
function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
```

```
nextSquares[i] = 'X';
} else {
  nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

const winner = calculateWinner(squares);

let status;
if (winner) {
  status = 'Winner: ' + winner;
} else {
  status = 'Next player: ' + (xIsNext ? 'X' : 'O');
}

return (
  <>
  <div className="status">{status}</div>
  <div className="board-row">
    <Square value={squares[0]} onClick={() => handleClick(0)} />
    <Square value={squares[1]} onClick={() => handleClick(1)} />
    <Square value={squares[2]} onClick={() => handleClick(2)} />
  </div>
  <div className="board-row">
    <Square value={squares[3]} onClick={() => handleClick(3)} />
    <Square value={squares[4]} onClick={() => handleClick(4)} />
    <Square value={squares[5]} onClick={() => handleClick(5)} />
  </div>
  <div className="board-row">
    <Square value={squares[6]} onClick={() => handleClick(6)} />
    <Square value={squares[7]} onClick={() => handleClick(7)} />
    <Square value={squares[8]} onClick={() => handleClick(8)} />
  </div>
</>
```

```
);

}

export default function Game() {

  const [xIsNext, setXIsNext] = useState(true);

  const [history, setHistory] = useState([Array(9).fill(null)]);

  const currentSquares = history[history.length - 1];

  function handlePlay(nextSquares) {

    setHistory([...history, nextSquares]);
    setXIsNext(!xIsNext);
  }

  function jumpTo(nextMove) {
    // TODO
  }

  const moves = history.map((squares, move) => {

    let description;

    if (move > 0) {

      description = 'Go to move #' + move;
    } else {

      description = 'Go to game start';
    }

    return (
      <li key={move}>
        <button onClick={() => jumpTo(move)}>{description}</button>
      </li>
    );
  });

  return (
    <div className="game">
      <div className="game-board">
```

```

<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
<div className="game-info">
  <ol>{moves}</ol>
</div>
</div>
);
}

```

```

function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}

```

Show more

Before you can implement `jumpTo`, you need the `Game` component to keep track of which step the user is currently viewing. To do this, define a new state variable called `currentMove`, defaulting to 0:

```

export default function Game() { const [xIsNext, setXIsNext] = useState(true); const [history, setHistory] =
useState([Array(9).fill(null)]); const [currentMove, setCurrentMove] = useState(0); const currentSquares =
history[history.length - 1]; //...}

```

Next, update the jumpTo function inside Game to update that currentMove. You'll also set xIsNext to true if the number that you're changing currentMove to is even.

```
export default function Game() { // ... function jumpTo(nextMove) { setCurrentMove(nextMove); setXIsNext(nextMove % 2 === 0); } //...}
```

You will now make two changes to the Game's handlePlay function which is called when you click on a square.

If you “go back in time” and then make a new move from that point, you only want to keep the history up to that point. Instead of adding nextSquares after all items (... spread syntax) in history, you'll add it after all items in history.slice(0, currentMove + 1) so that you're only keeping that portion of the old history.

Each time a move is made, you need to update currentMove to point to the latest history entry.

```
function handlePlay(nextSquares) { const nextHistory = [...history.slice(0, currentMove + 1), nextSquares]; setHistory(nextHistory); setCurrentMove(nextHistory.length - 1); setXIsNext(!xIsNext);}
```

Finally, you will modify the Game component to render the currently selected move, instead of always rendering the final move:

```
export default function Game() { const [xIsNext, setXIsNext] = useState(true); const [history, setHistory] = useState([Array(9).fill(null)]); const [currentMove, setCurrentMove] = useState(0); const currentSquares = history[currentMove]; // ...}
```

If you click on any step in the game's history, the tic-tac-toe board should immediately update to show what the board looked like after that step occurred.

```
App.js
```

```
App.js
import { useState } from 'react';

function Square({value, onSquareClick}) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}

}
```

```
function Board({ xIsNext, squares, onPlay }) {
```

```
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    const nextSquares = squares.slice();
    if (xIsNext) {
```

```
nextSquares[i] = 'X';
} else {
  nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

const winner = calculateWinner(squares);

let status;
if (winner) {
  status = 'Winner: ' + winner;
} else {
  status = 'Next player: ' + (xIsNext ? 'X' : 'O');
}

return (
  <>
  <div className="status">{status}</div>
  <div className="board-row">
    <Square value={squares[0]} onClick={() => handleClick(0)} />
    <Square value={squares[1]} onClick={() => handleClick(1)} />
    <Square value={squares[2]} onClick={() => handleClick(2)} />
  </div>
  <div className="board-row">
    <Square value={squares[3]} onClick={() => handleClick(3)} />
    <Square value={squares[4]} onClick={() => handleClick(4)} />
    <Square value={squares[5]} onClick={() => handleClick(5)} />
  </div>
  <div className="board-row">
    <Square value={squares[6]} onClick={() => handleClick(6)} />
    <Square value={squares[7]} onClick={() => handleClick(7)} />
    <Square value={squares[8]} onClick={() => handleClick(8)} />
  </div>
</>
```

```
);

}

export default function Game() {

  const [xIsNext, setXIsNext] = useState(true);

  const [history, setHistory] = useState([Array(9).fill(null)]);

  const [currentMove, setCurrentMove] = useState(0);

  const currentSquares = history[currentMove];

  function handlePlay(nextSquares) {

    const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];

    setHistory(nextHistory);

    setCurrentMove(nextHistory.length - 1);

    setXIsNext(!xIsNext);

  }

  function jumpTo(nextMove) {

    setCurrentMove(nextMove);

    setXIsNext(nextMove % 2 === 0);

  }

  const moves = history.map((squares, move) => {

    let description;

    if (move > 0) {

      description = 'Go to move #' + move;

    } else {

      description = 'Go to game start';

    }

    return (
      <li key={move}>
        <button onClick={() => jumpTo(move)}>{description}</button>
      </li>
    );
  });
}
```

```
return (

<div className="game">

  <div className="game-board">

    <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />

  </div>

  <div className="game-info">

    <ol>{moves}</ol>

  </div>

</div>

);

}
```

```
function calculateWinner(squares) {

  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];

  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}
```

Show more

Final cleanup

If you look at the code very closely, you may notice that `xIsNext === true` when `currentMove` is even and `xIsNext === false` when `currentMove` is odd. In other words, if you know the value of `currentMove`, then you can always figure out what `xIsNext` should be.

There's no reason for you to store both of these in state. In fact, always try to avoid redundant state. Simplifying what you store in state reduces bugs and makes your code easier to understand. Change `Game` so that it doesn't store `xIsNext` as a separate state variable and instead figures it out based on the `currentMove`:

```
export default function Game() { const [history, setHistory] = useState([Array(9).fill(null)]); const [currentMove, setCurrentMove] = useState(0); const xIsNext = currentMove % 2 === 0; const currentSquares = history[currentMove]; function handlePlay(nextSquares) { const nextHistory = [...history.slice(0, currentMove + 1), nextSquares]; setHistory(nextHistory); setCurrentMove(nextHistory.length - 1); } function jumpTo(nextMove) { setCurrentMove(nextMove); } // ...}
```

You no longer need the `xIsNext` state declaration or the calls to `setXIsNext`. Now, there's no chance for `xIsNext` to get out of sync with `currentMove`, even if you make a mistake while coding the components.

Wrapping up

Congratulations! You've created a tic-tac-toe game that:

Lets you play tic-tac-toe,

Indicates when a player has won the game,

Stores a game's history as a game progresses,

Allows players to review a game's history and see previous versions of a game's board.

Nice work! We hope you now feel like you have a decent grasp of how React works.

Check out the final result here:

```
App.js
```

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

function Square({ value, onSquareClick }) {
  return (
    <button className="square" onClick={onSquareClick}>
      {value}
    </button>
  );
}


```

```
function Board({ xIsNext, squares, onPlay }) {
  function handleClick(i) {
    if (calculateWinner(squares) || squares[i]) {
```

```
return;

}

const nextSquares = squares.slice();

if (xIsNext) {
    nextSquares[i] = 'X';
} else {
    nextSquares[i] = 'O';
}

onPlay(nextSquares);

}

const winner = calculateWinner(squares);

let status;

if (winner) {
    status = 'Winner: ' + winner;
} else {
    status = 'Next player: ' + (xIsNext ? 'X' : 'O');
}

return (
    <>
    <div className="status">{status}</div>
    <div className="board-row">
        <Square value={squares[0]} onClick={() => handleClick(0)} />
        <Square value={squares[1]} onClick={() => handleClick(1)} />
        <Square value={squares[2]} onClick={() => handleClick(2)} />
    </div>
    <div className="board-row">
        <Square value={squares[3]} onClick={() => handleClick(3)} />
        <Square value={squares[4]} onClick={() => handleClick(4)} />
        <Square value={squares[5]} onClick={() => handleClick(5)} />
    </div>
    <div className="board-row">
        <Square value={squares[6]} onClick={() => handleClick(6)} />
    </div>
)
```

```
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);

}

export default function Game() {
  const [history, setHistory] = useState([Array(9).fill(null)]);
  const [currentMove, setCurrentMove] = useState(0);
  const xIsNext = currentMove % 2 === 0;
  const currentSquares = history[currentMove];

  function handlePlay(nextSquares) {
    const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
    setHistory(nextHistory);
    setCurrentMove(nextHistory.length - 1);
  }

  function jumpTo(nextMove) {
    setCurrentMove(nextMove);
  }

  const moves = history.map((squares, move) => {
    let description;
    if (move > 0) {
      description = 'Go to move #' + move;
    } else {
      description = 'Go to game start';
    }
    return (
      <li key={move}>
        <button onClick={() => jumpTo(move)}>{description}</button>
      </li>
    );
  });
}
```

```
);

});

return (
<div className="game">
  <div className="game-board">
    <Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
  </div>
  <div className="game-info">
    <ol>{moves}</ol>
  </div>
</div>
);
```

```
}
```



```
function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}
```

Show more

If you have extra time or want to practice your new React skills, here are some ideas for improvements that you could make to the tic-tac-toe game, listed in order of increasing difficulty:

For the current move only, show “You are at move #...” instead of a button.

Rewrite Board to use two loops to make the squares instead of hardcoding them.

Add a toggle button that lets you sort the moves in either ascending or descending order.

When someone wins, highlight the three squares that caused the win (and when no one wins, display a message about the result being a draw).

Display the location for each move in the format (row, col) in the move history list.

Throughout this tutorial, you’ve touched on React concepts including elements, components, props, and state. Now that you’ve seen how these concepts work when building a game, check out Thinking in React to see how the same React concepts work when building an app’s UI. Previous Quick Start Next Thinking in React ©2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview What are you building? Setup for the tutorial Overview Inspecting the starter code Building the board Passing data through props Making an interactive component React Developer Tools Completing the game Lifting state up Why immutability is important Taking turns Declaring a winner Adding time travel Storing a history of moves Lifting state up, again Showing the past moves Picking a key Implementing time travel Final cleanup Wrapping up Thinking in React – React React v18.3.1 Search Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component’s Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Quick Start Thinking in React React can change how you think about the designs you look at and the apps you build. When you build a user interface with React, you will first break it apart into pieces called components. Then, you will describe the different visual states for each of your components. Finally, you will connect your components together so that the data flows through them. In this tutorial, we’ll guide you through the thought process of building a searchable product data table with React.

Start with the mockup

Imagine that you already have a JSON API and a mockup from a designer.

The JSON API returns some data that looks like this:

```
[ { category: "Fruits", price: "$1", stocked: true, name: "Apple" }, { category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit" }, { category: "Fruits", price: "$2", stocked: false, name: "Passionfruit" }, { category:
```

```
"Vegetables", price: "$2", stocked: true, name: "Spinach" }, { category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin" }, { category: "Vegetables", price: "$1", stocked: true, name: "Peas" }]
```

The mockup looks like this:

To implement a UI in React, you will usually follow the same five steps.

Step 1: Break the UI into a component hierarchy

Start by drawing boxes around every component and subcomponent in the mockup and naming them. If you work with a designer, they may have already named these components in their design tool. Ask them!

Depending on your background, you can think about splitting up a design into components in different ways:

Programming—use the same techniques for deciding if you should create a new function or object. One such technique is the single responsibility principle, that is, a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.

CSS—consider what you would make class selectors for. (However, components are a bit less granular.)

Design—consider how you would organize the design's layers.

If your JSON is well-structured, you'll often find that it naturally maps to the component structure of your UI. That's because UI and data models often have the same information architecture—that is, the same shape. Separate your UI into components, where each component matches one piece of your data model.

There are five components on this screen:

FilterableProductTable (grey) contains the entire app.

SearchBar (blue) receives the user input.

ProductTable (lavender) displays and filters the list according to the user input.

ProductCategoryRow (green) displays a heading for each category.

ProductRow (yellow) displays a row for each product.

If you look at ProductTable (lavender), you'll see that the table header (containing the “Name” and “Price” labels) isn't its own component. This is a matter of preference, and you could go either way. For this example, it is a part of ProductTable because it appears inside the ProductTable's list. However, if this header grows to be complex (e.g., if you add sorting), you can move it into its own ProductTableHeader component.

Now that you've identified the components in the mockup, arrange them into a hierarchy. Components that appear within another component in the mockup should appear as a child in the hierarchy:

FilterableProductTable

 SearchBar

 ProductTable

 ProductCategoryRow

 ProductRow

Step 2: Build a static version in React

Now that you have your component hierarchy, it's time to implement your app. The most straightforward approach is to build a version that renders the UI from your data model without adding any interactivity... yet! It's often easier to build the static version first and add interactivity later. Building a static version requires a lot of typing and no thinking, but adding interactivity requires a lot of thinking and not a lot of typing.

To build a static version of your app that renders your data model, you'll want to build components that reuse other components and pass data using props. Props are a way of passing data from parent to child. (If you're familiar with the concept of state, don't use state at all to build this static version. State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, you don't need it.)

You can either build "top down" by starting with building the components higher up in the hierarchy (like `FilterableProductTable`) or "bottom up" by working from components lower down (like `ProductRow`). In simpler examples, it's usually easier to go top-down, and on larger projects, it's easier to go bottom-up.

```
App.js
```

```
function ResetFork() {
  return (
    <table>
      <thead>
        <tr>
          <th>Category</th>
          <th>Product Name</th>
          <th>Price</th>
        </tr>
      </thead>
      <tbody>
        {products.map((product) => (
          <tr>
            <td>{product.category}</td>
            <td>{product.name}</td>
            <td>${product.price}</td>
          </tr>
        ))}
      </tbody>
    </table>
  )
}
```

```
<ProductCategoryRow>
```

```
{category}
```

```
</ProductCategoryRow>
```

```
<ProductRow>
```

```
{product}
```

```
</ProductRow>
```

```
</tbody>
```

```
<tbody>
```

```
const name = product.stocked ? product.name :
```

```
  <span style={{ color: 'red' }}>
```

```
    {product.name}
```

```
  </span>;
```

```
<tbody>
```

```
<tr>
```

```
  <td>{name}</td>
```

```
  <td>{product.price}</td>
```

```
</tr>
```

```
<tbody>
```

```
<tbody>
```

```
const rows = [];
```

```
let lastCategory = null;
```

```
products.forEach((product) => {
  if (product.category !== lastCategory) {
    rows.push(
      <ProductCategoryRow
        category={product.category}
        key={product.category} />
    );
  }
  rows.push(
    <ProductRow
      product={product}
      key={product.name} />
  );
  lastCategory = product.category;
});
```

```
return (
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>{rows}</tbody>
</table>
);
```

```
}
```

```
function SearchBar() {
  return (
    <form>
      <input type="text" placeholder="Search..." />
      <label>
```

```
<input type="checkbox" />
{' '}
Only show products in stock
</label>
</form>
);
}

function FilterableProductTable({ products }) {
return (
<div>
<SearchBar />
<ProductTable products={products} />
</div>
);
}

const PRODUCTS = [
{category: "Fruits", price: "$1", stocked: true, name: "Apple"},
{category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit"},
{category: "Fruits", price: "$2", stocked: false, name: "Passionfruit"},
{category: "Vegetables", price: "$2", stocked: true, name: "Spinach"},
{category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin"},
{category: "Vegetables", price: "$1", stocked: true, name: "Peas"}
];
}

export default function App() {
return <FilterableProductTable products={PRODUCTS} />;
}
```

Show more

(If this code looks intimidating, go through the Quick Start first!)

After building your components, you'll have a library of reusable components that render your data model. Because this is a static app, the components will only return JSX. The component at the top of the hierarchy

(FilterableProductTable) will take your data model as a prop. This is called one-way data flow because the data flows down from the top-level component to the ones at the bottom of the tree.

Pitfall At this point, you should not be using any state values. That's for the next step!

Step 3: Find the minimal but complete representation of UI state

To make the UI interactive, you need to let users change your underlying data model. You will use state for this.

Think of state as the minimal set of changing data that your app needs to remember. The most important principle for structuring state is to keep it DRY (Don't Repeat Yourself). Figure out the absolute minimal representation of the state your application needs and compute everything else on-demand. For example, if you're building a shopping list, you can store the items as an array in state. If you want to also display the number of items in the list, don't store the number of items as another state value—instead, read the length of your array.

Now think of all of the pieces of data in this example application:

The original list of products

The search text the user has entered

The value of the checkbox

The filtered list of products

Which of these are state? Identify the ones that are not:

Does it remain unchanged over time? If so, it isn't state.

Is it passed in from a parent via props? If so, it isn't state.

Can you compute it based on existing state or props in your component? If so, it definitely isn't state!

What's left is probably state.

Let's go through them one by one again:

The original list of products is passed in as props, so it's not state.

The search text seems to be state since it changes over time and can't be computed from anything.

The value of the checkbox seems to be state since it changes over time and can't be computed from anything.

The filtered list of products isn't state because it can be computed by taking the original list of products and filtering it according to the search text and value of the checkbox.

This means only the search text and the value of the checkbox are state! Nicely done!

Deep DiveProps vs State Show Details There are two types of “model” data in React: props and state. The two are very different:

Props are like arguments you pass to a function. They let a parent component pass data to a child component and customize its appearance. For example, a Form can pass a color prop to a Button.

State is like a component's memory. It lets a component keep track of some information and change it in response to interactions. For example, a Button might keep track of isHovered state.

Props and state are different, but they work together. A parent component will often keep some information in state (so that it can change it), and pass it down to child components as their props. It's okay if the difference still feels fuzzy on the first read. It takes a bit of practice for it to really stick!

Step 4: Identify where your state should live

After identifying your app's minimal state data, you need to identify which component is responsible for changing this state, or owns the state. Remember: React uses one-way data flow, passing data down the component hierarchy from parent to child component. It may not be immediately clear which component should own what state. This can be challenging if you're new to this concept, but you can figure it out by following these steps!

For each piece of state in your application:

Identify every component that renders something based on that state.

Find their closest common parent component—a component above them all in the hierarchy.

Decide where the state should live:

Often, you can put the state directly into their common parent.

You can also put the state into some component above their common parent.

If you can't find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common parent component.

In the previous step, you found two pieces of state in this application: the search input text, and the value of the checkbox. In this example, they always appear together, so it makes sense to put them into the same place.

Now let's run through our strategy for them:

Identify components that use state:

ProductTable needs to filter the product list based on that state (search text and checkbox value).

SearchBar needs to display that state (search text and checkbox value).

Find their common parent: The first parent component both components share is FilterableProductTable.

Decide where the state lives: We'll keep the filter text and checked state values in FilterableProductTable.

So the state values will live in FilterableProductTable.

Add state to the component with the useState() Hook. Hooks are special functions that let you “hook into” React. Add two state variables at the top of FilterableProductTable and specify their initial state:

```
function FilterableProductTable({ products }) { const [filterText, setFilterText] = useState(""); const [inStockOnly, setInStockOnly] = useState(false);
```

Then, pass filterText and inStockOnly to ProductTable and SearchBar as props:

```
<div> <SearchBar filterText={filterText} inStockOnly={inStockOnly} /> <ProductTable products={products} filterText={filterText} inStockOnly={inStockOnly} /></div>
```

You can start seeing how your application will behave. Edit the filterText initial value from useState("") to useState('fruit') in the sandbox code below. You'll see both the search input text and the table update:

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

```
function FilterableProductTable({ products }) {  
  const [filterText, setFilterText] = useState("");  
  const [inStockOnly, setInStockOnly] = useState(false);  
  
  return (  
    <div>  
      <SearchBar  
        filterText={filterText}  
        inStockOnly={inStockOnly} />  
      <ProductTable  
        products={products}  
        filterText={filterText}  
        inStockOnly={inStockOnly} />  
    </div>  
  );  
}
```

```
function ProductCategoryRow({ category }) {  
  return (  
    <tr>  
      <th colSpan="2">  
        {category}  
      </th>  
    </tr>  
  );  
}
```

```
}

function ProductRow({ product }) {
  const name = product.stocked ? product.name :
    <span style={{ color: 'red' }}>
      {product.name}
    </span>;
  return (
    <tr>
      <td>{name}</td>
      <td>{product.price}</td>
    </tr>
  );
}


```

```
function ProductTable({ products, filterText, inStockOnly }) {
  const rows = [];
  let lastCategory = null;

  products.forEach((product) => {
    if (
      product.name.toLowerCase().indexOf(
        filterText.toLowerCase()
      ) === -1
    ) {
      return;
    }
    if (inStockOnly && !product.stocked) {
      return;
    }
    if (product.category !== lastCategory) {
      rows.push(
        <ProductCategoryRow>
```

```
        category={product.category}
        key={product.category} />
    );
}

rows.push(
<ProductRow
    product={product}
    key={product.name} />
);

lastCategory = product.category;
});

return (
<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>{rows}</tbody>
</table>
);
}
```

```
function SearchBar({ filterText, inStockOnly }) {
    return (
<form>
<input
    type="text"
    value={filterText}
    placeholder="Search..."/>
<label>
<input
```

```

    type="checkbox"
    checked={inStockOnly} />
  {' '}
  Only show products in stock
</label>
</form>
);
}

```

```

const PRODUCTS = [
  {category: "Fruits", price: "$1", stocked: true, name: "Apple"},
  {category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit"},
  {category: "Fruits", price: "$2", stocked: false, name: "Passionfruit"},
  {category: "Vegetables", price: "$2", stocked: true, name: "Spinach"},
  {category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin"},
  {category: "Vegetables", price: "$1", stocked: true, name: "Peas"}
];

```

```

export default function App() {
  return <FilterableProductTable products={PRODUCTS} />;
}

```

Show more

Notice that editing the form doesn't work yet. There is a console error in the sandbox above explaining why:

ConsoleYou provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field.

In the sandbox above, ProductTable and SearchBar read the filterText and inStockOnly props to render the table, the input, and the checkbox. For example, here is how SearchBar populates the input value:

```
function SearchBar({ filterText, inStockOnly }) { return ( <form> <input type="text" value={filterText} placeholder="Search..."/> ) }
```

However, you haven't added any code to respond to the user actions like typing yet. This will be your final step.

Step 5: Add inverse data flow

Currently your app renders correctly with props and state flowing down the hierarchy. But to change the state according to user input, you will need to support data flowing the other way: the form components deep in the hierarchy need to update the state in FilterableProductTable.

React makes this data flow explicit, but it requires a little more typing than two-way data binding. If you try to type or check the box in the example above, you'll see that React ignores your input. This is intentional. By writing <input

`value={filterText} />`, you've set the value prop of the input to always be equal to the filterText state passed in from FilterableProductTable. Since filterText state is never set, the input never changes.

You want to make it so whenever the user changes the form inputs, the state updates to reflect those changes. The state is owned by FilterableProductTable, so only it can call setFilterText and setInStockOnly. To let SearchBar update the FilterableProductTable's state, you need to pass these functions down to SearchBar:

```
function FilterableProductTable({ products }) { const [filterText, setFilterText] = useState(""); const [inStockOnly, setInStockOnly] = useState(false); return ( <div> <SearchBar filterText={filterText} inStockOnly={inStockOnly} onFilterTextChange={setFilterText} onInStockOnlyChange={setInStockOnly} />
```

Inside the SearchBar, you will add the onChange event handlers and set the parent state from them:

```
function SearchBar({ filterText, inStockOnly, onFilterTextChange, onInStockOnlyChange }) { return ( <form> <input type="text" value={filterText} placeholder="Search..." onChange={(e) => onFilterTextChange(e.target.value)} /> <label> <input type="checkbox" checked={inStockOnly} onChange={(e) => onInStockOnlyChange(e.target.checked)}>
```

Now the application fully works!

```
App.js
```

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

```
function FilterableProductTable({ products }) {
```

```
const [filterText, setFilterText] = useState("");
```

```
const [inStockOnly, setInStockOnly] = useState(false);
```

```
return (
```

```
<div>
```

```
 <SearchBar
```

```
 filterText={filterText}
```

```
 inStockOnly={inStockOnly}
```

```
 onFilterTextChange={setFilterText}
```

```
 onInStockOnlyChange={setInStockOnly} />
```

```
 <ProductTable
```

```
 products={products}
```

```
 filterText={filterText}
```

```
 inStockOnly={inStockOnly} />
```

```
</div>
```

```
);
```

```
}
```

```
function ProductCategoryRow({ category }) {
```

```
return (
```

```
<tr>
<th colSpan="2">
  {category}
</th>
</tr>
);

}

function ProductRow({ product }) {
  const name = product.stocked ? product.name :
    <span style={{ color: 'red' }}>
      {product.name}
    </span>;
  return (
    <tr>
      <td>{name}</td>
      <td>{product.price}</td>
    </tr>
  );
}

function ProductTable({ products, filterText, inStockOnly }) {
  const rows = [];
  let lastCategory = null;

  products.forEach((product) => {
    if (
      product.name.toLowerCase().indexOf(
        filterText.toLowerCase()
      ) === -1
    ) {
      return;
    }
  });
}
```

```
if (inStockOnly && !product.stocked) {  
    return;  
}  
  
if (product.category !== lastCategory) {  
    rows.push(  
        <ProductCategoryRow  
            category={product.category}  
            key={product.category} />  
    );  
}  
  
rows.push(  
    <ProductRow  
        product={product}  
        key={product.name} />  
);  
  
lastCategory = product.category;  
});
```

```
return (  
    <table>  
        <thead>  
            <tr>  
                <th>Name</th>  
                <th>Price</th>  
            </tr>  
        </thead>  
        <tbody>{rows}</tbody>  
    </table>  
);  
}
```

```
function SearchBar({  
    filterText,  
    inStockOnly,
```

```

onFilterTextChange,
onInStockOnlyChange
}) {
return (
<form>
<input
  type="text"
  value={filterText} placeholder="Search..." 
  onChange={(e) => onFilterTextChange(e.target.value)} />
<label>
<input
  type="checkbox"
  checked={inStockOnly}
  onChange={(e) => onInStockOnlyChange(e.target.checked)} />
{' '}
Only show products in stock
</label>
</form>
);
}

```

```

const PRODUCTS = [
  {category: "Fruits", price: "$1", stocked: true, name: "Apple"},
  {category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit"},
  {category: "Fruits", price: "$2", stocked: false, name: "Passionfruit"},
  {category: "Vegetables", price: "$2", stocked: true, name: "Spinach"},
  {category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin"},
  {category: "Vegetables", price: "$1", stocked: true, name: "Peas"}
];

```

```

export default function App() {
  return <FilterableProductTable products={PRODUCTS} />;
}

```

Show more

You can learn all about handling events and updating state in the Adding Interactivity section.

Where to go from here

This was a very brief introduction to how to think about building components and applications with React. You can start a React project right now or dive deeper on all the syntax used in this tutorial. Previous Tutorial: Tic-Tac-Toe
Next Installation ©2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Start with the mockup Step 1: Break the UI into a component hierarchy Step 2: Build a static version in React Step 3: Find the minimal but complete representation of UI state Step 4: Identify where your state should live Step 5: Add inverse data flow Where to go from here Installation – React React v18.3.1 Search Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Installation React has been designed from the start for gradual adoption. You can use as little or as much React as you need. Whether you want to get a taste of React, add some interactivity to an HTML page, or start a complex React-powered app, this section will help you get started.

In this chapter

How to start a new React project

How to add React to an existing project

How to set up your editor

How to install React Developer Tools

Try React

You don't need to install anything to play with React. Try editing this sandbox!

```
App.js
App.js Reset Fork function Greeting({ name }) {
  return <h1>Hello, {name}</h1>;
}

export default function App() {
  return <Greeting name="world" />
}
```

You can edit it directly or open it in a new tab by pressing the “Fork” button in the upper right corner.

Most pages in the React documentation contain sandboxes like this. Outside of the React documentation, there are many online sandboxes that support React: for example, CodeSandbox, StackBlitz, or CodePen.

Try React locally

To try React locally on your computer, download this HTML page. Open it in your editor and in your browser!

Start a new React project

If you want to build an app or a website fully with React, start a new React project.

Add React to an existing project

If want to try using React in your existing app or a website, add React to an existing project.

Next steps

Head to the Quick Start guide for a tour of the most important React concepts you will encounter every day. Previous Thinking in React Next Start a New React Project ©2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs

Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Try React Try React locally Start a new React project Add React to an existing project Next steps Start a New React Project – React React v18.3.1 Search Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Installation Start a New React Project If you want to build a new app or a new website fully with React, we recommend picking one of the React-powered frameworks popular in the community.

You can use React without a framework, however we've found that most apps and sites eventually build solutions to common problems such as code-splitting, routing, data fetching, and generating HTML. These problems are common to all UI libraries, not just React.

By starting with a framework, you can get started with React quickly, and avoid essentially building your own framework later.

Deep Dive Can I use React without a framework? Show Details You can definitely use React without a framework—that's how you'd use React for a part of your page. However, if you're building a new app or a site fully with React, we recommend using a framework. Here's why. Even if you don't need routing or data fetching at first, you'll likely want to add some libraries for them. As your JavaScript bundle grows with every new feature, you might have to figure out how to split code for every route individually. As your data fetching needs get more complex, you are likely to encounter server-client network waterfalls that make your app feel very slow. As your audience includes more users with poor network conditions and low-end devices, you might need to generate HTML from your components to display content early—either on the server, or during the build time. Changing your setup to run some of your code on the server or during the build can be very tricky. These problems are not React-specific. This is why Svelte has SvelteKit, Vue has Nuxt, and so on. To solve these problems on your own, you'll need to integrate your bundler with your router and with your data fetching library. It's not hard to get an initial setup working, but there are a lot of subtleties involved in making an app that loads quickly even as it grows over time. You'll want to send down the minimal amount of app code but do so in a single client-server roundtrip, in parallel with any data required for the page. You'll likely want the page to be interactive before your JavaScript code even runs, to support progressive enhancement. You may want to generate a folder of fully static HTML files for your marketing pages that can be hosted anywhere and still work with JavaScript disabled. Building these capabilities yourself takes real work. React frameworks on this page solve problems like these by default, with no extra work from your side. They

let you start very lean and then scale your app with your needs. Each React framework has a community, so finding answers to questions and upgrading tooling is easier. Frameworks also give structure to your code, helping you and others retain context and skills between different projects. Conversely, with a custom setup it's easier to get stuck on unsupported dependency versions, and you'll essentially end up creating your own framework—albeit one with no community or upgrade path (and if it's anything like the ones we've made in the past, more haphazardly designed). If your app has unusual constraints not served well by these frameworks, or you prefer to solve these problems yourself, you can roll your own custom setup with React. Grab react and react-dom from npm, set up your custom build process with a bundler like Vite or Parcel, and add other tools as you need them for routing, static generation or server-side rendering, and more.

Production-grade React frameworks

These frameworks support all the features you need to deploy and scale your app in production and are working towards supporting our full-stack architecture vision. All of the frameworks we recommend are open source with active communities for support, and can be deployed to your own server or a hosting provider. If you're a framework author interested in being included on this list, please let us know.

Next.js

Next.js' Pages Router is a full-stack React framework. It's versatile and lets you create React apps of any size—from a mostly static blog to a complex dynamic application. To create a new Next.js project, run in your terminal:

```
Terminal Copynpx create-next-app@latest
```

If you're new to Next.js, check out the learn Next.js course.

Next.js is maintained by Vercel. You can deploy a Next.js app to any Node.js or serverless hosting, or to your own server. Next.js also supports a static export which doesn't require a server.

Remix

Remix is a full-stack React framework with nested routing. It lets you break your app into nested parts that can load data in parallel and refresh in response to the user actions. To create a new Remix project, run:

```
Terminal Copynpx create-remix
```

If you're new to Remix, check out the Remix blog tutorial (short) and app tutorial (long).

Remix is maintained by Shopify. When you create a Remix project, you need to pick your deployment target. You can deploy a Remix app to any Node.js or serverless hosting by using or writing an adapter.

Gatsby

Gatsby is a React framework for fast CMS-backed websites. Its rich plugin ecosystem and its GraphQL data layer simplify integrating content, APIs, and services into one website. To create a new Gatsby project, run:

```
Terminal Copynpx create-gatsby
```

If you're new to Gatsby, check out the Gatsby tutorial.

Gatsby is maintained by Netlify. You can deploy a fully static Gatsby site to any static hosting. If you opt into using server-only features, make sure your hosting provider supports them for Gatsby.

Expo (for native apps)

Expo is a React framework that lets you create universal Android, iOS, and web apps with truly native UIs. It provides an SDK for React Native that makes the native parts easier to use. To create a new Expo project, run:

```
Terminal Copynpx create-expo-app
```

If you're new to Expo, check out the Expo tutorial.

Expo is maintained by Expo (the company). Building apps with Expo is free, and you can submit them to the Google and Apple app stores without restrictions. Expo additionally provides opt-in paid cloud services.

Bleeding-edge React frameworks

As we've explored how to continue improving React, we realized that integrating React more closely with frameworks (specifically, with routing, bundling, and server technologies) is our biggest opportunity to help React users build better apps. The Next.js team has agreed to collaborate with us in researching, developing, integrating, and testing framework-agnostic bleeding-edge React features like React Server Components.

These features are getting closer to being production-ready every day, and we've been in talks with other bundler and framework developers about integrating them. Our hope is that in a year or two, all frameworks listed on this page will have full support for these features. (If you're a framework author interested in partnering with us to experiment with these features, please let us know!)

Next.js (App Router)

Next.js's App Router is a redesign of the Next.js APIs aiming to fulfill the React team's full-stack architecture vision. It lets you fetch data in asynchronous components that run on the server or even during the build.

Next.js is maintained by Vercel. You can deploy a Next.js app to any Node.js or serverless hosting, or to your own server. Next.js also supports static export which doesn't require a server.

Deep Dive Which features make up the React team's full-stack architecture vision? Show Details Next.js's App Router bundler fully implements the official React Server Components specification. This lets you mix build-time, server-only, and interactive components in a single React tree. For example, you can write a server-only React component as an async function that reads from a database or from a file. Then you can pass data down from it to your interactive components:// This component runs *only* on the server (or during the build).async function Talks({ confId }) { // 1. You're on the server, so you can talk to your data layer. API endpoint not required. const talks = await db.Talks.findAll({ confId }); // 2. Add any amount of rendering logic. It won't make your JavaScript bundle larger. const videos = talks.map(talk => talk.video); // 3. Pass the data down to the components that will run in the browser. return <SearchableVideoList videos={videos} />; } Next.js's App Router also integrates data fetching with Suspense. This lets you specify a loading state (like a skeleton placeholder) for different parts of your user interface directly in your React tree:<Suspense fallback=<TalksLoading />> <Talks confId={conf.id} /></Suspense> Server Components and Suspense are React features rather than Next.js features. However, adopting them at the framework level requires buy-in and non-trivial implementation work. At the moment, the Next.js App Router is the most complete implementation. The React team is working with bundler developers to make these features easier to implement in the next generation of frameworks. Previous Installation Next Add React to an Existing Project ©2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Production-grade React frameworks Next.js Remix Gatsby Expo (for native apps) Bleeding-edge React frameworks Next.js (App Router) Add React to an Existing Project – React React v18.3.1 Search Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Installation Add

React to an Existing Project If you want to add some interactivity to your existing project, you don't have to rewrite it in React. Add React to your existing stack, and render interactive React components anywhere.

Note You need to install Node.js for local development. Although you can try React online or with a simple HTML page, realistically most JavaScript tooling you'll want to use for development requires Node.js.

Using React for an entire subroute of your existing website

Let's say you have an existing web app at example.com built with another server technology (like Rails), and you want to implement all routes starting with example.com/some-app/ fully with React.

Here's how we recommend to set it up:

Build the React part of your app using one of the React-based frameworks.

Specify /some-app as the base path in your framework's configuration (here's how: Next.js, Gatsby).

Configure your server or a proxy so that all requests under /some-app/ are handled by your React app.

This ensures the React part of your app can benefit from the best practices baked into those frameworks.

Many React-based frameworks are full-stack and let your React app take advantage of the server. However, you can use the same approach even if you can't or don't want to run JavaScript on the server. In that case, serve the HTML/CSS/JS export (next export output for Next.js, default for Gatsby) at /some-app/ instead.

Using React for a part of your existing page

Let's say you have an existing page built with another technology (either a server one like Rails, or a client one like Backbone), and you want to render interactive React components somewhere on that page. That's a common way to integrate React—in fact, it's how most React usage looked at Meta for many years!

You can do this in two steps:

Set up a JavaScript environment that lets you use the JSX syntax, split your code into modules with the import / export syntax, and use packages (for example, React) from the npm package registry.

Render your React components where you want to see them on the page.

The exact approach depends on your existing page setup, so let's walk through some details.

Step 1: Set up a modular JavaScript environment

A modular JavaScript environment lets you write your React components in individual files, as opposed to writing all of your code in a single file. It also lets you use all the wonderful packages published by other developers on the npm registry—including React itself! How you do this depends on your existing setup:

If your app is already split into files that use import statements, try to use the setup you already have. Check whether writing <div /> in your JS code causes a syntax error. If it causes a syntax error, you might need to transform your JavaScript code with Babel, and enable the Babel React preset to use JSX.

If your app doesn't have an existing setup for compiling JavaScript modules, set it up with Vite. The Vite community maintains many integrations with backend frameworks, including Rails, Django, and Laravel. If your backend framework is not listed, follow this guide to manually integrate Vite builds with your backend.

To check whether your setup works, run this command in your project folder:

Terminal Copynpm install react react-dom

Then add these lines of code at the top of your main JavaScript file (it might be called index.js or main.js):

```
index.jsindex.js ResetForkimport { createRoot } from 'react-dom/client';
```

```
// Clear the existing HTML content
```

```
document.body.innerHTML = '<div id="app"></div>';
```

```
// Render your React component instead
```

```
const root = createRoot(document.getElementById('app'));
```

```
root.render(<h1>Hello, world</h1>);
```

If the entire content of your page was replaced by a “Hello, world!”, everything worked! Keep reading.

NoteIntegrating a modular JavaScript environment into an existing project for the first time can feel intimidating, but it's worth it! If you get stuck, try our community resources or the Vite Chat.

Step 2: Render React components anywhere on the page

In the previous step, you put this code at the top of your main file:

```
import { createRoot } from 'react-dom/client';// Clear the existing HTML contentdocument.body.innerHTML = '<div id="app"></div>';// Render your React component insteadconst root = createRoot(document.getElementById('app'));root.render(<h1>Hello, world</h1>);
```

Of course, you don't actually want to clear the existing HTML content!

Delete this code.

Instead, you probably want to render your React components in specific places in your HTML. Open your HTML page (or the server templates that generate it) and add a unique id attribute to any tag, for example:

```
<!-- ... somewhere in your html ... --><nav id="navigation"></nav><!-- ... more html ... -->
```

This lets you find that HTML element with document.getElementById and pass it to createRoot so that you can render your own React component inside:

```
index.jsindex.htmlindex.js ResetForkimport { createRoot } from 'react-dom/client';
```

```
function NavigationBar() {
```

```
// TODO: Actually implement a navigation bar  
return <h1>Hello from React!</h1>;  
}
```

```
const domNode = document.getElementById('navigation');  
const root = createRoot(domNode);  
root.render(<NavigationBar />);
```

Notice how the original HTML content from index.html is preserved, but your own NavigationBar React component now appears inside the `<nav id="navigation">` from your HTML. Read the `createRoot` usage documentation to learn more about rendering React components inside an existing HTML page.

When you adopt React in an existing project, it's common to start with small interactive components (like buttons), and then gradually keep "moving upwards" until eventually your entire page is built with React. If you ever reach that point, we recommend migrating to a React framework right after to get the most out of React.

Using React Native in an existing native mobile app

React Native can also be integrated into existing native apps incrementally. If you have an existing native app for Android (Java or Kotlin) or iOS (Objective-C or Swift), follow this guide to add a React Native screen to it. Previous Start a New React Project Next Editor Setup ©2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Using React for an entire subroute of your existing website Using React for a part of your existing page Step 1: Set up a modular JavaScript environment Step 2: Render React components anywhere on the page Using React Native in an existing native mobile app Editor Setup – React React v18.3.1 Search ⌂ Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Installation Editor Setup A properly configured editor can make code clearer to read and faster to write. It can even help you catch bugs as you write them! If this is your first time setting up an editor or you're looking to tune up your current editor, we have a few recommendations.

You will learn

What the most popular editors are

How to format your code automatically

Your editor

VS Code is one of the most popular editors in use today. It has a large marketplace of extensions and integrates well with popular services like GitHub. Most of the features listed below can be added to VS Code as extensions as well, making it highly configurable!

Other popular text editors used in the React community include:

WebStorm is an integrated development environment designed specifically for JavaScript.

Sublime Text has support for JSX and TypeScript, syntax highlighting and autocomplete built in.

Vim is a highly configurable text editor built to make creating and changing any kind of text very efficient. It is included as “vi” with most UNIX systems and with Apple OS X.

Recommended text editor features

Some editors come with these features built in, but others might require adding an extension. Check to see what support your editor of choice provides to be sure!

Linting

Code linters find problems in your code as you write, helping you fix them early. ESLint is a popular, open source linter for JavaScript.

Install ESLint with the recommended configuration for React (be sure you have Node installed!)

Integrate ESLint in VSCode with the official extension

Make sure that you've enabled all the eslint-plugin-react-hooks rules for your project. They are essential and catch the most severe bugs early. The recommended eslint-config-react-app preset already includes them.

Formatting

The last thing you want to do when sharing your code with another contributor is get into a discussion about tabs vs spaces! Fortunately, Prettier will clean up your code by reformatting it to conform to preset, configurable rules. Run Prettier, and all your tabs will be converted to spaces—and your indentation, quotes, etc will also all be changed to conform to the configuration. In the ideal setup, Prettier will run when you save your file, quickly making these edits for you.

You can install the Prettier extension in VSCode by following these steps:

Launch VS Code

Use Quick Open (press Ctrl/Cmd+P)

Paste in ext install esbenp.prettier-vscode

Press Enter

Formatting on save

Ideally, you should format your code on every save. VS Code has settings for this!

In VS Code, press CTRL/CMD + SHIFT + P.

Type “settings”

Hit Enter

In the search bar, type “format on save”

Be sure the “format on save” option is ticked!

If your ESLint preset has formatting rules, they may conflict with Prettier. We recommend disabling all formatting rules in your ESLint preset using eslint-config-prettier so that ESLint is only used for catching logical mistakes. If you want to enforce that files are formatted before a pull request is merged, use prettier --check for your continuous integration.

PreviousAdd React to an Existing ProjectNextUsing TypeScript©2024no uwu plzuwu?Logo
by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewYour editor Recommended text editor features Linting Formatting Using TypeScript – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?Learn ReactInstallationUsing TypeScriptTypeScript is a popular way to add type definitions to JavaScript codebases. Out of the box, TypeScript supports JSX and you can get full React Web support by adding @types/react and @types/react-dom to your project.

You will learn

TypeScript with React Components

Examples of typing with Hooks

Common types from @types/react

Further learning locations

Installation

All production-grade React frameworks offer support for using TypeScript. Follow the framework specific guide for installation:

Next.js

Remix

Gatsby

Expo

Adding TypeScript to an existing React project

To install the latest version of React's type definitions:

```
Terminal Copynpm install @types/react @types/react-dom
```

The following compiler options need to be set in your tsconfig.json:

dom must be included in lib (Note: If no lib option is specified, dom is included by default).

jsx must be set to one of the valid options. preserve should suffice for most applications.

If you're publishing a library, consult the jsx documentation on what value to choose.

TypeScript with React Components

NoteEvery file containing JSX must use the .tsx file extension. This is a TypeScript-specific extension that tells TypeScript that this file contains JSX.

Writing TypeScript with React is very similar to writing JavaScript with React. The key difference when working with a component is that you can provide types for your component's props. These types can be used for correctness checking and providing inline documentation in editors.

Taking the MyButton component from the Quick Start guide, we can add a type describing the title for the button:

```
App.tsxApp.tsx ResetForkTypeScript Playgroundfunction MyButton({ title }: { title: string }) {
```

```
    return (
```

```
        <button>{title}</button>
```

```
    );
```

```
}
```

```
export default function MyApp() {
```

```
    return (
```

```
        <div>
```

```
            <h1>Welcome to my app</h1>
```

```
            <MyButton title="I'm a button" />
```

```
        </div>
```

```
    );
```

```
}
```

NoteThese sandboxes can handle TypeScript code, but they do not run the type-checker. This means you can amend the TypeScript sandboxes to learn, but you won't get any type errors or warnings. To get type-checking, you can use the TypeScript Playground or use a more fully-featured online sandbox.

This inline syntax is the simplest way to provide types for a component, though once you start to have a few fields to describe it can become unwieldy. Instead, you can use an interface or type to describe the component's props:

```
App.tsx
App.tsx ResetForkTypeScript Playground
interface MyButtonProps {
  /** The text to display inside the button */
  title: string;
  /** Whether the button can be interacted with */
  disabled: boolean;
}

function MyButton({ title, disabled }: MyButtonProps) {
  return (
    <button disabled={disabled}>{title}</button>
  );
}

export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton title="I'm a disabled button" disabled={true}/>
    </div>
  );
}
```

Show more

The type describing your component's props can be as simple or as complex as you need, though they should be an object type described with either a type or interface. You can learn about how TypeScript describes objects in Object Types but you may also be interested in using Union Types to describe a prop that can be one of a few different types and the Creating Types from Types guide for more advanced use cases.

Example Hooks

The type definitions from `@types/react` include types for the built-in Hooks, so you can use them in your components without any additional setup. They are built to take into account the code you write in your component, so you will get inferred types a lot of the time and ideally do not need to handle the minutiae of providing the types.

However, we can look at a few examples of how to provide types for Hooks.

useState

The `useState` Hook will re-use the value passed in as the initial state to determine what the type of the value should be. For example:

```
// Infer the type as "boolean"const [enabled, setEnabled] = useState(false);
```

This will assign the type of boolean to enabled, and setEnabled will be a function accepting either a boolean argument, or a function that returns a boolean. If you want to explicitly provide a type for the state, you can do so by providing a type argument to the useState call:

```
// Explicitly set the type to "boolean"const [enabled, setEnabled] = useState<boolean>(false);
```

This isn't very useful in this case, but a common case where you may want to provide a type is when you have a union type. For example, status here can be one of a few different strings:

```
type Status = "idle" | "loading" | "success" | "error";const [status, setStatus] = useState<Status>("idle");
```

Or, as recommended in Principles for structuring state, you can group related state as an object and describe the different possibilities via object types:

```
type RequestState = | { status: 'idle' } | { status: 'loading' } | { status: 'success', data: any } | { status: 'error', error: Error };const [requestState, setRequestState] = useState<RequestState>({ status: 'idle' });
```

useReducer

The useReducer Hook is a more complex Hook that takes a reducer function and an initial state. The types for the reducer function are inferred from the initial state. You can optionally provide a type argument to the useReducer call to provide a type for the state, but it is often better to set the type on the initial state instead:

```
App.tsxApp.tsx ResetForkTypeScript Playgroundimport {useReducer} from 'react';
```

```
interface State {
```

```
  count: number
```

```
};
```

```
type CounterAction =
```

```
  | { type: "reset" }
```

```
  | { type: "setCount"; value: State["count"] }
```

```
const initialState: State = { count: 0 };
```

```
function stateReducer(state: State, action: CounterAction): State {
```

```
  switch (action.type) {
```

```
    case "reset":
```

```
      return initialState;
```

```
    case "setCount":
```

```
      return { ...state, count: action.value };
```

```
    default:
```

```
      throw new Error("Unknown action");
```

```
}
```

```
}
```

```
export default function App() {
  const [state, dispatch] = useReducer(stateReducer, initialState);

  const addFive = () => dispatch({ type: "setCount", value: state.count + 5 });
  const reset = () => dispatch({ type: "reset" });

  return (
    <div>
      <h1>Welcome to my counter</h1>

      <p>Count: {state.count}</p>
      <button onClick={addFive}>Add 5</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
```

Show more

We are using TypeScript in a few key places:

interface State describes the shape of the reducer's state.

type CounterAction describes the different actions which can be dispatched to the reducer.

const initialState: State provides a type for the initial state, and also the type which is used by useReducer by default.

stateReducer(state: State, action: CounterAction): State sets the types for the reducer function's arguments and return value.

A more explicit alternative to setting the type on initialState is to provide a type argument to useReducer:

```
import { stateReducer, State } from './your-reducer-implementation';
const initialState = { count: 0 };
export default function App() {
  const [state, dispatch] = useReducer<State>(stateReducer, initialState);
```

useContext

The useContext Hook is a technique for passing data down the component tree without having to pass props through components. It is used by creating a provider component and often by creating a Hook to consume the value in a child component.

The type of the value provided by the context is inferred from the value passed to the createContext call:

```
App.tsxApp.tsx ResetForkTypeScript Playgroundimport { createContext, useContext, useState } from 'react';

type Theme = "light" | "dark" | "system";

const ThemeContext = createContext<Theme>("system");

const useGetTheme = () => useContext(ThemeContext);

export default function MyApp() {
  const [theme, setTheme] = useState<Theme>'light');

  return (
    <ThemeContext.Provider value={theme}>
      <MyComponent />
    </ThemeContext.Provider>
  )
}

function MyComponent() {
  const theme = useGetTheme();

  return (
    <div>
      <p>Current theme: {theme}</p>
    </div>
  )
}
```

Show more

This technique works when you have a default value which makes sense - but there are occasionally cases when you do not, and in those cases null can feel reasonable as a default value. However, to allow the type-system to understand your code, you need to explicitly set ContextShape | null on the createContext.

This causes the issue that you need to eliminate the | null in the type for context consumers. Our recommendation is to have the Hook do a runtime check for it's existence and throw an error when not present:

```
import { createContext, useContext, useState, useMemo } from 'react';// This is a simpler example, but you can imagine a more complex object heretype ComplexObject = { kind: string};// The context is created with `| null` in the type, to accurately reflect the default value.const Context = createContext<ComplexObject | null>(null); // The `|
```

```
null` will be removed via the check in the Hook.  
const useGetComplexObject = () => { const object =  
useContext(Context); if (!object) { throw new Error("useGetComplexObject must be used within a Provider") }  
return object;}  
export default function MyApp() { const object = useMemo(() => ({ kind: "complex" }), []); return (  
<Context.Provider value={object}> <MyComponent /> </Context.Provider> )}  
function MyComponent() { const object = useGetComplexObject(); return ( <div> <p>Current object: {object.kind}</p> </div> )}
```

useMemo

The useMemo Hooks will create/re-access a memorized value from a function call, re-running the function only when dependencies passed as the 2nd parameter are changed. The result of calling the Hook is inferred from the return value from the function in the first parameter. You can be more explicit by providing a type argument to the Hook.

```
// The type of visibleTodos is inferred from the return value of filterTodos  
const visibleTodos = useMemo(() =>  
filterTodos(todos, tab), [todos, tab]);
```

useCallback

The useCallback provide a stable reference to a function as long as the dependencies passed into the second parameter are the same. Like useMemo, the function's type is inferred from the return value of the function in the first parameter, and you can be more explicit by providing a type argument to the Hook.

```
const handleClick = useCallback(() => { // ...}, [todos]);
```

When working in TypeScript strict mode useCallback requires adding types for the parameters in your callback. This is because the type of the callback is inferred from the return value of the function, and without parameters the type cannot be fully understood.

Depending on your code-style preferences, you could use the *EventHandler functions from the React types to provide the type for the event handler at the same time as defining the callback:

```
import { useState, useCallback } from 'react';  
export default function Form() { const [value, setValue] =  
useState("Change me"); const handleChange =  
useCallback<React.ChangeEventHandler<HTMLInputElement>>((event) => { setValue(event.currentTarget.value);  
}, [setValue]) return ( <> <input value={value} onChange={handleChange} /> <p>Value: {value}</p> </> );}
```

Useful Types

There is quite an expansive set of types which come from the @types/react package, it is worth a read when you feel comfortable with how React and TypeScript interact. You can find them in React's folder in DefinitelyTyped. We will cover a few of the more common types here.

DOM Events

When working with DOM events in React, the type of the event can often be inferred from the event handler. However, when you want to extract a function to be passed to an event handler, you will need to explicitly set the type of the event.

```
App.tsx  
App.tsx  
ResetForkTypeScript Playground  
import { useState } from 'react';
```

```
export default function Form() {  
const [value, setValue] = useState("Change me");  
  
function handleChange(event: React.ChangeEvent<HTMLInputElement>) {  
setValue(event.currentTarget.value);
```

```
}

return (
  <>
  <input value={value} onChange={handleChange} />
  <p>Value: {value}</p>
</>
);
}
```

Show more

There are many types of events provided in the React types - the full list can be found here which is based on the most popular events from the DOM.

When determining the type you are looking for you can first look at the hover information for the event handler you are using, which will show the type of the event.

If you need to use an event that is not included in this list, you can use the `React.SyntheticEvent` type, which is the base type for all events.

Children

There are two common paths to describing the children of a component. The first is to use the `React.ReactNode` type, which is a union of all the possible types that can be passed as children in JSX:

```
interface ModalRendererProps { title: string; children: React.ReactNode;}
```

This is a very broad definition of children. The second is to use the `React.ReactElement` type, which is only JSX elements and not JavaScript primitives like strings or numbers:

```
interface ModalRendererProps { title: string; children: React.ReactElement;}
```

Note, that you cannot use TypeScript to describe that the children are a certain type of JSX elements, so you cannot use the type-system to describe a component which only accepts `` children.

You can see an example of both `React.ReactNode` and `React.ReactElement` with the type-checker in this TypeScript playground.

Style Props

When using inline styles in React, you can use `React.CSSProperties` to describe the object passed to the style prop. This type is a union of all the possible CSS properties, and is a good way to ensure you are passing valid CSS properties to the style prop, and to get auto-complete in your editor.

```
interface MyComponentProps { style: React.CSSProperties;}
```

Further learning

This guide has covered the basics of using TypeScript with React, but there is a lot more to learn.

Individual API pages on the docs may contain more in-depth documentation on how to use them with TypeScript.

We recommend the following resources:

The TypeScript handbook is the official documentation for TypeScript, and covers most key language features.

The TypeScript release notes cover new features in depth.

React TypeScript Cheatsheet is a community-maintained cheatsheet for using TypeScript with React, covering a lot of useful edge cases and providing more breadth than this document.

TypeScript Community Discord is a great place to ask questions and get help with TypeScript and React issues.

PreviousEditor SetupNextReact Developer Tools©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewInstallation Adding TypeScript to an existing React project TypeScript with React Components Example Hooks useState useReducer useContext useMemo useCallback Useful Types DOM Events Children Style Props Further learning React Developer Tools –
ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN
REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?Learn ReactInstallationReact Developer ToolsUse React Developer Tools to inspect React components, edit props and state, and identify performance problems.

You will learn

How to install React Developer Tools

Browser extension

The easiest way to debug websites built with React is to install the React Developer Tools browser extension. It is available for several popular browsers:

Install for Chrome

[Install for Firefox](#)

[Install for Edge](#)

Now, if you visit a website built with React, you will see the Components and Profiler panels.

Safari and other browsers

For other browsers (for example, Safari), install the react-devtools npm package:

```
# Yarnyarn global add react-devtools# Npmnpm install -g react-devtools
```

Next open the developer tools from the terminal:

```
react-devtools
```

Then connect your website by adding the following <script> tag to the beginning of your website's <head>:

```
<html> <head>  <script src="http://localhost:8097"></script>
```

Reload your website in the browser now to view it in developer tools.

Mobile (React Native)

React Developer Tools can be used to inspect apps built with React Native as well.

The easiest way to use React Developer Tools is to install it globally:

```
# Yarnyarn global add react-devtools# Npmnpm install -g react-devtools
```

Next open the developer tools from the terminal.

```
react-devtools
```

It should connect to any local React Native app that's running.

Try reloading the app if developer tools doesn't connect after a few seconds.

Learn more about debugging React Native.
Previous
Using TypeScript
Next
React Compiler
©2024 no uwu plzuwu?
Logo
by@sawaratsuki1004
Learn React
Quick Start
Installation
Describing the UI
Adding Interactivity
Managing State
Escape Hatches
API Reference
React APIs
React DOM APIs
Community
Code of Conduct
Meet the Team
Docs
Contributors
Acknowledgements
More
Blog
React Native
Privacy
Terms
On this page
Overview
Browser extension
Safari and other browsers
Mobile (React Native)
React Compiler –
React
React v18.3.1
Search Ctrl+K
Learn
Reference
Community
Blog
GET STARTED
Quick Start Tutorial: Tic-Tac-Toe
Thinking in React
Installation
Start a New React Project
Add React to an Existing Project
Editor Setup Using TypeScript
React Developer Tools
React Compiler - This feature is available in the latest Canary
LEARN
REACT
Describing the UI
Your First Component
Importing and Exporting Components
Writing Markup with JSX
JavaScript in JSX with Curly Braces
Passing Props to a Component
Conditional Rendering
Rendering Lists
Keeping Components Pure
Your UI as a Tree
Adding Interactivity
Responding to Events
State: A Component's Memory
Render and Commit State as a Snapshot
Queueing a Series of State Updates
Updating Objects in State
Updating Arrays in State
Managing State
Reacting to Input with State
Choosing the State Structure
Sharing State Between Components
Preserving and Resetting State
Extracting State Logic into a Reducer
Passing Data Deeply with Context
Scaling Up with Reducer and Context
Escape Hatches
Referencing Values with Refs
Manipulating the DOM with Refs

Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects
Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?Learn ReactInstallationReact CompilerThis page will give you an introduction to the new experimental React Compiler and how to try it out successfully.

Under ConstructionThese docs are still a work in progress. More documentation is available in the React Compiler Working Group repo, and will be upstreamed into these docs when they are more stable.

You will learn

Getting started with the compiler

Installing the compiler and eslint plugin

Troubleshooting

NoteReact Compiler is a new experimental compiler that we've open sourced to get early feedback from the community. It still has rough edges and is not yet fully ready for production.React Compiler requires React 19 RC. If you are unable to upgrade to React 19, you may try a userspace implementation of the cache function as described in the Working Group. However, please note that this is not recommended and you should upgrade to React 19 when possible.

React Compiler is a new experimental compiler that we've open sourced to get early feedback from the community. It is a build-time only tool that automatically optimizes your React app. It works with plain JavaScript, and understands the Rules of React, so you don't need to rewrite any code to use it.

The compiler also includes an eslint plugin that surfaces the analysis from the compiler right in your editor. The plugin runs independently of the compiler and can be used even if you aren't using the compiler in your app. We recommend all React developers to use this eslint plugin to help improve the quality of your codebase.

What does the compiler do?

In order to optimize applications, React Compiler automatically memoizes your code. You may be familiar today with memoization through APIs such as useMemo, useCallback, and React.memo. With these APIs you can tell React that certain parts of your application don't need to recompute if their inputs haven't changed, reducing work on updates. While powerful, it's easy to forget to apply memoization or apply them incorrectly. This can lead to inefficient updates as React has to check parts of your UI that don't have any meaningful changes.

The compiler uses its knowledge of JavaScript and React's rules to automatically memoize values or groups of values within your components and hooks. If it detects breakages of the rules, it will automatically skip over just those components or hooks, and continue safely compiling other code.

If your codebase is already very well-memoized, you might not expect to see major performance improvements with the compiler. However, in practice memoizing the correct dependencies that cause performance issues is tricky to get right by hand.

Deep DiveWhat kind of memoization does React Compiler add? Show DetailsThe initial release of React Compiler is primarily focused on improving update performance (re-rendering existing components), so it focuses on these two use cases:

Skipping cascading re-rendering of components

Re-rendering <Parent /> causes many components in its component tree to re-render, even though only <Parent /> has changed

Skipping expensive calculations from outside of React

For example, calling `expensivelyProcessAReallyLargeArrayOfObjects()` inside of your component or hook that needs that data

Optimizing Re-renders React lets you express your UI as a function of their current state (more concretely: their props, state, and context). In its current implementation, when a component's state changes, React will re-render that component and all of its children — unless you have applied some form of manual memoization with `useMemo()`, `useCallback()`, or `React.memo()`. For example, in the following example, `<MessageButton>` will re-render whenever `<FriendList>`'s state changes:

```
function FriendList({ friends }) { const onlineCount = useFriendOnlineCount(); if (friends.length === 0) { return <NoFriends />; } return ( <div> <span>{onlineCount} online</span> {friends.map((friend) => ( <FriendListCard key={friend.id} friend={friend}> /> ))} <MessageButton /> </div> ); }
```

See this example in the React Compiler Playground

React Compiler automatically applies the equivalent of manual memoization, ensuring that only the relevant parts of an app re-render as state changes, which is sometimes referred to as “fine-grained reactivity”. In the above example, React Compiler determines that the return value of `<FriendListCard />` can be reused even as `friends` changes, and can avoid recreating this JSX and avoid re-rendering `<MessageButton>` as the count changes.

Expensive calculations also get memoized. The compiler can also automatically memoize for expensive calculations used during rendering:

```
// **Not** memoized by React Compiler, since this is not a component or hook
function expensivelyProcessAReallyLargeArrayOfObjects() { /* ... */ }
```

Memoized by React Compiler since this is a component:

```
function TableContainer({ items }) { // This function call would be memoized: const data = expensivelyProcessAReallyLargeArrayOfObjects(items); // ... }
```

See this example in the React Compiler Playground

However, if `expensivelyProcessAReallyLargeArrayOfObjects` is truly an expensive function, you may want to consider implementing its own memoization outside of React, because:

React Compiler only memoizes React components and hooks, not every function

React Compiler's memoization is not shared across multiple components or hooks

So if `expensivelyProcessAReallyLargeArrayOfObjects` was used in many different components, even if the same exact items were passed down, that expensive calculation would be run repeatedly. We recommend profiling first to see if it really is that expensive before making code more complicated.

What does the compiler assume?

React Compiler assumes that your code:

Is valid, semantic JavaScript

Tests that nullable/optional values and properties are defined before accessing them (for example, by enabling `strictNullChecks` if using TypeScript), i.e., if `(object.nullableProperty) { object.nullableProperty.foo }` or with optional-chaining `object.nullableProperty?.foo`

Follows the Rules of React

React Compiler can verify many of the Rules of React statically, and will safely skip compilation when it detects an error. To see the errors we recommend also installing `eslint-plugin-react-compiler`.

Should I try out the compiler?

Please note that the compiler is still experimental and has many rough edges. While it has been used in production at companies like Meta, rolling out the compiler to production for your app will depend on the health of your codebase and how well you've followed the Rules of React.

You don't have to rush into using the compiler now. It's okay to wait until it reaches a stable release before adopting it. However, we do appreciate trying it out in small experiments in your apps so that you can provide feedback to us to help make the compiler better.

Getting Started

In addition to these docs, we recommend checking the React Compiler Working Group for additional information and discussion about the compiler.

Checking compatibility

Prior to installing the compiler, you can first check to see if your codebase is compatible:

```
Terminal Copynpx react-compiler-healthcheck@experimental
```

This script will:

Check how many components can be successfully optimized: higher is better

Check for `<StrictMode>` usage: having this enabled and followed means a higher chance that the Rules of React are followed

Check for incompatible library usage: known libraries that are incompatible with the compiler

As an example:

```
Terminal CopySuccessfully compiled 8 out of 9 components.
```

StrictMode usage not found.

Found no usage of incompatible libraries.

Installing eslint-plugin-react-compiler

React Compiler also powers an eslint plugin. The eslint plugin can be used independently of the compiler, meaning you can use the eslint plugin even if you don't use the compiler.

```
Terminal Copynpm install eslint-plugin-react-compiler@experimental
```

Then, add it to your eslint config:

```
module.exports = { plugins: [ 'eslint-plugin-react-compiler', ], rules: { 'react-compiler/react-compiler': "error", }}
```

The eslint plugin will display any violations of the rules of React in your editor. When it does this, it means that the compiler has skipped over optimizing that component or hook. This is perfectly okay, and the compiler can recover and continue optimizing other components in your codebase.

You don't have to fix all eslint violations straight away. You can address them at your own pace to increase the amount of components and hooks being optimized, but it is not required to fix everything before you can use the compiler.

Rolling out the compiler to your codebase

Existing projects

The compiler is designed to compile functional components and hooks that follow the Rules of React. It can also handle code that breaks those rules by bailing out (skipping over) those components or hooks. However, due to the flexible nature of JavaScript, the compiler cannot catch every possible violation and may compile with false negatives: that is, the compiler may accidentally compile a component/hook that breaks the Rules of React which can lead to undefined behavior.

For this reason, to adopt the compiler successfully on existing projects, we recommend running it on a small directory in your product code first. You can do this by configuring the compiler to only run on a specific set of directories:

```
const ReactCompilerConfig = { sources: (filename) => { return filename.indexOf('src/path/to/dir') !== -1; },};
```

In rare cases, you can also configure the compiler to run in “opt-in” mode using the compilationMode: "annotation" option. This makes it so the compiler will only compile components and hooks annotated with a "use memo" directive. Please note that the annotation mode is a temporary one to aid early adopters, and that we don't intend for the "use memo" directive to be used for the long term.

```
const ReactCompilerConfig = { compilationMode: "annotation",};// src/app.jsxexport default function App() { "use memo"; // ...}
```

When you have more confidence with rolling out the compiler, you can expand coverage to other directories as well and slowly roll it out to your whole app.

New projects

If you're starting a new project, you can enable the compiler on your entire codebase, which is the default behavior.

Usage

Babel

Terminal Copynpm install babel-plugin-react-compiler@experimental

The compiler includes a Babel plugin which you can use in your build pipeline to run the compiler.

After installing, add it to your Babel config. Please note that it's critical that the compiler run first in the pipeline:

```
// babel.config.jsconst ReactCompilerConfig = { /* ... */ };module.exports = function () { return { plugins: [ ['babel-plugin-react-compiler', ReactCompilerConfig], // must run first! // ... ], }, },;
```

babel-plugin-react-compiler should run first before other Babel plugins as the compiler requires the input source information for sound analysis.

Vite

If you use Vite, you can add the plugin to vite-plugin-react:

```
// vite.config.jsconst ReactCompilerConfig = { /* ... */ };export default defineConfig(() => { return { plugins: [ react({ babel: { plugins: [ ["babel-plugin-react-compiler", ReactCompilerConfig], ], }, }, ), ], // ... }, };);
```

Next.js

Next.js has an experimental configuration to enable the React Compiler. It automatically ensures Babel is set up with babel-plugin-react-compiler.

Install Next.js canary, which uses React 19 Release Candidate

Install babel-plugin-react-compiler

Terminal Copynpm install next@canary babel-plugin-react-compiler@experimental

Then configure the experimental option in next.config.js:

```
// next.config.js/** @type {import('next').NextConfig} */const nextConfig = { experimental: { reactCompiler: true, };};module.exports = nextConfig;
```

Using the experimental option ensures support for the React Compiler in:

App Router

Pages Router

Webpack (default)

Turbopack (opt-in through --turbo)

Remix

Install vite-plugin-babel, and add the compiler's Babel plugin to it:

Terminal Copynpm install vite-plugin-babel

```
// vite.config.jsimport babel from "vite-plugin-babel";const ReactCompilerConfig = { /* ... */ };export default defineConfig({ plugins: [ remix({ /* ... */ }), babel({ filter: /\.[jt]sx?$/, babelConfig: { presets: ["@babel/preset-typescript"], // if you use TypeScript plugins: [ ["babel-plugin-react-compiler", ReactCompilerConfig], ], }, }), ],});
```

Webpack

You can create your own loader for React Compiler, like so:

```
const ReactCompilerConfig = { /* ... */ };const BabelPluginReactCompiler = require('babel-plugin-react-compiler');function reactCompilerLoader(sourceCode, sourceMap) { // ... const result = transformSync(sourceCode, { // ... plugins: [ [BabelPluginReactCompiler, ReactCompilerConfig], ], // ... }); if (result === null) { this.callback( Error(`Failed to transform "${options.filename}"` ) ); return; } this.callback( null, result.code, result.map === null ? undefined : result.map );}module.exports = reactCompilerLoader;
```

Expo

Please refer to Expo's docs to enable and use the React Compiler in Expo apps.

Metro (React Native)

React Native uses Babel via Metro, so refer to the Usage with Babel section for installation instructions.

Rspack

Please refer to Rspack's docs to enable and use the React Compiler in Rspack apps.

Rsbuild

Please refer to Rsbuild's docs to enable and use the React Compiler in Rsbuild apps.

Troubleshooting

To report issues, please first create a minimal repro on the React Compiler Playground and include it in your bug report. You can open issues in the facebook/react repo.

You can also provide feedback in the React Compiler Working Group by applying to be a member. Please see the README for more details on joining.

(0 , _c) is not a function error

This occurs if you are not using React 19 RC and up. To fix this, upgrade your app to React 19 RC first.

If you are unable to upgrade to React 19, you may try a userspace implementation of the cache function as described in the Working Group. However, please note that this is not recommended and you should upgrade to React 19 when possible.

How do I know my components have been optimized?

React Devtools (v5.0+) has built-in support for React Compiler and will display a “Memo” badge next to components that have been optimized by the compiler.

Something is not working after compilation

If you have eslint-plugin-react-compiler installed, the compiler will display any violations of the rules of React in your editor. When it does this, it means that the compiler has skipped over optimizing that component or hook. This is perfectly okay, and the compiler can recover and continue optimizing other components in your codebase. You don't have to fix all eslint violations straight away. You can address them at your own pace to increase the amount of components and hooks being optimized.

Due to the flexible and dynamic nature of JavaScript however, it's not possible to comprehensively detect all cases. Bugs and undefined behavior such as infinite loops may occur in those cases.

If your app doesn't work properly after compilation and you aren't seeing any eslint errors, the compiler may be incorrectly compiling your code. To confirm this, try to make the issue go away by aggressively opting out any component or hook you think might be related via the "use no memo" directive.

```
function SuspiciousComponent() { "use no memo"; // opts out this component from being compiled by React Compiler // ...}
```

Note "use no memo" "use no memo" is a temporary escape hatch that lets you opt-out components and hooks from being compiled by the React Compiler. This directive is not meant to be long lived the same way as eg "use client" is. It is not recommended to reach for this directive unless it's strictly necessary. Once you opt-out a component or hook, it is opted-out forever until the directive is removed. This means that even if you fix the code, the compiler will still skip over compiling it unless you remove the directive.

When you make the error go away, confirm that removing the opt out directive makes the issue come back. Then share a bug report with us (you can try to reduce it to a small repro, or if it's open source code you can also just paste the entire source) using the React Compiler Playground so we can identify and help fix the issue.

Other issues

Please see <https://github.com/reactwg/react-compiler/discussions/7>. Previous React Developer Tools © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview What does the compiler do? What does the compiler assume? Should I try out the compiler? Getting Started Checking compatibility Installing eslint-plugin-react-compiler Rolling out the compiler to your codebase Usage Babel Vite Next.js Remix Webpack Expo Metro (React Native) Rspack Rsbuild Troubleshooting (0 , _c) is not a function error How do I know my components have been optimized? Something is not working after compilation Other issues React Compiler – React React v18.3.1 Search Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in

[State Managing State Reacting to Input](#) [with State Choosing the State Structure](#) [Sharing State Between Components](#)
[Preserving and Resetting State](#) [Extracting State Logic into a Reducer](#) [Passing Data Deeply with Context](#) [Scaling Up with Reducer and Context](#) [Escape Hatches](#) [Referencing Values with Refs](#) [Manipulating the DOM with Refs](#)
[Synchronizing with Effects](#) [You Might Not Need an Effect](#) [Lifecycle of Reactive Effects](#) [Separating Events from Effects](#)
[Removing Effect Dependencies](#) [Reusing Logic with Custom Hooks](#) [Is this page useful?](#) [Learn React](#) [Installation](#) [React Compiler](#)
This page will give you an introduction to the new experimental React Compiler and how to try it out successfully.

Under Construction These docs are still a work in progress. More documentation is available in the React Compiler Working Group repo, and will be upstreamed into these docs when they are more stable.

You will learn

[Getting started with the compiler](#)

[Installing the compiler and eslint plugin](#)

[Troubleshooting](#)

Note React Compiler is a new experimental compiler that we've open sourced to get early feedback from the community. It still has rough edges and is not yet fully ready for production. React Compiler requires React 19 RC. If you are unable to upgrade to React 19, you may try a userspace implementation of the cache function as described in the Working Group. However, please note that this is not recommended and you should upgrade to React 19 when possible.

React Compiler is a new experimental compiler that we've open sourced to get early feedback from the community. It is a build-time only tool that automatically optimizes your React app. It works with plain JavaScript, and understands the Rules of React, so you don't need to rewrite any code to use it.

The compiler also includes an eslint plugin that surfaces the analysis from the compiler right in your editor. The plugin runs independently of the compiler and can be used even if you aren't using the compiler in your app. We recommend all React developers to use this eslint plugin to help improve the quality of your codebase.

What does the compiler do?

In order to optimize applications, React Compiler automatically memoizes your code. You may be familiar today with memoization through APIs such as `useMemo`, `useCallback`, and `React.memo`. With these APIs you can tell React that certain parts of your application don't need to recompute if their inputs haven't changed, reducing work on updates. While powerful, it's easy to forget to apply memoization or apply them incorrectly. This can lead to inefficient updates as React has to check parts of your UI that don't have any meaningful changes.

The compiler uses its knowledge of JavaScript and React's rules to automatically memoize values or groups of values within your components and hooks. If it detects breakages of the rules, it will automatically skip over just those components or hooks, and continue safely compiling other code.

If your codebase is already very well-memoized, you might not expect to see major performance improvements with the compiler. However, in practice memoizing the correct dependencies that cause performance issues is tricky to get right by hand.

Deep Dive What kind of memoization does React Compiler add? Show Details The initial release of React Compiler is primarily focused on improving update performance (re-rendering existing components), so it focuses on these two use cases:

[Skipping cascading re-rendering of components](#)

Re-rendering <Parent /> causes many components in its component tree to re-render, even though only <Parent /> has changed

Skipping expensive calculations from outside of React

For example, calling `expensivelyProcessAReallyLargeArrayOfObjects()` inside of your component or hook that needs that data

Optimizing Re-renders React lets you express your UI as a function of their current state (more concretely: their props, state, and context). In its current implementation, when a component's state changes, React will re-render that component and all of its children — unless you have applied some form of manual memoization with `useMemo()`, `useCallback()`, or `React.memo()`. For example, in the following example, `<MessageButton>` will re-render whenever `<FriendList>`'s state changes:

```
function FriendList({ friends }) { const onlineCount = useFriendOnlineCount(); if (friends.length === 0) { return <NoFriends />; } return ( <div> <span>{onlineCount} online</span> {friends.map((friend) => ( <FriendListCard key={friend.id} friend={friend}>/> ))} <MessageButton /> </div> ); }
```

See this example in the React Compiler Playground. React Compiler automatically applies the equivalent of manual memoization, ensuring that only the relevant parts of an app re-render as state changes, which is sometimes referred to as “fine-grained reactivity”. In the above example, React Compiler determines that the return value of `<FriendListCard />` can be reused even as `friends` changes, and can avoid recreating this JSX and avoid re-rendering `<MessageButton>` as the count changes. Expensive calculations also get memoized. The compiler can also automatically memoize for expensive calculations used during rendering:

```
// **Not** memoized by React Compiler, since this is not a component or hook
function expensivelyProcessAReallyLargeArrayOfObjects() { /* ... */ }
```

Memoized by React Compiler since this is a component:

```
function TableContainer({ items }) { // This function call would be memoized: const data = expensivelyProcessAReallyLargeArrayOfObjects(items); // ... }
```

See this example in the React Compiler Playground.

However, if `expensivelyProcessAReallyLargeArrayOfObjects` is truly an expensive function, you may want to consider implementing its own memoization outside of React, because:

React Compiler only memoizes React components and hooks, not every function

React Compiler's memoization is not shared across multiple components or hooks

So if `expensivelyProcessAReallyLargeArrayOfObjects` was used in many different components, even if the same exact items were passed down, that expensive calculation would be run repeatedly. We recommend profiling first to see if it really is that expensive before making code more complicated.

What does the compiler assume?

React Compiler assumes that your code:

Is valid, semantic JavaScript

Tests that nullable/optional values and properties are defined before accessing them (for example, by enabling `strictNullChecks` if using TypeScript), i.e., if `(object.nullableProperty) { object.nullableProperty.foo }` or with optional-chaining `object.nullableProperty?.foo`

Follows the Rules of React

React Compiler can verify many of the Rules of React statically, and will safely skip compilation when it detects an error. To see the errors we recommend also installing eslint-plugin-react-compiler.

Should I try out the compiler?

Please note that the compiler is still experimental and has many rough edges. While it has been used in production at companies like Meta, rolling out the compiler to production for your app will depend on the health of your codebase and how well you've followed the Rules of React.

You don't have to rush into using the compiler now. It's okay to wait until it reaches a stable release before adopting it. However, we do appreciate trying it out in small experiments in your apps so that you can provide feedback to us to help make the compiler better.

Getting Started

In addition to these docs, we recommend checking the React Compiler Working Group for additional information and discussion about the compiler.

Checking compatibility

Prior to installing the compiler, you can first check to see if your codebase is compatible:

```
Terminal Copynpx react-compiler-healthcheck@experimental
```

This script will:

Check how many components can be successfully optimized: higher is better

Check for <StrictMode> usage: having this enabled and followed means a higher chance that the Rules of React are followed

Check for incompatible library usage: known libraries that are incompatible with the compiler

As an example:

```
Terminal CopySuccessfully compiled 8 out of 9 components.
```

StrictMode usage not found.

Found no usage of incompatible libraries.

Installing eslint-plugin-react-compiler

React Compiler also powers an eslint plugin. The eslint plugin can be used independently of the compiler, meaning you can use the eslint plugin even if you don't use the compiler.

```
Terminal Copynpm install eslint-plugin-react-compiler@experimental
```

Then, add it to your eslint config:

```
module.exports = { plugins: [ 'eslint-plugin-react-compiler', ], rules: { 'react-compiler/react-compiler': "error", }}
```

The eslint plugin will display any violations of the rules of React in your editor. When it does this, it means that the compiler has skipped over optimizing that component or hook. This is perfectly okay, and the compiler can recover and continue optimizing other components in your codebase.

You don't have to fix all eslint violations straight away. You can address them at your own pace to increase the amount of components and hooks being optimized, but it is not required to fix everything before you can use the compiler.

Rolling out the compiler to your codebase

Existing projects

The compiler is designed to compile functional components and hooks that follow the Rules of React. It can also handle code that breaks those rules by bailing out (skipping over) those components or hooks. However, due to the flexible nature of JavaScript, the compiler cannot catch every possible violation and may compile with false negatives: that is, the compiler may accidentally compile a component/hook that breaks the Rules of React which can lead to undefined behavior.

For this reason, to adopt the compiler successfully on existing projects, we recommend running it on a small directory in your product code first. You can do this by configuring the compiler to only run on a specific set of directories:

```
const ReactCompilerConfig = { sources: (filename) => { return filename.indexOf('src/path/to/dir') !== -1; },};
```

In rare cases, you can also configure the compiler to run in “opt-in” mode using the compilationMode: "annotation" option. This makes it so the compiler will only compile components and hooks annotated with a "use memo" directive. Please note that the annotation mode is a temporary one to aid early adopters, and that we don’t intend for the "use memo" directive to be used for the long term.

```
const ReactCompilerConfig = { compilationMode: "annotation",};// src/app.jsxexport default function App() { "use memo"; // ...}
```

When you have more confidence with rolling out the compiler, you can expand coverage to other directories as well and slowly roll it out to your whole app.

New projects

If you’re starting a new project, you can enable the compiler on your entire codebase, which is the default behavior.

Usage

Babel

Terminal Copy
npm install babel-plugin-react-compiler@experimental

The compiler includes a Babel plugin which you can use in your build pipeline to run the compiler.

After installing, add it to your Babel config. Please note that it’s critical that the compiler run first in the pipeline:

```
// babel.config.jsconst ReactCompilerConfig = { /* ... */ };module.exports = function () { return { plugins: [ ['babel-plugin-react-compiler', ReactCompilerConfig], // must run first! // ... ], },};
```

babel-plugin-react-compiler should run first before other Babel plugins as the compiler requires the input source information for sound analysis.

Vite

If you use Vite, you can add the plugin to vite-plugin-react:

```
// vite.config.jsconst ReactCompilerConfig = { /* ... */ };export default defineConfig(() => { return { plugins: [ react({ babel: { plugins: [ ["babel-plugin-react-compiler", ReactCompilerConfig], // ... ] } } ), ] },});
```

Next.js

Next.js has an experimental configuration to enable the React Compiler. It automatically ensures Babel is set up with babel-plugin-react-compiler.

Install Next.js canary, which uses React 19 Release Candidate

Install babel-plugin-react-compiler

Terminal Copy `npm install next@canary babel-plugin-react-compiler@experimental`

Then configure the experimental option in `next.config.js`:

```
// next.config.js/** @type {import('next').NextConfig} */const nextConfig = { experimental: { reactCompiler: true, };};module.exports = nextConfig;
```

Using the experimental option ensures support for the React Compiler in:

App Router

Pages Router

Webpack (default)

Turbopack (opt-in through `--turbo`)

Remix

Install `vite-plugin-babel`, and add the compiler's Babel plugin to it:

Terminal Copy `npm install vite-plugin-babel`

```
// vite.config.jsimport babel from "vite-plugin-babel";const ReactCompilerConfig = { /* ... */ };export default defineConfig({ plugins: [ remix({ /* ... */ }), babel({ filter: /\.js?$/ , babelConfig: { presets: ["@babel/preset-typescript"], // if you use TypeScript plugins: [ "babel-plugin-react-compiler", ReactCompilerConfig ], }, }), ],});
```

Webpack

You can create your own loader for React Compiler, like so:

```
const ReactCompilerConfig = { /* ... */ };const BabelPluginReactCompiler = require('babel-plugin-react-compiler');function reactCompilerLoader(sourceCode, sourceMap) { // ... const result = transformSync(sourceCode, { // ... plugins: [ [BabelPluginReactCompiler, ReactCompilerConfig], ], // ... }); if (result === null) { this.callback( Error(`Failed to transform "${options.filename}"` ) ); return; } this.callback( null, result.code, result.map === null ? undefined : result.map );}module.exports = reactCompilerLoader;
```

Expo

Please refer to Expo's docs to enable and use the React Compiler in Expo apps.

Metro (React Native)

React Native uses Babel via Metro, so refer to the Usage with Babel section for installation instructions.

Rspack

Please refer to Rspack's docs to enable and use the React Compiler in Rspack apps.

Rsbuild

Please refer to Rsbuild's docs to enable and use the React Compiler in Rsbuild apps.

Troubleshooting

To report issues, please first create a minimal repro on the React Compiler Playground and include it in your bug report. You can open issues in the facebook/react repo.

You can also provide feedback in the React Compiler Working Group by applying to be a member. Please see the README for more details on joining.

(0 , _c) is not a function error

This occurs if you are not using React 19 RC and up. To fix this, upgrade your app to React 19 RC first.

If you are unable to upgrade to React 19, you may try a userspace implementation of the cache function as described in the Working Group. However, please note that this is not recommended and you should upgrade to React 19 when possible.

How do I know my components have been optimized?

React Devtools (v5.0+) has built-in support for React Compiler and will display a “Memo” badge next to components that have been optimized by the compiler.

Something is not working after compilation

If you have eslint-plugin-react-compiler installed, the compiler will display any violations of the rules of React in your editor. When it does this, it means that the compiler has skipped over optimizing that component or hook. This is perfectly okay, and the compiler can recover and continue optimizing other components in your codebase. You don't have to fix all eslint violations straight away. You can address them at your own pace to increase the amount of components and hooks being optimized.

Due to the flexible and dynamic nature of JavaScript however, it's not possible to comprehensively detect all cases. Bugs and undefined behavior such as infinite loops may occur in those cases.

If your app doesn't work properly after compilation and you aren't seeing any eslint errors, the compiler may be incorrectly compiling your code. To confirm this, try to make the issue go away by aggressively opting out any component or hook you think might be related via the "use no memo" directive.

```
function SuspiciousComponent() { "use no memo"; // opts out this component from being compiled by React Compiler // ...}
```

Note "use no memo" "use no memo" is a temporary escape hatch that lets you opt-out components and hooks from being compiled by the React Compiler. This directive is not meant to be long lived the same way as eg "use client" is. It is not recommended to reach for this directive unless it's strictly necessary. Once you opt-out a component or hook, it is opted-out forever until the directive is removed. This means that even if you fix the code, the compiler will still skip over compiling it unless you remove the directive.

When you make the error go away, confirm that removing the opt out directive makes the issue come back. Then share a bug report with us (you can try to reduce it to a small repro, or if it's open source code you can also just paste the entire source) using the React Compiler Playground so we can identify and help fix the issue.

Other issues

Please see <https://github.com/reactwg/react-compiler/discussions/7>. Previous React Developer Tools © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview What does the compiler do? What does the compiler assume? Should I try out the compiler? Getting Started Checking compatibility Installing eslint-plugin-react-compiler Rolling out the compiler to your codebase Usage Babel Vite Next.js Remix Webpack Expo Metro (React Native) Rspack Rsbuild Troubleshooting (0 , _c) is not a function error How do I know my components have been optimized? Something is not working after compilation Other issues Your First Component – React React v18.3.1 Search Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using

TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN
REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX
JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping
Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render
and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in
State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components
Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up
with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs
Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects
Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ``:

```
<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 ``. 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 ResetForkexport default function Profile() {
```

```
return (
```

```
  
)  
}
```

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 tag with src and alt attributes. 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 ( <div>  </div>);
```

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 ResetForkfunction Profile() {  
  return (  
      
  );  
}
```

```
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: . 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() { // 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() { // ...} // tickmark Declare components at the top level`
function 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 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.

Recap You'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 challenges! 1. 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.js

```
App.js
export function App() {
  return (
    
  );
}
```

Try to fix it yourself before looking at the solution! Show solution Next Challenge Previous Describing the UI
Next Importing and Exporting Components ©2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Components: UI building blocks Defining a component Step 1: Export the component Step 2: Define the function Step 3: Add markup Using a component What the browser sees Nesting and organizing components Recap Challenges Importing and Exporting Components –

ReactReactv18.3.1Search⌘FCtrlKLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe
Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using
TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN
REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX
JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping
Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render
and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in
State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components
Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up
with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs
Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects
Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ResetForkfunction 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. 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.js
Gallery.js
App.js
Reset
Fork
import 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.

Note You 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 Dive Default vs named exports Show Details There 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: Syntax Export statement Import statement Default export function Button() {}; import Button from './Button.js'; Named export 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!

Note To 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.js
Gallery.js
App.js ResetFor
import 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

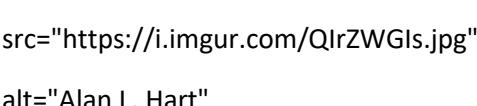
How to export multiple components from the same file

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 (

);

}

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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewThe root component file Exporting and importing a component Exporting and importing multiple components from the same file RecapChallengesWriting Markup with JSX – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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><ul> <li>Invent new traffic lights <li>Rehearse a movie scene <li>Improve the spectrum technology</ul>
```

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 ResetForkexport default function TodoList() {  
  return (  
    // This doesn't quite work!  
    <h1>Hedy Lamarr's Todos</h1>  
      
<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>:

```
<div> <h1>Hedy Lamarr's Todos</h1> <img alt="Hedy Lamarr" class="photo" /> <ul> ... </ul></div>
```

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

```
<> <h1>Hedy Lamarr's Todos</h1> <img 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 must become , and wrapping tags like oranges must be written as oranges.

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

```
<> <img 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.

Pitfall For 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 ResetForkexport default function TodoList() {  
  return (  
    <>  
      <h1>Hedy Lamarr's Todos</h1>  
        
      <ul>  
        <li>Invent new traffic lights</li>  
        <li>Rehearse a movie scene</li>  
        <li>Improve the spectrum technology</li>  
      </ul>  
    </>  
  );  
}
```

Show more

Recap Now 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 ResetForkexport default function Bio() {

```
return (  
  <div class="intro">  
    <h1>Welcome to my website!</h1>  
  </div>  
  
  <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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewJSX: Putting markup into JavaScript Converting HTML to JSX The Rules of JSX 1. Return a single root element 2. Close all the tags 3. camelCase all most of the things! Pro-tip: Use a JSX Converter RecapChallengesJavaScript in JSX with Curly Braces – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ResetForkexport 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 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 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 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 curly braces”: 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 ResetForKexport default function TodoList() {  
  return (  
    <ul style={{  
      backgroundColor: 'black',  
      color: 'pink'  
    }}>  
      <li>Improve the videophone</li>  
      <li>Prepare aeronautics lectures</li>  
      <li>Work on the alcohol-fuelled engine</li>  
    </ul>  
  );  
}
```

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 curly braces!

Pitfall Inline 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 ResetForKconst person = {  
  name: 'Gregorio Y. Zara',  
  theme: {  
    backgroundColor: 'black',  
    color: 'pink'  
  }  
};
```

```

export default function TodoList() {
  return (
    <div style={person.theme}>
      <h1>{person.name}'s Todos</h1>
      
      <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:

```
<div style={person.theme}> <h1>{person.name}'s Todos</h1>
```

JSX is very minimal as a templating language because it lets you organize data and logic using JavaScript.

Recap Now 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 challenges
 1. 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.js`
`Reset`
`const person = {`

```
  name: 'Gregorio Y. Zara',
```

```

theme: {
  backgroundColor: 'black',
  color: 'pink'
}
};

export default function TodoList() {
  return (
    <div style={person.theme}>
      <h1>{person}'s Todos</h1>
      
      <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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewPassing strings with quotes Using curly braces: A window into the JavaScript world Where to use curly braces Using “double curly braces”: CSS and other objects in JSX More fun with JavaScript objects and curly braces RecapChallengesPassing Props to a Component – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-ToeThinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components

Preserving and Resetting State
Extracting State Logic into a Reducer
Passing Data Deeply with Context
Scaling Up with Reducer and Context
Escape Hatches
Referencing Values with Refs
Manipulating the DOM with Refs
Synchronizing with Effects You Might Not Need an Effect
Lifecycle of Reactive Effects Separating Events from Effects
Removing Effect Dependencies Reusing Logic with Custom Hooks
Is this page useful? Learn React
Describing the UI
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, 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 :

```
App.jsApp.js ResetForkfunction Avatar() {
```

```
return (
```

```

```

```
);
```

```
}
```

```
export default function Profile() {
```

```
return (
```

```
<Avatar />
```

```
);
```

```
}
```

Show more

The props you can pass to an `` 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', imageUrl: '1bX5QH6' }} size={100} /> );}
```

Note the 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 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 ResetForImport { 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',
    imageUrl: 'YfeOqp2'
  }}
/>

<Avatar
  size={80}
  person={{
    name: 'Aklilu Lemma',
    imageUrl: 'OKS67lh'
  }}
/>

<Avatar
  size={50}
  person={{
    name: 'Lin Lanying',
    imageUrl: '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.

Pitfall Don'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.js
Avatar.js
utils.js
App.js
Reset.js
import 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',
          imageUrl: '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.js
utils.js
utils.js
import { useState } from 'react';
import { ResetFork } from './ResetFork';
import { imageUrl } from './utils';

function App() {
  const [color, setColor] = useState('red');
  const [time, setTime] = useState(0);

  return (
    

# Notable Scientists



- ## Maria Skłodowska-Curie



Profile: {time}
- ## Albert Einstein



Profile: {time}


  );
}

export default App;
```

```
Gallery.js
export default function Gallery() {
  return (
    <div>
      <h1>Notable Scientists</h1>
      <ul>
        <li>
          <img alt="Maria Skłodowska-Curie" data={color} style={{ width: 70, height: 70 }}/>
          <h2>Maria Skłodowska-Curie</h2>
          <p>Profile: {time}</p>
        </li>
        <li>
          <img alt="Albert Einstein" data={color} style={{ width: 70, height: 70 }}/>
          <h2>Albert Einstein</h2>
          <p>Profile: {time}</p>
        </li>
      </ul>
    </div>
  );
}
```

```
<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 (chemical 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>
    </li>
```

```

    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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewFamiliar props Passing props to a component Step 1: Pass props to the child component Step 2: Read props inside the child component Specifying a default value for a prop Forwarding props with the JSX spread syntax Passing JSX as children How props change over time RecapChallengesConditional Rendering – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ResetForkfunction Item({ name, isPacked }) {
  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

Notice that some of the Item components have their isPacked prop set to true instead of false. You want to add a checkmark (tickmark) to packed items if isPacked={true}.

You can write this as an if/else statement like so:

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

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 ResetForkfunction Item({ name, isPacked }) {
  if (isPacked) {
    return <li className="item">{name} tickmark</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 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} tickmark</li>
```

is very similar to

```
<li className="item">{name}</li>
```

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

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

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} tickmark</li>;}return <li className="item">{name}</li>;
```

You can write this:

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

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

Deep Dive Are these two examples fully equivalent? Show Details If 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 ``. 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 `` 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 ResetForkfunction Item({ name, isPacked }) {
```

```
    return (
```

```
        <li className="item">
```

```
            {isPacked ? (
```

```
                <del>
```

```
                    {name + ' tickmark'}
```

```
                </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 && 'tickmark'} </li>);
```

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

Here it is in action:

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

```
  return (
```

```
    <li className="item">
```

```
      {name} {isPacked && 'tickmark'}
```

```
    </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.

Pitfall Don’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 && <p>New messages</p>`. 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 && <p>New messages</p>`.

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 + " tickmark";}
```

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}</li>
```

This style is the most verbose, but it's also the most flexible. Here it is in action:

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

```
let itemContent = name;
```

```
if (isPacked) {
```

```
    itemContent = name + " tickmark";
```

```
}
```

```
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

Like before, this works not only for text, but for arbitrary JSX too:

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

```
  let itemContent = name;  
  
  if (isPacked) {  
  
    itemContent = (  
      <del>  
        {name + " tickmark"}  
      </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 /> : } means “if cond, render <A />, otherwise ”.

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 if isPacked isn't true.

```

App.js
function Item({ name, isPacked }) {

  return (
    <li className="item">
      {name} {isPacked && 'tickmark'}
    </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 Show solutionNext ChallengePreviousPassing Props to a ComponentNextRendering Lists©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewConditionally returning JSX Conditionally returning nothing with null Conditionally including JSX Conditional (ternary) operator (`? :`) Logical AND operator (`&&`) Conditionally assigning JSX to a variable RecapChallengesRendering Lists – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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.

```
<ul> <li>Creola Katherine Johnson: mathematician</li> <li>Mario José Molina-Pasquel Henríquez: chemist</li>
<li>Mohammad Abdus Salam: physicist</li> <li>Percy Lavon Julian: chemist</li> <li>Subrahmanyan Chandrasekhar:
astrophysicist</li></ul>
```

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 ``:

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

Here is the result:

```
App.jsApp.js 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' }, { id: 3, name: 'Percy Lavon Julian', profession: 'chemist' }, { id: 4, 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 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 ...});` 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}>...</li>
```

Note JSX 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',  
  imageUrl: 'MK3eW3A'  
}, {  
  id: 1, // Used in JSX as a key  
  name: 'Mario José Molina-Pasquel Henríquez',  
  profession: 'chemist',  
  accomplishment: 'discovery of Arctic ozone hole',  
  imageUrl: 'mynHUSA'  
}, {  
  id: 2, // Used in JSX as a key  
  name: 'Mohammad Abdus Salam',  
  profession: 'physicist',  
  accomplishment: 'electromagnetism theory',  
  imageUrl: 'bE7W1ji'  
}, {  
  id: 3, // Used in JSX as a key  
  name: 'Percy Lavon Julian',  
  profession: 'chemist',  
  accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',  
  imageUrl: 'IOjWm71'  
}, {  
  id: 4, // Used in JSX as a key  
  name: 'Subrahmanyan Chandrasekhar',  
  profession: 'astrophysicist',  
  accomplishment: 'white dwarf star mass calculations',
```

```
imageId: 'lrWQx8I'
```

```
};
```

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 `<div>`, 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> <p>{person.bio}</p> </Fragment>);Fragments disappear from the DOM, so this will produce a flat list of <h1>, <p>, <h1>, <p>, 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©2024no
uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding
InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of
ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this

pageOverviewRendering data from arrays Filtering arrays of items Keeping list items in order with key Where to get your key Rules of keys Why does React need keys? RecapChallengesKeeping Components Pure – ReactReactv18.3.1Search⌘CtrlKLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 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 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 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 Dive
[Detecting impure calculations with StrictMode](#) [Show Details](#)
Although 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 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 Dive Why does React care about purity? Show Details Writing 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 ListsNextYour UI as a Tree©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewPurity: Components as formulas Side Effects: (un)intended consequences Local mutation: Your component's little secret Where you can cause side effects RecapChallengesUnderstanding Your UI as a Tree – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?Learn ReactDescribing the UIUnderstanding Your UI as a TreeYour React app is taking shape with many components being nested within each other. How does React keep track of your app's component structure?React, and many other UI libraries, model UI as a tree. Thinking of your app as a tree is useful for understanding the relationship between components. This understanding will help you debug future concepts like performance and state management.

You will learn

How React “sees” component structures

What a render tree is and what it is useful for

What a module dependency tree is and what it is useful for

Your UI as a tree

Trees are a relationship model between items and UI is often represented using tree structures. For example, browsers use tree structures to model HTML (DOM) and CSS (CSSOM). Mobile platforms also use trees to represent their view hierarchy.

React creates a UI tree from your components. In this example, the UI tree is then used to render to the DOM.

Like browsers and mobile platforms, React also uses tree structures to manage and model the relationship between components in a React app. These trees are useful tools to understand how data flows through a React app and how to optimize rendering and app size.

The Render Tree

A major feature of components is the ability to compose components of other components. As we nest components, we have the concept of parent and child components, where each parent component may itself be a child of another component.

When we render a React app, we can model this relationship in a tree, known as the render tree.

Here is a React app that renders inspirational quotes.

```
App.js
FancyText.js
InspirationGenerator.js
Copyright.js
quotes.js
App.js
Reset.js
import FancyText from './FancyText';
import InspirationGenerator from './InspirationGenerator';
import Copyright from './Copyright';

export default function App() {
  return (
    <>
    <FancyText title="Get Inspired App" />
    <InspirationGenerator>
      <Copyright year={2004} />
    </InspirationGenerator>
  </>
);
}
```

React creates a render tree, a UI tree, composed of the rendered components.

From the example app, we can construct the above render tree.

The tree is composed of nodes, each of which represents a component. App, FancyText, Copyright, to name a few, are all nodes in our tree.

The root node in a React render tree is the root component of the app. In this case, the root component is App and it is the first component React renders. Each arrow in the tree points from a parent component to a child component.

Deep Dive Where are the HTML tags in the render tree? Show Details You'll notice in the above render tree, there is no mention of the HTML tags that each component renders. This is because the render tree is only composed of React components. React, as a UI framework, is platform agnostic. On react.dev, we showcase examples that render to the web, which uses HTML markup as its UI primitives. But a React app could just as likely render to a mobile or desktop platform, which may use different UI primitives like `UIView` or `FrameworkElement`. These platform UI primitives are not a part of React. React render trees can provide insight to our React app regardless of what platform your app renders to.

A render tree represents a single render pass of a React application. With conditional rendering, a parent component may render different children depending on the data passed.

We can update the app to conditionally render either an inspirational quote or color.

```
App.js
FancyText.js
Color.js
InspirationGenerator.js
Copyright.js
inspirations.js
App.js
Reset.js
fork.js
import FancyText from './FancyText';
```

```
import InspirationGenerator from './InspirationGenerator';
import Copyright from './Copyright';
```

```
export default function App() {
  return (
    <>
    <FancyText title="Get Inspired App" />
    <InspirationGenerator>
      <Copyright year={2004} />
    </InspirationGenerator>
  </>
);
}
```

With conditional rendering, across different renders, the render tree may render different components.

In this example, depending on what `inspiration.type` is, we may render `<FancyText>` or `<Color>`. The render tree may be different for each render pass.

Although render trees may differ across render passes, these trees are generally helpful for identifying what the top-level and leaf components are in a React app. Top-level components are the components nearest to the root component and affect the rendering performance of all the components beneath them and often contain the most complexity. Leaf components are near the bottom of the tree and have no child components and are often frequently re-rendered.

Identifying these categories of components are useful for understanding data flow and performance of your app.

The Module Dependency Tree

Another relationship in a React app that can be modeled with a tree are an app's module dependencies. As we break up our components and logic into separate files, we create JS modules where we may export components, functions, or constants.

Each node in a module dependency tree is a module and each branch represents an import statement in that module.

If we take the previous `Inspirations` app, we can build a module dependency tree, or dependency tree for short.

The module dependency tree for the `Inspirations` app.

The root node of the tree is the root module, also known as the entrypoint file. It often is the module that contains the root component.

Comparing to the render tree of the same app, there are similar structures but some notable differences:

The nodes that make-up the tree represent modules, not components.

Non-component modules, like `inspirations.js`, are also represented in this tree. The render tree only encapsulates components.

`Copyright.js` appears under `App.js` but in the render tree, `Copyright`, the component, appears as a child of `InspirationGenerator`. This is because `InspirationGenerator` accepts JSX as children props, so it renders `Copyright` as a child component but does not import the module.

Dependency trees are useful to determine what modules are necessary to run your React app. When building a React app for production, there is typically a build step that will bundle all the necessary JavaScript to ship to the client. The tool responsible for this is called a bundler, and bundlers will use the dependency tree to determine what modules should be included.

As your app grows, often the bundle size does too. Large bundle sizes are expensive for a client to download and run. Large bundle sizes can delay the time for your UI to get drawn. Getting a sense of your app's dependency tree may help with debugging these issues.

Recap

Trees are a common way to represent the relationship between entities. They are often used to model UI.

Render trees represent the nested relationship between React components across a single render.

With conditional rendering, the render tree may change across different renders. With different prop values, components may render different children components.

Render trees help identify what the top-level and leaf components are. Top-level components affect the rendering performance of all components beneath them and leaf components are often re-rendered frequently. Identifying them is useful for understanding and debugging rendering performance.

Dependency trees represent the module dependencies in a React app.

Dependency trees are used by build tools to bundle the necessary code to ship an app.

Dependency trees are useful for debugging large bundle sizes that slow time to paint and expose opportunities for optimizing what code is bundled.

Previous
[Keeping Components Pure](#)
[Next](#)
[Adding Interactivity](#)
©2024 no uwu plzuwu? Logo by [@sawaratsuki1004](#)
Learn React Quick Start
Installation
Describing the UI
Adding Interactivity
Managing State
Escape Hatches
API Reference
React APIs
React DOM APIs
Community
Code of Conduct
Meet the Team
Docs
Contributors
Acknowledgements
More
Blog
React Native
Privacy
Terms
On this page
Overview
Your UI as a tree
The

Render Tree The Module Dependency Tree Recap Responding to Events – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ResetForkexport 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 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 ResetForkfunction AlertButton({ message, children }) {
```

```
  return (  
    <button onClick={() => alert(message)}>  
      {children}  
    </button>  
  );  
}
```

```
export default function Toolbar() {
```

```
  return (  
    <div>  
      <AlertButton message="Playing!">  
        Play Movie  
      </AlertButton>  
      <AlertButton message="Uploading!">  
        Upload Image  
      </AlertButton>  
    </div>  
  );  
}
```

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 ResetForkfunction Button({ onClick, children }) {
```

```
  return (  
    <button onClick={onClick}>
```

```
{children}  
        );  
    }  
  
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 <div> 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 ResetForkfunction Button({ onSmash, children }) {
```

```
  return (
    <button onClick={onSmash}>
      {children}
    </button>
  );
}
```

```
export default function App() {
  return (
    <div>
      <Button onSmash={() => alert('Playing!')}>
        Play Movie
      </Button>
      <Button onSmash={() => alert('Uploading!')}>
        Upload Image
      </Button>
    </div>
  );
}
```

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 ResetForkexport default function App() {
```

```
  return (
```

```
    <Toolbar
```

```
      onPlayMovie={() => alert('Playing!')}
```

```
      onUploadImage={() => alert('Uploading!')}
```

```
    />
```

```
  );
```

```
}
```

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

```
  return (
```

```
    <div>
```

```
      <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.

Note Make sure that you use the appropriate HTML tags for your event handlers. For example, to handle clicks, use <button onClick={handleClick}> instead of <div onClick={handleClick}>. 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 <div> contains two buttons. Both the <div> and each button have their own onClick handlers. Which handlers do you think will fire when you click a button?

```
App.js
App.js 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 <div>'s onClick. So two messages will appear. If you click the toolbar itself, only the parent <div>'s onClick will run.

Pitfall All 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 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 `<div>`'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 `<div>`). 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()} /></div>`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.js
App.js 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.js
App.js 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.js`

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©2024no
uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding
InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of
ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this
pageOverviewAdding event handlers Reading props in event handlers Passing event handlers as props Naming event
handler props Event propagation Stopping propagation Passing handlers as alternative to propagation Preventing
default behavior Can event handlers have side effects? RecapChallengesState: A Component's Memory –
ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe
Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using
TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN
REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX
JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping
Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render
and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in
State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components
Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up
with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs

Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects
Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ResetForkimport { 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>
```

```
  <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

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.

Pitfall Hooks—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.

Note The 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 Dive How does React know which state to return? **Show Details** You 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:

```

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/wldGuZwm.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/AIHTAdDm.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 more You 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©2024no uwu
plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging
StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs
ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewWhen a regular variable
isn't enough Adding a state variable Meet your first Hook Anatomy of useState Giving a component multiple state
variables State is isolated and private RecapChallengesRender and Commit –
ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe
Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using
TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN
REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX
JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping
Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render
and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in
State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components
Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up
with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs
Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects
Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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:

```
index.jsGallery.jsGallery.js ResetForkexport default function Gallery() {
```

```
  return (
```

```
    <section>
```

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

function Image() {
  return (
    
  );
}
```

Show more

During the initial render, React will create the DOM nodes for `<section>`, `<h1>`, and three `` 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 Memory
NextState as a Snapshot
©2024no uwu plzuwu?Logo
by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape
HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs

Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Step 1: Trigger a render Initial render Re-renders when state updates Step 2: React renders your components Step 3: React commits changes to the DOM Epilogue: Browser paint Recap State as a Snapshot – React Reactv18.3.1 Search CtrlK Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Adding Interactivity State as a Snapshot State 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.js
ResetForm
import { 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 your function 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 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.js
import { 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 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 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:<input type="text" value={to}>
      </label>
      <select
        value={to}
        onChange={e => setTo(e.target.value)}>
        <option value="Alice">Alice</option>
        <option value="Bob">Bob</option>
      </select>
    </form>
  );
}
```

```
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 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 ©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewSetting state triggers renders Rendering takes a snapshot in time State over time RecapChallengesQueueing a Series of State Updates –

ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 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

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 ResetFor import { 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 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 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 update returns “replace with 5” 0 (unused) 5 n => n + 1 5 + 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.js
import { useState } from 'react';

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>
  );
}

export default RequestTracker;
```

```
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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReact batches state updates Updating the same state multiple times before the next render What happens if you update state after replacing it What happens if you replace state after updating it Naming conventions RecapChallengesUpdating Objects in State – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 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.js
import { 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 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 ResetFor import { 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 Dive Using a single event handler for multiple fields Show Details You 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.js
```

```
import { 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>
<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 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 Dive Objects are not really nested Show Details An 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 Dive How does Immer work? Show Details The 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:

```
package.json
App.js
package.json
ResetFork{  
  
  "dependencies": {  
    "immer": "1.7.3",  
    "react": "latest",  
    "react-dom": "latest",  
    "react-scripts": "latest",  
    "use-immer": "0.5.1"  
  },  
  "scripts": {  
    "start": "react-scripts start",  
    "build": "react-scripts build",  
    "test": "react-scripts test --env=jsdom",  
    "eject": "react-scripts eject"  
  },  
  "devDependencies": {}  
}
```

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 Dive Why is mutating state not recommended in React? Show Details There 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 challenges! 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.js
import { useState } from 'react';

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

  const handleScoreIncrease = () => {
    setPlayer((prevPlayer) => {
      const newScore = prevPlayer.score + 1;
      return { ...prevPlayer, score: newScore };
    });
  };

  return (
    <div>
      <h1>Scoreboard</h1>
      <p>First Name: {player.firstName}</p>
      <p>Last Name: {player.lastName}</p>
      <p>Score: {player.score}</p>
      <button onClick={handleScoreIncrease}>Increase Score</button>
    </div>
  );
}

export default App;
```

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

  const handleScoreIncrease = () => {
    setPlayer((prevPlayer) => {
      const newScore = prevPlayer.score + 1;
      return { ...prevPlayer, score: newScore };
    });
  };

  return (
    <div>
      <h1>Scoreboard</h1>
      <p>First Name: {player.firstName}</p>
      <p>Last Name: {player.lastName}</p>
      <p>Score: {player.score}</p>
      <button onClick={handleScoreIncrease}>Increase Score</button>
    </div>
  );
}
```

```
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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewWhat's a mutation? Treat state as read-only Copying objects with the spread syntax Updating a nested object Write concise update logic with Immer RecapChallengesUpdating Arrays in State – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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)
push, unshift, concat, [...arr] spread syntax	
(example) removing pop, shift, splice	filter, slice (example) replacing splice, arr[i] = ... assignment
(example) sorting reverse, sort	copy the array first (example)

Alternatively, you can use Immer which lets you use methods from both columns.

Pitfall Unfortunately, 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.js
App.js
ResetFor
import { 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 ResetForimport { 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>
  );
}
```

```

    ]);
} > 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 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 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={{
          position: 'absolute',
          top: `${shape.y}px`,
          left: `${shape.x}px`
```

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

```
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]
```

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 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 myList = [...myList]; const artwork = myList.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 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:

```

package.jsonApp.jsonpackage.json ResetFork{
  "dependencies": {
    "immer": "1.7.3",
    "react": "latest",
  }
}

```

```

  "react-dom": "latest",
  "react-scripts": "latest",
  "use-immer": "0.5.1"
},
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",
  "eject": "react-scripts eject"
},
"devDependencies": {}
}

```

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 challenges
 1. 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.js`
`App.js`
`ResetFor`
`import { 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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewUpdating arrays without mutation Adding to an array Removing from an array Transforming an array Replacing items in an array Inserting into an array Making other changes to an array Updating objects inside arrays Write concise update logic with Immer RecapChallengesReacting to Input with State – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ResetForKexport default function Form({
```

```

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 />
      <br />
      <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 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 inputs Computer inputs Illustrated 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.

Note Notice 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 Dive
Eliminating “impossible” states with a reducer Show Details
These 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.js

```
ResetFormimport { 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 <div>, but adds the picture--active class to the . 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 ResetForkexport default function Picture() {

```
return (
  <div className="background background--active">
    
```

```
</div>
```

```
);
```

```
}
```

Show solutionNext ChallengePreviousManaging StateNextChoosing the State Structure©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewHow declarative UI compares to imperative Thinking about UI declaratively Step 1: Identify your component's different visual states Step 2: Determine what triggers those state changes Step 3: Represent the state in memory with useState Step 4: Remove any non-essential state variables Step 5: Connect the event handlers to set state RecapChallengesChoosing the State Structure – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 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.

Pitfall! If 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.js
import { 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 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 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 ResetForimport { 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 Dive Don't mirror props in state Show Details A 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.js
import { 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 ResetForimport { 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 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
      }
    }
    return item;
  })
}
```

```
...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

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

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 Dive|Improving memory usage Show Details| Ideally, 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.

```
package.json
  "app": "places",
  "fork": "ResetFork"
  "dependencies": {
    "immer": "1.7.3",
    "react": "latest",
    "react-dom": "latest",
    "reselect": "latest"
  },
  "scripts": {
    "start": "node ./src/index.js"
  }
}
```

```
"dependencies": {
  "immer": "1.7.3",
  "react": "latest",
  "react-dom": "latest",
  "reselect": "latest"
},
```

```

"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",
  "eject": "react-scripts eject"
},
"devDependencies": {}
}

```

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©2024no
uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding
InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of
ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this
pageOverviewPrinciples for structuring state Group related state Avoid contradictions in state Avoid redundant state
Avoid duplication in state Avoid deeply nested state RecapChallengesSharing State Between Components –
ReactReactv18.3.1Search⌘CtrlKLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe
Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using
TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN
REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX
JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping
Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render
and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in
State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components
Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up
with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs
Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects
Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ResetForkimport { useState } from 'react';
```

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

```
const [isActive, setIsActive] = useState(false);

return (
  <section className="panel">
    <h3>{title}</h3>
    {isActive ? (
      <p>{children}</p>
    ) : (
      <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 <span lang="kk-KZ">алма</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>
    </>
  );
}
```

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 collapsed Clicking 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 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">алма</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 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 алма, 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 challenges 1. 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.js

```
import { 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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewLifting state up by example Step 1: Remove state from the child components Step 2: Pass hardcoded data from the common parent Step 3: Add state to the common parent A single source of truth for each state RecapChallengesPreserving and Resetting State – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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

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

State is tied to a position in the render tree

React builds render trees for the component structure in your UI.

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 render tree.

Here, there is only one `<Counter />` JSX tag, but it’s rendered at two different positions:

```
App.jsApp.js 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 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 in the tree. 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 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 ResetForkimport { useState } from 'react';
```

```
export default function App() {
  const [isFancy, setIsFancy] = useState(false);

  return (
    <div>
      {isFancy ? (
        <Counter isFancy={true} />
      ) : (
        <Counter isFancy={false} />
      )}
    </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.

Pitfall Remember 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.js`
`App.js`
`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 <div> 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 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 `<div>` 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 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}
      >
    </label>
  );
}
```

```
    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.

Pitfall This is why you should not nest component function definitions. Here, the MyTextField component function is defined inside MyComponent: App.js

```
import { 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>
  </>
);
}
```

```
  
```

```
  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 more Every 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.js
```

```
ResetFor
import { 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.js

```
ResetFor
import { 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 it’s 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 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.

Note Remember 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.js
ContactList.js
Chat.js
App.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 challenges
1. 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.js
import { useState } from 'react';

function App() {
  const [showHint, setShowHint] = useState(false);

  return (
    <div>
      <p><i>Hint: Your favorite city?</i></p>
      <Form />
      <button onClick={() => {
        setShowHint(false);
      }}>Hide hint</button>
    </div>
  );
}

export default App;
```

```
App.js
import { useState } from 'react';

function App() {
  const [showHint, setShowHint] = useState(false);

  return (
    <div>
      <p><i>Hint: Your favorite city?</i></p>
      <Form />
      <button onClick={() => {
        setShowHint(false);
      }}>Hide hint</button>
    </div>
  );
}

export default App;
```

```

        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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewState is tied to a position in the render tree Same component at the same position preserves state Different components at the same position reset state Resetting state at the same position Option 1: Rendering a component in different positions Option 2: Resetting state with a key Resetting a form with a key RecapChallengesExtracting State Logic into a Reducer – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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.js
```

```
ResetForKimport { 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 (
  <>
  <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

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 + 5
The 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.html
tasksReducer.js
index.js
ResetFor
import 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.js
App.js
ResetFor
import { 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:

```
package.json
App.js
package.json
ResetFork{  
  
  "dependencies": {  
    "immer": "1.7.3",  
    "react": "latest",  
    "react-dom": "latest",  
    "react-scripts": "latest",  
    "use-immer": "0.5.1"  
  },  
  "scripts": {  
    "start": "react-scripts start",  
    "build": "react-scripts build",  
    "test": "react-scripts test --env=jsdom",  
    "eject": "react-scripts eject"  
  },  
  "devDependencies": {}  
}
```

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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewConsolidate state logic with a reducer Step 1: Move from setting state to dispatching actions Step 2: Write a reducer function Step 3: Use the reducer from your component Comparing useState and useReducer Writing reducers well Writing concise reducers with Immer RecapChallengesPassing Data Deeply with Context – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 children Using 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.js
Section.js
Heading.js
LevelContext.js
LevelContext.js
ResetFork
import { 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.js
Section.js
Heading.js
LevelContext.js
App.js
ResetFork
import 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>
  );
}
```

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.js Section.js Heading.js LevelContext.js App.js ResetFork import 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>
);
}
```

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.

Note This 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 `<div>` 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.js
Section.js
Heading.js
LevelContext.js
App.js
Reset.js
import 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 `<div>`, 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 `<div>`'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.js`
`Context.js`
`data.js`
`utils.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);
          }}
        />
      </label>
      Use large images
    </>
  
```

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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewThe problem with passing props Context: an alternative to passing props Step 1: Create the context Step 2: Use the context Step 3: Provide the context Using and providing context from the same component Context passes through intermediate components Before you use context Use cases for context RecapChallengesScaling Up with Reducer and Context – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 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>Day off in Kyoto</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: 'Philosopher's Path', done: true },
  { id: 1, text: 'Visit the temple', done: false },
  { id: 2, text: 'Drink matcha', done: false }
];
```

Show 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.js TasksContext.js AddTask.js TaskList.js TasksContext.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.js TasksContext.js AddTask.js TaskList.js App.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.js TasksContext.js AddTask.js TaskList.js TaskList.js ResetFork import { 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: !task.done
          }
        });
      }}
    >
  </label>
);
```

```
...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 (<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.jsResetForkimport { 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©2024no uwu plzuwu?Logo
by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape
HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs
ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewCombining a reducer with
context Step 1: Create the context Step 2: Put state and dispatch into context Step 3: Use context anywhere in the
tree Moving all wiring into a single file RecapReferencing Values with Refs –
ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe
Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using
TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN
REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX
JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping
Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render
and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in
State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components
Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up
with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs
Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects
Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 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 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 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 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 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 Dive How does `useRef` work inside? Show Details Although 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 `<div ref={myRef}>`, React will put the corresponding DOM element into `myRef.current`. Once the element is

removed from the DOM, React will update myRef.current to be null. 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.js
import { useState } from 'react';

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 (
    <div>
      <input type="text" value={text} onChange={e => setText(e.target.value)} />
      <button onClick={handleSend}>Send</button>
      <button onClick={handleUndo}>Undo</button>
    </div>
  );
}

export default Chat;
```

```
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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewAdding a ref to your component Example: building a stopwatch Differences between refs and state When to use refs Best practices for refs Refs and the DOM RecapChallengesManipulating the DOM with Refs – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 <div>, 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 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 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>

```

```

</li>
<li>
  
</li>
<li>
  
</li>
</ul>
</div>
</>
);
}

```

Show more

Deep Dive How to manage a list of refs using a ref callback Show Details In 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:

```

export default function CatFriends() {
  const itemsRef = useRef(null);
  const [catList, setCatList] = useState(setupCatList);

```

```
function scrollToCat(cat) {  
  const map = getMap();  
  const node = map.get(cat);  
  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={() => scrollToCat(catList[0])}>Tom</button>  
    <button onClick={() => scrollToCat(catList[5])}>Maru</button>  
    <button onClick={() => scrollToCat(catList[9])}>Jellylorum</button>  
  </nav>  
  <div>  
    <ul>  
      {catList.map((cat) => (  
        <li  
          key={cat}  
          ref={(node) => {  
            const map = getMap();  
            if (node) {  
              map.set(cat, node);  
            }  
          }}>  
        </li>  
      ))}  
    </ul>  
  </div>  
</div>
```

```

    } else {
      map.delete(cat);
    }
  )}
>
<img src={cat} />
</li>
))}

</ul>
</div>
</>
);

}

```

```

function setupCatList() {
  const catList = [];
  for (let i = 0; i < 10; i++) {
    catList.push("https://loremflickr.com/320/240/cat?lock=" + i);
  }

  return catList;
}

```

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, node); } else { // Remove from the Map map.delete(cat); } }}>This lets you read individual DOM nodes from the Map later.CanyonThis example shows another approach for managing the Map with a ref callback cleanup function.<li key={cat.id} ref={node => { const map = getMap(); // Add to the Map map.set(cat, node); return () => { // Remove from the Map map.delete(cat); }; }}>

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 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 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 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.js

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

function TodoList() {
  const listRef = useRef(null);
  const [text, setText] = useState('');
  const [todos, setTodos] = useState(initialTodos);

  function handleAdd() {
    const newTodo = { id: nextId++, text };
    setText('');
    setTodos([ ...todos, newTodo ]);
    listRef.current.lastChild.scrollIntoView({
      behavior: 'smooth',
      block: 'nearest'
    });
  }

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>{todo.text}</li>
      ))}
    </ul>
  );
}

const initialTodos = [
  { id: 1, text: 'Buy milk' },
  { id: 2, text: 'Buy bread' },
  { id: 3, text: 'Buy eggs' }
];
```

```
export default TodoList;
```

```
const nextId = 4;
```

```

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.js`

```
App.jsimport { useState, useRef } from 'react';
function App() {
  const [text, setText] = useState('');
  const listRef = useRef();
  const todos = [
    { id: 1, text: 'Learn React' },
    { id: 2, text: 'Build a todo app' },
    { id: 3, text: 'Profit' }
  ];
  const handleAdd = () => {
    const newTodo = { id: nextId++, text };
    todos.push(newTodo);
    setTodos([ ...todos, newTodo]);
    listRef.current.lastChild.scrollIntoView();
  };
  const setTodos = (newTodos) => {
    setText('');
    todos.push(...newTodos);
  };
  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>
    </>
  );
}

export default App;
```

```
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 `useState`” 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 `useState`”:

```
App.jsApp.js 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 `<div>` 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 `<div ref={myRef}>`.

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.js`
`App.js`
`import { 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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewGetting a ref to the node Example: Focusing a text input Example: Scrolling to an element Accessing another component’s DOM nodes When React attaches the refs Best practices for DOM manipulation with refs RecapChallengesSynchronizing with Effects – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component’s Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components

Preserving and Resetting State
Extracting State Logic into a Reducer
Passing Data Deeply with Context
Scaling Up with Reducer and Context
Escape Hatches
Referencing Values with Refs
Manipulating the DOM with Refs
Synchronizing with Effects
You Might Not Need an Effect
Lifecycle of Reactive Effects
Separating Events from Effects
Removing Effect Dependencies
Reusing Logic with Custom Hooks
[Is this page useful?](#)
[Learn React](#)
Escape Hatches
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. 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).

Note
Here 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 commit.

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 <div />;}
```

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 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 ResetForKimport { useState, useRef, useEffect } from 'react';
```

```
function VideoPlayer({ src,.isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isFunction) {
      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.

Pitfall By 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 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 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 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 Details
This 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 "tickmark Connecting..." to be printed once in the console. However, if you check the console, "tickmark 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 "tickmark 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:

```
"tickmark Connecting..."
```

```
" Disconnected."
```

```
"tickmark 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 "tickmark 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.

Pitfall Don't use refs to prevent Effects from firing
A common pitfall for preventing Effects firing twice in development is to use a ref to prevent the Effect from running more than once. For example, you could "fix" the above bug with a useRef: `const connectionRef = useRef(null); useEffect(() => { // This wont fix the bug!!! if (!connectionRef.current) { connectionRef.current = createConnection(); connectionRef.current.connect(); } }, []);` This makes it so you only see "tickmark Connecting..." once in development, but it doesn't fix the bug. When the user navigates away, the connection still isn't closed and when they navigate back, a new connection is created. As the user navigates across the app, the connections would keep piling up, the same as it would before the "fix". To fix the bug, it is not enough to just make the Effect run once. The effect needs to work after re-mounting, which means the connection needs to be cleaned up like in the solution above. See the examples below for how to handle common patterns.

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 Dive What are good alternatives to data fetching in Effects? Show Details Writing 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(() => { // 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() { // tickmark 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 a user's perspective, visiting a page shouldn't be different from visiting it, clicking a link, then pressing Back to view the page again. 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 ResetForImport { useState, useEffect } from 'react';
```

```
function Playground() {
  const [text, setText] = useState('a');

  useEffect(() => {
    function onTimeout() {
      console.log('clock ' + text);
    }

    console.log('Schedule "' + text + '" log');
    const timeoutId = setTimeout(onTimeout, 3000);

    return () => {
      console.log('Cancel "' + text + '" log');
      clearTimeout(timeoutId);
    };
  }, [text]);

  return (
    <>
    <label>
      What to log:{' '}
    </label>
    <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 Dive
Each render has its own Effects Show Details
You 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 challenges 1. 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.js MyInput.js ResetFork import { 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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewWhat are Effects and how are they different from events? You might not need an Effect How to write an Effect Step 1: Declare an Effect Step 2: Specify the Effect dependencies Step 3: Add cleanup if needed How to handle the Effect firing twice in development? Controlling non-React widgets Subscribing to events Triggering animations Fetching data Sending analytics Not an Effect: Initializing the application Not an Effect: Buying a product Putting it all together RecapChallengesYou Might Not Need an Effect – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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'); // 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'); // tickmark 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(""); // 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(""); // tickmark 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(() => { // tickmark 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(""); // tickmark 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 Dive How to tell if a calculation is expensive? Show Details In 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(""); // 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 }) { // tickmark 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); // 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); // tickmark 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 }) { // 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 }) { // tickmark 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(""); // tickmark Good: This logic should run because the component was displayed useEffect(() => { post('/analytics/event', { eventName: 'visit_form' }); }, []); // 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(""); // tickmark Good: This logic runs because the component was displayed useEffect(() => { post('/analytics/event', { eventName: 'visit_form' }); }, []); function handleSubmit(e) { e.preventDefault(); // tickmark 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); // 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.

First 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.

The second problem is that 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); // tickmark Calculate what you can during rendering const isGameOver = round > 5; function handlePlaceCard(nextCard) { if (isGameOver) { throw Error('Game already ended.'); } // tickmark 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() { // 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; // tickmark 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. // tickmark 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); // 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) { // tickmark 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:

```
// tickmark Also good: the component is fully controlled by its parent
function 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(); // 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(); // ... // tickmark 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() { // tickmark 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(() => { // 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.js
import { useState, useEffect } from 'react';

Reset
import { 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>
  )
}
```

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

Show more Show hint Show solutionNext ChallengePreviousSynchronizing with EffectsNextLifecycle of Reactive Effects©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewHow to remove unnecessary Effects Updating state based on props or state Caching expensive calculations Resetting all state when a prop changes Adjusting some state when a prop changes Sharing logic between event handlers Sending a POST request Chains of computations Initializing the application Notifying parent components about state changes Passing data to the parent Subscribing to an external store Fetching data RecapChallengesLifecycle of Reactive Effects – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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.

Note Some 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:

tickmark Connecting to "general" room at https://localhost:1234... (development-only)

Disconnected from "general" room at https://localhost:1234. (development-only)

tickmark 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(); }; }, []); // tickmark 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 Dive Can global or mutable values be dependencies? Show Details Mutable 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]); // tickmark 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(); }; }, []); // tickmark 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(); }; }, []); // tickmark 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.

Pitfall The 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(() => { // ... // 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 challenges 1. 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 ResetForImport { 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 ChallengePrevious You Might Not Need an Effect Next Separating Events from Effects ©2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview The lifecycle of an Effect Why synchronization may need to happen more than once How React re-synchronizes your Effect Thinking from the Effect's perspective How React verifies that your Effect can re-synchronize How React knows that it needs to re-synchronize the Effect Each Effect represents a separate synchronization process Effects "react" to reactive values What an Effect with empty dependencies means All variables declared in the component body are reactive React verifies that you specified every reactive value as a dependency What to do when you don't want to re-synchronize Recap Challenges Separating Events from Effects – React React v18.3.1 Search Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Escape Hatches Separating Events from Effects Event 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]); // tickmark 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 Construction This 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]); // tickmark 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); }, []); // 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]); // tickmark 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 numberOfltems = items.length;
useEffect(() => { logVisit(url, numberOfltems); }, [url]); // React Hook useEffect has a missing dependency:
'numberOfltems' // ...}
```

You used numberOfltems 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 numberOfltems. If the user puts something into the shopping cart, and the numberOfltems 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 numberOfltems = items.length;
const onVisit = useEffectEvent(visitedUrl => { logVisit(visitedUrl, numberOfltems); });
useEffect(() => {
onVisit(url);
}, [url]); // tickmark 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.

Note You might be wondering if you could call `onVisit()` with no arguments, and read the `url` inside it: `const onVisit = useEffect(() => { 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 = useEffect(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 = useEffect(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 Dive Is it okay to suppress the dependency linter instead? Show Details In 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); // 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.js
import { 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(() => {
    document.addEventListener('mousemove', handleMove);
    return () => {
      document.removeEventListener('mousemove', handleMove);
    };
  });
}

ResetForKarma
```

```
App.js
import { 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(() => {
    document.addEventListener('mousemove', handleMove);
    return () => {
      document.removeEventListener('mousemove', handleMove);
    };
  });
}
```

```

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.js
```

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

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%',
```

```
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 Construction This 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 = useEffect(() => { setCount(count + 1); });
useTimer(onTick, 1000); // 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 = useEffect(() => { callback(); });
useEffect(() => { const id = setInterval(() => { onTick(); // tickmark 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.js
import { 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);
    };
  }, []);

  return (
    <>
    <h1>
      Counter: {count}
      <button onClick={() => setCount(0)}>Reset</button>
    </h1>
    <hr />
  );
}
```

```
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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewChoosing between event handlers and Effects Event handlers run in response to specific interactions Effects run whenever synchronization is needed Reactive values and reactive logic Logic inside event handlers is not reactive Logic inside Effects is reactive Extracting non-reactive logic out of Effects Declaring an Effect Event Reading latest props and state with Effect Events Limitations of Effect Events RecapChallengesRemoving Effect Dependencies – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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(); // ...})
```

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]); // tickmark 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]); // tickmark 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(); }, []); // 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(); }, []); // tickmark 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.

Pitfall If you have an existing codebase, you might have some Effects that suppress the linter like this:
`useEffect(() => {} // ... // 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 Dive Why is suppressing the dependency linter so dangerous? Show Details
 Suppressing the linter leads to very unintuitive bugs that are hard to find and fix. Here's one example:
`App.js`
`App.js`
`ResetFor`
`import { useState, useEffect } from 'react';`

```

export default function Timer() {

```

```
const [count, setCount] = useState(0);
const [increment, setIncrement] = useState(1);

function onTick() {
    setCount(count + increment);
}

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) { // 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) { // Avoid: Event-specific logic inside an Effect post('/api/register'); showNotification('Successfully registered!', theme); } }, [submitted, theme]); // tickmark 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() { // tickmark 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]); //  
tickmark 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=${country}`)  
.then(response => response.json()) .then(json => { if (!ignore) { setCities(json); } }); // Avoid: A  
single Effect synchronizes two independent processes if (city) { fetch(`/api/areas?city=${city}`)  
.then(response => response.json()) .then(json => { if (!ignore) { setAreas(json); } }); }  
return () => { ignore = true; }, [country, city]); // tickmark 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=${country}`) .then(response => response.json()) .then(json => { if (!ignore) {  
setCities(json); } }); return () => { ignore = true; }, [country]); // tickmark All dependencies declared  
const [city, setCity] = useState(null); const [areas, setAreas] = useState(null); useEffect(() => { if (city) { let ignore  
= false; fetch(`/api/areas?city=${city}`) .then(response => response.json()) .then(json => { if (!ignore) {  
setAreas(json); } }); } }, [city]); // tickmark 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]); // tickmark 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]); // tickmark 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 Construction This 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]); // tickmark 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]); // tickmark 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]); // tickmark 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]); // tickmark 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]); // tickmark 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]); // tickmark 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 render
const options1 = { serverUrl: 'https://localhost:1234', roomId: 'music' }; // During the next
render
const 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(); }, []); // tickmark 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(options); connection.connect(); return () => connection.disconnect(); }, []); // tickmark 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]); // tickmark 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]); // tickmark 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]); // tickmark 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]); // tickmark 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]); // tickmark 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 retrigerring 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.js
```

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

export default function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('tickmark Creating an interval');
    const id = setInterval(() => {
      console.log('clock Interval tick');
      setCount(count + 1);
    }, 1000);
    return () => {
      console.log(' Clearing an interval');
      clearInterval(id);
    };
  }, [count]);
}

return <h1>Counter: {count}</h1>
}
```

```
export default function Timer() {
```

```
  const [count, setCount] = useState(0);
```

```
  useEffect(() => {
```

```
    console.log('tickmark Creating an interval');
```

```
    const id = setInterval(() => {
```

```
      console.log('clock Interval tick');
```

```
      setCount(count + 1);
```

```
    }, 1000);
```

```
    return () => {
```

```
      console.log(' 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©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewDependencies should match the code To remove a dependency, prove that it’s not a dependency To

change the dependencies, change the code Removing unnecessary dependencies Should this code move to an event handler? Is your Effect doing several unrelated things? Are you reading some state to calculate the next state? Do you want to read a value without “reacting” to its changes? Does some reactive value change unintentionally? RecapChallengesReusing Logic with Custom Hooks –

ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?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 ResetForimport { 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 ? 'tickmark Online' : '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 ResetForImport { 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('tickmark 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 ? 'tickmark Online' : 'Disconnected'</h1>;}
function SaveButton() { const isOnline = useOnlineStatus(); function handleSaveClick() {
    console.log('tickmark 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 ? 'tickmark Online' : 'Disconnected'}</h1>

}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log('tickmark 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!

Note if 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 Dive Should all functions called during rendering start with the `use` prefix? Show Details No. 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:// Avoid: A Hook that doesn't use Hooksfunction `useSorted(items)` { `return items.slice().sort();`}// tickmark 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`) { // tickmark 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:// tickmark 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:// tickmark 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 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 Construction This 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]); // tickmark 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]); // tickmark 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.js
ChatRoom.js
useChatRoom.js
chat.js
notifications.js
ChatRoom.js
ResetFork
import { 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);
          }
        });
    }
  }, [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 Dive
Keep your custom Hooks focused on concrete high-level use cases. Show Details Start 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:

tickmark `useData(url)`

tickmark `useImpressionLog(eventName, extraData)`

tickmark `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:

tickmark `useMediaQuery(query)`

tickmark `useSocket(url)`

tickmark `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:

`useMount(fn)`

`useEffectOnce(fn)`

`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'); // Avoid: using custom "lifecycle" Hooks useMount(() => { const connection = createConnection({ roomId, serverUrl }); connection.connect(); post('/analytics/event', { eventName: 'visit_chat' }); // ...}); // Avoid: creating custom "lifecycle" Hooks function useMount(fn) { useEffect(() => { fn(); }, []); // 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'); // tickmark 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'); // tickmark 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 Dive Will React provide any built-in solution for data fetching? Show Details We'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.js
```

```
ResetForK
import { 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 ResetForImport { 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 challenges 1. 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 `App.js` file.
`import { 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©2024no uwu plzuwu?Logo
 by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape
 HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs
 ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewCustom Hooks: Sharing
 logic between components Extracting your own custom Hook from a component Hook names always start with use
 Custom Hooks let you share stateful logic, not state itself Passing reactive values between Hooks Passing event
 handlers to custom Hooks When to use custom Hooks Custom Hooks help you migrate to better patterns There is
 more than one way to do it RecapChallengesReact Reference Overview –
 ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useActionState - This
 feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId
 useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the
 latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>)
<Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext
forwardRef lazy memo startTransition use - This feature is available in the latest
Canaryexperimental_taintObjectReference - This feature is available in the latest
Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks
useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature
is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in
the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest
Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs
createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS
- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule -
This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule -
This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot
Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup
renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls
Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in
the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in
the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the
latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory
createRef isValidElement PureComponent Is this page useful?API ReferenceReact Reference OverviewThis section
provides detailed reference documentation for working with React. For an introduction to React, please visit the
Learn section.

The React reference documentation is broken down into functional subsections:

React

Programmatic React features:

Hooks - Use different React features from your components.

Components - Documents built-in components that you can use in your JSX.

APIs - APIs that are useful for defining components.

Directives - Provide instructions to bundlers compatible with React Server Components.

React DOM

React-dom contains features that are only supported for web applications (which run in the browser DOM environment). This section is broken into the following:

Hooks - Hooks for web applications which run in the browser DOM environment.

Components - React supports all of the browser built-in HTML and SVG components.

APIs - The react-dom package contains methods supported only in web applications.

Client APIs - The react-dom/client APIs let you render React components on the client (in the browser).

Server APIs - The react-dom/server APIs let you render React components to HTML on the server.

Rules of React

React has idioms — or rules — for how to express patterns in a way that is easy to understand and yields high-quality applications:

Components and Hooks must be pure – Purity makes your code easier to understand, debug, and allows React to automatically optimize your components and hooks correctly.

React calls Components and Hooks – React is responsible for rendering components and hooks when necessary to optimize the user experience.

Rules of Hooks – Hooks are defined using JavaScript functions, but they represent a special type of reusable UI logic with restrictions on where they can be called.

Legacy APIs

Legacy APIs - Exported from the react package, but not recommended for use in newly written code.

NextHooks©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReact React DOM Rules of React Legacy APIs useStateActionState –
ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useStateActionState - This feature is available in the latest CanaryuseStateCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>)<Profiler><StrictMode><Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS

- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children cloneElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceHooksuseActionState - This feature is available in the latest CanaryCanaryThe useActionState Hook is currently only available in React's Canary and experimental channels. Learn more about release channels here. In addition, you need to use a framework that supports React Server Components to get the full benefit of useActionState.

NoteIn earlier React Canary versions, this API was part of React DOM and called useFormState.

useActionState is a Hook that allows you to update state based on the result of a form action.`const [state, formAction] = useActionState(fn, initialState, permalink?);`

Reference useActionState(action, initialState, permalink?) Usage Using information returned by a form action Troubleshooting My action can no longer read the submitted form data

Reference

`useActionState(action, initialState, permalink?)`

Call useActionState at the top level of your component to create component state that is updated when a form action is invoked. You pass useActionState an existing form action function as well as an initial state, and it returns a new action that you use in your form, along with the latest form state. The latest form state is also passed to the function that you provided.

```
import { useActionState } from "react";async function increment(previousState, formData) { return previousState + 1;}function StatefulForm({}) { const [state, formAction] = useActionState(increment, 0); return ( <form> {state} <button formAction={formAction}>Increment</button> </form> )}
```

The form state is the value returned by the action when the form was last submitted. If the form has not yet been submitted, it is the initial state that you pass.

If used with a Server Action, useActionState allows the server's response from submitting the form to be shown even before hydration has completed.

See more examples below.

Parameters

`fn`: The function to be called when the form is submitted or button pressed. When the function is called, it will receive the previous state of the form (initially the initialState that you pass, subsequently its previous return value) as its initial argument, followed by the arguments that a form action normally receives.

`initialState`: The value you want the state to be initially. It can be any serializable value. This argument is ignored after the action is first invoked.

`optional permalink`: A string containing the unique page URL that this form modifies. For use on pages with dynamic content (eg: feeds) in conjunction with progressive enhancement: if fn is a server action and the form is submitted

before the JavaScript bundle loads, the browser will navigate to the specified permalink URL, rather than the current page's URL. Ensure that the same form component is rendered on the destination page (including the same action fn and permalink) so that React knows how to pass the state through. Once the form has been hydrated, this parameter has no effect.

Returns

useActionState returns an array with exactly two values:

The current state. During the first render, it will match the initialState you have passed. After the action is invoked, it will match the value returned by the action.

A new action that you can pass as the action prop to your form component or formAction prop to any button component within the form.

Caveats

When used with a framework that supports React Server Components, useActionState lets you make forms interactive before JavaScript has executed on the client. When used without Server Components, it is equivalent to component local state.

The function passed to useActionState receives an extra argument, the previous or initial state, as its first argument. This makes its signature different than if it were used directly as a form action without using useActionState.

Usage

Using information returned by a form action

Call useActionState at the top level of your component to access the return value of an action from the last time a form was submitted.

```
import { useActionState } from 'react'; import { action } from './actions.js'; function MyComponent() { const [state, formAction] = useActionState(action, null); // ... return ( <form action={formAction}> /* ... */ </form> ); }
```

useActionState returns an array with exactly two items:

The current state of the form, which is initially set to the initial state you provided, and after the form is submitted is set to the return value of the action you provided.

A new action that you pass to <form> as its action prop.

When the form is submitted, the action function that you provided will be called. Its return value will become the new current state of the form.

The action that you provide will also receive a new first argument, namely the current state of the form. The first time the form is submitted, this will be the initial state you provided, while with subsequent submissions, it will be

the return value from the last time the action was called. The rest of the arguments are the same as if useActionState had not been used.

```
function action(currentState, formData) { // ... return 'next state';}
```

Display information after submitting a form1. Display form errors 2. Display structured information after submitting a form Example 1 of 2: Display form errors To display messages such as an error message or toast that's returned by a Server Action, wrap the action in a call to useActionState.App.jsactions.jsApp.js ResetForkimport { useActionState, useState } from "react";

```
import { addToCart } from "./actions.js";
```

```
function AddToCartForm({itemID, itemTitle}) {
  const [message, formAction] = useActionState(addToCart, null);
  return (
    <form action={formAction}>
      <h2>{itemTitle}</h2>
      <input type="hidden" name="itemID" value={itemID} />
      <button type="submit">Add to Cart</button>
      {message}
    </form>
  );
}
```

```
export default function App() {
  return (
    <>
      <AddToCartForm itemID="1" itemTitle="JavaScript: The Definitive Guide" />
      <AddToCartForm itemID="2" itemTitle="JavaScript: The Good Parts" />
    </>
  )
}
```

[Show more](#)[Next Example](#)

[Troubleshooting](#)

[My action can no longer read the submitted form data](#)

When you wrap an action with useActionState, it gets an extra argument as its first argument. The submitted form data is therefore its second argument instead of its first as it would usually be. The new first argument that gets added is the current state of the form.

function action(currentState, formData) { // ...}Previous HooksNext useCallback©2024no uwu plzuwu?Logo
by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape
HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs
ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReference
useActionState(action, initialState, permalink?) Usage Using information returned by a form action Troubleshooting
My action can no longer read the submitted form data useCallback –
ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useActionState - This
feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId
useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the
latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>)
<Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext
forwardRef lazy memo startTransition use - This feature is available in the latest
Canaryexperimental_taintObjectReference - This feature is available in the latest
Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks
useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div>) <form> - This feature
is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in
the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest
Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs
createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS
- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule -
This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule -
This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot
Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup
renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls
Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in
the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in
the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the
latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory
createRef isValidElement PureComponent Is this page useful?API ReferenceHooksuseCallbackuseCallback is a React
Hook that lets you cache a function definition between re-renders.const cachedFn = useCallback(fn, dependencies)

Reference useCallback(fn, dependencies) Usage Skipping re-rendering of components Updating state from a
memoized callback Preventing an Effect from firing too often Optimizing a custom Hook Troubleshooting Every time
my component renders, useCallback returns a different function I need to call useCallback for each list item in a loop,
but it's not allowed

Reference

useCallback(fn, dependencies)

Call useCallback at the top level of your component to cache a function definition between re-renders:

```
import { useCallback } from 'react';export default function ProductPage({ productId, referrer, theme }) { const  
handleSubmit = useCallback((orderDetails) => { post('/product/' + productId + '/buy', { referrer, orderDetails,  
}); }, [productId, referrer]);
```

See more examples below.

Parameters

fn: The function value that you want to cache. It can take any arguments and return any values. React will return
(not call!) your function back to you during the initial render. On next renders, React will give you the same function

again if the dependencies have not changed since the last render. Otherwise, it will give you the function that you have passed during the current render, and store it in case it can be reused later. React will not call your function. The function is returned to you so you can decide when and whether to call it.

dependencies: The list of all reactive values referenced inside of the fn code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison algorithm.

Returns

On the initial render, useCallback returns the fn function you have passed.

During subsequent renders, it will either return an already stored fn function from the last render (if the dependencies haven't changed), or return the fn function you have passed during this render.

Caveats

useCallback is a Hook, so you can only call it at the top level of your component or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

React will not throw away the cached function unless there is a specific reason to do that. For example, in development, React throws away the cache when you edit the file of your component. Both in development and in production, React will throw away the cache if your component suspends during the initial mount. In the future, React may add more features that take advantage of throwing away the cache—for example, if React adds built-in support for virtualized lists in the future, it would make sense to throw away the cache for items that scroll out of the virtualized table viewport. This should match your expectations if you rely on useCallback as a performance optimization. Otherwise, a state variable or a ref may be more appropriate.

Usage

Skipping re-rendering of components

When you optimize rendering performance, you will sometimes need to cache the functions that you pass to child components. Let's first look at the syntax for how to do this, and then see in which cases it's useful.

To cache a function between re-renders of your component, wrap its definition into the useCallback Hook:

```
import { useCallback } from 'react';function ProductPage({ productId, referrer, theme }) { const handleSubmit = useCallback((orderDetails) => { post('/product/' + productId + '/buy', { referrer, orderDetails, }); }, [productId, referrer]); // ... }
```

You need to pass two things to useCallback:

A function definition that you want to cache between re-renders.

A list of dependencies including every value within your component that's used inside your function.

On the initial render, the returned function you'll get from `useCallback` will be the function you passed.

On the following renders, React will compare the dependencies with the dependencies you passed during the previous render. If none of the dependencies have changed (compared with `Object.is`), `useCallback` will return the same function as before. Otherwise, `useCallback` will return the function you passed on this render.

In other words, `useCallback` caches a function between re-renders until its dependencies change.

Let's walk through an example to see when this is useful.

Say you're passing a `handleSubmit` function down from the `ProductPage` to the `ShippingForm` component:

```
function ProductPage({ productId, referrer, theme }) { // ... return ( <div className={theme}> <ShippingForm onSubmit={handleSubmit} /> </div> );
```

You've noticed that toggling the `theme` prop freezes the app for a moment, but if you remove `<ShippingForm />` from your JSX, it feels fast. This tells you that it's worth trying to optimize the `ShippingForm` component.

By default, when a component re-renders, React re-renders all of its children recursively. This is why, when `ProductPage` re-renders with a different theme, the `ShippingForm` component also re-renders. This is fine for components that don't require much calculation to re-render. But if you verified a re-render is slow, you can tell `ShippingForm` to skip re-rendering when its props are the same as on last render by wrapping it in `memo`:

```
import { memo } from 'react';const ShippingForm = memo(function ShippingForm({ onSubmit }) { // ...});
```

With this change, `ShippingForm` will skip re-rendering if all of its props are the same as on the last render. This is when caching a function becomes important! Let's say you defined `handleSubmit` without `useCallback`:

```
function ProductPage({ productId, referrer, theme }) { // Every time the theme changes, this will be a different function... function handleSubmit(orderDetails) { post('/product/' + productId + '/buy', { referrer, orderDetails, }); } return ( <div className={theme}> /* ... so ShippingForm's props will never be the same, and it will re-render every time */ <ShippingForm onSubmit={handleSubmit} /> </div> );}
```

In JavaScript, a function `() {}` or `() => {}` always creates a different function, similar to how the `{}` object literal always creates a new object. Normally, this wouldn't be a problem, but it means that `ShippingForm` props will never be the same, and your `memo` optimization won't work. This is where `useCallback` comes in handy:

```
function ProductPage({ productId, referrer, theme }) { // Tell React to cache your function between re-renders... const handleSubmit = useCallback((orderDetails) => { post('/product/' + productId + '/buy', { referrer, orderDetails, }); }, [productId, referrer]); // ...so as long as these dependencies don't change... return ( <div className={theme}> /* ...ShippingForm will receive the same props and can skip re-rendering */ <ShippingForm onSubmit={handleSubmit} /> </div> );}
```

By wrapping `handleSubmit` in `useCallback`, you ensure that it's the same function between the re-renders (until dependencies change). You don't have to wrap a function in `useCallback` unless you do it for some specific reason. In this example, the reason is that you pass it to a component wrapped in `memo`, and this lets it skip re-rendering. There are other reasons you might need `useCallback` which are described further on this page.

Note You should only rely on `useCallback` as a performance optimization. If your code doesn't work without it, find the underlying problem and fix it first. Then you may add `useCallback` back.

Deep Dive How is `useCallback` related to `useMemo`? Show Details You will often see `useMemo` alongside `useCallback`. They are both useful when you're trying to optimize a child component. They let you memoize (or, in other words, cache) something you're passing down:
`import { useMemo, useCallback } from 'react';function ProductPage({ productId, referrer }) { const product = useState('/product/' + productId); const requirements = useMemo(() => { // Calls your function and caches its result return computeRequirements(product); }, [product]); const handleSubmit = useCallback((orderDetails) => { post('/product/' + productId + '/buy', { referrer, orderDetails, }); }, [productId, referrer]); return (<div> <h1>{product}</h1> <p>{requirements}</p> <button onClick={handleSubmit}>Buy</button> </div>);}`

```
= useCallback((orderDetails) => { // Caches your function itself post('/product/' + productId + '/buy', { referrer, orderDetails, }); }, [productId, referrer]); return ( <div className={theme}> <ShippingForm requirements={requirements} onSubmit={handleSubmit} /> </div> );}The difference is in what they're letting you cache:
```

useMemo caches the result of calling your function. In this example, it caches the result of calling computeRequirements(product) so that it doesn't change unless product has changed. This lets you pass the requirements object down without unnecessarily re-rendering ShippingForm. When necessary, React will call the function you've passed during rendering to calculate the result.

useCallback caches the function itself. Unlike useMemo, it does not call the function you provide. Instead, it caches the function you provided so that handleSubmit itself doesn't change unless productId or referrer has changed. This lets you pass the handleSubmit function down without unnecessarily re-rendering ShippingForm. Your code won't run until the user submits the form.

If you're already familiar with useMemo, you might find it helpful to think of useCallback as this:// Simplified implementation (inside React)function useCallback(fn, dependencies) { return useMemo(() => fn, dependencies); }Read more about the difference between useMemo and useCallback.

Deep DiveShould you add useCallback everywhere? Show DetailsIf your app is like this site, and most interactions are coarse (like replacing a page or an entire section), memoization is usually unnecessary. On the other hand, if your app is more like a drawing editor, and most interactions are granular (like moving shapes), then you might find memoization very helpful.Caching a function with useCallback is only valuable in a few cases:

You pass it as a prop to a component wrapped in memo. You want to skip re-rendering if the value hasn't changed. Memoization lets your component re-render only if dependencies changed.

The function you're passing is later used as a dependency of some Hook. For example, another function wrapped in useCallback depends on it, or you depend on this function from useEffect.

There is no benefit to wrapping a function in useCallback in other cases. There is no significant harm to doing that either, so some teams choose to not think about individual cases, and memoize as much as possible. The downside is that code becomes less readable. Also, not all memoization is effective: a single value that's "always new" is enough to break memoization for an entire component. Note that useCallback does not prevent creating the function. You're always creating a function (and that's fine!), but React ignores it and gives you back a cached function if nothing changed.In practice, you can make a lot of memoization unnecessary by following a few principles:

When a component visually wraps other components, let it accept JSX as children. Then, if the wrapper component updates its own state, React knows that its children don't need to re-render.

Prefer local state and don't lift state up any further than necessary. Don't keep transient state like forms and whether an item is hovered at the top of your tree or in a global state library.

Keep your rendering logic pure. If re-rendering a component causes a problem or produces some noticeable visual artifact, it's a bug in your component! Fix the bug instead of adding memoization.

Avoid unnecessary Effects that update state. Most performance problems in React apps are caused by chains of updates originating from Effects that cause your components to render over and over.

Try to remove unnecessary dependencies from your Effects. For example, instead of memoization, it's often simpler to move some object or a function inside an Effect or outside the component.

If a specific interaction still feels laggy, use the React Developer Tools profiler to see which components benefit the most from memoization, and add memoization where needed. These principles make your components easier to debug and understand, so it's good to follow them in any case. In long term, we're researching doing memoization automatically to solve this once and for all.

The difference between useCallback and declaring a function directly1. Skipping re-rendering with useCallback and memo 2. Always re-rendering a component Example 1 of 2: Skipping re-rendering with useCallback and memo In this

example, the `ShippingForm` component is artificially slowed down so that you can see what happens when a React component you're rendering is genuinely slow. Try incrementing the counter and toggling the theme. Incrementing the counter feels slow because it forces the slowed down `ShippingForm` to re-render. That's expected because the counter has changed, and so you need to reflect the user's new choice on the screen. Next, try toggling the theme. Thanks to `useCallback` together with `memo`, it's fast despite the artificial slowdown! `ShippingForm` skipped re-rendering because the `handleSubmit` function has not changed. The `handleSubmit` function has not changed because both `productId` and `referrer` (your `useCallback` dependencies) haven't changed since last render.

```
App.jsProductPage.jsShippingForm.jsProductPage.js ResetForKimport { useCallback } from 'react';
```

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

```
export default function ProductPage({ productId, referrer, theme }) {
```

```
  const handleSubmit = useCallback((orderDetails) => {
```

```
    post('/product/' + productId + '/buy', {
```

```
      referrer,
```

```
      orderDetails,
```

```
    });
  }, [productId, referrer]);
```

```
  return (
```

```
    <div className={theme}>
```

```
      <ShippingForm onSubmit={handleSubmit} />
```

```
    </div>
```

```
  );
```

```
}
```

```
function post(url, data) {
```

```
  // Imagine this sends a request...
```

```
  console.log('POST /' + url);
```

```
  console.log(data);
```

```
}
```

Show more [Next Example](#)

Updating state from a memoized callback

Sometimes, you might need to update state based on previous state from a memoized callback.

This `handleAddTodo` function specifies `todos` as a dependency because it computes the next todos from it:

```
function TodoList() { const [todos, setTodos] = useState([]); const handleAddTodo = useCallback((text) => { const newTodo = { id: nextId++, text }; setTodos([...todos, newTodo]); }, [todos]); // ...}
```

You'll usually want memoized functions to have as few dependencies as possible. When you read some state only to calculate the next state, you can remove that dependency by passing an updater function instead:

```
function TodoList() { const [todos, setTodos] = useState([]); const handleAddTodo = useCallback((text) => { const newTodo = { id: nextId++, text }; setTodos(todos => [...todos, newTodo]); }, []); // tickmark No need for the todos dependency // ...}
```

Here, instead of making todos a dependency and reading it inside, you pass an instruction about how to update the state (todos => [...todos, newTodo]) to React. Read more about [updater functions](#).

Preventing an Effect from firing too often

Sometimes, you might want to call a function from inside an Effect:

```
function ChatRoom({ roomId }) { const [message, setMessage] = useState(""); function createOptions() { return { serverUrl: 'https://localhost:1234', roomId: roomId }; } useEffect(() => { const options = createOptions(); const connection = createConnection(); connection.connect(); // ... }) }
```

This creates a problem. Every reactive value must be declared as a dependency of your Effect. However, if you declare createOptions as a dependency, it will cause your Effect to constantly reconnect to the chat room:

```
useEffect(() => { const options = createOptions(); const connection = createConnection(); connection.connect(); return () => connection.disconnect(); }, [createOptions]); // Problem: This dependency changes on every render // ...
```

To solve this, you can wrap the function you need to call from an Effect into `useCallback`:

```
function ChatRoom({ roomId }) { const [message, setMessage] = useState(""); const createOptions = useCallback(() => { return { serverUrl: 'https://localhost:1234', roomId: roomId }; }, [roomId]); // tickmark Only changes when roomId changes useEffect(() => { const options = createOptions(); const connection = createConnection(); connection.connect(); return () => connection.disconnect(); }, [createOptions]); // tickmark Only changes when createOptions changes // ... }) }
```

This ensures that the `createOptions` function is the same between re-renders if the `roomId` is the same. However, it's even better to remove the need for a function dependency. Move your function inside the Effect:

```
function ChatRoom({ roomId }) { const [message, setMessage] = useState(""); useEffect(() => { function createOptions() { // tickmark No need for useCallback or function dependencies! return { serverUrl: 'https://localhost:1234', roomId: roomId }; } const options = createOptions(); const connection = createConnection(); connection.connect(); return () => connection.disconnect(); }, [roomId]); // tickmark Only changes when roomId changes // ... }) }
```

Now your code is simpler and doesn't need `useCallback`. Learn more about [removing Effect dependencies](#).

Optimizing a custom Hook

If you're writing a custom Hook, it's recommended to wrap any functions that it returns into `useCallback`:

```
function useRouter() { const { dispatch } = useContext(RouterStateContext); const navigate = useCallback((url) => { dispatch({ type: 'navigate', url }); }, [dispatch]); const goBack = useCallback(() => { dispatch({ type: 'back' }); }, [dispatch]); return { navigate, goBack, }; }
```

This ensures that the consumers of your Hook can optimize their own code when needed.

Troubleshooting

Every time my component renders, useCallback returns a different function

Make sure you've specified the dependency array as a second argument!

If you forget the dependency array, useCallback will return a new function every time:

```
function ProductPage({ productId, referrer }) { const handleSubmit = useCallback((orderDetails) => {
post('/product/' + productId + '/buy', { referrer, orderDetails, }); // Returns a new function every time: no
dependency array // ...
```

This is the corrected version passing the dependency array as a second argument:

```
function ProductPage({ productId, referrer }) { const handleSubmit = useCallback((orderDetails) => {
post('/product/' + productId + '/buy', { referrer, orderDetails, }), [productId, referrer]); // tickmark Does not
return a new function unnecessarily // ...
```

If this doesn't help, then the problem is that at least one of your dependencies is different from the previous render. You can debug this problem by manually logging your dependencies to the console:

```
const handleSubmit = useCallback((orderDetails) => { // .. }, [productId, referrer]); console.log([productId,
referrer]);
```

You can then right-click on the arrays from different re-renders in the console and select "Store as a global variable" for both of them. Assuming the first one got saved as temp1 and the second one got saved as temp2, you can then use the browser console to check whether each dependency in both arrays is the same:

```
Object.is(temp1[0], temp2[0]); // Is the first dependency the same between the arrays?Object.is(temp1[1],
temp2[1]); // Is the second dependency the same between the arrays?Object.is(temp1[2], temp2[2]); // ... and so on
for every dependency ...
```

When you find which dependency is breaking memoization, either find a way to remove it, or memoize it as well.

I need to call useCallback for each list item in a loop, but it's not allowed

Suppose the Chart component is wrapped in memo. You want to skip re-rendering every Chart in the list when the ReportList component re-renders. However, you can't call useCallback in a loop:

```
function ReportList({ items }) { return ( <article> {items.map(item => { // You can't call useCallback in a loop
like this: const handleClick = useCallback(() => { sendReport(item) }, [item]); return ( <figure
key={item.id}> <Chart onClick={handleClick} /> </figure> ); })} </article> );}
```

Instead, extract a component for an individual item, and put useCallback there:

```
function ReportList({ items }) { return ( <article> {items.map(item => <Report key={item.id} item={item} />
}) </article> );}function Report({ item }) { // tickmark Call useCallback at the top level: const handleClick =
useCallback(() => { sendReport(item) }, [item]); return ( <figure> <Chart onClick={handleClick} /> </figure>
);}
```

Alternatively, you could remove useCallback in the last snippet and instead wrap Report itself in memo. If the item prop does not change, Report will skip re-rendering, so Chart will skip re-rendering too:

```
function ReportList({ items }) { // ...}const Report = memo(function Report({ item }) { function handleClick() {
sendReport(item); } return ( <figure> <Chart onClick={handleClick} /> </figure>
);});PrevioususeActionStateNextuseContext©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick
StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact
DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact
NativePrivacyTermsOn this pageOverviewReference useCallback(fn, dependencies) Usage Skipping re-rendering of
```

components Updating state from a memoized callback Preventing an Effect from firing too often Optimizing a custom Hook Troubleshooting Every time my component renders, useCallback returns a different function I need to call useCallback for each list item in a loop, but it's not allowed useContext –

ReactReactv18.3.1SearchCtrlLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useState - This feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceHooksuseContextuseContext is a React Hook that lets you read and subscribe to context from your component const value = useContext(SomeContext)

Reference useContext(SomeContext) Usage Passing data deeply into the tree Updating data passed via context Specifying a fallback default value Overriding context for a part of the tree Optimizing re-renders when passing objects and functions Troubleshooting My component doesn't see the value from my provider I am always getting undefined from my context although the default value is different

Reference

`useContext(SomeContext)`

Call useContext at the top level of your component to read and subscribe to context.

```
import { useContext } from 'react';function MyComponent() { const theme = useContext(ThemeContext); // ...}
```

See more examples below.

Parameters

`SomeContext`: The context that you've previously created with createContext. The context itself does not hold the information, it only represents the kind of information you can provide or read from components.

Returns

useContext returns the context value for the calling component. It is determined as the value passed to the closest SomeContext.Provider above the calling component in the tree. If there is no such provider, then the returned value

will be the defaultValue you have passed to createContext for that context. The returned value is always up-to-date. React automatically re-renders components that read some context if it changes.

Caveats

useContext() call in a component is not affected by providers returned from the same component. The corresponding <Context.Provider> needs to be above the component doing the useContext() call.

React automatically re-renders all the children that use a particular context starting from the provider that receives a different value. The previous and the next values are compared with the Object.is comparison. Skipping re-renders with memo does not prevent the children receiving fresh context values.

If your build system produces duplicates modules in the output (which can happen with symlinks), this can break context. Passing something via context only works if SomeContext that you use to provide context and SomeContext that you use to read it are exactly the same object, as determined by a === comparison.

Usage

Passing data deeply into the tree

Call useContext at the top level of your component to read and subscribe to context.

```
import { useContext } from 'react';function Button() { const theme = useContext(ThemeContext); // ...
```

useContext returns the context value for the context you passed. To determine the context value, React searches the component tree and finds the closest context provider above for that particular context.

To pass context to a Button, wrap it or one of its parent components into the corresponding context provider:

```
function MyPage() { return ( <ThemeContext.Provider value="dark"> <Form /> </ThemeContext.Provider> ); }function Form() { // ... renders buttons inside ... }
```

It doesn't matter how many layers of components there are between the provider and the Button. When a Button anywhere inside of Form calls useContext(ThemeContext), it will receive "dark" as the value.

PitfalluseContext() always looks for the closest provider above the component that calls it. It searches upwards and does not consider providers in the component from which you're calling useContext().

```
App.jsApp.js ResetForKimport { createContext, useContext } from 'react';
```

```
const ThemeContext = createContext(null);
```

```
export default function MyApp() {
```

```
    return (
```

```
        <ThemeContext.Provider value="dark">
```

```
            <Form />
```

```
        </ThemeContext.Provider>
```

```
    )
```

```
}
```

```
function Form() {
  return (
    <Panel title="Welcome">
      <Button>Sign up</Button>
      <Button>Log in</Button>
    </Panel>
  );
}
```

```
function Panel({ title, children }) {
  const theme = useContext(ThemeContext);
  const className = 'panel-' + theme;
  return (
    <section className={className}>
      <h1>{title}</h1>
      {children}
    </section>
  )
}
```

```
function Button({ children }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;
  return (
    <button className={className}>
      {children}
    </button>
  );
}
```

Show more

Updating data passed via context

Often, you'll want the context to change over time. To update context, combine it with state. Declare a state variable in the parent component, and pass the current state down as the context value to the provider.

```
function MyPage() { const [theme, setTheme] = useState('dark'); return ( <ThemeContext.Provider value={theme}> <Form /> <Button onClick={() => { setTheme('light'); }}> Switch to light theme </Button> </ThemeContext.Provider> );}
```

Now any Button inside of the provider will receive the current theme value. If you call setTheme to update the theme value that you pass to the provider, all Button components will re-render with the new 'light' value.

Examples of updating context1. Updating a value via context 2. Updating an object via context 3. Multiple contexts 4. Extracting providers to a component 5. Scaling up with context and a reducer Example 1 of 5: Updating a value via context In this example, the MyApp component holds a state variable which is then passed to the ThemeContext provider. Checking the "Dark mode" checkbox updates the state. Changing the provided value re-renders all the components using that context.

```
App.js
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={theme}>
      <Form />
      <label>
        <input
          type="checkbox"
          checked={theme === 'dark'}
          onChange={(e) => {
            setTheme(e.target.checked ? 'dark' : 'light')
          }}
        />
        Use dark mode
      </label>
    </ThemeContext.Provider>
  )
}

function Form({ children }) {
  return (
    <Panel title="Welcome">
```

```
const ThemeContext = createContext(null);
```

```
export default function MyApp() {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={theme}>
      <Form />
      <label>
        <input
          type="checkbox"
          checked={theme === 'dark'}
          onChange={(e) => {
            setTheme(e.target.checked ? 'dark' : 'light')
          }}
        />
        Use dark mode
      </label>
    </ThemeContext.Provider>
  )
}
```

```
function Form({ children }) {
  return (
    <Panel title="Welcome">
```

```

<Button>Sign up</Button>
<Button>Log in</Button>
</Panel>
);

}

function Panel({ title, children }) {
  const theme = useContext(ThemeContext);
  const className = 'panel-' + theme;
  return (
    <section className={className}>
      <h1>{title}</h1>
      {children}
    </section>
  )
}

function Button({ children }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;
  return (
    <button className={className}>
      {children}
    </button>
  );
}

```

Show moreNote that value="dark" passes the "dark" string, but value={theme} passes the value of the JavaScript theme variable with JSX curly braces. Curly braces also let you pass context values that aren't strings.Next Example

Specifying a fallback default value

If React can't find any providers of that particular context in the parent tree, the context value returned by useContext() will be equal to the default value that you specified when you created that context:

```
const ThemeContext = createContext(null);
```

The default value never changes. If you want to update context, use it with state as described above.

Often, instead of null, there is some more meaningful value you can use as a default, for example:

```
const ThemeContext = createContext('light');
```

This way, if you accidentally render some component without a corresponding provider, it won't break. This also helps your components work well in a test environment without setting up a lot of providers in the tests.

In the example below, the "Toggle theme" button is always light because it's outside any theme context provider and the default context theme value is 'light'. Try editing the default theme to be 'dark'.

```
App.jsApp.js ResetForkimport { createContext, useContext, useState } from 'react';
```

```
const ThemeContext = createContext('light');
```

```
export default function MyApp() {
  const [theme, setTheme] = useState('light');

  return (
    <>
      <ThemeContext.Provider value={theme}>
        <Form />
      </ThemeContext.Provider>
      <Button onClick={() => {
        setTheme(theme === 'dark' ? 'light' : 'dark');
      }}>
        Toggle theme
      </Button>
    </>
  )
}
```

```
function Form({ children }) {
  return (
    <Panel title="Welcome">
      <Button>Sign up</Button>
      <Button>Log in</Button>
    </Panel>
  );
}
```

```

function Panel({ title, children }) {
  const theme = useContext(ThemeContext);
  const className = 'panel-' + theme;
  return (
    <section className={className}>
      <h1>{title}</h1>
      {children}
    </section>
  )
}

```

```

function Button({ children, onClick }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;
  return (
    <button className={className} onClick={onClick}>
      {children}
    </button>
  );
}

```

[Show more](#)

Overriding context for a part of the tree

You can override the context for a part of the tree by wrapping that part in a provider with a different value.

```

<ThemeContext.Provider value="dark"> ... <ThemeContext.Provider value="light"> <Footer />
</ThemeContext.Provider> ...</ThemeContext.Provider>

```

You can nest and override providers as many times as you need.

Examples of overriding context1. Overriding a theme 2. Automatically nested headings Example 1 of 2: Overriding a theme Here, the button inside the Footer receives a different context value ("light") than the buttons outside ("dark").App.jsApp.js ResetForkimport { createContext, useContext } from 'react';

```
const ThemeContext = createContext(null);
```

```
export default function MyApp() {
```

```
return (

<ThemeContext.Provider value="dark">

<Form />

</ThemeContext.Provider>

)

}
```

```
function Form() {

return (

<Panel title="Welcome">

<Button>Sign up</Button>

<Button>Log in</Button>

<ThemeContext.Provider value="light">

<Footer />

</ThemeContext.Provider>

</Panel>

);

}
```

```
function Footer() {

return (

<footer>

<Button>Settings</Button>

</footer>

);

}
```

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

const theme = useContext(ThemeContext);

const className = 'panel-' + theme;

return (

<section className={className}>

{title && <h1>{title}</h1>}

{children}


```

```

        </section>
    )
}

function Button({ children }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;
  return (
    <button className={className}>
      {children}
    </button>
  );
}

```

[Show more](#)[Next Example](#)

Optimizing re-renders when passing objects and functions

You can pass any values via context, including objects and functions.

```
function MyApp() { const [currentUser, setCurrentUser] = useState(null); function login(response) {
  storeCredentials(response.credentials); setCurrentUser(response.user); } return ( <AuthContext.Provider
  value={{ currentUser, login }}> <Page /> </AuthContext.Provider> );}
```

Here, the context value is a JavaScript object with two properties, one of which is a function. Whenever `MyApp` re-renders (for example, on a route update), this will be a different object pointing at a different function, so React will also have to re-render all components deep in the tree that call `useContext(AuthContext)`.

In smaller apps, this is not a problem. However, there is no need to re-render them if the underlying data, like `currentUser`, has not changed. To help React take advantage of that fact, you may wrap the `login` function with `useCallback` and wrap the object creation into `useMemo`. This is a performance optimization:

```
import { useCallback, useMemo } from 'react';function MyApp() { const [currentUser, setCurrentUser] =
  useState(null); const login = useCallback((response) => { storeCredentials(response.credentials);
  setCurrentUser(response.user); }, []); const contextValue = useMemo(() => ({ currentUser, login }), [currentUser,
  login]); return ( <AuthContext.Provider value={contextValue}> <Page /> </AuthContext.Provider> );}
```

As a result of this change, even if `MyApp` needs to re-render, the components calling `useContext(AuthContext)` won't need to re-render unless `currentUser` has changed.

[Read more about `useMemo` and `useCallback`.](#)

Troubleshooting

My component doesn't see the value from my provider

There are a few common ways that this can happen:

You're rendering `<SomeContext.Provider>` in the same component (or below) as where you're calling `useContext()`. Move `<SomeContext.Provider>` above and outside the component calling `useContext()`.

You may have forgotten to wrap your component with `<SomeContext.Provider>`, or you might have put it in a different part of the tree than you thought. Check whether the hierarchy is right using React DevTools.

You might be running into some build issue with your tooling that causes `SomeContext` as seen from the providing component and `SomeContext` as seen by the reading component to be two different objects. This can happen if you use symlinks, for example. You can verify this by assigning them to globals like `window.SomeContext1` and `window.SomeContext2` and then checking whether `window.SomeContext1 === window.SomeContext2` in the console. If they're not the same, fix that issue on the build tool level.

I am always getting undefined from my context although the default value is different

You might have a provider without a value in the tree:

```
// Doesn't work: no value prop<ThemeContext.Provider> <Button /></ThemeContext.Provider>
```

If you forget to specify value, it's like passing `value={undefined}`.

You may have also mistakenly used a different prop name by mistake:

```
// Doesn't work: prop should be called "value"<ThemeContext.Provider theme={theme}> <Button /></ThemeContext.Provider>
```

In both of these cases you should see a warning from React in the console. To fix them, call the prop value:

```
// tickmark Passing the value prop<ThemeContext.Provider value={theme}> <Button /></ThemeContext.Provider>
```

Note that the default value from your `createContext(defaultValue)` call is only used if there is no matching provider above at all. If there is a `<SomeContext.Provider value={undefined}>` component somewhere in the parent tree, the component calling `useContext(SomeContext)` will receive undefined as the context

value.PrevioususeCallbackNextuseDebugValue©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact

DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReference useContext(SomeContext) Usage Passing data deeply into the tree Updating data passed via context Specifying a fallback default value Overriding context for a part of the tree Optimizing re-renders when passing objects and functions Troubleshooting My component doesn't see the value from my provider I am always getting undefined from my context although the default value is different

useDebugValue – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks

useActionState - This feature is available in the latest CanaryuseCallback useContext useDebugValue

useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic

- This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition

Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest

Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks

useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature

is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs

createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS

- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule -

This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule -

This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot

Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup
renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls
Components and Hooks Rules of Hooks React Server Components Server Components - This feature is available in
the latest Canary Server Actions - This feature is available in the latest Canary Directives - This feature is available in
the latest Canary 'use client' - This feature is available in the latest Canary 'use server' - This feature is available in the
latest Canary Legacy APIs Legacy React APIs Children cloneElement Component createElement createFactory
createRef isValidElement PureComponent Is this page useful? API Reference Hooks useDebugValue useDebugValue is a
React Hook that lets you add a label to a custom Hook in React DevTools. useDebugValue(value, format?)

Reference useDebugValue(value, format?) Usage Adding a label to a custom Hook Deferring formatting of a debug
value

Reference

useDebugValue(value, format?)

Call useDebugValue at the top level of your custom Hook to display a readable debug value:

```
import { useDebugValue } from 'react';function useOnlineStatus() { // ... useDebugValue(isOnline ? 'Online' :  
'Offline'); // ...}
```

See more examples below.

Parameters

value: The value you want to display in React DevTools. It can have any type.

optional format: A formatting function. When the component is inspected, React DevTools will call the formatting
function with the value as the argument, and then display the returned formatted value (which may have any type).
If you don't specify the formatting function, the original value itself will be displayed.

Returns

useDebugValue does not return anything.

Usage

Adding a label to a custom Hook

Call useDebugValue at the top level of your custom Hook to display a readable debug value for React DevTools.

```
import { useDebugValue } from 'react';function useOnlineStatus() { // ... useDebugValue(isOnline ? 'Online' :  
'Offline'); // ...}
```

This gives components calling useOnlineStatus a label like OnlineStatus: "Online" when you inspect them:

Without the useDebugValue call, only the underlying data (in this example, true) would be displayed.

```
App.jsuseOnlineStatus.jsuseOnlineStatus.js ResetForkimport { useSyncExternalStore, useDebugValue } from 'react';
```

```
export function useOnlineStatus() {
```

```
  const isOnline = useSyncExternalStore(subscribe, () => navigator.onLine, () => true);
```

```
useDebugValue(isOnline ? 'Online' : 'Offline');

return isOnline;
}

function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);

  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
}
```

Show more

NoteDon't add debug values to every custom Hook. It's most valuable for custom Hooks that are part of shared libraries and that have a complex internal data structure that's difficult to inspect.

Deferring formatting of a debug value

You can also pass a formatting function as the second argument to `useDebugValue`:

```
useDebugValue(date, date => date.toDateString());
```

Your formatting function will receive the debug value as a parameter and should return a formatted display value. When your component is inspected, React DevTools will call this function and display its result.

This lets you avoid running potentially expensive formatting logic unless the component is actually inspected. For example, if `date` is a `Date` value, this avoids calling `toDateString()` on it for every `render`. Previous use Context Next `useDeferredValue` © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Reference `useDebugValue` (value, format?) Usage Adding a label to a custom Hook Deferring formatting of a debug value `useDeferredValue` – React React v18.3.1 Search ⌂ Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks `useActionState` - This feature is available in the latest Canary `useCallback` `useContext` `useDebugValue` `useDeferredValue` `useEffect` `useId` `useImperativeHandle` `useInsertionEffect` `useLayoutEffect` `useMemo` `useOptimistic` - This feature is available in the latest Canary `useReducer` `useRef` `useState` `useSyncExternalStore` `useTransition` Components `<Fragment>` (`<>`) `<Profiler>` `<StrictMode>` `<Suspense>` APIs act cache - This feature is available in the latest Canary `createContext` `forwardRef` `lazy` `memo` `startTransition` `use` - This feature is available in the latest Canary `experimental_taintObjectReference` - This feature is available in the latest Canary `experimental_taintUniqueValue` - This feature is available in the latest Canary `react-dom@18.3.1` Hooks `useFormStatus` - This feature is available in the latest Canary Components Common (e.g. `<div>`) `<form>` - This feature is available in the latest Canary `<input>` `<option>` `<progress>` `<select>` `<textarea>` `<link>` - This feature is available in the latest Canary `<meta>` - This feature is available in the latest Canary `<script>` - This feature is available in the latest Canary `<style>` - This feature is available in the latest Canary `<title>` - This feature is available in the latest Canary APIs `createPortal` `flushSync` `findDOMNode` `hydrate` `preconnect` - This feature is available in the latest Canary `prefetchDNS`

- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children cloneElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceHooksuseDeferredValueuseDeferredValue is a React Hook that lets you defer updating a part of the UI.`const deferredValue = useDeferredValue(value)`

Reference useDeferredValue(value, initialValue?) Usage Showing stale content while fresh content is loading Indicating that the content is stale Deferring re-rendering for a part of the UI

Reference

`useDeferredValue(value, initialValue?)`

Call `useDeferredValue` at the top level of your component to get a deferred version of that value.

```
import { useState, useDeferredValue } from 'react';function SearchPage() { const [query, setQuery] = useState(""); const deferredQuery = useDeferredValue(query); // ...}
```

See more examples below.

Parameters

`value`: The value you want to defer. It can have any type.

`Canary` only optional `initialValue`: A value to use during the initial render of a component. If this option is omitted, `useDeferredValue` will not defer during the initial render, because there's no previous version of value that it can render instead.

Returns

`currentValue`: During the initial render, the returned deferred value will be the same as the value you provided. During updates, React will first attempt a re-render with the old value (so it will return the old value), and then try another re-render in the background with the new value (so it will return the updated value).

`Canary`In the latest React Canary versions, `useDeferredValue` returns the `initialValue` on initial render, and schedules a re-render in the background with the value returned.

Caveats

When an update is inside a Transition, `useDeferredValue` always returns the new value and does not spawn a deferred render, since the update is already deferred.

The values you pass to `useDeferredValue` should either be primitive values (like strings and numbers) or objects created outside of rendering. If you create a new object during rendering and immediately pass it to `useDeferredValue`, it will be different on every render, causing unnecessary background re-renders.

When `useDeferredValue` receives a different value (compared with `Object.is`), in addition to the current render (when it still uses the previous value), it schedules a re-render in the background with the new value. The background re-render is interruptible: if there's another update to the value, React will restart the background re-render from scratch. For example, if the user is typing into an input faster than a chart receiving its deferred value can re-render, the chart will only re-render after the user stops typing.

`useDeferredValue` is integrated with `<Suspense>`. If the background update caused by a new value suspends the UI, the user will not see the fallback. They will see the old deferred value until the data loads.

`useDeferredValue` does not by itself prevent extra network requests.

There is no fixed delay caused by `useDeferredValue` itself. As soon as React finishes the original re-render, React will immediately start working on the background re-render with the new deferred value. Any updates caused by events (like typing) will interrupt the background re-render and get prioritized over it.

The background re-render caused by `useDeferredValue` does not fire Effects until it's committed to the screen. If the background re-render suspends, its Effects will run after the data loads and the UI updates.

Usage

Showing stale content while fresh content is loading

Call `useDeferredValue` at the top level of your component to defer updating some part of your UI.

```
import { useState, useDeferredValue } from 'react';
function SearchPage() {
  const [query, setQuery] = useState("");
  const deferredQuery = useDeferredValue(query); // ...
}
```

During the initial render, the deferred value will be the same as the value you provided.

During updates, the deferred value will “lag behind” the latest value. In particular, React will first re-render without updating the deferred value, and then try to re-render with the newly received value in the background.

Let’s walk through an example to see when this is useful.

Note This example assumes you use a Suspense-enabled data source:

Data fetching with Suspense-enabled frameworks like Relay and Next.js

Lazy-loading component code with lazy

Reading the value of a Promise with use

Learn more about Suspense and its limitations.

In this example, the SearchResults component suspends while fetching the search results. Try typing "a", waiting for the results, and then editing it to "ab". The results for "a" get replaced by the loading fallback.

```
App.js
```

```
App.js
```

```
ResetFor
```

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

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

```
export default function App() {  
  const [query, setQuery] = useState("");  
  
  return (  
    <>  
    <label>  
      Search albums:  
      <input value={query} onChange={e => setQuery(e.target.value)} />  
    </label>  
    <Suspense fallback={<h2>Loading...</h2>}>  
      <SearchResults query={query} />  
    </Suspense>  
  </>  
);  
}
```

Show more

A common alternative UI pattern is to defer updating the list of results and to keep showing the previous results until the new results are ready. Call `useDeferredValue` to pass a deferred version of the query down:

```
export default function App() {  
  const [query, setQuery] = useState("");  
  const deferredQuery = useDeferredValue(query);  
  return (  
    <>  
    <label>  
      Search albums:  
      <input value={query} onChange={e => setQuery(e.target.value)} />  
    </label>  
    <Suspense fallback={<h2>Loading...</h2>}>  
      <SearchResults query={deferredQuery} />  
    </Suspense>  
  </>  
);  
}
```

The query will update immediately, so the input will display the new value. However, the `deferredQuery` will keep its previous value until the data has loaded, so `SearchResults` will show the stale results for a bit.

Enter "a" in the example below, wait for the results to load, and then edit the input to "ab". Notice how instead of the Suspense fallback, you now see the stale result list until the new results have loaded:

```
App.jsApp.js ResetForkimport { Suspense, useState, useDeferredValue } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
  const [query, setQuery] = useState("");
  const deferredQuery = useDeferredValue(query);
  return (
    <>
    <label>
      Search albums:
      <input value={query} onChange={e => setQuery(e.target.value)} />
    </label>
    <Suspense fallback={<h2>Loading...</h2>}>
      <SearchResults query={deferredQuery} />
    </Suspense>
  </>
);
}
```

Show more

Deep DiveHow does deferring a value work under the hood? Show DetailsYou can think of it as happening in two steps:

First, React re-renders with the new query ("ab") but with the old deferredQuery (still "a"). The deferredQuery value, which you pass to the result list, is deferred: it “lags behind” the query value.

In the background, React tries to re-render with both query and deferredQuery updated to "ab". If this re-render completes, React will show it on the screen. However, if it suspends (the results for "ab" have not loaded yet), React will abandon this rendering attempt, and retry this re-render again after the data has loaded. The user will keep seeing the stale deferred value until the data is ready.

The deferred “background” rendering is interruptible. For example, if you type into the input again, React will abandon it and restart with the new value. React will always use the latest provided value. Note that there is still a network request per each keystroke. What’s being deferred here is displaying results (until they’re ready), not the

network requests themselves. Even if the user continues typing, responses for each keystroke get cached, so pressing Backspace is instant and doesn't fetch again.

Indicating that the content is stale

In the example above, there is no indication that the result list for the latest query is still loading. This can be confusing to the user if the new results take a while to load. To make it more obvious to the user that the result list does not match the latest query, you can add a visual indication when the stale result list is displayed:

```
<div style={{ opacity: query !== deferredQuery ? 0.5 : 1 }}> <SearchResults query={deferredQuery} /></div>
```

With this change, as soon as you start typing, the stale result list gets slightly dimmed until the new result list loads. You can also add a CSS transition to delay dimming so that it feels gradual, like in the example below:

```
App.js
```

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

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

```
export default function App() {
  const [query, setQuery] = useState("");
  const deferredQuery = useDeferredValue(query);
  const isStale = query !== deferredQuery;
  return (
    <>
    <label>
      Search albums:
      <input value={query} onChange={e => setQuery(e.target.value)} />
    </label>
    <Suspense fallback={<h2>Loading...</h2>}>
      <div style={{
        opacity: isStale ? 0.5 : 1,
        transition: isStale ? 'opacity 0.2s 0.2s linear' : 'opacity 0s 0s linear'
      }}>
        <SearchResults query={deferredQuery} />
      </div>
    </Suspense>
    </>
  );
}
```

Show more

Deferring re-rendering for a part of the UI

You can also apply `useDeferredValue` as a performance optimization. It is useful when a part of your UI is slow to re-render, there's no easy way to optimize it, and you want to prevent it from blocking the rest of the UI.

Imagine you have a text field and a component (like a chart or a long list) that re-renders on every keystroke:

```
function App() { const [text, setText] = useState(""); return ( <> <input value={text} onChange={e => setText(e.target.value)} /> <SlowList text={text} /> </> );}
```

First, optimize `SlowList` to skip re-rendering when its props are the same. To do this, wrap it in `memo`:

```
const SlowList = memo(function SlowList({ text }) { // ...});
```

However, this only helps if the `SlowList` props are the same as during the previous render. The problem you're facing now is that it's slow when they're different, and when you actually need to show different visual output.

Concretely, the main performance problem is that whenever you type into the input, the `SlowList` receives new props, and re-rendering its entire tree makes the typing feel janky. In this case, `useDeferredValue` lets you prioritize updating the input (which must be fast) over updating the result list (which is allowed to be slower):

```
function App() { const [text, setText] = useState(""); const deferredText = useDeferredValue(text); return ( <> <input value={text} onChange={e => setText(e.target.value)} /> <SlowList text={deferredText} /> </> );}
```

This does not make re-rendering of the `SlowList` faster. However, it tells React that re-rendering the list can be deprioritized so that it doesn't block the keystrokes. The list will "lag behind" the input and then "catch up". Like before, React will attempt to update the list as soon as possible, but will not block the user from typing.

The difference between `useDeferredValue` and unoptimized re-rendering1. Deferred re-rendering of the list 2. Unoptimized re-rendering of the list Example 1 of 2: Deferred re-rendering of the list In this example, each item in the `SlowList` component is artificially slowed down so that you can see how `useDeferredValue` lets you keep the input responsive. Type into the input and notice that typing feels snappy while the list "lags behind"

```
it.App.jsSlowList.jsApp.js ResetForkimport { useState, useDeferredValue } from 'react';
```

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

```
export default function App() {
  const [text, setText] = useState("");
  const deferredText = useDeferredValue(text);
  return (
    <>
    <input value={text} onChange={e => setText(e.target.value)} />
    <SlowList text={deferredText} />
    </>
  );
}
```

[Next Example](#)

PitfallThis optimization requires SlowList to be wrapped in memo. This is because whenever the text changes, React needs to be able to re-render the parent component quickly. During that re-render, deferredText still has its previous value, so SlowList is able to skip re-rendering (its props have not changed). Without memo, it would have to re-render anyway, defeating the point of the optimization.

Deep DiveHow is deferring a value different from debouncing and throttling? Show DetailsThere are two common optimization techniques you might have used before in this scenario:

Debouncing means you'd wait for the user to stop typing (e.g. for a second) before updating the list.

Throttling means you'd update the list every once in a while (e.g. at most once a second).

While these techniques are helpful in some cases, useDeferredValue is better suited to optimizing rendering because it is deeply integrated with React itself and adapts to the user's device.Unlike debouncing or throttling, it doesn't require choosing any fixed delay. If the user's device is fast (e.g. powerful laptop), the deferred re-render would happen almost immediately and wouldn't be noticeable. If the user's device is slow, the list would "lag behind" the input proportionally to how slow the device is.Also, unlike with debouncing or throttling, deferred re-renders done by useDeferredValue are interruptible by default. This means that if React is in the middle of re-rendering a large list, but the user makes another keystroke, React will abandon that re-render, handle the keystroke, and then start rendering in the background again. By contrast, debouncing and throttling still produce a janky experience because they're blocking: they merely postpone the moment when rendering blocks the keystroke.If the work you're optimizing doesn't happen during rendering, debouncing and throttling are still useful. For example, they can let you fire fewer network requests. You can also use these techniques

together.PrevioususeDebugValueNextuseEffect©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReference useDeferredValue(value, initialValue?) Usage Showing stale content while fresh content is loading Indicating that the content is stale Deferring re-rendering for a part of the UI useEffect – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useActionState - This feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceHooksuseEffectuseEffect is a React Hook that lets you synchronize a component with an external system.useEffect(setup, dependencies?)

Reference useEffect(setup, dependencies?) Usage Connecting to an external system Wrapping Effects in custom Hooks Controlling a non-React widget Fetching data with Effects Specifying reactive dependencies Updating state

based on previous state from an Effect
Removing unnecessary object dependencies
Reading the latest props and state from an Effect
Displaying different content on the server and the client
Troubleshooting
My Effect runs twice when the component mounts
My Effect runs after every re-render
My Effect keeps re-running in an infinite cycle
My cleanup logic runs even though my component didn't unmount
My Effect does something visual, and I see a flicker before it runs

Reference

`useEffect(setup, dependencies?)`

Call `useEffect` at the top level of your component to declare an Effect:

```
import { 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(); }; }, [serverUrl, roomId]); // ... }
```

See more examples below.

Parameters

`setup`: The function with your Effect's logic. Your setup function may also optionally return a cleanup function. When your component is added to the DOM, React will run your setup function. After every re-render with changed dependencies, React will first run the cleanup function (if you provided it) with the old values, and then run your setup function with the new values. After your component is removed from the DOM, React will run your cleanup function.

`optional dependencies`: The list of all reactive values referenced inside of the setup code. Reactive values include `props`, `state`, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the `Object.is` comparison. If you omit this argument, your Effect will re-run after every re-render of the component. See the difference between passing an array of dependencies, an empty array, and no dependencies at all.

Returns

`useEffect` returns `undefined`.

Caveats

`useEffect` is a Hook, so you can only call it at the top level of your component or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

If you're not trying to synchronize with some external system, you probably don't need an Effect.

When Strict Mode is on, React will run one extra development-only setup+cleanup cycle before the first real setup. This is a stress-test that ensures that your cleanup logic “mirrors” your setup logic and that it stops or undoes whatever the setup is doing. If this causes a problem, implement the cleanup function.

If some of your dependencies are objects or functions defined inside the component, there is a risk that they will cause the Effect to re-run more often than needed. To fix this, remove unnecessary object and function dependencies. You can also extract state updates and non-reactive logic outside of your Effect.

If your Effect wasn't caused by an interaction (like a click), React will generally let the browser paint the updated screen first before running your Effect. If your Effect is doing something visual (for example, positioning a tooltip), and the delay is noticeable (for example, it flickers), replace `useEffect` with `useLayoutEffect`.

If your Effect is caused by an interaction (like a click), React may run your Effect before the browser paints the updated screen. This ensures that the result of the Effect can be observed by the event system. Usually, this works as expected. However, if you must defer the work until after paint, such as an `alert()`, you can use `setTimeout`. See [reactwg/react-18/128](#) for more information.

Even if your Effect was caused by an interaction (like a click), React may allow the browser to repaint the screen before processing the state updates inside your Effect. Usually, this works as expected. However, if you must block the browser from repainting the screen, you need to replace `useEffect` with `useLayoutEffect`.

Effects only run on the client. They don't run during server rendering.

Usage

Connecting to an external system

Some components need to stay connected to the network, some browser API, or a third-party library, while they are displayed on the page. These systems aren't controlled by React, so they are called external.

To connect your component to some external system, call `useEffect` at the top level of your component:

```
import { 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(); }; }, [serverUrl, roomId]); // ... }
```

You need to pass two arguments to `useEffect`:

A setup function with setup code that connects to that system.

It should return a cleanup function with cleanup code that disconnects from that system.

A list of dependencies including every value from your component used inside of those functions.

React calls your setup and cleanup functions whenever it's necessary, which may happen multiple times:

Your setup code runs when your component is added to the page (mounts).

After every re-render of your component where the dependencies have changed:

First, your cleanup code runs with the old props and state.

Then, your setup code runs with the new props and state.

Your cleanup code runs one final time after your component is removed from the page (unmounts).

Let's illustrate this sequence for the example above.

When the `ChatRoom` component above gets added to the page, it will connect to the chat room with the initial `serverUrl` and `roomId`. If either `serverUrl` or `roomId` change as a result of a re-render (say, if the user picks a different chat room in a dropdown), your Effect will disconnect from the previous room, and connect to the next one. When the `ChatRoom` component is removed from the page, your Effect will disconnect one last time.

To help you find bugs, in development React runs setup and cleanup one extra time before the setup. This is a stress-test that verifies your Effect's logic is implemented correctly. If this causes visible issues, your cleanup function is missing some logic. The cleanup function should stop or undo whatever the setup function was doing. The rule of thumb is that the user shouldn't be able to distinguish between the setup being called once (as in production) and a `setup → cleanup → setup` sequence (as in development). See common solutions.

Try to write every Effect as an independent process and think about a single setup/cleanup cycle at a time. It shouldn't matter whether your component is mounting, updating, or unmounting. When your cleanup logic correctly "mirrors" the setup logic, your Effect is resilient to running setup and cleanup as often as needed.

Note An Effect lets you keep your component synchronized with some external system (like a chat service). Here, external system means any piece of code that's not controlled by React, such as:

A timer managed with `setInterval()` and `clearInterval()`.

An event subscription using `window.addEventListener()` and `window.removeEventListener()`.

A third-party animation library with an API like `animation.start()` and `animation.reset()`.

If you're not connecting to any external system, you probably don't need an Effect.

Examples of connecting to an external system 1. Connecting to a chat server 2. Listening to a global browser event 3. Triggering an animation 4. Controlling a modal dialog 5. Tracking element visibility Example 1 of 5: Connecting to a chat server In this example, the ChatRoom component uses an Effect to stay connected to an external system defined in `chat.js`. Press “Open chat” to make the ChatRoom component appear. This sandbox runs in development mode, so there is an extra connect-and-disconnect cycle, as explained here. Try changing the `roomId` and `serverUrl` using the dropdown and the input, and see how the Effect re-connects to the chat. Press “Close chat” to see the Effect disconnect one last time.

```
App.js
import { 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');
  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](#)[Next Example](#)

Wrapping Effects in custom Hooks

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. If you find yourself often needing to manually write Effects, it’s usually a sign that you need to extract some custom Hooks for common behaviors your components rely on.

For example, this useChatRoom custom Hook “hides” the logic of your Effect behind a more declarative API:

```
function useChatRoom({ serverUrl, roomId }) { useEffect(() => { const options = { serverUrl: serverUrl, roomId: roomId }; const connection = createConnection(options); connection.connect(); return () => connection.disconnect(); }, [roomId, serverUrl]);}
```

Then you can use it from any component like this:

```
function ChatRoom({ roomId }) { const [serverUrl, setServerUrl] = useState('https://localhost:1234'); useChatRoom({ roomId: roomId, serverUrl: serverUrl }); // ...}
```

There are also many excellent custom Hooks for every purpose available in the React ecosystem.

Learn more about wrapping Effects in custom Hooks.

Examples of wrapping Effects in custom Hooks

1. Custom useChatRoom Hook
2. Custom useWindowListener Hook
3. Custom useIntersectionObserver Hook
Example 1 of 3: Custom useChatRoom Hook This example is identical to one of the earlier examples, but the logic is extracted to a custom Hook.

```
App.js
import { useState } from 'react';

useChatRoom.js
import { useChatRoom } from './useChatRoom.js';

ChatRoom.js
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>
  </>
);
}
```

```

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](#)[Next Example](#)

Controlling a non-React widget

Sometimes, you want to keep an external system synchronized to some prop or state of your component.

For example, if you have a third-party map widget or a video player component written without React, you can use an Effect to call methods on it that make its state match the current state of your React component. This Effect creates an instance of a MapWidget class defined in map-widget.js. When you change the zoomLevel prop of the Map component, the Effect calls the setZoom() on the class instance to keep it synchronized:

```

App.js
import { useRef, useEffect } from 'react';
import { MapWidget } from './map-widget.js';

```

```

export default function Map({ zoomLevel }) {
  const containerRef = useRef(null);
  const mapRef = useRef(null);

  useEffect(() => {
    if (mapRef.current === null) {
      mapRef.current = new MapWidget(containerRef.current);
    }
  });

  const map = mapRef.current;
  map.setZoom(zoomLevel);
}, [zoomLevel]);

return (
  <div
    style={{ width: 200, height: 200 }}
    ref={containerRef}
  />
);
}

```

Show more

In this example, a cleanup function is not needed because the `MapWidget` class manages only the DOM node that was passed to it. After the `Map` React component is removed from the tree, both the DOM node and the `MapWidget` class instance will be automatically garbage-collected by the browser JavaScript engine.

Fetching data with Effects

You can use an Effect to fetch data for your component. Note that if you use a framework, using your framework's data fetching mechanism will be a lot more efficient than writing Effects manually.

If you want to fetch data from an Effect manually, your code might look like this:

```

import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);

  useEffect(() => {
    let ignore = false;
    setBio(null);
    fetchBio(person).then(result => {
      if (!ignore) {
        setBio(result);
      }
    });
    return () => {
      ignore = true;
    };
  }, [person]);
}

```

Note the `ignore` variable which is initialized to `false`, and is set to `true` during cleanup. This ensures your code doesn't suffer from "race conditions": network responses may arrive in a different order than you sent them.

```
App.jsApp.js ResetForKimport { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);
  useEffect(() => {
    let ignore = false;
    setBio(null);
    fetchBio(person).then(result => {
      if (!ignore) {
        setBio(result);
      }
    });
    return () => {
      ignore = true;
    }
  }, [person]);
}

return (
  <>
  <select value={person} onChange={e => {
    setPerson(e.target.value);
  }}>
    <option value="Alice">Alice</option>
    <option value="Bob">Bob</option>
    <option value="Taylor">Taylor</option>
  </select>
  <hr />
  <p><i>{bio ?? 'Loading...'}</i></p>
</>
);
}
```

Show more

You can also rewrite using the `async / await` syntax, but you still need to provide a cleanup function:

```
App.jsApp.js ResetForkimport { useState, useEffect } from 'react';
```

```
import { fetchBio } from './api.js';
```

```
export default function Page() {  
  const [person, setPerson] = useState('Alice');  
  const [bio, setBio] = useState(null);  
  
  useEffect(() => {  
    async function startFetching() {  
      setBio(null);  
      const result = await fetchBio(person);  
      if (!ignore) {  
        setBio(result);  
      }  
    }  
  })  
}
```

```
let ignore = false;  
  
startFetching();  
  
return () => {  
  ignore = true;  
}  
}, [person]);
```

```
return (  
  <>  
  <select value={person} onChange={e => {  
    setPerson(e.target.value);  
  }}>  
  <option value="Alice">Alice</option>  
  <option value="Bob">Bob</option>  
  <option value="Taylor">Taylor</option>  
  </select>  
  <hr />
```

```
<p><i>{bio ?? 'Loading...'}</i></p>
</>
);
}
```

Show more

Writing data fetching directly in Effects gets repetitive and makes it difficult to add optimizations like caching and server rendering later. It's easier to use a custom Hook—either your own or maintained by the community.

Deep Dive What are good alternatives to data fetching in Effects? Show Details Writing 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 also 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.

Specifying reactive dependencies

Notice that you can't “choose” the dependencies of your Effect. Every reactive value used by your Effect's code must be declared as a dependency. Your Effect's dependency list is determined by the surrounding code:

```
function ChatRoom({ roomId }) { // This is a reactive value const [serverUrl, setServerUrl] =
useState('https://localhost:1234'); // This is a reactive value too useEffect(() => {
  const connection =
createConnection(serverUrl, roomId); // This Effect reads these reactive values connection.connect(); return () =>
connection.disconnect(); }, [serverUrl, roomId]); // tickmark So you must specify them as dependencies of your
Effect // ...}
```

If either serverUrl or roomId change, your Effect will reconnect to the chat using the new values.

Reactive values include props and all variables and functions declared directly inside of your component. Since roomId and serverUrl are reactive values, you can't remove them from the dependencies. If you try to omit them and your linter is correctly configured for React, the linter will flag this as a mistake you need to fix:

```
function ChatRoom({ roomId }) { const [serverUrl, setServerUrl] = useState('https://localhost:1234'); useEffect(() => { const connection = createConnection(serverUrl, roomId); connection.connect(); return () => connection.disconnect(); }, []); // React Hook useEffect has missing dependencies: 'roomId' and 'serverUrl' // ...}
```

To remove a dependency, you need to “prove” to the linter that it doesn't need to be a dependency. For example, you can move serverUrl out of your component to prove that it's not reactive and won't change on re-renders:

```
const serverUrl = 'https://localhost:1234'; // Not a reactive value anymorefunction ChatRoom({ roomId }) { useEffect(() => { const connection = createConnection(serverUrl, roomId); connection.connect(); return () => connection.disconnect(); }, [roomId]); // tickmark All dependencies declared // ...}
```

Now that serverUrl is not a reactive value (and can't change on a re-render), it doesn't need to be a dependency. If your Effect's code doesn't use any reactive values, its dependency list should be empty ([]):

```
const serverUrl = 'https://localhost:1234'; // Not a reactive value anymoreconst roomId = 'music'; // Not a reactive value anymorefunction ChatRoom() { useEffect(() => { const connection = createConnection(serverUrl, roomId); connection.connect(); return () => connection.disconnect(); }, []); // tickmark All dependencies declared // ...}
```

An Effect with empty dependencies doesn't re-run when any of your component's props or state change.

Pitfall If you have an existing codebase, you might have some Effects that suppress the linter like this:
`useEffect(() => {} // ... // Avoid suppressing the linter like this: // eslint-ignore-next-line react-hooks/exhaustive-deps), []);` When dependencies don't match the code, there is a high risk of introducing bugs. By suppressing the linter, you “lie” to React about the values your Effect depends on. Instead, prove they're unnecessary.

Examples of passing reactive dependencies
1. Passing a dependency array
2. Passing an empty dependency array
3. Passing no dependency array at all
Example 1 of 3: Passing a dependency array
If you specify the dependencies, your Effect runs after the initial render and after re-renders with changed dependencies.
`useEffect(() => {} // ...), [a, b]);`
Runs again if a or b are different
In the below example, serverUrl and roomId are reactive values, so they both must be specified as dependencies. As a result, selecting a different room in the dropdown or editing the server URL input causes the chat to re-connect. However, since message isn't used in the Effect (and so it isn't a dependency), editing the message doesn't re-connect to the chat.

```
App.js
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');
  const [message, setMessage] = useState('');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
}

export default ChatRoom;
```

```
ChatRoom.js
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');
  const [message, setMessage] = useState('');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
}

export default ChatRoom;
```

```
return (
  <>
  <label>
    Server URL:{' '}
  <input
    value={serverUrl}
    onChange={e => setServerUrl(e.target.value)}
  />
</label>
<h1>Welcome to the {roomId} room!</h1>
<label>
  Your message:{' '}
  <input value={message} onChange={e => setMessage(e.target.value)} />
</label>
</>
);
}
```

```
export default function App() {
  const [show, setShow] = useState(false);
  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>
  );
}
```

```

<button onClick={() => setShow(!show)}>
  {show ? 'Close chat' : 'Open chat'}
</button>
</label>
{show && <hr />}
{show && <ChatRoom roomId={roomId}/>}
</>
);
}

```

[Show more](#)[Next Example](#)

Updating state based on previous state from an Effect

When you want to update state based on previous state from an Effect, you might run into a problem:

```
function Counter() { const [count, setCount] = useState(0); useEffect(() => { const intervalId = setInterval(() => {
setCount(count + 1); // You want to increment the counter every second... }, 1000) return () =>
clearInterval(intervalId); }, [count]); // ... but specifying `count` as a dependency always resets the interval. // ...}
```

Since count is a reactive value, it must be specified in the list of dependencies. However, that causes the Effect to cleanup and setup again every time the count changes. This is not ideal.

To fix this, pass the `c => c + 1` state updater to `setCount`:

```
App.js
import { useState, useEffect } from 'react';
```

```

export default function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(c => c + 1); // tickmark Pass a state updater
    }, 1000);
    return () => clearInterval(intervalId);
  }, []); // tickmark Now count is not a dependency

  return <h1>{count}</h1>;
}

```

Now that you're passing `c => c + 1` instead of `count + 1`, your Effect no longer needs to depend on `count`. As a result of this fix, it won't need to cleanup and setup the interval again every time the `count` changes.

Removing unnecessary object dependencies

If your Effect depends on an object or a function created during rendering, it might run too often. For example, this Effect re-connects after every render because the `options` object is different for every render:

```
const serverUrl = 'https://localhost:1234';function ChatRoom({ roomId }) { const [message, setMessage] = useState(""); const options = { // This object is created from scratch on every re-render serverUrl: serverUrl, roomId: roomId }; useEffect(() => { const connection = createConnection(options); // It's used inside the Effect connection.connect(); return () => connection.disconnect(); }, [options]); // As a result, these dependencies are always different on a re-render // ... }
```

Avoid using an object created during rendering as a dependency. Instead, create the object inside the Effect:

```
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

Now that you create the options object inside the Effect, the Effect itself only depends on the roomId string.

With this fix, typing into the input doesn't reconnect the chat. Unlike an object which gets re-created, a string like roomId doesn't change unless you set it to another value. Read more about removing dependencies.

Removing unnecessary function dependencies

If your Effect depends on an object or a function created during rendering, it might run too often. For example, this Effect re-connects after every render because the createOptions function is different for every render:

```

function ChatRoom({ roomId }) { const [message, setMessage] = useState(''); function createOptions() { // This
  function is created from scratch on every re-render  return { serverUrl: serverUrl, roomId: roomId }; }
  useEffect(() => { const options = createOptions(); // It's used inside the Effect  const connection =

```

```
createConnection(); connection.connect(); return () => connection.disconnect(); }, [createOptions]); // As a result, these dependencies are always different on a re-render // ...
```

By itself, creating a function from scratch on every re-render is not a problem. You don't need to optimize that. However, if you use it as a dependency of your Effect, it will cause your Effect to re-run after every re-render.

Avoid using a function created during rendering as a dependency. Instead, declare it inside the Effect:

```
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(() => {
```

```
    function createOptions() {
```

```
      return {
```

```
        serverUrl: serverUrl,
```

```
        roomId: roomId
```

```
      };
```

```
    }
```

```
    const options = createOptions();
```

```
    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

Now that you define the `createOptions` function inside the Effect, the Effect itself only depends on the `roomId` string. With this fix, typing into the input doesn't reconnect the chat. Unlike a function which gets re-created, a string like `roomId` doesn't change unless you set it to another value. Read more about removing dependencies.

Reading the latest props and state from an Effect

Under Construction This section describes an experimental API that has not yet been released in a stable version of React.

By default, when you read a reactive value from an Effect, you have to add it as a dependency. This ensures that your Effect “reacts” to every change of that value. For most dependencies, that’s the behavior you want.

However, sometimes you’ll want to read the latest props and state from an Effect without “reacting” to them. For example, imagine you want to log the number of the items in the shopping cart for every page visit:

```
function Page({ url, shoppingCart }) { useEffect(() => { logVisit(url, shoppingCart.length); }, [url, shoppingCart]); // tickmark All dependencies declared // ...}
```

What if you want to log a new page visit after every url change, but not if only the `shoppingCart` changes? You can’t exclude `shoppingCart` from dependencies without breaking the reactivity rules. However, you can express that you

don't want a piece of code to "react" to changes even though it is called from inside an Effect. Declare an Effect Event with the `useEffectEvent` Hook, and move the code reading `shoppingCart` inside of it:

```
function Page({ url, shoppingCart }) { const onVisit = useEffectEvent(visitedUrl => { logVisit(visitedUrl, shoppingCart.length) }); useEffect(() => { onVisit(url); }, [url]); // tickmark All dependencies declared // ...}
```

Effect Events are not reactive and must always be omitted from dependencies of your Effect. This is what lets you put non-reactive code (where you can read the latest value of some props and state) inside of them. By reading `shoppingCart` inside of `onVisit`, you ensure that `shoppingCart` won't re-run your Effect.

Read more about how Effect Events let you separate reactive and non-reactive code.

Displaying different content on the server and the client

If your app uses server rendering (either directly or via a framework), your component will render in two different environments. On the server, it will render to produce the initial HTML. On the client, React will run the rendering code again so that it can attach your event handlers to that HTML. This is why, for hydration to work, your initial render output must be identical on the client and the server.

In rare cases, you might need to display different content on the client. For example, if your app reads some data from `localStorage`, it can't possibly do that on the server. Here is how you could implement this:

```
function MyComponent() { const [didMount, setDidMount] = useState(false); useEffect(() => { setDidMount(true); }, []); if (didMount) { // ... return client-only JSX ... } else { // ... return initial JSX ... }}
```

While the app is loading, the user will see the initial render output. Then, when it's loaded and hydrated, your Effect will run and set `didMount` to true, triggering a re-render. This will switch to the client-only render output. Effects don't run on the server, so this is why `didMount` was false during the initial server render.

Use this pattern sparingly. Keep in mind that users with a slow connection will see the initial content for quite a bit of time—potentially, many seconds—so you don't want to make jarring changes to your component's appearance. In many cases, you can avoid the need for this by conditionally showing different things with CSS.

Troubleshooting

My Effect runs twice when the component mounts

When Strict Mode is on, in development, React runs setup and cleanup one extra time before the actual setup.

This is a stress-test that verifies your Effect's logic is implemented correctly. If this causes visible issues, your cleanup function is missing some logic. The cleanup function should stop or undo whatever the setup function was doing. The rule of thumb is that the user shouldn't be able to distinguish between the setup being called once (as in production) and a setup → cleanup → setup sequence (as in development).

Read more about how this helps find bugs and how to fix your logic.

My Effect runs after every re-render

First, check that you haven't forgotten to specify the dependency array:

```
useEffect(() => { // ...}); // No dependency array: re-runs after every render!
```

If you've specified the dependency array but your Effect still re-runs in a loop, it's because one of your dependencies is different on every re-render.

You can debug this problem by manually logging your dependencies to the console:

```
useEffect(() => { // .. }, [serverUrl, roomId]); console.log([serverUrl, roomId]);
```

You can then right-click on the arrays from different re-renders in the console and select “Store as a global variable” for both of them. Assuming the first one got saved as temp1 and the second one got saved as temp2, you can then use the browser console to check whether each dependency in both arrays is the same:

```
Object.is(temp1[0], temp2[0]); // Is the first dependency the same between the arrays? Object.is(temp1[1], temp2[1]); // Is the second dependency the same between the arrays? Object.is(temp1[2], temp2[2]); // ... and so on for every dependency ...
```

When you find the dependency that is different on every re-render, you can usually fix it in one of these ways:

Updating state based on previous state from an Effect

Removing unnecessary object dependencies

Removing unnecessary function dependencies

Reading the latest props and state from an Effect

As a last resort (if these methods didn’t help), wrap its creation with useMemo or useCallback (for functions).

My Effect keeps re-running in an infinite cycle

If your Effect runs in an infinite cycle, these two things must be true:

Your Effect is updating some state.

That state leads to a re-render, which causes the Effect’s dependencies to change.

Before you start fixing the problem, ask yourself whether your Effect is connecting to some external system (like DOM, network, a third-party widget, and so on). Why does your Effect need to set state? Does it synchronize with that external system? Or are you trying to manage your application’s data flow with it?

If there is no external system, consider whether removing the Effect altogether would simplify your logic.

If you’re genuinely synchronizing with some external system, think about why and under what conditions your Effect should update the state. Has something changed that affects your component’s visual output? If you need to keep track of some data that isn’t used by rendering, a ref (which doesn’t trigger re-renders) might be more appropriate. Verify your Effect doesn’t update the state (and trigger re-renders) more than needed.

Finally, if your Effect is updating the state at the right time, but there is still a loop, it’s because that state update leads to one of the Effect’s dependencies changing. Read how to debug dependency changes.

My cleanup logic runs even though my component didn’t unmount

The cleanup function runs not only during unmount, but before every re-render with changed dependencies. Additionally, in development, React runs setup+cleanup one extra time immediately after component mounts.

If you have cleanup code without corresponding setup code, it’s usually a code smell:

```
useEffect(() => { // Avoid: Cleanup logic without corresponding setup logic return () => { doSomething(); }; }, []);
```

Your cleanup logic should be “symmetrical” to the setup logic, and should stop or undo whatever setup did:

```
useEffect(() => { const connection = createConnection(serverUrl, roomId); connection.connect(); return () => { connection.disconnect(); }; }, [serverUrl, roomId]);
```

Learn how the Effect lifecycle is different from the component’s lifecycle.

My Effect does something visual, and I see a flicker before it runs

If your Effect must block the browser from painting the screen, replace useEffect with useLayoutEffect. Note that this shouldn’t be needed for the vast majority of Effects. You’ll only need this if it’s crucial to run your Effect before the browser paint: for example, to measure and position a tooltip before the user sees

it.PrevioususeDeferredValueNextusId©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReference useEffect(setup, dependencies?) Usage Connecting to an external system Wrapping Effects in custom Hooks Controlling a non-React widget Fetching data with Effects Specifying reactive dependencies Updating state based on previous state from an Effect Removing unnecessary object dependencies Removing unnecessary function dependencies Reading the latest props and state from an Effect Displaying different content on the server and the client Troubleshooting My Effect runs twice when the component mounts My Effect runs after every re-render My Effect keeps re-running in an infinite cycle My cleanup logic runs even though my component didn’t unmount My Effect does something visual, and I see a flicker before it runs uselId – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useActionState - This feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect uselId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div>) <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceHooksuselIduselId is a React Hook for generating unique IDs that can be passed to accessibility attributes.const id = uselId()

Reference uselId() Usage Generating unique IDs for accessibility attributes Generating IDs for several related elements Specifying a shared prefix for all generated IDs Using the same ID prefix on the client and the server

Reference

uselId()

Call `useId` at the top level of your component to generate a unique ID:

```
import { useId } from 'react';function PasswordField() { const passwordHintId = useId(); // ...}
```

See more examples below.

Parameters

`useId` does not take any parameters.

Returns

`useId` returns a unique ID string associated with this particular `useId` call in this particular component.

Caveats

`useId` is a Hook, so you can only call it at the top level of your component or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

`useId` should not be used to generate keys in a list. Keys should be generated from your data.

Usage

Pitfall Do not call `useId` to generate keys in a list. Keys should be generated from your data.

Generating unique IDs for accessibility attributes

Call `useId` at the top level of your component to generate a unique ID:

```
import { useId } from 'react';function PasswordField() { const passwordHintId = useId(); // ...}
```

You can then pass the generated ID to different attributes:

```
<> <input type="password" aria-describedby={passwordHintId} /> <p id={passwordHintId}></p>
```

Let's walk through an example to see when this is useful.

HTML accessibility attributes like `aria-describedby` let you specify that two tags are related to each other. For example, you can specify that an element (like an input) is described by another element (like a paragraph).

In regular HTML, you would write it like this:

```
<label> Password: <input type="password" aria-describedby="password-hint" /></label><p id="password-hint"> The password should contain at least 18 characters</p>
```

However, hardcoding IDs like this is not a good practice in React. A component may be rendered more than once on the page—but IDs have to be unique! Instead of hardcoding an ID, generate a unique ID with `useId`:

```
import { useId } from 'react';function PasswordField() { const passwordHintId = useId(); return ( <> <label> Password: <input type="password" aria-describedby={passwordHintId} /> </label> <p id={passwordHintId}> The password should contain at least 18 characters </p> </> );}
```

Now, even if `PasswordField` appears multiple times on the screen, the generated IDs won't clash.

```
App.jsApp.js ResetForkimport { useId } from 'react';
```

```
function PasswordField() {
  const passwordHintId = useId();
  return (
    <>
    <label>
      Password:
      <input
        type="password"
        aria-describedby={passwordHintId}
      />
    </label>
    <p id={passwordHintId}>
      The password should contain at least 18 characters
    </p>
    </>
  );
}
```

```
export default function App() {
  return (
    <>
    <h2>Choose password</h2>
    <PasswordField />
    <h2>Confirm password</h2>
    <PasswordField />
    </>
  );
}
```

Show more

Watch this video to see the difference in the user experience with assistive technologies.

Pitfall With server rendering, `useId` requires an identical component tree on the server and the client. If the trees you render on the server and the client don't match exactly, the generated IDs won't match.

Deep Dive Why is `useId` better than an incrementing counter? Show Details You might be wondering why `useId` is better than incrementing a global variable like `nextId++`. The primary benefit of `useId` is that React ensures that it works with server rendering. During server rendering, your components generate HTML output. Later, on the client, hydration attaches your event handlers to the generated HTML. For hydration to work, the client output must match the server HTML. This is very difficult to guarantee with an incrementing counter because the order in which the Client Components are hydrated may not match the order in which the server HTML was emitted. By calling `useId`, you ensure that hydration will work, and the output will match between the server and the client. Inside React, `useId` is generated from the "parent path" of the calling component. This is why, if the client and the server tree are the same, the "parent path" will match up regardless of rendering order.

Generating IDs for several related elements

If you need to give IDs to multiple related elements, you can call `useId` to generate a shared prefix for them:

```
App.js
App.js ResetFor
import { useId } from 'react';
```

```
export default function Form() {
  const id = useId();
  return (
    <form>
      <label htmlFor={id + '-firstName'}>First Name:</label>
      <input id={id + '-firstName'} type="text" />
      <hr />
      <label htmlFor={id + '-lastName'}>Last Name:</label>
      <input id={id + '-lastName'} type="text" />
    </form>
  );
}
```

This lets you avoid calling `useId` for every single element that needs a unique ID.

Specifying a shared prefix for all generated IDs

If you render multiple independent React applications on a single page, pass `identifierPrefix` as an option to your `createRoot` or `hydrateRoot` calls. This ensures that the IDs generated by the two different apps never clash because every identifier generated with `useId` will start with the distinct prefix you've specified.

```
index.js
index.html
App.js
index.js ResetFor
import { createRoot } from 'react-dom/client';
import App from './App.js';
```

```

import './styles.css';

const root1 = createRoot(document.getElementById('root1'), {
  identifierPrefix: 'my-first-app-'
});
root1.render(<App />);

const root2 = createRoot(document.getElementById('root2'), {
  identifierPrefix: 'my-second-app-'
});
root2.render(<App />);

```

Using the same ID prefix on the client and the server

If you render multiple independent React apps on the same page, and some of these apps are server-rendered, make sure that the identifierPrefix you pass to the hydrateRoot call on the client side is the same as the identifierPrefix you pass to the server APIs such as renderToPipeableStream.

```

// Serverimport { renderToPipeableStream } from 'react-dom/server';const { pipe } = renderToPipeableStream( <App />, { identifierPrefix: 'react-app1' });

// Clientimport { hydrateRoot } from 'react-dom/client';const domNode = document.getElementById('root');const root = hydrateRoot( domNode, reactNode, { identifierPrefix: 'react-app1' });

```

You do not need to pass identifierPrefix if you only have one React app on the page. PrevioususeEffectNextuseImperativeHandle©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReference useId() Usage Generating unique IDs for accessibility attributes Generating IDs for several related elements Specifying a shared prefix for all generated IDs Using the same ID prefix on the client and the server useImperativeHandle – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useActionState - This feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule -

This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children cloneElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API

ReferenceHooksuseImperativeHandleuseImperativeHandle is a React Hook that lets you customize the handle exposed as a ref.useImperativeHandle(ref, createHandle, dependencies?)

Reference useImperativeHandle(ref, createHandle, dependencies?) Usage Exposing a custom ref handle to the parent component Exposing your own imperative methods

Reference

useImperativeHandle(ref, createHandle, dependencies?)

Call useImperativeHandle at the top level of your component to customize the ref handle it exposes:

```
import { forwardRef, useImperativeHandle } from 'react';const MyInput = forwardRef(function MyInput(props, ref) {  useImperativeHandle(ref, () => {    return {      // ... your methods ...    }; }, []); // ...});
```

See more examples below.

Parameters

ref: The ref you received as the second argument from the forwardRef render function.

createHandle: A function that takes no arguments and returns the ref handle you want to expose. That ref handle can have any type. Usually, you will return an object with the methods you want to expose.

optional dependencies: The list of all reactive values referenced inside of the createHandle code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison. If a re-render resulted in a change to some dependency, or if you omitted this argument, your createHandle function will re-execute, and the newly created handle will be assigned to the ref.

Returns

useImperativeHandle returns undefined.

Usage

Exposing a custom ref handle to the parent component

By default, components don't expose their DOM nodes to parent components. For example, if you want the parent component of MyInput to have access to the <input> DOM node, you have to opt in with forwardRef:

```
import { forwardRef } from 'react';const MyInput = forwardRef(function MyInput(props, ref) { return <input {...props} ref={ref} />});
```

With the code above, a ref to MyInput will receive the <input> DOM node. However, you can expose a custom value instead. To customize the exposed handle, call useImperativeHandle at the top level of your component:

```
import { forwardRef, useImperativeHandle } from 'react';const MyInput = forwardRef(function MyInput(props, ref) { useImperativeHandle(ref, () => { return { // ... your methods ... }; }, []); return <input {...props} />});
```

Note that in the code above, the ref is no longer forwarded to the <input>.

For example, suppose you don't want to expose the entire <input> DOM node, but you want to expose two of its methods: focus and scrollIntoView. To do this, keep the real browser DOM in a separate ref. Then use useImperativeHandle to expose a handle with only the methods that you want the parent component to call:

```
import { forwardRef, useRef, useImperativeHandle } from 'react';const MyInput = forwardRef(function MyInput(props, ref) { const inputRef = useRef(null); useImperativeHandle(ref, () => { return { focus() { inputRef.current.focus(); }, scrollIntoView() { inputRef.current.scrollIntoView(); }; }; }, []); return <input {...props} ref={inputRef} />});
```

Now, if the parent component gets a ref to MyInput, it will be able to call the focus and scrollIntoView methods on it. However, it will not have full access to the underlying <input> DOM node.

```
App.jsMyInput.jsApp.js ResetForkimport { useRef } from 'react';
```

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

```
export default function Form() {
```

```
  const ref = useRef(null);
```

```
  function handleClick() {
```

```
    ref.current.focus();
```

```
    // This won't work because the DOM node isn't exposed:
```

```
    // ref.current.style.opacity = 0.5;
```

```
}
```

```
  return (
```

```
    <form>
```

```
      <MyInput placeholder="Enter your name" ref={ref} />
```

```
      <button type="button" onClick={handleClick}>
```

Edit

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


```

Show more

Exposing your own imperative methods

The methods you expose via an imperative handle don't have to match the DOM methods exactly. For example, this Post component exposes a scrollAndFocusAddComment method via an imperative handle. This lets the parent Page scroll the list of comments and focus the input field when you click the button:

```
App.jsPost.jsCommentList.jsAddComment.jsApp.js ResetForkimport { useRef } from 'react';
import Post from './Post.js';


```

```
export default function Page() {
  const postRef = useRef(null);

  function handleClick() {
    postRef.current.scrollAndFocusAddComment();
  }

  return (
    <>
      <button onClick={handleClick}>
        Write a comment
      </button>
      <Post ref={postRef} />
    </>
  );
}
```

Show more

Pitfall Do not overuse refs. You should only use refs for imperative behaviors that you can't express as props: for example, scrolling to a node, focusing a node, triggering an animation, selecting text, and so on. If you can express something as a prop, you should not use a ref. For example, instead of exposing an imperative handle like { open, close } from a Modal component, it is better to take isOpen as a prop like <Modal isOpen={isOpen} />. Effects can

help you expose imperative behaviors via props.PrevioususeIdNextuseInsertionEffect©2024no uwu plzuwu?Logo
by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape
HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs
ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReference
useImperativeHandle(ref, createHandle, dependencies?) Usage Exposing a custom ref handle to the parent
component Exposing your own imperative methods useMemo –
ReactReactv18.3.1Search⌘CtrlKLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useState - This
feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId
useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the
latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>)
<Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext
forwardRef lazy memo startTransition use - This feature is available in the latest
Canaryexperimental_taintObjectReference - This feature is available in the latest
Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks
useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div>) <form> - This feature
is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in
the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest
Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs
createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS
- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule -
This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule -
This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot
Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup
renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls
Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in
the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in
the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the
latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory
createRef isValidElement PureComponent Is this page useful?API ReferenceHooksuseMemouseMemo is a React
Hook that lets you cache the result of a calculation between re-renders.const cachedValue =
useMemo(calculateValue, dependencies)

Reference useMemo(calculateValue, dependencies) Usage Skipping expensive recalculations Skipping re-rendering
of components Memoizing a dependency of another Hook Memoizing a function Troubleshooting My calculation
runs twice on every re-render My useMemo call is supposed to return an object, but returns undefined Every time
my component renders, the calculation in useMemo re-runs I need to call useMemo for each list item in a loop, but
it's not allowed

Reference

useMemo(calculateValue, dependencies)

Call useMemo at the top level of your component to cache a calculation between re-renders:

```
import { useMemo } from 'react';function TodoList({ todos, tab }) { const visibleTodos = useMemo( () =>  
filterTodos(todos, tab), [todos, tab] ); // ...}
```

See more examples below.

Parameters

`calculateValue`: The function calculating the value that you want to cache. It should be pure, should take no arguments, and should return a value of any type. React will call your function during the initial render. On next renders, React will return the same value again if the dependencies have not changed since the last render. Otherwise, it will call `calculateValue`, return its result, and store it so it can be reused later.

`dependencies`: The list of all reactive values referenced inside of the `calculateValue` code. Reactive values include `props`, `state`, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the `Object.is` comparison.

Returns

On the initial render, `useMemo` returns the result of calling `calculateValue` with no arguments.

During next renders, it will either return an already stored value from the last render (if the dependencies haven't changed), or call `calculateValue` again, and return the result that `calculateValue` has returned.

Caveats

`useMemo` is a Hook, so you can only call it at the top level of your component or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

In Strict Mode, React will call your calculation function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your calculation function is pure (as it should be), this should not affect your logic. The result from one of the calls will be ignored.

React will not throw away the cached value unless there is a specific reason to do that. For example, in development, React throws away the cache when you edit the file of your component. Both in development and in production, React will throw away the cache if your component suspends during the initial mount. In the future, React may add more features that take advantage of throwing away the cache—for example, if React adds built-in support for virtualized lists in the future, it would make sense to throw away the cache for items that scroll out of the virtualized table viewport. This should be fine if you rely on `useMemo` solely as a performance optimization. Otherwise, a state variable or a ref may be more appropriate.

Note Caching return values like this is also known as memoization, which is why this Hook is called `useMemo`.

Usage

Skipping expensive recalculations

To cache a calculation between re-renders, wrap it in a `useMemo` call at the top level of your component:

```
import { useMemo } from 'react';
function TodoList({ todos, tab, theme }) {
  const visibleTodos = useMemo(() =>
    filterTodos(todos, tab),
    [todos, tab]
  );
}
```

You need to pass two things to `useMemo`:

A calculation function that takes no arguments, like `() =>`, and returns what you wanted to calculate.

A list of dependencies including every value within your component that's used inside your calculation.

On the initial render, the value you'll get from `useMemo` will be the result of calling your calculation.

On every subsequent render, React will compare the dependencies with the dependencies you passed during the last render. If none of the dependencies have changed (compared with `Object.is`), `useMemo` will return the value you already calculated before. Otherwise, React will re-run your calculation and return the new value.

In other words, `useMemo` caches a calculation result between re-renders until its dependencies change.

Let's walk through an example to see when this is useful.

By default, React will re-run the entire body of your component every time that it re-renders. For example, if this `TodoList` updates its state or receives new props from its parent, the `filterTodos` function will re-run:

```
function TodoList({ todos, tab, theme }) { const visibleTodos = filterTodos(todos, tab); // ...}
```

Usually, this isn't a problem because most calculations are very fast. However, if you're filtering or transforming a large array, or doing some expensive computation, you might want to skip doing it again if data hasn't changed. If both `todos` and `tab` are the same as they were during the last render, wrapping the calculation in `useMemo` like earlier lets you reuse `visibleTodos` you've already calculated before.

This type of caching is called memoization.

Note You should only rely on `useMemo` as a performance optimization. If your code doesn't work without it, find the underlying problem and fix it first. Then you may add `useMemo` to improve performance.

Deep Dive How to tell if a calculation is expensive? Show Details In 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 = filterTodos(todos, tab);`
`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:
`const visibleTodos = useMemo(() => { return filterTodos(todos, tab); }, [todos, tab]);`
`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.

Deep Dive Should you add `useMemo` everywhere? Show Details If your app is like this site, and most interactions are coarse (like replacing a page or an entire section), memoization is usually unnecessary. On the other hand, if your app is more like a drawing editor, and most interactions are granular (like moving shapes), then you might find memoization very helpful. Optimizing with `useMemo` is only valuable in a few cases:

The calculation you're putting in `useMemo` is noticeably slow, and its dependencies rarely change.

You pass it as a prop to a component wrapped in `memo`. You want to skip re-rendering if the value hasn't changed. Memoization lets your component re-render only when dependencies aren't the same.

The value you're passing is later used as a dependency of some Hook. For example, maybe another `useMemo` calculation value depends on it. Or maybe you are depending on this value from `useEffect`.

There is no benefit to wrapping a calculation in `useMemo` in other cases. There is no significant harm to doing that either, so some teams choose to not think about individual cases, and memoize as much as possible. The downside of this approach is that code becomes less readable. Also, not all memoization is effective: a single value that's "always new" is enough to break memoization for an entire component. In practice, you can make a lot of memoization unnecessary by following a few principles:

When a component visually wraps other components, let it accept JSX as children. This way, when the wrapper component updates its own state, React knows that its children don't need to re-render.

Prefer local state and don't lift state up any further than necessary. For example, don't keep transient state like forms and whether an item is hovered at the top of your tree or in a global state library.

Keep your rendering logic pure. If re-rendering a component causes a problem or produces some noticeable visual artifact, it's a bug in your component! Fix the bug instead of adding memoization.

Avoid unnecessary Effects that update state. Most performance problems in React apps are caused by chains of updates originating from Effects that cause your components to render over and over.

Try to remove unnecessary dependencies from your Effects. For example, instead of memoization, it's often simpler to move some object or a function inside an Effect or outside the component.

If a specific interaction still feels laggy, use the React Developer Tools profiler to see which components would benefit the most from memoization, and add memoization where needed. These principles make your components easier to debug and understand, so it's good to follow them in any case. In the long term, we're researching doing granular memoization automatically to solve this once and for all.

The difference between `useMemo` and calculating a value directly¹. Skipping recalculation with `useMemo` 2. Always recalculating a value Example 1 of 2: Skipping recalculation with `useMemo` In this example, the `filterTodos` implementation is artificially slowed down so that you can see what happens when some JavaScript function you're calling during rendering is genuinely slow. Try switching the tabs and toggling the theme. Switching the tabs feels slow because it forces the slowed down `filterTodos` to re-execute. That's expected because the tab has changed, and so the entire calculation needs to re-run. (If you're curious why it runs twice, it's explained here.) Toggle the theme. Thanks to `useMemo`, it's fast despite the artificial slowdown! The slow `filterTodos` call was skipped because both `todos` and `tab` (which you pass as dependencies to `useMemo`) haven't changed since the last render.

```
App.js
import { TodoList } from './TodoList';
import { useState } from 'react';
import { useTodos } from './useTodos';
import { useTheme } from './useTheme';
import { filterTodos } from './utils';

const ResetFork = () => {
  const [ todos, setTodos ] = useState(useTodos());
  const [ theme, setTheme ] = useState(useTheme());
  const [ tab, setTab ] = useState('all');

  const visibleTodos = useMemo(() => filterTodos(todos, tab), [ todos, tab ]);

  return (
    <div>
      <p><b>Note:</b> <code>filterTodos</code> is artificially slowed down!</p>
      <ul>
        {visibleTodos.map(todo => (
          <li key={todo.id}>
            {todo.text}
          </li>
        ))
      }
    </div>
  );
}

ResetFork.propTypes = {
  todos: PropTypes.array.isRequired,
  theme: PropTypes.string.isRequired,
  tab: PropTypes.string.isRequired,
};

ResetFork.defaultProps = {
  todos: [],
  theme: 'light',
  tab: 'all',
};
```

```
TodoList.js
export default function TodoList({ todos, theme, tab }) {
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [ todos, tab ]);

  return (
    <div className={theme}>
      <p><b>Note:</b> <code>filterTodos</code> is artificially slowed down!</p>
      <ul>
        {visibleTodos.map(todo => (
          <li key={todo.id}>
            {todo.text}
          </li>
        ))
      }
    </div>
  );
}
```

```

{todo.completed ?
  <s>{todo.text}</s>:
  todo.text
}
</li>
)})
</ul>
</div>
);
}

```

[Show more](#)[Next Example](#)

Skipping re-rendering of components

In some cases, `useMemo` can also help you optimize performance of re-rendering child components. To illustrate this, let's say this `TodoList` component passes the `visibleTodos` as a prop to the child `List` component:

```
export default function TodoList({ todos, tab, theme }) { // ... return ( <div className={theme}> <List items={visibleTodos} /> </div> );}
```

You've noticed that toggling the `theme` prop freezes the app for a moment, but if you remove `<List />` from your JSX, it feels fast. This tells you that it's worth trying to optimize the `List` component.

By default, when a component re-renders, React re-renders all of its children recursively. This is why, when `TodoList` re-renders with a different theme, the `List` component also re-renders. This is fine for components that don't require much calculation to re-render. But if you've verified that a re-render is slow, you can tell `List` to skip re-rendering when its props are the same as on last render by wrapping it in `memo`:

```
import { memo } from 'react';const List = memo(function List({ items }) { // ...});
```

With this change, `List` will skip re-rendering if all of its props are the same as on the last render. This is where caching the calculation becomes important! Imagine that you calculated `visibleTodos` without `useMemo`:

```
export default function TodoList({ todos, tab, theme }) { // Every time the theme changes, this will be a different array... const visibleTodos = filterTodos(todos, tab); return ( <div className={theme}> /* ... so List's props will never be the same, and it will re-render every time */ <List items={visibleTodos} /> </div> );}
```

In the above example, the `filterTodos` function always creates a different array, similar to how the `{}` object literal always creates a new object. Normally, this wouldn't be a problem, but it means that `List` props will never be the same, and your `memo` optimization won't work. This is where `useMemo` comes in handy:

```
export default function TodoList({ todos, tab, theme }) { // Tell React to cache your calculation between re-renders... const visibleTodos = useMemo( () => filterTodos(todos, tab), [todos, tab] // ...so as long as these dependencies don't change... ); return ( <div className={theme}> /* ...List will receive the same props and can skip re-rendering */ <List items={visibleTodos} /> </div> );}
```

By wrapping the `visibleTodos` calculation in `useMemo`, you ensure that it has the same value between the re-renders (until dependencies change). You don't have to wrap a calculation in `useMemo` unless you do it for some specific

reason. In this example, the reason is that you pass it to a component wrapped in memo, and this lets it skip re-rendering. There are a few other reasons to add useMemo which are described further on this page.

Deep DiveMemoizing individual JSX nodes Show DetailsInstead of wrapping List in memo, you could wrap the <List /> JSX node itself in useMemo:
export default function TodoList({ todos, tab, theme }) { const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]); const children = useMemo(() => <List items={visibleTodos} />, [visibleTodos]); return (<div className={theme}> {children} </div>); }The behavior would be the same. If the visibleTodos haven't changed, List won't be re-rendered. A JSX node like <List items={visibleTodos} /> is an object like { type: List, props: { items: visibleTodos } }. Creating this object is very cheap, but React doesn't know whether its contents is the same as last time or not. This is why by default, React will re-render the List component. However, if React sees the same exact JSX as during the previous render, it won't try to re-render your component. This is because JSX nodes are immutable. A JSX node object could not have changed over time, so React knows it's safe to skip a re-render. However, for this to work, the node has to actually be the same object, not merely look the same in code. This is what useMemo does in this example. Manually wrapping JSX nodes into useMemo is not convenient. For example, you can't do this conditionally. This is usually why you would wrap components with memo instead of wrapping JSX nodes.

The difference between skipping re-renders and always re-rendering
1. Skipping re-rendering with useMemo and memo
2. Always re-rendering a component Example 1 of 2: Skipping re-rendering with useMemo and memo In this example, the List component is artificially slowed down so that you can see what happens when a React component you're rendering is genuinely slow. Try switching the tabs and toggling the theme. Switching the tabs feels slow because it forces the slowed down List to re-render. That's expected because the tab has changed, and so you need to reflect the user's new choice on the screen. Next, try toggling the theme. Thanks to useMemo together with memo, it's fast despite the artificial slowdown! The List skipped re-rendering because the visibleTodos array has not changed since the last render. The visibleTodos array has not changed because both todos and tab (which you pass as dependencies to useMemo) haven't changed since the last render.
App.js
TodoList.js
List.js
utils.js
TodoList.js
ResetFork
import { useMemo } from 'react';

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

```
import { filterTodos } from './utils.js'
```

```
export default function TodoList({ todos, theme, tab }) {
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab]
  );
  return (
    <div className={theme}>
      <p><b>Note: <code>List</code> is artificially slowed down!</b></p>
      <List items={visibleTodos} />
    </div>
  );
}
```

Show more
Next Example

Memoizing a dependency of another Hook

Suppose you have a calculation that depends on an object created directly in the component body:

```
function Dropdown({ allItems, text }) { const searchOptions = { matchMode: 'whole-word', text }; const visibleItems = useMemo(() => { return searchItems(allItems, searchOptions); }, [allItems, searchOptions]); // Caution: Dependency on an object created in the component body // ...}
```

Depending on an object like this defeats the point of memoization. When a component re-renders, all of the code directly inside the component body runs again. The lines of code creating the searchOptions object will also run on every re-render. Since searchOptions is a dependency of your useMemo call, and it's different every time, React knows the dependencies are different, and recalculate searchItems every time.

To fix this, you could memoize the searchOptions object itself before passing it as a dependency:

```
function Dropdown({ allItems, text }) { const searchOptions = useMemo(() => { return { matchMode: 'whole-word', text }; }, [text]); // tickmark Only changes when text changes const visibleItems = useMemo(() => { return searchItems(allItems, searchOptions); }, [allItems, searchOptions]); // tickmark Only changes when allItems or searchOptions changes // ...}
```

In the example above, if the text did not change, the searchOptions object also won't change. However, an even better fix is to move the searchOptions object declaration inside of the useMemo calculation function:

```
function Dropdown({ allItems, text }) { const visibleItems = useMemo(() => { const searchOptions = { matchMode: 'whole-word', text }; return searchItems(allItems, searchOptions); }, [allItems, text]); // tickmark Only changes when allItems or text changes // ...}
```

Now your calculation depends on text directly (which is a string and can't "accidentally" become different).

Memoizing a function

Suppose the Form component is wrapped in memo. You want to pass a function to it as a prop:

```
export default function ProductPage({ productId, referrer }) { function handleSubmit(orderDetails) { post('/product/' + productId + '/buy', { referrer, orderDetails }); } return <Form onSubmit={handleSubmit} />;}
```

Just as {} creates a different object, function declarations like function() {} and expressions like () => {} produce a different function on every re-render. By itself, creating a new function is not a problem. This is not something to avoid! However, if the Form component is memoized, presumably you want to skip re-rendering it when no props have changed. A prop that is always different would defeat the point of memoization.

To memoize a function with useMemo, your calculation function would have to return another function:

```
export default function Page({ productId, referrer }) { const handleSubmit = useMemo(() => { return (orderDetails) => { post('/product/' + productId + '/buy', { referrer, orderDetails }); }; }, [productId, referrer]); return <Form onSubmit={handleSubmit} />;}
```

This looks clunky! Memoizing functions is common enough that React has a built-in Hook specifically for that. Wrap your functions into useCallback instead of useMemo to avoid having to write an extra nested function:

```
export default function Page({ productId, referrer }) { const handleSubmit = useCallback((orderDetails) => { post('/product/' + productId + '/buy', { referrer, orderDetails }); }, [productId, referrer]); return <Form onSubmit={handleSubmit} />;}
```

The two examples above are completely equivalent. The only benefit to useCallback is that it lets you avoid writing an extra nested function inside. It doesn't do anything else. Read more about useCallback.

Troubleshooting

My calculation runs twice on every re-render

In Strict Mode, React will call some of your functions twice instead of once:

```
function TodoList({ todos, tab }) { // This component function will run twice for every render. const visibleTodos = useMemo(() => { // This calculation will run twice if any of the dependencies change. return filterTodos(todos, tab); }, [todos, tab]); // ...}
```

This is expected and shouldn't break your code.

This development-only behavior helps you keep components pure. React uses the result of one of the calls, and ignores the result of the other call. As long as your component and calculation functions are pure, this shouldn't affect your logic. However, if they are accidentally impure, this helps you notice and fix the mistake.

For example, this impure calculation function mutates an array you received as a prop:

```
const visibleTodos = useMemo(() => { // Mistake: mutating a prop todos.push({ id: 'last', text: 'Go for a walk!' }); const filtered = filterTodos(todos, tab); return filtered; }, [todos, tab]);
```

React calls your function twice, so you'd notice the todo is added twice. Your calculation shouldn't change any existing objects, but it's okay to change any new objects you created during the calculation. For example, if the filterTodos function always returns a different array, you can mutate that array instead:

```
const visibleTodos = useMemo(() => { const filtered = filterTodos(todos, tab); // tickmark Correct: mutating an object you created during the calculation filtered.push({ id: 'last', text: 'Go for a walk!' }); return filtered; }, [todos, tab]);
```

Read [keeping components pure](#) to learn more about purity.

Also, check out the guides on [updating objects](#) and [updating arrays without mutation](#).

My useMemo call is supposed to return an object, but returns undefined

This code doesn't work:

```
// You can't return an object from an arrow function with () => { const searchOptions = useMemo(() => { matchMode: 'whole-word', text: text }, [text]);
```

In JavaScript, () => { starts the arrow function body, so the { brace is not a part of your object. This is why it doesn't return an object, and leads to mistakes. You could fix it by adding parentheses like ({ and }):

```
// This works, but is easy for someone to break again const searchOptions = useMemo(() => ({ matchMode: 'whole-word', text: text }), [text]);
```

However, this is still confusing and too easy for someone to break by removing the parentheses.

To avoid this mistake, write a return statement explicitly:

```
// tickmark This works and is explicit const searchOptions = useMemo(() => { return { matchMode: 'whole-word', text: text }; }, [text]);
```

Every time my component renders, the calculation in useMemo re-runs

Make sure you've specified the dependency array as a second argument!

If you forget the dependency array, useMemo will re-run the calculation every time:

```
function TodoList({ todos, tab }) { // Recalculates every time: no dependency array const visibleTodos = useMemo(() => filterTodos(todos, tab)); // ...
```

This is the corrected version passing the dependency array as a second argument:

```
function TodoList({ todos, tab }) { // tickmark Does not recalculate unnecessarily const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]); // ...
```

If this doesn't help, then the problem is that at least one of your dependencies is different from the previous render. You can debug this problem by manually logging your dependencies to the console:

```
const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]); console.log([todos, tab]);
```

You can then right-click on the arrays from different re-renders in the console and select "Store as a global variable" for both of them. Assuming the first one got saved as temp1 and the second one got saved as temp2, you can then use the browser console to check whether each dependency in both arrays is the same:

```
Object.is(temp1[0], temp2[0]); // Is the first dependency the same between the arrays? Object.is(temp1[1], temp2[1]); // Is the second dependency the same between the arrays? Object.is(temp1[2], temp2[2]); // ... and so on for every dependency ...
```

When you find which dependency breaks memoization, either find a way to remove it, or memoize it as well.

I need to call useMemo for each list item in a loop, but it's not allowed

Suppose the Chart component is wrapped in memo. You want to skip re-rendering every Chart in the list when the ReportList component re-renders. However, you can't call useMemo in a loop:

```
function ReportList({ items }) { return ( <article> {items.map(item => { // You can't call useMemo in a loop like this: const data = useMemo(() => calculateReport(item), [item]); return ( <figure key={item.id}> <Chart data={data} /> </figure> ); }) } </article> );}
```

Instead, extract a component for each item and memoize data for individual items:

```
function ReportList({ items }) { return ( <article> {items.map(item => <Report key={item.id} item={item} />) } </article> );} function Report({ item }) { // tickmark Call useMemo at the top level: const data = useMemo(() => calculateReport(item), [item]); return ( <figure> <Chart data={data} /> </figure> );}
```

Alternatively, you could remove useMemo and instead wrap Report itself in memo. If the item prop does not change, Report will skip re-rendering, so Chart will skip re-rendering too:

```
function ReportList({ items }) { // ...} const Report = memo(function Report({ item }) { const data = calculateReport(item); return ( <figure> <Chart data={data} /> </figure> );}); Previous useLayoutEffect Next useOptimistic © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Reference useMemo(calculateValue, dependencies) Usage Skipping expensive recalculations Skipping re-rendering of components Memoizing a dependency of another Hook Memoizing a function Troubleshooting My calculation runs twice on every re-render My useMemo call is supposed to return an object, but returns undefined Every time my component renders, the calculation in useMemo re-runs I need to call useMemo for each list item in a loop, but it's not allowed useReducer – React React v18.3.1 Search Ctrl+K Learn Reference Community Blog react@18.3.1 Overview Hooks useState – This feature is available in the latest Canary useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic – This feature is available in the latest Canary useRef useState useSyncExternalStore useTransition Components <Fragment> (<> <Profiler> <StrictMode> <Suspense> APIs act cache – This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use – This feature is available in the latest
```

Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceHooksuseReduceruseReducer is a React Hook that lets you add a reducer to your component.const [state, dispatch] = useReducer(reducer, initialArg, init?)

Reference useReducer(reducer, initialArg, init?) dispatch function Usage Adding a reducer to a component Writing the reducer function Avoiding recreating the initial state Troubleshooting I've dispatched an action, but logging gives me the old state value I've dispatched an action, but the screen doesn't update A part of my reducer state becomes undefined after dispatching My entire reducer state becomes undefined after dispatching I'm getting an error: "Too many re-renders" My reducer or initializer function runs twice

Reference

`useReducer(reducer, initialArg, init?)`

Call `useReducer` at the top level of your component to manage its state with a reducer.

```
import { useReducer } from 'react';function reducer(state, action) { // ...}function MyComponent() { const [state, dispatch] = useReducer(reducer, { age: 42 }); // ...}
```

See more examples below.

Parameters

`reducer`: The reducer function that specifies how the state gets updated. It must be pure, should take the state and action as arguments, and should return the next state. State and action can be of any types.

`initialArg`: The value from which the initial state is calculated. It can be a value of any type. How the initial state is calculated from it depends on the next `init` argument.

`optional init`: The initializer function that should return the initial state. If it's not specified, the initial state is set to `initialArg`. Otherwise, the initial state is set to the result of calling `init(initialArg)`.

Returns

`useReducer` returns an array with exactly two values:

The current state. During the first render, it's set to `init(initialArg)` or `initialArg` (if there's no `init`).

The dispatch function that lets you update the state to a different value and trigger a re-render.

Caveats

useReducer is a Hook, so you can only call it at the top level of your component or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

In Strict Mode, React will call your reducer and initializer twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your reducer and initializer are pure (as they should be), this should not affect your logic. The result from one of the calls is ignored.

dispatch function

The dispatch function returned by useReducer lets you update the state to a different value and trigger a re-render. You need to pass the action as the only argument to the dispatch function:

```
const [state, dispatch] = useReducer(reducer, { age: 42 });
function handleClick() {
  dispatch({ type: 'incremented_age' });
} // ...
```

React will set the next state to the result of calling the reducer function you've provided with the current state and the action you've passed to dispatch.

Parameters

action: The action performed by the user. It can be a value of any type. By convention, an action is usually an object with a type property identifying it and, optionally, other properties with additional information.

Returns

dispatch functions do not have a return value.

Caveats

The dispatch function only updates the state variable for the next render. If you read the state variable after calling the dispatch function, you will still get the old value that was on the screen before your call.

If the new value you provide is identical to the current state, as determined by an Object.is comparison, React will skip re-rendering the component and its children. This is an optimization. React may still need to call your component before ignoring the result, but it shouldn't affect your code.

React batches state updates. It updates the screen after all the event handlers have run and have called their set functions. This prevents multiple re-renders during a single event. In the rare case that you need to force React to update the screen earlier, for example to access the DOM, you can use flushSync.

Usage

Adding a reducer to a component

Call useReducer at the top level of your component to manage state with a reducer.

```
import { useReducer } from 'react';function reducer(state, action) { // ...}function MyComponent() { const [state, dispatch] = useReducer(reducer, { age: 42 }); // ...}
```

useReducer returns an array with exactly two items:

The current state of this state variable, initially set to the initial state you provided.

The dispatch function that lets you change it in response to interaction.

To update what's on the screen, call dispatch with an object representing what the user did, called an action:

```
function handleClick() { dispatch({ type: 'incremented_age' });}
```

React will pass the current state and the action to your reducer function. Your reducer will calculate and return the next state. React will store that next state, render your component with it, and update the UI.

```
App.jsApp.js ResetForkimport { useReducer } from 'react';
```

```
function reducer(state, action) {  
  if (action.type === 'incremented_age') {  
    return {  
      age: state.age + 1  
    };  
  }  
  throw Error('Unknown action.');//
```

```
}  
  
export default function Counter() {  
  const [state, dispatch] = useReducer(reducer, { age: 42 });  
  
  return (  
    <>
```

```

<button onClick={() => {
  dispatch({ type: 'incremented_age' })
}}>
  Increment age
</button>

<p>Hello! You are {state.age}.</p>
</>
);
}

```

Show more

`useReducer` is very similar to `useState`, but it lets you move the state update logic from event handlers into a single function outside of your component. Read more about choosing between `useState` and `useReducer`.

Writing the reducer function

A reducer function is declared like this:

```
function reducer(state, action) { // ...}
```

Then you need to fill in the code that will calculate and return the next state. By convention, it is common to write it as a switch statement. For each case in the switch, calculate and return some next state.

```
function reducer(state, action) { switch (action.type) { case 'incremented_age': { return { name: state.name, age: state.age + 1 }; } case 'changed_name': { return { name: action.nextName, age: state.age }; } } throw Error('Unknown action: ' + action.type);}
```

Actions can have any shape. By convention, it's common to pass objects with a `type` property identifying the action. It should include the minimal necessary information that the reducer needs to compute the next state.

```
function Form() { const [state, dispatch] = useReducer(reducer, { name: 'Taylor', age: 42 }); function handleButtonClick() { dispatch({ type: 'incremented_age' }); } function handleInputChange(e) { dispatch({ type: 'changed_name', nextName: e.target.value }); } // ...}
```

The action type names are local to your component. Each action describes a single interaction, even if that leads to multiple changes in data. The shape of the state is arbitrary, but usually it'll be an object or an array.

Read extracting state logic into a reducer to learn more.

`PitfallState` is read-only. Don't modify any objects or arrays in state:
`function reducer(state, action) { switch (action.type) { case 'incremented_age': { // Don't mutate an object in state like this: state.age = state.age + 1; return state; }Instead, always return new objects from your reducer:function reducer(state, action) { switch (action.type) { case 'incremented_age': { // tickmark Instead, return a new object return { ...state, age: state.age + 1 }; }Read updating objects in state and updating arrays in state to learn more.`

Basic `useReducer` examples
1. Form (object)
2. Todo list (array)
3. Writing concise update logic with Immer
Example 1 of 3: Form (object)
In this example, the reducer manages a state object with two fields: `name` and `age`.
App.js
`import { useReducer } from 'react';`

```
function reducer(state, action) {  
  switch (action.type) {  
    case 'incremented_age': {  
      return {  
        name: state.name,  
        age: state.age + 1  
      };  
    }  
    case 'changed_name': {  
      return {  
        name: action.nextName,  
        age: state.age  
      };  
    }  
  }  
  throw Error('Unknown action: ' + action.type);  
}  
}
```

```
const initialState = { name: 'Taylor', age: 42 };
```

```
export default function Form() {  
  const [state, dispatch] = useReducer(reducer, initialState);
```

```
  function handleButtonClick() {  
    dispatch({ type: 'incremented_age' });  
  }
```

```
  function handleInputChange(e) {  
    dispatch({  
      type: 'changed_name',  
      nextName: e.target.value  
    });  
  }
```

```

return (
  <>
  <input
    value={state.name}
    onChange={handleInputChange}
  />
  <button onClick={handleButtonClick}>
    Increment age
  </button>
  <p>Hello, {state.name}. You are {state.age}.</p>
</>
);
}

```

[Show more](#)[Next Example](#)

Avoiding recreating the initial state

React saves the initial state once and ignores it on the next renders.

```
function createInitialState(username) { // ...}function TodoList({ username }) { const [state, dispatch] =
useReducer(reducer, createInitialState(username)); // ...
```

Although the result of `createInitialState(username)` is only used for the initial render, you're still calling this function on every render. This can be wasteful if it's creating large arrays or performing expensive calculations.

To solve this, you may pass it as an initializer function to `useReducer` as the third argument instead:

```
function createInitialState(username) { // ...}function TodoList({ username }) { const [state, dispatch] =
useReducer(reducer, username, createInitialState); // ...
```

Notice that you're passing `createInitialState`, which is the function itself, and not `createInitialState()`, which is the result of calling it. This way, the initial state does not get re-created after initialization.

In the above example, `createInitialState` takes a `username` argument. If your initializer doesn't need any information to compute the initial state, you may pass null as the second argument to `useReducer`.

The difference between passing an initializer and passing the initial state directly1. Passing the initializer function 2. Passing the initial state directly Example 1 of 2: Passing the initializer function This example passes the initializer function, so the `createInitialState` function only runs during initialization. It does not run when component re-renders, such as when you type into the input.

```
TodoList.js
import { useState } from 'react';
import { useReducer } from 'react';

function createTodos() {
  const todos = [];
  for (let i = 0; i < 50; i++) {
    todos.push(`Todo ${i + 1}`);
  }
  return todos;
}

function reducer(state, action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state, action.payload];
    case 'DELETE_TODO':
      return state.filter((todo, index) => index !== action.payload);
    default:
      return state;
  }
}

function TodoList() {
  const [todos, setTodos] = useState(createTodos());
  const [count, setCount] = useState(0);

  const handleAddTodo = () => {
    setCount(count + 1);
  };

  const handleDeleteTodo = (index) => {
    setTodos(todos.filter((todo, i) => i !== index));
  };

  return (
    <div>
      <h1>Todos</h1>
      <ul>
        {todos.map((todo, index) => (
          <li key={index}>{todo}</li>
        ))}
      </ul>
      <button onClick={handleAddTodo}>Add Todo</button>
      <button onClick={() => handleDeleteTodo(count)}>Delete Todo</button>
    </div>
  );
}

export default TodoList;
```

```
initialTodos.push({
  id: i,
  text: username + "'s task #" + (i + 1)
});
}

return {
  draft: '',
  todos: initialTodos,
};

}

function reducer(state, action) {
  switch (action.type) {
    case 'changed_draft': {
      return {
        draft: action.nextDraft,
        todos: state.todos,
      };
    };
    case 'added_todo': {
      return {
        draft: '',
        todos: [
          {
            id: state.todos.length,
            text: state.draft
          }, ...state.todos
        ]
      }
    }
  }
  throw Error('Unknown action: ' + action.type);
}

export default function TodoList({ username }) {
  const [state, dispatch] = useReducer(
```

```
reducer,  
username,  
createInitialState  
);  
return (  
<>  
<input  
value={state.draft}  
onChange={e => {  
  dispatch({  
    type: 'changed_draft',  
    nextDraft: e.target.value  
  })  
}}  
>  
</>  
<button onClick={() => {  
  dispatch({ type: 'added_todo' });  
}}>Add</button>  
<ul>  
{state.todos.map(item => (  
  <li key={item.id}>  
    {item.text}  
  </li>  
>))}  
</ul>  
</>  
);  
}
```

Show moreNext Example

Troubleshooting

I've dispatched an action, but logging gives me the old state value

Calling the dispatch function does not change state in the running code:

```
function handleClick() { console.log(state.age); // 42 dispatch({ type: 'incremented_age' }); // Request a re-render with 43 console.log(state.age); // Still 42! setTimeout(() => { console.log(state.age); // Also 42! }, 5000);}
```

This is because states behaves like a snapshot. Updating state requests another render with the new state value, but does not affect the state JavaScript variable in your already-running event handler.

If you need to guess the next state value, you can calculate it manually by calling the reducer yourself:

```
const action = { type: 'incremented_age' }; dispatch(action); const nextState = reducer(state, action); console.log(state); // { age: 42 } console.log(nextState); // { age: 43 }
```

I've dispatched an action, but the screen doesn't update

React will ignore your update if the next state is equal to the previous state, as determined by an Object.is comparison. This usually happens when you change an object or an array in state directly:

```
function reducer(state, action) { switch (action.type) { case 'incremented_age': { // Wrong: mutating existing object state.age++; return state; } case 'changed_name': { // Wrong: mutating existing object state.name = action.nextName; return state; } // ... }}
```

You mutated an existing state object and returned it, so React ignored the update. To fix this, you need to ensure that you're always updating objects in state and updating arrays in state instead of mutating them:

```
function reducer(state, action) { switch (action.type) { case 'incremented_age': { // tickmark Correct: creating a new object return { ...state, age: state.age + 1 }; } case 'changed_name': { // tickmark Correct: creating a new object return { ...state, name: action.nextName }; } // ... }}
```

A part of my reducer state becomes undefined after dispatching

Make sure that every case branch copies all of the existing fields when returning the new state:

```
function reducer(state, action) { switch (action.type) { case 'incremented_age': { return { ...state, // Don't forget this! age: state.age + 1 }; } // ... }}
```

Without ...state above, the returned next state would only contain the age field and nothing else.

My entire reducer state becomes undefined after dispatching

If your state unexpectedly becomes undefined, you're likely forgetting to return state in one of the cases, or your action type doesn't match any of the case statements. To find why, throw an error outside the switch:

```
function reducer(state, action) { switch (action.type) { case 'incremented_age': { // ... } case 'edited_name': { // ... } } throw Error('Unknown action: ' + action.type);}
```

You can also use a static type checker like TypeScript to catch such mistakes.

I'm getting an error: "Too many re-renders"

You might get an error that says: Too many re-renders. React limits the number of renders to prevent an infinite loop. Typically, this means that you're unconditionally dispatching an action during render, so your component enters a loop: render, dispatch (which causes a render), render, dispatch (which causes a render), and so on. Very often, this is caused by a mistake in specifying an event handler:

```
// Wrong: calls the handler during render return <button onClick={handleClick()}>Click me</button> // tickmark  
Correct: passes down the event handler return <button onClick={handleClick}>Click me</button> // tickmark  
Correct: passes down an inline function return <button onClick={(e) => handleClick(e)}>Click me</button>
```

If you can't find the cause of this error, click on the arrow next to the error in the console and look through the JavaScript stack to find the specific dispatch function call responsible for the error.

My reducer or initializer function runs twice

In Strict Mode, React will call your reducer and initializer functions twice. This shouldn't break your code.

This development-only behavior helps you keep components pure. React uses the result of one of the calls, and ignores the result of the other call. As long as your component, initializer, and reducer functions are pure, this shouldn't affect your logic. However, if they are accidentally impure, this helps you notice the mistakes.

For example, this impure reducer function mutates an array in state:

```
function reducer(state, action) { switch (action.type) { case 'added_todo': { // Mistake: mutating state  
state.todos.push({ id: nextId++, text: action.text }); return state; } // ... }}
```

Because React calls your reducer function twice, you'll see the todo was added twice, so you'll know that there is a mistake. In this example, you can fix the mistake by replacing the array instead of mutating it:

```
function reducer(state, action) { switch (action.type) { case 'added_todo': { // tickmark Correct: replacing with  
new state return { ...state, todos: [ ...state.todos, { id: nextId++, text: action.text } ] }; }  
// ... }}
```

Now that this reducer function is pure, calling it an extra time doesn't make a difference in behavior. This is why React calling it twice helps you find mistakes. Only component, initializer, and reducer functions need to be pure. Event handlers don't need to be pure, so React will never call your event handlers twice.

Read keeping components pure to learn more. Previous useOptimisticNext useRef ©2024 no uwu plzuwu? Logo
by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Reference useReducer(reducer, initialArg, init?) dispatch function Usage Adding a reducer to a component Writing the reducer function Avoiding recreating the initial state Troubleshooting I've dispatched an action, but logging gives me the old state value I've dispatched an action, but the screen doesn't update A part of my reducer state becomes undefined after dispatching My entire reducer state becomes undefined after dispatching I'm getting an error: "Too many re-renders" My reducer or initializer function runs twice useRef – React React v18.3.1 Search Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks useActionState - This feature is available in the latest Canary useCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest Canary useReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>)<Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use - This feature is available in the latest Canary experimental_taintObjectReference - This feature is available in the latest Canary experimental_taintUniqueValue - This feature is available in the latest Canary react-dom@18.3.1 Hooks useFormStatus - This feature is available in the latest Canary Components Common (e.g. <div>) <form> - This feature is available in the latest Canary <input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary <meta> - This feature is available in the latest Canary <script> - This feature is available in the latest Canary <style> - This feature is available in the latest Canary <title> - This feature is available in the latest Canary APIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest Canary prefetchDNS - This feature is available in the latest Canary preinit - This feature is available in the latest Canary preinitModule - This feature is available in the latest Canary preload - This feature is available in the latest Canary preloadModule - This feature is available in the latest Canary render unmountComponentAtNode Client APIs createRoot hydrateRoot

Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup
renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls
Components and Hooks Rules of Hooks React Server Components Server Components - This feature is available in
the latest Canary Server Actions - This feature is available in the latest Canary Directives - This feature is available in
the latest Canary 'use client' - This feature is available in the latest Canary 'use server' - This feature is available in the
latest Canary Legacy APIs Legacy React APIs Children cloneElement Component createElement createFactory
createRef isValidElement PureComponent Is this page useful? API Reference Hooks useRef RefuseRef is a React Hook that
lets you reference a value that's not needed for rendering. const ref = useRef(initialValue)

Reference useRef(initialValue) Usage Referencing a value with a ref Manipulating the DOM with a ref Avoiding
recreating the ref contents Troubleshooting I can't get a ref to a custom component

Reference

useRef(initialValue)

Call useRef at the top level of your component to declare a ref.

```
import { useRef } from 'react';function MyComponent() { const intervalRef = useRef(0); const inputRef = useRef(null); // ...}
```

See more examples below.

Parameters

initialValue: The value you want the ref object's current property to be initially. It can be a value of any type. This argument is ignored after the initial render.

Returns

useRef returns an object with a single property:

current: Initially, it's set to the initialValue you have passed. You can later set it to something else. If you pass the ref object to React as a ref attribute to a JSX node, React will set its current property.

On the next renders, useRef will return the same object.

Caveats

You can mutate the ref.current property. Unlike state, it is mutable. However, if it holds an object that is used for rendering (for example, a piece of your state), then you shouldn't mutate that object.

When you change the ref.current property, React does not re-render your component. React is not aware of when you change it because a ref is a plain JavaScript object.

Do not write or read ref.current during rendering, except for initialization. This makes your component's behavior unpredictable.

In Strict Mode, React will call your component function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. Each ref object will be created twice, but one of the versions will be discarded. If your component function is pure (as it should be), this should not affect the behavior.

Usage

Referencing a value with a ref

Call `useRef` at the top level of your component to declare one or more refs.

```
import { useRef } from 'react';function Stopwatch() { const intervalRef = useRef(0); // ... }
```

`useRef` returns a ref object with a single `current` property initially set to the initial value you provided.

On the next renders, `useRef` will return the same object. You can change its `current` property to store information and read it later. This might remind you of state, but there is an important difference.

Changing a ref does not trigger a re-render. This means refs are perfect for storing information that doesn't affect the visual output of your component. For example, if you need to store an interval ID and retrieve it later, you can put it in a ref. To update the value inside the ref, you need to manually change its `current` property:

```
function handleStartClick() { const intervalId = setInterval(() => { // ... }, 1000); intervalRef.current = intervalId; }
```

Later, you can read that interval ID from the ref so that you can call clear that interval:

```
function handleStopClick() { const intervalId = intervalRef.current; clearInterval(intervalId); }
```

By using a ref, you ensure that:

You can store information between re-renders (unlike regular variables, which reset on every render).

Changing it does not trigger a re-render (unlike state variables, which trigger a re-render).

The information is local to each copy of your component (unlike the variables outside, which are shared).

Changing a ref does not trigger a re-render, so refs are not appropriate for storing information you want to display on the screen. Use state for that instead. Read more about choosing between `useRef` and `useState`.

Examples of referencing a value with `useRef`. Click counter 2. A stopwatch Example 1 of 2: Click counter This component uses a ref to keep track of how many times the button was clicked. Note that it's okay to use a ref instead of state here because the click count is only read and written in an event handler.

```
App.js
import { useRef } from 'react';

ResetForK import { 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>
  );
}
```

```

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

```

Show more if you show {ref.current} in the JSX, the number won't update on click. This is because setting ref.current does not trigger a re-render. Information that's used for rendering should be state instead. Next Example

Pitfall Do not write or read ref.current during rendering. React expects that the body of your component behaves like a pure function:

If the inputs (props, state, and context) are the same, it should return exactly the same JSX.

Calling it in a different order or with different arguments should not affect the results of other calls.

Reading or writing a ref during rendering breaks these expectations.

```

function MyComponent() { // ... // Don't write
  a ref during rendering
  myRef.current = 123; // ... // Don't read a ref during rendering
  return <h1>{myOtherRef.current}</h1>; }
}

You can read or write refs from event handlers or effects instead.

function MyComponent() { // ... useEffect(() => { // tickmark You can read or write refs in effects
  myRef.current = 123; });
  // ...
  function handleClick() { // tickmark You can read or write refs in event handlers
    doSomething(myOtherRef.current); }
  // ...
}

If you have to read or write something during rendering, use state instead. When you break these rules, your component might still work, but most of the newer features we're adding to React will rely on these expectations. Read more about keeping your components pure.

```

Manipulating the DOM with a ref

It's particularly common to use a ref to manipulate the DOM. React has built-in support for this.

First, declare a ref object with an initial value of null:

```
import { useRef } from 'react';
function MyComponent() { const inputRef = useRef(null); // ... }
```

Then pass your ref object as the ref attribute to the JSX of the DOM node you want to manipulate:

```
// ... return <input ref={inputRef} />;
```

After React creates the DOM node and puts it on the screen, React will set the current property of your ref object to that DOM node. Now you can access the <input>'s DOM node and call methods like focus():

```
function handleClick() { inputRef.current.focus(); }
```

React will set the current property back to null when the node is removed from the screen.

Read more about manipulating the DOM with refs.

Examples of manipulating the DOM with useRef 1. Focusing a text input 2. Scrolling an image into view 3. Playing and pausing a video 4. Exposing a ref to your own component Example 1 of 4: Focusing a text input In this example, clicking the button will focus the input:

```
App.js
import { useRef } from 'react';
function App() {
  const inputRef = useRef(null);

  function handleClick() { inputRef.current.focus(); }

  return (
    <div>
      <input type="text" ref={inputRef}>
      <button onClick={handleClick}>Focus me!</button>
    </div>
  );
}

export default App;
```

```

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](#)[Next Example](#)

Avoiding recreating the ref contents

React saves the initial ref value once and ignores it on the next renders.

```
function Video() { const playerRef = useRef(new VideoPlayer()); // ...
```

Although the result of new VideoPlayer() is only used for the initial render, you're still calling this function on every render. This can be wasteful if it's creating expensive objects.

To solve it, you may initialize the ref like this instead:

```
function Video() { const playerRef = useRef(null); if (playerRef.current === null) { playerRef.current = new VideoPlayer(); } // ...
```

Normally, writing or reading ref.current during render is not allowed. However, it's fine in this case because the result is always the same, and the condition only executes during initialization so it's fully predictable.

Deep Dive How to avoid null checks when initializing useRef later [Show Details](#)
 If you use a type checker and don't want to always check for null, you can try a pattern like this instead:
`function Video() { const playerRef = useRef(null);
 function getPlayer() { if (playerRef.current !== null) { return playerRef.current; } const player = new VideoPlayer(); playerRef.current = player; return player; } // ...`
 Here, the playerRef itself is nullable. However, you should be able to convince your type checker that there is no case in which getPlayer() returns null. Then use getPlayer() in your event handlers.

Troubleshooting

I can't get a ref to a custom component

If you try to pass a ref to your own component like this:

```
const inputRef = useRef(null); return <MyInput ref={inputRef} />;
```

You might get an error in the console:

ConsoleWarning: Function components cannot be given refs. Attempts to access this ref will fail. Did you mean to use React.forwardRef()?

By default, your own components don't expose refs to the DOM nodes inside them.

To fix this, find the component that you want to get a ref to:

```
export default function MyInput({ value, onChange }) { return ( <input value={value} onChange={onChange} /> ); }
```

And then wrap it in forwardRef like this:

```
import { forwardRef } from 'react'; const MyInput = forwardRef(({ value, onChange }, ref) => { return ( <input value={value} onChange={onChange} ref={ref} /> ); }); export default MyInput;
```

Then the parent component can get a ref to it.

Read more about accessing another component's DOM nodes. Previous useReducer Next useState © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Reference useRef(initialValue) Usage Referencing a value with a ref Manipulating the DOM with a ref Avoiding recreating the ref contents Troubleshooting I can't get a ref to a custom component useState – React React v18.3.1 Search ⌂ Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks useActionState - This feature is available in the latest Canary useCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest Canary useReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use - This feature is available in the latest Canary experimental_taintObjectReference - This feature is available in the latest Canary experimental_taintUniqueValue - This feature is available in the latest Canary react-dom@18.3.1 Hooks useFormStatus - This feature is available in the latest Canary Components Common (e.g. <div> <form> - This feature is available in the latest Canary <input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary <meta> - This feature is available in the latest Canary <script> - This feature is available in the latest Canary <style> - This feature is available in the latest Canary <title> - This feature is available in the latest Canary APIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest Canary prefetchDNS - This feature is available in the latest Canary preinit - This feature is available in the latest Canary preinitModule - This feature is available in the latest Canary preload - This feature is available in the latest Canary preloadModule - This feature is available in the latest Canary render unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server Components Server Components - This feature is available in the latest Canary Server Actions - This feature is available in the latest Canary Directives - This feature is available in the latest Canary 'use client' - This feature is available in the latest Canary 'use server' - This feature is available in the latest Canary Legacy APIs Legacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful? API Reference Hooks useState useState is a React Hook that lets you add a state variable to your component. const [state, setState] = useState(initialState)

Reference useState(initialState) set functions, like setSomething(nextState) Usage Adding state to a component Updating state based on the previous state Updating objects and arrays in state Avoiding recreating the initial state Resetting state with a key Storing information from previous renders Troubleshooting I've updated the state, but logging gives me the old value I've updated the state, but the screen doesn't update I'm getting an error: "Too many re-renders" My initializer or updater function runs twice I'm trying to set state to a function, but it gets called instead

Reference

useState(initialState)

Call useState at the top level of your component to declare a state variable.

```
import { useState } from 'react';function MyComponent() { const [age, setAge] = useState(28); const [name, setName] = useState('Taylor'); const [todos, setTodos] = useState(() => createTodos()); // ...}
```

The convention is to name state variables like [something, setSomething] using array destructuring.

See more examples below.

Parameters

initialState: The value you want the state to be initially. It can be a value of any type, but there is a special behavior for functions. This argument is ignored after the initial render.

If you pass a function as initialState, it will be treated as an initializer function. It should be pure, should take no arguments, and should return a value of any type. React will call your initializer function when initializing the component, and store its return value as the initial state. See an example below.

Returns

useState returns an array with exactly two values:

The current state. During the first render, it will match the initialState you have passed.

The set function that lets you update the state to a different value and trigger a re-render.

Caveats

useState is a Hook, so you can only call it at the top level of your component or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

In Strict Mode, React will call your initializer function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your initializer function is pure (as it should be), this should not affect the behavior. The result from one of the calls will be ignored.

set functions, like setSomething(nextState)

The set function returned by useState lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

```
const [name, setName] = useState('Edward');function handleClick() { setName('Taylor'); setAge(a => a + 1); // ...
```

Parameters

`nextState`: The value that you want the state to be. It can be a value of any type, but there is a special behavior for functions.

If you pass a function as `nextState`, it will be treated as an updater function. It must be pure, should take the pending state as its only argument, and should return the next state. React will put your updater function in a queue and re-render your component. During the next render, React will calculate the next state by applying all of the queued updaters to the previous state. See an example below.

Returns

`set` functions do not have a return value.

Caveats

The `set` function only updates the state variable for the next render. If you read the state variable after calling the `set` function, you will still get the old value that was on the screen before your call.

If the new value you provide is identical to the current state, as determined by an `Object.is` comparison, React will skip re-rendering the component and its children. This is an optimization. Although in some cases React may still need to call your component before skipping the children, it shouldn't affect your code.

React batches state updates. It updates the screen after all the event handlers have run and have called their `set` functions. This prevents multiple re-renders during a single event. In the rare case that you need to force React to update the screen earlier, for example to access the DOM, you can use `flushSync`.

Calling the `set` function during rendering is only allowed from within the currently rendering component. React will discard its output and immediately attempt to render it again with the new state. This pattern is rarely needed, but you can use it to store information from the previous renders. See an example below.

In Strict Mode, React will call your updater function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your updater function is pure (as it should be), this should not affect the behavior. The result from one of the calls will be ignored.

Usage

Adding state to a component

Call useState at the top level of your component to declare one or more state variables.

```
import { useState } from 'react';function MyComponent() { const [age, setAge] = useState(42); const [name, setName] = useState('Taylor'); // ...}
```

The convention is to name state variables like [something, setSomething] using array destructuring.

useState returns an array with exactly two items:

The current state of this state variable, initially set to the initial state you provided.

The set function that lets you change it to any other value in response to interaction.

To update what's on the screen, call the set function with some next state:

```
function handleClick() { setName('Robin');
```

React will store the next state, render your component again with the new values, and update the UI.

PitfallCalling the set function does not change the current state in the already executing code:
function handleClick() {
 setName('Robin'); console.log(name); // Still "Taylor"!}
It only affects what useState will return starting from the next render.

Basic useState examples 1. Counter (number) 2. Text field (string) 3. Checkbox (boolean) 4. Form (two variables)

Example 1 of 4: Counter (number)
In this example, the count state variable holds a number. Clicking the button increments it.

```
App.js
import { useState } from 'react';

export default function App() {
  const [count, setCount] = useState(0);
```

```
  function handleClick() {
    setCount(count + 1);
  }
```

```
  return (
    <button onClick={handleClick}>
      You pressed me {count} times
    </button>
  );
}
```

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

Next Example

Updating state based on the previous state

Suppose the age is 42. This handler calls `setAge(age + 1)` three times:

```
function handleClick() { setAge(age + 1); // setAge(42 + 1) setAge(age + 1); // setAge(42 + 1) setAge(age + 1); // setAge(42 + 1)}
```

However, after one click, age will only be 43 rather than 45! This is because calling the `set` function does not update the age state variable in the already running code. So each `setAge(age + 1)` call becomes `setAge(43)`.

To solve this problem, you may pass an updater function to `setAge` instead of the next state:

```
function handleClick() { setAge(a => a + 1); // setAge(42 => 43) setAge(a => a + 1); // setAge(43 => 44) setAge(a => a + 1); // setAge(44 => 45)}
```

Here, `a => a + 1` is your updater function. It takes the pending state and calculates the next state from it.

React puts your updater functions in a queue. Then, during the next render, it will call them in the same order:

`a => a + 1` will receive 42 as the pending state and return 43 as the next state.

`a => a + 1` will receive 43 as the pending state and return 44 as the next state.

`a => a + 1` will receive 44 as the pending state and return 45 as the next state.

There are no other queued updates, so React will store 45 as the current state in the end.

By convention, it's common to name the pending state argument for the first letter of the state variable name, like `a` for `age`. However, you may also call it like `prevAge` or something else that you find clearer.

React may call your updaters twice in development to verify that they are pure.

Deep Dive! Using an updater always preferred? Show Details You might hear a recommendation to always write code like `setAge(a => a + 1)` if the state you're setting is calculated from the previous state. There is no harm in it, but it is also not always necessary. In most cases, there is no difference between these two approaches. React always makes sure that for intentional user actions, like clicks, the age state variable would be updated before the next click. This means there is no risk of a click handler seeing a "stale" age at the beginning of the event handler. However, if you do multiple updates within the same event, updaters can be helpful. They're also helpful if accessing the state variable itself is inconvenient (you might run into this when optimizing re-renders). If you prefer consistency over slightly more verbose syntax, it's reasonable to always write an updater if the state you're setting is calculated from the previous state. If it's calculated from the previous state of some other state variable, you might want to combine them into one object and use a reducer.

The difference between passing an updater and passing the next state directly 1. Passing the updater function 2. Passing the next state directly Example 1 of 2: Passing the updater function This example passes the updater function, so the "+3" button works.

```
App.js
import { useState } from 'react';

function App() {
  const [age, setAge] = useState(42);

  const handleClick = () => {
    setAge(age + 1);
  };

  return (
    <div>
      <p>Age: {age}</p>
      <button onClick={handleClick}>+1</button>
      <button onClick={handleClick}>+2</button>
      <button onClick={handleClick}>+3</button>
    </div>
  );
}

export default App;
```

```
export default function Counter() {
  const [age, setAge] = useState(42);

  function increment() {
    setAge(a => a + 1);
  }

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

[Show more](#)[Next Example](#)

Updating objects and arrays in state

You can put objects and arrays into state. In React, state is considered read-only, so you should replace it rather than mutate your existing objects. For example, if you have a form object in state, don't mutate it:

```
// Don't mutate an object in state like this:form.firstName = 'Taylor';
```

Instead, replace the whole object by creating a new one:

```
// ✅ Replace state with a new object setForm({ ...form, firstName: 'Taylor'});
```

[Read updating objects in state and updating arrays in state to learn more.](#)

Examples of objects and arrays in state 1. Form (object) 2. Form (nested object) 3. List (array) 4. Writing concise update logic with Immer Example 1 of 4: Form (object) In this example, the form state variable holds an object. Each input has a change handler that calls setForm with the next state of the entire form. The { ...form } spread syntax ensures that the state object is replaced rather than mutated.

App.js

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

function App() {
  const [form, setForm] = useState({
    name: '',
    email: ''
  });

  const handleChange = e => {
    const { name, value } = e.target;
    setForm({ ...form, [name]: value });
  };

  return (
    <div>
      <input type="text" name="name" value={form.name} onChange={handleChange}>
      <input type="text" name="email" value={form.email} onChange={handleChange}>
    </div>
  );
}

export default App;
```

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

  return (
    <>
    <label>
      First name:
      <input
        value={form.firstName}
        onChange={e => {
          setForm({
            ...form,
            firstName: e.target.value
          });
        }}
      />
    </label>
    <label>
      Last name:
      <input
        value={form.lastName}
        onChange={e => {
          setForm({
            ...form,
            lastName: e.target.value
          });
        }}
      />
    </label>
  );
}
```

```
<label>
  Email:
  <input
    value={form.email}
    onChange={e => {
      setForm({
        ...form,
        email: e.target.value
      });
    }}
  />
</label>
<p>
  {form.firstName}' '}
  {form.lastName}' '}
  ({form.email})
</p>
</>
);
}
```

Show moreNext Example

Avoiding recreating the initial state

React saves the initial state once and ignores it on the next renders.

```
function TodoList() { const [todos, setTodos] = useState(createInitialTodos()); // ...
```

Although the result of `createInitialTodos()` is only used for the initial render, you're still calling this function on every render. This can be wasteful if it's creating large arrays or performing expensive calculations.

To solve this, you may pass it as an initializer function to `useState` instead:

```
function TodoList() { const [todos, setTodos] = useState(createInitialTodos); // ...
```

Notice that you're passing `createInitialTodos`, which is the function itself, and not `createInitialTodos()`, which is the result of calling it. If you pass a function to `useState`, React will only call it during initialization.

React may call your initializers twice in development to verify that they are pure.

The difference between passing an initializer and passing the initial state directly1. Passing the initializer function 2. Passing the initial state directly Example 1 of 2: Passing the initializer function This example passes the initializer

function, so the createInitialTodos function only runs during initialization. It does not run when component re-renders, such as when you type into the input.

```
App.js
```

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

```
function createInitialTodos() {
  const initialTodos = [];
  for (let i = 0; i < 50; i++) {
    initialTodos.push({
      id: i,
      text: 'Item ' + (i + 1)
    });
  }
  return initialTodos;
}
```

```
export default function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos);
  const [text, setText] = useState('');
}
```

```
return (
  <>
  <input
    value={text}
    onChange={e => setText(e.target.value)}
  />
  <button onClick={() => {
    setText('');
    setTodos([
      {
        id: todos.length,
        text: text
      }, ...todos
    ]);
  }}>Add</button>
  <ul>
    {todos.map(item => (
      <li key={item.id}>
```

```
{item.text}  
      </li>  
    ))}  
  </ul>  
  </>  
);  
}
```

Show moreNext Example

Resetting state with a key

You'll often encounter the key attribute when rendering lists. However, it also serves another purpose.

You can reset a component's state by passing a different key to a component. In this example, the Reset button changes the version state variable, which we pass as a key to the Form. When the key changes, React re-creates the Form component (and all of its children) from scratch, so its state gets reset.

Read preserving and resetting state to learn more.

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

```
export default function App() {  
  const [version, setVersion] = useState(0);  
  
  function handleReset() {  
    setVersion(version + 1);  
  }  
  
  return (  
    <>  
      <button onClick={handleReset}>Reset</button>  
      <Form key={version} />  
    </>  
  );  
}
```

```
function Form() {  
  const [name, setName] = useState('Taylor');
```

```
return (
  <>
  <input
    value={name}
    onChange={e => setName(e.target.value)}
  />
  <p>Hello, {name}.</p>
</>
);
}
```

Show more

Storing information from previous renders

Usually, you will update state in event handlers. However, in rare cases you might want to adjust state in response to rendering — for example, you might want to change a state variable when a prop changes.

In most cases, you don't need this:

If the value you need can be computed entirely from the current props or other state, remove that redundant state altogether. If you're worried about recomputing too often, the `useMemo` Hook can help.

If you want to reset the entire component tree's state, pass a different key to your component.

If you can, update all the relevant state in the event handlers.

In the rare case that none of these apply, there is a pattern you can use to update state based on the values that have been rendered so far, by calling a `set` function while your component is rendering.

Here's an example. This `CountLabel` component displays the `count` prop passed to it:

```
export default function CountLabel({ count }) { return <h1>{count}</h1>}
```

Say you want to show whether the counter has increased or decreased since the last change. The `count` prop doesn't tell you this — you need to keep track of its previous value. Add the `prevCount` state variable to track it. Add another state variable called `trend` to hold whether the `count` has increased or decreased. Compare `prevCount` with `count`, and if they're not equal, update both `prevCount` and `trend`. Now you can show both the current `count` prop and how it has changed since the last render.

```
App.jsCountLabel.jsCountLabel.js ResetForkimport { useState } from 'react';
```

```
export default function CountLabel({ count }) {
  const [prevCount, setPrevCount] = useState(count);
```

```
const [trend, setTrend] = useState(null);

if (prevCount !== count) {
  setPrevCount(count);
  setTrend(count > prevCount ? 'increasing' : 'decreasing');
}

return (
  <>
  <h1>{count}</h1>
  {trend && <p>The count is {trend}</p>}
</>
);
}
```

Show more

Note that if you call a set function while rendering, it must be inside a condition like `prevCount !== count`, and there must be a call like `setPrevCount(count)` inside of the condition. Otherwise, your component would re-render in a loop until it crashes. Also, you can only update the state of the currently rendering component like this. Calling the set function of another component during rendering is an error. Finally, your set call should still update state without mutation — this doesn't mean you can break other rules of pure functions.

This pattern can be hard to understand and is usually best avoided. However, it's better than updating state in an effect. When you call the set function during render, React will re-render that component immediately after your component exits with a return statement, and before rendering the children. This way, children don't need to render twice. The rest of your component function will still execute (and the result will be thrown away). If your condition is below all the Hook calls, you may add an early return; to restart rendering earlier.

Troubleshooting

I've updated the state, but logging gives me the old value

Calling the set function does not change state in the running code:

```
function handleClick() { console.log(count); // 0 setCount(count + 1); // Request a re-render with 1
  console.log(count); // Still 0! setTimeout(() => { console.log(count); // Also 0! }, 5000);}
```

This is because states behaves like a snapshot. Updating state requests another render with the new state value, but does not affect the count JavaScript variable in your already-running event handler.

If you need to use the next state, you can save it in a variable before passing it to the set function:

```
const nextCount = count + 1; setCount(nextCount); console.log(count); // 0 console.log(nextCount); // 1
```

I've updated the state, but the screen doesn't update

React will ignore your update if the next state is equal to the previous state, as determined by an `Object.is` comparison. This usually happens when you change an object or an array in state directly:

```
obj.x = 10; // Wrong: mutating existing objectsetObj(obj); // Doesn't do anything
```

You mutated an existing obj object and passed it back to setObj, so React ignored the update. To fix this, you need to ensure that you're always replacing objects and arrays in state instead of mutating them:

```
// tickmark Correct: creating a new objectsetObj({ ...obj, x: 10});
```

I'm getting an error: "Too many re-renders"

You might get an error that says: Too many re-renders. React limits the number of renders to prevent an infinite loop. Typically, this means that you're unconditionally setting state during render, so your component enters a loop: render, set state (which causes a render), render, set state (which causes a render), and so on. Very often, this is caused by a mistake in specifying an event handler:

```
// Wrong: calls the handler during renderreturn <button onClick={handleClick()}>Click me</button>// tickmark  
Correct: passes down the event handlerreturn <button onClick={handleClick}>Click me</button>// tickmark Correct:  
passes down an inline functionreturn <button onClick={(e) => handleClick(e)}>Click me</button>
```

If you can't find the cause of this error, click on the arrow next to the error in the console and look through the JavaScript stack to find the specific set function call responsible for the error.

My initializer or updater function runs twice

In Strict Mode, React will call some of your functions twice instead of once:

```
function TodoList() { // This component function will run twice for every render. const [todos, setTodos] =  
useState(() => { // This initializer function will run twice during initialization. return createTodos(); }); function  
handleClick() { setTodos(prevTodos => { // This updater function will run twice for every click. return  
[...prevTodos, createTodo()]; }); } // ...
```

This is expected and shouldn't break your code.

This development-only behavior helps you keep components pure. React uses the result of one of the calls, and ignores the result of the other call. As long as your component, initializer, and updater functions are pure, this shouldn't affect your logic. However, if they are accidentally impure, this helps you notice the mistakes.

For example, this impure updater function mutates an array in state:

```
setTodos(prevTodos => { // Mistake: mutating state prevTodos.push(createTodo());});
```

Because React calls your updater function twice, you'll see the todo was added twice, so you'll know that there is a mistake. In this example, you can fix the mistake by replacing the array instead of mutating it:

```
setTodos(prevTodos => { // tickmark Correct: replacing with new state return [...prevTodos, createTodo()];});
```

Now that this updater function is pure, calling it an extra time doesn't make a difference in behavior. This is why React calling it twice helps you find mistakes. Only component, initializer, and updater functions need to be pure. Event handlers don't need to be pure, so React will never call your event handlers twice.

Read [keeping components pure](#) to learn more.

I'm trying to set state to a function, but it gets called instead

You can't put a function into state like this:

```
const [fn, setFn] = useState(someFunction);function handleClick() { setFn(someOtherFunction);}
```

Because you're passing a function, React assumes that someFunction is an initializer function, and that someOtherFunction is an updater function, so it tries to call them and store the result. To actually store a function, you have to put () => before them in both cases. Then React will store the functions you pass.

```
const [fn, setFn] = useState(() => someFunction);function handleClick() { setFn(() =>
someOtherFunction);}PrevioususeRefNextuseSyncExternalStore©2024no uwu plzuwu?Logo
by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape
HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs
ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewReference
useState(initialState) set functions, like setSomething(nextState) Usage Adding state to a component Updating state
based on the previous state Updating objects and arrays in state Avoiding recreating the initial state Resetting state
with a key Storing information from previous renders Troubleshooting I've updated the state, but logging gives me
the old value I've updated the state, but the screen doesn't update I'm getting an error: "Too many re-renders" My
initializer or updater function runs twice I'm trying to set state to a function, but it gets called instead
useSyncExternalStore – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview
Hooks useActionState - This feature is available in the latest CanaryuseCallback useContext useDebugValue
useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic
- This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition
Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the
latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest
Canaryexperimental_taintObjectReference - This feature is available in the latest
Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks
useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div>) <form> - This feature
is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in
the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest
Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs
createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS
- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule -
This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule -
This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot
Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup
renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls
Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in
the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in
the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the
latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory
createRef isValidElement PureComponent Is this page useful?API
ReferenceHooksuseSyncExternalStoreuseSyncExternalStore is a React Hook that lets you subscribe to an external
store.const snapshot = useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?)
```

Reference useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?) Usage Subscribing to an external store
Subscribing to a browser API Extracting the logic to a custom Hook Adding support for server rendering
Troubleshooting I'm getting an error: "The result of getSnapshot should be cached" My subscribe function gets
called after every re-render

Reference

```
useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?)
```

Call useSyncExternalStore at the top level of your component to read a value from an external data store.

```
import { useSyncExternalStore } from 'react';import { todosStore } from './todoStore.js';function TodosApp() { const
todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot); // ...}
```

It returns the snapshot of the data in the store. You need to pass two functions as arguments:

The subscribe function should subscribe to the store and return a function that unsubscribes.

The getSnapshot function should read a snapshot of the data from the store.

See more examples below.

Parameters

subscribe: A function that takes a single callback argument and subscribes it to the store. When the store changes, it should invoke the provided callback. This will cause the component to re-render. The subscribe function should return a function that cleans up the subscription.

getSnapshot: A function that returns a snapshot of the data in the store that's needed by the component. While the store has not changed, repeated calls to getSnapshot must return the same value. If the store changes and the returned value is different (as compared by Object.is), React re-renders the component.

optional getServerSnapshot: A function that returns the initial snapshot of the data in the store. It will be used only during server rendering and during hydration of server-rendered content on the client. The server snapshot must be the same between the client and the server, and is usually serialized and passed from the server to the client. If you omit this argument, rendering the component on the server will throw an error.

Returns

The current snapshot of the store which you can use in your rendering logic.

Caveats

The store snapshot returned by getSnapshot must be immutable. If the underlying store has mutable data, return a new immutable snapshot if the data has changed. Otherwise, return a cached last snapshot.

If a different subscribe function is passed during a re-render, React will re-subscribe to the store using the newly passed subscribe function. You can prevent this by declaring subscribe outside the component.

If the store is mutated during a non-blocking Transition update, React will fall back to performing that update as blocking. Specifically, for every Transition update, React will call `getSnapshot` a second time just before applying changes to the DOM. If it returns a different value than when it was called originally, React will restart the update from scratch, this time applying it as a blocking update, to ensure that every component on screen is reflecting the same version of the store.

It's not recommended to suspend a render based on a store value returned by `useSyncExternalStore`. The reason is that mutations to the external store cannot be marked as non-blocking Transition updates, so they will trigger the nearest Suspense fallback, replacing already-rendered content on screen with a loading spinner, which typically makes a poor UX.

For example, the following are discouraged:

```
const LazyProductDetailPage = lazy(() => import('./ProductDetailPage.js'));
```

```
function ShoppingApp() { const selectedProductId = useSyncExternalStore(...); // Calling `use` with a Promise dependent on `selectedProductId`
```

```
const data = use(fetchItem(selectedProductId)) // Conditionally rendering a lazy component based on
```

```
`selectedProductId` return selectedProductId != null ? <LazyProductDetailPage /> : <FeaturedProducts />;}
```

Usage

Subscribing to an external store

Most of your React components will only read data from their props, state, and context. However, sometimes a component needs to read some data from some store outside of React that changes over time. This includes:

Third-party state management libraries that hold state outside of React.

Browser APIs that expose a mutable value and events to subscribe to its changes.

Call `useSyncExternalStore` at the top level of your component to read a value from an external data store.

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

```
import { todosStore } from './todoStore.js';
```

```
function TodosApp() { const todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot); // ...}
```

It returns the snapshot of the data in the store. You need to pass two functions as arguments:

The `subscribe` function should subscribe to the store and return a function that unsubscribes.

The `getSnapshot` function should read a snapshot of the data from the store.

React will use these functions to keep your component subscribed to the store and re-render it on changes.

For example, in the sandbox below, `todosStore` is implemented as an external store that stores data outside of React. The `TodosApp` component connects to that external store with the `useSyncExternalStore` Hook.

```
App.js
import { ResetFork } from 'react';
import { todosStore } from './todoStore.js';

export default function TodosApp() {
  const todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot);

  return (
    <>
      <button onClick={() => todosStore.addTodo()}>Add todo</button>
      <hr />
      <ul>
        {todos.map(todo => (
          <li key={todo.id}>{todo.text}</li>
        )));
      </ul>
    </>
  );
}

}
```

Show more

NoteWhen possible, we recommend using built-in React state with useState and useReducer instead. The useSyncExternalStore API is mostly useful if you need to integrate with existing non-React code.

Subscribing to a browser API

Another reason to add useSyncExternalStore is when you want to subscribe to some value exposed by the browser that changes over time. For example, suppose that you want your component to display whether the network connection is active. The browser exposes this information via a property called navigator.onLine.

This value can change without React's knowledge, so you should read it with useSyncExternalStore.

```
import { useSyncExternalStore } from 'react';
function ChatIndicator() {
  const isOnline =
    useSyncExternalStore(subscribe, getSnapshot); // ...
}
```

To implement the getSnapshot function, read the current value from the browser API:

```
function getSnapshot() {
  return navigator.onLine;
}
```

Next, you need to implement the subscribe function. For example, when navigator.onLine changes, the browser fires the online and offline events on the window object. You need to subscribe the callback argument to the corresponding events, and then return a function that cleans up the subscriptions:

```
function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);
  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
}
```

Now React knows how to read the value from the external navigator.onLine API and how to subscribe to its changes. Disconnect your device from the network and notice that the component re-renders in response:

```
App.jsApp.js ResetForkimport { useSyncExternalStore } from 'react';
```

```
export default function ChatIndicator() {
  const isOnline = useSyncExternalStore(subscribe, getSnapshot);
  return <h1>{isOnline ? 'tickmark Online' : ' Disconnected'}</h1>;
}
```

```
function getSnapshot() {
  return navigator.onLine;
}
```

```
function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);
  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
}
```

Show more

Extracting the logic to a custom Hook

Usually you won't write useSyncExternalStore directly in your components. Instead, you'll typically call it from your own custom Hook. This lets you use the same external store from different components.

For example, this custom useOnlineStatus Hook tracks whether the network is online:

```
import { useSyncExternalStore } from 'react';
export function useOnlineStatus() {
  const isOnline = useSyncExternalStore(subscribe, getSnapshot);
  return isOnline;
}

function getSnapshot() { // ... }

function subscribe(callback) { // ... }
```

Now different components can call useOnlineStatus without repeating the underlying implementation:

```
App.jsuseOnlineStatus.jsApp.js ResetForkimport { useOnlineStatus } from './useOnlineStatus.js';
```

```
function StatusBar() {
  const isOnline = useOnlineStatus();
```

```
return <h1>{isOnline ? 'tickmark Online' : 'Disconnected'}</h1>;
}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log('tickmark Progress saved');
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? 'Save progress' : 'Reconnecting...'}
    </button>
  );
}

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

Show more

Adding support for server rendering

If your React app uses server rendering, your React components will also run outside the browser environment to generate the initial HTML. This creates a few challenges when connecting to an external store:

If you're connecting to a browser-only API, it won't work because it does not exist on the server.

If you're connecting to a third-party data store, you'll need its data to match between the server and client.

To solve these issues, pass a `getServerSnapshot` function as the third argument to `useSyncExternalStore`:

```
import { useSyncExternalStore } from 'react';
export function useOnlineStatus() {
  const isOnline = useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot);
  return isOnline;
}

function getSnapshot() {
  return navigator.onLine;
}

function getServerSnapshot() {
  return true; // Always show "Online" for server-generated HTML
}

function subscribe(callback) {
  // ...
}
```

The `getServerSnapshot` function is similar to `getSnapshot`, but it runs only in two situations:

It runs on the server when generating the HTML.

It runs on the client during hydration, i.e. when React takes the server HTML and makes it interactive.

This lets you provide the initial snapshot value which will be used before the app becomes interactive. If there is no meaningful initial value for the server rendering, omit this argument to force rendering on the client.

Note Make sure that `getServerSnapshot` returns the same exact data on the initial client render as it returned on the server. For example, if `getServerSnapshot` returned some prepopulated store content on the server, you need to transfer this content to the client. One way to do this is to emit a `<script>` tag during server rendering that sets a global like `window.MY_STORE_DATA`, and read from that global on the client in `getServerSnapshot`. Your external store should provide instructions on how to do that.

Troubleshooting

I'm getting an error: "The result of `getSnapshot` should be cached"

This error means your `getSnapshot` function returns a new object every time it's called, for example:

```
function getSnapshot() {
  // Do not return always different objects from getSnapshot
  return {
    todos: myStore.todos
  };
}
```

React will re-render the component if `getSnapshot` return value is different from the last time. This is why, if you always return a different value, you will enter an infinite loop and get this error.

Your `getSnapshot` object should only return a different object if something has actually changed. If your store contains immutable data, you can return that data directly:

```
function getSnapshot() {
  // tickmark You can return immutable data
  return myStore.todos;
}
```

If your store data is mutable, your `getSnapshot` function should return an immutable snapshot of it. This means it does need to create new objects, but it shouldn't do this for every single call. Instead, it should store the last calculated snapshot, and return the same snapshot as the last time if the data in the store has not changed. How you determine whether mutable data has changed depends on your mutable store.

My `subscribe` function gets called after every re-render

This `subscribe` function is defined inside a component so it is different on every re-render:

```
function ChatIndicator() {
  const isOnline = useSyncExternalStore(subscribe, getSnapshot);
  // Always a different function, so React will resubscribe on every re-render
  function subscribe() {
    // ...
  }
  // ...
}
```

React will resubscribe to your store if you pass a different `subscribe` function between re-renders. If this causes performance issues and you'd like to avoid resubscribing, move the `subscribe` function outside:

```
function ChatIndicator() { const isOnline = useSyncExternalStore(subscribe, getSnapshot); // ...}// tickmark Always  
the same function, so React won't need to resubscribe
```

Alternatively, wrap subscribe into useCallback to only resubscribe when some argument changes:

```
function ChatIndicator({ userId }) { const isOnline = useSyncExternalStore(subscribe, getSnapshot); // tickmark  
Same function as long as userId doesn't change const subscribe = useCallback(() => { // ... }, [userId]); //  
...}PrevioususeStateNextuseTransition©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick  
StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact  
DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact  
NativePrivacyTermsOn this pageOverviewReference useSyncExternalStore(subscribe, getSnapshot,  
getServerSnapshot?) Usage Subscribing to an external store Subscribing to a browser API Extracting the logic to a  
custom Hook Adding support for server rendering Troubleshooting I'm getting an error: "The result of getSnapshot  
should be cached" My subscribe function gets called after every re-render Not Found –  
ReactReactv18.3.1Search⌘CtrlKLearnReferenceCommunityBlogGET STARTEDQuick Start Tutorial: Tic-Tac-Toe  
Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using  
TypeScript React Developer Tools React Compiler - This feature is available in the latest CanaryLEARN  
REACTDescribing the UI Your First Component Importing and Exporting Components Writing Markup with JSX  
JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping  
Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render  
and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in  
State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components  
Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up  
with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs  
Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects  
Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful?Learn ReactNot FoundThis page  
doesn't exist.If this is a mistake, let us know, and we will try to fix it!©2024no uwu plzuwu?Logo  
by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape  
HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs  
ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsBuilt-in React Components –  
ReactReactv18.3.1Search⌘CtrlKLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useState - This  
feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect usedId  
useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the  
latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>)  
<Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext  
forwardRef lazy memo startTransition use - This feature is available in the latest  
Canaryexperimental_taintObjectReference - This feature is available in the latest  
Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks  
useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature  
is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in  
the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest  
Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs  
createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS  
- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule -  
This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule -  
This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot  
Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup  
renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls  
Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in  
the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in  
the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the  
latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory  
createRef isValidElement PureComponent Is this page useful?API ReferenceBuilt-in React ComponentsReact exposes  
a few built-in components that you can use in your JSX.
```

Built-in components

<Fragment>, alternatively written as <>...</>, lets you group multiple JSX nodes together.

<Profiler> lets you measure rendering performance of a React tree programmatically.

<Suspense> lets you display a fallback while the child components are loading.

<StrictMode> enables extra development-only checks that help you find bugs early.

Your own components

You can also define your own components as JavaScript functions. Previous useTransition Next <Fragment> (<>) © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Built-in components Your own components Built-in React APIs – React React v18.3.1 Search ⌂ Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks useState – This feature is available in the latest Canary useState useCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic – This feature is available in the latest Canary useReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache – This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use – This feature is available in the latest Canary experimental_taintObjectReference – This feature is available in the latest Canary experimental_taintUniqueValue – This feature is available in the latest Canary react-dom@18.3.1 Hooks useFormStatus – This feature is available in the latest Canary Components Common (e.g. <div> <form> – This feature is available in the latest Canary <input> <option> <progress> <select> <textarea> <link> – This feature is available in the latest Canary <meta> – This feature is available in the latest Canary <script> – This feature is available in the latest Canary <style> – This feature is available in the latest Canary <title> – This feature is available in the latest Canary APIs createPortal flushSync findDOMNode hydrate preconnect – This feature is available in the latest Canary prefetchDNS – This feature is available in the latest Canary preinit – This feature is available in the latest Canary preinitModule – This feature is available in the latest Canary preload – This feature is available in the latest Canary preloadModule – This feature is available in the latest Canary render unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server Components Server Components – This feature is available in the latest Canary Server Actions – This feature is available in the latest Canary Directives – This feature is available in the latest Canary 'use client' – This feature is available in the latest Canary 'use server' – This feature is available in the latest Canary Legacy APIs Legacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful? API Reference Built-in React APIs In addition to Hooks and Components, the react package exports a few other APIs that are useful for defining components. This page lists all the remaining modern React APIs.

createContext lets you define and provide context to the child components. Used with useContext.

forwardRef lets your component expose a DOM node as a ref to the parent. Used with useRef.

`lazy` lets you defer loading a component's code until it's rendered for the first time.

`memo` lets your component skip re-renders with same props. Used with `useMemo` and `useCallback`.

`startTransition` lets you mark a state update as non-urgent. Similar to `useTransition`.

`act` lets you wrap renders and interactions in tests to ensure updates have processed before making assertions.

Resource APIs

Resources can be accessed by a component without having them as part of their state. For example, a component can read a message from a Promise or read styling information from a context.

To read a value from a resource, use this API:

`use` lets you read the value of a resource like a Promise or context.

```
function MessageComponent({ messagePromise }) { const message = use(messagePromise); const theme = use(ThemeContext); // ...}Previous<Suspense>Nextact©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewResource APIs forwardRef - ReactReactv18.3.1Search⌘FCtrlKLearReferenceCommunityBlogreact@18.3.1Overview Hooks useActionState - This feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useState useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div>) <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceAPIsforwardRefforwardRef lets your component expose a DOM node to parent component with a ref.const SomeComponent = forwardRef(render) Reference forwardRef(render) render function Usage Exposing a DOM node to the parent component Forwarding a ref through multiple components Exposing an imperative handle instead of a DOM node Troubleshooting My component is wrapped in forwardRef, but the ref to it is always null
```

Reference

forwardRef(render)

Call `forwardRef()` to let your component receive a ref and forward it to a child component:

```
import { forwardRef } from 'react';const MyInput = forwardRef(function MyInput(props, ref) { // ...});
```

See more examples below.

Parameters

`render`: The render function for your component. React calls this function with the props and ref that your component received from its parent. The JSX you return will be the output of your component.

Returns

`forwardRef` returns a React component that you can render in JSX. Unlike React components defined as plain functions, a component returned by `forwardRef` is also able to receive a `ref` prop.

Caveats

In Strict Mode, React will call your render function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your render function is pure (as it should be), this should not affect the logic of your component. The result from one of the calls will be ignored.

render function

`forwardRef` accepts a render function as an argument. React calls this function with props and ref:

```
const MyInput = forwardRef(function MyInput(props, ref) { return ( <label> {props.label} <input ref={ref} /> </label> );});
```

Parameters

`props`: The props passed by the parent component.

`ref`: The ref attribute passed by the parent component. The ref can be an object or a function. If the parent component has not passed a ref, it will be null. You should either pass the ref you receive to another component, or pass it to `useImperativeHandle`.

Returns

`forwardRef` returns a React component that you can render in JSX. Unlike React components defined as plain functions, the component returned by `forwardRef` is able to take a `ref` prop.

Usage

Exposing a DOM node to the parent component

By default, each component's DOM nodes are private. However, sometimes it's useful to expose a DOM node to the parent—for example, to allow focusing it. To opt in, wrap your component definition into `forwardRef()`:

```
import { forwardRef } from 'react';const MyInput = forwardRef(function MyInput(props, ref) { const { label, ...otherProps } = props; return ( <label> {label} <input {...otherProps} /> </label> );});
```

You will receive a `ref` as the second argument after `props`. Pass it to the DOM node that you want to expose:

```
import { forwardRef } from 'react';const MyInput = forwardRef(function MyInput(props, ref) { const { label, ...otherProps } = props; return ( <label> {label} <input {...otherProps} ref={ref} /> </label> );});
```

This lets the parent `Form` component access the `<input>` DOM node exposed by `MyInput`:

```
function Form() { const ref = useRef(null); function handleClick() { ref.current.focus(); } return ( <form> <MyInput label="Enter your name:" ref={ref} /> <button type="button" onClick={handleClick}> Edit </button> </form> );}
```

This `Form` component passes a `ref` to `MyInput`. The `MyInput` component forwards that `ref` to the `<input>` browser tag. As a result, the `Form` component can access that `<input>` DOM node and call `focus()` on it.

Keep in mind that exposing a `ref` to the DOM node inside your component makes it harder to change your component's internals later. You will typically expose DOM nodes from reusable low-level components like buttons or text inputs, but you won't do it for application-level components like an avatar or a comment.

Examples of forwarding a `ref`:
1. Focusing a text input
2. Playing and pausing a video
Example 1 of 2: Focusing a text input
Clicking the button will focus the input. The `Form` component defines a `ref` and passes it to the `MyInput` component. The `MyInput` component forwards that `ref` to the browser `<input>`. This lets the `Form` component focus the `<input>`.

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

```
export default function Form() { const ref = useRef(null); function handleClick() { ref.current.focus(); } return ( <form> <MyInput label="Enter your name:" ref={ref} /> <button type="button" onClick={handleClick}>
```

```
    Edit  
    </button>  
  </form>  
);  
}
```

Show moreNext Example

Forwarding a ref through multiple components

Instead of forwarding a ref to a DOM node, you can forward it to your own component like MyInput:

```
constFormField = forwardRef(functionFormField(props, ref) { // ... return ( <> <MyInput ref={ref} /> ... </> );});
```

If that MyInput component forwards a ref to its <input>, a ref to FormField will give you that <input>:

```
functionForm() { constref = useRef(null); function handleClick() { ref.current.focus(); } return ( <form> <FormField label="Enter your name:" ref={ref}isRequired={true} /> <button type="button" onClick={handleClick}> Edit </button> </form> );}
```

The Form component defines a ref and passes it to FormField. The FormField component forwards that ref to MyInput, which forwards it to a browser <input> DOM node. This is how Form accesses that DOM node.

```
App.js  
FormField.js  
MyInput.js  
App.js  
ResetForm  
import {useRef} from 'react';  
importFormField from './FormField.js';
```

```
export default functionForm() {  
  constref = useRef(null);  
  
  function handleClick() {  
    ref.current.focus();  
  }  
  
  return (  
    <form>  
      <FormField label="Enter your name:" ref={ref}isRequired={true} />  
      <button type="button" onClick={handleClick}>  
        Edit  
      </button>  
    </form>  
  );  
}
```

```
}
```

Show more

Exposing an imperative handle instead of a DOM node

Instead of exposing an entire DOM node, you can expose a custom object, called an imperative handle, with a more constrained set of methods. To do this, you'd need to define a separate ref to hold the DOM node:

```
const MyInput = forwardRef(function MyInput(props, ref) { const inputRef = useRef(null); // ... return <input {...props} ref={inputRef} />});
```

Pass the ref you received to useImperativeHandle and specify the value you want to expose to the ref:

```
import { forwardRef, useRef, useImperativeHandle } from 'react';const MyInput = forwardRef(function MyInput(props, ref) { const inputRef = useRef(null); useImperativeHandle(ref, () => { return { focus() { inputRef.current.focus(); }, scrollIntoView() { inputRef.current.scrollIntoView(); } }; }, []); return <input {...props} ref={inputRef} />});
```

If some component gets a ref to MyInput, it will only receive your { focus, scrollIntoView } object instead of the DOM node. This lets you limit the information you expose about your DOM node to the minimum.

```
App.jsMyInput.jsApp.js ResetForkimport { useRef } from 'react';
```

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

```
export default function Form() {
```

```
  const ref = useRef(null);
```

```
  function handleClick() {
```

```
    ref.current.focus();
```

```
    // This won't work because the DOM node isn't exposed:
```

```
    // ref.current.style.opacity = 0.5;
```

```
}
```

```
  return (
```

```
    <form>
```

```
      <MyInput placeholder="Enter your name" ref={ref} />
```

```
      <button type="button" onClick={handleClick}>
```

```
        Edit
```

```
      </button>
```

```
    </form>
```

```
);
```

}

Show more

Read more about using imperative handles.

Pitfall Do not overuse refs. You should only use refs for imperative behaviors that you can't express as props: for example, scrolling to a node, focusing a node, triggering an animation, selecting text, and so on. If you can express something as a prop, you should not use a ref. For example, instead of exposing an imperative handle like { open, close } from a Modal component, it is better to take isOpen as a prop like <Modal isOpen={isOpen} />. Effects can help you expose imperative behaviors via props.

Troubleshooting

My component is wrapped in forwardRef, but the ref to it is always null

This usually means that you forgot to actually use the ref that you received.

For example, this component doesn't do anything with its ref:

```
const MyInput = forwardRef(function MyInput({ label }, ref) { return ( <label> {label} <input /> </label> );});
```

To fix it, pass the ref down to a DOM node or another component that can accept a ref:

```
const MyInput = forwardRef(function MyInput({ label }, ref) { return ( <label> {label} <input ref={ref} /> </label> );});
```

The ref to MyInput could also be null if some of the logic is conditional:

```
const MyInput = forwardRef(function MyInput({ label, showInput }, ref) { return ( <label> {label} {showInput && <input ref={ref} />} </label> );});
```

If showInput is false, then the ref won't be forwarded to any node, and a ref to MyInput will remain empty. This is particularly easy to miss if the condition is hidden inside another component, like Panel in this example:

```
const MyInput = forwardRef(function MyInput({ label, showInput }, ref) { return ( <label> {label} <Panel isExpanded={showInput}> <input ref={ref} /> </Panel> </label> );});
```

Previous createContext Next lazy ©2024 no
uuw plzuwu?Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Reference forwardRef(render) render function Usage Exposing a DOM node to the parent component Forwarding a ref through multiple components Exposing an imperative handle instead of a DOM node Troubleshooting My component is wrapped in forwardRef, but the ref to it is always null lazy –

React React v18.3.1 Search Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks useState - This feature is available in the latest Canary useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest Canary useReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use - This feature is available in the latest Canary experimental_taintObjectReference - This feature is available in the latest Canary experimental_taintUniqueValue - This feature is available in the latest Canary react-dom@18.3.1 Hooks useFormStatus - This feature is available in the latest Canary Components Common (e.g. <div> <form> - This feature is available in the latest Canary <input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary <meta> - This feature is available in the latest Canary <script> - This feature is available in the latest Canary <style> - This feature is available in the latest Canary <title> - This feature is available in the latest Canary APIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest Canary prefetchDNS

- This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children cloneElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceAPIslazylazy lets you defer loading component's code until it is rendered for the first time.`const SomeComponent = lazy(load)`

Reference [lazy\(load\)](#) load function Usage Lazy-loading components with Suspense Troubleshooting My lazy component's state gets reset unexpectedly

Reference

`lazy(load)`

Call `lazy` outside your components to declare a lazy-loaded React component:

```
import { lazy } from 'react';const MarkdownPreview = lazy(() => import('./MarkdownPreview.js'));
```

See more examples below.

Parameters

`load`: A function that returns a Promise or another thenable (a Promise-like object with a `then` method). React will not call `load` until the first time you attempt to render the returned component. After React first calls `load`, it will wait for it to resolve, and then render the resolved value's `.default` as a React component. Both the returned Promise and the Promise's resolved value will be cached, so React will not call `load` more than once. If the Promise rejects, React will throw the rejection reason for the nearest Error Boundary to handle.

Returns

`lazy` returns a React component you can render in your tree. While the code for the lazy component is still loading, attempting to render it will suspend. Use `<Suspense>` to display a loading indicator while it's loading.

load function

Parameters

`load` receives no parameters.

Returns

You need to return a Promise or some other thenable (a Promise-like object with a `then` method). It needs to eventually resolve to an object whose `.default` property is a valid React component type, such as a function, `memo`, or a `forwardRef` component.

Usage

[Lazy-loading components with Suspense](#)

Usually, you import components with the static import declaration:

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

To defer loading this component's code until it's rendered for the first time, replace this import with:

```
import { lazy } from 'react';const MarkdownPreview = lazy(() => import('./MarkdownPreview.js'));
```

This code relies on dynamic import(), which might require support from your bundler or framework. Using this pattern requires that the lazy component you're importing was exported as the default export.

Now that your component's code loads on demand, you also need to specify what should be displayed while it is loading. You can do this by wrapping the lazy component or any of its parents into a <Suspense> boundary:

```
<Suspense fallback=<Loading />> <h2>Preview</h2> <MarkdownPreview /></Suspense>
```

In this example, the code for MarkdownPreview won't be loaded until you attempt to render it. If MarkdownPreview hasn't loaded yet, Loading will be shown in its place. Try ticking the checkbox:

```
App.jsLoading.jsMarkdownPreview.jsApp.js ResetForkimport { useState, Suspense, lazy } from 'react';
```

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

```
const MarkdownPreview = lazy(() => delayForDemo(import('./MarkdownPreview.js')));
```

```
export default function MarkdownEditor() {
  const [showPreview, setShowPreview] = useState(false);
  const [markdown, setMarkdown] = useState('Hello, **world**!');
  return (
    <>
      <textarea value={markdown} onChange={e => setMarkdown(e.target.value)} />
      <label>
        <input type="checkbox" checked={showPreview} onChange={e => setShowPreview(e.target.checked)} />
        Show preview
      </label>
      <hr />
      {showPreview && (
        <Suspense fallback=<Loading />>
          <h2>Preview</h2>
          <MarkdownPreview markdown={markdown} />
        </Suspense>
      )}
    </>
  );
}
```

```
}

// Add a fixed delay so you can see the loading state

function delayForDemo(promise) {
  return new Promise(resolve => {
    setTimeout(resolve, 2000);
  }).then(() => promise);
}
```

Show more

This demo loads with an artificial delay. The next time you untick and tick the checkbox, Preview will be cached, so there will be no loading state. To see the loading state again, click “Reset” on the sandbox.

Learn more about managing loading states with Suspense.

Troubleshooting

My lazy component's state gets reset unexpectedly

Do not declare lazy components inside other components:

```
import { lazy } from 'react';
function Editor() { // Bad: This will cause all state to be reset on re-renders
  const MarkdownPreview = lazy(() => import('./MarkdownPreview.js'));
  // ...
}
```

Instead, always declare them at the top level of your module:

```
import { lazy } from 'react';
// tickmark Good: Declare lazy components outside of your components
const MarkdownPreview = lazy(() => import('./MarkdownPreview.js'));
function Editor() { // ...
}PreviousforwardRefNextmemo©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuickStartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReactDOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReactNativePrivacyTermsOn this pageOverviewReference lazy(load) load function Usage Lazy-loading components withSuspense Troubleshooting My lazy component's state gets reset unexpectedly Built-in React DOM Hooks –ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useStateAction - This feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContextforwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1HooksuseFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div>) <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIscreatePortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIscreateRoot hydrateRoot
```

Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server Components Server Components - This feature is available in the latest Canary Server Actions - This feature is available in the latest Canary Directives - This feature is available in the latest Canary 'use client' - This feature is available in the latest Canary 'use server' - This feature is available in the latest Canary Legacy APIs Legacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful? API Reference Built-in React DOM Hooks The react-dom package contains Hooks that are only supported for web applications (which run in the browser DOM environment). These Hooks are not supported in non-browser environments like iOS, Android, or Windows applications. If you are looking for Hooks that are supported in web browsers and other environments see the React Hooks page. This page lists all the Hooks in the react-dom package.

Form Hooks

Canary Form Hooks are currently only available in React's canary and experimental channels. Learn more about React's release channels here.

Forms let you create interactive controls for submitting information. To manage forms in your components, use one of these Hooks:

useFormStatus allows you to make updates to the UI based on the status of the a form.

```
function Form({ action }) { async function increment(n) { return n + 1; } const [count, incrementFormAction] = useState(increment, 0); return ( <form action={action}> <button formAction={incrementFormAction}>Count: {count}</button> <Button /> </form> ); } function Button() { const { pending } = useFormStatus(); return ( <button disabled={pending} type="submit"> Submit </button> ); } Next useFormStatus © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Form Hooks useFormStatus – React React v18.3.1 Search Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks useState - This feature is available in the latest Canary useCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest Canary useReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use - This feature is available in the latest Canary experimental_taintObjectReference - This feature is available in the latest Canary experimental_taintUniqueValue - This feature is available in the latest Canary react-dom@18.3.1 Hooks useFormStatus - This feature is available in the latest Canary Components Common (e.g. <div> <form> - This feature is available in the latest Canary <input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary <meta> - This feature is available in the latest Canary <script> - This feature is available in the latest Canary <style> - This feature is available in the latest Canary <title> - This feature is available in the latest Canary APIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest Canary prefetchDNS - This feature is available in the latest Canary preinit - This feature is available in the latest Canary preinitModule - This feature is available in the latest Canary preload - This feature is available in the latest Canary preloadModule - This feature is available in the latest Canary render unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server Components Server Components - This feature is available in the latest Canary Server Actions - This feature is available in the latest Canary Directives - This feature is available in
```

the latest Canary 'use client' - This feature is available in the latest Canary 'use server' - This feature is available in the latest CanaryLegacy APIs Legacy React APIs Children cloneElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful? API Reference Hooks useFormStatus - This feature is available in the latest Canary Canary The useFormStatus Hook is currently only available in React's Canary and experimental channels. Learn more about React's release channels here.

useFormStatus is a Hook that gives you status information of the last form submission.
const { pending, data, method, action } = useFormStatus();

Reference useFormStatus() Usage Display a pending state during form submission Read the form data being submitted Troubleshooting status.pending is never true

Reference

useFormStatus()

The useFormStatus Hook provides status information of the last form submission.

```
import { useFormStatus } from "react-dom"; import action from './actions'; function Submit() { const status = useFormStatus(); return <button disabled={status.pending}>Submit</button>} export default function App() { return ( <form action={action}> <Submit /> </form> );}
```

To get status information, the Submit component must be rendered within a <form>. The Hook returns information like the pending property which tells you if the form is actively submitting.

In the above example, Submit uses this information to disable <button> presses while the form is submitting.

See more examples below.

Parameters

useFormStatus does not take any parameters.

Returns

A status object with the following properties:

pending: A boolean. If true, this means the parent <form> is pending submission. Otherwise, false.

data: An object implementing the FormData interface that contains the data the parent <form> is submitting. If there is no active submission or no parent <form>, it will be null.

method: A string value of either 'get' or 'post'. This represents whether the parent <form> is submitting with either a GET or POST HTTP method. By default, a <form> will use the GET method and can be specified by the method property.

action: A reference to the function passed to the action prop on the parent <form>. If there is no parent <form>, the property is null. If there is a URI value provided to the action prop, or no action prop specified, status.action will be null.

Caveats

The useFormStatus Hook must be called from a component that is rendered inside a <form>.

useFormStatus will only return status information for a parent <form>. It will not return status information for any <form> rendered in that same component or children components.

Usage

Display a pending state during form submission

To display a pending state while a form is submitting, you can call the useFormStatus Hook in a component rendered in a <form> and read the pending property returned.

Here, we use the pending property to indicate the form is submitting.

```
App.jsApp.js ResetForkimport { useFormStatus } from "react-dom";
```

```
import { submitForm } from "./actions.js";
```

```
function Submit() {
  const { pending } = useFormStatus();
  return (
    <button type="submit" disabled={pending}>
      {pending ? "Submitting..." : "Submit"}
    </button>
  );
}
```

```
function Form({ action }) {
  return (
    <form action={action}>
      <Submit />
    </form>
  );
}
```

```
export default function App() {
  return <Form action={submitForm} />;
}
```

Show more

Pitfall useFormStatus will not return status information for a <form> rendered in the same component. The useFormStatus Hook only returns status information for a parent <form> and not for any <form> rendered in the same component calling the Hook, or child components.

```
function Form() { // `pending` will never be true //
  const { pending } = useFormStatus();
  return <form action={submit}></form>;}
Instead call useFormStatus from inside a component that is located inside <form>.function Submit() { // tickmark `pending` will be derived from the form that wraps the Submit component
  const { pending } = useFormStatus();
  return <button disabled={pending}>...</button>;}
function Form() { // This is the <form> `useFormStatus` tracks
  return ( <form action={submit}> <Submit /> </form> );}
```

Read the form data being submitted

You can use the data property of the status information returned from useFormStatus to display what data is being submitted by the user.

Here, we have a form where users can request a username. We can use useFormStatus to display a temporary status message confirming what username they have requested.

```
UsernameForm.jsApp.jsUsernameForm.js ResetForkimport {useState, useMemo, useRef} from 'react';
import {useFormStatus} from 'react-dom';
```

```
export default function UsernameForm() {
  const {pending, data} = useFormStatus();

  return (
    <div>
      <h3>Request a Username: </h3>
      <input type="text" name="username" disabled={pending}/>
      <button type="submit" disabled={pending}>
        Submit
      </button>
      <br />
      <p>{data ? `Requesting ${data?.get("username")}` : ""}</p>
    </div>
  );
}
```

Show more

Troubleshooting

status.pending is never true

useFormStatus will only return status information for a parent <form>.

If the component that calls useFormStatus is not nested in a <form>, status.pending will always return false. Verify useFormStatus is called in a component that is a child of a <form> element.

useFormStatus will not track the status of a <form> rendered in the same component. See Pitfall for more details. Previous Hooks Next Components ©2024 no uwu plzuwu? Logo by@sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Reference useFormStatus() Usage Display a pending state during form submission Read the form data being submitted Troubleshooting status.pending is never true Client React DOM APIs – React React v18.3.1 Search ⌂ Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks useState – This feature is available in the latest Canary useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic – This feature is available in the latest Canary useReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache – This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use – This feature is available in the latest Canary experimental_taintObjectReference – This feature is available in the latest Canary experimental_taintUniqueValue – This feature is available in the latest Canary react-dom@18.3.1 Hooks useFormStatus – This feature is available in the latest Canary Components Common (e.g. <div>) <form> – This feature is available in the latest Canary <input> <option> <progress> <select> <textarea> <link> – This feature is available in the latest Canary <meta> – This feature is available in the latest Canary <script> – This feature is available in the latest Canary <style> – This feature is available in the latest Canary <title> – This feature is available in the latest Canary APIs createPortal flushSync findDOMNode hydrate preconnect – This feature is available in the latest Canary prefetchDNS – This feature is available in the latest Canary preinit – This feature is available in the latest Canary preinitModule – This feature is available in the latest Canary preload – This feature is available in the latest Canary preloadModule – This feature is available in the latest Canary render unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server Components Server Components – This feature is available in the latest Canary Server Actions – This feature is available in the latest Canary Directives – This feature is available in the latest Canary 'use client' – This feature is available in the latest Canary 'use server' – This feature is available in the latest Canary Legacy APIs Legacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful? API Reference Client React DOM APIs The react-dom/client APIs let you render React components on the client (in the browser). These APIs are typically used at the top level of your app to initialize your React tree. A framework may call them for you. Most of your components don't need to import or use them.

Client APIs

createRoot lets you create a root to display React components inside a browser DOM node.

hydrateRoot lets you display React components inside a browser DOM node whose HTML content was previously generated by react-dom/server.

Browser support

React supports all popular browsers, including Internet Explorer 9 and above. Some polyfills are required for older browsers such as IE 9 and IE 10. Previous unmountComponentAtNode Next createRoot ©2024 no uwu plzuwu? Logo by@sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Client APIs Browser support createRoot – React React v18.3.1 Search Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks useActionState - This feature is available in the latest Canary useCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest Canary useReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use - This feature is available in the latest Canary experimental_taintObjectReference - This feature is available in the latest Canary experimental_taintUniqueValue - This feature is available in the latest Canary react-dom@18.3.1 Hooks useFormStatus - This feature is available in the latest Canary Components Common (e.g. <div> <form> - This feature is available in the latest Canary <input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary <meta> - This feature is available in the latest Canary <script> - This feature is available in the latest Canary <style> - This feature is available in the latest Canary <title> - This feature is available in the latest Canary APIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest Canary prefetchDNS - This feature is available in the latest Canary preinit - This feature is available in the latest Canary preinitModule - This feature is available in the latest Canary preload - This feature is available in the latest Canary preloadModule - This feature is available in the latest Canary render unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server Components Server Components - This feature is available in the latest Canary Server Actions - This feature is available in the latest Canary Directives - This feature is available in the latest Canary 'use client' - This feature is available in the latest Canary 'use server' - This feature is available in the latest Canary Legacy APIs Legacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful? API Reference Client APIs createRoot createRoot lets you create a root to display React components inside a browser DOM node const root = createRoot(domNode, options?)

Reference createRoot(domNode, options?) root.render(reactNode) root.unmount() Usage Rendering an app fully built with React Rendering a page partially built with React Updating a root component Show a dialog for uncaught errors Displaying Error Boundary errors Displaying a dialog for recoverable errors Troubleshooting I've created a root, but nothing is displayed I'm getting an error: "You passed a second argument to root.render" I'm getting an error: "Target container is not a DOM element" I'm getting an error: "Functions are not valid as a React child." My server-rendered HTML gets re-created from scratch

Reference

createRoot(domNode, options?)

Call createRoot to create a React root for displaying content inside a browser DOM element.

```
import { createRoot } from 'react-dom/client';const domNode = document.getElementById('root');const root = createRoot(domNode);
```

React will create a root for the domNode, and take over managing the DOM inside it. After you've created a root, you need to call root.render to display a React component inside of it:

```
root.render(<App />);
```

An app fully built with React will usually only have one `createRoot` call for its root component. A page that uses “sprinkles” of React for parts of the page may have as many separate roots as needed.

See more examples below.

Parameters

`domNode`: A DOM element. React will create a root for this DOM element and allow you to call functions on the root, such as `render` to display rendered React content.

`optional options`: An object with options for this React root.

Canary only optional `onCaughtError`: Callback called when React catches an error in an Error Boundary. Called with the error caught by the Error Boundary, and an `errorInfo` object containing the `componentStack`.

Canary only optional `onUncaughtError`: Callback called when an error is thrown and not caught by an Error Boundary. Called with the error that was thrown, and an `errorInfo` object containing the `componentStack`.

optional `onRecoverableError`: Callback called when React automatically recovers from errors. Called with an error React throws, and an `errorInfo` object containing the `componentStack`. Some recoverable errors may include the original error cause as `error.cause`.

optional `identifierPrefix`: A string prefix React uses for IDs generated by `useId`. Useful to avoid conflicts when using multiple roots on the same page.

Returns

`createRoot` returns an object with two methods: `render` and `unmount`.

Caveats

If your app is server-rendered, using `createRoot()` is not supported. Use `hydrateRoot()` instead.

You’ll likely have only one `createRoot` call in your app. If you use a framework, it might do this call for you.

When you want to render a piece of JSX in a different part of the DOM tree that isn’t a child of your component (for example, a modal or a tooltip), use `createPortal` instead of `createRoot`.

```
root.render(reactNode)
```

Call `root.render` to display a piece of JSX (“React node”) into the React root’s browser DOM node.

```
root.render(<App />);
```

React will display `<App />` in the root, and take over managing the DOM inside it.

See more examples below.

Parameters

`reactNode`: A React node that you want to display. This will usually be a piece of JSX like `<App />`, but you can also pass a React element constructed with `createElement()`, a string, a number, null, or undefined.

Returns

`root.render` returns undefined.

Caveats

The first time you call `root.render`, React will clear all the existing HTML content inside the React root before rendering the React component into it.

If your root's DOM node contains HTML generated by React on the server or during the build, use `hydrateRoot()` instead, which attaches the event handlers to the existing HTML.

If you call `render` on the same root more than once, React will update the DOM as necessary to reflect the latest JSX you passed. React will decide which parts of the DOM can be reused and which need to be recreated by “matching it up” with the previously rendered tree. Calling `render` on the same root again is similar to calling the `set` function on the root component: React avoids unnecessary DOM updates.

`root.unmount()`

Call `root.unmount` to destroy a rendered tree inside a React root.

```
root.unmount();
```

An app fully built with React will usually not have any calls to `root.unmount`.

This is mostly useful if your React root's DOM node (or any of its ancestors) may get removed from the DOM by some other code. For example, imagine a jQuery tab panel that removes inactive tabs from the DOM. If a tab gets removed, everything inside it (including the React roots inside) would get removed from the DOM as well. In that case, you need to tell React to “stop” managing the removed root's content by calling `root.unmount`. Otherwise, the components inside the removed root won't know to clean up and free up global resources like subscriptions.

Calling `root.unmount` will unmount all the components in the root and “detach” React from the root DOM node, including removing any event handlers or state in the tree.

Parameters

`root.unmount` does not accept any parameters.

Returns

`root.unmount` returns `undefined`.

Caveats

Calling `root.unmount` will unmount all the components in the tree and “detach” React from the root DOM node.

Once you call `root.unmount` you cannot call `root.render` again on the same root. Attempting to call `root.render` on an unmounted root will throw a “`Cannot update an unmounted root`” error. However, you can create a new root for the same DOM node after the previous root for that node has been unmounted.

Usage

Rendering an app fully built with React

If your app is fully built with React, create a single root for your entire app.

```
import { createRoot } from 'react-dom/client';const root = createRoot(document.getElementById('root'));root.render(<App />);
```

Usually, you only need to run this code once at startup. It will:

Find the browser DOM node defined in your HTML.

Display the React component for your app inside.

```
index.jsindex.htmlApp.jsindex.js ResetForkimport { createRoot } from 'react-dom/client';import App from './App.js';import './styles.css';const root = createRoot(document.getElementById('root'));root.render(<App />);
```

If your app is fully built with React, you shouldn’t need to create any more roots, or to call `root.render` again.

From this point on, React will manage the DOM of your entire app. To add more components, nest them inside the App component. When you need to update the UI, each of your components can do this by using state. When you need to display extra content like a modal or a tooltip outside the DOM node, render it with a portal.

Note When your HTML is empty, the user sees a blank page until the app's JavaScript code loads and runs: <div id="root"></div> This can feel very slow! To solve this, you can generate the initial HTML from your components on the server or during the build. Then your visitors can read text, see images, and click links before any of the JavaScript code loads. We recommend using a framework that does this optimization out of the box. Depending on when it runs, this is called server-side rendering (SSR) or static site generation (SSG).

Pitfall Apps using server rendering or static generation must call `hydrateRoot` instead of `createRoot`. React will then hydrate (reuse) the DOM nodes from your HTML instead of destroying and re-creating them.

Rendering a page partially built with React

If your page isn't fully built with React, you can call `createRoot` multiple times to create a root for each top-level piece of UI managed by React. You can display different content in each root by calling `root.render`.

Here, two different React components are rendered into two DOM nodes defined in the `index.html` file:

```
index.jsindex.htmlComponents.jsindex.js ResetForkimport './styles.css';
```

```
import { createRoot } from 'react-dom/client';
```

```
import { Comments, Navigation } from './Components.js';
```

```
const navDomNode = document.getElementById('navigation');
```

```
const navRoot = createRoot(navDomNode);
```

```
navRoot.render(<Navigation />);
```

```
const commentDomNode = document.getElementById('comments');
```

```
const commentRoot = createRoot(commentDomNode);
```

```
commentRoot.render(<Comments />);
```

You could also create a new DOM node with `document.createElement()` and add it to the document manually.

```
const domNode = document.createElement('div');const root = createRoot(domNode); root.render(<Comment />);document.body.appendChild(domNode); // You can add it anywhere in the document
```

To remove the React tree from the DOM node and clean up all the resources used by it, call `root.unmount`.

```
root.unmount();
```

This is mostly useful if your React components are inside an app written in a different framework.

Updating a root component

You can call render more than once on the same root. As long as the component tree structure matches up with what was previously rendered, React will preserve the state. Notice how you can type in the input, which means that the updates from repeated render calls every second in this example are not destructive:

```
index.jsApp.jsindex.js ResetForkimport { createRoot } from 'react-dom/client';
import './styles.css';

import App from './App.js';

const root = createRoot(document.getElementById('root'));

let i = 0;

setInterval(() => {
  root.render(<App counter={i} />);
  i++;
}, 1000);
```

It is uncommon to call render multiple times. Usually, your components will update state instead.

Show a dialog for uncaught errors

CanaryonUncaughtError is only available in the latest React Canary release.

By default, React will log all uncaught errors to the console. To implement your own error reporting, you can provide the optional onUncaughtError root option:

```
import { createRoot } from 'react-dom/client';const root = createRoot( document.getElementById('root'), {
onUncaughtError: (error, errorInfo) => {   console.error(     'Uncaught error',     error,
errorInfo.componentStack   ); }
});root.render(<App />);
```

The onUncaughtError option is a function called with two arguments:

The error that was thrown.

An errorInfo object that contains the componentStack of the error.

You can use the onUncaughtError root option to display error dialogs:

```
index.jsApp.jsindex.js ResetForkimport { createRoot } from "react-dom/client";
import App from "./App.js";
import {reportUncaughtError} from "./reportError";
import './styles.css';

const container = document.getElementById("root");
```

```
const root = createRoot(container, {
  onUncaughtError: (error, errorInfo) => {
    if (error.message !== 'Known error') {
      reportUncaughtError({
        error,
        componentStack: errorInfo.componentStack
      });
    }
  }
});
root.render(<App />);
```

Show more

Displaying Error Boundary errors

CanaryonCaughtError is only available in the latest React Canary release.

By default, React will log all errors caught by an Error Boundary to console.error. To override this behavior, you can provide the optional onCaughtError root option to handle errors caught by an Error Boundary:

```
import { createRoot } from 'react-dom/client';const root = createRoot( document.getElementById('root'), {
  onCaughtError: (error, errorInfo) => {    console.error(      'Caught error',      error,      errorInfo.componentStack
);  } });root.render(<App />);
```

The onCaughtError option is a function called with two arguments:

The error that was caught by the boundary.

An errorInfo object that contains the componentStack of the error.

You can use the onCaughtError root option to display error dialogs or filter known errors from logging:

```
index.jsApp.jsindex.js ResetForkimport { createRoot } from "react-dom/client";
import App from "./App.js";
import {reportCaughtError} from "./reportError";
import "./styles.css";

const container = document.getElementById("root");
const root = createRoot(container, {
  onCaughtError: (error, errorInfo) => {
    if (error.message !== 'Known error') {
```

```
reportCaughtError({  
  error,  
  componentStack: errorInfo.componentStack,  
});  
}  
}  
});  
root.render(<App />);
```

Show more

Displaying a dialog for recoverable errors

React may automatically render a component a second time to attempt to recover from an error thrown in render. If successful, React will log a recoverable error to the console to notify the developer. To override this behavior, you can provide the optional `onRecoverableError` root option:

```
import { createRoot } from 'react-dom/client';const root = createRoot( document.getElementById('root'), {  
  onRecoverableError: (error, errorInfo) => {    console.error(      'Recoverable error',      error,      error.cause,  
      errorInfo.componentStack,    );  } });root.render(<App />);
```

The `onRecoverableError` option is a function called with two arguments:

The error that React throws. Some errors may include the original cause as `error.cause`.

An `errorInfo` object that contains the `componentStack` of the error.

You can use the `onRecoverableError` root option to display error dialogs:

```
index.jsApp.jsindex.js ResetForkimport { createRoot } from "react-dom/client";  
  
import App from "./App.js";  
  
import {reportRecoverableError} from "./reportError";  
  
import "./styles.css";  
  
  
const container = document.getElementById("root");  
const root = createRoot(container, {  
  onRecoverableError: (error, errorInfo) => {  
    reportRecoverableError({  
      error,  
      cause: error.cause,  
      componentStack: errorInfo.componentStack,  
    });  
  },  
});
```

```
}

});

root.render(<App />);
```

Show more

Troubleshooting

I've created a root, but nothing is displayed

Make sure you haven't forgotten to actually render your app into the root:

```
import { createRoot } from 'react-dom/client';import App from './App.js';const root = createRoot(document.getElementById('root'));root.render(<App />);
```

Until you do that, nothing is displayed.

I'm getting an error: "You passed a second argument to root.render"

A common mistake is to pass the options for createRoot to root.render(...):

ConsoleWarning: You passed a second argument to root.render(...) but it only accepts one argument.

To fix, pass the root options to createRoot(...), not root.render(...):

```
// Wrong: root.render only takes one argument.root.render(App, {onUncaughtError});// tickmark Correct: pass options to createRoot.const root = createRoot(container, {onUncaughtError}); root.render(<App />);
```

I'm getting an error: "Target container is not a DOM element"

This error means that whatever you're passing to createRoot is not a DOM node.

If you're not sure what's happening, try logging it:

```
const domNode = document.getElementById('root');console.log(domNode); // ???const root = createRoot(domNode);root.render(<App />);
```

For example, if domNode is null, it means that getElementById returned null. This will happen if there is no node in the document with the given ID at the time of your call. There may be a few reasons for it:

The ID you're looking for might differ from the ID you used in the HTML file. Check for typos!

Your bundle's <script> tag cannot "see" any DOM nodes that appear after it in the HTML.

Another common way to get this error is to write createRoot(<App />) instead of createRoot(domNode).

I'm getting an error: "Functions are not valid as a React child."

This error means that whatever you're passing to root.render is not a React component.

This may happen if you call root.render with Component instead of <Component />:

```
// Wrong: App is a function, not a Component.root.render(App); // tickmark Correct: <App /> is a component.root.render(<App />);
```

Or if you pass a function to root.render, instead of the result of calling it:

```
// Wrong: createApp is a function, not a component.root.render(createApp); // tickmark Correct: call createApp to return a component.root.render(createApp());
```

My server-rendered HTML gets re-created from scratch

If your app is server-rendered and includes the initial HTML generated by React, you might notice that creating a root and calling root.render deletes all that HTML, and then re-creates all the DOM nodes from scratch. This can be slower, resets focus and scroll positions, and may lose other user input.

Server-rendered apps must use hydrateRoot instead of createRoot:

```
import { hydrateRoot } from 'react-dom/client'; import App from './App.js'; hydrateRoot( document.getElementById('root'), <App />);
```

Note that its API is different. In particular, usually there will be no further root.render call. Previous Client APIs Next hydrateRoot © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Reference createRoot(domNode, options?) root.render(reactNode) root.unmount() Usage Rendering an app fully built with React Rendering a page partially built with React Updating a root component Show a dialog for uncaught errors Displaying Error Boundary errors Displaying a dialog for recoverable errors Troubleshooting I've created a root, but nothing is displayed I'm getting an error: "You passed a second argument to root.render" I'm getting an error: "Target container is not a DOM element" I'm getting an error: "Functions are not valid as a React child." My server-rendered HTML gets re-created from scratch Not Found –

React React v18.3.1 Search ☰ Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Not Found This page doesn't exist. If this is a mistake, let us know, and we will try to fix it! © 2024 no uwu plzuwu? Logo by @sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms Not Found –

React React v18.3.1 Search ☰ Ctrl K Learn Reference Community Blog GET STARTED Quick Start Tutorial: Tic-Tac-Toe Thinking in React Installation Start a New React Project Add React to an Existing Project Editor Setup Using TypeScript React Developer Tools React Compiler - This feature is available in the latest Canary LEARN REACT Describing the UI Your First Component Importing and Exporting Components Writing Markup with JSX JavaScript in JSX with Curly Braces Passing Props to a Component Conditional Rendering Rendering Lists Keeping Components Pure Your UI as a Tree Adding Interactivity Responding to Events State: A Component's Memory Render and Commit State as a Snapshot Queueing a Series of State Updates Updating Objects in State Updating Arrays in State Managing State Reacting to Input with State Choosing the State Structure Sharing State Between Components Preserving and Resetting State Extracting State Logic into a Reducer Passing Data Deeply with Context Scaling Up

with Reducer and Context Escape Hatches Referencing Values with Refs Manipulating the DOM with Refs Synchronizing with Effects You Might Not Need an Effect Lifecycle of Reactive Effects Separating Events from Effects Removing Effect Dependencies Reusing Logic with Custom Hooks Is this page useful? Learn React Not Found This page doesn't exist. If this is a mistake, let us know, and we will try to fix it! ©2024 no uwu plzuwu? Logo by@sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms React Community – React React v18.3.1 Search Ctrl K Learn Reference Community Blog GET INVOLVED Community React Conferences React Meetups React Videos Meet the Team Docs Contributors Translations Acknowledgements Versioning Policy Is this page useful? Community React Community React has a community of millions of developers. On this page we've listed some React-related communities that you can be a part of; see the other pages in this section for additional online and in-person learning materials.

Code of Conduct

Before participating in React's communities, please read our Code of Conduct. We have adopted the Contributor Covenant and we expect that all community members adhere to the guidelines within.

Stack Overflow

Stack Overflow is a popular forum to ask code-level questions or if you're stuck with a specific error. Read through the existing questions tagged with reactjs or ask your own!

Popular Discussion Forums

There are many online forums which are a great place for discussion about best practices and application architecture as well as the future of React. If you have an answerable code-level question, Stack Overflow is usually a better fit.

Each community consists of many thousands of React users.

DEV's React community

Hashnode's React community

Reactiflux online chat

Reddit's React community

News

For the latest news about React, follow @reactjs on Twitter and the official React blog on this website. Next React Conferences ©2024 no uwu plzuwu? Logo by@sawaratsuki1004 Learn React Quick Start Installation Describing the UI Adding Interactivity Managing State Escape Hatches API Reference React APIs React DOM APIs Community Code of Conduct Meet the Team Docs Contributors Acknowledgements More Blog React Native Privacy Terms On this page Overview Code of Conduct Stack Overflow Popular Discussion Forums News Rules of React – React React v18.3.1 Search Ctrl K Learn Reference Community Blog react@18.3.1 Overview Hooks useActionState - This feature is available in the latest Canary useCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest Canary useReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest Canary createContext forwardRef lazy memo startTransition use - This feature is available in the latest Canary experimental_taintObjectReference - This feature is available in the latest Canary experimental_taintUniqueValue - This feature is available in the latest Canary react-dom@18.3.1 Hooks useFormStatus - This feature is available in the latest Canary Components Common (e.g. <div> <form> - This feature

is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest Canary APIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest Canary prefetchDNS - This feature is available in the latest Canary preinit - This feature is available in the latest Canary preinitModule - This feature is available in the latest Canary preload - This feature is available in the latest Canary preloadModule - This feature is available in the latest Canary render unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of React Overview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server Components Server Components - This feature is available in the latest Canary Server Actions - This feature is available in the latest Canary Directives - This feature is available in the latest Canary 'use client' - This feature is available in the latest Canary 'use server' - This feature is available in the latest Canary Legacy APIs Legacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful? API Reference Rules of React Just as different programming languages have their own ways of expressing concepts, React has its own idioms — or rules — for how to express patterns in a way that is easy to understand and yields high-quality applications.

Components and Hooks must be pure React calls Components and Hooks Rules of Hooks

Note To learn more about expressing UIs with React, we recommend reading Thinking in React.

This section describes the rules you need to follow to write idiomatic React code. Writing idiomatic React code can help you write well organized, safe, and composable applications. These properties make your app more resilient to changes and makes it easier to work with other developers, libraries, and tools.

These rules are known as the Rules of React. They are rules – and not just guidelines – in the sense that if they are broken, your app likely has bugs. Your code also becomes unidiomatic and harder to understand and reason about.

We strongly recommend using Strict Mode alongside React's ESLint plugin to help your codebase follow the Rules of React. By following the Rules of React, you'll be able to find and address these bugs and keep your application maintainable.

Components and Hooks must be pure

Purity in Components and Hooks is a key rule of React that makes your app predictable, easy to debug, and allows React to automatically optimize your code.

Components must be idempotent – React components are assumed to always return the same output with respect to their inputs – props, state, and context.

Side effects must run outside of render – Side effects should not run in render, as React can render components multiple times to create the best possible user experience.

Props and state are immutable – A component's props and state are immutable snapshots with respect to a single render. Never mutate them directly.

Return values and arguments to Hooks are immutable – Once values are passed to a Hook, you should not modify them. Like props in JSX, values become immutable when passed to a Hook.

Values are immutable after being passed to JSX – Don't mutate values after they've been used in JSX. Move the mutation before the JSX is created.

React calls Components and Hooks

React is responsible for rendering components and hooks when necessary to optimize the user experience. It is declarative: you tell React what to render in your component's logic, and React will figure out how best to display it to your user.

Never call component functions directly – Components should only be used in JSX. Don't call them as regular functions.

Never pass around hooks as regular values – Hooks should only be called inside of components. Never pass it around as a regular value.

Rules of Hooks

Hooks are defined using JavaScript functions, but they represent a special type of reusable UI logic with restrictions on where they can be called. You need to follow the Rules of Hooks when using them.

Only call Hooks at the top level – Don't call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function, before any early returns.

Only call Hooks from React functions – Don't call Hooks from regular JavaScript functions.

Components and Hooks must be pure
©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewComponents and Hooks must be pure React calls Components and Hooks Rules of Hooks Directives – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useState - This feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceDirectives - This feature is available in the latest CanaryCanaryThese directives are needed only if you're using React Server Components or building a library compatible with them.

Directives provide instructions to bundlers compatible with React Server Components.

Source code directives

'use client' lets you mark what code runs on the client.

'use server' marks server-side functions that can be called from client-side code.

PreviousServer ActionsNext'use client'©2024no uwu plzuwu?Logo by@sawaratsuki1004Learn ReactQuick StartInstallationDescribing the UIAdding InteractivityManaging StateEscape HatchesAPI ReferenceReact APIsReact DOM APIsCommunityCode of ConductMeet the TeamDocs ContributorsAcknowledgementsMoreBlogReact NativePrivacyTermsOn this pageOverviewSource code directives Legacy React APIs – ReactReactv18.3.1Search⌘KLearnReferenceCommunityBlogreact@18.3.1Overview Hooks useState – This feature is available in the latest CanaryuseCallback useContext useDebugValue useDeferredValue useEffect useId useImperativeHandle useInsertionEffect useLayoutEffect useMemo useOptimistic - This feature is available in the latest CanaryuseReducer useRef useState useSyncExternalStore useTransition Components <Fragment> (<>) <Profiler> <StrictMode> <Suspense> APIs act cache - This feature is available in the latest CanarycreateContext forwardRef lazy memo startTransition use - This feature is available in the latest Canaryexperimental_taintObjectReference - This feature is available in the latest Canaryexperimental_taintUniqueValue - This feature is available in the latest Canaryreact-dom@18.3.1Hooks useFormStatus - This feature is available in the latest CanaryComponents Common (e.g. <div> <form> - This feature is available in the latest Canary<input> <option> <progress> <select> <textarea> <link> - This feature is available in the latest Canary<meta> - This feature is available in the latest Canary<script> - This feature is available in the latest Canary<style> - This feature is available in the latest Canary<title> - This feature is available in the latest CanaryAPIs createPortal flushSync findDOMNode hydrate preconnect - This feature is available in the latest CanaryprefetchDNS - This feature is available in the latest Canarypreinit - This feature is available in the latest CanarypreinitModule - This feature is available in the latest Canarypreload - This feature is available in the latest CanarypreloadModule - This feature is available in the latest Canaryrender unmountComponentAtNode Client APIs createRoot hydrateRoot Server APIs renderToNodeStream renderToPipeableStream renderToReadableStream renderToStaticMarkup renderToStaticNodeStream renderToString Rules of ReactOverview Components and Hooks must be pure React calls Components and Hooks Rules of Hooks React Server ComponentsServer Components - This feature is available in the latest CanaryServer Actions - This feature is available in the latest CanaryDirectives - This feature is available in the latest Canary'use client' - This feature is available in the latest Canary'use server' - This feature is available in the latest CanaryLegacy APIsLegacy React APIs Children createElement Component createElement createFactory createRef isValidElement PureComponent Is this page useful?API ReferenceLegacy React APIsThese APIs are exported from the react package, but they are not recommended for use in newly written code. See the linked individual API pages for the suggested alternatives.

Legacy APIs

Children lets you manipulate and transform the JSX received as the children prop. See alternatives.

createElement lets you create a React element using another element as a starting point. See alternatives.

Component lets you define a React component as a JavaScript class. See alternatives.

createElement lets you create a React element. Typically, you'll use JSX instead.

createRef creates a ref object which can contain arbitrary value. See alternatives.

isValidElement checks whether a value is a React element. Typically used with createElement.

PureComponent is similar to Component, but it skip re-renders with same props. See alternatives.

Deprecated APIs

DeprecatedThese APIs will be removed in a future major version of React.

createFactory lets you create a function that produces React elements of a certain type.