

REACT Docs - English

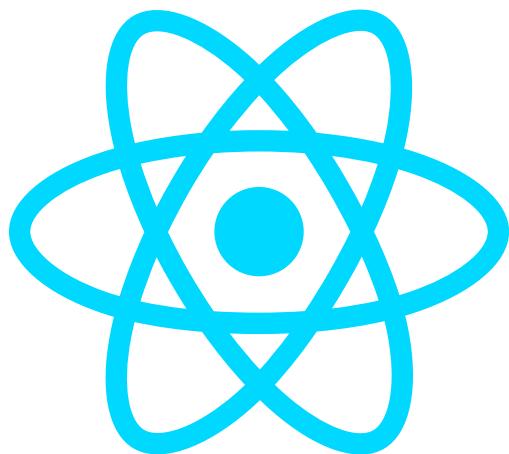


Table of contents

• Add react to an existing project	4
• Adding interactivity	8
• Choosing the state structure	21
• Conditional rendering	62
• Describing the ui	74
• Editor setup	82
• Escape hatches	84
• Extracting state logic into a reducer	98
• Importing and exporting components	138
• Installation	145
• Javascript in jsx with curly braces	147
• Keeping components pure	156
• Lifecycle of reactive effects	170
• Managing state	206
• Manipulating the dom with refs	222
• Passing data deeply with context	241
• Passing props to a component	260
• Preserving and resetting state	278
• Queueing a series of state updates	311
• React developer tools	321
• React index	323
• Reacting to input with state	332
• Referencing values with refs	352
• Removing effect dependencies	363
• Render and commit	402
• Rendering lists	406
• Responding to events	425
• Reusing logic with custom hooks	437
• Scaling up with reducer and context	477
• Separating events from effects	498
• Sharing state between components	528
• Start a new react project	538
• State a components memory	542
• State as a snapshot	571

• Synchronizing with effects	578
• Thinking in react	607
• Tutorial tic tac toe	619
• Updating arrays in state	670
• Updating objects in state	700
• Writing markup with jsx	726
• You might not need an effect	732
• Your first component	763

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.

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 {/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:

1. **Build the React part of your app** using one of the [React-based frameworks](#).
2. **Specify `/some-app` as the base path** in your framework's configuration (here's how: [Next.js](#), [Gatsby](#)).
3. **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 {/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:

1. **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.
2. **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 {/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:

```
npm 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`):

```
<!DOCTYPE html>
<html>
  <head><title>My app</title></head>
  <body>
    <!-- Your existing page content (in this example, it gets replaced) -->
  </body>
</html>
```

```
import { 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.

Integrating 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 {/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 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>);
```

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:

```
<!DOCTYPE html>
<html>
  <head><title>My app</title></head>
  <body>
    <p>This paragraph is a part of HTML.</p>
    <nav id="navigation"></nav>
    <p>This paragraph is also a part of HTML.</p>
  </body>
</html>
```

```
import { 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 *{/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.

Adding Interactivity

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

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

Responding to events {/responding-to-events/}

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

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

```
export 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>
  );
}
```

```

        </button>
    );
}

button { margin-right: 10px; }

```

Read [Responding to Events](#) to learn how to add event handlers.

State: a component's memory *{/state-a-components-memory/}*

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

You can add state to a component with a `useState` Hook. *Hooks* are special functions that let your components use React features (state is one of those features). The `useState` Hook lets you declare a state variable. It takes the initial state and returns a pair of values: the current state, and a state setter function that lets you update it.

```

const [index, setIndex] = useState(0);
const [showMore, setShowMore] = useState(false);

```

Here is how an image gallery uses and updates state on click:

```

import { useState } from 'react';
import { sculptureList } from './data.js';

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

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

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

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleNextClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
      </h2>
      <img alt={sculpture.name} src={sculpture.url} />
      <div>
        {sculpture.description}
      </div>
      {sculpture.images.map((image, i) => (
        <img alt={image.name} key={i} src={image.url} />
      ))}
    </>
  );
}

```

```

        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}
      />
    </>
  );
}

```

```

export const 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/Sd1AgU0m.jpg',
    alt: 'A large mosaic sculpture of a whimsical dancing female figure in a'
  }
]

```

```

colorful costume emanating joy.'
}, {
  name: 'Ultimate Form',
  artist: 'Barbara Hepworth',
  description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',
  url: 'https://i.imgur.com/2heNQDcm.jpg',
  alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'
}, {
  name: 'Cavaliere',
  artist: 'Lamidi Olonade Fakeye',
  description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",
  url: 'https://i.imgur.com/wIdGuZwm.png',
  alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'
}, {
  name: 'Big Bellies',
  artist: 'Alina Szapocznikow',
  description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",
  url: 'https://i.imgur.com/AlHTAdDm.jpg',
  alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'
}, {
  name: 'Terracotta Army',
  artist: 'Unknown Artist',
  description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',
  url: 'https://i.imgur.com/HMFmH6m.jpg',
  alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'
}, {
  name: 'Lunar Landscape',
  artist: 'Louise Nevelson',
  description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism's geometric abstraction of space and form.',
  url: 'https://i.imgur.com/rN7hY6om.jpg',
  alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'
}, {
  name: 'Aureole',
  artist: 'Ranjani Shettar',
  description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',
  url: 'https://i.imgur.com/okTpbHhm.jpg',
  alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'
}, {
  name: 'Hippos',

```

```

    artist: 'Taipei Zoo',
    description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged
    hippos at play.',
    url: 'https://i.imgur.com/6o5Vuyu.jpg',
    alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if
    they were swimming.'
}];

```

```

h2 { margin-top: 10px; margin-bottom: 0; }
h3 {
  margin-top: 5px;
  font-weight: normal;
  font-size: 100%;
}
img { width: 120px; height: 120px; }
button {
  display: block;
  margin-top: 10px;
  margin-bottom: 10px;
}

```

Read [State: A Component's Memory](#) to learn how to remember a value and update it on interaction.

Render and commit *{/render-and-commit/}*

Before your components are displayed on the screen, they must be rendered by React. Understanding the steps in this process will help you think about how your code executes and explain its behavior.

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

1. **Triggering** a render (delivering the diner's order to the kitchen)
2. **Rendering** the component (preparing the order in the kitchen)
3. **Committing** to the DOM (placing the order on the table)

Read [Render and Commit](#) to learn the lifecycle of a UI update.

State as a snapshot *{/state-as-a-snapshot/}*

Unlike regular JavaScript variables, React state behaves more like a snapshot. Setting it does not change the state variable you already have, but instead triggers a re-render. This can be surprising at first!

```

console.log(count); // 0
setCount(count + 1); // Request a re-render with 1
console.log(count); // Still 0!

```

This behavior helps you avoid subtle bugs. Here is a little chat app. Try to guess what happens if you press "Send" first and *then* change the recipient to Bob. Whose name will appear in the `alert` five seconds later?

```

import { useState } from 'react';
export default function Form() {

```

```

const [to, setTo] = useState('Alice');
const [message, setMessage] = useState('Hello');

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

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

```

```
label, textarea { margin-bottom: 10px; display: block; }
```

Read [State as a Snapshot](#) to learn why state appears "fixed" and unchanging inside the event handlers.

Queueing a series of state updates *{/queueing-a-series-of-state-updates/}*

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

```

import { useState } from 'react';

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

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

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

```

react

```
</>
})
```

```
button { display: inline-block; margin: 10px; font-size: 20px; }
```

State as a Snapshot explains why this is happening. Setting state requests a new re-render, but does not change it in the already running code. So `score` continues to be `0` right after you call `setScore(score + 1)`.

```
console.log(score); // 0
setScore(score + 1); // setScore(0 + 1);
console.log(score); // 0
setScore(score + 1); // setScore(0 + 1);
console.log(score); // 0
setScore(score + 1); // setScore(0 + 1);
console.log(score); // 0
```

You can fix this by passing an *update function* when setting state. Notice how replacing `setScore(score + 1)` with `setScore(s => s + 1)` fixes the "+3" button. This lets you queue multiple state updates.

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

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

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

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

```
button { display: inline-block; margin: 10px; font-size: 20px; }
```

Read [Queueing a Series of State Updates](#) to learn how to queue a sequence of state updates.

Updating objects in state { /updating-objects-in-state/ }

State can hold any kind of JavaScript value, including objects. But you shouldn't change objects and arrays that you hold in the React state directly. Instead, when you want to update an object and array, you need to create a new one (or make a copy of an existing one), and then update the state to use that copy.

Usually, you will use the `...` spread syntax to copy objects and arrays that you want to change. For example, updating a nested object could look like this:

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

export default function Form() {
  const [person, setPerson] = useState({
    name: 'Niki de Saint Phalle',
    artwork: {
      # 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgU0m.jpg',
    }
  });

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

  function handleTitleChange(e) {
    setPerson({
      ...person,
      artwork: {
        ...person.artwork,
        # 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>
    </>
  );
}
```

react

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

```
label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
img { width: 200px; height: 200px; }
```

If copying objects in code gets tedious, you can use a library like [Immer](#) to reduce repetitive code:

```
import { useImmer } from 'use-immer';

export default function Form() {
  const [person, updatePerson] = useImmer({
    name: 'Niki de Saint Phalle',
    artwork: {
      # 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgU0m.jpg',
    }
  });

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

  function handleTitleChange(e) {
```

```

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

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

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

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

```

```
{
  "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"
  }
}
```

```
label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
img { width: 200px; height: 200px; }
```

Read [Updating Objects in State](#) to learn how to update objects correctly.

Updating arrays in state *{/updating-arrays-in-state/}*

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

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

let nextId = 3;
const initialList = [
];

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

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

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

```

    );
}

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

```

If copying arrays in code gets tedious, you can use a library like [Immer](#) to reduce repetitive code:

```

import { useState } from 'react';
import { useImmer } from 'use-immer';

let nextId = 3;
const initialList = [];

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

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

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

function ItemList({ artworks, onToggle }) {
  return (

```

```

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

```

```
{
  "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"
  }
}
```

Read [Updating Arrays in State](#) to learn how to update arrays correctly.

What's next? {/whats-next/}

Head over to [Responding to Events](#) to start reading this chapter page by page!

Or, if you're already familiar with these topics, why not read about [Managing State](#)?

Choosing the State Structure

Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs. Here are some tips you should consider when structuring state.

- 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 {/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:

1. **Group related state.** If you always update two or more state variables at the same time, consider merging them into a single state variable.
2. **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.
3. **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.
4. **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.
5. **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 {/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:

```
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>
  )
}

body { margin: 0; padding: 0; height: 250px; }
```

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.

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 {/avoid-contradictions-in-state/}

Here is a hotel feedback form with `isSubmitting` and `isSent` state variables:

```

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);
  });
}

```

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`' :

```

import { 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);
  });
}

```

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 {/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?

```

import { 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(firstName + ' ' + 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>
    </>
  );
}

```

```
label { display: block; margin-bottom: 5px; }
```

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:

```

import { 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);
  }

```

react

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

```
label { display: block; margin-bottom: 5px; }
```

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.

Don't mirror props in state *(/don-t-mirror-props-in-state/)*

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 `{/avoid-duplication-in-state/}`

This menu list component lets you choose a single travel snack out of several:

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

const initialItems = [
];

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>
    </>
  );
}

button { margin-top: 10px; }
```

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:

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

const initialItems = [
```

react

```
];
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,
          # 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>
    </>
  );
}
```

```
button { margin-top: 10px; }
```

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:

```

import { useState } from 'react';

const initialItems = [
];

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,
          # 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>
    </>
  );
}

```

```
button { margin-top: 10px; }
```

(Alternatively, you may hold the selected index in state.)

The state used to be duplicated like this:

But after the change it's like this:

- 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 {/avoid-deeply-nested-state/}

Imagine a travel plan consisting of planets, continents, and countries. You might be tempted to structure its state using nested objects and arrays, like in this example:

```
import { useState } from 'react';
import { initialTravelPlan } from './places.js';

function PlaceTree({ place }) {
  const childPlaces = place.childPlaces;
  return (
    <li>
      {place.title}
      {childPlaces.length > 0 && (
        <ol>
          {childPlaces.map(place => (
            <PlaceTree key={place.id} place={place} />
          ))}
        </ol>
      )}
    </li>
  );
}

export default function TravelPlan() {
  const [plan, setPlan] = useState(initialTravelPlan);
  const planets = plan.childPlaces;
  return (
    <>
      <h2>Places to visit</h2>
      <ol>
        {planets.map(place => (
          <PlaceTree key={place.id} place={place} />
        ))}
      </ol>
    </>
  );
}

export const initialTravelPlan = {
  # '(Root)',
  childPlaces: [
    # 'Earth',
    childPlaces: [
      # 'Africa',
      childPlaces: [
        # 'Botswana',
        childPlaces: []
      ],
      {
        # 'Asia',
        childPlaces: [
          # 'China',
          childPlaces: []
        ]
      }
    ],
    {
      # 'North America',
      childPlaces: [
        # 'USA',
        childPlaces: [
          # 'California',
          childPlaces: []
        ]
      ]
    }
  ],
  {
    # 'South America',
    childPlaces: [
      # 'Brazil',
      childPlaces: [
        # 'Rio de Janeiro',
        childPlaces: []
      ]
    ]
  }
};
```

```
# 'Egypt',
childPlaces: []
}, {
# 'Kenya',
childPlaces: []
}, {
# 'Madagascar',
childPlaces: []
}, {
# 'Morocco',
childPlaces: []
}, {
# 'Nigeria',
childPlaces: []
}, {
# 'South Africa',
childPlaces: []
}]
}, {
# 'Americas',
childPlaces: [
{
# 'Argentina',
childPlaces: []
}, {
# 'Brazil',
childPlaces: []
}, {
# 'Barbados',
childPlaces: []
}, {
# 'Canada',
childPlaces: []
}, {
# 'Jamaica',
childPlaces: []
}, {
# 'Mexico',
childPlaces: []
}, {
# 'Trinidad and Tobago',
childPlaces: []
}, {
# 'Venezuela',
childPlaces: []
}
]
}, {
# 'Asia',
childPlaces: [
{
# 'China',
childPlaces: []
}, {
# 'Hong Kong',
childPlaces: []
}, {
# 'India',
childPlaces: []
}, {
# 'Singapore',
childPlaces: []
}, {
# 'South Korea',
childPlaces: []
]
```

```
    }, {
      "# 'Thailand',
      childPlaces: []
    }, {
      "# 'Vietnam',
      childPlaces: []
    }]
}, {
  "# 'Europe',
  childPlaces: [{{
    "# 'Croatia',
    childPlaces: []
  }, {
    "# 'France',
    childPlaces: []
  }, {
    "# 'Germany',
    childPlaces: []
  }, {
    "# 'Italy',
    childPlaces: []
  }, {
    "# 'Portugal',
    childPlaces: []
  }, {
    "# 'Spain',
    childPlaces: []
  }, {
    "# 'Turkey',
    childPlaces: []
  }]
}, {
  "# 'Oceania',
  childPlaces: [{{
    "# 'Australia',
    childPlaces: []
  }, {
    "# 'Bora Bora (French Polynesia)',
    childPlaces: []
  }, {
    "# 'Easter Island (Chile)',
    childPlaces: []
  }, {
    "# 'Fiji',
    childPlaces: []
  }, {
    "# 'Hawaii (the USA)',
    childPlaces: []
  }, {
    "# 'New Zealand',
    childPlaces: []
  }, {
    "# 'Vanuatu',
    childPlaces: []
  }]
}], {
  "# 'Moon',
  childPlaces: [{{
    "# 'Rheita',
    childPlaces: []
  }, {

```

```

        # 'Piccolomini',
        childPlaces: []
    }, {
        # 'Tycho',
        childPlaces: []
    }]
}, {
    # 'Mars',
    childPlaces: [
        # 'Corn Town',
        childPlaces: []
    ],
    # 'Green Hill',
    childPlaces: []
}]
};

}

```

Now let's say you want to add a button to delete a place you've already visited. How would you go about it? [Updating nested state](#) involves making copies of objects all the way up from the part that changed. Deleting a deeply nested place would involve copying its entire parent place chain. Such code can be very verbose.

If the state is too nested to update easily, consider making it "flat". Here is one way you can re-structure this data. Instead of a tree-like structure where each `place` has an array of *its child places*, you can have each place hold an array of *its child place IDs*. Then store a mapping from each place ID to the corresponding place.

This data restructuring might remind you of seeing a database table:

```

import { useState } from 'react';
import { initialTravelPlan } from './places.js';

function PlaceTree({ id, placesById }) {
    const place = placesById[id];
    const childIds = place.childIds;
    return (
        <li>
            {place.title}
            {childIds.length > 0 && (
                <ol>
                    {childIds.map(childId => (
                        <PlaceTree
                            key={childId}
                            id={childId}
                            placesById={placesById}
                        />
                    ))}
                </ol>
            )}
        </li>
    );
}

export default function TravelPlan() {
    const [plan, setPlan] = useState(initialTravelPlan);
    const root = plan[0];
    const planetIds = root.childIds;
    return (
        <>

```

```

<h2>Places to visit</h2>
<ol>
  {planetIds.map(id => (
    <PlaceTree
      key={id}
      id={id}
      placesById={plan}
    />
  ))}
</ol>
</>
};

export const initialTravelPlan = {
  0: {
    # '(Root)',
    childIds: [1, 43, 47],
  },
  1: {
    # 'Earth',
    childIds: [2, 10, 19, 27, 35]
  },
  2: {
    # 'Africa',
    childIds: [3, 4, 5, 6, 7, 8, 9]
  },
  3: {
    # 'Botswana',
    childIds: []
  },
  4: {
    # 'Egypt',
    childIds: []
  },
  5: {
    # 'Kenya',
    childIds: []
  },
  6: {
    # 'Madagascar',
    childIds: []
  },
  7: {
    # 'Morocco',
    childIds: []
  },
  8: {
    # 'Nigeria',
    childIds: []
  },
  9: {
    # 'South Africa',
    childIds: []
  },
  10: {
    # 'Americas',
    childIds: [11, 12, 13, 14, 15, 16, 17, 18],
  },
  11: {
    # 'Argentina',
    childIds: []
  }
};

```

```
  },
12: {
  # 'Brazil',
  childIds: []
},
13: {
  # 'Barbados',
  childIds: []
},
14: {
  # 'Canada',
  childIds: []
},
15: {
  # 'Jamaica',
  childIds: []
},
16: {
  # 'Mexico',
  childIds: []
},
17: {
  # 'Trinidad and Tobago',
  childIds: []
},
18: {
  # 'Venezuela',
  childIds: []
},
19: {
  # 'Asia',
  childIds: [20, 21, 22, 23, 24, 25, 26],
},
20: {
  # 'China',
  childIds: []
},
21: {
  # 'Hong Kong',
  childIds: []
},
22: {
  # 'India',
  childIds: []
},
23: {
  # 'Singapore',
  childIds: []
},
24: {
  # 'South Korea',
  childIds: []
},
25: {
  # 'Thailand',
  childIds: []
},
26: {
  # 'Vietnam',
  childIds: []
},
27: {
```

```
# 'Europe',
  childIds: [28, 29, 30, 31, 32, 33, 34],
},
28: {
  # 'Croatia',
  childIds: []
},
29: {
  # 'France',
  childIds: []
},
30: {
  # 'Germany',
  childIds: []
},
31: {
  # 'Italy',
  childIds: []
},
32: {
  # 'Portugal',
  childIds: []
},
33: {
  # 'Spain',
  childIds: []
},
34: {
  # 'Turkey',
  childIds: []
},
35: {
  # 'Oceania',
  childIds: [36, 37, 38, 39, 40, 41, 42],
},
36: {
  # 'Australia',
  childIds: []
},
37: {
  # 'Bora Bora (French Polynesia)',
  childIds: []
},
38: {
  # 'Easter Island (Chile)',
  childIds: []
},
39: {
  # 'Fiji',
  childIds: []
},
40: {
  # 'Hawaii (the USA)',
  childIds: []
},
41: {
  # 'New Zealand',
  childIds: []
},
42: {
  # 'Vanuatu',
  childIds: []
}
```

```

},
43: {
  # 'Moon',
  childIds: [44, 45, 46]
},
44: {
  # 'Rheita',
  childIds: []
},
45: {
  # 'Piccolomini',
  childIds: []
},
46: {
  # 'Tycho',
  childIds: []
},
47: {
  # 'Mars',
  childIds: [48, 49]
},
48: {
  # 'Corn Town',
  childIds: []
},
49: {
  # 'Green Hill',
  childIds: []
}
};

```

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:

```

import { 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>
  );
}

```

```

export const initialTravelPlan = {
  0: {
    # '(Root)',
    childIds: [1, 43, 47],
  },
  1: {
    # 'Earth',
    childIds: [2, 10, 19, 27, 35]
  },
  2: {

```

```
# 'Africa',
  childIds: [3, 4, 5, 6, 7, 8, 9]
},
3: {
  # 'Botswana',
  childIds: []
},
4: {
  # 'Egypt',
  childIds: []
},
5: {
  # 'Kenya',
  childIds: []
},
6: {
  # 'Madagascar',
  childIds: []
},
7: {
  # 'Morocco',
  childIds: []
},
8: {
  # 'Nigeria',
  childIds: []
},
9: {
  # 'South Africa',
  childIds: []
},
10: {
  # 'Americas',
  childIds: [11, 12, 13, 14, 15, 16, 17, 18],
},
11: {
  # 'Argentina',
  childIds: []
},
12: {
  # 'Brazil',
  childIds: []
},
13: {
  # 'Barbados',
  childIds: []
},
14: {
  # 'Canada',
  childIds: []
},
15: {
  # 'Jamaica',
  childIds: []
},
16: {
  # 'Mexico',
  childIds: []
},
17: {
  # 'Trinidad and Tobago',
  childIds: []
}
```

```
  },
18: {
  # 'Venezuela',
  childIds: []
},
19: {
  # 'Asia',
  childIds: [20, 21, 22, 23, 24, 25, 26],
},
20: {
  # 'China',
  childIds: []
},
21: {
  # 'Hong Kong',
  childIds: []
},
22: {
  # 'India',
  childIds: []
},
23: {
  # 'Singapore',
  childIds: []
},
24: {
  # 'South Korea',
  childIds: []
},
25: {
  # 'Thailand',
  childIds: []
},
26: {
  # 'Vietnam',
  childIds: []
},
27: {
  # 'Europe',
  childIds: [28, 29, 30, 31, 32, 33, 34],
},
28: {
  # 'Croatia',
  childIds: []
},
29: {
  # 'France',
  childIds: []
},
30: {
  # 'Germany',
  childIds: []
},
31: {
  # 'Italy',
  childIds: []
},
32: {
  # 'Portugal',
  childIds: []
},
33: {
```

```

        # 'Spain',
        childIds: []
    },
34: {
    # 'Turkey',
    childIds: []
},
35: {
    # 'Oceania',
    childIds: [36, 37, 38, 39, 40, 41,, 42],
},
36: {
    # 'Australia',
    childIds: []
},
37: {
    # 'Bora Bora (French Polynesia)',
    childIds: []
},
38: {
    # 'Easter Island (Chile)',
    childIds: []
},
39: {
    # 'Fiji',
    childIds: []
},
40: {
    # 'Hawaii (the USA)',
    childIds: []
},
41: {
    # 'New Zealand',
    childIds: []
},
42: {
    # 'Vanuatu',
    childIds: []
},
43: {
    # 'Moon',
    childIds: [44, 45, 46]
},
44: {
    # 'Rheita',
    childIds: []
},
45: {
    # 'Piccolomini',
    childIds: []
},
46: {
    # 'Tycho',
    childIds: []
},
47: {
    # 'Mars',
    childIds: [48, 49]
},
48: {
    # 'Corn Town',
    childIds: []
}

```

```

    },
49: {
  # 'Green Hill',
  childIds: []
}
};


```

```
button { margin: 10px; }
```

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.

Improving memory usage [{/improving-memory-usage/}](#)

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.

```

import { useImmer } from 'use-immer';
import { initialTravelPlan } from './places.js';

export default function TravelPlan() {
  const [plan, updatePlan] = useImmer(initialTravelPlan);

  function handleComplete(parentId, childId) {
    updatePlan(draft => {
      // Remove from the parent place's child IDs.
      const parent = draft[parentId];
      parent.childIds = parent.childIds
        .filter(id => id !== childId);

      // Forget this place and all its subtree.
      deleteAllChildren(childId);
      function deleteAllChildren(id) {
        const place = draft[id];
        place.childIds.forEach(deleteAllChildren);
        delete draft[id];
      }
    });
  }

  const root = plan[0];
  const planetIds = root.childIds;
  return (
    <>
      <h2>Places to visit</h2>
      <ol>
        {planetIds.map(id => (
          <PlaceTree
            key={id}
            id={id}
            parentId={0}
            placesById={plan}
            onComplete={handleComplete}
          />
        ))}
      </ol>
    </>
  );
}


```

```

function PlaceTree({ id, parentId, placesById, onComplete }) {
  const place = placesById[id];
  const childIds = place.childIds;
  return (
    <li>
      {place.title}
      <button onClick={() => {
        onComplete(parentId, id);
      }}>
        Complete
      </button>
      {childIds.length > 0 &&
        <ol>
          {childIds.map(childId => (
            <PlaceTree
              key={childId}
              id={childId}
              parentId={id}
              placesById={placesById}
              onComplete={onComplete}
            />
          ))}
        </ol>
      }
    </li>
  );
}

```

```

export const initialTravelPlan = {
  0: {
    # '(Root)',
    childIds: [1, 43, 47],
  },
  1: {
    # 'Earth',
    childIds: [2, 10, 19, 27, 35]
  },
  2: {
    # 'Africa',
    childIds: [3, 4, 5, 6, 7, 8, 9]
  },
  3: {
    # 'Botswana',
    childIds: []
  },
  4: {
    # 'Egypt',
    childIds: []
  },
  5: {
    # 'Kenya',
    childIds: []
  },
  6: {
    # 'Madagascar',
    childIds: []
  },
  7: {
    # 'Morocco',
    childIds: []
  },
  8: {

```

```
# 'Nigeria',
childIds: []
},
9: {
# 'South Africa',
childIds: []
},
10: {
# 'Americas',
childIds: [11, 12, 13, 14, 15, 16, 17, 18],
},
11: {
# 'Argentina',
childIds: []
},
12: {
# 'Brazil',
childIds: []
},
13: {
# 'Barbados',
childIds: []
},
14: {
# 'Canada',
childIds: []
},
15: {
# 'Jamaica',
childIds: []
},
16: {
# 'Mexico',
childIds: []
},
17: {
# 'Trinidad and Tobago',
childIds: []
},
18: {
# 'Venezuela',
childIds: []
},
19: {
# 'Asia',
childIds: [20, 21, 22, 23, 24, 25, 26],
},
20: {
# 'China',
childIds: []
},
21: {
# 'Hong Kong',
childIds: []
},
22: {
# 'India',
childIds: []
},
23: {
# 'Singapore',
childIds: []
}
```

```

},
24: {
  # 'South Korea',
  childIds: []
},
25: {
  # 'Thailand',
  childIds: []
},
26: {
  # 'Vietnam',
  childIds: []
},
27: {
  # 'Europe',
  childIds: [28, 29, 30, 31, 32, 33, 34],
},
28: {
  # 'Croatia',
  childIds: []
},
29: {
  # 'France',
  childIds: []
},
30: {
  # 'Germany',
  childIds: []
},
31: {
  # 'Italy',
  childIds: []
},
32: {
  # 'Portugal',
  childIds: []
},
33: {
  # 'Spain',
  childIds: []
},
34: {
  # 'Turkey',
  childIds: []
},
35: {
  # 'Oceania',
  childIds: [36, 37, 38, 39, 40, 41,, 42],
},
36: {
  # 'Australia',
  childIds: []
},
37: {
  # 'Bora Bora (French Polynesia)',
  childIds: []
},
38: {
  # 'Easter Island (Chile)',
  childIds: []
},
39: {
}

```

react

```
# 'Fiji',
  childIds: []
},
40: {
  # 'Hawaii (the USA)',
  childIds: []
},
41: {
  # 'New Zealand',
  childIds: []
},
42: {
  # 'Vanuatu',
  childIds: []
},
43: {
  # 'Moon',
  childIds: [44, 45, 46]
},
44: {
  # 'Rheita',
  childIds: []
},
45: {
  # 'Piccolomini',
  childIds: []
},
46: {
  # 'Tycho',
  childIds: []
},
47: {
  # 'Mars',
  childIds: [48, 49]
},
48: {
  # 'Corn Town',
  childIds: []
},
49: {
  # 'Green Hill',
  childIds: []
};
};
```

```
button { margin: 10px; }
```

```
{
  "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"
  }
}
```

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.

- 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.

Fix a component that's not updating [{/fix-a-component-thats-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.

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

export default function Clock(props) {
  const [color, setColor] = useState(props.color);
  return (
    <h1 style={{ color: color }}>
      {props.time}
    </h1>
  );
}

import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return time;
}

export default function App() {
  const time = useTime();
  const [color, setColor] = useState('lightcoral');
  return (
    <div>
      <p>
        Pick a color:{' '}
        <select value={color} onChange={e => setColor(e.target.value)}>
          <option value="lightcoral">lightcoral</option>
          <option value="midnightblue">midnightblue</option>
          <option value="rebeccapurple">rebeccapurple</option>
        </select>
      </p>
    </div>
  );
}
```

react

```
</p>
<Clock color={color} time={time.toLocaleTimeString()} />
</div>
);
}
```

The issue is that this component has `color` state initialized with the initial value of the `color` prop. But when the `color` prop changes, this does not affect the state variable! So they get out of sync. To fix this issue, remove the state variable altogether, and use the `color` prop directly.

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

export default function Clock(props) {
  return (
    <h1 style={{ color: props.color }}>
      {props.time}
    </h1>
  );
}
```

```
import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return time;
}

export default function App() {
  const time = useTime();
  const [color, setColor] = useState('lightcoral');
  return (
    <div>
      <p>
        Pick a color:{' '}
        <select value={color} onChange={e => setColor(e.target.value)}>
          <option value="lightcoral">lightcoral</option>
          <option value="midnightblue">midnightblue</option>
          <option value="rebeccapurple">rebeccapurple</option>
        </select>
      </p>
      <Clock color={color} time={time.toLocaleTimeString()} />
    </div>
  );
}
```

Or, using the destructuring syntax:

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

export default function Clock({ color, time }) {
  return (
    <h1 style={{ color: color }}>
```

```

    {time}
  </h1>
);
}

```

```

import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return time;
}

export default function App() {
  const time = useTime();
  const [color, setColor] = useState('lightcoral');
  return (
    <div>
      <p>
        Pick a color:{' '}
        <select value={color} onChange={e => setColor(e.target.value)}>
          <option value="lightcoral">lightcoral</option>
          <option value="midnightblue">midnightblue</option>
          <option value="rebeccapurple">rebeccapurple</option>
        </select>
      </p>
      <Clock color={color} time={time.toLocaleTimeString()} />
    </div>
  );
}

```

Fix a broken packing list {/fix-a-broken-packing-list/}

This packing list has a footer that shows how many items are packed, and how many items there are overall. It seems to work at first, but it is buggy. For example, if you mark an item as packed and then delete it, the counter will not be updated correctly. Fix the counter so that it's always correct.

Is any state in this example redundant?

```

import { useState } from 'react';
import AddItem from './AddItem.js';
import PackingList from './PackingList.js';

let nextId = 3;
const initialItems = [];

export default function TravelPlan() {
  const [items, setItems] = useState(initialItems);
  const [total, setTotal] = useState(3);
  const [packed, setPacked] = useState(1);

  function handleAddItem(title) {
    setTotal(total + 1);
  }
}

```

react

```
setItems([
  ...items,
  {
    # title,
    packed: false
  }
]);
}

function handleChangeItem(nextItem) {
  if (nextItem.packed) {
    setPacked(packed + 1);
  } else {
    setPacked(packed - 1);
  }
  setItems(items.map(item => {
    if (item.id === nextItem.id) {
      return nextItem;
    } else {
      return item;
    }
  }));
}

function handleDeleteItem(itemId) {
  setTotal(total - 1);
  setItems(
    items.filter(item => item.id !== itemId)
  );
}

return (
  <>
    <AddItem
      onAddItem={handleAddItem}
    />
    <PackingList
      items={items}
      onChangeItem={handleChangeItem}
      onDeleteItem={handleDeleteItem}
    />
    <hr />
    <b>{packed} out of {total} packed!</b>
  </>
);
}
```

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

export default function AddItem({ onAddItem }) {
  const [title, setTitle] = useState('');
  return (
    <>
      <input
        placeholder="Add item"
        value={title}
        onChange={e => setTitle(e.target.value)}
      />
      <button onClick={() => {
        setTitle('');
        onAddItem(title);
      }}>Add</button>
    </>
  );
}
```

```

        </>
    }

import { useState } from 'react';

export default function PackingList({
  items,
  onChangeItem,
  onDeleteItem
}) {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>
          <label>
            <input
              type="checkbox"
              checked={item.packed}
              onChange={e => {
                onChangeItem({
                  ...item,
                  packed: e.target.checked
                });
              }}
            />
            {' '}
            {item.title}
          </label>
          <button onClick={() => onDeleteItem(item.id)}>
            Delete
          </button>
        </li>
      ))}
    </ul>
  );
}

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

Although you could carefully change each event handler to update the `total` and `packed` counters correctly, the root problem is that these state variables exist at all. They are redundant because you can always calculate the number of items (packed or total) from the `items` array itself. Remove the redundant state to fix the bug:

```

import { useState } from 'react';
import AddItem from './AddItem.js';
import PackingList from './PackingList.js';

let nextId = 3;
const initialItems = [];

export default function TravelPlan() {
  const [items, setItems] = useState(initialItems);

  const total = items.length;

```

react

```
const packed = items
  .filter(item => item.packed)
  .length;

function handleAddItem(title) {
  setItems([
    ...items,
    {
      # title,
      packed: false
    }
  ]);
}

function handleChangeItem(nextItem) {
  setItems(items.map(item => {
    if (item.id === nextItem.id) {
      return nextItem;
    } else {
      return item;
    }
  }));
}

function handleDeleteItem(itemId) {
  setItems(
    items.filter(item => item.id !== itemId)
  );
}

return (
  <>
    <AddItem
      onAddItem={handleAddItem}
    />
    <PackingList
      items={items}
      onChangeItem={handleChangeItem}
      onDeleteItem={handleDeleteItem}
    />
    <hr />
    <b>{packed} out of {total} packed!</b>
  </>
);
}
```

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

export default function AddItem({ onAddItem }) {
  const [title, setTitle] = useState('');
  return (
    <>
      <input
        placeholder="Add item"
        value={title}
        onChange={e => setTitle(e.target.value)}
      />
      <button onClick={() => {
        setTitle('');
        onAddItem(title);
      }}>Add</button>
    </>
  );
}
```

```

        </>
    )
}

import { useState } from 'react';

export default function PackingList({
  items,
  onChangeItem,
  onDeleteItem
}) {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>
          <label>
            <input
              type="checkbox"
              checked={item.packed}
              onChange={e => {
                onChangeItem({
                  ...item,
                  packed: e.target.checked
                });
              }}
            />
            {' '}
            {item.title}
          </label>
          <button onClick={() => onDeleteItem(item.id)}>
            Delete
          </button>
        </li>
      ))}
    </ul>
  );
}

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

Notice how the event handlers are only concerned with calling `setItems` after this change. The item counts are now calculated during the next render from `items`, so they are always up-to-date.

Fix the disappearing selection [{/fix-the-disappearing-selection/}](#)

There is a list of `letters` in state. When you hover or focus a particular letter, it gets highlighted. The currently highlighted letter is stored in the `highlightedLetter` state variable. You can "star" and "unstar" individual letters, which updates the `letters` array in state.

This code works, but there is a minor UI glitch. When you press "Star" or "Unstar", the highlighting disappears for a moment. However, it reappears as soon as you move your pointer or switch to another letter with keyboard. Why is this happening? Fix it so that the highlighting doesn't disappear after the button click.

react

```
import { useState } from 'react';
import { initialLetters } from './data.js';
import Letter from './Letter.js';

export default function MailClient() {
  const [letters, setLetters] = useState(initialLetters);
  const [highlightedLetter, setHighlightedLetter] = useState(null);

  function handleHover(letter) {
    setHighlightedLetter(letter);
  }

  function handleStar(starred) {
    setLetters(letters.map(letter => {
      if (letter.id === starred.id) {
        return {
          ...letter,
          isStarred: !letter.isStarred
        };
      } else {
        return letter;
      }
    }));
  }

  return (
    <>
      <h2>Inbox</h2>
      <ul>
        {letters.map(letter => (
          <Letter
            key={letter.id}
            letter={letter}
            isHighlighted={
              letter === highlightedLetter
            }
            onHover={handleHover}
            onToggleStar={handleStar}
          />
        ))}
      </ul>
    </>
  );
}
```

```
export default function Letter({
  letter,
  isHighlighted,
  onHover,
  onToggleStar,
}) {
  return (
    <li
      className={
        isHighlighted ? 'highlighted' : ''
      }
      onFocus={() => {
        onHover(letter);
      }}
      onPointerMove={() => {
        onHover(letter);
      }}
    >
```

```

    >   <button onClick={() => {
        onToggleStar(letter);
      }}>
        {letter.isStarred ? 'Unstar' : 'Star'}
      </button>
    {letter.subject}
  </li>
)
}

export const initialLetters = [
  {
    subject: 'Ready for adventure?',
    isStarred: true,
  },
  {
    subject: 'Time to check in!',
    isStarred: false,
  },
  {
    subject: 'Festival Begins in Just SEVEN Days!',
    isStarred: false,
  }
];

button { margin: 5px; }
li { border-radius: 5px; }
.highlighted { background: #d2eaff; }

```

The problem is that you're holding the letter object in `highlightedLetter`. But you're also holding the same information in the `letters` array. So your state has duplication! When you update the `letters` array after the button click, you create a new letter object which is different from `highlightedLetter`. This is why `highlightedLetter === letter` check becomes `false`, and the highlight disappears. It reappears the next time you call `setHighlightedLetter` when the pointer moves.

To fix the issue, remove the duplication from state. Instead of storing *the letter itself* in two places, store the `highlightedId` instead. Then you can check `isHighlighted` for each letter with `letter.id === highlightedId`, which will work even if the `letter` object has changed since the last render.

```

import { useState } from 'react';
import { initialLetters } from './data.js';
import Letter from './Letter.js';

export default function MailClient() {
  const [letters, setLetters] = useState(initialLetters);
  const [highlightedId, setHighlightedId] = useState(null);

  function handleHover(letterId) {
    setHighlightedId(letterId);
  }

  function handleStar(starredId) {
    setLetters(letters.map(letter => {
      if (letter.id === starredId) {
        return {
          ...letter,
          isStarred: !letter.isStarred
        };
      } else {
        return letter;
      }
    }));
  }
}

```

```

        });
    }

    return (
      <>
      <h2>Inbox</h2>
      <ul>
        {letters.map(letter => (
          <Letter
            key={letter.id}
            letter={letter}
            isHighlighted={
              letter.id === highlightedId
            }
            onHover={handleHover}
            onToggleStar={handleStar}
          />
        )));
      </ul>
    );
  }
}

```

```

export default function Letter({
  letter,
  isHighlighted,
  onHover,
  onToggleStar,
}) {
  return (
    <li
      className={
        isHighlighted ? 'highlighted' : ''
      }
      onFocus={() => {
        onHover(letter.id);
      }}
      onPointerMove={() => {
        onHover(letter.id);
      }}
    >
      <button onClick={() => {
        onToggleStar(letter.id);
      }}>
        {letter.isStarred ? 'Unstar' : 'Star'}
      </button>
      {letter.subject}
    </li>
  );
}

```

```

export const initialLetters = [
  {
    subject: 'Ready for adventure?',
    isStarred: true,
  },
  {
    subject: 'Time to check in!',
    isStarred: false,
  },
  {
    subject: 'Festival Begins in Just SEVEN Days!',
    isStarred: false,
  },
];

```

```
button { margin: 5px; }
li { border-radius: 5px; }
.highlighted { background: #d2eaff; }
```

Implement multiple selection {/implement-multiple-selection/}

In this example, each `Letter` has an `isSelected` prop and an `onToggle` handler that marks it as selected. This works, but the state is stored as a `selectedId` (either `null` or an ID), so only one letter can get selected at any given time.

Change the state structure to support multiple selection. (How would you structure it? Think about this before writing the code.) Each checkbox should become independent from the others. Clicking a selected letter should uncheck it. Finally, the footer should show the correct number of the selected items.

Instead of a single selected ID, you might want to hold an array or a `Set` of selected IDs in state.

```
import { useState } from 'react';
import { letters } from './data.js';
import Letter from './Letter.js';

export default function MailClient() {
  const [selectedId, setSelectedId] = useState(null);

  // TODO: allow multiple selection
  const selectedCount = 1;

  function handleToggle(toggledId) {
    // TODO: allow multiple selection
    setSelectedId(toggledId);
  }

  return (
    <>
      <h2>Inbox</h2>
      <ul>
        {letters.map(letter => (
          <Letter
            key={letter.id}
            letter={letter}
            isSelected={
              // TODO: allow multiple selection
              letter.id === selectedId
            }
            onToggle={handleToggle}
          />
        ))}
      <hr />
      <p>
        <b>
          You selected {selectedCount} letters
        </b>
      </p>
    </ul>
  </>
);
}
```

```
export default function Letter({  
  letter,  
  onToggle,  
  isSelected,  
}) {  
  return (  
    <li className={  
      isSelected ? 'selected' : ''  
    }>  
      <label>  
        <input  
          type="checkbox"  
          checked={isSelected}  
          onChange={() => {  
            onToggle(letter.id);  
          }}  
        />  
        {letter.subject}  
      </label>  
    </li>  
  )  
}
```

```
export const letters = [  
  {  
    subject: 'Ready for adventure?',  
    isStarred: true,  
  }, {  
    subject: 'Time to check in!',  
    isStarred: false,  
  }, {  
    subject: 'Festival Begins in Just SEVEN Days!',  
    isStarred: false,  
  }];
```

```
input { margin: 5px; }  
li { border-radius: 5px; }  
label { width: 100%; padding: 5px; display: inline-block; }  
.selected { background: #d2eaff; }
```

Instead of a single `selectedId`, keep a `selectedIds` array in state. For example, if you select the first and the last letter, it would contain `[0, 2]`. When nothing is selected, it would be an empty `[]` array:

```
import { useState } from 'react';  
import { letters } from './data.js';  
import Letter from './Letter.js';  
  
export default function MailClient() {  
  const [selectedIds, setSelectedIds] = useState([]);  
  
  const selectedCount = selectedIds.length;  
  
  function handleToggle(toggledId) {  
    // Was it previously selected?  
    if (selectedIds.includes(toggledId)) {  
      // Then remove this ID from the array.  
      setSelectedIds(selectedIds.filter(id =>  
        id !== toggledId  
      ));  
    } else {  
      // Otherwise, add this ID to the array.  
      setSelectedIds([...selectedIds, toggledId]);  
    }  
  }  
  return (  
    <ul>  
      {letters.map((letter) => (  
        <li key={letter.id}>  
          <Label checked={selectedIds.includes(letter.id)}  
            onClick={handleToggle} />  
          {letter.subject}  
        </li>  
      ))}  
    </ul>  
  );  
}
```

```

    setSelectedIds([
      ...selectedIds,
      toggledId
    ]);
  }

  return (
    <>
    <h2>Inbox</h2>
    <ul>
      {letters.map(letter => (
        <Letter
          key={letter.id}
          letter={letter}
          isSelected={selectedIds.includes(letter.id)}
        >
        onToggle={handleToggle}
      />
    ))}
    <hr />
    <p>
      <b>
        You selected {selectedCount} letters
      </b>
    </p>
    </ul>
  />
);
}

```

```

export default function Letter({
  letter,
  onToggle,
  isSelected,
}) {
  return (
    <li className={
      isSelected ? 'selected' : ''
    }>
      <label>
        <input
          type="checkbox"
          checked={isSelected}
          onChange={() => {
            onToggle(letter.id);
          }}
        />
        {letter.subject}
      </label>
    </li>
  )
}

```

```

export const letters = [
  {
    subject: 'Ready for adventure?',
    isStarred: true,
  },
  {
    subject: 'Time to check in!',
    isStarred: false,
  },
]

```

```
    subject: 'Festival Begins in Just SEVEN Days!',  
    isStarred: false,  
};
```

```
input { margin: 5px; }  
li { border-radius: 5px; }  
label { width: 100%; padding: 5px; display: inline-block; }  
.selected { background: #d2eaff; }
```

One minor downside of using an array is that for each item, you're calling `selectedIds.includes(letter.id)` to check whether it's selected. If the array is very large, this can become a performance problem because array search with `includes()` takes linear time, and you're doing this search for each individual item.

To fix this, you can hold a `Set` in state instead, which provides a fast `has()` operation:

```
import { useState } from 'react';  
import { letters } from './data.js';  
import Letter from './Letter.js';  
  
export default function MailClient() {  
  const [selectedIds, setSelectedIds] = useState(  
    new Set()  
  );  
  
  const selectedCount = selectedIds.size;  
  
  function handleToggle(toggledId) {  
    // Create a copy (to avoid mutation).  
    const nextIds = new Set(selectedIds);  
    if (nextIds.has(toggledId)) {  
      nextIds.delete(toggledId);  
    } else {  
      nextIds.add(toggledId);  
    }  
    setSelectedIds(nextIds);  
  }  
  
  return (  
    <>  
    <h2>Inbox</h2>  
    <ul>  
      {letters.map(letter => (  
        <Letter  
          key={letter.id}  
          letter={letter}  
          isSelected={  
            selectedIds.has(letter.id)  
          }  
          onToggle={handleToggle}  
        />  
      ))}  
    <hr />  
    <p>  
      <b>  
        You selected {selectedCount} letters  
      </b>  
    </p>  
  </ul>
```

```

        </>
    );
}

export default function Letter({
  letter,
  onToggle,
  isSelected,
}) {
  return (
    <li className={
      isSelected ? 'selected' : ''
    }>
      <label>
        <input
          type="checkbox"
          checked={isSelected}
          onChange={() => {
            onToggle(letter.id);
          }}
        />
        {letter.subject}
      </label>
    </li>
  )
}

```

```

export const letters = [
  {
    subject: 'Ready for adventure?',
    isStarred: true,
  },
  {
    subject: 'Time to check in!',
    isStarred: false,
  },
  {
    subject: 'Festival Begins in Just SEVEN Days!',
    isStarred: false,
  }];

```

```

input { margin: 5px; }
li { border-radius: 5px; }
label { width: 100%; padding: 5px; display: inline-block; }
.selected { background: #d2eaff; }

```

Now each item does a `selectedIds.has(letter.id)` check, which is very fast.

Keep in mind that you `should not mutate objects in state`, and that includes Sets, too. This is why the `handleToggle` function creates a *copy* of the Set first, and then updates that copy.

Conditional Rendering

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

- 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 `{ /conditionally-returning-jsx/ }`

Let's say you have a `PackingList` component rendering several `Item`s, which can be marked as packed or not:

```
function 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>
  );
}
```

Notice that some of the `Item` components have their `isPacked` prop set to `true` instead of `false`. You want to add a checkmark () to packed items if `isPacked={true}`.

You can write this as an `if / else` statement like so:

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

```
function Item({ name, isPacked }) {
  if (isPacked) {
    return <li className="item">{name} ✓</li>;
  }
  return <li className="item">{name}</li>;
}

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

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` *{/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.

```
function 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>
  );
}
```

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 {/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} ✓</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} ✓</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 (`? :`) {/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} ✓</li>;
}
return <li className="item">{name}</li>;
```

You can write this:

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

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

Are these two examples fully equivalent? {/are-these-two-examples-fully-equivalent/}

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:

```
function Item({ name, isPacked }) {
  return (
    <li className="item">
      {isPacked ? (
        <del>
          {name + ' ✓'}
        </del>
      ) : (
        name
      )}
    </li>
  );
}

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

```

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

```

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 (`&&`) {/logical-and-operator-/}

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

```

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

```

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

Here it is in action:

```

function Item({ name, isPacked }) {
  return (
    <li className="item">
      {name} {isPacked && '✓'}
    </li>
  );
}

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

```

```

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

```

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.

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 {/conditionally-assigning-jsx-to-a-variable/}

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

```
let itemContent = name;
```

Use an `if` statement to reassign a JSX expression to `itemContent` if `isPacked` is `true`:

```

if (isPacked) {
  itemContent = name + " ✓";
}

```

Curly braces open the "window into JavaScript". Embed the variable with curly braces in the returned JSX tree, nesting the previously calculated expression inside of JSX:

```

<li className="item">
  {itemContent}
</li>

```

This style is the most verbose, but it's also the most flexible. Here it is in action:

```

function Item({ name, isPacked }) {
  let itemContent = name;
  if (isPacked) {
    itemContent = name + " ✓";
  }
}

```

```

    }
    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>
  );
}

```

Like before, this works not only for text, but for arbitrary JSX too:

```

function Item({ name, isPacked }) {
  let itemContent = name;
  if (isPacked) {
    itemContent = (
      <del>
        {name + " ✓"}
      </del>
    );
  }
  return (
    <li className="item">
      {itemContent}
    </li>
  );
}

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

```

```

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

```

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.

- 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`.

Show an icon for incomplete items with `? :` [{/show-an-icon-for-incomplete-items-with--/}](#)

Use the conditional operator (`cond ? a : b`) to render a `✗` if `isPacked` isn't `true`.

```

function Item({ name, isPacked }) {
  return (
    <li className="item">
      {name} {isPacked && '✓'}
    </li>
  );
}

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

```

```

function Item({ name, isPacked }) {
  return (
    <li className="item">
      {name} {isPacked ? '✓' : '✗'}
    </li>
  );
}

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

```

Show the item importance with `&&` `{/show-the-item-importance-with-/}`

In this example, each `Item` receives a numerical `importance` prop. Use the `&&` operator to render "*(Importance: X)*" in italics, but only for items that have non-zero importance. Your item list should end up looking like this:

- Space suit (*Importance: 9*)
- Helmet with a golden leaf
- Photo of Tam (*Importance: 6*)

Don't forget to add a space between the two labels!

```

function Item({ name, importance }) {
  return (
    <li className="item">
      {name}
    </li>
  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          importance={9}
          name="Space suit"
        />
      </ul>
    </section>
  );
}

```

```

    />
  <Item
    importance={0}
    name="Helmet with a golden leaf"
  />
  <Item
    importance={6}
    name="Photo of Tam"
  />
</ul>
</section>
);
}

```

This should do the trick:

```

function Item({ name, importance }) {
  return (
    <li className="item">
      {name}
      {importance > 0 && ' '}
      {importance > 0 &&
        <i>(Importance: {importance})</i>
      }
    </li>
  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          importance={9}
          name="Space suit"
        />
        <Item
          importance={0}
          name="Helmet with a golden leaf"
        />
        <Item
          importance={6}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}

```

Note that you must write `importance > 0 && ...` rather than `importance && ...` so that if the `importance` is `0`, `0` isn't rendered as the result!

In this solution, two separate conditions are used to insert a space between then name and the importance label. Alternatively, you could use a fragment with a leading space: `importance > 0 && <> <i>...</i>` `</>` or add a space immediately inside the `<i>: importance > 0 && <i> ...</i>`.

Refactor a series of `? :` to `if` and variables [{/refactor-a-series-of-to-if-and-variables/}](#)

This `Drink` component uses a series of `? :` conditions to show different information depending on whether the `name` prop is `"tea"` or `"coffee"`. The problem is that the information about each drink is spread across multiple conditions. Refactor this code to use a single `if` statement instead of three `? :` conditions.

```
function Drink({ name }) {
  return (
    <section>
      <h1>{name}</h1>
      <dl>
        <dt>Part of plant</dt>
        <dd>{name === 'tea' ? 'leaf' : 'bean'}</dd>
        <dt>Caffeine content</dt>
        <dd>{name === 'tea' ? '15-70 mg/cup' : '80-185 mg/cup'}</dd>
        <dt>Age</dt>
        <dd>{name === 'tea' ? '4,000+ years' : '1,000+ years'}</dd>
      </dl>
    </section>
  );
}

export default function DrinkList() {
  return (
    <div>
      <Drink name="tea" />
      <Drink name="coffee" />
    </div>
  );
}
```

Once you've refactored the code to use `if`, do you have further ideas on how to simplify it?

There are multiple ways you could go about this, but here is one starting point:

```
function Drink({ name }) {
  let part, caffeine, age;
  if (name === 'tea') {
    part = 'leaf';
    caffeine = '15-70 mg/cup';
    age = '4,000+ years';
  } else if (name === 'coffee') {
    part = 'bean';
    caffeine = '80-185 mg/cup';
    age = '1,000+ years';
  }
  return (
    <section>
      <h1>{name}</h1>
      <dl>
        <dt>Part of plant</dt>
        <dd>{part}</dd>
        <dt>Caffeine content</dt>
        <dd>{caffeine}</dd>
        <dt>Age</dt>
        <dd>{age}</dd>
      </dl>
    </section>
  );
}
```

```

    );
}

export default function DrinkList() {
  return (
    <div>
      <Drink name="tea" />
      <Drink name="coffee" />
    </div>
  );
}

```

Here the information about each drink is grouped together instead of being spread across multiple conditions. This makes it easier to add more drinks in the future.

Another solution would be to remove the condition altogether by moving the information into objects:

```

const drinks = {
  tea: {
    part: 'leaf',
    caffeine: '15-70 mg/cup',
    age: '4,000+ years'
  },
  coffee: {
    part: 'bean',
    caffeine: '80-185 mg/cup',
    age: '1,000+ years'
  }
};

function Drink({ name }) {
  const info = drinks[name];
  return (
    <section>
      <h1>{name}</h1>
      <dl>
        <dt>Part of plant</dt>
        <dd>{info.part}</dd>
        <dt>Caffeine content</dt>
        <dd>{info.caffeine}</dd>
        <dt>Age</dt>
        <dd>{info.age}</dd>
      </dl>
    </section>
  );
}

export default function DrinkList() {
  return (
    <div>
      <Drink name="tea" />
      <Drink name="coffee" />
    </div>
  );
}

```

[Go to TOC](#)

Describing the UI

React is a JavaScript library for rendering user interfaces (UI). UI is built from small units like buttons, text, and images. React lets you combine them into reusable, nestable *components*. From web sites to phone apps, everything on the screen can be broken down into components. In this chapter, you'll learn to create, customize, and conditionally display React components.

- [How to write your first React component](#)
- [When and how to create multi-component files](#)
- [How to add markup to JavaScript with JSX](#)
- [How to use curly braces with JSX to access JavaScript functionality from your components](#)
- [How to configure components with props](#)
- [How to conditionally render components](#)
- [How to render multiple components at a time](#)
- [How to avoid confusing bugs by keeping components pure](#)

Your first component `{/your-first-component/}`

React applications are built from isolated pieces of UI called *components*. A React component is a JavaScript function that you can sprinkle with markup. Components can be as small as a button, or as large as an entire page. Here is a `Gallery` component rendering three `Profile` components:

```
function Profile() {
  return (
    
  );
}

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

```
img { margin: 0 10px 10px 0; height: 90px; }
```

Read [Your First Component](#) to learn how to declare and use React components.

Importing and exporting components {/importing-and-exporting-components/}

You can declare many components in one file, but large files can get difficult to navigate. To solve this, you can *export* a component into its own file, and then *import* that component from another file:

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

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

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

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

```
export default function Profile() {
  return (
    
  );
}
```

```
img { margin: 0 10px 10px 0; }
```

Read [Importing and Exporting Components](#) to learn how to split components into their own files.

Writing markup with JSX {/writing-markup-with-jsx/}

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

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

```
export 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 spectrum technology
</ul>
};

}
```

```
img { height: 90px; }
```

If you have existing HTML like this, you can fix it using a [converter](#):

```
export 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 spectrum technology</li>
      </ul>
    </>
  );
}


```

```
img { height: 90px; }
```

Read [Writing Markup with JSX](#) to learn how to write valid JSX.

JavaScript in JSX with curly braces {/javascript-in-jsx-with-curly-braces/}

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

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

```

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

}


```

```

body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }


```

Read [JavaScript in JSX with Curly Braces](#) to learn how to access JavaScript data from JSX.

Passing props to a component {/passing-props-to-a-component/}

React components use *props* to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, functions, and even JSX!

```

import { getImageUrl } from './utils.js'

export default function Profile() {
  return (
    <Card>
      <Avatar
        size={100}
        person={{
          name: 'Katsuko Saruhashi',
          imageId: 'Yfe0qp2'
        }}
      />
    </Card>
  );
}

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

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

```

```

    {children}
  </div>
);
}

```

```

export function getImageUrl(person, size = 's') {
  return (
    `https://i.imgur.com/${person.imageId +
      size +
      '.jpg'}`);
}

```

```

.card {
  width: fit-content;
  margin: 5px;
  padding: 5px;
  font-size: 20px;
  text-align: center;
  border: 1px solid #aaa;
  border-radius: 20px;
  background: #fff;
}

.avatar {
  margin: 20px;
  border-radius: 50%;
}

```

Read [Passing Props to a Component](#) to learn how to pass and read props.

Conditional rendering {/conditional-rendering/}

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

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

```

function Item({ name, isPacked }) {
  return (
    <li className="item">
      {name} {isPacked && '✓'}
    </li>
  );
}

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

```

```

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

```

Read [Conditional Rendering](#) to learn the different ways to render content conditionally.

Rendering lists {/rendering-lists/}

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

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

```

import { 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>
  );
}

export const people = [
  {
    name: 'Creola Katherine Johnson',
    profession: 'mathematician',
    accomplishment: 'spaceflight calculations',
    imageId: 'MK3eW3A'
  },
  {
    name: 'Mario José Molina-Pasquel Henríquez',
    profession: 'chemist',
    accomplishment: 'discovery of Arctic ozone hole',
    imageId: 'mynHUSa'
  },
  {
    name: 'Mohammad Abdus Salam',
    profession: 'physicist',
  }
];

```

```

    accomplishment: 'electromagnetism theory',
    imageId: 'bE7W1ji'
  }, {
    name: 'Percy Lavon Julian',
    profession: 'chemist',
    accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
    imageId: 'IOjWm71'
  }, {
    name: 'Subrahmanyan Chandrasekhar',
    profession: 'astrophysicist',
    accomplishment: 'white dwarf star mass calculations',
    imageId: 'lrWQx8l'
  }];
}

```

```

export function getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    '.jpg'
);
}

```

```

ul { list-style-type: none; padding: 0px 10px; }
li {
  margin-bottom: 10px;
  display: grid;
  grid-template-columns: 1fr 1fr;
  align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
h1 { font-size: 22px; }
h2 { font-size: 20px; }

```

Read [Rendering Lists](#) to learn how to render a list of components, and how to choose a key.

Keeping components pure {/keeping-components-pure/}

Some JavaScript functions are *pure*. A pure function:

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

By strictly only writing your components as pure functions, you can avoid an entire class of baffling bugs and unpredictable behavior as your codebase grows. Here is an example of an impure component:

```

let 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 />
  </>
};

}

```

You can make this component pure by passing a prop instead of modifying a preexisting variable:

```

function Cup({ guest }) {
  return <h2>Tea cup for guest #{guest}</h2>;
}

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

```

Read [Keeping Components Pure](#) to learn how to write components as pure, predictable functions.

What's next? *{/whats-next/}*

Head over to [Your First Component](#) to start reading this chapter page by page!

Or, if you're already familiar with these topics, why not read about [Adding Interactivity](#)?

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.

- What the most popular editors are
- How to format your code automatically

Your editor *{/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 *{/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 *{/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 *{/formatting/}*

The last thing you want to do when sharing your code with another contributor is get into an 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:

1. Launch VS Code
2. Use Quick Open (press Ctrl/Cmd+P)
3. Paste in `ext install esbenp.prettier-vscode`
4. Press Enter

Formatting on save *{/formatting-on-save/}*

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

1. In VS Code, press `CTRL/CMD + SHIFT + P`.
2. Type "settings"
3. Hit Enter
4. In the search bar, type "format on save"
5. 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.

Escape Hatches

Some of your components may need to control and synchronize with systems outside of React. For example, you might need to focus an input using the browser API, play and pause a video player implemented without React, or connect and listen to messages from a remote server. In this chapter, you'll learn the escape hatches that let you "step outside" React and connect to external systems. Most of your application logic and data flow should not rely on these features.

- How to "remember" information without re-rendering
- How to access DOM elements managed by React
- How to synchronize components with external systems
- How to remove unnecessary Effects from your components
- How an Effect's lifecycle is different from a component's
- How to prevent some values from re-triggering Effects
- How to make your Effect re-run less often
- How to share logic between components

Referencing values with refs {/referencing-values-with-refs/}

When you want a component to "remember" some information, but you don't want that information to trigger new renders, you can use a `ref`:

```
const ref = useRef(0);
```

Like state, refs are retained by React between re-renders. However, setting state re-renders a component. Changing a ref does not! You can access the current value of that ref through the `ref.current` property.

```
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>
  );
}
```

A ref is like a secret pocket of your component that React doesn't track. For example, you can use refs to store `timeout IDs`, `DOM elements`, and other objects that don't impact the component's rendering output.

Read [Referencing Values with Refs](#) to learn how to use refs to remember information.

Manipulating the DOM with refs {/manipulating-the-dom-with-refs/}

React automatically updates the DOM to match your render output, so your components won't often need to manipulate it. However, sometimes you might need access to the DOM elements managed by React—for example, to focus a node, scroll to it, or measure its size and position. There is no built-in way to do those things in React, so you will need a ref to the DOM node. For example, clicking the button will focus the input using a ref:

```
import { 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>
    </>
  );
}
```

Read [Manipulating the DOM with Refs](#) to learn how to access DOM elements managed by React.

Synchronizing with Effects {/synchronizing-with-effects/}

Some components need to synchronize with external systems. For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. Unlike event handlers, which let you handle particular events, *Effects* let you run some code after rendering. Use them to synchronize your component with a system outside of React.

Press Play/Pause a few times and see how the video player stays synchronized to the `isPlaying` prop value:

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

function VideoPlayer({ src,.isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    }
  });
}
```

```

    } else {
      ref.current.pause();
    },
  [isPlaying]);

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  return (
    <>
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
      />
    </>
  );
}

```

```

button { display: block; margin-bottom: 20px; }
video { width: 250px; }

```

Many Effects also "clean up" after themselves. For example, an Effect that sets up a connection to a chat server should return a *cleanup function* that tells React how to disconnect your component from that server:

```

import { 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>;
}

```

```

export function createConnection() {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting...');
    },
    disconnect() {
      console.log('❌ Disconnected.');
    }
  };
}

```

```

input { display: block; margin-bottom: 20px; }

```

In development, React will immediately run and clean up your Effect one extra time. This is why you see "✅ Connecting..." printed twice. This ensures that you don't forget to implement the cleanup function.

Read [Synchronizing with Effects](#) to learn how to synchronize components with external systems.

You Might Not Need An Effect [{/you-might-not-need-an-effect/}](#)

Effects are an escape hatch from the React paradigm. They let you "step outside" of React and synchronize your components with some external system. If there is no external system involved (for example, if you want to update a component's state when some props or state change), you shouldn't need an Effect. Removing unnecessary Effects will make your code easier to follow, faster to run, and less error-prone.

There are two common cases in which you don't need Effects:

- **You don't need Effects to transform data for rendering.**
- **You don't need Effects to handle user events.**

For example, you don't need an Effect to adjust some state based on other state:

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

  const [fullName, setFullName] = useState('');
  useEffect(() => {
    setFullName(firstName + ' ' + lastName);
  }, [firstName, lastName]);
  // ...
}
```

Instead, calculate as much as you can while rendering:

```
function Form() {
  const [firstName, setFirstName] = useState('Taylor');
  const [lastName, setLastName] = useState('Swift');
  // ✅ Good: calculated during rendering
  const fullName = firstName + ' ' + lastName;
  // ...
}
```

However, you *do* need Effects to synchronize with external systems.

Read [You Might Not Need an Effect](#) to learn how to remove unnecessary Effects.

Lifecycle of reactive effects [{/lifecycle-of-reactive-effects/}](#)

Effects have a different lifecycle from components. Components may mount, update, or unmount. An Effect can only do two things: to start synchronizing something, and later to stop synchronizing it. This cycle can happen multiple times if your Effect depends on props and state that change over time.

This Effect depends on the value of the `roomId` prop. Props are *reactive values*, which means they can change on a re-render. Notice that the Effect *re-synchronizes* (and re-connects to the server) if `roomId` changes:

```
import { 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} />
    </>
  );
}

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

      disconnect() {
        console.log('❌ Disconnected from "' + roomId + '" room at ' + serverUrl);
      }
    };
  };
}

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

React provides a linter rule to check that you've specified your Effect's dependencies correctly. If you forget to specify `roomId` in the list of dependencies in the above example, the linter will find that bug automatically.

Read [Lifecycle of Reactive Events](#) to learn how an Effect's lifecycle is different from a component's.

Separating events from Effects {/separating-events-from-effects/}

This section describes an **experimental API that has not yet been released** in a stable version of React.

Event handlers only re-run when you perform the same interaction again. Unlike event handlers, Effects re-synchronize if any of the values they read, like props or state, are different than during last render. Sometimes, you want a mix of both behaviors: an Effect that re-runs in response to some values but not others.

All code inside Effects is *reactive*. It will run again if some reactive value it reads has changed due to a re-render. For example, this Effect will re-connect to the chat if either `roomId` or `theme` have changed:

```
{
  "dependencies": {
    "react": "latest",
    "react-dom": "latest",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```
import { 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:<input type="text" value={roomId} onChange={(e) => setRoomId(e.target.value)} />
      </label>
      <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'}
    />
  </>
);
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  let connectedCallback;
  let timeout;
  return {
    connect() {
      timeout = setTimeout(() => {
        if (connectedCallback) {
          connectedCallback();
        }
      }, 100);
    },
    on(event, callback) {
      if (connectedCallback) {
        throw Error('Cannot add the handler twice.');
      }
      if (event !== 'connected') {
        throw Error('Only "connected" event is supported.');
      }
      connectedCallback = callback;
    },
    disconnect() {
      clearTimeout(timeout);
    }
  };
}

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {

```

```

        background: theme === 'dark' ? 'black' : 'white',
        color: theme === 'dark' ? 'white' : 'black',
    },
}).showToast();
}

```

```
label { display: block; margin-top: 10px; }
```

This is not ideal. You want to re-connect to the chat only if the `roomId` has changed. Switching the `theme` shouldn't re-connect to the chat! Move the code reading `theme` out of your Effect into an *Effect Event*:

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```

import { 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:{' '}
      </label>
      <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'}
  />
</>
);
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  let connectedCallback;
  let timeout;
  return {
    connect() {
      timeout = setTimeout(() => {
        if (connectedCallback) {
          connectedCallback();
        }
      }, 100);
    },
    on(event, callback) {
      if (connectedCallback) {
        throw Error('Cannot add the handler twice.');
      }
      if (event !== 'connected') {
        throw Error('Only "connected" event is supported.');
      }
      connectedCallback = callback;
    },
    disconnect() {
      clearTimeout(timeout);
    }
  };
}

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    }
  }).showToast();
}

```

```

        },
      }).showToast();
    }

  label { display: block; margin-top: 10px; }

```

Code inside Effect Events isn't reactive, so changing the `theme` no longer makes your Effect re-connect.

Read [Separating Events from Effects](#) to learn how to prevent some values from re-triggering Effects.

Removing Effect dependencies {/removing-effect-dependencies/}

When you write an Effect, the linter will verify that you've included every reactive value (like props and state) that the Effect reads in the list of your Effect's dependencies. This ensures that your Effect remains synchronized with the latest props and state of your component. Unnecessary dependencies may cause your Effect to run too often, or even create an infinite loop. The way you remove them depends on the case.

For example, this Effect depends on the `options` object which gets re-created every time you edit the input:

```

import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const options = {
    serverUrl,
    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} />
  </>
);
}

export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

You don't want the chat to re-connect every time you start typing a message in that chat. To fix this problem, move creation of the `options` object inside the Effect so that the Effect only depends on the `roomId` string:

```

import { 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} />
  </>
);
}

```

```

export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

Notice that you didn't start by editing the dependency list to remove the `options` dependency. That would be wrong. Instead, you changed the surrounding code so that the dependency became *unnecessary*. Think of the dependency list as a list of all the reactive values used by your Effect's code. You don't intentionally choose what to put on that list. The list describes your code. To change the dependency list, change the code.

Read [Removing Effect Dependencies](#) to learn how to make your Effect re-run less often.

Reusing logic with custom Hooks *{/reusing-logic-with-custom-hooks/}*

React comes with built-in Hooks like `useState`, `useContext`, and `useEffect`. Sometimes, you'll wish that there was a Hook for some more specific purpose: for example, to fetch data, to keep track of whether the user is online, or to connect to a chat room. To do this, you can create your own Hooks for your application's needs.

In this example, the `usePointerPosition` custom Hook tracks the cursor position, while `useDelayedValue` custom Hook returns a value that's "lagging behind" the value you passed by a certain number of milliseconds. Move the cursor over the sandbox preview area to see a moving trail of dots following the cursor:

```
import { usePointerPosition } from './usePointerPosition.js';
import { useDelayedValue } from './useDelayedValue.js';

export default function Canvas() {
  const pos1 = usePointerPosition();
  const pos2 = useDelayedValue(pos1, 100);
  const pos3 = useDelayedValue(pos2, 200);
  const pos4 = useDelayedValue(pos3, 100);
  const pos5 = useDelayedValue(pos4, 50);
  return (
    <>
      <Dot position={pos1} opacity={1} />
      <Dot position={pos2} opacity={0.8} />
      <Dot position={pos3} opacity={0.6} />
      <Dot position={pos4} opacity={0.4} />
      <Dot position={pos5} opacity={0.2} />
    </>
  );
}

function Dot({ position, opacity }) {
  return (
    <div style={{
      position: 'absolute',
      backgroundColor: 'pink',
      borderRadius: '50%',
      opacity,
      transform: `translate(${position.x}px, ${position.y}px)`,
      pointerEvents: 'none',
      left: -20,
      top: -20,
      width: 40,
      height: 40,
    }} />
  );
}
```

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

export function usePointerPosition() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  useEffect(() => {
    function handleMove(e) {
      setPosition({ x: e.clientX, y: e.clientY });
    }
    window.addEventListener('pointermove', handleMove);
    return () => window.removeEventListener('pointermove', handleMove);
  }, []);
  return position;
}
```

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

export function useDelayedValue(value, delay) {
  const [delayedValue, setDelayedValue] = useState(value);
```

```
useEffect(() => {
  setTimeout(() => {
    setDelayedValue(value);
  }, delay);
}, [value, delay]);

return delayedValue;
}
```

```
body { min-height: 300px; }
```

You can create custom Hooks, compose them together, pass data between them, and reuse them between components. As your app grows, you will write fewer Effects by hand because you'll be able to reuse custom Hooks you already wrote. There are also many excellent custom Hooks maintained by the React community.

Read [Reusing Logic with Custom Hooks](#) to learn how to share logic between components.

What's next? *{/whats-next/}*

Head over to [Referencing Values with Refs](#) to start reading this chapter page by page!

Extracting State Logic into a Reducer

Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called a *reducer*.

- 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 {/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:

```
import { 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,
      {
        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 = [
];

import { useState } from 'react';

export default function AddTask({onAddTask}) {
  const [text, setText] = useState('');
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <button
        onClick={() => {
          setText('');
          onAddTask(text);
        }}
      >
        Add
      </button>
    </>
  );
}

import { useState } from 'react';

export default function TaskList({tasks, onChangeTask, onDeleteTask}) {
  return (
    <ul>
      {tasks.map((task) => (
        <li key={task.id}>
          <Task task={task} onChange={onChangeTask} onDelete={onDeleteTask} />
        </li>
      ))}
    </ul>
  );
}

function Task({task, onChange, onDelete}) {
  const [isEditing, setIsEditing] = useState(false);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={(e) => {
            onChange({
              ...task,
              text: e.target.value
            });
            setIsEditing(false);
          }}
        />
    );
  } else {
    taskContent = (
      <p>{task.text}</p>
      <button
        onClick={() => {
          onChange({
            ...task,
            id: nextId
          });
          setIsEditing(true);
        }}
      >
        Edit
      </button>
    );
  }
  return (
    <li key={task.id}>
      {taskContent}
      <button
        onClick={() => onDelete(task.id)}
      >
        Delete
      </button>
    </li>
  );
}

```

react

```
        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) => {
    onChange({
        ...task,
        done: e.target.checked,
    });
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>Delete</button>
</label>
);
}
```

```
button {
  margin: 5px;
}
li {
  list-style-type: none;
}
ul,
li {
  margin: 0;
  padding: 0;
}
```

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:

1. **Move** from setting state to dispatching actions.
2. **Write** a reducer function.
3. **Use** the reducer from your component.

Step 1: Move from setting state to dispatching actions {/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,
    {
      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',
    text: text,
  });
}

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

function handleDeleteTask(taskId) {
```

```
dispatch({
  type: 'deleted',
});
```

The object you pass to `dispatch` is called an "action":

```
function handleDeleteTask(taskId) {
  dispatch(
    // "action" object:
    {
      type: 'deleted',
    }
  );
}
```

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.)

An 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 {/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:

1. Declare the current state (`tasks`) as the first argument.
2. Declare the `action` object as the second argument.
3. 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,
      action.task
    ];
  }
  return tasks;
}
```

```

    ...tasks,
  [
    {
      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.

The 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,
        {
          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 `{}` curly braces so that variables declared inside of different `case`s 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.

Why are reducers called this way? [{/why-are-reducers-called-this-way/}](#)

Although 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:

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

let initialState = [];
let actions = [
];

let finalState = actions.reduce(tasksReducer, initialState);

const output = document.getElementById('output');
output.textContent = JSON.stringify(finalState, null, 2);
```

```
export default function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [
        ...tasks,
        {
          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);
    }
  }
}

```

```
<pre id="output"></pre>
```

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 {/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:

1. A reducer function
2. An initial state

And it returns:

1. A stateful value
2. 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:

```

import { 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',

```

```

        text: text,
    });

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

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

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,
                {
                    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 = [
];

```

```

import { useState } from 'react';

export default function AddTask({onAddTask}) {
  const [text, setText] = useState('');
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <button
        onClick={() => {
          setText('');
          onAddTask(text);
        }}
      >Add
      </button>
    </>
  );
}

```

```

import { useState } from 'react';

export default function TaskList({tasks, onChangeTask, onDeleteTask}) {
  return (
    <ul>
      {tasks.map((task) => (
        <li key={task.id}>
          <Task task={task} onChange={onChangeTask} onDelete={onDeleteTask} />
        </li>
      ))}
    </ul>
  );
}

function Task({task, onChange, onDelete}) {
  const [isEditing, setIsEditing] = useState(false);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={(e) => {
            onChange({
              ...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) => {
        onChange({
          ...task,
          done: e.target.checked,
        });
      }}
    />
    {taskContent}
    <button onClick={() => onDelete(task.id)}>Delete</button>
  </label>
);
}

button {
  margin: 5px;
}
li {
  list-style-type: none;
}
ul,
li {
  margin: 0;
  padding: 0;
}

```

If you want, you can even move the reducer to a different file:

```

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',
      text: text,
    });
  }

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

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

  return (

```

```

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

let nextId = 3;
const initialTasks = [
];

```

```

export default function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [
        ...tasks,
        {
          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);
    }
  }
}

```

```

import { useState } from 'react';

export default function AddTask({onAddTask}) {
  const [text, setText] = useState('');
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <button
        onClick={() => {
          setText('');
          onAddTask(text);
        }}
      >Add
    </>
  );
}

```

```

        </button>
      </>
    );
}

import { useState } from 'react';

export default function TaskList({tasks, onChangeTask, onDeleteTask}) {
  return (
    <ul>
      {tasks.map((task) => (
        <li key={task.id}>
          <Task task={task} onChange={onChangeTask} onDelete={onDeleteTask} />
        </li>
      ))}
    </ul>
  );
}

function Task({task, onChange, onDelete}) {
  const [isEditing, setIsEditing] = useState(false);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={(e) => {
            onChange({
              ...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) => {
          onChange({
            ...task,
            done: e.target.checked,
          });
        }}
      />
      {taskContent}
      <button onClick={() => onDelete(task.id)}>Delete</button>
    </label>
  );
}

```

```

button {
  margin: 5px;
}
li {
  list-style-type: none;
}
ul,
li {
  margin: 0;
  padding: 0;
}

```

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` {/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 {/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 {/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:

```
import { useImmerReducer } from 'use-immer';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

function tasksReducer(draft, action) {
  switch (action.type) {
    case 'added': {
      draft.push({
        text: action.text,
        done: false,
      });
      break;
    }
    case 'changed': {
      const index = draft.findIndex((t) => t.id === action.task.id);
      draft[index] = action.task;
      break;
    }
    case 'deleted': {
      return draft.filter((t) => t.id !== action.id);
    }
    default: {
      throw Error(`Unknown action: ${action.type}`);
    }
  }
}

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

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

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

```

        type: 'changed',
        task: task,
    });

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

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

let nextId = 3;
const initialTasks = [
];

```

```

import { useState } from 'react';

export default function AddTask({onAddTask}) {
    const [text, setText] = useState('');
    return (
        <>
        <input
            placeholder="Add task"
            value={text}
            onChange={(e) => setText(e.target.value)}
        />
        <button
            onClick={() => {
                setText('');
                onAddTask(text);
            }}
        >
            Add
        </button>
        </>
    );
}

```

```

import { useState } from 'react';

export default function TaskList({tasks, onChangeTask, onDeleteTask}) {
    return (
        <ul>
            {tasks.map((task) => (
                <li key={task.id}>
                    <Task task={task} onChange={onChangeTask} onDelete={onDeleteTask} />
                </li>
            ))}
        </ul>
    );
}

```

```

}

function Task({task, onChange, onDelete}) {
  const [isEditing, setIsEditing] = useState(false);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={(e) => {
            onChange({
              ...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) => {
          onChange({
            ...task,
            done: e.target.checked,
          });
        }}
      />
      {taskContent}
      <button onClick={() => onDelete(task.id)}>Delete</button>
    </label>
  );
}

```

```

button {
  margin: 5px;
}
li {
  list-style-type: none;
}
ul,
li {
  margin: 0;
  padding: 0;
}

```

```
{
  "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"
  }
}

```

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.

- To convert from `useState` to `useReducer` :
 1. Dispatch actions from event handlers.
 2. Write a reducer function that returns the next state for a given state and action.
 3. 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.

Dispatch actions from event handlers *(/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 `// TODO`s 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`.

The `dispatch` function is already available in both of these components because it was passed as a prop. So you need to call `dispatch` with the corresponding action object.

To check the action object shape, you can look at the reducer and see which `action` fields it expects to see. For example, the `changed_selection` case in the reducer looks like this:

```

case 'changed_selection': {
  return {
    ...state,
    selectedId: action.contactId
  };
}

```

This means that your action object should have a `type: 'changed_selection'`. You also see the `action.contactId` being used, so you need to include a `contactId` property into your action.

```

import { 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 = [
];

```

```

export const initialState = {
  selectedId: 0,
  message: 'Hello',
};

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection':
      return {
        ...state,
        selectedId: action.contactId,
        message: '',
      };
    case 'edited_message':
      return {
        ...state,
        message: action.message,
      };
    default:
      throw Error(`Unknown action: ${action.type}`);
  }
}

```

```

export default function ContactList({contacts, selectedId, dispatch}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map((contact) => (

```

```

        <li key={contact.id}>
          <button
            onClick={() => {
              // TODO: dispatch changed_selection
            }}
            {selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
          </button>
        </li>
      )))
    </ul>
  </section>
);
}

```

```

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          // TODO: dispatch edited_message
          // (Read the input value from e.target.value)
        }}
      />
      <br />
      <button>Send to {contact.email}</button>
    </section>
  );
}

```

```

.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
ul,
li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}

```

From the reducer code, you can infer that actions need to look like this:

```

// When the user presses "Alice"
dispatch({
  type: 'changed_selection',
  contactId: 1,
});

```

```
// When user types "Hello!"
dispatch({
  type: 'edited_message',
  message: 'Hello!',
});
```

Here is the example updated to dispatch the corresponding messages:

```
import { 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 = [
];

export const initialState = {
  selectedId: 0,
  message: 'Hello',
};

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection': {
      return {
        ...state,
        selectedId: action.contactId,
        message: '',
      };
    }
    case 'edited_message': {
      return {
        ...state,
        message: action.message,
      };
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}
```

```

        }
    }
}
```

```

export default function ContactList({contacts, selectedId, dispatch}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map((contact) => (
          <li key={contact.id}>
            <button
              onClick={() => {
                dispatch({
                  type: 'changed_selection',
                  contactId: contact.id,
                });
              }}
            >
              {selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
            </button>
          </li>
        ))}
      </ul>
    </section>
  );
}
```

```

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          dispatch({
            type: 'edited_message',
            message: e.target.value,
          });
        }}
      />
      <br />
      <button>Send to {contact.email}</button>
    </section>
  );
}
```

```

.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
ul,
li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
}
```

```

    margin-right: 10px;
}
textarea {
  height: 150px;
}

```

Clear the input on sending a message {/clear-the-input-on-sending-a-message/}

Currently, pressing "Send" doesn't do anything. Add an event handler to the "Send" button that will:

1. Show an `alert` with the recipient's email and the message.
2. Clear the message input.

```

import { 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 = [
];

```

```

export const initialState = {
  selectedId: 0,
  message: 'Hello',
};

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection': {
      return {
        ...state,
        selectedId: action.contactId,
        message: '',
      };
    }
    case 'edited_message': {
      return {
        ...state,
        message: action.message,
      };
    }
  }
}

```

```

        };
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

```

```

export default function ContactList({contacts, selectedId, dispatch}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map((contact) => (
          <li key={contact.id}>
            <button
              onClick={() => {
                dispatch({
                  type: 'changed_selection',
                  contactId: contact.id,
                });
              }}
            >{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}</button>
          </li>
        ))}
      </ul>
    </section>
  );
}

```

```

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          dispatch({
            type: 'edited_message',
            message: e.target.value,
          });
        }}
      />
      <br />
      <button>Send to {contact.email}</button>
    </section>
  );
}

```

```

.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
ul,
li {
  list-style: none;
  margin: 0;
  padding: 0;
}

```

```

}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}

```

There are a couple of ways you could do it in the "Send" button event handler. One approach is to show an alert and then dispatch an `edited_message` action with an empty `message`:

```

import { 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 = [
];

export const initialState = {
  selectedId: 0,
  message: 'Hello',
};

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection':
      return {
        ...state,
        selectedId: action.contactId,
        message: '',
      };
    case 'edited_message':
      return {
        ...state,
        message: action.message,
      };
  }
}

```

```

        }
      default: {
        throw Error('Unknown action: ' + action.type);
      }
    }
  }
}

```

```

export default function ContactList({contacts, selectedId, dispatch}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map((contact) => (
          <li key={contact.id}>
            <button
              onClick={() => {
                dispatch({
                  type: 'changed_selection',
                  contactId: contact.id,
                });
              }}
            >{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}</button>
          </li>
        ))}
      </ul>
    </section>
  );
}

```

```

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          dispatch({
            type: 'edited_message',
            message: e.target.value,
          });
        }}
      />
      <br />
      <button
        onClick={() => {
          alert(`Sending "${message}" to ${contact.email}`);
          dispatch({
            type: 'edited_message',
            message: '',
          });
        }}
      >Send to {contact.email}</button>
    </section>
  );
}

```

```
.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
ul,
li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}
```

This works and clears the input when you hit "Send".

However, *from the user's perspective*, sending a message is a different action than editing the field. To reflect that, you could instead create a *new* action called `sent_message`, and handle it separately in the reducer:

```
import { 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 = [
];

export const initialState = {
  selectedId: 0,
  message: 'Hello',
};
```

```

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection': {
      return {
        ...state,
        selectedId: action.contactId,
        message: '',
      };
    }
    case 'edited_message': {
      return {
        ...state,
        message: action.message,
      };
    }
    case 'sent_message': {
      return {
        ...state,
        message: '',
      };
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

```

```

export default function ContactList({contacts, selectedId, dispatch}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map((contact) => (
          <li key={contact.id}>
            <button
              onClick={() => {
                dispatch({
                  type: 'changed_selection',
                  contactId: contact.id,
                });
              }}
            >{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}</button>
          </li>
        ))}
      </ul>
    </section>
  );
}

```

```

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          dispatch({
            type: 'edited_message',
            message: e.target.value,
          });
        }}
      ></textarea>
    </section>
  );
}

```

```

        }
      />
      <br />
      <button
        onClick={() => {
          alert(`Sending "${message}" to ${contact.email}`);
          dispatch({
            type: 'sent_message',
          });
        }}
      > Send to {contact.email}
      </button>
    </section>
  );
}

```

```

.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
ul,
li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}

```

The resulting behavior is the same. But keep in mind that action types should ideally describe "what the user did" rather than "how you want the state to change". This makes it easier to later add more features.

With either solution, it's important that you **don't** place the `alert` inside a reducer. The reducer should be a pure function--it should only calculate the next state. It should not "do" anything, including displaying messages to the user. That should happen in the event handler. (To help catch mistakes like this, React will call your reducers multiple times in Strict Mode. This is why, if you put an alert in a reducer, it fires twice.)

Restore input values when switching between tabs {/restore-input-values-when-switching-between-tabs/}

In this example, switching between different recipients always clears the text input:

```

case 'changed_selection': {
  return {
    ...state,
    selectedId: action.contactId,
    message: '' // Clears the input
  };
}

```

This is because you don't want to share a single message draft between several recipients. But it would be better if your app "remembered" a draft for each contact separately, restoring them when you switch contacts.

Your task is to change the way the state is structured so that you remember a separate message draft *per contact*. You would need to make a few changes to the reducer, the initial state, and the components.

You can structure your state like this:

```
export const initialState = {
  selectedId: 0,
  messages: {
    0: 'Hello, Taylor', // Draft for contactId = 0
    1: 'Hello, Alice', // Draft for contactId = 1
  },
};
```

The `[key]: value` **computed property** syntax can help you update the `messages` object:

```
{
  ...state.messages,
  [id]: message
}

import { 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 = [
];

export const initialState = {
  selectedId: 0,
  message: 'Hello',
};
```

```

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection': {
      return {
        ...state,
        selectedId: action.contactId,
        message: '',
      };
    }
    case 'edited_message': {
      return {
        ...state,
        message: action.message,
      };
    }
    case 'sent_message': {
      return {
        ...state,
        message: '',
      };
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

```

```

export default function ContactList({contacts, selectedId, dispatch}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map((contact) => (
          <li key={contact.id}>
            <button
              onClick={() => {
                dispatch({
                  type: 'changed_selection',
                  contactId: contact.id,
                });
              }}
            >{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}</button>
          </li>
        )));
      </ul>
    </section>
  );
}

```

```

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          dispatch({
            type: 'edited_message',
            message: e.target.value,
          });
        }}
      ></textarea>
    </section>
  );
}

```

```

        }
      />
      <br />
      <button
        onClick={() => {
          alert(`Sending "${message}" to ${contact.email}`);
          dispatch({
            type: 'sent_message',
          });
        }}
      > Send to {contact.email}
      </button>
    </section>
  );
}

```

```

.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
ul,
li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}

```

You'll need to update the reducer to store and update a separate message draft per contact:

```

// When the input is edited
case 'edited_message': {
  return {
    // Keep other state like selection
    ...state,
    messages: {
      // Keep messages for other contacts
      ...state.messages,
      // But change the selected contact's message
      [state.selectedId]: action.message
    }
  };
}

```

You would also update the `Messenger` component to read the message for the currently selected contact:

```

const message = state.messages[state.selectedId];

```

Here is the complete solution:

```

import { 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.messages[state.selectedId];
  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 = [
];

```

```

export const initialState = {
  selectedId: 0,
  messages: {
    0: 'Hello, Taylor',
    1: 'Hello, Alice',
    2: 'Hello, Bob',
  },
};

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection': {
      return {
        ...state,
        selectedId: action.contactId,
      };
    }
    case 'edited_message': {
      return {
        ...state,
        messages: {
          ...state.messages,
          [state.selectedId]: action.message,
        },
      };
    }
    case 'sent_message': {
      return {
        ...state,
        messages: {
          ...state.messages,
          [state.selectedId]: '',
        },
      };
    }
  }
}

```

```

        };
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

```

```

export default function ContactList({contacts, selectedId, dispatch}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map((contact) => (
          <li key={contact.id}>
            <button
              onClick={() => {
                dispatch({
                  type: 'changed_selection',
                  contactId: contact.id,
                });
              }}
            >{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}</button>
          </li>
        ))}
      </ul>
    </section>
  );
}

```

```

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          dispatch({
            type: 'edited_message',
            message: e.target.value,
          });
        }}
      />
      <br />
      <button
        onClick={() => {
          alert(`Sending "${message}" to ${contact.email}`);
          dispatch({
            type: 'sent_message',
          });
        }}
      >Send to {contact.email}</button>
    </section>
  );
}

```

```
.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
ul,
li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}
```

Notably, you didn't need to change any of the event handlers to implement this different behavior. Without a reducer, you would have to change every event handler that updates the state.

Implement `useReducer` from scratch [`/implement-usereducer-from-scratch/`](#)

In the earlier examples, you imported the `useReducer` Hook from React. This time, you will implement *the `useReducer` Hook itself!* Here is a stub to get you started. It shouldn't take more than 10 lines of code.

To test your changes, try typing into the input or select a contact.

Here is a more detailed sketch of the implementation:

```
export function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    // ???
  }

  return [state, dispatch];
}
```

Recall that a reducer function takes two arguments--the current state and the action object--and it returns the next state. What should your `dispatch` implementation do with it?

```
import { useReducer } from './MyReact.js';
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.messages[state.selectedId];
  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 = [
];

export const initialState = {
  selectedId: 0,
  messages: {
    0: 'Hello, Taylor',
    1: 'Hello, Alice',
    2: 'Hello, Bob',
  },
};

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection': {
      return {
        ...state,
        selectedId: action.contactId,
      };
    }
    case 'edited_message': {
      return {
        ...state,
        messages: {
          ...state.messages,
          [state.selectedId]: action.message,
        },
      };
    }
    case 'sent_message': {
      return {
        ...state,
        messages: {
          ...state.messages,
          [state.selectedId]: '',
        },
      };
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}

import { useState } from 'react';

export function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);
}

```

```
// ???

return [state, dispatch];
}

export default function ContactList({contacts, selectedId, dispatch}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map((contact) => (
          <li key={contact.id}>
            <button
              onClick={() => {
                dispatch({
                  type: 'changed_selection',
                  contactId: contact.id,
                });
              }}
            >
              {selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
            </button>
          </li>
        ))}
      </ul>
    </section>
  );
}

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          dispatch({
            type: 'edited_message',
            message: e.target.value,
          });
        }}
      />
      <br />
      <button
        onClick={() => {
          alert(`Sending "${message}" to ${contact.email}`);
          dispatch({
            type: 'sent_message',
          });
        }}
      >
        Send to {contact.email}
      </button>
    </section>
  );
}

.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
```

```

ul {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}

```

Dispatching an action calls a reducer with the current state and the action, and stores the result as the next state. This is what it looks like in code:

```

import { useReducer } from './MyReact.js';
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.messages[state.selectedId];
  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 = [
];

export const initialState = {
  selectedId: 0,
  messages: {
    0: 'Hello, Taylor',
    1: 'Hello, Alice',
    2: 'Hello, Bob',
  },
};

export function messengerReducer(state, action) {
  switch (action.type) {
    case 'changed_selection': {
      return {
        ...state,
      };
    }
  }
}

```

```

        selectedId: action.contactId,
    );
}

case 'edited_message': {
    return {
        ...state,
        messages: {
            ...state.messages,
            [state.selectedId]: action.message,
        },
    };
}

case 'sent_message': {
    return {
        ...state,
        messages: {
            ...state.messages,
            [state.selectedId]: '',
        },
    };
}

default: {
    throw Error('Unknown action: ' + action.type);
}
}
}

```

```

import { useState } from 'react';

export function useReducer(reducer, initialState) {
    const [state, setState] = useState(initialState);

    function dispatch(action) {
        const nextState = reducer(state, action);
        setState(nextState);
    }

    return [state, dispatch];
}

```

```

export default function ContactList({contacts, selectedId, dispatch}) {
    return (
        <section className="contact-list">
            <ul>
                {contacts.map((contact) => (
                    <li key={contact.id}>
                        <button
                            onClick={() => {
                                dispatch({
                                    type: 'changed_selection',
                                    contactId: contact.id,
                                });
                            }}>
                            {selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
                        </button>
                    </li>
                )));
            </ul>
        </section>
    );
}

```

```

import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
  return (
    <section className="chat">
      <textarea
        value={message}
        placeholder={'Chat to ' + contact.name}
        onChange={(e) => {
          dispatch({
            type: 'edited_message',
            message: e.target.value,
          });
        }}
      />
      <br />
      <button
        onClick={() => {
          alert(`Sending "${message}" to ${contact.email}`);
          dispatch({
            type: 'sent_message',
          });
        }}
      >Send to {contact.email}</button>
    </section>
  );
}

```

```

.chat,
.contact-list {
  float: left;
  margin-bottom: 20px;
}
ul,
li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}

```

Though it doesn't matter in most cases, a slightly more accurate implementation looks like this:

```

function dispatch(action) {
  setState((s) => reducer(s, action));
}

```

This is because the dispatched actions are queued until the next render, [similar to the updater functions](#).

[Go to TOC](#)

Importing and Exporting Components

The 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.

- 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 *{/the-root-component-file/}*

In [Your First Component](#), you made a `Profile` component and a `Gallery` component that renders it:

```
function Profile() {
  return (
    
  );
}

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

```
img { margin: 0 10px 10px 0; height: 90px; }
```

These currently live in a **root component file**, named `App.js` in this example. In [Create React App](#), your app lives in `src/App.js`. Depending on your setup, your root component could be in another file, though. If you use a framework with file-based routing, such as Next.js, your root component will be different for every page.

Exporting and importing a component {/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:

1. **Make** a new JS file to put the components in.
2. **Export** your function component from that file (using either `default` or `named` exports).
3. **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`:

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

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

function Profile() {
  return (
    
  );
}

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

img { margin: 0 10px 10px 0; height: 90px; }
```

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

1. `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**.
2. `App.js`:
 - Imports `Gallery` as a **default import** from `Gallery.js`.

- Exports the root `App` component as a **default export**.

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.

Default vs named exports *{/default-vs-named-exports/}*

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.**

Default and named exports

Default and named exports

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	<code>export default function Button() {}</code>	<code>import Button from './Button.js';</code>
Named	<code>export function Button() {}</code>	<code>import { Button } from './Button.js';</code>

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 *{/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!**

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:

```
import Gallery from './Gallery.js';
import { Profile } from './Gallery.js';

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

export function Profile() {
  return (
    
  );
}

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

img { margin: 0 10px 10px 0; height: 90px; }
```

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**.

On 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

Split the components further *(/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:

Syntax	Export statement	Import statement
Default	<code>export default function Button() {}</code>	<code>import Button from './Button.js';</code>
Named	<code>export function Button() {}</code>	<code>import { Button } from './Button.js';</code>

Don't forget to import your components where they are called. Doesn't `Gallery` use `Profile`, too?

```
import Gallery from './Gallery.js';
import { Profile } from './Gallery.js';

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

```
// Move me to Profile.js!
export function Profile() {
  return (
    
  );
}
```

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

img { margin: 0 10px 10px 0; height: 90px; }
```

After you get it working with one kind of exports, make it work with the other kind.

This is the solution with named exports:

```
import Gallery from './Gallery.js';
import { Profile } from './Profile.js';

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

import { Profile } from './Profile.js';

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

export function Profile() {
  return (
    
  );
}

img { margin: 0 10px 10px 0; height: 90px; }
```

This is the solution with default exports:

```
import Gallery from './Gallery.js';
import Profile from './Profile.js';

export default function App() {
```

react

```
return (
  <div>
    <Profile />
    <Gallery />
  </div>
);
```

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

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

```
export default function Profile() {
  return (
    
  );
}
```

```
img { margin: 0 10px 10px 0; height: 90px; }
```

[Go to TOC](#)

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.

- [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 *{/try-react/}*

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

```
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 *{/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 *{/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 *{/add-react-to-an-existing-project/}*

If you want to try using React in your existing app or a website, [add React to an existing project](#).

Next steps *{/next-steps/}*

Head to the [Quick Start](#) guide for a tour of the most important React concepts you will encounter every day.

react

[Go to TOC](#)

JavaScript in JSX with Curly Braces

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

- 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 *{/passing-strings-with-quotes/}*

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

```
export default function Avatar() {
  return (
    
  );
}

.avatar { border-radius: 50%; height: 90px; }
```

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 }**:

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

.avatar { border-radius: 50%; height: 90px; }
```

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 {/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>`:

```
export 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()`:

```
const 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 {/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

{ /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:

```
export 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>
  );
}
```

```
body { padding: 0; margin: 0 }  
ul { padding: 20px 20px 20px 40px; margin: 0; }
```

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!

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

{ /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:

```

const 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>
  );
}

```

```

body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }

```

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.

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.

Fix the mistake `{/fix-the-mistake/}`

This code crashes with an error saying `Objects are not valid as a React child:`

```

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>
  );
}

```

```

body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }

```

Can you find the problem?

Look for what's inside the curly braces. Are we putting the right thing there?

This is happening because this example renders *an object itself* into the markup rather than a string: `<h1>{person}'s Todos</h1>` is trying to render the entire `person` object! Including raw objects as text content throws an error because React doesn't know how you want to display them.

To fix it, replace `<h1>{person}'s Todos</h1>` with `<h1>{person.name}'s Todos</h1>`:

```

const 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>

```

react

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

```
body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }
```

Extract information into an object {/extract-information-into-an-object/}

Extract the image URL into the `person` object.

```
const 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>
  );
}
```

```
body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }
```

Move the image URL into a property called `person.imageUrl` and read it from the `` tag using the curly braces:

```
const person = {
  name: 'Gregorio Y. Zara',
  imageUrl: "https://i.imgur.com/7vQD0fPs.jpg",
  theme: {
    backgroundColor: 'black',
    color: 'pink'
  }
};

export default function TodoList() {
  return (
    <div style={person.theme}>
      <h1>{person.name}'s Todos</h1>
      <img
        className="avatar"
        src={person.imageUrl}
        alt="Gregorio Y. Zara"
      />
      <ul>
        <li>Improve the videophone</li>
        <li>Prepare aeronautics lectures</li>
        <li>Work on the alcohol-fuelled engine</li>
      </ul>
    </div>
  );
}
```

```

<div style={person.theme}>
  <h1>{person.name}'s Todos</h1>
  <img
    className="avatar"
    src={person.imageUrl}
    alt="Gregorio Y. Zara"
  />
  <ul>
    <li>Improve the videophone</li>
    <li>Prepare aeronautics lectures</li>
    <li>Work on the alcohol-fuelled engine</li>
  </ul>
</div>
);
}

```

```

body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }

```

Write an expression inside JSX curly braces {/write-an-expression-inside-jsx-curly-braces/}

In the object below, the full image URL is split into four parts: base URL, `imageId`, `imageSize`, and file extension.

We want the image URL to combine these attributes together: base URL (always `'https://i.imgur.com/'`), `imageId` (`'7vQD0fP'`), `imageSize` (`'s'`), and file extension (always `'.jpg'`). However, something is wrong with how the `` tag specifies its `src`.

Can you fix it?

```

const baseUrl = 'https://i.imgur.com/';
const person = {
  name: 'Gregorio Y. Zara',
  imageId: '7vQD0fP',
  imageSize: 's',
  theme: {
    backgroundColor: 'black',
    color: 'pink'
  }
};

export default function TodoList() {
  return (
    <div style={person.theme}>
      <h1>{person.name}'s Todos</h1>
      <img
        className="avatar"
        src={`${baseUrl}${person.imageId}${person.imageSize}.jpg`}
        alt={person.name}
      />
      <ul>
        <li>Improve the videophone</li>
        <li>Prepare aeronautics lectures</li>
        <li>Work on the alcohol-fuelled engine</li>
      </ul>
    </div>
  );
}

```

```

    </div>
  );
}

```

```

body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; }

```

To check that your fix worked, try changing the value of `imageSize` to `'b'`. The image should resize after your edit.

You can write it as `src={baseUrl + person.imageId + person.imageSize + '.jpg'}`.

1. `{` opens the JavaScript expression
2. `baseUrl + person.imageId + person.imageSize + '.jpg'` produces the correct URL string
3. `}` closes the JavaScript expression

```

const baseUrl = 'https://i.imgur.com/';
const person = {
  name: 'Gregorio Y. Zara',
  imageId: '7vQD0fP',
  imageSize: 's',
  theme: {
    backgroundColor: 'black',
    color: 'pink'
  }
};

export default function TodoList() {
  return (
    <div style={person.theme}>
      <h1>{person.name}'s Todos</h1>
      <img
        className="avatar"
        src={baseUrl + person.imageId + person.imageSize + '.jpg'}
        alt={person.name}
      />
      <ul>
        <li>Improve the videophone</li>
        <li>Prepare aeronautics lectures</li>
        <li>Work on the alcohol-fuelled engine</li>
      </ul>
    </div>
  );
}

```

```

body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; }

```

You can also move this expression into a separate function like `getImageUrl` below:

```

import { getImageUrl } from './utils.js'

const person = {
  name: 'Gregorio Y. Zara',
  imageId: '7vQD0fP',
  imageSize: 's',
}

```

```

    theme: {
      backgroundColor: 'black',
      color: 'pink'
    };

  export default function TodoList() {
    return (
      <div style={person.theme}>
        <h1>{person.name}'s Todos</h1>
        <img
          className="avatar"
          src={getImageUrl(person)}
          alt={person.name}
        />
        <ul>
          <li>Improve the videophone</li>
          <li>Prepare aeronautics lectures</li>
          <li>Work on the alcohol-fuelled engine</li>
        </ul>
      </div>
    );
  }

```

```

export function getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    person.imageSize +
    '.jpg'
  );
}

```

```

body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; }

```

Variables and functions can help you keep the markup simple!

Keeping Components Pure

Some 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.

- 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 {/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: `Unexpected text node: ' = 2'`.

If `Unexpected text node: ' = 2'` then `Unexpected text node: ' = 4'`. Always.

If `Unexpected text node: ' = 3'` then `Unexpected text node: ' = 6'`. Always.

If `Unexpected text node: ' = 3'`, `y` won't sometimes be `Unexpected text node: '9'` or `Unexpected text node: '-1'` or `Unexpected text node: '2.5'` depending on the time of day or the state of the stock market.

If `Unexpected text node: ' = 2'` and `Unexpected text node: ' = 3'`, `y` will *always* be `Unexpected text node: '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:

```

function 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>
  );
}

```

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`.

Side Effects: (un)intended consequences {/side-effects-unintended-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:

```

let 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 />
    </>
  );
}

```

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 `Unexpected text node: ' = 2`, now even if `Unexpected text node: ' = 2`, we cannot trust that `Unexpected text node: ' = 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:

```
function 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 `Unexpected text node: ' = 2` before or after `Unexpected text node: ' = 5`: 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!

Detecting impure calculations with StrictMode [{/detecting-impure-calculations-with-strict-mode/}](#)

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 `Unexpected text node: ' = 2` 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 { /local-mutation-your-components-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:

```
function 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 { /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!

Why does React care about purity? {/why-does-react-care-about-purity/}

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.

- 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.

Fix a broken clock {/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!

Rendering is a *calculation*, it shouldn't try to "do" things. Can you express the same idea differently?

```
export 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>
  );
}
```

```

import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return time;
}

export default function App() {
  const time = useTime();
  return (
    <Clock time={time} />
  );
}

```

```

body > * {
  width: 100%;
  height: 100%;
}
.day {
  background: #fff;
  color: #222;
}
.night {
  background: #222;
  color: #fff;
}

```

You can fix this component by calculating the `className` and including it in the render output:

```

export default function Clock({ time }) {
  let hours = time.getHours();
  let className;
  if (hours >= 0 && hours <= 6) {
    className = 'night';
  } else {
    className = 'day';
  }
  return (
    <h1 className={className}>
      {time.toLocaleTimeString()}
    </h1>
  );
}

```

```

import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return time;
}

```

```

    }, []);
    return time;
}

export default function App() {
  const time = useTime();
  return (
    <Clock time={time} />
  );
}

body > * {
  width: 100%;
  height: 100%;
}
.day {
  background: #fff;
  color: #222;
}
.night {
  background: #222;
  color: #fff;
}

```

In this example, the side effect (modifying the DOM) was not necessary at all. You only needed to return JSX.

Fix a broken profile [{/fix-a-broken-profile/}](#)

Two `Profile` components are rendered side by side with different data. Press "Collapse" on the first profile, and then "Expand" it. You'll notice that both profiles now show the same person. This is a bug.

Find the cause of the bug and fix it.

The buggy code is in `Profile.js`. Make sure you read it all from top to bottom!

```

import Panel from './Panel.js';
import { getImageUrl } from './utils.js';

let currentPerson;

export default function Profile({ person }) {
  currentPerson = person;
  return (
    <Panel>
      <Header />
      <Avatar />
    </Panel>
  )
}

function Header() {
  return <h1>{currentPerson.name}</h1>;
}

function Avatar() {
  return (
    <img
      className="avatar"

```

```

        src={getImageUrl(currentPerson)}
        alt={currentPerson.name}
        width={50}
        height={50}
      />
    );
}

import { useState } from 'react';

export default function Panel({ children }) {
  const [open, setOpen] = useState(true);
  return (
    <section className="panel">
      <button onClick={() => setOpen(!open)}>
        {open ? 'Collapse' : 'Expand'}
      </button>
      {open && children}
    </section>
  );
}

```

```

import Profile from './Profile.js';

export default function App() {
  return (
    <>
      <Profile person={{ imageId: 'lrWQx8l', name: 'Subrahmanyam Chandrasekhar', }} />
      <Profile person={{ imageId: 'MK3eW3A', name: 'Creola Katherine Johnson', }} />
    </>
  )
}

```

```

export function getImageUrl(person, size = 's') {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    size +
    '.jpg'
  );
}

```

```

.avatar { margin: 5px; border-radius: 50%; }
.panel {
  border: 1px solid #aaa;
  border-radius: 6px;
  margin-top: 20px;
  padding: 10px;
  width: 200px;
}
h1 { margin: 5px; font-size: 18px; }

```

The problem is that the `Profile` component writes to a preexisting variable called `currentPerson`, and the `Header` and `Avatar` components read from it. This makes *all three of them* impure and difficult to predict.

To fix the bug, remove the `currentPerson` variable. Instead, pass all information from `Profile` to `Header` and `Avatar` via props. You'll need to add a `person` prop to both components and pass it all the way down.

```
import Panel from './Panel.js';
import { getImageUrl } from './utils.js';

export default function Profile({ person }) {
  return (
    <Panel>
      <Header person={person} />
      <Avatar person={person} />
    </Panel>
  )
}

function Header({ person }) {
  return <h1>{person.name}</h1>;
}

function Avatar({ person }) {
  return (
    <img
      className="avatar"
      src={getImageUrl(person)}
      alt={person.name}
      width={50}
      height={50}
    />
  );
}
```

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

export default function Panel({ children }) {
  const [open, setOpen] = useState(true);
  return (
    <section className="panel">
      <button onClick={() => setOpen(!open)}>
        {open ? 'Collapse' : 'Expand'}
      </button>
      {open && children}
    </section>
  );
}
```

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

export default function App() {
  return (
    <>
      <Profile person={{
        imageUrl: 'lrWQx8l',
        name: 'Subrahmanyam Chandrasekhar',
      }}/>
    </>
  );
}
```

```

        }};
      <Profile person={{
        imageUrl: 'MK3eW3A',
        name: 'Creola Katherine Johnson',
      }} />
    </>
  );
}

```

```

export function getImageUrl(person, size = 's') {
  return (
    `https://i.imgur.com/${person.imageUrl +
    size +
    '.jpg'}`);
}

```

```

.avatar { margin: 5px; border-radius: 50%; }
.panel {
  border: 1px solid #aaa;
  border-radius: 6px;
  margin-top: 20px;
  padding: 10px;
  width: 200px;
}
h1 { margin: 5px; font-size: 18px; }

```

Remember that React does not guarantee that component functions will execute in any particular order, so you can't communicate between them by setting variables. All communication must happen through props.

Fix a broken story tray {/fix-a-broken-story-tray/}

The CEO of your company is asking you to add "stories" to your online clock app, and you can't say no. You've written a `StoryTray` component that accepts a list of `stories`, followed by a "Create Story" placeholder.

You implemented the "Create Story" placeholder by pushing one more fake story at the end of the `stories` array that you receive as a prop. But for some reason, "Create Story" appears more than once. Fix the issue.

```

export default function StoryTray({ stories }) {
  stories.push({
    label: 'Create Story'
  });

  return (
    <ul>
      {stories.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
    </ul>
  );
}

```

```

import { useState, useEffect } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [
];

export default function App() {
  let [stories, setStories] = useState([...initialStories])
  let time = useTime();

  // HACK: Prevent the memory from growing forever while you read docs.
  // We're breaking our own rules here.
  if (stories.length > 100) {
    stories.length = 100;
  }

  return (
    <div
      style={{
        width: '100%',
        height: '100%',
        textAlign: 'center',
      }}
    >
      <h2>It is {time.toLocaleTimeString()} now.</h2>
      <StoryTray stories={stories} />
    </div>
  );
}

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return time;
}

```

```

ul {
  margin: 0;
  list-style-type: none;
}

li {
  border: 1px solid #aaa;
  border-radius: 6px;
  float: left;
  margin: 5px;
  margin-bottom: 20px;
  padding: 5px;
  width: 70px;
  height: 100px;
}

```

```
{
  "hardReloadOnChange": true
}
```

Notice how whenever the clock updates, "Create Story" is added *twice*. This serves as a hint that we have a mutation during rendering--Strict Mode calls components twice to make these issues more noticeable.

`StoryTray` function is not pure. By calling `push` on the received `stories` array (a prop!), it is mutating an object that was created *before* `StoryTray` started rendering. This makes it buggy and very difficult to predict.

The simplest fix is to not touch the array at all, and render "Create Story" separately:

```
export default function StoryTray({ stories }) {
  return (
    <ul>
      {stories.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
      <li>Create Story</li>
    </ul>
  );
}
```

```
import { useState, useEffect } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [
];

export default function App() {
  let [stories, setStories] = useState([...initialStories])
  let time = useTime();

  // HACK: Prevent the memory from growing forever while you read docs.
  // We're breaking our own rules here.
  if (stories.length > 100) {
    stories.length = 100;
  }

  return (
    <div
      style={{
        width: '100%',
        height: '100%',
        textAlign: 'center',
      }}>
      <h2>It is {time.toLocaleTimeString()} now.</h2>
      <StoryTray stories={stories} />
    </div>
  );
}

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  });
}
```

```
}, []);
return time;
}
```

```
ul {
  margin: 0;
  list-style-type: none;
}

li {
  border: 1px solid #aaa;
  border-radius: 6px;
  float: left;
  margin: 5px;
  margin-bottom: 20px;
  padding: 5px;
  width: 70px;
  height: 100px;
}
```

Alternatively, you could create a *new* array (by copying the existing one) before you push an item into it:

```
export default function StoryTray({ stories }) {
  // Copy the array!
  let storiesToDisplay = stories.slice();

  // Does not affect the original array:
  storiesToDisplay.push({
    label: 'Create Story'
  });

  return (
    <ul>
      {storiesToDisplay.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
    </ul>
  );
}

import { useState, useEffect } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [];

export default function App() {
  let [stories, setStories] = useState([...initialStories])
  let time = useTime();

  // HACK: Prevent the memory from growing forever while you read docs.
  // We're breaking our own rules here.
  if (stories.length > 100) {
    stories.length = 100;
  }

  return (
    <div
      style={{
```

```

        width: '100%',
        height: '100%',
        textAlign: 'center',
    )}
> <h2>It is {time.toLocaleTimeString()} now.</h2>
<StoryTray stories={stories} />
</div>
);
}

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return time;
}

```

```

ul {
  margin: 0;
  list-style-type: none;
}

li {
  border: 1px solid #aaa;
  border-radius: 6px;
  float: left;
  margin: 5px;
  margin-bottom: 20px;
  padding: 5px;
  width: 70px;
  height: 100px;
}

```

This keeps your mutation local and your rendering function pure. However, you still need to be careful: for example, if you tried to change any of the array's existing items, you'd have to clone those items too.

It is useful to remember which operations on arrays mutate them, and which don't. For example, `push`, `pop`, `reverse`, and `sort` will mutate the original array, but `slice`, `filter`, and `map` will create a new one.

'Lifecycle of Reactive Effects'

Effects have a different lifecycle from components. Components may mount, update, or unmount. An Effect can only do two things: to start synchronizing something, and later to stop synchronizing it. This cycle can happen multiple times if your Effect depends on props and state that change over time. 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.

- 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 {/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.

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 {/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:

1. Stop synchronizing with the old roomId (disconnect from the "general" room)
2. 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 *{/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 { /thinking-from-the-effects-perspective/ }

Let's recap everything that's happened from the `ChatRoom` component's perspective:

1. `ChatRoom` mounted with `roomId` set to `"general"`
2. `ChatRoom` updated with `roomId` set to `"travel"`
3. `ChatRoom` updated with `roomId` set to `"music"`
4. `ChatRoom` unmounted

During each of these points in the component's lifecycle, your Effect did different things:

1. Your Effect connected to the `"general"` room
2. Your Effect disconnected from the `"general"` room and connected to the `"travel"` room
3. Your Effect disconnected from the `"travel"` room and connected to the `"music"` room
4. 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:

1. Your Effect connected to the "general" room (until it disconnected)
2. Your Effect connected to the "travel" room (until it disconnected)
3. 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 { /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:

```
import { 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>
    </>
  );
}
```

```

        </button>
      {show && <hr />}
      {show && <ChatRoom roomId={roomId} />}
    </>
  );
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

Notice that when the component mounts for the first time, you see three logs:

1. ✓ Connecting to "general" room at https://localhost:1234... (development-only)
2. ✗ Disconnected from "general" room at https://localhost:1234. (development-only)
3. ✓ 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 {/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:

1. You knew `roomId` is a prop, which means it can change over time.
2. You knew that your Effect reads `roomId` (so its logic depends on a value that may change later).
3. 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 {/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 {/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:

```

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');
  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} />
    );
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

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-an-effect-with-empty-dependencies-means/}

What happens if you move both `serverUrl` and `roomId` outside the component?

```

const serverUrl = 'https://localhost:1234';
const roomId = 'general';

function ChatRoom() {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, []); // ✓ All dependencies declared
}

```

Now your Effect's code does not use *any* reactive values, so its dependencies can be empty (`[]`).

Thinking from the component's perspective, the empty `[]` dependency array means this Effect connects to the chat room only when the component mounts, and disconnects only when the component unmounts. (Keep in mind that React would still [re-synchronize it an extra time](#) in development to stress-test your logic.)

```

import { 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 />}
    </>
  );
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('❌ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

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 {/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.

Can global or mutable values be dependencies? {/can-global-or-mutable-values-be-dependencies/}

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 {/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:

```

import { 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} />
    </>
  );
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

This may look like a React error, but really React is pointing out a bug in your code. Both `roomId` and `serverUrl` may change over time, but you're forgetting to re-synchronize your Effect when they change. You will remain connected to the initial `roomId` and `serverUrl` even after the user picks different values in the UI.

To fix the bug, follow the linter's suggestion to specify `roomId` and `serverUrl` as dependencies of your Effect:

```
function ChatRoom({ roomId }) { // roomId is reactive
  const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // serverUrl is reactive
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]); // ✅ All dependencies declared
  // ...
}
```

Try this fix in the sandbox above. Verify that the linter error is gone, and the chat re-connects when needed.

In 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 {/what-to-do-when-you-dont-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 reactive
const roomId = 'general'; // roomId is not reactive

function ChatRoom() {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, []); // ✅ All dependencies declared
  // ...
}
```

You can also move them *inside the Effect*. They aren't calculated during rendering, so they're not reactive:

```
function ChatRoom() {
  useEffect(() => {
    const serverUrl = 'https://localhost:1234'; // serverUrl is not reactive
    const roomId = 'general'; // roomId is not reactive
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, []); // ✅ All dependencies declared
  // ...
}
```

Effects are reactive blocks of code. They re-synchronize when the values you read inside of them change. Unlike event handlers, which only run once per interaction, Effects run whenever synchronization is necessary.

You can't "choose" your dependencies. Your dependencies must include every [reactive value](#) you read in the Effect. The linter enforces this. Sometimes this may lead to problems like infinite loops and to your Effect re-synchronizing too often. Don't fix these problems by suppressing the linter! Here's what to try instead:

- **Check that your Effect represents an independent synchronization process.** If your Effect doesn't synchronize anything, [it might be unnecessary](#). If it synchronizes several independent things, [split it up](#).
- **If you want to read the latest value of props or state without "reacting" to it and re-synchronizing the Effect,** you can split your Effect into a reactive part (which you'll keep in the Effect) and a non-reactive part (which you'll extract into something called an *Effect Event*). [Read about separating Events from Effects](#).
- **Avoid relying on objects and functions as dependencies.** If you create objects and functions during rendering and then read them from an Effect, they will be different on every render. This will cause your Effect to re-synchronize every time. [Read more about removing unnecessary dependencies from Effects](#).

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!

- 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.

Fix reconnecting on every keystroke `{/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.

You might need to add a dependency array for this Effect. What dependencies should be there?

```
import { 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} />
</>
);
};

export function createConnection(serverUrl, roomId) {
    // A real implementation would actually connect to the server
    return {
        connect() {
            console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
            '...');

        },
        disconnect() {
            console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
        }
    };
}

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

This Effect didn't have a dependency array at all, so it re-synchronized after every re-render. First, add a dependency array. Then, make sure that every reactive value used by the Effect is specified in the array. For example, `roomId` is reactive (because it's a prop), so it should be included in the array. This ensures that when the user selects a different room, the chat reconnects. On the other hand, `serverUrl` is defined outside the component. This is why it doesn't need to be in the array.

```

import { 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();
    }, [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} />
  </>
);
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

Switch synchronization on and off {/switch-synchronization-on-and-off/}

In this example, an Effect subscribes to the window `pointermove` event to move a pink dot on the screen. Try hovering over the preview area (or touching the screen if you're on a mobile device), and see how the pink dot follows your movement.

There is also a checkbox. Ticking the checkbox toggles the `canMove` state variable, but this state variable is not used anywhere in the code. Your task is to change the code so that when `canMove` is `false` (the checkbox is ticked off), the dot should stop moving. After you toggle the checkbox back on (and set `canMove` to `true`), the box should follow the movement again. In other words, whether the dot can move or not should stay synchronized to whether the checkbox is checked.

You can't declare an Effect conditionally. However, the code inside the Effect can use conditions!

```

import { useState, useEffect } from 'react';

export default function App() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const [canMove, setCanMove] = useState(true);

  useEffect(() => {

```

```

function handleMove(e) {
  setPosition({ x: e.clientX, y: e.clientY });
}
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
}, []);

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,
  }} />
</>
);
}

body {
  height: 200px;
}

```

One solution is to wrap the `setPosition` call into an `if (canMove) { ... }` condition:

```

import { useState, useEffect } from 'react';

export default function App() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const [canMove, setCanMove] = useState(true);

  useEffect(() => {
    function handleMove(e) {
      if (canMove) {
        setPosition({ x: e.clientX, y: e.clientY });
      }
    }
    window.addEventListener('pointermove', handleMove);
    return () => window.removeEventListener('pointermove', handleMove);
  }, [canMove]);

  return (
<>
  <label>
    <input type="checkbox"
      checked={canMove}
      onChange={e => setCanMove(e.target.checked)}>
  </label>

```

```

    />
    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,
}} />
</>
);
}

```

```

body {
  height: 200px;
}

```

Alternatively, you could wrap the event *subscription* logic into an `if (canMove) { ... }` condition:

```

import { useState, useEffect } from 'react';

export default function App() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const [canMove, setCanMove] = useState(true);

  useEffect(() => {
    function handleMove(e) {
      setPosition({ x: e.clientX, y: e.clientY });
    }
    if (canMove) {
      window.addEventListener('pointermove', handleMove);
      return () => window.removeEventListener('pointermove', handleMove);
    }
  }, [canMove]);

  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,
    }]>
</>
);
}

body {
  height: 200px;
}

```

In both of these cases, `canMove` is a reactive variable that you read inside the Effect. This is why it must be specified in the list of Effect dependencies. This ensures that the Effect re-synchronizes after every change to its value.

Investigate a stale value bug [{/investigate-a-stale-value-bug/}](#)

In this example, the pink dot should move when the checkbox is on, and should stop moving when the checkbox is off. The logic for this has already been implemented: the `handleMove` event handler checks the `canMove` state variable.

However, for some reason, the `canMove` state variable inside `handleMove` appears to be "stale": it's always `true`, even after you tick off the checkbox. How is this possible? Find the mistake in the code and fix it.

If you see a linter rule being suppressed, remove the suppression! That's where the mistakes usually are.

```

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(() => {
    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,
    }]}
);
}

body {
  height: 200px;
}

```

The problem with the original code was suppressing the dependency linter. If you remove the suppression, you'll see that this Effect depends 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. There are a few different ways to solve this bug, but you should always start by removing the linter suppression. Then change the code to fix the lint error.

You can change the Effect dependencies to `[handleMove]`, but since it's going to be a newly defined function for every render, you might as well remove dependencies array altogether. Then the Effect *will* re-synchronize after every re-render:

```

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(() => {
    window.addEventListener('pointermove', handleMove);
    return () => window.removeEventListener('pointermove', handleMove);
  });
}

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,
  }} />
</>
);
}

body {
  height: 200px;
}

```

This solution works, but it's not ideal. If you put `console.log('Resubscribing')` inside the Effect, you'll notice that it resubscribes after every re-render. Resubscribing is fast, but it would still be nice to avoid doing it so often.

A better fix would be to move the `handleMove` function *inside* the Effect. Then `handleMove` won't be a reactive value, and so your Effect won't depend on a function. Instead, it will need to depend on `canMove` which your code now reads from inside the Effect. This matches the behavior you wanted, since your Effect will now stay synchronized with the value of `canMove`:

```

import { useState, useEffect } from 'react';

export default function App() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const [canMove, setCanMove] = useState(true);

  useEffect(() => {
    function handleMove(e) {
      if (canMove) {
        setPosition({ x: e.clientX, y: e.clientY });
      }
    }

    window.addEventListener('pointermove', handleMove);
    return () => window.removeEventListener('pointermove', handleMove);
  }, [canMove]);

  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,
}} />
</>
);
}

body {
  height: 200px;
}

```

Try adding `console.log('Resubscribing')` inside the Effect body and notice that now it only resubscribes when you toggle the checkbox (`canMove` changes) or edit the code. This makes it better than the previous approach that always resubscribed.

You'll learn a more general approach to this type of problem in [Separating Events from Effects](#).

Fix a connection switch {/fix-a-connection-switch/}

In this example, the chat service in `chat.js` exposes two different APIs: `createEncryptedConnection` and `createUnencryptedConnection`. The root `App` component lets the user choose whether to use encryption or not, and then passes down the corresponding API method to the child `ChatRoom` component as the `createConnection` prop.

Notice that initially, the console logs say the connection is not encrypted. Try toggling the checkbox on: nothing will happen. However, if you change the selected room after that, then the chat will reconnect *and* enable encryption (as you'll see from the console messages). This is a bug. Fix the bug so that toggling the checkbox *also* causes the chat to reconnect.

Suppressing the linter is always suspicious. Could this be a bug?

```

import { useState } from 'react';
import ChatRoom from './ChatRoom.js';
import {
  createEncryptedConnection,
  createUnencryptedConnection,
} from './chat.js';

export default function App() {
  const [roomId, set roomId] = useState('general');

```

```

const [isEncrypted, setIsEncrypted] = 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={isEncrypted}
        onChange={e => setIsEncrypted(e.target.checked)}
      />
      Enable encryption
    </label>
    <hr />
    <ChatRoom
      roomId={roomId}
      createConnection={isEncrypted ?
        createEncryptedConnection :
        createUnencryptedConnection
      }
    />
  </>
);
}

```

```

import { useState, useEffect } from 'react';

export default function ChatRoom({ roomId, createConnection }) {
  useEffect(() => {
    const connection = createConnection(roomId);
    connection.connect();
    return () => connection.disconnect();
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [roomId]);

  return <h1>Welcome to the {roomId} room!</h1>;
}

```

```

export function createEncryptedConnection(roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ 🔒 Connecting to "' + roomId + '... (encrypted)');
    },
    disconnect() {
      console.log('✗ 🔒 Disconnected from "' + roomId + '" room (encrypted)');
    }
  };
}

export function createUnencryptedConnection(roomId) {
  // A real implementation would actually connect to the server
  return {

```

```

    connect() {
      console.log('✓ Connecting to "' + roomId + '... (unencrypted)');
    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room (unencrypted)');
    }
  };
}

```

```
label { display: block; margin-bottom: 10px; }
```

If you remove the linter suppression, you will see a lint error. The problem is that `createConnection` is a prop, so it's a reactive value. It can change over time! (And indeed, it should--when the user ticks the checkbox, the parent component passes a different value of the `createConnection` prop.) This is why it should be a dependency. Include it in the list to fix the bug:

```

import { useState } from 'react';
import ChatRoom from './ChatRoom.js';
import {
  createEncryptedConnection,
  createUnencryptedConnection,
} from './chat.js';

export default function App() {
  const [roomId, setRoomId] = useState('general');
  const [isEncrypted, setIsEncrypted] = 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={isEncrypted}
          onChange={e => setIsEncrypted(e.target.checked)}
        />
        Enable encryption
      </label>
    <hr />
    <ChatRoom
      roomId={roomId}
      createConnection={isEncrypted ?
        createEncryptedConnection :
        createUnencryptedConnection
      }
    />
  );
}

```

```

import { useState, useEffect } from 'react';

export default function ChatRoom({ roomId, createConnection }) {
  useEffect(() => {
    const connection = createConnection(roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, createConnection]);
}

return <h1>Welcome to the {roomId} room!</h1>;
}

export function createEncryptedConnection(roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ 🔒 Connecting to "' + roomId + '... (encrypted)');
    },
    disconnect() {
      console.log('❌ 🔒 Disconnected from "' + roomId + '" room (encrypted)');
    }
  };
}

export function createUnencryptedConnection(roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting to "' + roomId + '... (unencrypted)');
    },
    disconnect() {
      console.log('❌ Disconnected from "' + roomId + '" room (unencrypted)');
    }
  };
}

```

```
label { display: block; margin-bottom: 10px; }
```

It is correct that `createConnection` is a dependency. However, this code is a bit fragile because someone could edit the `App` component to pass an inline function as the value of this prop. In that case, its value would be different every time the `App` component re-renders, so the Effect might re-synchronize too often. To avoid this, you can pass `isEncrypted` down instead:

```

import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
  const [roomId, setRoomId] = useState('general');
  const [isEncrypted, setIsEncrypted] = useState(false);
  return (
    <>
      <label>
        Choose the chat room:<input type="text" value={roomId}>
      </label>
      <select
        value={roomId}
        onChange={e => setRoomId(e.target.value)}
      >
        <option value="general">general</option>
        <option value="travel">travel</option>
      </select>
    </>
  );
}

```

```

        <option value="music">music</option>
    </select>
</label>
<label>
    <input
        type="checkbox"
        checked={isEncrypted}
        onChange={e => setIsEncrypted(e.target.checked)}
    />
    Enable encryption
</label>
<hr />
<ChatRoom
    roomId={roomId}
    isEncrypted={isEncrypted}
/>
</>
);
}

```

```

import { useState, useEffect } from 'react';
import {
  createEncryptedConnection,
  createUnencryptedConnection,
} from './chat.js';

export default function ChatRoom({ roomId, isEncrypted }) {
  useEffect(() => {
    const createConnection = isEncrypted ?
      createEncryptedConnection :
      createUnencryptedConnection;
    const connection = createConnection(roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, isEncrypted]);

  return <h1>Welcome to the {roomId} room!</h1>;
}

```

```

export function createEncryptedConnection(roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ 🔒 Connecting to "' + roomId + '... (encrypted)');
    },
    disconnect() {
      console.log('❌ 🔒 Disconnected from "' + roomId + '" room (encrypted)');
    }
  };
}

export function createUnencryptedConnection(roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting to "' + roomId + '... (unencrypted)');
    },
    disconnect() {
      console.log('❌ Disconnected from "' + roomId + '" room (unencrypted)');
    }
  };
}

```

```
label { display: block; margin-bottom: 10px; }
```

In this version, the `App` component passes a boolean prop instead of a function. Inside the Effect, you decide which function to use. Since both `createEncryptedConnection` and `createUnencryptedConnection` are declared outside the component, they aren't reactive, and don't need to be dependencies. You'll learn more about this in [Removing Effect Dependencies](#).

Populate a chain of select boxes [{/populate-a-chain-of-select-boxes/}](#)

In this example, there are two select boxes. One select box lets the user pick a planet. Another select box lets the user pick a place *on that planet*. The second box doesn't work yet. Your task is to make it show the places on the chosen planet.

Look at how the first select box works. It populates the `planetList` state with the result from the `"/planets"` API call. The currently selected planet's ID is kept in the `planetId` state variable. You need to find where to add some additional code so that the `placeList` state variable is populated with the result of the `"/planets/" + planetId + "/places"` API call.

If you implement this right, selecting a planet should populate the place list. Changing a planet should change the place list.

If you have two independent synchronization processes, you need to write two separate Effects.

```
import { useState, useEffect } from 'react';
import { fetchData } from './api.js';

export default function Page() {
  const [planetList, setPlanetList] = useState([]);
  const [planetId, setPlanetId] = useState('');

  const [placeList, setPlaceList] = useState([]);
  const [placeId, setPlaceId] = useState('');

  useEffect(() => {
    let ignore = false;
    fetchData('/planets').then(result => {
      if (!ignore) {
        console.log('Fetched a list of planets.');
        setPlanetList(result);
        setPlanetId(result[0].id); // Select the first planet
      }
    });
    return () => {
      ignore = true;
    }
  }, []);
}

return (
  <>
  <label>
    Pick a planet:<' '>
    <select value={planetId} onChange={e => {
      setPlanetId(e.target.value);
    }}>
      {planetList.map(planet =>
        <option key={planet.id} value={planet.id}>{planet.name}</option>
      )}
    </select>
  </label>
)
```

```

        )}
      </select>
    </label>
    <label>
      Pick a place:{' '}
      <select value={placeId} onChange={e => {
        setPlaceId(e.target.value);
      }}>
        {placeList.map(place =>
          <option key={place.id} value={place.id}>{place.name}</option>
        )}
      </select>
    </label>
    <hr />
    <p>You are going to: {placeId || '???'} on {planetId || '???'} </p>
  </>
);
}

```

```

export function fetchData(url) {
  if (url === '/planets') {
    return fetchPlanets();
  } else if (url.startsWith('/planets/')) {
    const match = url.match(/^\/planets\/([w-]+)\places(\/)?$/);
    if (!match || !match[1] || !match[1].length) {
      throw Error('Expected URL like "/planets/earth/places". Received: "' + url +
      '".');
    }
    return fetchPlaces(match[1]);
  } else throw Error('Expected URL like "/planets" or "/planets/earth/places". Received: "' + url + '".');
}

async function fetchPlanets() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve([
        {name: 'Earth'},
        {name: 'Venus'},
        {name: 'Mars'}
      ]);
    }, 1000);
  });
}

async function fetchPlaces(planetId) {
  if (typeof planetId !== 'string') {
    throw Error(
      'fetchPlaces(planetId) expects a string argument. ' +
      'Instead received: ' + planetId + '.'
    );
  }
  return new Promise(resolve => {
    setTimeout(() => {
      if (planetId === 'earth') {
        resolve([
          {name: 'Laos'},
          {name: 'Spain'}
        ]);
      }
    }, 1000);
  });
}

```

```

        name: 'Vietnam'
    }]);
} else if (planetId === 'venus') {
    resolve([
        {
            name: 'Aurelia'
        },
        {
            name: 'Diana Chasma'
        },
        {
            name: 'Kümsöng Vallis'
        }
]);
} else if (planetId === 'mars') {
    resolve([
        {
            name: 'Aluminum City'
        },
        {
            name: 'New New York'
        },
        {
            name: 'Vishniac'
        }
]);
} else throw Error('Unknown planet ID: ' + planetId);
}, 1000);
});
}

```

```
label { display: block; margin-bottom: 10px; }
```

There are two independent synchronization processes:

- The first select box is synchronized to the remote list of planets.
- The second select box is synchronized to the remote list of places for the current `planetId`.

This is why it makes sense to describe them as two separate Effects. Here's an example of how you could do this:

```

import { useState, useEffect } from 'react';
import { fetchData } from './api.js';

export default function Page() {
    const [planetList, setPlanetList] = useState([]);
    const [planetId, setPlanetId] = useState('');

    const [placeList, setPlaceList] = useState([]);
    const [placeId, setPlaceId] = useState('');

    useEffect(() => {
        let ignore = false;
        fetchData('/planets').then(result => {
            if (!ignore) {
                console.log('Fetched a list of planets.');
                setPlanetList(result);
                setPlanetId(result[0].id); // Select the first planet
            }
        });
        return () => {
            ignore = true;
        }
    }, []);
}

useEffect(() => {
    if (planetId === '') {
        // Nothing is selected in the first box yet
    }
})

```

```

        return;
    }

    let ignore = false;
    fetchData('/planets/' + planetId + '/places').then(result => {
        if (!ignore) {
            console.log('Fetched a list of places on "' + planetId + '".');
            setPlaceList(result);
            setPlaceId(result[0].id); // Select the first place
        }
    });
    return () => {
        ignore = true;
    }
}, [planetId]);

return (
    <>
    <label>
        Pick a planet:{' '}
        <select value={planetId} onChange={e => {
            setPlanetId(e.target.value);
        }}>
            {planetList.map(planet =>
                <option key={planet.id} value={planet.id}>{planet.name}</option>
            )}
        </select>
    </label>
    <label>
        Pick a place:{' '}
        <select value={placeId} onChange={e => {
            setPlaceId(e.target.value);
        }}>
            {placeList.map(place =>
                <option key={place.id} value={place.id}>{place.name}</option>
            )}
        </select>
    </label>
    <hr />
    <p>You are going to: {placeId || '???'} on {planetId || '???'} </p>
)
);
}

```

```

export function fetchData(url) {
    if (url === '/planets') {
        return fetchPlanets();
    } else if (url.startsWith('/planets/')) {
        const match = url.match(/^\//planets\//([\w-]+)\//places(\//)?$/);
        if (!match || !match[1] || !match[1].length) {
            throw Error('Expected URL like "/planets/earth/places". Received: "' + url +
            '".');
        }
        return fetchPlaces(match[1]);
    } else throw Error('Expected URL like "/planets" or "/planets/earth/places". Received: "' + url + '".');
}

async function fetchPlanets() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve([

```

```

        name: 'Earth'
    }, {
        name: 'Venus'
    }, {
        name: 'Mars'
    }]);
}, 1000);
});

async function fetchPlaces(planetId) {
    if (typeof planetId !== 'string') {
        throw Error(
            `fetchPlaces(${planetId}) expects a string argument. +
            Instead received: ${planetId}.`
        );
    }
    return new Promise(resolve => {
        setTimeout(() => {
            if (planetId === 'earth') {
                resolve([
                    {
                        name: 'Laos'
                    }, {
                        name: 'Spain'
                    }, {
                        name: 'Vietnam'
                    }]);
            } else if (planetId === 'venus') {
                resolve([
                    {
                        name: 'Aurelia'
                    }, {
                        name: 'Diana Chasma'
                    }, {
                        name: 'Künsöng Vallis'
                    }]);
            } else if (planetId === 'mars') {
                resolve([
                    {
                        name: 'Aluminum City'
                    }, {
                        name: 'New New York'
                    }, {
                        name: 'Vishniac'
                    }]);
            } else throw Error(`Unknown planet ID: ${planetId}`);
        }, 1000);
    });
}

```

`label { display: block; margin-bottom: 10px; }`

This code is a bit repetitive. However, that's not a good reason to combine it into a single Effect! If you did this, you'd have to combine both Effect's dependencies into one list, and then changing the planet would refetch the list of all planets. Effects are not a tool for code reuse.

Instead, to reduce repetition, you can extract some logic into a custom Hook like `useSelectOptions` below:

```

import { useState } from 'react';
import { useSelectOptions } from './useSelectOptions.js';

export default function Page() {
  const [
    planetList,
    planetId,
    setPlanetId
  ] = useSelectOptions('/planets');

  const [
    placeList,
    placeId,
    setPlaceId
  ] = useSelectOptions(planetId ? `/planets/${planetId}/places` : null);

  return (
    <>
      <label>
        Pick a planet:{' '}
        <select value={planetId} onChange={e => {
          setPlanetId(e.target.value);
        }}>
          {planetList?.map(planet =>
            <option key={planet.id} value={planet.id}>{planet.name}</option>
          )}
        </select>
      </label>
      <label>
        Pick a place:{' '}
        <select value={placeId} onChange={e => {
          setPlaceId(e.target.value);
        }}>
          {placeList?.map(place =>
            <option key={place.id} value={place.id}>{place.name}</option>
          )}
        </select>
      </label>
      <hr />
      <p>You are going to: {placeId || '...'} on {planetId || '...'} </p>
    </>
  );
}

```

```

import { useState, useEffect } from 'react';
import { fetchData } from './api.js';

export function useSelectOptions(url) {
  const [list, setList] = useState(null);
  const [selectedId, setSelectedId] = useState('');
  useEffect(() => {
    if (url === null) {
      return;
    }

    let ignore = false;
    fetchData(url).then(result => {
      if (!ignore) {
        setList(result);
        setSelectedId(result[0].id);
      }
    });
  });
}

```

```

        return () => {
          ignore = true;
        }
      }, [url]);
      return [list, selectedId, setSelectedId];
    }
  }

export function fetchData(url) {
  if (url === '/planets') {
    return fetchPlanets();
  } else if (url.startsWith('/planets/')) {
    const match = url.match(/^\/planets\/([w-]+)\places(\/)?$/);
    if (!match || !match[1] || !match[1].length) {
      throw Error('Expected URL like "/planets/earth/places". Received: "' + url +
      '".');
    }
    return fetchPlaces(match[1]);
  } else throw Error('Expected URL like "/planets" or "/planets/earth/places". Received: "' + url + '".');
}

async function fetchPlanets() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve([
        {name: 'Earth'},
        {name: 'Venus'},
        {name: 'Mars'}
      ]);
    }, 1000);
  });
}

async function fetchPlaces(planetId) {
  if (typeof planetId !== 'string') {
    throw Error(
      'fetchPlaces(planetId) expects a string argument. ' +
      'Instead received: ' + planetId + '.'
    );
  }
  return new Promise(resolve => {
    setTimeout(() => {
      if (planetId === 'earth') {
        resolve([
          {name: 'Laos'},
          {name: 'Spain'},
          {name: 'Vietnam'}
        ]);
      } else if (planetId === 'venus') {
        resolve([
          {name: 'Aurelia'},
          {name: 'Diana Chasma'},
          {name: 'Kümsöng Vallis'}
        ]);
      } else if (planetId === 'mars') {
        resolve([
          ...
        ]);
      }
    });
  });
}

```

```
        name: 'Aluminum City'
    }, {
        name: 'New New York'
    }, {
        name: 'Vishniac'
    }]);
} else throw Error('Unknown planet ID: ' + planetId);
}, 1000);
};

label { display: block; margin-bottom: 10px; }
```

Check the `useSelectOptions.js` tab in the sandbox to see how it works. Ideally, most Effects in your application should eventually be replaced by custom Hooks, whether written by you or by the community. Custom Hooks hide the synchronization logic, so the calling component doesn't know about the Effect. As you keep working on your app, you'll develop a palette of Hooks to choose from, and eventually you won't need to write Effects in your components very often.

Managing State

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

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

Reacting to input with state {/reacting-to-input-with-state/}

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

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

```
import { 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);
  });
}

.Error { color: red; }

```

Read [Reacting to Input with State](#) to learn how to approach interactions with a state-driven mindset.

Choosing the state structure *{/choosing-the-state-structure/}*

Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs. The most important principle is that state shouldn't contain redundant or duplicated information. If there's unnecessary state, it's easy to forget to update it, and introduce bugs!

For example, this form has a **redundant** `fullName` state variable:

```
import { 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(firstName + ' ' + 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>
    </>
  );
}
```

```
label { display: block; margin-bottom: 5px; }
```

You can remove it and simplify the code by calculating `fullName` while the component is rendering:

```
import { 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>
    </>
);
}

```

```
label { display: block; margin-bottom: 5px; }
```

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

Read [Choosing the State Structure](#) to learn how to design the state shape to avoid bugs.

Sharing state between components {/sharing-state-between-components/}

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

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

```

import { 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.
        </Panel>
        <Panel
            title="History"
            isActive={activeIndex === 1}
            onShow={() => setActiveIndex(1)}
        >
            Almaty was founded in 1854 as a gold mining town.
        </Panel>
        <Panel
            title="Culture"
            isActive={activeIndex === 2}
            onShow={() => setActiveIndex(2)}
        >
            The city is home to several museums and theaters.
        </Panel>
    </>
)
}

```

```

From 1929 to 1997, it was its capital city.
  </Panel>
  <Panel
    title="Etymology"
    isActive={activeIndex === 1}
    onShow={() => setActiveIndex(1)}
  >
    The name comes from <span lang="kk-KZ">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,
  onShow
}) {
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={onShow}>
          Show
        </button>
      )}
    </section>
  );
}

```

```

h3, p { margin: 5px 0px; }
.panel {
  padding: 10px;
  border: 1px solid #aaa;
}

```

Read [Sharing State Between Components](#) to learn how to lift state up and keep components in sync.

Preserving and resetting state {/preserving-and-re-setting-state/}

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

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

```

import { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

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

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

```

```

export default function ContactList({
  selectedContact,
  contacts,
  onSelect
}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map(contact =>
          <li key={contact.email}>
            <button onClick={() => {
              onSelect(contact);
            }}>
              {contact.name}
            </button>
          </li>
        )}
      </ul>
    </section>
  );
}

```

```

import { useState } from 'react';

export default function Chat({ contact }) {
  const [text, setText] = useState('');
  return (
    <section className="chat">
      <textarea
        value={text}
        placeholder={'Chat to ' + contact.name}
        onChange={e => setText(e.target.value)}
      />
      <br />
      <button>Send to {contact.email}</button>
    </section>
  );
}

```

```
.chat, .contact-list {
  float: left;
  margin-bottom: 20px;
}
ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}
```

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

```
import { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

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

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

```
export default function ContactList({
  selectedContact,
  contacts,
  onSelect
}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map(contact =>
          <li key={contact.email}>
            <button onClick={() => {
```

```

        onSelect(contact);
    > }
    {contact.name}
    </button>
</li>
)
</ul>
</section>
);
}

```

```

import { useState } from 'react';

export default function Chat({ contact }) {
  const [text, setText] = useState('');
  return (
    <section className="chat">
      <textarea
        value={text}
        placeholder={'Chat to ' + contact.name}
        onChange={e => setText(e.target.value)}
      />
      <br />
      <button>Send to {contact.email}</button>
    </section>
  );
}

```

```

.chat, .contact-list {
  float: left;
  margin-bottom: 20px;
}
ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}

```

Read [Preserving and Resetting State](#) to learn the lifetime of state and how to control it.

Extracting state logic into a reducer {/extracting-state-logic-into-a-reducer/}

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

```

import { 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',
      text: text,
    });
  }

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

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

  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, {
        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 = [
];

```

```

import { useState } from 'react';

export default function AddTask({ onAddTask }) {
  const [text, setText] = useState('');
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button onClick={() => {
        setText('');
        onAddTask(text);
      }}>Add</button>
    </>
  )
}

```

```

import { useState } from 'react';

export default function TaskList({
  tasks,
  onChangeTask,
  onDeleteTask
}) {
  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>
          <Task
            task={task}
            onChange={onChangeTask}
            onDelete={onDeleteTask}
          />
        </li>
      ))}
    </ul>
  );
}

function Task({ task, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}

```

```

        onChange={e => {
          onChange({
            ...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 => {
        onChange({
          ...task,
          done: e.target.checked
        });
      }}
    />
    {taskContent}
    <button onClick={() => onDelete(task.id)}>
      Delete
    </button>
  </label>
);
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

Read [Extracting State Logic into a Reducer](#) to learn how to consolidate logic in the reducer function.

Passing data deeply with context *{/passing-data-deeply-with-context/}*

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

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

```
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>
    );
}
```

```
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>
  );
}
```

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
  const level = useContext(LevelContext);
  switch (level) {
    case 0:
      throw Error('Heading must be inside a Section!');
    case 1:
      return <h1>{children}</h1>;
    case 2:
      return <h2>{children}</h2>;
    case 3:
      return <h3>{children}</h3>;
    case 4:
      return <h4>{children}</h4>;
  }
}
```

```

    case 5:
      return <h5>{children}</h5>;
    case 6:
      return <h6>{children}</h6>;
    default:
      throw Error('Unknown level: ' + level);
  }
}

```

```

import { createContext } from 'react';
export const LevelContext = createContext(0);

```

```

.section {
  padding: 10px;
  margin: 5px;
  border-radius: 5px;
  border: 1px solid #aaa;
}

```

Read [Passing Data Deeply with Context](#) to learn about using context as an alternative to passing props.

Scaling up with reducer and context {/scaling-up-with-reducer-and-context/}

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

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

```

import 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>
  );
}

```

```

import { createContext, useContext, useReducer } from 'react';

const TasksContext = createContext(null);
const TasksDispatchContext = createContext(null);

export function TasksProvider({ children }) {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  )
  const value = {
    tasks,
    dispatch
  }
  return (
    <TasksContext.Provider value={value}>
      <TasksDispatchContext.Provider value={dispatch}>
        {children}
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  )
}

```

```

);
return (
  <TasksContext.Provider value={tasks}>
    <TasksDispatchContext.Provider
      value={dispatch}
    >
      {children}
    </TasksDispatchContext.Provider>
  </TasksContext.Provider>
);
}

export function useTasks() {
  return useContext(TasksContext);
}

export function useTasksDispatch() {
  return useContext(TasksDispatchContext);
}

function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [...tasks, {
        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);
    }
  }
}

const initialTasks = [
];

import { useState, useContext } from 'react';
import { useTasksDispatch } from './TasksContext.js';

export default function AddTask({ onAddTask }) {
  const [text, setText] = useState('');
  const dispatch = useTasksDispatch();
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={e => setText(e.target.value)}
      >
    </>
  );
}

```

```

        />
      <button onClick={() => {
        setText('');
        dispatch({
          type: 'added',
          text: text,
        });
      }}>Add</button>
    );
  }

let nextId = 3;

import { useState, useContext } 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',
      });
    }}>
      Delete
    </button>
  </label>
);
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

Read [Scaling Up with Reducer and Context](#) to learn how state management scales in a growing app.

What's next? *{/whats-next/}*

Head over to [Reacting to Input with State](#) to start reading this chapter page by page!

Or, if you're already familiar with these topics, why not read about [Escape Hatches](#)?

'Manipulating the DOM with Refs'

React automatically updates the [DOM](#) to match your render output, so your components won't often need to manipulate it. However, sometimes you might need access to the DOM elements managed by React--for example, to focus a node, scroll to it, or measure its size and position. There is no built-in way to do those things in React, so you will need a `ref` to the DOM node.

- 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 {/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 it to the DOM node as the `ref` attribute:

```
<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 {/example-focusing-a-text-input/}

In this example, clicking the button will focus the input:

```
import { 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>
      </>
    );
}

```

To implement this:

1. Declare `inputRef` with the `useRef` Hook.
2. Pass it as `<input ref={inputRef}>`. This tells React to **put this `<input>`'s DOM node into `inputRef.current`**.
3. In the `handleClick` function, read the input DOM node from `inputRef.current` and call `focus()` on it with `inputRef.current.focus()`.
4. 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 {/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:

```

import { 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() {
  }
}

```

react

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

```
div {
  width: 100%;
  overflow: hidden;
}

nav {
  text-align: center;
}

button {
  margin: .25rem;
}
```

```
ul,
li {
  list-style: none;
  white-space: nowrap;
}

li {
  display: inline;
  padding: 0.5rem;
}
```

How to manage a list of refs using a ref callback {/how-to-manage-a-list-of-refs-using-a-ref-callback/}

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**:

```
<ul>
  {items.map((item) => {
    // Doesn't work!
    const ref = useRef(null);
    return <li ref={ref} />;
  })}
</ul>
```

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:

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

export default function CatFriends() {
  const itemsRef = useRef(null);

  function scrollToId(itemId) {
    const map = getMap();
    const node = map.get(itemId);
    node.scrollIntoView({
      behavior: 'smooth',
      block: 'nearest',
      inline: 'center'
    });
  }

  function getMap() {
    if (!itemsRef.current) {
      // Initialize the Map on first usage.
    }
  }
}
```

```

        itemsRef.current = new Map();
    }
    return itemsRef.current;
}

return (
<>
<nav>
  <button onClick={() => scrollToId(0)}>
    Tom
  </button>
  <button onClick={() => scrollToId(5)}>
    Maru
  </button>
  <button onClick={() => scrollToId(9)}>
    Jellylorum
  </button>
</nav>
<div>
  <ul>
    {catList.map(cat => (
      <li
        key={cat.id}
        ref={(node) => {
          const map = getMap();
          if (node) {
            map.set(cat.id, node);
          } else {
            map.delete(cat.id);
          }
        }}
      >
        <img
          src={cat.imageUrl}
          alt={'Cat #' + cat.id}
        />
      </li>
    ))}
  </ul>
</div>
</>
);
}

const catList = [];
for (let i = 0; i < 10; i++) {
  catList.push({
    imageUrl: 'https://placekitten.com/250/200?image=' + i
  });
}

```

```

div {
  width: 100%;
  overflow: hidden;
}

nav {
  text-align: center;
}

button {
  margin: .25rem;
}

```

```

}
ul,
li {
  list-style: none;
  white-space: nowrap;
}

li {
  display: inline;
  padding: 0.5rem;
}

```

In this example, `itemsRef` doesn't hold a single DOM node. Instead, it holds a `Map` from item ID to a DOM node. (Refs can hold any values!) The `ref` callback on every list item takes care to update the Map:

```

<li
  key={cat.id}
  ref={node => {
    const map = getMap();
    if (node) {
      // Add to the Map
      map.set(cat.id, node);
    } else {
      // Remove from the Map
      map.delete(cat.id);
    }
  }}
>
```

This lets you read individual DOM nodes from the Map later.

Accessing another component's DOM nodes {/accessing-another-components-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:

```

import { 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>
    </>
  );
}

```

To help you notice the issue, React also prints an error to the console:

Warning: 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:

1. `<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.
2. 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`.
3. `MyInput` itself passes the `ref` it received to the `<input>` inside of it.

Now clicking the button to focus the input works:

```

import { 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>
    </>
  );
}

```

```

        </>
    );
}

```

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.

Exposing a subset of the API with an imperative handle {/exposing-a-subset-of-the-api-with-an-imperative-handle/}

In 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`:

```

import {
  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>
    </>
  );
}

```

Here, `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 {/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.

Flushing state updates synchronously with flushSync {/flushing-state-updates-synchronously-with-flush-sync/}

Consider 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:

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

export default function TodoList() {
  const listRef = useRef(null);
  const [text, setText] = useState('');
  const [todos, setTodos] = useState(
    initialTodos
  );

  function handleAdd() {
    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({
    text: 'Todo #' + (i + 1)
  });
}

```

The 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:

```

import { useState, useRef } from 'react';
import { flushSync } from 'react-dom';

export default function TodoList() {
  const listRef = useRef(null);
  const [text, setText] = useState('');
  const [todos, setTodos] = useState(
    initialTodos
  );

  function handleAdd() {
    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({
    text: 'Todo #' + (i + 1)
  });
}

```

Best practices for DOM manipulation with refs *{/best-practices-for-dom-manipulation-with-refs/}*

Refs are an escape hatch. You should only use them when you have to "step outside React". Common examples of this include managing focus, scroll position, or calling browser APIs that React does not expose.

If you stick to non-destructive actions like focusing and scrolling, you shouldn't encounter any problems. However, if you try to **modify** the DOM manually, you can risk conflicting with the changes React is making.

To illustrate this problem, this example includes a welcome message and two buttons. The first button toggles its presence using [conditional rendering](#) and [state](#), as you would usually do in React. The second button uses the [remove\(\) DOM API](#) to forcefully remove it from the DOM outside of React's control.

Try pressing "Toggle with setState" a few times. The message should disappear and appear again. Then press "Remove from the DOM". This will forcefully remove it. Finally, press "Toggle with setState":

```

import { 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>
    </div>
  );
}

```

```

    }]>
    Remove from the DOM
  </button>
  {show && <p ref={ref}>Hello world</p>}
</div>
);
}

```

```

p,
button {
  display: block;
  margin: 10px;
}

```

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.

- 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.

Play and pause the video [{/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.

```

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>
  )
}

```

```

        </button>
      <video width="250">
        <source
          src="https://interactive-examples.mdn.mozilla.net/media/cc0-
videos/flower.mp4"
          type="video/mp4"
        />
      </video>
    </>
  )
}

```

```
button { display: block; margin-bottom: 20px; }
```

For 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.

Declare a ref and put it on the `<video>` element. Then call `ref.current.play()` and `ref.current.pause()` in the event handler depending on the next state.

```

import { useState, useRef } from 'react';

export default function VideoPlayer() {
  const [isPlaying, setIsPlaying] = useState(false);
  const ref = useRef(null);

  function handleClick() {
    const nextIsPlaying = !isPlaying;
    setIsPlaying(nextIsPlaying);

    if (nextIsPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  }

  return (
    <>
      <button onClick={handleClick}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <video
        width="250"
        ref={ref}
        onPlay={() => setIsPlaying(true)}
        onPause={() => setIsPlaying(false)}
      >
        <source
          src="https://interactive-examples.mdn.mozilla.net/media/cc0-
videos/flower.mp4"
          type="video/mp4"
        />
      </video>
    </>
  )
}

```

```
button { display: block; margin-bottom: 20px; }
```

In order to handle the built-in browser controls, you can add `onPlay` and `onPause` handlers to the `<video>` element and call `setIsPlaying` from them. This way, if the user plays the video using the browser controls, the state will adjust accordingly.

Focus the search field `{/focus-the-search-field/}`

Make it so that clicking the "Search" button puts focus into the field.

```
export default function Page() {
  return (
    <>
      <nav>
        <button>Search</button>
      </nav>
      <input
        placeholder="Looking for something?"
      />
    </>
  );
}
```

```
button { display: block; margin-bottom: 10px; }
```

Add a ref to the input, and call `focus()` on the DOM node to focus it:

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

export default function Page() {
  const inputRef = useRef(null);
  return (
    <>
      <nav>
        <button onClick={() => {
          inputRef.current.focus();
        }}>
          Search
        </button>
      </nav>
      <input
        ref={inputRef}
        placeholder="Looking for something?"
      />
    </>
  );
}
```

```
button { display: block; margin-bottom: 10px; }
```

Scrolling an image carousel `{/scrolling-an-image-carousel/}`

This image carousel has a "Next" button that switches the active image. Make the gallery scroll horizontally to the active image on click. You will want to call `scrollIntoView()` on the DOM node of the active image:

react

```
node.scrollIntoView({
  behavior: 'smooth',
  block: 'nearest',
  inline: 'center'
});
```

You don't need to have a ref to every image for this exercise. It should be enough to have a ref to the currently active image, or to the list itself. Use `flushSync` to ensure the DOM is updated *before* you scroll.

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

export default function CatFriends() {
  const [index, setIndex] = useState(0);
  return (
    <>
      <nav>
        <button onClick={() => {
          if (index < catList.length - 1) {
            setIndex(index + 1);
          } else {
            setIndex(0);
          }
        }}>
          Next
        </button>
      </nav>
      <div>
        <ul>
          {catList.map((cat, i) => (
            <li key={cat.id}>
              <img
                className={
                  index === i ?
                    'active' :
                    ''
                }
                src={cat.imageUrl}
                alt={'Cat #' + cat.id}
              />
            </li>
          )));
        </ul>
      </div>
    </>
  );
}

const catList = [];
for (let i = 0; i < 10; i++) {
  catList.push({
    imageUrl: 'https://placekitten.com/250/200?image=' + i
  });
}
```

```
div {
  width: 100%;
  overflow: hidden;
}

nav {
  text-align: center;
```

```

}

button {
  margin: .25rem;
}

ul,
li {
  list-style: none;
  white-space: nowrap;
}

li {
  display: inline;
  padding: 0.5rem;
}

img {
  padding: 10px;
  margin: -10px;
  transition: background 0.2s linear;
}

.active {
  background: rgba(0, 100, 150, 0.4);
}

```

You can declare a `selectedRef`, and then pass it conditionally only to the current image:

```
<li ref={index === i ? selectedRef : null}>
```

When `index === i`, meaning that the image is the selected one, the `` will receive the `selectedRef`. React will make sure that `selectedRef.current` always points at the correct DOM node.

Note that the `flushSync` call is necessary to force React to update the DOM before the scroll. Otherwise, `selectedRef.current` would always point at the previously selected item.

```

import { useRef, useState } from 'react';
import { flushSync } from 'react-dom';

export default function CatFriends() {
  const selectedRef = useRef(null);
  const [index, setIndex] = useState(0);

  return (
    <>
      <nav>
        <button onClick={() => {
          flushSync(() => {
            if (index < catList.length - 1) {
              setIndex(index + 1);
            } else {
              setIndex(0);
            }
          });
          selectedRef.current.scrollIntoView({
            behavior: 'smooth',
            block: 'nearest',
            inline: 'center'
          });
        }}>

```

```

    });
  > Next
  </button>
</nav>
<div>
  <ul>
    {catList.map((cat, i) => (
      <li
        key={cat.id}
        ref={index === i ?
          selectedRef :
          null
        }
      >
        <img
          className={
            index === i ?
              'active'
            :
            ''
          }
          src={cat.imageUrl}
          alt={'Cat #' + cat.id}
        />
      </li>
    ))}
  </ul>
</div>
</>
);
}

const catList = [];
for (let i = 0; i < 10; i++) {
  catList.push({
    imageUrl: 'https://placekitten.com/250/200?image=' + i
  });
}

```

```

div {
  width: 100%;
  overflow: hidden;
}

nav {
  text-align: center;
}

button {
  margin: .25rem;
}

ul,
li {
  list-style: none;
  white-space: nowrap;
}

li {
  display: inline;
  padding: 0.5rem;
}

```

```
img {
  padding: 10px;
  margin: -10px;
  transition: background 0.2s linear;
}

.active {
  background: rgba(0, 100, 150, 0.4);
}
```

Focus the search field with separate components {/focus-the-search-field-with-separate-components/}

Make it so that clicking the "Search" button puts focus into the field. Note that each component is defined in a separate file and shouldn't be moved out of it. How do you connect them together?

You'll need `forwardRef` to opt into exposing a DOM node from your own component like `SearchInput`.

```
import SearchButton from './SearchButton.js';
import SearchInput from './SearchInput.js';

export default function Page() {
  return (
    <>
      <nav>
        <SearchButton />
      </nav>
      <SearchInput />
    </>
  );
}
```

```
export default function SearchButton() {
  return (
    <button>
      Search
    </button>
  );
}
```

```
export default function SearchInput() {
  return (
    <input
      placeholder="Looking for something?"
    />
  );
}
```

```
button { display: block; margin-bottom: 10px; }
```

You'll need to add an `onClick` prop to the `SearchButton`, and make the `SearchButton` pass it down to the browser `<button>`. You'll also pass a ref down to `<SearchInput>`, which will forward it to the real `<input>` and populate it. Finally, in the click handler, you'll call `focus` on the DOM node stored inside that ref.

react

```
import { useRef } from 'react';
import SearchButton from './SearchButton.js';
import SearchInput from './SearchInput.js';

export default function Page() {
  const inputRef = useRef(null);
  return (
    <>
      <nav>
        <SearchButton onClick={() => {
          inputRef.current.focus();
        }} />
      </nav>
      <SearchInput ref={inputRef} />
    </>
  );
}
```

```
export default function SearchButton({ onClick }) {
  return (
    <button onClick={onClick}>
      Search
    </button>
  );
}
```

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

export default forwardRef(
  function SearchInput(props, ref) {
    return (
      <input
        ref={ref}
        placeholder="Looking for something?"
      />
    );
  }
);
```

```
button { display: block; margin-bottom: 10px; }
```

[Go to TOC](#)

Passing Data Deeply with Context

Usually, you will pass information from a parent component to a child component via props. But passing props can become 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.

- 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 {/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 up

Prop 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-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:

```
import 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>
    );
}

export default function Section({ children }) {
    return (
        <section className="section">
            {children}
        </section>
    );
}

export default function Heading({ level, children }) {
    switch (level) {
        case 1:
            return <h1>{children}</h1>;
        case 2:
            return <h2>{children}</h2>;
        case 3:
            return <h3>{children}</h3>;
        case 4:
            return <h4>{children}</h4>;
        case 5:
            return <h5>{children}</h5>;
        case 6:
            return <h6>{children}</h6>;
        default:
            throw Error('Unknown level: ' + level);
    }
}

.section {
    padding: 10px;
    margin: 5px;
    border-radius: 5px;
    border: 1px solid #aaa;
}

```

Let's say you want multiple headings within the same `Section` to always have the same size:

```

import 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>
    );
}

```

```

        </Section>
    </Section>
</Section>
);
}

```

```

export default function Section({ children }) {
  return (
    <section className="section">
      {children}
    </section>
  );
}

```

```

export default function Heading({ level, children }) {
  switch (level) {
    case 1:
      return <h1>{children}</h1>;
    case 2:
      return <h2>{children}</h2>;
    case 3:
      return <h3>{children}</h3>;
    case 4:
      return <h4>{children}</h4>;
    case 5:
      return <h5>{children}</h5>;
    case 6:
      return <h6>{children}</h6>;
    default:
      throw Error('Unknown level: ' + level);
  }
}

```

```

.section {
  padding: 10px;
  margin: 5px;
  border-radius: 5px;
  border: 1px solid #aaa;
}

```

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:

1. **Create** a context. (You can call it `LevelContext`, since it's for the heading level.)
2. **Use** that context from the component that needs the data. (`Heading` will use `LevelContext`.)
3. **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 `{/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:

```
import 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>
    );
}
```

```
export default function Section({ children }) {
  return (
    <section className="section">
      {children}
    </section>
  );
}
```

```
export default function Heading({ level, children }) {
  switch (level) {
    case 1:
      return <h1>{children}</h1>;
    case 2:
      return <h2>{children}</h2>;
    case 3:
      return <h3>{children}</h3>;
    case 4:
      return <h4>{children}</h4>;
    case 5:
      return <h5>{children}</h5>;
    case 6:
      return <h6>{children}</h6>;
    default:
      throw Error('Unknown level: ' + level);
  }
}
```

```
import { createContext } from 'react';
export const LevelContext = createContext(1);
```

```
.section {
  padding: 10px;
  margin: 5px;
  border-radius: 5px;
  border: 1px solid #aaa;
}
```

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 [{/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:

```
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>
  );
}
```

```
export default function Section({ children }) {
  return (
    <section className="section">
      {children}
    </section>
  );
}
```

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
  const level = useContext(LevelContext);
  switch (level) {
    case 1:
      return <h1>{children}</h1>;
    case 2:
      return <h2>{children}</h2>;
    case 3:
      return <h3>{children}</h3>;
    case 4:
      return <h4>{children}</h4>;
    case 5:
      return <h5>{children}</h5>;
    case 6:
      return <h6>{children}</h6>;
    default:
      throw Error('Unknown level: ' + level);
  }
}
```

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

export const LevelContext = createContext(1);

.section {
  padding: 10px;
  margin: 5px;
  border-radius: 5px;
  border: 1px solid #aaa;
}
```

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 [{/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:

react

```
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.

```
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>
  );
}
```

```
import { LevelContext } from './LevelContext.js';

export default function Section({ level, children }) {
  return (
    <section className="section">
      <LevelContext.Provider value={level}>
        {children}
      </LevelContext.Provider>
    </section>
  );
}
```

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
  const level = useContext(LevelContext);
  switch (level) {
    case 1:
```

```

        return <h1>{children}</h1>;
    case 2:
        return <h2>{children}</h2>;
    case 3:
        return <h3>{children}</h3>;
    case 4:
        return <h4>{children}</h4>;
    case 5:
        return <h5>{children}</h5>;
    case 6:
        return <h6>{children}</h6>;
    default:
        throw Error('Unknown level: ' + level);
    }
}

```

```

import { createContext } from 'react';

export const LevelContext = createContext(1);

```

```

.section {
  padding: 10px;
  margin: 5px;
  border-radius: 5px;
  border: 1px solid #aaa;
}

```

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:

1. You pass a `level` prop to the `<Section>`.
2. `Section` wraps its children into `<LevelContext.Provider value={level}>`.
3. `Heading` asks the closest value of `LevelContext` above with `useContext(LevelContext)`.

Using and providing context from the same component *{/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:

react

```
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>`:

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

```
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>
  );
}
```

```
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
  const level = useContext(LevelContext);
```

```

switch (level) {
  case 0:
    throw Error('Heading must be inside a Section!');
  case 1:
    return <h1>{children}</h1>;
  case 2:
    return <h2>{children}</h2>;
  case 3:
    return <h3>{children}</h3>;
  case 4:
    return <h4>{children}</h4>;
  case 5:
    return <h5>{children}</h5>;
  case 6:
    return <h6>{children}</h6>;
  default:
    throw Error('Unknown level: ' + level);
}
}

```

```

import { createContext } from 'react';

export const LevelContext = createContext(0);

.section {
  padding: 10px;
  margin: 5px;
  border-radius: 5px;
  border: 1px solid #aaa;
}

```

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.

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

{/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>`:

```

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>
  );
}

import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Section({ children, isFancy }) {
  const level = useContext(LevelContext);
  return (
    <section className={
      'section ' +
      (isFancy ? 'fancy' : '')
    }>
      <LevelContext.Provider value={level + 1}>
        {children}
      </LevelContext.Provider>
    
```

```

        </section>
    );
}

import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
  const level = useContext(LevelContext);
  switch (level) {
    case 0:
      throw Error('Heading must be inside a Section!');
    case 1:
      return <h1>{children}</h1>;
    case 2:
      return <h2>{children}</h2>;
    case 3:
      return <h3>{children}</h3>;
    case 4:
      return <h4>{children}</h4>;
    case 5:
      return <h5>{children}</h5>;
    case 6:
      return <h6>{children}</h6>;
    default:
      throw Error('Unknown level: ' + level);
  }
}

```

```

import { createContext } from 'react';

export const LevelContext = createContext(0);

.section {
  padding: 10px;
  margin: 5px;
  border-radius: 5px;
  border: 1px solid #aaa;
}

.fancy {
  border: 4px dashed pink;
}

```

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 {/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:

1. **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.
2. **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 {/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.

- Context lets a component provide some information to the entire tree below it.
- To pass context:
 1. Create and export it with `export const MyContext = createContext(defaultValue)`.
 2. Pass it to the `useContext(MyContext)` Hook to read it in any child component, no matter how deep.
 3. 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`.

Replace prop drilling with context [{/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`.

```
import { useState } from 'react';
import { places } from './data.js';
import { getImageUrl } from './utils.js';

export default function App() {
  const [isLarge, setIsLarge] = useState(false);
  const imageSize = isLarge ? 150 : 100;
  return (
    <>
      <label>
        <input
          type="checkbox"
          checked={isLarge}
          onChange={e => {
            setIsLarge(e.target.checked);
          }}
        />
        Use large images
      </label>
      <hr />
      <List imageSize={imageSize} />
    </>
  )
}

function List({ imageSize }) {
  const listItems = places.map(place =>
    <li key={place.id}>
      <Place
        place={place}
        imageSize={imageSize}
      />
    </li>
  );
  return <ul>{listItems}</ul>;
}
```

```

function Place({ place, imageSize }) {
  return (
    <>
      <PlaceImage
        place={place}
        imageSize={imageSize}
      />
      <p>
        <b>{place.name}</b>
        {': ' + place.description}
      </p>
    </>
  );
}

function PlaceImage({ place, imageSize }) {
  return (
    <img
      src={getImageUrl(place)}
      alt={place.name}
      width={imageSize}
      height={imageSize}
    />
  );
}

```

```

export const places = [
  {
    name: 'Bo-Kaap in Cape Town, South Africa',
    description: 'The tradition of choosing bright colors for houses began in the late 20th century.',
    imageId: 'K9HVAGH'
  },
  {
    name: 'Rainbow Village in Taichung, Taiwan',
    description: 'To save the houses from demolition, Huang Yung-Fu, a local resident, painted all 1,200 of them in 1924.',
    imageId: '9EAYZrt'
  },
  {
    name: 'Macromural de Pachuca, Mexico',
    description: 'One of the largest murals in the world covering homes in a hillside neighborhood.',
    imageId: 'DgXHVwu'
  },
  {
    name: 'Selarón Staircase in Rio de Janeiro, Brazil',
    description: 'This landmark was created by Jorge Selarón, a Chilean-born artist, as a "tribute to the Brazilian people."',
    imageId: 'ae03rpI'
  },
  {
    name: 'Burano, Italy',
    description: 'The houses are painted following a specific color system dating back to 16th century.',
    imageId: 'kxsph5C'
  },
  {
    name: 'Chefchaouen, Morocco',
    description: 'There are a few theories on why the houses are painted blue, including that the color repells mosquitos or that it symbolizes sky and heaven.',
    imageId: 'rTqKo46'
  },
  {
    name: 'Gamcheon Culture Village in Busan, South Korea',
    description: 'In 2009, the village was converted into a cultural hub by painting the houses and featuring exhibitions and art installations.',
    imageId: 'ZfQ00zf'
  }];

```

```
export function getImageUrl(place) {
  return (
    `https://i.imgur.com/` +
    place.imageId +
    `l.jpg`
  );
}
```

```
ul { list-style-type: none; padding: 0px 10px; }
li {
  margin-bottom: 10px;
  display: grid;
  grid-template-columns: auto 1fr;
  gap: 20px;
  align-items: center;
}
```

Remove `imageSize` prop from all the components.

Create and export `ImageSizeContext` from `Context.js`. Then wrap the List into `<ImageSizeContext.Provider value={imageSize}>` to pass the value down, and `useContext(ImageSizeContext)` to read it in the `PlaceImage`:

```
import { useState, useContext } from 'react';
import { places } from './data.js';
import { getImageUrl } from './utils.js';
import { ImageSizeContext } from './Context.js';

export default function App() {
  const [isLarge, setIsLarge] = useState(false);
  const imageSize = isLarge ? 150 : 100;
  return (
    <ImageSizeContext.Provider
      value={imageSize}
    >
      <label>
        <input
          type="checkbox"
          checked={isLarge}
          onChange={e => {
            setIsLarge(e.target.checked);
          }}
        />
        Use large images
      </label>
      <hr />
      <List />
    </ImageSizeContext.Provider>
  )
}

function List() {
  const listItems = places.map(place =>
    <li key={place.id}>
      <Place place={place} />
    </li>
  );
  return <ul>{listItems}</ul>;
}
```

```

function Place({ place }) {
  return (
    <>
      <PlaceImage place={place} />
      <p>
        <b>{place.name}</b>
        {': ' + place.description}
      </p>
    </>
  );
}

function PlaceImage({ place }) {
  const imageSize = useContext(ImageSizeContext);
  return (
    <img
      src={getImageUrl(place)}
      alt={place.name}
      width={imageSize}
      height={imageSize}
    />
  );
}

```

```

import { createContext } from 'react';

export const ImageSizeContext = createContext(500);

```

```

export const places = [
  {
    name: 'Bo-Kaap in Cape Town, South Africa',
    description: 'The tradition of choosing bright colors for houses began in the late 20th century.',
    imageId: 'K9HVAGH'
  },
  {
    name: 'Rainbow Village in Taichung, Taiwan',
    description: 'To save the houses from demolition, Huang Yung-Fu, a local resident, painted all 1,200 of them in 1924.',
    imageId: '9EAYZrt'
  },
  {
    name: 'Macromural de Pachuca, Mexico',
    description: 'One of the largest murals in the world covering homes in a hillside neighborhood.',
    imageId: 'DgXHVwu'
  },
  {
    name: 'Selarón Staircase in Rio de Janeiro, Brazil',
    description: 'This landmark was created by Jorge Selarón, a Chilean-born artist, as a "tribute to the Brazilian people".',
    imageId: 'ae03rpI'
  },
  {
    name: 'Burano, Italy',
    description: 'The houses are painted following a specific color system dating back to 16th century.',
    imageId: 'kxsph5C'
  },
  {
    name: 'Chefchaouen, Marocco',
    description: 'There are a few theories on why the houses are painted blue, including that the color repells mosquitos or that it symbolizes sky and heaven.',
    imageId: 'rTqKo46'
  },
  {
    name: 'Gamcheon Culture Village in Busan, South Korea',
    description: 'In 2009, the village was converted into a cultural hub by painting'
  }
];

```

```
the houses and featuring exhibitions and art installations.',  
  imageUrl: 'ZfQ00zf'  
};
```

```
export function getImageUrl(place) {  
  return (  
    'https://i.imgur.com/' +  
    place.imageUrl +  
    'l.jpg'  
  );  
}
```

```
ul { list-style-type: none; padding: 0px 10px; }  
li {  
  margin-bottom: 10px;  
  display: grid;  
  grid-template-columns: auto 1fr;  
  gap: 20px;  
  align-items: center;  
}
```

Note how components in the middle don't need to pass `imageSize` anymore.

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.

- 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 {/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 ``:

```
function Avatar() {
  return (
    
  );
}

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

```
body { min-height: 120px; }
.avatar { margin: 20px; border-radius: 50%; }
```

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 {/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 {/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}
    />
  );
}
```

If 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 {/step-2-read-props-inside-the-child-component/}

You can read these props by listing their names `person`, `size` separated by the commas inside `{` and `}` directly after `function Avatar`. This lets you use them inside the `Avatar` code, like you would with a variable.

```
function Avatar({ person, size }) {
  // person and size are available here
}
```

Add some logic to `Avatar` that uses the `person` and `size` props for rendering, and you're done.

Now you can configure `Avatar` to render in many different ways with different props. Try tweaking the values!

```
import { 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: 'Yfe0qp2'
        }}
      />
      <Avatar
        size={80}
        person={{
          name: 'Aklilu Lemma',
          imageUrl: 'OKS67lh'
        }}
      />
      <Avatar
        size={50}
        person={{
          name: 'Lin Lanying',
          imageUrl: '1bX5QH6'
        }}
      />
    </div>
  );
}

```

```

export function getImageUrl(person, size = 's') {
  return (
    'https://i.imgur.com/' +
    person.imageUrl +
    size +
    '.jpg'
  );
}

```

```

body { min-height: 120px; }
.avatar { margin: 10px; border-radius: 50%; }

```

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.

Don't miss the pair of `{` and `}` curly braces 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 *{/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 *{/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}>
      </Avatar>
    </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 {/passing-javascript-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:

```
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',
          imageId: 'Yfe0qp2'
        }}
      />
    </Card>
  );
}
```

```
import { getImageUrl } from './utils.js';

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

```
export function getUrl(person, size = 's') {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    size +
    '.jpg'
  );
}
```

```
.card {
  width: fit-content;
  margin: 5px;
  padding: 5px;
  font-size: 20px;
  text-align: center;
  border: 1px solid #aaa;
  border-radius: 20px;
  background: #fff;
}

.avatar {
  margin: 20px;
  border-radius: 50%;
```

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.

How props change over time {/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:

```

export default function Clock({ color, time }) {
  return (
    <h1 style={{ color: color }}>
      {time}
    </h1>
  );
}

import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
  const [time, setTime] = useState(() => new Date());
  useEffect(() => {
    const id = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return time;
}

export default function App() {
  const time = useTime();
  const [color, setColor] = useState('lightcoral');
  return (
    <div>
      <p>
        Pick a color:{' '}
        <select value={color} onChange={e => setColor(e.target.value)}>
          <option value="lightcoral">lightcoral</option>
          <option value="midnightblue">midnightblue</option>
          <option value="rebeccapurple">rebeccapurple</option>
        </select>
      </p>
      <Clock color={color} time={time.toLocaleTimeString()} />
    </div>
  );
}

```

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](#).

- 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.

Extract a component `{/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.

```
import { getImageUrl } from './utils.js';

export default function Gallery() {
  return (
    <div>
      <h1>Notable Scientists</h1>
      <section className="profile">
        <h2>Maria Skłodowska-Curie</h2>
        <img
          className="avatar"
          src={getImageUrl('szV5sdG')}
          alt="Maria Skłodowska-Curie"
          width={70}
          height={70}
        />
        <ul>
          <li>
            <b>Profession:</b>
            physicist and chemist
          </li>
          <li>
            <b>Awards:</b> 4
            (Nobel Prize in Physics, Nobel Prize in Chemistry, Davy Medal,
            Matteucci Medal)
          </li>
          <li>
            <b>Discovered:</b>
            polonium (element)
          </li>
        </ul>
      </section>
      <section className="profile">
        <h2>Katsuko Saruhashi</h2>
        <img
          className="avatar"
          src={getImageUrl('Yfe0qp2')}
          alt="Katsuko Saruhashi"
          width={70}
          height={70}
        />
        <ul>
          <li>
            <b>Profession:</b>
            geochemist
          </li>
          <li>
            <b>Awards:</b> 2
            (Miyake Prize for geochemistry, Tanaka Prize)
          </li>
          <li>
            <b>Discovered:</b>
          </li>
        </ul>
      </section>
    </div>
  );
}
```

```

        a method for measuring carbon dioxide in seawater
    </li>
  </ul>
</section>
</div>
);
}

```

```

export function getImageUrl(imageId, size = 's') {
  return (
    `https://i.imgur.com/${imageId}${size}.jpg`
  );
}

```

```

.avatar { margin: 5px; border-radius: 50%; min-height: 70px; }
.profile {
  border: 1px solid #aaa;
  border-radius: 6px;
  margin-top: 20px;
  padding: 10px;
}
h1, h2 { margin: 5px; }
h1 { margin-bottom: 10px; }
ul { padding: 0px 10px 0px 20px; }
li { margin: 5px; }

```

Start by extracting the markup for one of the scientists. Then find the pieces that don't match it in the second example, and make them configurable by props.

In this solution, the `Profile` component accepts multiple props: `imageId` (a string), `name` (a string), `profession` (a string), `awards` (an array of strings), `discovery` (a string), and `imageSize` (a number).

Note that the `imageSize` prop has a default value, which is why we don't pass it to the component.

```

import { getImageUrl } from './utils.js';

function Profile({
  imageId,
  name,
  profession,
  awards,
  discovery,
  imageSize = 70
}) {
  return (
    <section className="profile">
      <h2>{name}</h2>
      <img
        className="avatar"
        src={getImageUrl(imageId)}
        alt={name}
        width={imageSize}
        height={imageSize}
      />
      <ul>
        <li><b>Profession:</b> {profession}</li>
      
```

```

        <li>
          <b>Awards: {awards.length} </b>
          ({awards.join(', ')})
        </li>
        <li>
          <b>Discovered: </b>
          {discovery}
        </li>
      </ul>
    </section>
  );
}

export default function Gallery() {
  return (
    <div>
      <h1>Notable Scientists</h1>
      <Profile
        imageUrl="szV5sdG"
        name="Maria Skłodowska-Curie"
        profession="physicist and chemist"
        discovery="polonium (chemical element)"
        awards={[
          'Nobel Prize in Physics',
          'Nobel Prize in Chemistry',
          'Davy Medal',
          'Matteucci Medal'
        ]}
      />
      <Profile
        imageUrl='Yfe0qp2'
        name='Katsuko Saruhashi'
        profession='geochemist'
        discovery="a method for measuring carbon dioxide in seawater"
        awards={[
          'Miyake Prize for geochemistry',
          'Tanaka Prize'
        ]}
      />
    </div>
  );
}

```

```

export function getImageUrl(imageId, size = 's') {
  return (
    'https://i.imgur.com/' +
    imageUrl +
    size +
    '.jpg'
  );
}

```

```

.avatar { margin: 5px; border-radius: 50%; min-height: 70px; }
.profile {
  border: 1px solid #aaa;
  border-radius: 6px;
  margin-top: 20px;
  padding: 10px;
}
h1, h2 { margin: 5px; }

```

```
h1 { margin-bottom: 10px; }
ul { padding: 0px 10px 0px 20px; }
li { margin: 5px; }
```

Note how you don't need a separate `awardCount` prop if `awards` is an array. Then you can use `awards.length` to count the number of awards. Remember that props can take any values, and that includes arrays too!

Another solution, which is more similar to the earlier examples on this page, is to group all information about a person in a single object, and pass that object as one prop:

```
import { getImageUrl } from './utils.js';

function Profile({ person, imageSize = 70 }) {
  const imageSrc = getImageUrl(person)

  return (
    <section className="profile">
      <h2>{person.name}</h2>
      <img
        className="avatar"
        src={imageSrc}
        alt={person.name}
        width={imageSize}
        height={imageSize}
      />
      <ul>
        <li>
          <b>Profession:</b> {person.profession}
        </li>
        <li>
          <b>Awards: {person.awards.length}</b>
          ({person.awards.join(', ')})
        </li>
        <li>
          <b>Discovered:</b>
          {person.discovery}
        </li>
      </ul>
    </section>
  )
}

export default function Gallery() {
  return (
    <div>
      <h1>Notable Scientists</h1>
      <Profile person={{
        imageUrl: 'szV5sdG',
        name: 'Maria Skłodowska-Curie',
        profession: 'physicist and chemist',
        discovery: 'polonium (chemical element)',
        awards: [
          'Nobel Prize in Physics',
          'Nobel Prize in Chemistry',
          'Davy Medal',
          'Matteucci Medal'
        ],
      }} />
      <Profile person={{
```

```

        imageUrl: 'Yfe0qp2',
        name: 'Katsuko Saruhashi',
        profession: 'geochemist',
        discovery: 'a method for measuring carbon dioxide in seawater',
        awards: [
          'Miyake Prize for geochemistry',
          'Tanaka Prize'
        ],
      }]}
    </div>
  );
}

```

```

export function getImageUrl(person, size = 's') {
  return (
    `https://i.imgur.com/${person.imageUrl +
    size +
    '.jpg'}`);
}

```

```

.avatar { margin: 5px; border-radius: 50%; min-height: 70px; }
.profile {
  border: 1px solid #aaa;
  border-radius: 6px;
  margin-top: 20px;
  padding: 10px;
}
h1, h2 { margin: 5px; }
h1 { margin-bottom: 10px; }
ul { padding: 0px 10px 0px 20px; }
li { margin: 5px; }

```

Although the syntax looks slightly different because you're describing properties of a JavaScript object rather than a collection of JSX attributes, these examples are mostly equivalent, and you can pick either approach.

Adjust the image size based on a prop [{/adjust-the-image-size-based-on-a-prop/}](#)

In this example, `Avatar` receives a numeric `size` prop which determines the `` width and height. The `size` prop is set to `40` in this example. However, if you open the image in a new tab, you'll notice that the image itself is larger (`160` pixels). The real image size is determined by which thumbnail size you're requesting.

Change the `Avatar` component to request the closest image size based on the `size` prop. Specifically, if the `size` is less than `90`, pass `'s'` ("small") rather than `'b'` ("big") to the `getImageUrl` function. Verify that your changes work by rendering avatars with different values of the `size` prop and opening images in a new tab.

```

import { getImageUrl } from './utils.js';

function Avatar({ person, size }) {
  return (
    <img
      className="avatar"

```

```

    src={getImageUrl(person, 'b')}
    alt={person.name}
    width={size}
    height={size}
  />
);
}

export default function Profile() {
  return (
    <Avatar
      size={40}
      person={{
        name: 'Gregorio Y. Zara',
        imageUrl: '7vQD0fP'
      }}
    />
  );
}

```

```

export function getImageUrl(person, size) {
  return (
    'https://i.imgur.com/' +
    person.imageUrl +
    size +
    '.jpg'
  );
}

```

```
.avatar { margin: 20px; border-radius: 50%; }
```

Here is how you could go about it:

```

import { getImageUrl } from './utils.js';

function Avatar({ person, size }) {
  let thumbnailSize = 's';
  if (size > 90) {
    thumbnailSize = 'b';
  }
  return (
    <img
      className="avatar"
      src={getImageUrl(person, thumbnailSize)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}

export default function Profile() {
  return (
    <>
      <Avatar
        size={40}
        person={{
          name: 'Gregorio Y. Zara',
          imageUrl: '7vQD0fP'
        }}
      />
    </>
  );
}

```

```

<Avatar
  size={120}
  person={{
    name: 'Gregorio Y. Zara',
    imageUrl: '7vQD0fP'
  }}
/>
</>
);
}

```

```

export function getImageUrl(person, size) {
  return (
    'https://i.imgur.com/' +
    person.imageUrl +
    size +
    '.jpg'
  );
}

```

```
.avatar { margin: 20px; border-radius: 50%; }
```

You could also show a sharper image for high DPI screens by taking `window.devicePixelRatio` into account:

```

import { getImageUrl } from './utils.js';

const ratio = window.devicePixelRatio;

function Avatar({ person, size }) {
  let thumbnailSize = 's';
  if (size * ratio > 90) {
    thumbnailSize = 'b';
  }
  return (
    <img
      className="avatar"
      src={getImageUrl(person, thumbnailSize)}
      alt={person.name}
      width={size}
      height={size}
    />
  );
}

export default function Profile() {
  return (
    <>
      <Avatar
        size={40}
        person={{
          name: 'Gregorio Y. Zara',
          imageUrl: '7vQD0fP'
        }}
      />
      <Avatar
        size={70}
        person={{
          name: 'Gregorio Y. Zara',
          imageUrl: '7vQD0fP'
        }}
      />
    </>
  );
}

```

```

        }
      />
    <Avatar
      size={120}
      person={{
        name: 'Gregorio Y. Zara',
        imageUrl: '7vQD0fP'
      }}
    />
  );
}

```

```

export function getImageUrl(person, size) {
  return (
    `https://i.imgur.com/${person.imageUrl}/${
      size + '.jpg'
    }`);
}

```

```
.avatar { margin: 20px; border-radius: 50%; }
```

Props let you encapsulate logic like this inside the `Avatar` component (and change it later if needed) so that everyone can use the `<Avatar>` component without thinking about how the images are requested and resized.

Passing JSX in a `children` prop {/passing-jsx-in-a-children-prop/}

Extract a `Card` component from the markup below, and use the `children` prop to pass different JSX to it:

```

export default function Profile() {
  return (
    <div>
      <div className="card">
        <div className="card-content">
          <h1>Photo</h1>
          
        </div>
      </div>
      <div className="card">
        <div className="card-content">
          <h1>About</h1>
          <p>Aklilu Lemma was a distinguished Ethiopian scientist who discovered a natural treatment to schistosomiasis.</p>
        </div>
      </div>
    </div>
  );
}

```

```
.card {
  width: fit-content;
  margin: 20px;
  padding: 20px;
  border: 1px solid #aaa;
  border-radius: 20px;
  background: #fff;
}

.card-content {
  text-align: center;
}

.avatar {
  margin: 10px;
  border-radius: 50%;
}

h1 {
  margin: 5px;
  padding: 0;
  font-size: 24px;
}
```

Any JSX you put inside of a component's tag will be passed as the `children` prop to that component.

This is how you can use the `Card` component in both places:

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

export default function Profile() {
  return (
    <div>
      <Card>
        <h1>Photo</h1>
        
      </Card>
      <Card>
        <h1>About</h1>
        <p>Aklilu Lemma was a distinguished Ethiopian scientist who discovered a natural treatment to schistosomiasis.</p>
      </Card>
    </div>
  );
}
```

```
.card {
  width: fit-content;
  margin: 20px;
  padding: 20px;
```

```

border: 1px solid #aaa;
border-radius: 20px;
background: #fff;
}
.card-content {
  text-align: center;
}
.avatar {
  margin: 10px;
  border-radius: 50%;
}
h1 {
  margin: 5px;
  padding: 0;
  font-size: 24px;
}

```

You can also make `title` a separate prop if you want every `Card` to always have a `#`

```

function Card({ children, title }) {
  return (
    <div className="card">
      <div className="card-content">
        <h1>{title}</h1>
        {children}
      </div>
    </div>
  );
}

export default function Profile() {
  return (
    <div>
      <Card title="Photo">
        
      </Card>
      <Card title="About">
        <p>Aklilu Lemma was a distinguished Ethiopian scientist who discovered a natural treatment to schistosomiasis.</p>
      </Card>
    </div>
  );
}

.card {
  width: fit-content;
  margin: 20px;
  padding: 20px;
  border: 1px solid #aaa;
  border-radius: 20px;
  background: #fff;
}
.card-content {
  text-align: center;
}

```

```
.avatar {  
  margin: 10px;  
  border-radius: 50%;  
}  
h1 {  
  margin: 5px;  
  padding: 0;  
  font-size: 24px;  
}
```

Preserving and Resetting State

State 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.

- How React "sees" component structures
- When React chooses to preserve or reset the state
- How to force React to reset component's state
- How keys and types affect whether the state is preserved

The UI tree {/the-ui-tree/}

Browsers use many tree structures to model UI. The [DOM](#) represents HTML elements, the [CSSOM](#) does the same for CSS. There's even an [Accessibility tree](#)!

React also uses tree structures to manage and model the UI you make. React makes **UI trees** from your JSX. Then React DOM updates the browser DOM elements to match that UI tree. (React Native translates these trees into elements specific to mobile platforms.)

From components, React creates a UI tree which React DOM uses to render the DOM

State is tied to a position in the tree {/state-is-tied-to-a-position-in-the-tree/}

When you give a component state, you might think the state "lives" inside the component. But the state is actually held inside React. React associates each piece of state it's holding with the correct component by where that component sits in the UI tree.

Here, there is only one `<Counter />` JSX tag, but it's rendered at two different positions:

```
import { 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>
    );
  }
}

```

```

label {
  display: block;
  clear: both;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
  float: left;
}

.hover {
  background: #ffffd8;
}

```

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:

```

import { 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>
  );
}

```

```

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
  float: left;
}

.hover {
  background: #ffffd8;
}

```

As you can see, when one counter is updated, only the state for that component is updated:

Updating state

React will keep the state around for as long as you render the same component at the same position. To see this, increment both counters, then remove the second component by unchecking "Render the second counter" checkbox, and then add it back by ticking it again:

```

import { 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)
          }}
        />
      </label>
    </div>
  );
}

```

```

        />
      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>
  );
}

```

```

label {
  display: block;
  clear: both;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
  float: left;
}

.hover {
  background: #ffffd8;
}

```

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 {/same-component-at-the-same-position-preserves-state/}

In this example, there are two different `<Counter />` tags:

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

export default function App() {
  const [isFancy, setIsFancy] = useState(false);
  return (
    <div>
      {isFancy ? (
        <Counter isFancy={true} />
      ) : (
        <Counter isFancy={false} />
      )}
      <label>
        <input
          type="checkbox"
          checked={isFancy}
          onChange={e => {
            setIsFancy(e.target.checked)
          }}
        />
        Use fancy styling
      </label>
    </div>
  );
}

function Counter({ isFancy }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }
  if (isFancy) {
    className += ' fancy';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    
```

```

        </div>
    );
}

label {
  display: block;
  clear: both;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
  float: left;
}

.fancy {
  border: 5px solid gold;
  color: #ff6767;
}

.hover {
  background: #ffffd8;
}

```

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.

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`:

```

import { 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>
  );
}

```

```

label {
  display: block;
  clear: both;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
  float: left;
}

.fancy {
  border: 5px solid gold;
  color: #ff6767;
}

```

```

}
.hover {
  background: #ffffd8;
}

```

You 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 *{/different-components-at-the-same-position-reset-state/}*

In this example, ticking the checkbox will replace `<Counter>` with a `<p>`:

```

import { 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>
);
}

```

```

label {
  display: block;
  clear: both;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
  float: left;
}

.hover {
  background: #ffffd8;
}

```

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:

```

import { useState } from 'react';

export default function App() {
  const [isFancy, setIsFancy] = useState(false);
  return (
    <div>
      {isFancy ? (
        <div>
          <Counter isFancy={true} />
        </div>
      ) : (
        <section>
          <Counter isFancy={false} />
        </section>
      )}
      <label>
        <input

```

```

        type="checkbox"
        checked={isFancy}
        onChange={e => {
          setIsFancy(e.target.checked)
        }}
      />
      Use fancy styling
    </label>
  </div>
);
}

function Counter({ isFancy }) {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
    className += ' hover';
  }
  if (isFancy) {
    className += ' fancy';
  }

  return (
    <div
      className={className}
      onPointerEnter={() => setHover(true)}
      onPointerLeave={() => setHover(false)}
    >
      <h1>{score}</h1>
      <button onClick={() => setScore(score + 1)}>
        Add one
      </button>
    </div>
  );
}

```

```

label {
  display: block;
  clear: both;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
  float: left;
}

.fancy {
  border: 5px solid gold;
  color: #ff6767;
}

.hover {
  background: #fffffd8;
}

```

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.

This is why you should not nest component function definitions.

Here, the `MyTextField` component function is defined *inside* `MyComponent`:

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

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 {/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:

```
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>
  );
}
```

```
h1 {
  font-size: 18px;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
```

```

padding: 20px;
margin: 0 20px 20px 0;
}

.hover {
  background: #ffffd8;
}

```

Currently, when you change the player, the score is preserved. The two `Counter`s 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:

1. Render components in different positions
2. Give each component an explicit identity with `key`

Option 1: Rendering a component in different positions {/option-1-rendering-a-component-in-different-positions/}

If you want these two `Counter`s to be independent, you can render them in two different positions:

```

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>
);
}

```

```

h1 {
  font-size: 18px;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
}

.hover {
  background: #ffffd8;
}

```

- 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 state

Clicking "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 `{/option-2-resetting-state-with-a-key/}`

There is also another, more generic, way to reset a component's state.

You might have seen `key`s 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:

```
import { 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>
  );
}
```

```
h1 {
  font-size: 18px;
}

.counter {
  width: 100px;
  text-align: center;
  border: 1px solid gray;
  border-radius: 4px;
  padding: 20px;
  margin: 0 20px 20px 0;
}
```

```
.hover {
  background: #ffffd8;
}
```

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.

Remember that keys are not globally unique. They only specify the position *within the parent*.

Resetting a form with a key {/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:

```
import { 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 = [
];

export default function ContactList({
  selectedContact,
  contacts,
  onSelect
}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map(contact =>
```

react

```
<li key={contact.id}>
  <button onClick={() => {
    onSelect(contact);
  }}>
    {contact.name}
  </button>
</li>
)
</ul>
</section>
);
}
```

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

export default function Chat({ contact }) {
  const [text, setText] = useState('');
  return (
    <section className="chat">
      <textarea
        value={text}
        placeholder={'Chat to ' + contact.name}
        onChange={e => setText(e.target.value)}
      />
      <br />
      <button>Send to {contact.email}</button>
    </section>
  );
}
```

```
.chat, .contact-list {
  float: left;
  margin-bottom: 20px;
}
ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}
```

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:

```
import { 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 = [
];
```

```
export default function ContactList({
  selectedContact,
  contacts,
  onSelect
}) {
  return (
    <section className="contact-list">
      <ul>
        {contacts.map(contact =>
          <li key={contact.id}>
            <button onClick={() => {
              onSelect(contact);
            }}>
              {contact.name}
            </button>
          </li>
        )}
      </ul>
    </section>
  );
}
```

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

export default function Chat({ contact }) {
  const [text, setText] = useState('');
  return (
    <section className="chat">
      <textarea
        value={text}
        placeholder={'Chat to ' + contact.name}
        onChange={e => setText(e.target.value)}
      />
      <br />
    </section>
  );
}
```

```

    <button>Send to {contact.email}</button>
  </section>
);
}

```

```

.chat, .contact-list {
  float: left;
  margin-bottom: 20px;
}
ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li button {
  width: 100px;
  padding: 10px;
  margin-right: 10px;
}
textarea {
  height: 150px;
}

```

Preserving state for removed components [{/preserving-state-for-removed-components/}](#)

In 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.

- 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.

Fix disappearing input text [{/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.

```

import { useState } from 'react';
export default function App() {

```

```

const [showHint, setShowHint] = useState(false);
if (showHint) {
  return (
    <div>
      <p><i>Hint: Your favorite city?</i></p>
      <Form />
      <button onClick={() => {
        setShowHint(false);
      }}>Hide hint</button>
    </div>
  );
}
return (
  <div>
    <Form />
    <button onClick={() => {
      setShowHint(true);
    }}>Show hint</button>
  </div>
);
}

function Form() {
  const [text, setText] = useState('');
  return (
    <textarea
      value={text}
      onChange={e => setText(e.target.value)}
    />
  );
}

```

`textarea { display: block; margin: 10px 0; }`

The problem is that `Form` is rendered in different positions. In the `if` branch, it is the second child of the `<div>`, but in the `else` branch, it is the first child. Therefore, the component type in each position changes. The first position changes between holding a `p` and a `Form`, while the second position changes between holding a `Form` and a `button`. React resets the state every time the component type changes.

The easiest solution is to unify the branches so that `Form` always renders in the same position:

```

import { useState } from 'react';

export default function App() {
  const [showHint, setShowHint] = useState(false);
  return (
    <div>
      {showHint &&
        <p><i>Hint: Your favorite city?</i></p>
      }
      <Form />
      {showHint ? (
        <button onClick={() => {
          setShowHint(false);
        }}>Hide hint</button>
      ) : (
        <button onClick={() => {
          setShowHint(true);
        }}>Show hint</button>
      )}
    </div>
  );
}

```

```

        )}
      </div>
    );
}

function Form() {
  const [text, setText] = useState('');
  return (
    <textarea
      value={text}
      onChange={e => setText(e.target.value)}
    />
  );
}

textarea { display: block; margin: 10px 0; }

```

Technically, you could also add `null` before `<Form />` in the `else` branch to match the `if` branch structure:

```

import { useState } from 'react';

export default function App() {
  const [showHint, setShowHint] = useState(false);
  if (showHint) {
    return (
      <div>
        <p><i>Hint: Your favorite city?</i></p>
        <Form />
        <button onClick={() => {
          setShowHint(false);
        }}>Hide hint</button>
      </div>
    );
  }
  return (
    <div>
      {null}
      <Form />
      <button onClick={() => {
        setShowHint(true);
      }}>Show hint</button>
    </div>
  );
}

function Form() {
  const [text, setText] = useState('');
  return (
    <textarea
      value={text}
      onChange={e => setText(e.target.value)}
    />
  );
}

textarea { display: block; margin: 10px 0; }

```

This way, `Form` is always the second child, so it stays in the same position and keeps its state. But this approach is much less obvious and introduces a risk that someone else will remove that `null`.

Swap two form fields {/swap-two-form-fields/}

This form lets you enter first and last name. It also has a checkbox controlling which field goes first. When you tick the checkbox, the "Last name" field will appear before the "First name" field.

It almost works, but there is a bug. If you fill in the "First name" input and tick the checkbox, the text will stay in the first input (which is now "Last name"). Fix it so that the input text *also* moves when you reverse the order.

It seems like for these fields, their position within the parent is not enough. Is there some way to tell React how to match up the state between re-renders?

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

export default function App() {
  const [reverse, setReverse] = useState(false);
  let checkbox = (
    <label>
      <input
        type="checkbox"
        checked={reverse}
        onChange={e => setReverse(e.target.checked)}
      />
      Reverse order
    </label>
  );
  if (reverse) {
    return (
      <>
        <Field label="Last name" />
        <Field label="First name" />
        {checkbox}
      </>
    );
  } else {
    return (
      <>
        <Field label="First name" />
        <Field label="Last name" />
        {checkbox}
      </>
    );
  }
}

function Field({ label }) {
  const [text, setText] = useState('');
  return (
    <label>
      {label}:{' '}
      <input
        type="text"
        value={text}
        placeholder={label}
        onChange={e => setText(e.target.value)}
      />
    </label>
  );
}
```

```

        />
      </label>
    );
}

```

```
label { display: block; margin: 10px 0; }
```

Give a `key` to both `<Field>` components in both `if` and `else` branches. This tells React how to "match up" the correct state for either `<Field>` even if their order within the parent changes:

```

import { useState } from 'react';

export default function App() {
  const [reverse, setReverse] = useState(false);
  let checkbox = (
    <label>
      <input
        type="checkbox"
        checked={reverse}
        onChange={e => setReverse(e.target.checked)}
      />
      Reverse order
    </label>
  );
  if (reverse) {
    return (
      <>
        <Field key="lastName" label="Last name" />
        <Field key="firstName" label="First name" />
        {checkbox}
      </>
    );
  } else {
    return (
      <>
        <Field key="firstName" label="First name" />
        <Field key="lastName" label="Last name" />
        {checkbox}
      </>
    );
  }
}

function Field({ label }) {
  const [text, setText] = useState('');
  return (
    <label>
      {label}:{' '}
      <input
        type="text"
        value={text}
        placeholder={label}
        onChange={e => setText(e.target.value)}
      />
    </label>
  );
}

```

```
label { display: block; margin: 10px 0; }
```

Reset a detail form *{/reset-a-detail-form/}*

This is an editable contact list. You can edit the selected contact's details and then either press "Save" to update it, or "Reset" to undo your changes.

When you select a different contact (for example, Alice), the state updates but the form keeps showing the previous contact's details. Fix it so that the form gets reset when the selected contact changes.

```
import { useState } from 'react';
import ContactList from './ContactList.js';
import EditContact from './EditContact.js';

export default function ContactManager() {
  const [
    contacts,
    setContacts
  ] = useState(initialContacts);
  const [
    selectedId,
    setSelectedId
  ] = useState(0);
  const selectedContact = contacts.find(c =>
    c.id === selectedId
  );

  function handleSave(updatedData) {
    const nextContacts = contacts.map(c => {
      if (c.id === updatedData.id) {
        return updatedData;
      } else {
        return c;
      }
    });
    setContacts(nextContacts);
  }

  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedId={selectedId}
        onSelect={id => setSelectedId(id)}
      />
      <hr />
      <EditContact
        initialData={selectedContact}
        onSave={handleSave}
      />
    </div>
  )
}

const initialContacts = [
];

export default function ContactList({
  contacts,
  selectedId,
  onSelect
}) {
```

```

    return (
      <section>
        <ul>
          {contacts.map(contact =>
            <li key={contact.id}>
              <button onClick={() => {
                onSelect(contact.id);
              }}>
                {contact.id === selectedId ?
                  <b>{contact.name}</b> :
                  contact.name
                }
              </button>
            </li>
          )}
        </ul>
      </section>
    );
}

```

```

import { useState } from 'react';

export default function EditContact({ initialValue, onSave }) {
  const [name, setName] = useState(initialValue.name);
  const [email, setEmail] = useState(initialValue.email);
  return (
    <section>
      <label>
        Name:{' '}
        <input
          type="text"
          value={name}
          onChange={e => setName(e.target.value)}
        />
      </label>
      <label>
        Email:{' '}
        <input
          type="email"
          value={email}
          onChange={e => setEmail(e.target.value)}
        />
      </label>
      <button onClick={() => {
        const updatedData = {
          name: name,
          email: email
        };
        onSave(updatedData);
      }}>
        Save
      </button>
      <button onClick={() => {
        setName(initialValue.name);
        setEmail(initialValue.email);
      }}>
        Reset
      </button>
    </section>
  );
}

```

```

ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li { display: inline-block; }
li button {
  padding: 10px;
}
label {
  display: block;
  margin: 10px 0;
}
button {
  margin-right: 10px;
  margin-bottom: 10px;
}

```

Give `key={selectedId}` to the `EditContact` component. This way, switching between different contacts will reset the form:

```

import { useState } from 'react';
import ContactList from './ContactList.js';
import EditContact from './EditContact.js';

export default function ContactManager() {
  const [
    contacts,
    setContacts
  ] = useState(initialContacts);
  const [
    selectedId,
    setSelectedId
  ] = useState(0);
  const selectedContact = contacts.find(c =>
    c.id === selectedId
  );

  function handleSave(updatedData) {
    const nextContacts = contacts.map(c => {
      if (c.id === updatedData.id) {
        return updatedData;
      } else {
        return c;
      }
    });
    setContacts(nextContacts);
  }

  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedId={selectedId}
        onSelect={id => setSelectedId(id)}
      />
      <hr />
      <EditContact
        key={selectedId}
        initialData={selectedContact}
        onSave={handleSave}
      >
    </div>
  );
}

```

```

        />
      </div>
    )
}

const initialContacts = [
];

export default function ContactList({
  contacts,
  selectedId,
  onSelect
}) {
  return (
    <section>
      <ul>
        {contacts.map(contact =>
          <li key={contact.id}>
            <button onClick={() => {
              onSelect(contact.id);
            }}>
              {contact.id === selectedId ?
                <b>{contact.name}</b> :
                contact.name
              }
            </button>
          </li>
        )}
      </ul>
    </section>
  );
}

```

```

import { useState } from 'react';

export default function EditContact({ initialValue, onSave }) {
  const [name, setName] = useState(initialValue.name);
  const [email, setEmail] = useState(initialValue.email);
  return (
    <section>
      <label>
        Name:{' '}
        <input
          type="text"
          value={name}
          onChange={e => setName(e.target.value)}
        />
      </label>
      <label>
        Email:{' '}
        <input
          type="email"
          value={email}
          onChange={e => setEmail(e.target.value)}
        />
      </label>
      <button onClick={() => {
        const updatedData = {
          name: name,
          email: email
        };
        onSave(updatedData);
      }}>
        Save
      </button>
    </section>
  );
}

```

```

    }>
    Save
  </button>
  <button onClick={() => {
    setName(initialData.name);
    setEmail(initialData.email);
  }}>
    Reset
  </button>
</section>
);
}

```

```

ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li { display: inline-block; }
li button {
  padding: 10px;
}
label {
  display: block;
  margin: 10px 0;
}
button {
  margin-right: 10px;
  margin-bottom: 10px;
}

```

Clear an image while it's loading [{/clear-an-image-while-its-loading/}](#)

When you press "Next", the browser starts loading the next image. However, because it's displayed in the same `` tag, by default you would still see the previous image until the next one loads. This may be undesirable if it's important for the text to always match the image. Change it so that the moment you press "Next", the previous image immediately clears.

Is there a way to tell React to re-create the DOM instead of reusing it?

```

import { useState } from 'react';

export default function Gallery() {
  const [index, setIndex] = useState(0);
  const hasNext = index < images.length - 1;

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

  let image = images[index];
  return (
    <>
      <button onClick={handleClick}>
        Next
    </button>
  );
}

```

```

        </button>
      <h3>
        Image {index + 1} of {images.length}
      </h3>
      <img src={image.src} />
      <p>
        {image.place}
      </p>
    </>
  );
}

let images = [
  {
    place: 'Penang, Malaysia',
    src: 'https://i.imgur.com/FJeJR8M.jpg'
  },
  {
    place: 'Lisbon, Portugal',
    src: 'https://i.imgur.com/dB2LRbj.jpg'
  },
  {
    place: 'Bilbao, Spain',
    src: 'https://i.imgur.com/z08o2TS.jpg'
  },
  {
    place: 'Valparaíso, Chile',
    src: 'https://i.imgur.com/Y3utgTi.jpg'
  },
  {
    place: 'Schwyz, Switzerland',
    src: 'https://i.imgur.com/JBbMpWY.jpg'
  },
  {
    place: 'Prague, Czechia',
    src: 'https://i.imgur.com/QwUKKmF.jpg'
  },
  {
    place: 'Ljubljana, Slovenia',
    src: 'https://i.imgur.com/3aIiwm.jpg'
  }
];

```

```
img { width: 150px; height: 150px; }
```

You can provide a `key` to the `` tag. When that `key` changes, React will re-create the `` DOM node from scratch. This causes a brief flash when each image loads, so it's not something you'd want to do for every image in your app. But it makes sense if you want to ensure the image always matches the text.

```

import { useState } from 'react';

export default function Gallery() {
  const [index, setIndex] = useState(0);
  const hasNext = index < images.length - 1;

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

  let image = images[index];
  return (
    <>
      <button onClick={handleClick}>
        Next
    </button>
  );
}

```

```

        </button>
      <h3>
        Image {index + 1} of {images.length}
      </h3>
      <img key={image.src} src={image.src} />
      <p>
        {image.place}
      </p>
    </>
  );
}

let images = [
  {
    place: 'Penang, Malaysia',
    src: 'https://i.imgur.com/FJeJR8M.jpg'
  },
  {
    place: 'Lisbon, Portugal',
    src: 'https://i.imgur.com/dB2LRbj.jpg'
  },
  {
    place: 'Bilbao, Spain',
    src: 'https://i.imgur.com/z08o2TS.jpg'
  },
  {
    place: 'Valparaíso, Chile',
    src: 'https://i.imgur.com/Y3utgTi.jpg'
  },
  {
    place: 'Schwyz, Switzerland',
    src: 'https://i.imgur.com/JBbMpWY.jpg'
  },
  {
    place: 'Prague, Czechia',
    src: 'https://i.imgur.com/QwUKKmF.jpg'
  },
  {
    place: 'Ljubljana, Slovenia',
    src: 'https://i.imgur.com/3aIiwm.jpg'
  }
];

```

```
img { width: 150px; height: 150px; }
```

Fix misplaced state in the list {/fix-misplaced-state-in-the-list/}

In this list, each `Contact` has state that determines whether "Show email" has been pressed for it. Press "Show email" for Alice, and then tick the "Show in reverse order" checkbox. You will notice that it's *Taylor's* email that is expanded now, but Alice's--which has moved to the bottom--appears collapsed.

Fix it so that the expanded state is associated with each contact, regardless of the chosen ordering.

```

import { useState } from 'react';
import Contact from './Contact.js';

export default function ContactList() {
  const [reverse, setReverse] = useState(false);

  const displayedContacts = [...contacts];
  if (reverse) {
    displayedContacts.reverse();
  }

  return (
    <>
      <label>
        <input

```

```

        type="checkbox"
        value={reverse}
        onChange={e => {
          setReverse(e.target.checked)
        }}
      />{' '}
    Show in reverse order
  </label>
  <ul>
    {displayedContacts.map((contact, i) =>
      <li key={i}>
        <Contact contact={contact} />
      </li>
    )}
  </ul>
</>
);
}

const contacts = [
];

```

```

import { useState } from 'react';

export default function Contact({ contact }) {
  const [expanded, setExpanded] = useState(false);
  return (
    <>
      <p><b>{contact.name}</b></p>
      {expanded &&
        <p><i>{contact.email}</i></p>
      }
      <button onClick={() => {
        setExpanded(!expanded);
      }}>
        {expanded ? 'Hide' : 'Show'} email
      </button>
    </>
  );
}

```

```

ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li {
  margin-bottom: 20px;
}
label {
  display: block;
  margin: 10px 0;
}
button {
  margin-right: 10px;
  margin-bottom: 10px;
}

```

The problem is that this example was using index as a `key`:

```
{displayedContacts.map((contact, i) =>
  <li key={i}>
```

However, you want the state to be associated with *each particular contact*.

Using the contact ID as a `key` instead fixes the issue:

```
import { useState } from 'react';
import Contact from './Contact.js';

export default function ContactList() {
  const [reverse, setReverse] = useState(false);

  const displayedContacts = [...contacts];
  if (reverse) {
    displayedContacts.reverse();
  }

  return (
    <>
      <label>
        <input
          type="checkbox"
          value={reverse}
          onChange={e => {
            setReverse(e.target.checked)
          }}
        />{' '}
        Show in reverse order
      </label>
      <ul>
        {displayedContacts.map(contact =>
          <li key={contact.id}>
            <Contact contact={contact} />
          </li>
        )}
      </ul>
    </>
  );
}

const contacts = [
];
```

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

export default function Contact({ contact }) {
  const [expanded, setExpanded] = useState(false);
  return (
    <>
      <p><b>{contact.name}</b></p>
      {expanded &&
        <p><i>{contact.email}</i></p>
      }
      <button onClick={() => {
        setExpanded(!expanded);
      }}>
        {expanded ? 'Hide' : 'Show'} email
      </button>
    </>
  );
}
```

```
        </>
    );
}

ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li {
  margin-bottom: 20px;
}
label {
  display: block;
  margin: 10px 0;
}
button {
  margin-right: 10px;
  margin-bottom: 10px;
}
```

State is associated with the tree position. A `key` lets you specify a named position instead of relying on order.

Queueing a Series of State Updates

Setting 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.

- 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 { /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:

```
import { 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>
    </>
  )
}

button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

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.

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 {/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:

```
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>
    </>
  )
}

button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

Here, `n => n + 1` is called an **update function**. When you pass it to a state setter:

1. React queues this function to be processed after all the other code in the event handler has run.
2. 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:

1. `setNumber(n => n + 1)`: `n => n + 1` is a function. React adds it to a queue.
2. `setNumber(n => n + 1)`: `n => n + 1` is a function. React adds it to a queue.
3. `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 update	n	returns
<code>n => n + 1</code>	<code>0</code>	<code>0 + 1 = 1</code>
<code>n => n + 1</code>	<code>1</code>	<code>1 + 1 = 2</code>
<code>n => n + 1</code>	<code>2</code>	<code>2 + 1 = 3</code>

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-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);
}}>
```

```
import { 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>
    </>
  )
}
```

```
button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

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

1. `setNumber(number + 5)`: `number` is `0`, so `setNumber(0 + 5)`. React adds "replace with `5`" to its queue.
2. `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 update	n	returns
"replace with 5"	0 (unused)	5
<code>n => n + 1</code>	5	<code>5 + 1 = 6</code>

React stores 6 as the final result and returns it from `useState`.

You 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 {/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);
}}>

import { 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>
    </>
  )
}

button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

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

1. `setNumber(number + 5)`: `number` is 0, so `setNumber(0 + 5)`. React adds "replace with 5" to its queue.
2. `setNumber(n => n + 1)`: `n => n + 1` is an updater function. React adds that function to its queue.
3. `setNumber(42)`: React adds "replace with 42" to its queue.

During the next render, React goes through the state queue:

queued update	n	returns
"replace with 5"	0 (unused)	5
n => n + 1	5	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 {/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)`.

- 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.

Fix a request counter {/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.

```
import { useState } from 'react';
export default function RequestTracker() {
```

```

const [pending, setPending] = useState(0);
const [completed, setCompleted] = useState(0);

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

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

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

```

Inside the `handleClick` event handler, the values of `pending` and `completed` correspond to what they were at the time of the click event. For the first render, `pending` was `0`, so `setPending(pending - 1)` becomes `setPending(-1)`, which is wrong. Since you want to *increment* or *decrement* the counters, rather than set them to a concrete value determined during the click, you can instead pass the updater functions:

```

import { useState } from 'react';

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

  async function handleClick() {
    setPending(p => p + 1);
    await delay(3000);
    setPending(p => p - 1);
    setCompleted(c => c + 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);
  });
}

```

This ensures that when you increment or decrement a counter, you do it in relation to its *latest* state rather than what the state was at the time of the click.

Implement the state queue yourself {/implement-the-state-queue-yourself/}

In this challenge, you will reimplement a tiny part of React from scratch! It's not as hard as it sounds.

Scroll through the sandbox preview. Notice that it shows **four test cases**. They correspond to the examples you've seen earlier on this page. Your task is to implement the `getFinalState` function so that it returns the correct result for each of those cases. If you implement it correctly, all four tests should pass.

You will receive two arguments: `baseState` is the initial state (like `0`), and the `queue` is an array which contains a mix of numbers (like `5`) and updater functions (like `n => n + 1`) in the order they were added.

Your task is to return the final state, just like the tables on this page show!

If you're feeling stuck, start with this code structure:

```

export function getFinalState(baseState, queue) {
  let finalState = baseState;

  for (let update of queue) {
    if (typeof update === 'function') {
      // TODO: apply the updater function
    } else {
      // TODO: replace the state
    }
  }

  return finalState;
}

```

Fill out the missing lines!

```

export function getFinalState(baseState, queue) {
  let finalState = baseState;

  // TODO: do something with the queue...

  return finalState;
}

```

```

import { getFinalState } from './processQueue.js';

function increment(n) {
  return n + 1;
}

```

```

increment.toString = () => 'n => n+1';

export default function App() {
  return (
    <>
      <TestCase
        baseState={0}
        queue={[1, 1, 1]}
        expected={1}
      />
      <hr />
      <TestCase
        baseState={0}
        queue={[
          increment,
          increment,
          increment
        ]}
        expected={3}
      />
      <hr />
      <TestCase
        baseState={0}
        queue={[
          5,
          increment,
        ]}
        expected={6}
      />
      <hr />
      <TestCase
        baseState={0}
        queue={[
          5,
          increment,
          42,
        ]}
        expected={42}
      />
    </>
  );
}

function TestCase({
  baseState,
  queue,
  expected
}) {
  const actual = getFinalState(baseState, queue);
  return (
    <>
      <p>Base state: <b>{baseState}</b></p>
      <p>Queue: <b>[{queue.join(', ')}]</b></p>
      <p>Expected result: <b>{expected}</b></p>
      <p style={{
        color: actual === expected ?
          'green' :
          'red'
      }}>
        Your result: <b>{actual}</b>
        {` `}
        ({actual === expected ?

```

```

        'correct' :
        'wrong'
    })
</p>
</>
);
}

```

This is the exact algorithm described on this page that React uses to calculate the final state:

```

export function getFinalState(baseState, queue) {
  let finalState = baseState;

  for (let update of queue) {
    if (typeof update === 'function') {
      // Apply the updater function.
      finalState = update(finalState);
    } else {
      // Replace the next state.
      finalState = update;
    }
  }

  return finalState;
}

```

```

import { getFinalState } from './processQueue.js';

function increment(n) {
  return n + 1;
}
increment.toString = () => 'n => n+1';

export default function App() {
  return (
    <>
      <TestCase
        baseState={0}
        queue={[1, 1, 1]}
        expected={1}>
      />
      <hr />
      <TestCase
        baseState={0}
        queue={[increment,
          increment,
          increment
        ]}
        expected={3}>
      />
      <hr />
      <TestCase
        baseState={0}
        queue={[5,
          increment,
        ]}
        expected={6}>
      />
      <hr />
    </>
  )
}

```

```
<TestCase
  baseState={0}
  queue={[
    5,
    increment,
    42,
  ]}
  expected={42}
/>
</>
);
}

function TestCase({
  baseState,
  queue,
  expected
}) {
  const actual = getFinalState(baseState, queue);
  return (
    <>
      <p>Base state: <b>{baseState}</b></p>
      <p>Queue: <b>[{queue.join(', ')}]</b></p>
      <p>Expected result: <b>{expected}</b></p>
      <p style={{
        color: actual === expected ?
          'green' :
          'red'
      }}>
        Your result: <b>{actual}</b>
        {` `}
        ({actual === expected ?
          'correct' :
          'wrong'
        })
      </p>
    );
}
```

Now you know how this part of React works!

React Developer Tools

Use React Developer Tools to inspect React [components](#), edit [props](#) and [state](#), and identify performance problems.

- How to install React Developer Tools

Browser extension {/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.

 React Developer Tools extension

React Developer Tools extension

Safari and other browsers {/safari-and-other-browsers/}

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

```
# Yarn
yarn global add react-devtools

# Npm
npm 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.

 React Developer Tools standalone

React Developer Tools standalone

Mobile (React Native) {/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:

```
# Yarn  
yarn global add react-devtools  
  
# Npm  
npm 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.](#)

[Go to TOC](#)

Quick Start

Welcome 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.

- 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 {/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:

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

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

```

<div>
  <h1>Welcome to my app</h1>
  <MyButton />
</div>
);
}

```

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 {/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 {/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 { /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](#):

```
const user = {
  name: 'Hedy Lamarr',
  imageUrl: 'https://i.imgur.com/yX0vd0Ss.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
        }}
      />
    </>
  );
}
```

```
.avatar {
  border-radius: 50%;
}

.large {
  border: 4px solid gold;
}
```

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 {/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 {/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 = [
```

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.

```
const products = [];

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>
  );
}
```

Responding to events *{/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 { /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:

```
import { 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>
);
}

button {
  display: block;
  margin-bottom: 5px;
}

```

Notice how each button "remembers" its own `count` state and doesn't affect other buttons.

Using Hooks {/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 {/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 `0`

The 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 children

On 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.

```
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>
  );
}

button {
  display: block;
  margin-bottom: 5px;
}
```

Next Steps {/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.

Reacting to Input with State

React 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.

- 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 { /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.

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](#):

```
async 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;

```

```
{
  "hardReloadOnChange": true
}
```

```

<form id="form">
  <h2>City quiz</h2>
  <p>
    What city is located on two continents?
  </p>
  <textarea id="textarea"></textarea>
  <br />
  <button id="button" disabled>Submit</button>
  <p id="loading" style="display: none">Loading...</p>
  <p id="error" style="display: none; color: red;"></p>

```

```
</form>
<h1 id="success" style="display: none">That's right!</h1>

<style>
* { box-sizing: border-box; }
body { font-family: sans-serif; margin: 20px; padding: 0; }
</style>
```

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!

Thinking about UI declaratively {/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:

1. **Identify** your component's different visual states
2. **Determine** what triggers those state changes
3. **Represent** the state in memory using `useState`
4. **Remove** any non-essential state variables
5. **Connect** the event handlers to set the state

Step 1: Identify your component's different visual states {/step-1-identify-your-components-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'`:

```
export 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>
    </>
  )
}
```

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:

```
export 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>
</>
);
}

```

```
.Error { color: red; }
```

Displaying many visual states at once *{/displaying-many-visual-states-at-once/}*

If a component has a lot of visual states, it can be convenient to show them all on one page:

```

import 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>
      ))}
    </>
  );
}

export default function Form({ status }) {
  if (status === 'success') {
    return <h1>That's right!</h1>
  }
  return (
    <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>
  );
}

```

```
section { border-bottom: 1px solid #aaa; padding: 20px; }
h4 { color: #222; }
body { margin: 0; }
.Error { color: red; }
```

Pages like this are often called "living styleguides" or "storybooks".

Step 2: Determine what triggers those state changes {/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.

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.

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` {/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 {/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.

Eliminating "impossible" states with a reducer {/eliminating-impossible-states-with-a-reducer/}

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 {/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:

```
import { 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);
  });
}

.Error { color: red; }

```

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.

- Declarative programming means describing the UI for each visual state rather than micromanaging the UI (imperative).
- When developing a component:
 1. Identify all its visual states.
 2. Determine the human and computer triggers for state changes.
 3. Model the state with `useState`.
 4. Remove non-essential state to avoid bugs and paradoxes.
 5. Connect the event handlers to set state.

Add and remove a CSS class { /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.

```

export default function Picture() {
  return (
    <div className="background background--active">
      
    </div>
  );
}

```

```

body { margin: 0; padding: 0; height: 250px; }

.background {
  width: 100vw;
  height: 100vh;
  display: flex;
  justify-content: center;
  align-items: center;
  background: #eee;
}

.background--active {
  background: #a6b5ff;
}

.picture {
  width: 200px;
  height: 200px;
  border-radius: 10px;
}

.picture--active {
  border: 5px solid #a6b5ff;
}

```

This component has two visual states: when the image is active, and when the image is inactive:

- When the image is active, the CSS classes are `background` and `picture picture--active`.
- When the image is inactive, the CSS classes are `background background--active` and `picture`.

A single boolean state variable is enough to remember whether the image is active. The original task was to remove or add CSS classes. However, in React you need to *describe* what you want to see rather than *manipulate* the UI elements. So you need to calculate both CSS classes based on the current state. You also need to [stop the propagation](#) so that clicking the image doesn't register as a click on the background.

Verify that this version works by clicking the image and then outside of it:

```

import { useState } from 'react';

export default function Picture() {
  const [isActive, setIsActive] = useState(false);

  let backgroundClassName = 'background';
  let pictureClassName = 'picture';
  if (isActive) {
    pictureClassName += ' picture--active';
  } else {
    backgroundClassName += ' background--active';
  }

  return (
    <div
      className={backgroundClassName}
      onClick={() => setIsActive(false)}
    >
      <img
        onClick={e => {
          e.stopPropagation();
          setIsActive(true);
        }}
      />
    
```

```

        }
        className={pictureClassName}
        alt="Rainbow houses in Kampung Pelangi, Indonesia"
        src="https://i.imgur.com/5qwVYb1.jpeg"
      />
    </div>
  );
}

```

```

body { margin: 0; padding: 0; height: 250px; }

.background {
  width: 100vw;
  height: 100vh;
  display: flex;
  justify-content: center;
  align-items: center;
  background: #eee;
}

.background--active {
  background: #a6b5ff;
}

.picture {
  width: 200px;
  height: 200px;
  border-radius: 10px;
  border: 5px solid transparent;
}

.picture--active {
  border: 5px solid #a6b5ff;
}

```

Alternatively, you could return two separate chunks of JSX:

```

import { useState } from 'react';

export default function Picture() {
  const [isActive, setIsActive] = useState(false);
  if (isActive) {
    return (
      <div
        className="background"
        onClick={() => setIsActive(false)}
      >
         e.stopPropagation()}
        />
      </div>
    );
  }
  return (
    <div className="background background--active">
      <img
        className="picture"
        alt="Rainbow houses in Kampung Pelangi, Indonesia"
      />
    </div>
  );
}

```

```

        src="https://i.imgur.com/5qwVYb1.jpeg"
        onClick={() => setIsActive(true)}
      />
    </div>
  );
}

```

```

body { margin: 0; padding: 0; height: 250px; }

.background {
  width: 100vw;
  height: 100vh;
  display: flex;
  justify-content: center;
  align-items: center;
  background: #eee;
}

.background--active {
  background: #a6b5ff;
}

.picture {
  width: 200px;
  height: 200px;
  border-radius: 10px;
  border: 5px solid transparent;
}

.picture--active {
  border: 5px solid #a6b5ff;
}

```

Keep in mind that if two different JSX chunks describe the same tree, their nesting (first `<div>` → first ``) has to line up. Otherwise, toggling `isActive` would recreate the whole tree below and **reset its state**. This is why, if a similar JSX tree gets returned in both cases, it is better to write them as a single piece of JSX.

Profile editor `{/profile-editor/}`

Here is a small form implemented with plain JavaScript and DOM. Play with it to understand its behavior:

```

function handleFormSubmit(e) {
  e.preventDefault();
  if (editButton.textContent === 'Edit Profile') {
    editButton.textContent = 'Save Profile';
    hide(firstNameText);
    hide(lastNameText);
    show(firstNameInput);
    show(lastNameInput);
  } else {
    editButton.textContent = 'Edit Profile';
    hide(firstNameInput);
    hide(lastNameInput);
    show(firstNameText);
    show(lastNameText);
  }
}

```

react

```
function handleFirstNameChange() {
  firstNameText.textContent = firstNameInput.value;
  helloText.textContent = (
    'Hello ' +
    firstNameInput.value + ' ' +
    lastNameInput.value + '!'
  );
}

function handleLastNameChange() {
  lastNameText.textContent = lastNameInput.value;
  helloText.textContent = (
    'Hello ' +
    firstNameInput.value + ' ' +
    lastNameInput.value + '!'
  );
}

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

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

let form = document.getElementById('form');
let editButton = document.getElementById('editButton');
let firstNameInput = document.getElementById('firstNameInput');
let firstNameText = document.getElementById('firstNameText');
let lastNameInput = document.getElementById('lastNameInput');
let lastNameText = document.getElementById('lastNameText');
let helloText = document.getElementById('helloText');
form.onsubmit = handleFormSubmit;
firstNameInput.oninput = handleFirstNameChange;
lastNameInput.oninput = handleLastNameChange;
```

```
{
  "hardReloadOnChange": true
}
```

```
<form id="form">
  <label>
    First name:
    <b id="firstNameText">Jane</b>
    <input
      id="firstNameInput"
      value="Jane"
      style="display: none">
  </label>
  <label>
    Last name:
    <b id="lastNameText">Jacobs</b>
    <input
      id="lastNameInput"
      value="Jacobs"
      style="display: none">
  </label>
  <button type="submit" id="editButton">Edit Profile</button>
  <p><i id="helloText">Hello, Jane Jacobs!</i></p>
</form>
```

```
<style>
* { box-sizing: border-box; }
body { font-family: sans-serif; margin: 20px; padding: 0; }
label { display: block; margin-bottom: 20px; }
</style>
```

This form switches between two modes: in the editing mode, you see the inputs, and in the viewing mode, you only see the result. The button label changes between "Edit" and "Save" depending on the mode you're in. When you change the inputs, the welcome message at the bottom updates in real time.

Your task is to reimplement it in React in the sandbox below. For your convenience, the markup was already converted to JSX, but you'll need to make it show and hide the inputs like the original does.

Make sure that it updates the text at the bottom, too!

```
export default function EditProfile() {
  return (
    <form>
      <label>
        First name:{' '}
        <b>Jane</b>
        <input />
      </label>
      <label>
        Last name:{' '}
        <b>Jacobs</b>
        <input />
      </label>
      <button type="submit">
        Edit Profile
      </button>
      <p><i>Hello, Jane Jacobs!</i></p>
    </form>
  );
}
```

```
label { display: block; margin-bottom: 20px; }
```

You will need two state variables to hold the input values: `firstName` and `lastName`. You're also going to need an `isEditing` state variable that holds whether to display the inputs or not. You should *not* need a `fullName` variable because the full name can always be calculated from the `firstName` and the `lastName`.

Finally, you should use `conditional rendering` to show or hide the inputs depending on `isEditing`.

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

export default function EditProfile() {
  const [isEditing, setIsEditing] = useState(false);
  const [firstName, setFirstName] = useState('Jane');
  const [lastName, setLastName] = useState('Jacobs');

  return (
    <form onSubmit={e => {
      e.preventDefault();
      setIsEditing(!isEditing);
    }}>
```

```

<label>
  First name:{' '}
  {isEditing ? (
    <input
      value={firstName}
      onChange={e => {
        setFirstName(e.target.value)
      }}
    />
  ) : (
    <b>{firstName}</b>
  )}
</label>
<label>
  Last name:{' '}
  {isEditing ? (
    <input
      value={lastName}
      onChange={e => {
        setLastName(e.target.value)
      }}
    />
  ) : (
    <b>{lastName}</b>
  )}
</label>
<button type="submit">
  {isEditing ? 'Save' : 'Edit'} Profile
</button>
<p><i>Hello, {firstName} {lastName}!</i></p>
</form>
);
}

```

```
label { display: block; margin-bottom: 20px; }
```

Compare this solution to the original imperative code. How are they different?

Refactor the imperative solution without React {/refactor-the-imperative-solution-without-react/}

Here is the original sandbox from the previous challenge, written imperatively without React:

```

function handleFormSubmit(e) {
  e.preventDefault();
  if (editButton.textContent === 'Edit Profile') {
    editButton.textContent = 'Save Profile';
    hide(firstNameText);
    hide(lastNameText);
    show(firstNameInput);
    show(lastNameInput);
  } else {
    editButton.textContent = 'Edit Profile';
    hide(firstNameInput);
    hide(lastNameInput);
    show(firstNameText);
    show(lastNameText);
  }
}

```

```

function handleFirstNameChange() {
  firstNameText.textContent = firstNameInput.value;
  helloText.textContent = (
    'Hello ' +
    firstNameInput.value + ' ' +
    lastNameInput.value + '!'
  );
}

function handleLastNameChange() {
  lastNameText.textContent = lastNameInput.value;
  helloText.textContent = (
    'Hello ' +
    firstNameInput.value + ' ' +
    lastNameInput.value + '!'
  );
}

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

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

let form = document.getElementById('form');
let editButton = document.getElementById('editButton');
let firstNameInput = document.getElementById('firstNameInput');
let firstNameText = document.getElementById('firstNameText');
let lastNameInput = document.getElementById('lastNameInput');
let lastNameText = document.getElementById('lastNameText');
let helloText = document.getElementById('helloText');
form.onsubmit = handleFormSubmit;
firstNameInput.oninput = handleFirstNameChange;
lastNameInput.oninput = handleLastNameChange;

```

```
{
  "hardReloadOnChange": true
}
```

```

<form id="form">
  <label>
    First name:
    <b id="firstNameText">Jane</b>
    <input
      id="firstNameInput"
      value="Jane"
      style="display: none">
  </label>
  <label>
    Last name:
    <b id="lastNameText">Jacobs</b>
    <input
      id="lastNameInput"
      value="Jacobs"
      style="display: none">
  </label>
  <button type="submit" id="editButton">Edit Profile</button>
  <p><i id="helloText">Hello, Jane Jacobs!</i></p>
</form>

```

```
<style>
* { box-sizing: border-box; }
body { font-family: sans-serif; margin: 20px; padding: 0; }
label { display: block; margin-bottom: 20px; }
</style>
```

Imagine React didn't exist. Can you refactor this code in a way that makes the logic less fragile and more similar to the React version? What would it look like if the state was explicit, like in React?

If you're struggling to think where to start, the stub below already has most of the structure in place. If you start here, fill in the missing logic in the `updateDOM` function. (Refer to the original code where needed.)

```
let firstName = 'Jane';
let lastName = 'Jacobs';
let isEditing = false;

function handleFormSubmit(e) {
  e.preventDefault();
  setIsEditing(!isEditing);
}

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

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

function setFirstName(value) {
  firstName = value;
  updateDOM();
}

function setLastName(value) {
  lastName = value;
  updateDOM();
}

function setIsEditing(value) {
  isEditing = value;
  updateDOM();
}

function updateDOM() {
  if (isEditing) {
    editButton.textContent = 'Save Profile';
    // TODO: show inputs, hide content
  } else {
    editButton.textContent = 'Edit Profile';
    // TODO: hide inputs, show content
  }
  // TODO: update text labels
}

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

function show(el) {
```

```

el.style.display = '';
}

let form = document.getElementById('form');
let editButton = document.getElementById('editButton');
let firstNameInput = document.getElementById('firstNameInput');
let firstNameText = document.getElementById('firstNameText');
let lastNameInput = document.getElementById('lastNameInput');
let lastNameText = document.getElementById('lastNameText');
let helloText = document.getElementById('helloText');
form.onsubmit = handleFormSubmit;
firstNameInput.oninput = handleFirstNameChange;
lastNameInput.oninput = handleLastNameChange;

{
  "hardReloadOnChange": true
}

<form id="form">
  <label>
    First name:
    <b id="firstNameText">Jane</b>
    <input
      id="firstNameInput"
      value="Jane"
      style="display: none">
  </label>
  <label>
    Last name:
    <b id="lastNameText">Jacobs</b>
    <input
      id="lastNameInput"
      value="Jacobs"
      style="display: none">
  </label>
  <button type="submit" id="editButton">Edit Profile</button>
  <p><i id="helloText">Hello, Jane Jacobs!</i></p>
</form>

<style>
* { box-sizing: border-box; }
body { font-family: sans-serif; margin: 20px; padding: 0; }
label { display: block; margin-bottom: 20px; }
</style>

```

The missing logic included toggling the display of inputs and content, and updating the labels:

```

let firstName = 'Jane';
let lastName = 'Jacobs';
let isEditing = false;

function handleFormSubmit(e) {
  e.preventDefault();
  setIsEditing(!isEditing);
}

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

function handleLastNameChange(e) {

```

```
    setLastName(e.target.value);
}

function setFirstName(value) {
  firstName = value;
  updateDOM();
}

function setLastName(value) {
  lastName = value;
  updateDOM();
}

function setIsEditing(value) {
  isEditing = value;
  updateDOM();
}

function updateDOM() {
  if (isEditing) {
    editButton.textContent = 'Save Profile';
    hide(firstNameText);
    hide(lastNameText);
    show(firstNameInput);
    show(lastNameInput);
  } else {
    editButton.textContent = 'Edit Profile';
    hide(firstNameInput);
    hide(lastNameInput);
    show(firstNameText);
    show(lastNameText);
  }
  firstNameText.textContent = firstName;
  lastNameText.textContent = lastName;
  helloText.textContent = (
    'Hello ' +
    firstName + ' ' +
    lastName + '!'
);
}

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

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

let form = document.getElementById('form');
let editButton = document.getElementById('editButton');
let firstNameInput = document.getElementById('firstNameInput');
let firstNameText = document.getElementById('firstNameText');
let lastNameInput = document.getElementById('lastNameInput');
let lastNameText = document.getElementById('lastNameText');
let helloText = document.getElementById('helloText');
form.onsubmit = handleFormSubmit;
firstNameInput.oninput = handleFirstNameChange;
lastNameInput.oninput = handleLastNameChange;
```

```
{
  "hardReloadOnChange": true
}
```

```
<form id="form">
  <label>
    First name:
    <b id="firstNameText">Jane</b>
    <input
      id="firstNameInput"
      value="Jane"
      style="display: none">
  </label>
  <label>
    Last name:
    <b id="lastNameText">Jacobs</b>
    <input
      id="lastNameInput"
      value="Jacobs"
      style="display: none">
  </label>
  <button type="submit" id="editButton">Edit Profile</button>
  <p><i id="helloText">Hello, Jane Jacobs!</i></p>
</form>

<style>
* { box-sizing: border-box; }
body { font-family: sans-serif; margin: 20px; padding: 0; }
label { display: block; margin-bottom: 20px; }
</style>
```

The `updateDOM` function you wrote shows what React does under the hood when you set the state. (However, React also avoids touching the DOM for properties that have not changed since the last time they were set.)

'Referencing Values with Refs'

When you want a component to "remember" some information, but you don't want that information to trigger new renders, you can use a *ref*.

- 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 {/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
}
```

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:

```
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>
});
```

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 {/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:

```
import { 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>
    </>
  );
}
```

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:**

```
import { 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>
    </>
  );
}
```

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 {/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:

refs	state
<code>useRef(initialValue)</code> returns <code>{ current: initialValue }</code>	<code>useState(initialValue)</code> returns the current value of a state variable and a state setter function (<code>[value, setValue]</code>)
Doesn't trigger re-render when you change it.	Triggers re-render when you change it.
Mutable—you can modify and update <code>current</code> '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 <code>current</code> value during rendering.	You can read state at any time. However, each render has its own <code>snapshot</code> of state which does not change.

Here is a counter button that's implemented with state:

```
import { 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**:

```
import { 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>
  );
}
```

This is why reading `ref.current` during render leads to unreliable code. If you need that, use state instead.

How does `useRef` work inside? {/how-does-use-ref-work-inside/}

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 React
function 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 {/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 {/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 {/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`. You can read more about this in [Manipulating the DOM with Refs](#).

- 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.

Fix a broken chat input {/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.

Regular variables like `let timeoutID` don't "survive" between re-renders because every render runs your component (and initializes its variables) from scratch. Should you keep the timeout ID somewhere else?

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

export default function Chat() {
  const [text, setText] = useState('');
  const [isSending, setIsSending] = useState(false);
  let timeoutID = null;

  function handleSend() {
    setIsSending(true);
    timeoutID = setTimeout(() => {
      alert('Sent!');
      setIsSending(false);
    }, 3000);
  }

  function handleUndo() {
    setIsSending(false);
    clearTimeout(timeoutID);
  }
}
```

```

return (
  <>
    <input
      disabled={isSending}
      value={text}
      onChange={e => setText(e.target.value)}
    />
    <button
      disabled={isSending}
      onClick={handleSend}>
      {isSending ? 'Sending...' : 'Send'}
    </button>
    {isSending &&
      <button onClick={handleUndo}>
        Undo
      </button>
    }
  </>
);
}

```

Whenever your component re-renders (such as when you set state), all local variables get initialized from scratch. This is why you can't save the timeout ID in a local variable like `timeoutID` and then expect another event handler to "see" it in the future. Instead, store it in a ref, which React will preserve between renders.

```

import { useState, useRef } from 'react';

export default function Chat() {
  const [text, setText] = useState('');
  const [isSending, setIsSending] = useState(false);
  const timeoutRef = useRef(null);

  function handleSend() {
    setIsSending(true);
    timeoutRef.current = setTimeout(() => {
      alert('Sent!');
      setIsSending(false);
    }, 3000);
  }

  function handleUndo() {
    setIsSending(false);
    clearTimeout(timeoutRef.current);
  }

  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>
    }
  );
}

```

Fix a component failing to re-render {/fix-a-component-failing-to-re-render/}

This button is supposed to toggle between showing "On" and "Off". However, it always shows "Off". What is wrong with this code? Fix it.

```

import { useRef } from 'react';

export default function Toggle() {
  const isOnRef = useRef(false);

  return (
    <button onClick={() => {
      isOnRef.current = !isOnRef.current;
    }}>
      {isOnRef.current ? 'On' : 'Off'}
    </button>
  );
}

```

In this example, the current value of a ref is used to calculate the rendering output: `{isOnRef.current ? 'On' : 'Off'}`. This is a sign that this information should not be in a ref, and should have instead been put in state. To fix it, remove the ref and use state instead:

```

import { useState } from 'react';

export default function Toggle() {
  const [isOn, setIsOn] = useState(false);

  return (
    <button onClick={() => {
      setIsOn(!isOn);
    }}>
      {isOn ? 'On' : 'Off'}
    </button>
  );
}

```

Fix debouncing {/fix-debouncing/}

In this example, all button click handlers are "debounced". To see what this means, press one of the buttons. Notice how the message appears a second later. If you press the button while waiting for the message, the timer will reset. So if you keep clicking the same button fast many times, the message won't appear until a second *after* you stop clicking. Debouncing lets you delay some action until the user "stops doing things".

This example works, but not quite as intended. The buttons are not independent. To see the problem, click one of the buttons, and then immediately click another button. You'd expect that after a delay, you would see both button's messages. But only the last button's message shows up. The first button's message gets lost.

Why are the buttons interfering with each other? Find and fix the issue.

The last timeout ID variable is shared between all `DebouncedButton` components. This is why clicking one button resets another button's timeout. Can you store a separate timeout ID for each button?

```
let timeoutID;

function DebouncedButton({ onClick, children }) {
  return (
    <button onClick={() => {
      clearTimeout(timeoutID);
      timeoutID = setTimeout(() => {
        onClick();
      }, 1000);
    }}>
      {children}
    </button>
  );
}

export default function Dashboard() {
  return (
    <>
      <DebouncedButton
        onClick={() => alert('Spaceship launched!')}
      >
        Launch the spaceship
      </DebouncedButton>
      <DebouncedButton
        onClick={() => alert('Soup boiled!')}
      >
        Boil the soup
      </DebouncedButton>
      <DebouncedButton
        onClick={() => alert('Lullaby sung!')}
      >
        Sing a lullaby
      </DebouncedButton>
    </>
  )
}

button { display: block; margin: 10px; }
```

A variable like `timeoutID` is shared between all components. This is why clicking on the second button resets the first button's pending timeout. To fix this, you can keep timeout in a ref. Each button will get its own ref, so they won't conflict with each other. Notice how clicking two buttons fast will show both messages.

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

function DebouncedButton({ onClick, children }) {
```

```

const timeoutRef = useRef(null);
return (
  <button onClick={() => {
    clearTimeout(timeoutRef.current);
    timeoutRef.current = setTimeout(() => {
      onClick();
    }, 1000);
  }}>
    {children}
  </button>
);
}

export default function Dashboard() {
  return (
    <>
      <DebouncedButton
        onClick={() => alert('Spaceship launched!')}
      >
        Launch the spaceship
      </DebouncedButton>
      <DebouncedButton
        onClick={() => alert('Soup boiled!')}
      >
        Boil the soup
      </DebouncedButton>
      <DebouncedButton
        onClick={() => alert('Lullaby sung!')}
      >
        Sing a lullaby
      </DebouncedButton>
    </>
  )
}

```

```
button { display: block; margin: 10px; }
```

Read the latest state {/read-the-latest-state/}

In this example, after you press "Send", there is a small delay before the message is shown. Type "hello", press Send, and then quickly edit the input again. Despite your edits, the alert would still show "hello" (which was the value of state `at the time` the button was clicked).

Usually, this behavior is what you want in an app. However, there may be occasional cases where you want some asynchronous code to read the *latest* version of some state. Can you think of a way to make the alert show the *current* input text rather than what it was at the time of the click?

```

import { useState, useRef } from 'react';

export default function Chat() {
  const [text, setText] = useState('');

  function handleSend() {
    setTimeout(() => {
      alert(`Sending: ${text}`);
    }, 3000);
  }

  return (

```

```

    <>
      <input
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button
        onClick={handleSend}>
        Send
      </button>
    </>
  );
}

```

State works [like a snapshot](#), so you can't read the latest state from an asynchronous operation like a timeout. However, you can keep the latest input text in a ref. A ref is mutable, so you can read the `current` property at any time. Since the current text is also used for rendering, in this example, you will need *both* a state variable (for rendering), *and* a ref (to read it in the timeout). You will need to update the current ref value manually.

```

import { useState, useRef } from 'react';

export default function Chat() {
  const [text, setText] = useState('');
  const textRef = useRef(text);

  function handleChange(e) {
    setText(e.target.value);
    textRef.current = e.target.value;
  }

  function handleSend() {
    setTimeout(() => {
      alert('Sending: ' + textRef.current);
    }, 3000);
  }

  return (
    <>
      <input
        value={text}
        onChange={handleChange}
      />
      <button
        onClick={handleSend}>
        Send
      </button>
    </>
  );
}

```

'Removing Effect Dependencies'

When you write an Effect, the linter will verify that you've included every reactive value (like props and state) that the Effect reads in the list of your Effect's dependencies. This ensures that your Effect remains synchronized with the latest props and state of your component. Unnecessary dependencies may cause your Effect to run too often, or even create an infinite loop. Follow this guide to review and remove unnecessary dependencies from your Effects.

- 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 {/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:

```
import { 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:<input type="text" value={roomId} />
      </label>
      <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} />
  </>
);
};

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

Fill them in according to what the linter says:

```

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✓ All dependencies declared
  // ...
}

```

Effects "react" to reactive values. Since `roomId` is a reactive value (it can change due to a re-render), the linter verifies that you've specified it as a dependency. If `roomId` receives a different value, React will resynchronize your Effect. This ensures that the chat stays connected to the selected room and "reacts" to the dropdown:

```

import { 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} />
    </>
  );
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

To remove a dependency, prove that it's not a dependency {/to-remove-a-dependency-prove-that-its-not-a-dependency/}

Notice that you can't "choose" the dependencies of your Effect. Every reactive value used by your Effect's code must be declared in your dependency list. The dependency list is determined by the surrounding code:

```

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) { // This is a reactive value
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId); // This Effect reads
    // that reactive value
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✓ So you must specify that reactive value as a dependency of
  // your Effect
  // ...
}

```

Reactive values include props and all variables and functions declared directly inside of your component. Since `roomId` is a reactive value, you can't remove it from the dependency list. The linter wouldn't allow it:

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // 🚫 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 anymore

function ChatRoom() {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // ✅ All dependencies declared
  // ...
}
```

Now that `roomId` is not a reactive value (and can't change on a re-render), it doesn't need to be a dependency:

```
import { 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>;
}

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('❌ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}
```

```
}
```

```
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

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 *{/to-change-the-dependencies-change-the-code/}*

You might have noticed a pattern in your workflow:

1. First, you **change the code** of your Effect or how your reactive values are declared.
2. Then, you follow the linter and adjust the dependencies to **match the code you have changed**.
3. 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.

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.

Why is suppressing the dependency linter so dangerous? *{/why-is-suppressing-the-dependency-linter-so-dangerous/}*

Suppressing the linter leads to very unintuitive bugs that are hard to find and fix. Here's one example:

```
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>
  </>
);
};

button { margin: 10px; }

```

Let'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 {/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? {/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) {
      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) {
      post('/api/register');
      showNotification('Successfully registered!', theme);
    }
  }, [submitted, theme]); // ✅ All dependencies declared
```

```
function handleSubmit() {
  setSubmitted(true);
}

// ...
```

By doing this, you've introduced a bug. Imagine you submit the form first and then switch between Dark and Light themes. The `theme` will change, the Effect will re-run, and so it will display the same notification again!

The problem here is that this shouldn't be an Effect in the first place. You want to send this POST request and show the notification in response to *submitting the form*, which is a particular interaction. To run some code in response to particular interaction, put that logic directly into the corresponding event handler:

```
function Form() {
  const theme = useContext(ThemeContext);

  function handleSubmit() {
    // ✅ Good: Event-specific logic is called from event handlers
    post('/api/register');
    showNotification('Successfully registered!', theme);
  }

  // ...
}
```

Now that the code is in an event handler, it's not reactive--so it will only run when the user submits the form. Read more about [choosing between event handlers and Effects](#) and [how to delete unnecessary Effects](#).

Is your Effect doing several unrelated things? {/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]);
}

export default ShippingForm;
```

```
    };
}, [country]); // ✅ All dependencies declared
// ...
```

This is a good example of [fetching data in an Effect](#). You are synchronizing the `cities` state with the network according to the `country` prop. You can't do this in an event handler because you need to fetch as soon as `ShippingForm` is displayed and whenever the `country` changes (no matter which interaction causes it).

Now let's say you're adding a second select box for city areas, which should fetch the `areas` for the currently selected `city`. You might start by adding a second `fetch` call for the list of areas inside the same Effect:

```
function ShippingForm({ country }) {
  const [cities, setCities] = useState(null);
  const [city, setCity] = useState(null);
  const [areas, setAreas] = useState(null);

  useEffect(() => {
    let ignore = false;
    fetch(`/api/cities?country=${country}`)
      .then(response => response.json())
      .then(json => {
        if (!ignore) {
          setCities(json);
        }
      });
    if (city) {
      fetch(`/api/areas?city=${city}`)
        .then(response => response.json())
        .then(json => {
          if (!ignore) {
            setAreas(json);
          }
        });
    }
    return () => {
      ignore = true;
    };
  }, [country, city]); // ✅ All dependencies declared
// ...
```

However, since the Effect now uses the `city` state variable, you've had to add `city` to the list of dependencies. That, in turn, introduced a problem: when the user selects a different city, the Effect will re-run and call `fetchCities(country)`. As a result, you will be unnecessarily refetching the list of cities many times.

The problem with this code is that you're synchronizing two different unrelated things:

1. You want to synchronize the `cities` state to the network based on the `country` prop.
2. You want to synchronize the `areas` state to the network based on the `city` state.

Split the logic into two Effects, each of which reacts to the prop that it needs to synchronize with:

```

function ShippingForm({ country }) {
  const [cities, setCities] = useState(null);
  useEffect(() => {
    let ignore = false;
    fetch(`/api/cities?country=${country}`)
      .then(response => response.json())
      .then(json => {
        if (!ignore) {
          setCities(json);
        }
      });
    return () => {
      ignore = true;
    };
  }, [country]); // ✅ All dependencies declared

  const [city, setCity] = useState(null);
  const [areas, setAreas] = useState(null);
  useEffect(() => {
    if (city) {
      let ignore = false;
      fetch(`/api/areas?city=${city}`)
        .then(response => response.json())
        .then(json => {
          if (!ignore) {
            setAreas(json);
          }
        });
      return () => {
        ignore = true;
      };
    }
  }, [city]); // ✅ All dependencies declared

  // ...
}

```

Now the first Effect only re-runs if the `country` changes, while the second Effect re-runs when the `city` changes. You've separated them by purpose: two different things are synchronized by two separate Effects. Two separate Effects have two separate dependency lists, so they won't trigger each other unintentionally.

The final code is longer than the original, but splitting these Effects is still correct. [Each Effect should represent an independent synchronization process](#). In this example, deleting one Effect doesn't break the other Effect's logic. This means they *synchronize different things*, and it's good to split them up. If you're concerned about duplication, you can improve this code by [extracting repetitive logic into a custom Hook](#).

Are you reading some state to calculate the next state? {/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]);
    });
  }, []);
}

```

```
    setMessages([...messages, receivedMessage]);
  });
// ...
```

It uses the `messages` variable to [create a new array](#) starting with all the existing messages and adds the new message at the end. However, since `messages` is a reactive value read by an Effect, it must be a dependency:

```
function ChatRoom({ roomId }) {
  const [messages, setMessages] = useState([]);
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
    connection.on('message', (receivedMessage) => {
      setMessages([...messages, receivedMessage]);
    });
    return () => connection.disconnect();
  }, [roomId, messages]); // ✅ All dependencies declared
// ...
```

And making `messages` a dependency introduces a problem.

Every time you receive a message, `setMessages()` causes the component to re-render with a new `messages` array that includes the received message. However, since this Effect now depends on `messages`, this will *also* re-synchronize the Effect. So every new message will make the chat re-connect. The user would not like that!

To fix the issue, don't read `messages` inside the Effect. Instead, pass an [updater function](#) to `setMessages`:

```
function ChatRoom({ roomId }) {
  const [messages, setMessages] = useState([]);
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
    connection.on('message', (receivedMessage) => {
      setMessages(msgs => [...msgs, receivedMessage]);
    });
    return () => connection.disconnect();
  }, [roomId]); // ✅ All dependencies declared
// ...
```

Notice how your Effect does not read the `messages` variable at all now. You only need to pass an updater function like `msgs => [...msgs, receivedMessage]`. React [puts your updater function in a queue](#) and will provide the `msgs` argument to it during the next render. This is why the Effect itself doesn't need to depend on `messages` anymore. As a result of this fix, receiving a chat message will no longer make the chat re-connect.

Do you want to read a value without "reacting" to its changes? {/do-you-want-to-read-a-value-without-reacting-to-its-changes/}

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]); // ✅ All dependencies declared
// ...
```

The problem is that every time `isMuted` changes (for example, when the user presses the "Muted" toggle), the Effect will re-synchronize, and reconnect to the chat. This is not the desired user experience! (In this example, even disabling the linter would not work--if you do that, `isMuted` would get "stuck" with its old value.)

To solve this problem, you need to extract the logic that shouldn't be reactive out of the Effect. You don't want this Effect to "react" to the changes in `isMuted`. [Move this non-reactive piece of logic into an Effect Event:](#)

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

function ChatRoom({ roomId }) {
  const [messages, setMessages] = useState([]);
  const [isMuted, setIsMuted] = useState(false);

  const onMessage = useEffectEvent(receivedMessage => {
    setMessages(msgs => [...msgs, receivedMessage]);
    if (!isMuted) {
      playSound();
    }
  });

  useEffect(() => {
    const connection = createConnection();
    // ...
  });
}
```

```

connection.connect();
connection.on('message', (receivedMessage) => {
  onMessage(receivedMessage);
});
return () => connection.disconnect();
}, [roomId]); // ✅ All dependencies declared
// ...

```

Effect Events let you split an Effect into reactive parts (which should "react" to reactive values like `roomId` and their changes) and non-reactive parts (which only read their latest values, like `onMessage` reads `isMuted`). **Now that you read `isMuted` inside an Effect Event, it doesn't need to be a dependency of your Effect.** As a result, the chat won't re-connect when you toggle the "Muted" setting on and off, solving the original issue!

Wrapping an event handler from the props { /wrapping-an-event-handler-from-the-props/ }

You might run into a similar problem when your component receives an event handler as a prop:

```

function ChatRoom({ roomId, onReceiveMessage }) {
  const [messages, setMessages] = useState([]);
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
    connection.on('message', (receivedMessage) => {
      onReceiveMessage(receivedMessage);
    });
    return () => connection.disconnect();
  }, [roomId, onReceiveMessage]); // ✅ All dependencies declared
  // ...
}

```

Suppose that the parent component passes a *different* `onReceiveMessage` function on every render:

```

<ChatRoom
  roomId={roomId}
  onReceiveMessage={receivedMessage => {
    // ...
  }}
/>

```

Since `onReceiveMessage` is a dependency, it would cause the Effect to re-synchronize after every parent re-render. This would make it re-connect to the chat. To solve this, wrap the call in an Effect Event:

```

function ChatRoom({ roomId, onReceiveMessage }) {
  const [messages, setMessages] = useState([]);
  const onMessage = useEffectEvent(receivedMessage => {
    onReceiveMessage(receivedMessage);
  });
  useEffect(() => {
    const connection = createConnection();
    connection.connect();
    connection.on('message', (receivedMessage) => {
      onMessage(receivedMessage);
    });
  });
}

```

```
    return () => connection.disconnect();
}, [roomId]); // ✅ All dependencies declared
// ...
```

Effect Events aren't reactive, so you don't need to specify them as dependencies. As a result, the chat will no longer re-connect even if the parent component passes a function that's different on every re-render.

Separating reactive and non-reactive code {/separating-reactive-and-non-reactive-code/}

In this example, you want to log a visit every time `roomId` changes. You want to include the current `notificationCount` with every log, but you *don't* want a change to `notificationCount` to trigger a log event.

The solution is again to split out the non-reactive code into an Effect Event:

```
function Chat({ roomId, notificationCount }) {
  const onVisit = useEffectEvent(visitedRoomId => {
    logVisit(visitedRoomId, notificationCount);
  });

  useEffect(() => {
    onVisit(roomId);
  }, [roomId]); // ✅ All dependencies declared
// ...
}
```

You want your logic to be reactive with regards to `roomId`, so you read `roomId` inside of your Effect. However, you don't want a change to `notificationCount` to log an extra visit, so you read `notificationCount` inside of the Effect Event. [Learn more about reading the latest props and state from Effects using Effect Events](#).

Does some reactive value change unintentionally? {/does-some-reactive-value-change-unintentionally/}

Sometimes, you *do* want your Effect to "react" to a certain value, but that value changes more often than you'd like--and might not reflect any actual change from the user's perspective. For example, let's say that you create an `options` object in the body of your component, and then read that object from inside of your Effect:

```
function ChatRoom({ roomId }) {
// ...
const options = {
  serverUrl: serverUrl,
  roomId: roomId
};

useEffect(() => {
  const connection = createConnection(options);
  connection.connect();
// ...
}
```

This object is declared in the component body, so it's a [reactive value](#). When you read a reactive value like this inside an Effect, you declare it as a dependency. This ensures your Effect "reacts" to its changes:

```
// ...
useEffect(() => {
  const connection = createConnection(options);
  connection.connect();
  return () => connection.disconnect();
}, [options]); // ✅ All dependencies declared
// ...
```

It is important to declare it as a dependency! This ensures, for example, that if the `roomId` changes, your Effect will re-connect to the chat with the new `options`. However, there is also a problem with the code above. To see it, try typing into the input in the sandbox below, and watch what happens in the console:

```
import { 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} />
    </>
  );
}
```

```

        </>
    );
}

export function createConnection({ serverUrl, roomId }) {
    // A real implementation would actually connect to the server
    return {
        connect() {
            console.log('✅ Connecting to "' + roomId + '" room at ' + serverUrl +
            '...');

        },
        disconnect() {
            console.log('❌ Disconnected from "' + roomId + '" room at ' + serverUrl);
        }
    };
}

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

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 {/move-static-objects-and-functions-outside-your-component/}

If the object does not depend on any props and state, you can move that object outside your component:

```

const options = {
  serverUrl: 'https://localhost:1234',
  roomId: 'music'
};

function ChatRoom() {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, []); // ✅ All dependencies declared
// ...

```

This way, you *prove* to the linter that it's not reactive. It can't change as a result of a re-render, so it doesn't need to be a dependency. Now re-rendering `ChatRoom` won't cause your Effect to re-synchronize.

This works for functions too:

```

function createOptions() {
  return {
    serverUrl: 'https://localhost:1234',
    roomId: 'music'
  };
}

function ChatRoom() {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, []); // ✅ All dependencies declared
// ...

```

Since `createOptions` is declared outside your component, it's not a reactive value. This is why it doesn't need to be specified in your Effect's dependencies, and why it won't ever cause your Effect to re-synchronize.

Move dynamic objects and functions inside your Effect {/move-dynamic-objects-and-functions-inside-your-effect/}

If your object depends on some reactive value that may change as a result of a re-render, like a `roomId` prop, you can't pull it *outside* your component. You can, however, move its creation *inside* of your Effect's code:

```

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const options = {
      serverUrl: serverUrl,

```

```

    roomId: roomId
  };
  const connection = createConnection(options);
  connection.connect();
  return () => connection.disconnect();
}, [roomId]); // ✅ All dependencies declared
// ...

```

Now that `options` is declared inside of your Effect, it is no longer a dependency of your Effect. Instead, the only reactive value used by your Effect is `roomId`. Since `roomId` is not an object or function, you can be sure that it won't be *unintentionally* different. In JavaScript, numbers and strings are compared by their content:

```

// During the first render
const roomId1 = 'music';

// During the next render
const 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:

```

import { 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)}
      </Select>
    </>
  );
}

```

```

    >
      <option value="general">general</option>
      <option value="travel">travel</option>
      <option value="music">music</option>
    </select>
  </label>
  <hr />
  <ChatRoom roomId={roomId} />
</>
);
}

```

```

export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

However, it *does* re-connect when you change the `roomId` dropdown, as you would expect.

This works for functions, too:

```

const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    function createOptions() {
      return {
        serverUrl: serverUrl,
        roomId: roomId
      };
    }

    const options = createOptions();
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✓ All dependencies declared
// ...

```

You can write your own functions to group pieces of logic inside your Effect. As long as you also declare them *inside* your Effect, they're not reactive values, and so they don't need to be dependencies of your Effect.

Read primitive values from objects [{/read-primitive-values-from-objects/}](#)

Sometimes, you may receive an object from props:

```
function ChatRoom({ options }) {
  const [message, setMessage] = useState('');
  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [options]); // ✅ All dependencies declared
  // ...
}
```

The risk here is that the parent component will create the object during rendering:

```
<ChatRoom
  roomId={roomId}
  options={{
    serverUrl: serverUrl,
    roomId: roomId
  }}
/>
```

This would cause your Effect to re-connect every time the parent component re-renders. To fix this, read information from the object *outside* the Effect, and avoid having object and function dependencies:

```
function ChatRoom({ options }) {
  const [message, setMessage] = useState('');
  const { roomId, serverUrl } = options;
  useEffect(() => {
    const connection = createConnection({
      roomId,
      serverUrl
    });
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]); // ✅ All dependencies declared
  // ...
}
```

The logic gets a little repetitive (you read some values from an object outside an Effect, and then create an object with the same values inside the Effect). But it makes it very explicit what information your Effect *actually* depends on. If an object is re-created unintentionally by the parent component, the chat would not re-connect. However, if `options.roomId` or `options.serverUrl` really are different, the chat would re-connect.

Calculate primitive values from functions [{/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,
      serverUrl
    });
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]); // ✅ All dependencies declared
// ...
```

This only works for `pure` functions because they are safe to call during rendering. If your function is an event handler, but you don't want its changes to re-synchronize your Effect, [wrap it into an Effect Event instead](#).

- 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.

Fix a resetting interval *{/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.

It seems like this Effect's code depends on `count`. Is there some way to not need this dependency? There should be a way to update the `count` state based on its previous value without adding a dependency on that value.

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

export default function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('✅ Creating an interval');
    const id = setInterval(() => {
      console.log('⌚ Interval tick');
    }, 1000);
  }, []);
}
```

```

    setCount(count + 1);
}, 1000);
return () => {
  console.log('✖ Clearing an interval');
  clearInterval(id);
};
}, [count]);
}

return <h1>Counter: {count}</h1>
}

```

You want to update the `count` state to be `count + 1` from inside the Effect. However, this makes your Effect depend on `count`, which changes with every tick, and that's why your interval gets re-created on every tick.

To solve this, use the [updater function](#) and write `setCount(c => c + 1)` instead of `setCount(count + 1)`:

```

import { useState, useEffect } from 'react';

export default function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('✓ Creating an interval');
    const id = setInterval(() => {
      console.log('⌚ Interval tick');
      setCount(c => c + 1);
    }, 1000);
    return () => {
      console.log('✖ Clearing an interval');
      clearInterval(id);
    };
  }, []);
}

return <h1>Counter: {count}</h1>
}

```

Instead of reading `count` inside the Effect, you pass a `c => c + 1` instruction ("increment this number!") to React. React will apply it on the next render. And since you don't need to read the value of `count` inside your Effect anymore, so you can keep your Effect's dependencies empty (`[]`). This prevents your Effect from re-creating the interval on every tick.

Fix a retriggering animation [{/fix-a-retriggering-animation/}](#)

In this example, when you press "Show", a welcome message fades in. The animation takes a second. When you press "Remove", the welcome message immediately disappears. The logic for the fade-in animation is implemented in the `animation.js` file as plain JavaScript [animation loop](#). You don't need to change that logic. You can treat it as a third-party library. Your Effect creates an instance of `FadeInAnimation` for the DOM node, and then calls `start(duration)` or `stop()` to control the animation. The `duration` is controlled by a slider. Adjust the slider and see how the animation changes.

This code already works, but there is something you want to change. Currently, when you move the slider that controls the `duration` state variable, it retriggers the animation. Change the behavior so that the Effect does not "react" to the `duration` variable. When you press "Show", the Effect should use the current `duration` on the slider. However, moving the slider itself should not by itself retrigger the animation.

Is there a line of code inside the Effect that should not be reactive? How can you move non-reactive code out of the Effect?

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```
import { useState, useEffect, useRef } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { FadeInAnimation } from './animation.js';

function Welcome({ duration }) {
  const ref = useRef(null);

  useEffect(() => {
    const animation = new FadeInAnimation(ref.current);
    animation.start(duration);
    return () => {
      animation.stop();
    };
  }, [duration]);

  return (
    <h1
      ref={ref}
      style={{
        opacity: 0,
        color: 'white',
        padding: 50,
        textAlign: 'center',
        fontSize: 50,
        backgroundImage: 'radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%)'
      }}
    >
      Welcome
    </h1>
  );
}

export default function App() {
  const [duration, setDuration] = useState(1000);
  const [show, setShow] = useState(false);
```

```

    return (
      <>
        <label>
          <input
            type="range"
            min="100"
            max="3000"
            value={duration}
            onChange={e => setDuration(Number(e.target.value))}>
        />
        <br />
        Fade in duration: {duration} ms
      </label>
      <button onClick={() => setShow(!show)}>
        {show ? 'Remove' : 'Show'}
      </button>
      <hr />
      {show && <Welcome duration={duration} />}
    </>
  );
}

```

```

export class FadeInAnimation {
  constructor(node) {
    this.node = node;
  }
  start(duration) {
    this.duration = duration;
    if (this.duration === 0) {
      // Jump to end immediately
      this.onProgress(1);
    } else {
      this.onProgress(0);
      // Start animating
      this.startTime = performance.now();
      this.frameId = requestAnimationFrame(() => this.onFrame());
    }
  }
  onFrame() {
    const timePassed = performance.now() - this.startTime;
    const progress = Math.min(timePassed / this.duration, 1);
    this.onProgress(progress);
    if (progress < 1) {
      // We still have more frames to paint
      this.frameId = requestAnimationFrame(() => this.onFrame());
    }
  }
  onProgress(progress) {
    this.node.style.opacity = progress;
  }
  stop() {
    cancelAnimationFrame(this.frameId);
    this.startTime = null;
    this.frameId = null;
    this.duration = 0;
  }
}

```

```

label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }

```

Your Effect needs to read the latest value of `duration`, but you don't want it to "react" to changes in `duration`. You use `duration` to start the animation, but starting animation isn't reactive. Extract the non-reactive line of code into an Effect Event, and call that function from your Effect.

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```
import { useState, useEffect, useRef } from 'react';
import { FadeInAnimation } from './animation.js';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

function Welcome({ duration }) {
  const ref = useRef(null);

  const onAppear = useEffectEvent(animation => {
    animation.start(duration);
  });

  useEffect(() => {
    const animation = new FadeInAnimation(ref.current);
    onAppear(animation);
    return () => {
      animation.stop();
    };
  }, []);
}

return (
  <h1
    ref={ref}
    style={{
      opacity: 0,
      color: 'white',
      padding: 50,
      textAlign: 'center',
      fontSize: 50,
      backgroundImage: 'radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%)'
    }}
  >
  Welcome
  </h1>
);
}

export default function App() {
  const [duration, setDuration] = useState(1000);
  const [show, setShow] = useState(false);

  return (
    <div>
      {show ? <Welcome duration={duration} /> : null}
      <br/>
      Duration: {duration}<br/>
      <button onClick={() => setDuration(1500)}>Change Duration</button>
    </div>
  );
}
```

```

    <>
      <label>
        <input
          type="range"
          min="100"
          max="3000"
          value={duration}
          onChange={e => setDuration(Number(e.target.value))}>
        />
        <br />
        Fade in duration: {duration} ms
      </label>
      <button onClick={() => setShow(!show)}>
        {show ? 'Remove' : 'Show'}
      </button>
      <hr />
      {show && <Welcome duration={duration} />}
    </>
  );
}

```

```

export class FadeInAnimation {
  constructor(node) {
    this.node = node;
  }
  start(duration) {
    this.duration = duration;
    this.onProgress(0);
    this.startTime = performance.now();
    this.frameId = requestAnimationFrame(() => this.onFrame());
  }
  onFrame() {
    const timePassed = performance.now() - this.startTime;
    const progress = Math.min(timePassed / this.duration, 1);
    this.onProgress(progress);
    if (progress < 1) {
      // We still have more frames to paint
      this.frameId = requestAnimationFrame(() => this.onFrame());
    }
  }
  onProgress(progress) {
    this.node.style.opacity = progress;
  }
  stop() {
    cancelAnimationFrame(this.frameId);
    this.startTime = null;
    this.frameId = null;
    this.duration = 0;
  }
}

```

```

label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }

```

Effect Events like `onAppear` are not reactive, so you can read `duration` inside without retriggering the animation.

Fix a reconnecting chat [{/fix-a-reconnecting-chat/}](#)

In this example, every time you press "Toggle theme", the chat re-connects. Why does this happen? Fix the mistake so that the chat re-connects only when you edit the Server URL or choose a different chat room.

Treat `chat.js` as an external third-party library: you can consult it to check its API, but don't edit it.

There's more than one way to fix this, but ultimately you want to avoid having an object as your dependency.

```
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
  const [isDark, setIsDark] = useState(false);
  const [roomId, setRoomId] = useState('general');
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  const options = {
    serverUrl,
    roomId
  };

  return (
    <div className={isDark ? 'dark' : 'light'}>
      <button onClick={() => setIsDark(!isDark)}>
        Toggle theme
      </button>
      <label>
        Server URL:{' '}
        <input
          value={serverUrl}
          onChange={e => setServerUrl(e.target.value)}
        />
      </label>
      <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 options={options} />
    </div>
  );
}
```

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

export default function ChatRoom({ options }) {
  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  });
}
```

```

    }, [options]);
}

return <h1>Welcome to the {options.roomId} room!</h1>;
}

```

```

export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  if (typeof serverUrl !== 'string') {
    throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
  }
  if (typeof roomId !== 'string') {
    throw Error('Expected roomId to be a string. Received: ' + roomId);
  }
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');

    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

label, button { display: block; margin-bottom: 5px; }
.dark { background: #222; color: #eee; }

```

Your Effect is re-running because it depends on the `options` object. Objects can be re-created unintentionally, you should try to avoid them as dependencies of your Effects whenever possible.

The least invasive fix is to read `roomId` and `serverUrl` right outside the Effect, and then make the Effect depend on those primitive values (which can't change unintentionally). Inside the Effect, create an object and it pass to `createConnection`:

```

import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
  const [isDark, setIsDark] = useState(false);
  const [roomId, setRoomId] = useState('general');
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  const options = {
    serverUrl,
    roomId
  };

  return (
    <div className={isDark ? 'dark' : 'light'}>
      <button onClick={() => setIsDark(!isDark)}>
        Toggle theme
      </button>
      <label>
        Server URL:{' '}
        <input
          value={serverUrl}
          onChange={e => setServerUrl(e.target.value)}
        />
      </label>
    </div>
  );
}

```

```

<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 options={options} />
</div>
);
}

```

```

import { useEffect } from 'react';
import { createConnection } from './chat.js';

export default function ChatRoom({ options }) {
  const { roomId, serverUrl } = options;
  useEffect(() => {
    const connection = createConnection({
      roomId,
      serverUrl
    });
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]);
  return <h1>Welcome to the {options.roomId} room!</h1>;
}

```

```

export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  if (typeof serverUrl !== 'string') {
    throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
  }
  if (typeof roomId !== 'string') {
    throw Error('Expected roomId to be a string. Received: ' + roomId);
  }
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');
    },
    disconnect() {
      console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```

label, button { display: block; margin-bottom: 5px; }
.dark { background: #222; color: #eee; }

```

It would be even better to replace the object `options` prop with the more specific `roomId` and `serverUrl` props:

```

import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
  const [isDark, setIsDark] = useState(false);
  const [roomId, set roomId] = useState('general');
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  return (
    <div className={isDark ? 'dark' : 'light'}>
      <button onClick={() => setIsDark(!isDark)}>
        Toggle theme
      </button>
      <label>
        Server URL:{' '}
        <input
          value={serverUrl}
          onChange={e => setServerUrl(e.target.value)}
        />
      </label>
      <label>
        Choose the chat room:{' '}
        <select
          value={roomId}
          onChange={e => set roomId(e.target.value)}
        >
          <option value="general">general</option>
          <option value="travel">travel</option>
          <option value="music">music</option>
        </select>
      </label>
      <hr />
      <ChatRoom
        roomId={roomId}
        serverUrl={serverUrl}
      />
    </div>
  );
}

```

```

import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

export default function ChatRoom({ roomId, serverUrl }) {
  useEffect(() => {
    const connection = createConnection({
      roomId,
      serverUrl
    });
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]);

  return <h1>Welcome to the {roomId} room!</h1>;
}

```

```

export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  if (typeof serverUrl !== 'string') {
    throw Error(`Expected serverUrl to be a string. Received: ${serverUrl}`);
  }
  if (typeof roomId !== 'string') {

```

```

        throw Error('Expected roomId to be a string. Received: ' + roomId);
    }
    return {
      connect() {
        console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
        '...');
      },
      disconnect() {
        console.log('✗ Disconnected from "' + roomId + '" room at ' + serverUrl);
      }
    };
}

```

```

label, button { display: block; margin-bottom: 5px; }
.dark { background: #222; color: #eee; }

```

Sticking to primitive props where possible makes it easier to optimize your components later.

Fix a reconnecting chat, again {/fix-a-reconnecting-chat-again/}

This example connects to the chat either with or without encryption. Toggle the checkbox and notice the different messages in the console when the encryption is on and off. Try changing the room. Then, try toggling the theme. When you're connected to a chat room, you will receive new messages every few seconds. Verify that their color matches the theme you've picked.

In this example, the chat re-connects every time you try to change the theme. Fix this. After the fix, changing the theme should not re-connect the chat, but toggling encryption settings or changing the room should re-connect.

Don't change any code in `chat.js`. Other than that, you can change any code as long as it results in the same behavior. For example, you may find it helpful to change which props are being passed down.

You're passing down two functions: `onMessage` and `createConnection`. Both of them are created from scratch every time `App` re-renders. They are considered to be new values every time, which is why they re-trigger your Effect.

One of these functions is an event handler. Do you know some way to call an event handler an Effect without "reacting" to the new values of the event handler function? That would come in handy!

Another of these functions only exists to pass some state to an imported API method. Is this function really necessary? What is the essential information that's being passed down? You might need to move some imports from `App.js` to `ChatRoom.js`.

```

{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
  }
}

```

```

    "eject": "react-scripts eject"
}

import { useState } from 'react';
import ChatRoom from './ChatRoom.js';
import {
  createEncryptedConnection,
  createUnencryptedConnection,
} from './chat.js';
import { showNotification } from './notifications.js';

export default function App() {
  const [isDark, setIsDark] = useState(false);
  const [roomId, setRoomId] = useState('general');
  const [isEncrypted, setIsEncrypted] = useState(false);

  return (
    <>
      <label>
        <input
          type="checkbox"
          checked={isDark}
          onChange={e => setIsDark(e.target.checked)}
        />
        Use dark theme
      </label>
      <label>
        <input
          type="checkbox"
          checked={isEncrypted}
          onChange={e => setIsEncrypted(e.target.checked)}
        />
        Enable encryption
      </label>
      <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}
        onMessage={msg => {
          showNotification(`New message: ${msg}, isDark ? 'dark' : 'light'`);
        }}
        createConnection={() => {
          const options = {
            serverUrl: 'https://localhost:1234',
            roomId: roomId
          };
          if (isEncrypted) {
            return createEncryptedConnection(options);
          } else {
            return createUnencryptedConnection(options);
          }
        }}
      >
    </>
  );
}

```

```

        }
      />
    </>
  );
}

```

```

import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function ChatRoom({ roomId, createConnection, onMessage }) {
  useEffect(() => {
    const connection = createConnection();
    connection.on('message', (msg) => onMessage(msg));
    connection.connect();
    return () => connection.disconnect();
  }, [createConnection, onMessage]);

  return <h1>Welcome to the {roomId} room!</h1>;
}

```

```

export function createEncryptedConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  if (typeof serverUrl !== 'string') {
    throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
  }
  if (typeof roomId !== 'string') {
    throw Error('Expected roomId to be a string. Received: ' + roomId);
  }
  let intervalId;
  let messageCallback;
  return {
    connect() {
      console.log('✅ 🔒 Connecting to "' + roomId + '" room... (encrypted)');
      clearInterval(intervalId);
      intervalId = setInterval(() => {
        if (messageCallback) {
          if (Math.random() > 0.5) {
            messageCallback('hey')
          } else {
            messageCallback('lol');
          }
        }
      }, 3000);
    },
    disconnect() {
      clearInterval(intervalId);
      messageCallback = null;
      console.log('❌ 🔒 Disconnected from "' + roomId + '" room (encrypted)');
    },
    on(event, callback) {
      if (messageCallback) {
        throw Error('Cannot add the handler twice.');
      }
      if (event !== 'message') {
        throw Error('Only "message" event is supported.');
      }
      messageCallback = callback;
    },
  };
}

export function createUnencryptedConnection({ serverUrl, roomId }) {

```

```
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
  throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
  throw Error('Expected roomId to be a string. Received: ' + roomId);
}
let intervalId;
let messageCallback;
return {
  connect() {
    console.log('✓ Connecting to "' + roomId + '" room (unencrypted)...');
    clearInterval(intervalId);
    intervalId = setInterval(() => {
      if (messageCallback) {
        if (Math.random() > 0.5) {
          messageCallback('hey')
        } else {
          messageCallback('lol');
        }
      }
    }, 3000);
  },
  disconnect() {
    clearInterval(intervalId);
    messageCallback = null;
    console.log('✗ Disconnected from "' + roomId + '" room (unencrypted)');
  },
  on(event, callback) {
    if (messageCallback) {
      throw Error('Cannot add the handler twice.');
    }
    if (event !== 'message') {
      throw Error('Only "message" event is supported.');
    }
    messageCallback = callback;
  },
};
}
```

```
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}
```

```
label, button { display: block; margin-bottom: 5px; }
```

There's more than one correct way to solve this, but here is one possible solution.

In the original example, toggling the theme caused different `onMessage` and `createConnection` functions to be created and passed down. Since the Effect depended on these functions, the chat would re-connect every time you toggle the theme.

To fix the problem with `onMessage`, you needed to wrap it into an Effect Event:

```
export default function ChatRoom({ roomId, createConnection, onMessage }) {
  const onReceiveMessage = useEffectEvent(onMessage);

  useEffect(() => {
    const connection = createConnection();
    connection.on('message', (msg) => onReceiveMessage(msg));
    // ...
  });
}
```

Unlike the `onMessage` prop, the `onReceiveMessage` Effect Event is not reactive. This is why it doesn't need to be a dependency of your Effect. As a result, changes to `onMessage` won't cause the chat to re-connect.

You can't do the same with `createConnection` because it *should* be reactive. You *want* the Effect to re-trigger if the user switches between an encrypted and an unencryption connection, or if the user switches the current room. However, because `createConnection` is a function, you can't check whether the information it reads has *actually* changed or not. To solve this, instead of passing `createConnection` down from the `App` component, pass the raw `roomId` and `isEncrypted` values:

```
<ChatRoom
  roomId={roomId}
  isEncrypted={isEncrypted}
  onMessage={msg => {
    showNotification('New message: ' + msg, isDark ? 'dark' : 'light');
  }}
/>
```

Now you can move the `createConnection` function *inside* the Effect instead of passing it down from the `App`:

```
import {
  createEncryptedConnection,
  createUnencryptedConnection,
} from './chat.js';

export default function ChatRoom({ roomId, isEncrypted, onMessage }) {
  const onReceiveMessage = useEffectEvent(onMessage);

  useEffect(() => {
    function createConnection() {
      const options = {
        serverUrl: 'https://localhost:1234',
        roomId: roomId
      };
      if (isEncrypted) {
        return createEncryptedConnection(options);
      } else {
        return createUnencryptedConnection(options);
      }
    }
  });
}
```

```

    }
    // ...
}
```

After these two changes, your Effect no longer depends on any function values:

```

export default function ChatRoom({ roomId, isEncrypted, onMessage }) { // Reactive
  values
  const onReceiveMessage = useEffectEvent(onMessage); // Not reactive

  useEffect(() => {
    function createConnection() {
      const options = {
        serverUrl: 'https://localhost:1234',
        roomId // Reading a reactive value
      };
      if (isEncrypted) { // Reading a reactive value
        return createEncryptedConnection(options);
      } else {
        return createUnencryptedConnection(options);
      }
    }

    const connection = createConnection();
    connection.on('message', (msg) => onReceiveMessage(msg));
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, isEncrypted]); // ✅ All dependencies declared
}
```

As a result, the chat re-connects only when something meaningful (`roomId` or `isEncrypted`) changes:

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```

import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

import { showNotification } from './notifications.js';

export default function App() {
  const [isDark, setIsDark] = useState(false);
  const [roomId, setRoomId] = useState('general');
  const [isEncrypted, setIsEncrypted] = useState(false);

  return (
    <>
      <label>
        <input
```

```

        type="checkbox"
        checked={isDark}
        onChange={e => setIsDark(e.target.checked)}
      />
      Use dark theme
    </label>
    <label>
      <input
        type="checkbox"
        checked={isEncrypted}
        onChange={e => setIsEncrypted(e.target.checked)}
      />
      Enable encryption
    </label>
    <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}
      isEncrypted={isEncrypted}
      onMessage={msg => {
        showNotification('New message: ' + msg, isDark ? 'dark' : 'light');
      }}
    />
  </>
);
}

```

```

import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import {
  createEncryptedConnection,
  createUnencryptedConnection,
} from './chat.js';

export default function ChatRoom({ roomId, isEncrypted, onMessage }) {
  const onReceiveMessage = useEffectEvent(onMessage);

  useEffect(() => {
    function createConnection() {
      const options = {
        serverUrl: 'https://localhost:1234',
        roomId: roomId
      };
      if (isEncrypted) {
        return createEncryptedConnection(options);
      } else {
        return createUnencryptedConnection(options);
      }
    }

    const connection = createConnection();
    connection.on('message', (msg) => onReceiveMessage(msg));
  });
}

```

```

        connection.connect();
        return () => connection.disconnect();
    }, [roomId, isEncrypted]);

    return <h1>Welcome to the {roomId} room!</h1>;
}

export function createEncryptedConnection({ serverUrl, roomId }) {
    // A real implementation would actually connect to the server
    if (typeof serverUrl !== 'string') {
        throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
    }
    if (typeof roomId !== 'string') {
        throw Error('Expected roomId to be a string. Received: ' + roomId);
    }
    let intervalId;
    let messageCallback;
    return {
        connect() {
            console.log('✓ 🔒 Connecting to "' + roomId + '" room... (encrypted)');
            clearInterval(intervalId);
            intervalId = setInterval(() => {
                if (messageCallback) {
                    if (Math.random() > 0.5) {
                        messageCallback('hey')
                    } else {
                        messageCallback('lol');
                    }
                }
            }, 3000);
        },
        disconnect() {
            clearInterval(intervalId);
            messageCallback = null;
            console.log('✖ 🔒 Disconnected from "' + roomId + '" room (encrypted)');
        },
        on(event, callback) {
            if (messageCallback) {
                throw Error('Cannot add the handler twice.');
            }
            if (event !== 'message') {
                throw Error('Only "message" event is supported.');
            }
            messageCallback = callback;
        },
    };
}

export function createUnencryptedConnection({ serverUrl, roomId }) {
    // A real implementation would actually connect to the server
    if (typeof serverUrl !== 'string') {
        throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
    }
    if (typeof roomId !== 'string') {
        throw Error('Expected roomId to be a string. Received: ' + roomId);
    }
    let intervalId;
    let messageCallback;
    return {
        connect() {
            console.log('✓ 🔒 Connecting to "' + roomId + '" room (unencrypted)...');
            clearInterval(intervalId);
        }
    };
}

```

```

intervalId = setInterval(() => {
  if (messageCallback) {
    if (Math.random() > 0.5) {
      messageCallback('hey')
    } else {
      messageCallback('lol');
    }
  }
}, 3000);
},
disconnect() {
  clearInterval(intervalId);
  messageCallback = null;
  console.log('✖ Disconnected from "' + roomId + '" room (unencrypted)');
},
on(event, callback) {
  if (messageCallback) {
    throw Error('Cannot add the handler twice.');
  }
  if (event !== 'message') {
    throw Error('Only "message" event is supported.');
  }
  messageCallback = callback;
},
};
}

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}

```

```

label, button { display: block; margin-bottom: 5px; }

```

Render and Commit

Before 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.

- 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:

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

Step 1: Trigger a render *{/step-1-trigger-a-render/}*

There are two reasons for a component to render:

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

Initial render *{/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:

```
import Image from './Image.js';
import { createRoot } from 'react-dom/client';

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

export default function Image() {
  return (
    
  );
}
```

Try commenting out the `root.render()` call and see the component disappear!

Re-renders when state updates *{/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.)

Step 2: React renders your components *{/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:

```
export default function Gallery() {
  return (
    <section>
      <h1>Inspiring Sculptures</h1>
      <Image />
      <Image />
      <Image />
    </section>
  );
}

function Image() {
  return (
    
  );
}

import Gallery from './Gallery.js';
import { createRoot } from 'react-dom/client';

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

img { margin: 0 10px 10px 0; }
```

- **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.

Rendering 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.

Optimizing performance {/optimizing-performance/}

The 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 {/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:

```
export default function Clock({ time }) {
  return (
    <>
      <h1>{time}</h1>
      <input />
    </>
  );
}
```

```
import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
```

```

const [time, setTime] = useState(() => new Date());
useEffect(() => {
  const id = setInterval(() => {
    setTime(new Date());
  }, 1000);
  return () => clearInterval(id);
}, []);
return time;
}

export default function App() {
  const time = useTime();
  return (
    <Clock time={time.toLocaleTimeString()} />
  );
}

```

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 {/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.

- Any screen update in a React app happens in three steps:
 1. Trigger
 2. Render
 3. 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

Rendering Lists

You 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.

- 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 {/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:

1. **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'
];
```

2. **Map** the `people` members into a new array of JSX nodes, `listItems`:

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

3. **Return** `listItems` from your component wrapped in a ``:

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

Here is the result:

```

const 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>;
}

li { margin-bottom: 10px; }

```

Notice the sandbox above displays a console error:

Warning: 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 { /filtering-arrays-of-items/ }

This data can be structured even more.

```

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

```

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

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

1. **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'
);
```

2. Now **map** over `chemists`:

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

3. Lastly, **return** the `listItems` from your component:

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

```
import { 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>;
}
```

```
export const people = [
  {
    name: 'Creola Katherine Johnson',
    profession: 'mathematician',
    accomplishment: 'spaceflight calculations',
    imageId: 'MK3eW3A'
  },
  {
    name: 'Mario José Molina-Pasquel Henríquez',
    profession: 'chemist',
    accomplishment: 'discovery of Arctic ozone hole',
    imageId: 'mynHUSA'
  },
  {
    name: 'Mohammad Abdus Salam',
    profession: 'physicist',
```

```

    accomplishment: 'electromagnetism theory',
    imageId: 'bE7W1ji'
  }, {
    name: 'Percy Lavon Julian',
    profession: 'chemist',
    accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
    imageId: 'IOjWm71'
  }, {
    name: 'Subrahmanyan Chandrasekhar',
    profession: 'astrophysicist',
    accomplishment: 'white dwarf star mass calculations',
    imageId: 'lrWQx8l'
  }];
}

```

```

export function getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageId +
    '.jpg'
);
}

```

```

ul { list-style-type: none; padding: 0px 10px; }
li {
  margin-bottom: 10px;
  display: grid;
  grid-template-columns: auto 1fr;
  gap: 20px;
  align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }

```

Arrow functions implicitly return the expression right after `=>`, so you didn't need a `return` statement:

```

const listItems = chemists.map(person =>
  <li>...</li> // Implicit return!
);

```

However, **you must write `return` explicitly if your `=>` is followed by a `{` curly brace!**

```

const listItems = chemists.map(person => { // Curly brace
  return <li>...</li>;
});

```

Arrow functions containing `=> {` are said to have a "block body". They let you write more than a single line of code, but you *have to* write a `return` statement yourself. If you forget it, nothing gets returned!

Keeping list items in order with `key` `{/keeping-list-items-in-order-with-key/}`

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

Warning: 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>
```

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:

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

```
export const people = [
  {
    name: 'Creola Katherine Johnson',
    profession: 'mathematician',
    accomplishment: 'spaceflight calculations',
    imageId: 'MK3eW3A'
  },
  {
    name: 'Mario José Molina-Pasquel Henríquez',
    profession: 'chemist',
    accomplishment: 'discovery of Arctic ozone hole',
    imageId: 'mynHUSA'
  },
  {
    name: 'Mohammad Abdus Salam',
    profession: 'physicist',
    accomplishment: 'electromagnetism theory',
    imageId: 'bE7W1ji'
  },
  {
    name: 'Percy Lavon Julian',
    profession: 'chemist',
    accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
    imageId: 'IOjWm71'
  },
  {
    name: 'Subrahmanyan Chandrasekhar',
    profession: 'astrophysicist',
  }
];
```

```
    accomplishment: 'white dwarf star mass calculations',
    imageUrl: '1rWQx8l'
  ];
}
```

```
export function getImageUrl(person) {
  return (
    `https://i.imgur.com/${person.imageUrl}.jpg`
  );
}
```

```
ul { list-style-type: none; padding: 0px 10px; }
li {
  margin-bottom: 10px;
  display: grid;
  grid-template-columns: auto 1fr;
  gap: 20px;
  align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
```

Displaying several DOM nodes for each list item {/displaying-several-dom-nodes-for-each-list-item/}

What 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 {/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 {/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? {/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.

You 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}>`.

On 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.

Splitting a list in two {/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'`.

```
import { 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)}>
    </li>
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

```

        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>
  );
}

```

```

export const people = [
  {
    name: 'Creola Katherine Johnson',
    profession: 'mathematician',
    accomplishment: 'spaceflight calculations',
    imageUrl: 'MK3eW3A'
  },
  {
    name: 'Mario José Molina-Pasquel Henríquez',
    profession: 'chemist',
    accomplishment: 'discovery of Arctic ozone hole',
    imageUrl: 'mynHUSA'
  },
  {
    name: 'Mohammad Abdus Salam',
    profession: 'physicist',
    accomplishment: 'electromagnetism theory',
    imageUrl: 'bE7W1ji'
  },
  {
    name: 'Percy Lavon Julian',
    profession: 'chemist',
    accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
    imageUrl: 'I0jWm71'
  },
  {
    name: 'Subrahmanyan Chandrasekhar',
    profession: 'astrophysicist',
    accomplishment: 'white dwarf star mass calculations',
    imageUrl: 'lrWQx8l'
  }
];

```

```

export function getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageUrl +
    '.jpg'
  );
}

```

```

ul { list-style-type: none; padding: 0px 10px; }
li {
  margin-bottom: 10px;
  display: grid;
  grid-template-columns: auto 1fr;
  gap: 20px;
  align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }

```

You could use `filter()` twice, creating two separate arrays, and then `map` over both of them:

```
import { people } from './data.js';
import { getImageUrl } from './utils.js';

export default function List() {
  const chemists = people.filter(person =>
    person.profession === 'chemist'
  );
  const everyoneElse = people.filter(person =>
    person.profession !== 'chemist'
  );
  return (
    <article>
      <h1>Scientists</h1>
      <h2>Chemists</h2>
      <ul>
        {chemists.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>
        )}
      </ul>
      <h2>Everyone Else</h2>
      <ul>
        {everyoneElse.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>
        )}
      </ul>
    </article>
  );
}
```

```
export const people = [
  {
    name: 'Creola Katherine Johnson',
    profession: 'mathematician',
    accomplishment: 'spaceflight calculations',
    imageId: 'MK3eW3A'
  },
  {
    name: 'Mario José Molina-Pasquel Henríquez',
    profession: 'chemist',
    accomplishment: 'discovery of Arctic ozone hole',
  }
];
```

```

        imageUrl: 'mynHUSA'
    }, {
        name: 'Mohammad Abdus Salam',
        profession: 'physicist',
        accomplishment: 'electromagnetism theory',
        imageUrl: 'bE7W1ji'
    }, {
        name: 'Percy Lavon Julian',
        profession: 'chemist',
        accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
        imageUrl: 'IOjWm71'
    }, {
        name: 'Subrahmanyan Chandrasekhar',
        profession: 'astrophysicist',
        accomplishment: 'white dwarf star mass calculations',
        imageUrl: 'lrWQx8l'
    }];
}

export function getImageUrl(person) {
    return (
        'https://i.imgur.com/' +
        person.imageUrl +
        '.jpg'
    );
}

ul { list-style-type: none; padding: 0px 10px; }
li {
    margin-bottom: 10px;
    display: grid;
    grid-template-columns: auto 1fr;
    gap: 20px;
    align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }

```

In this solution, the `map` calls are placed directly inline into the parent `` elements, but you could introduce variables for them if you find that more readable.

There is still a bit duplication between the rendered lists. You can go further and extract the repetitive parts into a `<ListSection>` component:

```

import { people } from './data.js';
import { getImageUrl } from './utils.js';

function ListSection({ title, people }) {
    return (
        <>
            <h2>{title}</h2>
            <ul>
                {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>
                )}
            </ul>
    );
}

```

```

        </p>
      </li>
    )
  )
</ul>
</>
);
};

export default function List() {
  const chemists = people.filter(person =>
  person.profession === 'chemist'
);
  const everyoneElse = people.filter(person =>
  person.profession !== 'chemist'
);
  return (
    <article>
      <h1>Scientists</h1>
      <ListSection
        title="Chemists"
        people={chemists}
      />
      <ListSection
        title="Everyone Else"
        people={everyoneElse}
      />
    </article>
  );
}

export const people = [
  {
    name: 'Creola Katherine Johnson',
    profession: 'mathematician',
    accomplishment: 'spaceflight calculations',
    imageUrl: 'MK3eW3A'
  },
  {
    name: 'Mario José Molina-Pasquel Henríquez',
    profession: 'chemist',
    accomplishment: 'discovery of Arctic ozone hole',
    imageUrl: 'mynHUSA'
  },
  {
    name: 'Mohammad Abdus Salam',
    profession: 'physicist',
    accomplishment: 'electromagnetism theory',
    imageUrl: 'bE7W1ji'
  },
  {
    name: 'Percy Lavon Julian',
    profession: 'chemist',
    accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
    imageUrl: 'IOjWm71'
  },
  {
    name: 'Subrahmanyan Chandrasekhar',
    profession: 'astrophysicist',
    accomplishment: 'white dwarf star mass calculations',
    imageUrl: 'lrWQx8l'
  },
];

export function getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageUrl +
  );
}

```

```
's.jpg'
});
```

```
ul { list-style-type: none; padding: 0px 10px; }
li {
  margin-bottom: 10px;
  display: grid;
  grid-template-columns: auto 1fr;
  gap: 20px;
  align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
```

A very attentive reader might notice that with two `filter` calls, we check each person's profession twice. Checking a property is very fast, so in this example it's fine. If your logic was more expensive than that, you could replace the `filter` calls with a loop that manually constructs the arrays and checks each person once.

In fact, if `people` never change, you could move this code out of your component. From React's perspective, all that matters is that you give it an array of JSX nodes in the end. It doesn't care how you produce that array:

```
import { people } from './data.js';
import { getImageUrl } from './utils.js';

let chemists = [];
let everyoneElse = [];
people.forEach(person => {
  if (person.profession === 'chemist') {
    chemists.push(person);
  } else {
    everyoneElse.push(person);
  }
});

function ListSection({ title, people }) {
  return (
    <>
      <h2>{title}</h2>
      <ul>
        {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>
        )}
      </ul>
    </>
  );
}
```

```
export default function List() {
  return (
    <article>
      <h1>Scientists</h1>
      <ListSection
        title="Chemists"
        people={chemists}
      />
      <ListSection
        title="Everyone Else"
        people={everyoneElse}
      />
    </article>
  );
}
```

```
export const people = [
  {
    name: 'Creola Katherine Johnson',
    profession: 'mathematician',
    accomplishment: 'spaceflight calculations',
    imageUrl: 'MK3eW3A'
  },
  {
    name: 'Mario José Molina-Pasquel Henríquez',
    profession: 'chemist',
    accomplishment: 'discovery of Arctic ozone hole',
    imageUrl: 'mynHUSA'
  },
  {
    name: 'Mohammad Abdus Salam',
    profession: 'physicist',
    accomplishment: 'electromagnetism theory',
    imageUrl: 'bE7W1ji'
  },
  {
    name: 'Percy Lavon Julian',
    profession: 'chemist',
    accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
    imageUrl: 'I0jWm71'
  },
  {
    name: 'Subrahmanyan Chandrasekhar',
    profession: 'astrophysicist',
    accomplishment: 'white dwarf star mass calculations',
    imageUrl: 'lrWQx8l'
  }];
]
```

```
export function getImageUrl(person) {
  return (
    'https://i.imgur.com/' +
    person.imageUrl +
    '.jpg'
  );
}
```

```
ul { list-style-type: none; padding: 0px 10px; }
li {
  margin-bottom: 10px;
  display: grid;
  grid-template-columns: auto 1fr;
  gap: 20px;
  align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
```

Nested lists in one component {/nested-lists-in-one-component/}

Make a list of recipes from this array! For each recipe in the array, display its name as an `<h2>` and list its ingredients in a ``.

This will require nesting two different `map` calls.

```
import { recipes } from './data.js';

export default function RecipeList() {
  return (
    <div>
      <h1>Recipes</h1>
    </div>
  );
}

export const recipes = [
  {
    name: 'Greek Salad',
    ingredients: ['tomatoes', 'cucumber', 'onion', 'olives', 'feta']
  },
  {
    name: 'Hawaiian Pizza',
    ingredients: ['pizza crust', 'pizza sauce', 'mozzarella', 'ham', 'pineapple']
  },
  {
    name: 'Hummus',
    ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']
  }];

```

Here is one way you could go about it:

```
import { recipes } from './data.js';

export default function RecipeList() {
  return (
    <div>
      <h1>Recipes</h1>
      {recipes.map(recipe =>
        <div key={recipe.id}>
          <h2>{recipe.name}</h2>
          <ul>
            {recipe.ingredients.map(ingredient =>
              <li key={ingredient}>
                {ingredient}
              </li>
            )}
          </ul>
        </div>
      )}
    </div>
  );
}

export const recipes = [
  {
    name: 'Greek Salad',
    ingredients: ['tomatoes', 'cucumber', 'onion', 'olives', 'feta']
  },
  {
    name: 'Hawaiian Pizza',
    ingredients: ['pizza crust', 'pizza sauce', 'mozzarella', 'ham', 'pineapple']
  },
  {
    name: 'Hummus',
    ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']
  }];

```

```
    name: 'Hummus',
    ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']
}];
```

Each of the `recipes` already includes an `id` field, so that's what the outer loop uses for its `key`. There is no ID you could use to loop over ingredients. However, it's reasonable to assume that the same ingredient won't be listed twice within the same recipe, so its name can serve as a `key`. Alternatively, you could change the data structure to add IDs, or use index as a `key` (with the caveat that you can't safely reorder ingredients).

Extracting a list item component [{/extracting-a-list-item-component/}](#)

This `RecipeList` component contains two nested `map` calls. To simplify it, extract a `Recipe` component from it which will accept `id`, `name`, and `ingredients` props. Where do you place the outer `key` and why?

```
import { recipes } from './data.js';

export default function RecipeList() {
  return (
    <div>
      <h1>Recipes</h1>
      {recipes.map(recipe =>
        <div key={recipe.id}>
          <h2>{recipe.name}</h2>
          <ul>
            {recipe.ingredients.map(ingredient =>
              <li key={ingredient}>
                {ingredient}
              </li>
            )}
          </ul>
        </div>
      )}
    </div>
  );
}
```

```
export const recipes = [
  {
    name: 'Greek Salad',
    ingredients: ['tomatoes', 'cucumber', 'onion', 'olives', 'feta']
  },
  {
    name: 'Hawaiian Pizza',
    ingredients: ['pizza crust', 'pizza sauce', 'mozzarella', 'ham', 'pineapple']
  },
  {
    name: 'Hummus',
    ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']
  };
];
```

You can copy-paste the JSX from the outer `map` into a new `Recipe` component and return that JSX. Then you can change `recipe.name` to `name`, `recipe.id` to `id`, and so on, and pass them as props to the `Recipe`:

```
import { recipes } from './data.js';

function Recipe({ id, name, ingredients }) {
```

```

return (
  <div>
    <h2>{name}</h2>
    <ul>
      {ingredients.map(ingredient =>
        <li key={ingredient}>
          {ingredient}
        </li>
      )}
    </ul>
  </div>
);
}

export default function RecipeList() {
  return (
    <div>
      <h1>Recipes</h1>
      {recipes.map(recipe =>
        <Recipe {...recipe} key={recipe.id} />
      )}
    </div>
);
}

```

```

export const recipes = [
  {
    name: 'Greek Salad',
    ingredients: ['tomatoes', 'cucumber', 'onion', 'olives', 'feta']
  },
  {
    name: 'Hawaiian Pizza',
    ingredients: ['pizza crust', 'pizza sauce', 'mozzarella', 'ham', 'pineapple']
  },
  {
    name: 'Hummus',
    ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']
  }];

```

Here, `<Recipe {...recipe} key={recipe.id} />` is a syntax shortcut saying "pass all properties of the `recipe` object as props to the `Recipe` component". You could also write each prop explicitly: `<Recipe id={recipe.id} name={recipe.name} ingredients={recipe.ingredients} key={recipe.id} />`.

Note that the `key` is specified on the `<Recipe>` itself rather than on the root `<div>` returned from `Recipe`. This is because this `key` is needed directly within the context of the surrounding array. Previously, you had an array of `<div>`s so each of them needed a `key`, but now you have an array of `<Recipe>`s. In other words, when you extract a component, don't forget to leave the `key` outside the JSX you copy and paste.

List with a separator [{/list-with-a-separator/}](#)

This example renders a famous haiku by Katsushika Hokusai, with each line wrapped in a `<p>` tag. Your job is to insert an `<hr />` separator between each paragraph. Your resulting structure should look like this:

```

<article>
  <p>I write, erase, rewrite</p>
  <hr />
  <p>Erase again, and then</p>

```

```
<hr />
<p>A poppy blooms.</p>
</article>
```

A haiku only contains three lines, but your solution should work with any number of lines. Note that `<hr />` elements only appear *between* the `<p>` elements, not in the beginning or the end!

```
const poem = {
  lines: [
    'I write, erase, rewrite',
    'Erase again, and then',
    'A poppy blooms.'
  ];

  export default function Poem() {
    return (
      <article>
        {poem.lines.map((line, index) =>
          <p key={index}>
            {line}
          </p>
        )}
      </article>
    );
  }
}
```

```
body {
  text-align: center;
}
p {
  font-family: Georgia, serif;
  font-size: 20px;
  font-style: italic;
}
hr {
  margin: 0 120px 0 120px;
  border: 1px dashed #45c3d8;
}
```

(This is a rare case where `index` as a key is acceptable because a poem's lines will never reorder.)

You'll either need to convert `map` to a manual loop, or use a fragment.

You can write a manual loop, inserting `<hr />` and `<p>...</p>` into the output array as you go:

```
const poem = {
  lines: [
    'I write, erase, rewrite',
    'Erase again, and then',
    'A poppy blooms.'
  ];

  export default function Poem() {
    let output = [];

    // Fill the output array
    poem.lines.forEach((line, i) => {
```

```

        output.push(
          <hr key={i + '-separator'} />
        );
        output.push(
          <p key={i + '-text'}>
            {line}
          </p>
        );
      );
    // Remove the first <hr />
    output.shift();

    return (
      <article>
        {output}
      </article>
    );
  }
}

```

```

body {
  text-align: center;
}
p {
  font-family: Georgia, serif;
  font-size: 20px;
  font-style: italic;
}
hr {
  margin: 0 120px 0 120px;
  border: 1px dashed #45c3d8;
}

```

Using the original line index as a `key` doesn't work anymore because each separator and paragraph are now in the same array. However, you can give each of them a distinct key using a suffix, e.g. `key={i + '-text'}`.

Alternatively, you could render a collection of fragments which contain `<hr />` and `<p>...</p>`. However, the `<>...</>` shorthand syntax doesn't support passing keys, so you'd have to write `<Fragment>` explicitly:

```

import { Fragment } from 'react';

const poem = {
  lines: [
    'I write, erase, rewrite',
    'Erase again, and then',
    'A poppy blooms.'
  ];
}

export default function Poem() {
  return (
    <article>
      {poem.lines.map((line, i) =>
        <Fragment key={i}>
          {i > 0 && <hr />}
          <p>{line}</p>
        </Fragment>
      )}
    </article>
  );
}

```

react

```
</article>
);

body {
  text-align: center;
}
p {
  font-family: Georgia, serif;
  font-size: 20px;
  font-style: italic;
}
hr {
  margin: 0 120px 0 120px;
  border: 1px dashed #45c3d8;
}
```

Remember, fragments (often written as `<> </>`) let you group JSX nodes without adding extra `<div>`s!

[Go to TOC](#)

Responding to Events

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

- 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 {/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:

```
export 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:

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

```
export default function Button() {
  function handleClick() {
    alert('You clicked me!');
  }

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

```
button { margin-right: 10px; }
```

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.

Functions 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 {/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:

```
function 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>
  );
}

button { margin-right: 10px; }
```

This lets these two buttons show different messages. Try changing the messages passed to them.

Passing event handlers as props {/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:

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

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

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

```

        </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>
    );
}

```

`button { margin-right: 10px; }`

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 *{/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`:

```

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

export default function App() {
    return (
        <div>
            <Button onSmash={() => alert('Playing!')}>
                Play Movie
            </Button>
        </div>
    );
}

```

```

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

button { margin-right: 10px; }

```

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:

```

export 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>
  );
}

button { margin-right: 10px; }

```

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.

Event propagation {/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?

```
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>
  );
}

.Toolbar {
  background: #aaa;
  padding: 5px;
}
button { margin: 5px; }
```

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.

All events propagate in React except `onScroll`, which only works on the JSX tag you attach it to.

Stopping propagation {/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:

```
function 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>
);
}

.Toolbar {
  background: #aaa;
  padding: 5px;
}
button { margin: 5px; }

```

When you click on a button:

1. React calls the `onClick` handler passed to `<button>`.
2. 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.
3. That function, defined in the `Toolbar` component, displays the button's own alert.
4. 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.

Capture phase events {/capture-phase-events/}

In 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:

1. It travels down, calling all `onClickCapture` handlers.
2. It runs the clicked element's `onClick` handler.
3. 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 {/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 {/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:

```
export default function Signup() {
  return (
    <form onSubmit={() => alert('Submitting!')}>
      <input />
      <button>Send</button>
    </form>
  );
}

button { margin-left: 5px; }
```

You can call `e.preventDefault()` on the event object to stop this from happening:

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

```
button { margin-left: 5px; }
```

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? {/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.

- 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.

Fix an event handler {/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.)

```
export 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>
  );
}
```

The problem is that `<button onClick={handleClick()}>` calls the `handleClick` function while rendering instead of passing it. Removing the `()` call so that it's `<button onClick={handleClick}>` fixes the issue:

```
export 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>
  );
}
```

Alternatively, you could wrap the call into another function, like `<button onClick={() => handleClick()}>`:

```
export 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>
  );
}
```

Wire up the events {/wire-up-the-events/}

This `ColorSwitch` component renders a button. It's supposed to change the page color. Wire it up to the `onChangeColor` event handler prop it receives from the parent so that clicking the button changes the color.

After you do this, notice that clicking the button also increments the page click counter. Your colleague who wrote the parent component insists that `onChangeColor` does not increment any counters. What else might be happening? Fix it so that clicking the button *only* changes the color, and does *not* increment the counter.

```
export default function ColorSwitch({
  onChangeColor
}) {
  return (

```

```

    <button>
      Change color
    </button>
  );
}

```

```

import { useState } from 'react';
import ColorSwitch from './ColorSwitch.js';

export default function App() {
  const [clicks, setClicks] = useState(0);

  function handleClickOutside() {
    setClicks(c => c + 1);
  }

  function getRandomLightColor() {
    let r = 150 + Math.round(100 * Math.random());
    let g = 150 + Math.round(100 * Math.random());
    let b = 150 + Math.round(100 * Math.random());
    return `rgb(${r}, ${g}, ${b})`;
  }

  function handleChangeColor() {
    let bodyStyle = document.body.style;
    bodyStyle.backgroundColor = getRandomLightColor();
  }

  return (
    <div style={{ width: '100%', height: '100%' }} onClick={handleClickOutside}>
      <ColorSwitch onChangeColor={handleChangeColor} />
      <br />
      <br />
      <h2>Clicks on the page: {clicks}</h2>
    </div>
  );
}

```

First, you need to add the event handler, like `<button onClick={onChangeColor}>`.

However, this introduces the problem of the incrementing counter. If `onChangeColor` does not do this, as your colleague insists, then the problem is that this event propagates up, and some handler above does it. To solve this problem, you need to stop the propagation. But don't forget that you should still call `onChangeColor`.

```

export default function ColorSwitch({
  onChangeColor
}) {
  return (
    <button onClick={e => {
      e.stopPropagation();
      onChangeColor();
    }}>
      Change color
    </button>
  );
}

```

```
import { useState } from 'react';
import ColorSwitch from './ColorSwitch.js';

export default function App() {
  const [clicks, setClicks] = useState(0);

  function handleClickOutside() {
    setClicks(c => c + 1);
  }

  function getRandomLightColor() {
    let r = 150 + Math.round(100 * Math.random());
    let g = 150 + Math.round(100 * Math.random());
    let b = 150 + Math.round(100 * Math.random());
    return `rgb(${r}, ${g}, ${b})`;
  }

  function handleChangeColor() {
    let bodyStyle = document.body.style;
    bodyStyle.backgroundColor = getRandomLightColor();
  }

  return (
    <div style={{ width: '100%', height: '100%' }} onClick={handleClickOutside}>
      <ColorSwitch onChangeColor={handleChangeColor} />
      <br />
      <br />
      <h2>Clicks on the page: {clicks}</h2>
    </div>
  );
}
```

'Reusing Logic with Custom Hooks'

React 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.

- 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

{/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:

1. A piece of state that tracks whether the network is online.
2. 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:

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

export default function StatusBar() {
  const [isOnline, setIsOnline] = useState(true);
  useEffect(() => {
    function handleOnline() {
      setIsOnline(true);
    }
    function handleOffline() {
      setIsOnline(false);
    }
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);
    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);
  return <h1>{isOnline ? '✅ Online' : '🔴 Disconnected'}</h1>;
}
```

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`:

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

export default function SaveButton() {
  const [isOnline, setIsOnline] = useState(true);
  useEffect(() => {
    function handleOnline() {
      setIsOnline(true);
    }
    function handleOffline() {
      setIsOnline(false);
    }
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);
    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);
  function handleSaveClick() {
    console.log('✓ Progress saved');
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? 'Save progress' : 'Reconnecting...'}
    </button>
  );
}
```

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 {/extracting-your-own-custom-hook-from-a-component/}

Imagine for a moment that, similar to `useState` and `useEffect`, there was a built-in `useOnlineStatus` Hook. Then both of these components could be simplified and you could remove the duplication between them:

```
function StatusBar() {
  const isOnline = useOnlineStatus();
  return <h1>{isOnline ? '✓ Online' : '✗ Disconnected'}</h1>;
}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log('✓ Progress saved');
```

```

        }

      return (
        <button disabled={!isOnline} onClick={handleSaveClick}>
          {isOnline ? 'Save progress' : 'Reconnecting...'}
        </button>
      );
    }
  }
}

```

Although there is no such built-in Hook, you can write it yourself. Declare a function called `useOnlineStatus` and move all the duplicated code into it from the components you wrote earlier:

```

function useOnlineStatus() {
  const [isOnline, setIsOnline] = useState(true);
  useEffect(() => {
    function handleOnline() {
      setIsOnline(true);
    }
    function handleOffline() {
      setIsOnline(false);
    }
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);
    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);
  return isOnline;
}

```

At the end of the function, return `isOnline`. This lets your components read that value:

```

import { useOnlineStatus } from './useOnlineStatus.js';

function StatusBar() {
  const isOnline = useOnlineStatus();
  return <h1>{isOnline ? '✓ Online' : '✗ Disconnected'}</h1>;
}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log('✓ Progress saved');
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? 'Save progress' : 'Reconnecting...'}
    </button>
  );
}

export default function App() {
  return (
    <>
      <SaveButton />
      <StatusBar />
    </>
  );
}

```

```

        </>
    );
}

import { 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;
}

```

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` *{/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!

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!

Should all functions called during rendering start with the `use` prefix? {/should-all-functions-called-during-rendering-start-with-the-use-prefix/}

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:

```
function useSorted(items) {
  return items.slice().sort();
}

// ✅ Good: A regular function that doesn't use Hooks
function getSorted(items) {
  return items.slice().sort();
}
```

This ensures that your code can call this regular function anywhere, including conditions:

```
function List({ items, shouldSort }) {
  let displayedItems = items;
  if (shouldSort) {
    // ✅ It's ok to call getSorted() conditionally because it's not a Hook
    displayedItems = getSorted(items);
  }
  // ...
}
```

You should give `use` prefix to a function (and thus make it a Hook) if it uses at least one Hook inside of it:

```
// ✅ Good: A Hook that uses other Hooks
function useAuth() {
  return useContext(Auth);
}
```

Technically, this isn't enforced by React. In principle, you could make a Hook that doesn't call other Hooks. This is often confusing and limiting so it's best to avoid that pattern. However, there may be rare cases where it is helpful. For example, maybe your function doesn't use any Hooks right now, but you plan to add some Hook calls to it in the future. Then it makes sense to name it with the `use` prefix:

```
// ✅ Good: A Hook that will likely use some other Hooks later
function 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 {/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:

```
import { 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>
  </>
);
}

```

```

label { display: block; }
input { margin-left: 10px; }

```

There's some repetitive logic for each form field:

1. There's a piece of state (`firstName` and `lastName`).
2. There's a change handler (`handleFirstNameChange` and `handleLastNameChange`).
3. 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:

```

import { useFormInput } from './useFormInput.js';

export default function Form() {
  const firstNameProps = useFormInput('Mary');
  const lastNameProps = useFormInput('Poppins');

  return (
    <>
      <label>
        First name:
        <input {...firstNameProps} />
      </label>
      <label>
        Last name:
        <input {...lastNameProps} />
      </label>
      <p><b>Good morning, {firstNameProps.value} {lastNameProps.value}.</b></p>
    </>
  );
}

```

```

import { 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;
}
```

```
label { display: block; }
input { margin-left: 10px; }
```

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 *{/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:

```
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

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}
      />
    </>
  );
}

```

```

import { 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>
    </>
  );
}

```

```

export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  if (typeof serverUrl !== 'string') {
    throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
  }
  if (typeof roomId !== 'string') {
    throw Error('Expected roomId to be a string. Received: ' + roomId);
  }
  let intervalId;
  let messageCallback;
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');
      clearInterval(intervalId);
      intervalId = setInterval(() => {
        if (messageCallback) {
          if (Math.random() > 0.5) {
            messageCallback('hey')
          } else {
            messageCallback('lol');
          }
        }
      }, 3000);
    },
  };
}

```

```

    disconnect() {
      clearInterval(intervalId);
      messageCallback = null;
      console.log('✖ Disconnected from "' + roomId + '" room at ' + serverUrl +
    '');
    },
    on(event, callback) {
      if (messageCallback) {
        throw Error('Cannot add the handler twice.');
      }
      if (event !== 'message') {
        throw Error('Only "message" event is supported.');
      }
      messageCallback = callback;
    },
  };
}

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme = 'dark') {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}

```

```

{
  "dependencies": {
    "react": "latest",
    "react-dom": "latest",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

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,
      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:

```
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

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}
/>
</>
);
}

import { 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>
    </>
  );
}

import { useEffect } from 'react';
import { createConnection } from './chat.js';
import { showNotification } from './notifications.js';

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]);
}

export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  if (typeof serverUrl !== 'string') {
    throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
  }
  if (typeof roomId !== 'string') {
    throw Error('Expected roomId to be a string. Received: ' + roomId);
  }
  let intervalId;
  let messageCallback;
  return {
    connect() {
      console.log('✓ Connecting to "' + roomId + '" room at ' + serverUrl +
    }
  }
}

```

```
'...');
    clearInterval(intervalId);
    intervalId = setInterval(() => {
      if (messageCallback) {
        if (Math.random() > 0.5) {
          messageCallback('hey')
        } else {
          messageCallback('lol');
        }
      }
    }, 3000);
  },
  disconnect() {
    clearInterval(intervalId);
    messageCallback = null;
    console.log('✖ Disconnected from "' + roomId + '" room at ' + serverUrl +
  '');
},
on(event, callback) {
  if (messageCallback) {
    throw Error('Cannot add the handler twice.');
  }
  if (event !== 'message') {
    throw Error('Only "message" event is supported.');
  }
  messageCallback = callback;
},
};
}
```

```
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme = 'dark') {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}
```

```
{
  "dependencies": {
    "react": "latest",
    "react-dom": "latest",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

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 {/passing-event-handlers-to-custom-hooks/}

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,
    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,
      roomId
    };
    const connection = createConnection(options);
    connection.connect();
    connection.on('message', (msg) => {
      onReceiveMessage(msg);
    });
    return () => connection.disconnect();
  }, [roomId, serverUrl, onReceiveMessage]); // ✅ All dependencies declared
}
```

This will work, but there's one more improvement you can do when your custom Hook accepts event handlers.

Adding a dependency on `onReceiveMessage` is not ideal because it will cause the chat to re-connect every time the component re-renders. Wrap this event handler into an Effect Event to remove it from the dependencies:

```
import { useEffect, useEffectEvent } from 'react';
// ...

export function useChatRoom({ serverUrl, roomId, onReceiveMessage }) {
  const onMessage = useEffectEvent(onReceiveMessage);

  useEffect(() => {
    const options = {
      serverUrl,
      roomId
    };
    const connection = createConnection(options);
    connection.connect();
    connection.on('message', (msg) => {
      onMessage(msg);
    });
    return () => connection.disconnect();
  }, [roomId, serverUrl]); // ✅ All dependencies declared
}
```

Now the chat won't re-connect every time that the `ChatRoom` component re-renders. Here is a fully working demo of passing an event handler to a custom Hook that you can play with:

```
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
  const [roomId, set roomId] = useState('general');
  return (
    <>
      <label>
        Choose the chat room: {' '}
      </label>
      <select
        value={roomId}
        onChange={e => set roomId(e.target.value)}
      >
        <option value="general">general</option>
        <option value="travel">travel</option>
        <option value="music">music</option>
      </select>
    </>
    <hr />
    <ChatRoom
      roomId={roomId}
    />
  </>
);
}
```

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

```
import { useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection } from './chat.js';

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]);
```

```
export function createConnection({ serverUrl, roomId }) {
  // A real implementation would actually connect to the server
  if (typeof serverUrl !== 'string') {
    throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
  }
  if (typeof roomId !== 'string') {
    throw Error('Expected roomId to be a string. Received: ' + roomId);
  }
  let intervalId;
  let messageCallback;
  return {
    connect() {
      console.log('✅ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');
      clearInterval(intervalId);
      intervalId = setInterval(() => {
        if (messageCallback) {
          if (Math.random() > 0.5) {
            messageCallback('hey')
          } else {
            messageCallback('lol');
          }
        }
      }, 3000);
    },
    disconnect() {
      clearInterval(intervalId);
      messageCallback = null;
      console.log('❌ Disconnected from "' + roomId + '" room at ' + serverUrl +
      '>');
    },
    on(event, callback) {
      if (messageCallback) {
        throw Error('Cannot add the handler twice.');
      }
      if (event !== 'message') {
        throw Error('Only "message" event is supported.');
      }
      messageCallback = callback;
    },
  };
}
```

```
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme = 'dark') {
  Toastify({
```

```

        text: message,
        duration: 2000,
        gravity: 'top',
        position: 'right',
        style: {
          background: theme === 'dark' ? 'black' : 'white',
          color: theme === 'dark' ? 'white' : 'black',
        },
      }).showToast();
}

```

```

{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}

```

```

input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }

```

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 {/when-to-use-custom-hooks/}

You don't need to extract a custom Hook for every little duplicated bit of code. Some duplication is fine. For example, extracting a `useFormInput` Hook to wrap a single `useState` call like earlier is probably unnecessary.

However, whenever you write an Effect, consider whether it would be clearer to also wrap it in a custom Hook. [You shouldn't need Effects very often](#), so if you're writing one, it means that you need to "step outside React" to synchronize with some external system or to do something that React doesn't have a built-in API for. Wrapping it into a custom Hook lets you precisely communicate your intent and how the data flows through it.

For example, consider a `ShippingForm` component that displays two dropdowns: one shows the list of cities, and another shows the list of areas in the selected city. You might start with some code that looks like this:

```

function ShippingForm({ country }) {
  const [cities, setCities] = useState(null);
  // This Effect fetches cities for a country
  useEffect(() => {

```

```

let ignore = false;
fetch(`/api/cities?country=${country}`)
  .then(response => response.json())
  .then(json => {
    if (!ignore) {
      setCities(json);
    }
  });
return () => {
  ignore = true;
};
}, [country]);

const [city, setCity] = useState(null);
const [areas, setAreas] = useState(null);
// This Effect fetches areas for the selected city
useEffect(() => {
  if (city) {
    let ignore = false;
    fetch(`/api/areas?city=${city}`)
      .then(response => response.json())
      .then(json => {
        if (!ignore) {
          setAreas(json);
        }
      });
    return () => {
      ignore = true;
    };
  }
}, [city]);
// ...

```

Although this code is quite repetitive, it's correct to keep these Effects separate from each other. They synchronize two different things, so you shouldn't merge them into one Effect. Instead, you can simplify the `ShippingForm` component above by extracting the common logic between them into your own `useData` Hook:

```

function useData(url) {
  const [data, setData] = useState(null);
  useEffect(() => {
    if (url) {
      let ignore = false;
      fetch(url)
        .then(response => response.json())
        .then(json => {
          if (!ignore) {
            setData(json);
          }
        });
      return () => {
        ignore = true;
      };
    }
  }, [url]);
  return data;
}

```

Now you can replace both Effects in the `ShippingForm` components with calls to `useData`:

```
function ShippingForm({ country }) {
  const cities = useData(`api/cities?country=${country}`);
  const [city, setCity] = useState(null);
  const areas = useData(city ? `api/areas?city=${city}` : null);
  // ...
```

Extracting a custom Hook makes the data flow explicit. You feed the `url` in and you get the `data` out. By "hiding" your Effect inside `useData`, you also prevent someone working on the `ShippingForm` component from adding `unnecessary dependencies` to it. With time, most of your app's Effects will be in custom Hooks.

Keep your custom Hooks focused on concrete high-level use cases {/keep-your-custom-hooks-focused-on-concrete-high-level-use-cases/}

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:

- `useData(url)`
- `useImpressionLog(eventName, extraData)`
- `useChatRoom(options)`

When you synchronize with an external system, your custom Hook name may be more technical and use jargon specific to that system. It's good as long as it would be clear to a person familiar with that system:

- `useMediaQuery(query)`
- `useSocket(url)`
- `useIntersectionObserver(ref, options)`

Keep custom Hooks focused on concrete high-level use cases. Avoid creating and using custom "life-cycle" 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');

  useMount(() => {
    const connection = createConnection({ roomId, serverUrl });
    connection.connect();

    post('/analytics/event', { eventName: 'visit_chat' });
  });
  // ...
}
```

```
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');

  // ✅ Good: two raw Effects separated by purpose

  useEffect(() => {
    const connection = createConnection({ serverUrl, roomId });
    connection.connect();
    return () => connection.disconnect();
  }, [serverUrl, roomId]);

  useEffect(() => {
    post('/analytics/event', { eventName: 'visit_chat', roomId });
  }, [roomId]);

  // ...
}
```

Then, you can (but don't have to) extract custom Hooks for different high-level use cases:

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  // ✅ Great: custom Hooks named after their purpose
  useChatRoom({ serverUrl, roomId });
  useImpressionLog('visit_chat', { roomId });
  // ...
}
```

A good custom Hook makes the calling code more declarative by constraining what it does. For example, `useChatRoom(options)` can only connect to the chat room, while `useImpressionLog(eventName, extraData)` can only send an impression log to the analytics. If your custom Hook API doesn't constrain the use cases and is very abstract, in the long run it's likely to introduce more problems than it solves.

Custom Hooks help you migrate to better patterns {/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:

```
import { useOnlineStatus } from './useOnlineStatus.js';

function StatusBar() {
  const isOnline = useOnlineStatus();
  return <h1>{isOnline ? '✓ Online' : '✗ Disconnected'}</h1>;
}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log('✓ Progress saved');
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? 'Save progress' : 'Reconnecting...'}
    </button>
  );
}

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

```
import { 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;
}
```

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:

```
import { useOnlineStatus } from './useOnlineStatus.js';

function StatusBar() {
  const isOnline = useOnlineStatus();
  return <h1>{isOnline ? '✓ Online' : '✗ Disconnected'}</h1>;
}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log('✓ Progress saved');
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? 'Save progress' : 'Reconnecting...'}
    </button>
  );
}

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

```
import { 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
);
}

```

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:

1. You make the data flow to and from your Effects very explicit.
2. You let your components focus on the intent rather than on the exact implementation of your Effects.
3. 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.

Will React provide any built-in solution for data fetching? {/will-react-provide-any-built-in-solution-for-data-fetching/}

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 {/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:

```

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 />}
    </>
  );
}

```

```

label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
.welcome {
  opacity: 0;
  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%);
}

```

To make the component more readable, you might extract the logic into a `useFadeIn` custom Hook:

```

import { 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 />}
    </>
  );
}

```

```

import { useEffect } from 'react';

export function useFadeIn(ref, duration) {
  useEffect(() => {
    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) {
  // ...
}

```

```

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();
}, [ref, duration]);
}

```

```

label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }

.welcome {
  opacity: 0;
  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%);
}

```

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:

```

import { 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 />}
    </>
  );
}

```

```
</>
);
```

```
import { 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]);
}
```

```
label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
.welcome {
  opacity: 0;
  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%);
}
```

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  },
  "scripts": {
    "start": "react-scripts start",
  }
}
```

```

    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
}

```

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](#):

```

import { 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 />}
    </>
  );
}

```

```

import { 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]);
}

```

```

export class FadeInAnimation {
  constructor(node) {
    this.node = node;
  }
  start(duration) {
    this.duration = duration;
    this.onProgress(0);
    this.startTime = performance.now();
    this.frameId = requestAnimationFrame(() => this.onFrame());
  }
  onFrame() {
    const progress = (performance.now() - this.startTime) / this.duration;
    this.onProgress(progress);
    if (progress < 1) {
      this.frameId = requestAnimationFrame(this.onFrame);
    }
  }
  onProgress(progress) {
    this.node.style.opacity = progress;
  }
}

```

```

    }
    onFrame() {
      const timePassed = performance.now() - this.startTime;
      const progress = Math.min(timePassed / this.duration, 1);
      this.onProgress(progress);
      if (progress === 1) {
        this.stop();
      } else {
        // We still have more frames to paint
        this.frameId = requestAnimationFrame(() => this.onFrame());
      }
    }
    onProgress(progress) {
      this.node.style.opacity = progress;
    }
    stop() {
      cancelAnimationFrame(this.frameId);
      this.startTime = null;
      this.frameId = null;
      this.duration = 0;
    }
  }
}

```

```

label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
.welcome {
  opacity: 0;
  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%);
}

```

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](#):

```

import { useState, useEffect, useRef } from 'react';
import './welcome.css';

function Welcome() {
  return (
    <h1 className="welcome">
      Welcome
    </h1>
  );
}

export default function App() {
  const [show, setShow] = useState(false);
  return (
    <>

```

```

        <button onClick={() => setShow(!show)}>
          {show ? 'Remove' : 'Show'}
        </button>
        <hr />
        {show && <Welcome />}
      </>
    );
}

label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }

```

```

.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!

- 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.

Extract a `useCounter` Hook [/extract-a-usecounter-hook/](#)

This component uses a state variable and an Effect to display a number that increments every second. Extract this logic into a custom Hook called `useCounter`. Your goal is to make the `Counter` component implementation look exactly like this:

```

export default function Counter() {
  const count = useCounter();
  return <h1>Seconds passed: {count}</h1>;
}

```

You'll need to write your custom Hook in `useCounter.js` and import it into the `Counter.js` file.

```
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>;
}
```

// Write your custom Hook in this file!

Your code should look like this:

```
import { useCounter } from './useCounter.js';

export default function Counter() {
  const count = useCounter();
  return <h1>Seconds passed: {count}</h1>;
}
```

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

export function useCounter() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return count;
}
```

Notice that `App.js` doesn't need to import `useState` or `useEffect` anymore.

Make the counter delay configurable [/make-the-counter-delay-configurable/](#)

In this example, there is a `delay` state variable controlled by a slider, but its value is not used. Pass the `delay` value to your custom `useCounter` Hook, and change the `useCounter` Hook to use the passed `delay` instead of hardcoding `1000` ms.

```
import { useState } from 'react';
import { useCounter } from './useCounter.js';

export default function Counter() {
  const [delay, setDelay] = useState(1000);
  const count = useCounter();
  return (
    <>
      <label>
        Tick duration: {delay} ms
      <br />
      <input
        type="range"

```

```

        value={delay}
        min="10"
        max="2000"
        onChange={e => setDelay(Number(e.target.value))}}
      />
    </label>
    <hr />
    <h1>Ticks: {count}</h1>
  </>
);
}

```

```

import { useState, useEffect } from 'react';

export function useCounter() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return count;
}

```

Pass the `delay` to your Hook with `useCounter(delay)`. Then, inside the Hook, use `delay` instead of the hardcoded `1000` value. You'll need to add `delay` to your Effect's dependencies. This ensures that a change in `delay` will reset the interval.

```

import { useState } from 'react';
import { useCounter } from './useCounter.js';

export default function Counter() {
  const [delay, setDelay] = useState(1000);
  const count = useCounter(delay);
  return (
    <>
      <label>
        Tick duration: {delay} ms
        <br />
        <input
          type="range"
          value={delay}
          min="10"
          max="2000"
          onChange={e => setDelay(Number(e.target.value))}>
      />
    </label>
    <hr />
    <h1>Ticks: {count}</h1>
  </>
);
}

```

```

import { useState, useEffect } from 'react';

export function useCounter(delay) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const id = setInterval(() => {

```

```

    setCount(c => c + 1);
}, delay);
return () => clearInterval(id);
}, [delay]);
return count;
}

```

Extract `useInterval` out of `useCounter` {/extract-useinterval-out-of-usecounter/}

Currently, your `useCounter` Hook does two things. It sets up an interval, and it also increments a state variable on every interval tick. Split out the logic that sets up the interval into a separate Hook called `useInterval`. It should take two arguments: the `onTick` callback, and the `delay`. After this change, your `useCounter` implementation should look like this:

```

export function useCounter(delay) {
  const [count, setCount] = useState(0);
  useInterval(() => {
    setCount(c => c + 1);
  }, delay);
  return count;
}

```

Write `useInterval` in the `useInterval.js` file and import it into the `useCounter.js` file.

```

import { useState } from 'react';
import { useCounter } from './useCounter.js';

export default function Counter() {
  const count = useCounter(1000);
  return <h1>Seconds passed: {count}</h1>;
}

```

```

import { useState, useEffect } from 'react';

export function useCounter(delay) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1);
    }, delay);
    return () => clearInterval(id);
  }, [delay]);
  return count;
}

```

```
// Write your Hook here!
```

The logic inside `useInterval` should set up and clear the interval. It doesn't need to do anything else.

```

import { useCounter } from './useCounter.js';

export default function Counter() {
  const count = useCounter(1000);
  return <h1>Seconds passed: {count}</h1>;
}

```

```

import { useState } from 'react';
import { useInterval } from './useInterval.js';

export function useCounter(delay) {
  const [count, setCount] = useState(0);
  useInterval(() => {
    setCount(c => c + 1);
  }, delay);
  return count;
}

import { useEffect } from 'react';

export function useInterval(onTick, delay) {
  useEffect(() => {
    const id = setInterval(onTick, delay);
    return () => clearInterval(id);
  }, [onTick, delay]);
}

```

Note that there is a bit of a problem with this solution, which you'll solve in the next challenge.

Fix a resetting interval [{/fix-a-resetting-interval/}](#)

In this example, there are *two* separate intervals.

The `App` component calls `useCounter`, which calls `useInterval` to update the counter every second. But the `App` component *also* calls `useInterval` to randomly update the page background color every two seconds.

For some reason, the callback that updates the page background never runs. Add some logs inside `useInterval`:

```

useEffect(() => {
  console.log('✓ Setting up an interval with delay ', delay)
  const id = setInterval(onTick, delay);
  return () => {
    console.log('✗ Clearing an interval with delay ', delay)
    clearInterval(id);
  };
}, [onTick, delay]);

```

Do the logs match what you expect to happen? If some of your Effects seem to re-synchronize unnecessarily, can you guess which dependency is causing that to happen? Is there some way to [remove that dependency](#) from your Effect?

After you fix the issue, you should expect the page background to update every two seconds.

It looks like your `useInterval` Hook accepts an event listener as an argument. Can you think of some way to wrap that event listener so that it doesn't need to be a dependency of your Effect?

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  }
}
```

```
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```
import { useCounter } from './useCounter.js';
import { useInterval } from './useInterval.js';

export default function Counter() {
  const count = useCounter(1000);

  useInterval(() => {
    const randomColor = `hsla(${Math.random() * 360}, 100%, 50%, 0.2)`;
    document.body.style.backgroundColor = randomColor;
  }, 2000);

  return <h1>Seconds passed: {count}</h1>;
}
```

```
import { useState } from 'react';
import { useInterval } from './useInterval.js';

export function useCounter(delay) {
  const [count, setCount] = useState(0);
  useInterval(() => {
    setCount(c => c + 1);
  }, delay);
  return count;
}
```

```
import { useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export function useInterval(onTick, delay) {
  useEffect(() => {
    const id = setInterval(onTick, delay);
    return () => {
      clearInterval(id);
    };
  }, [onTick, delay]);
}
```

Inside `useInterval`, wrap the tick callback into an Effect Event, as you did [earlier on this page](#).

This will allow you to omit `onTick` from dependencies of your Effect. The Effect won't re-synchronize on every re-render of the component, so the page background color change interval won't get reset every second before it has a chance to fire.

With this change, both intervals work as expected and don't interfere with each other:

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  }
}
```

```

    },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}

```

```

import { useCounter } from './useCounter.js';
import { useInterval } from './useInterval.js';

export default function Counter() {
  const count = useCounter(1000);

  useInterval(() => {
    const randomColor = `hsla(${Math.random() * 360}, 100%, 50%, 0.2)`;
    document.body.style.backgroundColor = randomColor;
  }, 2000);

  return <h1>Seconds passed: {count}</h1>;
}

```

```

import { useState } from 'react';
import { useInterval } from './useInterval.js';

export function useCounter(delay) {
  const [count, setCount] = useState(0);
  useInterval(() => {
    setCount(c => c + 1);
  }, delay);
  return count;
}

```

```

import { useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export function useInterval(callback, delay) {
  const onTick = useEffectEvent(callback);
  useEffect(() => {
    const id = setInterval(onTick, delay);
    return () => clearInterval(id);
  }, [delay]);
}

```

Implement a staggering movement [{/implement-a-staggering-movement/}](#)

In this example, the `usePointerPosition()` Hook tracks the current pointer position. Try moving your cursor or your finger over the preview area and see the red dot follow your movement. Its position is saved in the `pos1` variable.

In fact, there are five (!) different red dots being rendered. You don't see them because currently they all appear at the same position. This is what you need to fix. What you want to implement instead is a "staggered" movement: each dot should "follow" the previous dot's path. For example, if you quickly move your cursor, the first dot should follow it immediately, the second dot should follow the first dot with a small delay, the third dot should follow the second dot, and so on.

You need to implement the `useDelayedValue` custom Hook. Its current implementation returns the `value` provided to it. Instead, you want to return the value back from `delay` milliseconds ago. You might need some state and an Effect to do this.

After you implement `useDelayedValue`, you should see the dots move following one another.

You'll need to store the `delayedValue` as a state variable inside your custom Hook. When the `value` changes, you'll want to run an Effect. This Effect should update `delayedValue` after the `delay`. You might find it helpful to call `setTimeout`.

Does this Effect need cleanup? Why or why not?

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

export function useDelayedValue(value, delay) {
  // TODO: Implement this Hook
  return value;
}

export default function Canvas() {
  const pos1 = usePointerPosition();
  const pos2 = useDelayedValue(pos1, 100);
  const pos3 = useDelayedValue(pos2, 200);
  const pos4 = useDelayedValue(pos3, 100);
  const pos5 = useDelayedValue(pos3, 50);
  return (
    <>
      <Dot position={pos1} opacity={1} />
      <Dot position={pos2} opacity={0.8} />
      <Dot position={pos3} opacity={0.6} />
      <Dot position={pos4} opacity={0.4} />
      <Dot position={pos5} opacity={0.2} />
    </>
  );
}

function Dot({ position, opacity }) {
  return (
    <div style={{
      position: 'absolute',
      backgroundColor: 'pink',
      borderRadius: '50%',
      opacity,
      transform: `translate(${position.x}px, ${position.y}px)`,
      pointerEvents: 'none',
      left: -20,
      top: -20,
      width: 40,
      height: 40,
    }} />
  );
}

import { usePointerPosition } from './usePointerPosition.js';

export function useDelayedValue(value, delay) {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  useEffect(() => {
    function handleMove(e) {
      setPosition({ x: e.clientX, y: e.clientY });
    }
    window.addEventListener('mousemove', handleMove);
    return () => {
      window.removeEventListener('mousemove', handleMove);
    };
  }, []);
}
```

```

        setPosition({ x: e.clientX, y: e.clientY });
    }
    window.addEventListener('pointermove', handleMove);
    return () => window.removeEventListener('pointermove', handleMove);
}, []);
return position;
}

body { min-height: 300px; }

```

Here is a working version. You keep the `delayedValue` as a state variable. When `value` updates, your Effect schedules a timeout to update the `delayedValue`. This is why the `delayedValue` always "lags behind" the actual `value`.

```

import { useState, useEffect } from 'react';
import { usePointerPosition } from './usePointerPosition.js';

function useDelayedValue(value, delay) {
  const [delayedValue, setDelayedValue] = useState(value);

  useEffect(() => {
    setTimeout(() => {
      setDelayedValue(value);
    }, delay);
  }, [value, delay]);

  return delayedValue;
}

export default function Canvas() {
  const pos1 = usePointerPosition();
  const pos2 = useDelayedValue(pos1, 100);
  const pos3 = useDelayedValue(pos2, 200);
  const pos4 = useDelayedValue(pos3, 100);
  const pos5 = useDelayedValue(pos3, 50);
  return (
    <>
      <Dot position={pos1} opacity={1} />
      <Dot position={pos2} opacity={0.8} />
      <Dot position={pos3} opacity={0.6} />
      <Dot position={pos4} opacity={0.4} />
      <Dot position={pos5} opacity={0.2} />
    </>
  );
}

function Dot({ position, opacity }) {
  return (
    <div style={{
      position: 'absolute',
      backgroundColor: 'pink',
      borderRadius: '50%',
      opacity,
      transform: `translate(${position.x}px, ${position.y}px)`,
      pointerEvents: 'none',
      left: -20,
      top: -20,
      width: 40,
      height: 40,
    }}>
  
```

```
    }});
}
```

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

export function usePointerPosition() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  useEffect(() => {
    function handleMove(e) {
      setPosition({ x: e.clientX, y: e.clientY });
    }
    window.addEventListener('pointermove', handleMove);
    return () => window.removeEventListener('pointermove', handleMove);
  }, []);
  return position;
}

body { min-height: 300px; }
```

Note that this Effect *does not* need cleanup. If you called `clearTimeout` in the cleanup function, then each time the `value` changes, it would reset the already scheduled timeout. To keep the movement continuous, you want all the timeouts to fire.

Scaling Up with Reducer and Context

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

- 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 {/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:

```
import { 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',
      text: text,
    });
  }

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

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

  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, {
        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 = [
];

```

```

import { useState } from 'react';

export default function AddTask({ onAddTask }) {
  const [text, setText] = useState('');
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button onClick={() => {
        setText('');
        onAddTask(text);
      }}>Add</button>
    </>
  )
}

```

```

import { useState } from 'react';

export default function TaskList({
  tasks,
  onChangeTask,
  onDeleteTask
}

```

```

}) {
  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>
          <Task
            task={task}
            onChange={onChangeTask}
            onDelete={onDeleteTask}
          />
        </li>
      )));
    </ul>
  );
}

function Task({ task, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={e => {
            onChange({
              ...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 => {
          onChange({
            ...task,
            done: e.target.checked
          });
        }}
      />
      {taskContent}
      <button onClick={() => onDelete(task.id)}>
        Delete
      </button>
    </label>
  );
}

```

```
</label>
);
```

```
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

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:

1. **Create** the context.
2. **Put** state and dispatch into context.
3. **Use** context anywhere in the tree.

Step 1: Create the context `{/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:

```

import { 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',
      text: text,
    });
  }

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

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

  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, {
        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 = [
];

```

```

import { createContext } from 'react';

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);

```

```

import { useState } from 'react';

export default function AddTask({ onAddTask }) {
    const [text, setText] = useState('');
    return (
        <>
            <input
                placeholder="Add task"
                value={text}
                onChange={e => setText(e.target.value)}
            />
            <button onClick={() => {
                setText('');
                onAddTask(text);
            }}>Add</button>
        </>
    )
}

```

```

import { useState } from 'react';

export default function TaskList({
    tasks,
    onChangeTask,
    onDeleteTask
}) {
    return (
        <ul>
            {tasks.map(task => (
                <li key={task.id}>
                    <Task
                        task={task}
                        onChange={onChangeTask}
                        onDelete={onDeleteTask}
                    />
                </li>
            ))}
        </ul>
    );
}

```

```

function Task({ task, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={e => {
            onChange({
              ...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 => {
          onChange({
            ...task,
            done: e.target.checked
          });
        }}
      />
      {taskContent}
      <button onClick={() => onDelete(task.id)}>
        Delete
      </button>
    </label>
  );
}

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

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 {/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:

```
import { 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',
      text: text,
    });
  }

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

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

  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        <h1>Day off in Kyoto</h1>
        <AddTask
          onAddTask={handleAddTask}
        >
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}
```

```

        />
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </TasksDispatchContext.Provider>
  </TasksContext.Provider>
);
}

function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [...tasks, {
        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 = [
];

import { createContext } from 'react';

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);

import { useState } from 'react';

export default function AddTask({ onAddTask }) {
  const [text, setText] = useState('');
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button onClick={() => {
        setText('');
        onAddTask(text);
      }}>Add</button>
    </>
  );
}

```

```

        </>
    )
}

import { useState } from 'react';

export default function TaskList({
  tasks,
  onChangeTask,
  onDeleteTask
}) {
  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>
          <Task
            task={task}
            onChange={onChangeTask}
            onDelete={onDeleteTask}
          />
        </li>
      ))}
    </ul>
  );
}

function Task({ task, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let taskContent;
  if (isEditing) {
    taskContent = (
      <>
        <input
          value={task.text}
          onChange={e => {
            onChange({
              ...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 => {
          onChange({
            ...
          });
        }} />
    </label>
  );
}

```

```

        ...task,
        done: e.target.checked
    );
}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>
    Delete
</button>
</label>
);
}

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

In the next step, you will remove prop passing.

Step 3: Use context anywhere in the tree {/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',
        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:

```

import { 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
  );

  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        <h1>Day off in Kyoto</h1>
        <AddTask />
        <TaskList />
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}

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

const initialTasks = [
];

import { createContext } from 'react';

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);

```

```

import { useState, useContext } from 'react';
import { TasksDispatchContext } from './TasksContext.js';

export default function AddTask() {
  const [text, setText] = useState('');
  const dispatch = useContext(TasksDispatchContext);
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button onClick={() => {
        setText('');
        dispatch({
          type: 'added',
          text: text,
        });
      }}>Add</button>
    </>
  );
}

let nextId = 3;

```

```

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 = (
      <div>
        {task.text}
      </div>
    );
  }
  return (
    <li key={task.id}>
      {taskContent}
    </li>
  );
}

```

```

        </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',
        ]);
      }}>
        Delete
      </button>
    </label>
  );
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

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 *{/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:

1. It will manage the state with a reducer.
2. It will provide both contexts to components below.
3. It will `take children as a prop` so you can pass JSX to it.

```

export function TasksProvider({ children }) {
  const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);

  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        {children}
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}

```

This removes all the complexity and wiring from your `TaskApp` component:

```

import 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>
  );
}

```

```

import { createContext, useReducer } from 'react';

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);

export function TasksProvider({ children }) {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        {children}
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}

```

```

}

function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'added': {
      return [...tasks, {
        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);
    }
  }
}

const initialTasks = [
];

```

```

import { useState, useContext } from 'react';
import { TasksDispatchContext } from './TasksContext.js';

export default function AddTask() {
  const [text, setText] = useState('');
  const dispatch = useContext(TasksDispatchContext);
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={e => setText(e.target.value)}>
    />
    <button onClick={() => {
      setText('');
      dispatch({
        type: 'added',
        text: text,
      });
    }}>Add</button>
    </>
  );
}

let nextId = 3;

```

```

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: e.target.checked
            }
          });
        }}
      />
      {taskContent}
      <button onClick={() => {
        dispatch({

```

```

        type: 'deleted',
      });
    }});
  Delete
  </button>
</label>
);
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

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:**

```

import 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>
  );
}

```

```

import { createContext, useContext, useReducer } from 'react';
const TasksContext = createContext(null);
const TasksDispatchContext = createContext(null);

export function TasksProvider({ children }) {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

```

```

    return (
      <TasksContext.Provider value={tasks}>
        <TasksDispatchContext.Provider value={dispatch}>
          {children}
        </TasksDispatchContext.Provider>
      </TasksContext.Provider>
    );
}

export function useTasks() {
  return useContext(TasksContext);
}

export function useTasksDispatch() {
  return useContext(TasksDispatchContext);
}

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

const initialTasks = [
];

```

```

import { useState } from 'react';
import { useTasksDispatch } from './TasksContext.js';

export default function AddTask() {
  const [text, setText] = useState('');
  const dispatch = useTasksDispatch();
  return (
    <>
      <input
        placeholder="Add task"
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button onClick={() => {
        setText('');
        dispatch({

```

```

        type: 'added',
        text: text,
    });
}]}>Add</button>
</>
);
}

let nextId = 3;

import { 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',
        });
      }}>
        Delete
      </button>
    </label>
  );
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

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.

Functions 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.

- 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:
 1. Create two contexts (for state and for dispatch functions).
 2. Provide both contexts from the component that uses the reducer.
 3. 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.

'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.

- 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 *{/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:

1. Your component should automatically connect to the selected chat room.
2. 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 *{/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 {/effects-run-when-ever-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*.

```
import { 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} />}
    </>
  );
}

```

```

export function sendMessage(message) {
  console.log('🌐 You sent: ' + message);
}

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting to "' + roomId + '" room at ' + serverUrl +
      '...');
    },
    disconnect() {
      console.log('❌ Disconnected from "' + roomId + '" room at ' + serverUrl);
    }
  };
}

```

```
input, select { margin-right: 20px; }
```

Reactive values and reactive logic {/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 {/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 {/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 {/extracting-non-reactive-logic-out-of-effects/}

Things get more tricky when you want to mix reactive logic with non-reactive logic.

For example, imagine that you want to show a notification when the user connects to the chat. You read the current theme (dark or light) from the props so that you can show the notification in the correct color:

```
function ChatRoom({ roomId, theme }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.on('connected', () => {
      showNotification('Connected!', theme);
    });
    connection.connect();
  // ...
});
```

However, `theme` is a reactive value (it can change as a result of re-rendering), and **every reactive value read by an Effect must be declared as its dependency.** Now you have to specify `theme` as a dependency of your Effect:

```
function ChatRoom({ roomId, theme }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.on('connected', () => {
      showNotification('Connected!', theme);
    });
    connection.connect();
  return () => {
    connection.disconnect()
  };
});
```

```

    };
  }, [roomId, theme]); // ✅ All dependencies declared
  // ...
}

```

Play with this example and see if you can spot the problem with this user experience:

```

{
  "dependencies": {
    "react": "latest",
    "react-dom": "latest",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}

import { 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)}>
      </label>
    </>
  );
}

```

```

        />
      Use dark theme
    </label>
    <hr />
    <ChatRoom
      roomId={roomId}
      theme={isDark ? 'dark' : 'light'}
    />
  </>
);
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  let connectedCallback;
  let timeout;
  return {
    connect() {
      timeout = setTimeout(() => {
        if (connectedCallback) {
          connectedCallback();
        }
      }, 100);
    },
    on(event, callback) {
      if (connectedCallback) {
        throw Error('Cannot add the handler twice.');
      }
      if (event !== 'connected') {
        throw Error('Only "connected" event is supported.');
      }
      connectedCallback = callback;
    },
    disconnect() {
      clearTimeout(timeout);
    }
  };
}

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}

```

```

label { display: block; margin-top: 10px; }

```

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 *{/declaring-an-effect-event/}*

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]); // ✅ 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:

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
```

```

    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
}

import { 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'}>
    </>
  )
}

```

```

        </>
    );
}

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  let connectedCallback;
  let timeout;
  return {
    connect() {
      timeout = setTimeout(() => {
        if (connectedCallback) {
          connectedCallback();
        }
      }, 100);
    },
    on(event, callback) {
      if (connectedCallback) {
        throw Error('Cannot add the handler twice.');
      }
      if (event !== 'connected') {
        throw Error('Only "connected" event is supported.');
      }
      connectedCallback = callback;
    },
    disconnect() {
      clearTimeout(timeout);
    }
  };
}

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}

```

```
label { display: block; margin-top: 10px; }
```

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 [*/reading-latest-props-and-state-with-effect-events/*](#)

This 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]);
  // ✅ All dependencies declared
  // ...
}
```

Now let's say you want to include the number of items in the shopping cart together with every page visit:

```
function Page({ url }) {
  const { items } = useContext(ShoppingCartContext);
  const numberOfItems = items.length;

  useEffect(() => {
    logVisit(url, numberOfItems);
  }, [url]);
  // ❌ React Hook useEffect has a missing dependency: 'numberOfItems'
  // ...
}
```

You used `numberOfItems` inside the Effect, so the linter asks you to add it as a dependency. However, you *don't* want the `logVisit` call to be reactive with respect to `numberOfItems`. If the user puts something into the shopping cart, and the `numberOfItems` changes, this *does not mean* that the user visited the page again. In other words, *visiting the page* is, in some sense, an "event". It happens at a precise moment in time.

Split the code in two parts:

```

function Page({ url }) {
  const { items } = useContext(ShoppingCartContext);
  const numberofItems = items.length;

  const onVisit = useEffectEvent(visitedUrl => {
    logVisit(visitedUrl, numberofItems);
  });

  useEffect(() => {
    onVisit(url);
  }, [url]); // ✅ All dependencies declared
  // ...
}

```

Here, `onVisit` is an Effect Event. The code inside it isn't reactive. This is why you can use `numberofItems` (or any other reactive value!) without worrying that it will cause the surrounding code to re-execute on changes.

On the other hand, the Effect itself remains reactive. Code inside the Effect uses the `url` prop, so the Effect will re-run after every re-render with a different `url`. This, in turn, will call the `onVisit` Effect Event.

As a result, you will call `logVisit` for every change to the `url`, and always read the latest `numberofItems`. However, if `numberofItems` changes on its own, this will not cause any of the code to re-run.

You might be wondering if you could call `onVisit()` with no arguments, and read the `url` inside it:

```

const onVisit = useEffectEvent(() => {
  logVisit(url, numberofItems);
});

useEffect(() => {
  onVisit();
}, [url]);

```

This would work, but it's better to pass this `url` to the Effect Event explicitly. **By passing `url` as an argument to your Effect Event, you are saying that visiting a page with a different `url` constitutes a separate "event" from the user's perspective.** The `visitedUrl` is a *part* of the "event" that happened:

```

const onVisit = useEffectEvent(visitedUrl => {
  logVisit(visitedUrl, numberofItems);
};

useEffect(() => {
  onVisit(url);
}, [url]);

```

Since your Effect Event explicitly "asks" for the `visitedUrl`, now you can't accidentally remove `url` from the Effect's dependencies. If you remove the `url` dependency (causing distinct page visits to be counted as one), the linter will warn you about it. You want `onVisit` to be reactive with regards to the `url`, so instead of reading the `url` inside (where it wouldn't be reactive), you pass it *from* your Effect.

This becomes especially important if there is some asynchronous logic inside the Effect:

```
const onVisit = useEffectEvent(visitedUrl => {
  logVisit(visitedUrl, numberofItems);
});

useEffect(() => {
  setTimeout(() => {
    onVisit(url);
  }, 5000); // Delay logging visits
}, [url]);
```

Here, `url` inside `onVisit` corresponds to the *latest* `url` (which could have already changed), but `visitedUrl` corresponds to the `url` that originally caused this Effect (and this `onVisit` call) to run.

Is it okay to suppress the dependency linter instead? {/is-it-okay-to-suppress-the-dependency-linter-instead/}

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?

```
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(() => {
  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,
    }} />
  </>
);
}

body {
  height: 200px;
}

```

The 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:

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
  }
}
```

```

    "react-scripts": "latest"
},
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",
  "eject": "react-scripts eject"
}
}

```

```

import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function App() {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const [canMove, setCanMove] = useState(true);

  const onMove = useEffectEvent(e => {
    if (canMove) {
      setPosition({ x: e.clientX, y: e.clientY });
    }
  });

  useEffect(() => {
    window.addEventListener('pointermove', onMove);
    return () => window.removeEventListener('pointermove', onMove);
  }, []);
}

return (
  <>
    <label>
      <input type="checkbox"
        checked={canMove}
        onChange={e => setCanMove(e.target.checked)}
      />
      The dot is allowed to move
    </label>
    <hr />
    <div style={{
      position: 'absolute',
      backgroundColor: 'pink',
      borderRadius: '50%',
      opacity: 0.6,
      transform: `translate(${position.x}px, ${position.y}px)`,
      pointerEvents: 'none',
      left: -20,
      top: -20,
      width: 40,
      height: 40,
    }} />
  </>
);
}

```

```

body {
  height: 200px;
}

```

This 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 {/limitations-of-effect-events/}

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 = useEffectEvent(() => {
    setCount(count + 1);
  });

  return <h1>{count}</h1>
}

function useTimer(callback, delay) {
  useEffect(() => {
    const id = setInterval(() => {
      callback();
    }, delay);
    return () => {
      clearInterval(id);
    };
  }, [delay, callback]); // Need to specify "callback" in dependencies
}
```

Instead, always declare Effect Events directly next to the Effects that use them:

```
function Timer() {
  const [count, setCount] = useState(0);
  useTimer(() => {
    setCount(count + 1);
  }, 1000);
  return <h1>{count}</h1>
}

function useTimer(callback, delay) {
  const onTick = useEffectEvent(() => {
    callback();
  });

  useEffect(() => {
    const id = setInterval(() => {
      onTick(); // ✓ Good: Only called locally inside an Effect
    }, delay);
  }, [delay]);
}
```

```

    }, 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.

- 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.

Fix a variable that doesn't update {/fix-a-variable-that-doesnt-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.

To fix this code, it's enough to follow the rules.

```

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);
    };
    // 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);
        }}>{increment}</button>
      </p>
    </>
  );
}

```

```

        }]>-</button>
      <b>{increment}</b>
      <button onClick={() => {
        setIncrement(i => i + 1);
      }}>+</button>
    </p>
  );
}

```

```
button { margin: 10px; }
```

As usual, when you're looking for bugs in Effects, start by searching for linter suppressions.

If you remove the suppression comment, React will tell you that this Effect's code depends on `increment`, but you "lied" to React by claiming that this Effect does not depend on any reactive values (`[]`). Add `increment` to the dependency array:

```

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);
    };
  }, [increment]);

  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>
    </>
  );
}

```

```
button { margin: 10px; }
```

Now, when `increment` changes, React will re-synchronize your Effect, which will restart the interval.

Fix a freezing counter {/fix-a-freezing-counter/}

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, which you can control it with the plus and minus buttons. For example, try pressing the plus button nine times, and notice that the `count` now increases each second by ten rather than by one.

There is a small issue with this user interface. You might notice that if you keep pressing the plus or minus buttons faster than once per second, the timer itself seems to pause. It only resumes after a second passes since the last time you've pressed either button. Find why this is happening, and fix the issue so that the timer ticks on *every* second without interruptions.

It seems like the Effect which sets up the timer "reacts" to the `increment` value. Does the line that uses the current `increment` value in order to call `setCount` really need to be reactive?

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } 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);
    };
  }, [increment]);

  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>
    </p>
  )
}
```

```

        <b>{increment}</b>
        <button onClick={() => {
          setIncrement(i => i + 1);
        }}>+</button>
      </p>
    );
}

```

```
button { margin: 10px; }
```

The issue is that the code inside the Effect uses the `increment` state variable. Since it's a dependency of your Effect, every change to `increment` causes the Effect to re-synchronize, which causes the interval to clear. If you keep clearing the interval every time before it has a chance to fire, it will appear as if the timer has stalled.

To solve the issue, extract an `onTick` Effect Event from the Effect:

```

{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}

```

```

import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function Timer() {
  const [count, setCount] = useState(0);
  const [increment, setIncrement] = useState(1);

  const onTick = useEffectEvent(() => {
    setCount(c => c + increment);
  });

  useEffect(() => {
    const id = setInterval(() => {
      onTick();
    }, 1000);
    return () => {
      clearInterval(id);
    };
  }, []);
}

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>
</>
);
}
}

button { margin: 10px; }

```

Since `onClick` is an Effect Event, the code inside it isn't reactive. The change to `increment` does not trigger any Effects.

Fix a non-adjustable delay [{/fix-a-non-adjustable-delay/}](#)

In this example, you can customize the interval delay. It's stored in a `delay` state variable which is updated by two buttons. However, even if you press the "plus 100 ms" button until the `delay` is 1000 milliseconds (that is, a second), you'll notice that the timer still increments very fast (every 100 ms). It's as if your changes to the `delay` are ignored. Find and fix the bug.

Code inside Effect Events is not reactive. Are there cases in which you would *want* the `setInterval` call to re-run?

```

{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}

```

```

import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function Timer() {
  const [count, setCount] = useState(0);
  const [increment, setIncrement] = useState(1);
  const [delay, setDelay] = useState(100);

  const onTick = useEffectEvent(() => {
    setCount(c => c + increment);
  });

  const onMount = useEffectEvent(() => {
    return setInterval(() => {
      onTick();
    }, delay);
  });
}

```

```

    }, delay);
};

useEffect(() => {
  const id = onMount();
  return () => {
    clearInterval(id);
  }
}, []);

return (
  <>
  <h1>
    Counter: {count}
    <button onClick={() => setCount(0)}>Reset</button>
  </h1>
  <hr />
  <p>
    Increment by:
    <button disabled={increment === 0} onClick={() => {
      setIncrement(i => i - 1);
    }}>-</button>
    <b>{increment}</b>
    <button onClick={() => {
      setIncrement(i => i + 1);
    }}>+</button>
  </p>
  <p>
    Increment delay:
    <button disabled={delay === 100} onClick={() => {
      setDelay(d => d - 100);
    }}>-100 ms</button>
    <b>{delay} ms</b>
    <button onClick={() => {
      setDelay(d => d + 100);
    }}>+100 ms</button>
  </p>
</>
);
}

```

```
button { margin: 10px; }
```

The problem with the above example is that it extracted an Effect Event called `onMount` without considering what the code should actually be doing. You should only extract Effect Events for a specific reason: when you want to make a part of your code non-reactive. However, the `setInterval` call *should* be reactive with respect to the `delay` state variable. If the `delay` changes, you want to set up the interval from scratch! To fix this code, pull all the reactive code back inside the Effect:

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
  }
}
```

```

    "eject": "react-scripts eject"
}

import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function Timer() {
  const [count, setCount] = useState(0);
  const [increment, setIncrement] = useState(1);
  const [delay, setDelay] = useState(100);

  const onTick = useEffectEvent(() => {
    setCount(c => c + increment);
  });

  useEffect(() => {
    const id = setInterval(() => {
      onTick();
    }, delay);
    return () => {
      clearInterval(id);
    }
  }, [delay]);
}

return (
  <>
    <h1>
      Counter: {count}
      <button onClick={() => setCount(0)}>Reset</button>
    </h1>
    <hr />
    <p>
      Increment by:
      <button disabled={increment === 0} onClick={() => {
        setIncrement(i => i - 1);
      }}>-</button>
      <b>{increment}</b>
      <button onClick={() => {
        setIncrement(i => i + 1);
      }}>+</button>
    </p>
    <p>
      Increment delay:
      <button disabled={delay === 100} onClick={() => {
        setDelay(d => d - 100);
      }}>-100 ms</button>
      <b>{delay} ms</b>
      <button onClick={() => {
        setDelay(d => d + 100);
      }}>+100 ms</button>
    </p>
  </>
);
}

```

```
button { margin: 10px; }
```

In general, you should be suspicious of functions like `onMount` that focus on the *timing* rather than the *purpose* of a piece of code. It may feel "more descriptive" at first but it obscures your intent. As a rule of thumb, Effect Events should correspond to something that happens from the *user's perspective*. For example, `onMessage`, `onTick`, `onVisit`, or `onConnected` are good Effect Event names. Code inside them would likely not need to be reactive. On the other hand, `onMount`, `onUpdate`, `onUnmount`, or `onAfterRender` are so generic that it's easy to accidentally put code that *should* be reactive into them. This is why you should name your Effect Events after *what the user thinks has happened*, not when some code happened to run.

Fix a delayed notification [{/fix-a-delayed-notification/}](#)

When you join a chat room, this component shows a notification. However, it doesn't show the notification immediately. Instead, the notification is artificially delayed by two seconds so that the user has a chance to look around the UI.

This almost works, but there is a bug. Try changing the dropdown from "general" to "travel" and then to "music" very quickly. If you do it fast enough, you will see two notifications (as expected!) but they will *both* say "Welcome to music".

Fix it so that when you switch from "general" to "travel" and then to "music" very quickly, you see two notifications, the first one being "Welcome to travel" and the second one being "Welcome to music". (For an additional challenge, assuming you've *already* made the notifications show the correct rooms, change the code so that only the latter notification is displayed.)

Your Effect knows which room it connected to. Is there any information that you might want to pass to your Effect Event?

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

```
import { 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('Welcome to ' + roomId, theme);
  });
}
```

```

useEffect(() => {
  const connection = createConnection(serverUrl, roomId);
  connection.on('connected', () => {
    setTimeout(() => {
      onConnected();
    }, 2000);
  });
  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'}
      />
    </>
  );
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  let connectedCallback;
  let timeout;
  return {
    connect() {
      timeout = setTimeout(() => {
        if (connectedCallback) {
          connectedCallback();
        }
      }, 100);
    },
    on(event, callback) {
      if (connectedCallback) {
        throw Error('Cannot add the handler twice.');
      }
    }
  };
}

```

```

    }
    if (event !== 'connected') {
      throw Error('Only "connected" event is supported.');
    }
    connectedCallback = callback;
  },
  disconnect() {
    clearTimeout(timeout);
  }
};
}

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}

label { display: block; margin-top: 10px; }

```

Inside your Effect Event, `roomId` is the value at the time Effect Event was called.

Your Effect Event is called with a two second delay. If you're quickly switching from the travel to the music room, by the time the travel room's notification shows, `roomId` is already `"music"`. This is why both notifications say "Welcome to music".

To fix the issue, instead of reading the latest `roomId` inside the Effect Event, make it a parameter of your Effect Event, like `connectedRoomId` below. Then pass `roomId` from your Effect by calling `onConnected(roomId)`:

```

{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}

```

```

import { 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((connectedRoomId => {
    showNotification('Welcome to ' + connectedRoomId, theme);
  }));

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.on('connected', () => {
      setTimeout(() => {
        onConnected(roomId);
      }, 2000);
    });
    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)}>
        </input>
        Use dark theme
      </label>
      <hr />
      <ChatRoom
        roomId={roomId}
        theme={isDark ? 'dark' : 'light'}>
      </ChatRoom>
    </>
  );
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  let connectedCallback;
  let timeout;
  return {
    connect() {

```

```

    timeout = setTimeout(() => {
      if (connectedCallback) {
        connectedCallback();
      }
    }, 100);
  },
  on(event, callback) {
    if (connectedCallback) {
      throw Error('Cannot add the handler twice.');
    }
    if (event !== 'connected') {
      throw Error('Only "connected" event is supported.');
    }
    connectedCallback = callback;
  },
  disconnect() {
    clearTimeout(timeout);
  }
};

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}

```

```
label { display: block; margin-top: 10px; }
```

The Effect that had `roomId` set to `"travel"` (so it connected to the `"travel"` room) will show the notification for `"travel"`. The Effect that had `roomId` set to `"music"` (so it connected to the `"music"` room) will show the notification for `"music"`. In other words, `connectedRoomId` comes from your Effect (which is reactive), while `theme` always uses the latest value.

To solve the additional challenge, save the notification timeout ID and clear it in the cleanup function of your Effect:

```
{
  "dependencies": {
    "react": "experimental",
    "react-dom": "experimental",
    "react-scripts": "latest",
    "toastify-js": "1.12.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
  }
}
```

```

    "eject": "react-scripts eject"
}

import { 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((connectedRoomId => {
    showNotification('Welcome to ' + connectedRoomId, theme);
  });

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    let notificationTimeoutId;
    connection.on('connected', () => {
      notificationTimeoutId = setTimeout(() => {
        onConnected(roomId);
      }, 2000);
    });
    connection.connect();
    return () => {
      connection.disconnect();
      if (notificationTimeoutId !== undefined) {
        clearTimeout(notificationTimeoutId);
      }
    };
  }, [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'}
      />
    </>
  );
}

```

```

export function createConnection(serverUrl, roomId) {
  // A real implementation would actually connect to the server
  let connectedCallback;
  let timeout;
  return {
    connect() {
      timeout = setTimeout(() => {
        if (connectedCallback) {
          connectedCallback();
        }
      }, 100);
    },
    on(event, callback) {
      if (connectedCallback) {
        throw Error('Cannot add the handler twice.');
      }
      if (event !== 'connected') {
        throw Error('Only "connected" event is supported.');
      }
      connectedCallback = callback;
    },
    disconnect() {
      clearTimeout(timeout);
    }
  };
}

```

```

import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
  Toastify({
    text: message,
    duration: 2000,
    gravity: 'top',
    position: 'right',
    style: {
      background: theme === 'dark' ? 'black' : 'white',
      color: theme === 'dark' ? 'white' : 'black',
    },
  }).showToast();
}

```

```

label { display: block; margin-top: 10px; }

```

This ensures that already scheduled (but not yet displayed) notifications get cancelled when you change rooms.

[Go to TOC](#)

Sharing State Between Components

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

- How to share state between components by lifting it up
- What are controlled and uncontrolled components

Lifting state up by example {/lifting-state-up-by-example/}

In this example, a parent `Accordion` component renders two separate `Panel`s:

- `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:

```
import { 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>
    </>
  );
}
```

```
h3, p { margin: 5px 0px; }
.panel {
  padding: 10px;
  border: 1px solid #aaa;
}
```

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:

1. **Remove** state from the child components.
2. **Pass** hardcoded data from the common parent.
3. **Add** state to the common parent and pass it down together with the event handlers.

This will allow the `Accordion` component to coordinate both `Panel`s and only expand one at a time.

Step 1: Remove state from the child components {/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 {/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:

```
import { 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>
  );
}
```

```
h3, p { margin: 5px 0px; }
.panel {
  padding: 10px;
  border: 1px solid #aaa;
}
```

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 {/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:

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

export default function Accordion() {
  const [activeIndex, setActiveIndex] = useState(0);
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel
        title="About"
        isActive={activeIndex === 0}
        onShow={() => setActiveIndex(0)}
      >
        With a population of about 2 million, Almaty is Kazakhstan's largest city.
        From 1929 to 1997, it was its capital city.
      </Panel>
      <Panel
        title="Etymology"
        isActive={activeIndex === 1}
        onShow={() => setActiveIndex(1)}
      >
        The name comes from <span lang="kk-KZ">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,  
  onShow  
) {  
  return (  
    <section className="panel">  
      <h3>{title}</h3>  
      {isActive ? (  
        <p>{children}</p>  
      ) : (  
        <button onClick={onShow}>  
          Show  
        </button>  
      )}  
    </section>  
  );
}

```

```

h3, p { margin: 5px 0px; }  
.panel {  
  padding: 10px;  
  border: 1px solid #aaa;  

}

```

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 = true`

When `Accordion`'s `activeIndex` state changes to `1`, the second `Panel` receives `isActive = true` instead

Controlled and uncontrolled components {/controlled-and-uncontrolled-components/}

It 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 {/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](#).

- 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).

Synced inputs {/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.

You'll need to lift their state up into the parent component.

```
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>
);
}

input { margin: 5px; }
label { display: block; }

```

Move the `text` state variable into the parent component along with the `handleChange` handler. Then pass them down as props to both of the `Input` components. This will keep them in sync.

```

import { useState } from 'react';

export default function SyncedInputs() {
  const [text, setText] = useState('');

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <>
      <Input
        label="First input"
        value={text}
        onChange={handleChange}
      />
      <Input
        label="Second input"
        value={text}
        onChange={handleChange}
      />
    </>
  );
}

function Input({ label, value, onChange }) {
  return (
    <label>
      {label}
      {' '}
      <input
        value={value}
        onChange={onChange}
      />
    </label>
  );
}

```

```
input { margin: 5px; }
label { display: block; }
```

Filtering a list {/filtering-a-list/}

In this example, the `searchBar` has its own `query` state that controls the text input. Its parent `FilterableList` component displays a `List` of items, but it doesn't take the search query into account.

Use the `filterItems(foods, query)` function to filter the list according to the search query. To test your changes, verify that typing "s" into the input filters down the list to "Sushi", "Shish kebab", and "Dim sum".

Note that `filterItems` is already implemented and imported so you don't need to write it yourself!

You will want to remove the `query` state and the `handleChange` handler from the `SearchBar`, and move them to the `FilterableList`. Then pass them down to `SearchBar` as `query` and `onChange` props.

```
import { useState } from 'react';
import { foods, filterItems } from './data.js';

export default function FilterableList() {
  return (
    <>
      <SearchBar />
      <hr />
      <List items={foods} />
    </>
  );
}

function SearchBar() {
  const [query, setQuery] = useState('');

  function handleChange(e) {
    setQuery(e.target.value);
  }

  return (
    <label>
      Search:{' '}
      <input
        value={query}
        onChange={handleChange}
      />
    </label>
  );
}

function List({ items }) {
  return (
    <table>
      <tbody>
        {items.map(food => (
          <tr key={food.id}>
            <td>{food.name}</td>
            <td>{food.description}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}
```

```

        </table>
    );
}

export function filterItems(items, query) {
    query = query.toLowerCase();
    return items.filter(item =>
        item.name.split(' ').some(word =>
            word.toLowerCase().startsWith(query)
        )
    );
}

export const foods = [
    {
        name: 'Sushi',
        description: 'Sushi is a traditional Japanese dish of prepared vinegared rice'
    },
    {
        name: 'Dal',
        description: 'The most common way of preparing dal is in the form of a soup to which onions, tomatoes and various spices may be added'
    },
    {
        name: 'Pierogi',
        description: 'Pierogi are filled dumplings made by wrapping unleavened dough around a savoury or sweet filling and cooking in boiling water'
    },
    {
        name: 'Shish kebab',
        description: 'Shish kebab is a popular meal of skewered and grilled cubes of meat.'
    },
    {
        name: 'Dim sum',
        description: 'Dim sum is a large range of small dishes that Cantonese people traditionally enjoy in restaurants for breakfast and lunch'
    },
];

```

Lift the `query` state up into the `FilterableList` component. Call `filterItems(foods, query)` to get the filtered list and pass it down to the `List`. Now changing the query input is reflected in the list:

```

import { useState } from 'react';
import { foods, filterItems } from './data.js';

export default function FilterableList() {
    const [query, setQuery] = useState('');
    const results = filterItems(foods, query);

    function handleChange(e) {
        setQuery(e.target.value);
    }

    return (
        <>
            <SearchBar
                query={query}
                onChange={handleChange}
            />
            <hr />
            <List items={results} />
        </>
    );
}

function SearchBar({ query, onChange }) {

```

```

    return (
      <label>
        Search: {' '}
        <input
          value={query}
          onChange={onChange}
        />
      </label>
    );
}

function List({ items }) {
  return (
    <table>
      <tbody>
        {items.map(food => (
          <tr key={food.id}>
            <td>{food.name}</td>
            <td>{food.description}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}

```

```

export function filterItems(items, query) {
  query = query.toLowerCase();
  return items.filter(item =>
    item.name.split(' ').some(word =>
      word.toLowerCase().startsWith(query)
    )
  );
}

export const foods = [
  {
    name: 'Sushi',
    description: 'Sushi is a traditional Japanese dish of prepared vinegared rice'
  },
  {
    name: 'Dal',
    description: 'The most common way of preparing dal is in the form of a soup to which onions, tomatoes and various spices may be added'
  },
  {
    name: 'Pierogi',
    description: 'Pierogi are filled dumplings made by wrapping unleavened dough around a savoury or sweet filling and cooking in boiling water'
  },
  {
    name: 'Shish kebab',
    description: 'Shish kebab is a popular meal of skewered and grilled cubes of meat.'
  },
  {
    name: 'Dim sum',
    description: 'Dim sum is a large range of small dishes that Cantonese people traditionally enjoy in restaurants for breakfast and lunch'
  }];

```

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. Frameworks provide features that most apps and sites eventually need, including routing, data fetching, and generating HTML.

You need to install [Node.js](#) for local development. You can *also* choose to use Node.js in production, but you don't have to. Many React frameworks support export to a static HTML/CSS/JS folder.

Production-grade React frameworks {/production-grade-react-frameworks/}

Next.js{/nextjs/}

Next.js 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:

```
npx create-next-app
```

If you're new to Next.js, check out the [Next.js tutorial](#).

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. [Fully static Next.js apps](#) can be deployed to any static hosting.

Remix{/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:

```
npx 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/}

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:

```
npx 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/}

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:

```
npx 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.

Can I use React without a framework? {/can-i-use-react-without-a-framework/}

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 you're still not convinced, or your app has unusual constraints not served well by these frameworks and you'd like to roll your own custom setup, we can't stop you--go for it! 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.

Bleeding-edge React frameworks {/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) {/nextjs-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.

Next.js's App Router is **currently in beta and is not yet recommended for production** (as of Mar 2023). To experiment with it in an existing Next.js project, [follow this incremental migration guide](#).

Which features make up the React team's full-stack architecture vision? {/which-features-make-up-the-react-teams-full-stack-architecture-vision/}

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.

"State: A Component's Memory"

Components often need to change what's on the screen as a result of an interaction. Typing into the form should update the input field, clicking "next" on an image carousel should change which image is displayed, clicking "buy" 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*.

- 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 { /when-a-regular-variable-isnt-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!):

```
import { 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>
    </>
  );
}
```

```

export const 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/Sd1AgU0m.jpg',
    alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'
  }, {
    name: 'Ultimate Form',
    artist: 'Barbara Hepworth',
    description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',
    url: 'https://i.imgur.com/2heNQDcm.jpg',
    alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'
  }, {
    name: 'Cavaliere',
    artist: 'Lamidi Olonade Fakeye',
    description: 'Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.',
    url: 'https://i.imgur.com/wIdGuZwm.png',
  }
]

```

```

        alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'
    }, {
        name: 'Big Bellies',
        artist: 'Alina Szapocznikow',
        description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",
        url: 'https://i.imgur.com/AlHTAdDm.jpg',
        alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'
    }, {
        name: 'Terracotta Army',
        artist: 'Unknown Artist',
        description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',
        url: 'https://i.imgur.com/HMFmH6m.jpg',
        alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'
    }, {
        name: 'Lunar Landscape',
        artist: 'Louise Nevelson',
        description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism's geometric abstraction of space and form.',
        url: 'https://i.imgur.com/rN7hY6om.jpg',
        alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'
    }, {
        name: 'Aureole',
        artist: 'Ranjani Shettar',
        description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',
        url: 'https://i.imgur.com/okTpbHhm.jpg',
        alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'
    }, {
        name: 'Hippos',
        artist: 'Taipei Zoo',
        description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',
        url: 'https://i.imgur.com/6o5Vuyu.jpg',
        alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'
    }];

```

```

h2 { margin-top: 10px; margin-bottom: 0; }
h3 {
    margin-top: 5px;
    font-weight: normal;
    font-size: 100%;
}
img { width: 120px; height: 120px; }
button {
    display: block;

```

```
margin-top: 10px;
margin-bottom: 10px;
}
```

The `handleClick` event handler is updating a local variable, `index`. But two things prevent that change from being visible:

1. **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.
2. **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:

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

The `useState` Hook provides those two things:

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

Adding a state variable {/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 `[]` 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:

```

import { 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>
    </>
  );
}

```

```

export const 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/Sd1AgU0m.jpg',
  alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'
}, {
  name: 'Ultimate Form',
  artist: 'Barbara Hepworth',
  description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',
  url: 'https://i.imgur.com/2heNQDcm.jpg',
  alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'
}, {
  name: 'Cavaliere',
  artist: 'Lamidi Olonade Fakeye',
  description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",
  url: 'https://i.imgur.com/wIdGuZwm.png',
  alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'
}, {
  name: 'Big Bellies',
  artist: 'Alina Szapocznikow',
  description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",
  url: 'https://i.imgur.com/AlHTAdDm.jpg',
  alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'
}, {
  name: 'Terracotta Army',
  artist: 'Unknown Artist',
  description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',
  url: 'https://i.imgur.com/HMFmH6m.jpg',
  alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'
}, {
  name: 'Lunar Landscape',
  artist: 'Louise Nevelson',
  description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing'
}

```

```

and gluing them into boxes that reflect the influence of Cubism's geometric
abstraction of space and form.',
  url: 'https://i.imgur.com/rN7hY6om.jpg',
  alt: 'A black matte sculpture where the individual elements are initially
indistinguishable.'
}, {
  name: 'Aureole',
  artist: 'Ranjani Shettar',
  description: 'Shettar merges the traditional and the modern, the natural and the
industrial. Her art focuses on the relationship between man and nature. Her work
was described as compelling both abstractly and figuratively, gravity defying, and
a "fine synthesis of unlikely materials."',
  url: 'https://i.imgur.com/okTpbHhm.jpg',
  alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the
floor. It appears light.'
}, {
  name: 'Hippos',
  artist: 'Taipei Zoo',
  description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged
hippos at play.',
  url: 'https://i.imgur.com/6o5Vuyu.jpg',
  alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if
they were swimming.'
}];

```

```

h2 { margin-top: 10px; margin-bottom: 0; }
h3 {
  margin-top: 5px;
  font-weight: normal;
  font-size: 100%;
}
img { width: 120px; height: 120px; }
button {
  display: block;
  margin-top: 10px;
  margin-bottom: 10px;
}

```

Meet your first Hook *{/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.

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` *{/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`.

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:

1. The **state variable** (`index`) with the value you stored.
2. 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);
```

1. **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.
2. **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.
3. **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.
4. And so on!

Giving a component multiple state variables {/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":

```
import { 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}
    />
  </>
);
}

```

```

export const 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/Sd1AgU0m.jpg',
    alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'
}, {
    name: 'Ultimate Form',
    artist: 'Barbara Hepworth',
    description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',
    url: 'https://i.imgur.com/2heNQDcm.jpg',
    alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'
}, {
    name: 'Cavaliere',
    artist: 'Lamidi Olonade Fakeye',
    description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",
    url: 'https://i.imgur.com/wIdGuZwm.png',
    alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'
}, {
    name: 'Big Bellies',
    artist: 'Alina Szapocznikow',
    description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",
    url: 'https://i.imgur.com/AlHTAdDm.jpg',
    alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'
}, {
    name: 'Terracotta Army',
    artist: 'Unknown Artist',
    description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',
    url: 'https://i.imgur.com/HMFmH6m.jpg',
    alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'
}, {
    name: 'Lunar Landscape',
    artist: 'Louise Nevelson',
    description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism's geometric abstraction of space and form.',
    url: 'https://i.imgur.com/rN7hY6om.jpg',
    alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'
}, {
    name: 'Aureole',
    artist: 'Ranjani Shettar',
    description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work'

```

```

was described as compelling both abstractly and figuratively, gravity defying, and
a "fine synthesis of unlikely materials."',
url: 'https://i.imgur.com/okTpbHhm.jpg',
alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the
floor. It appears light.'
}, {
  name: 'Hippos',
  artist: 'Taipei Zoo',
  description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged
hippos at play.',
  url: 'https://i.imgur.com/6o5Vuyu.jpg',
  alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if
they were swimming.'
}];

```

```

h2 { margin-top: 10px; margin-bottom: 0; }
h3 {
  margin-top: 5px;
  font-weight: normal;
  font-size: 100%;
}
img { width: 120px; height: 120px; }
button {
  display: block;
  margin-top: 10px;
  margin-bottom: 10px;
}

```

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.

How does React know which state to return? [{/how-does-react-know-which-state-to-return/}](#)

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 componentHooks = [];
let currentHookIndex = 0;

// How useState works inside React (simplified).
function useState(initialState) {

```

```

let pair = componentHooks[currentHookIndex];
if (pair) {
  // This is not the first render,
  // so the state pair already exists.
  // Return it and prepare for next Hook call.
  currentHookIndex++;
  return pair;
}

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

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

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

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

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

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

  let sculpture =雕塑列表[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 ${雕塑列表.length}`,
    more: `${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/Sd1AgU0m.jpg',  

  alt: 'A large mosaic sculpture of a whimsical dancing female figure in a  

colorful costume emanating joy.'  

}, {  

  name: 'Ultimate Form',  

  artist: 'Barbara Hepworth',  

  description: 'This abstract bronze sculpture is a part of The Family of Man  

series located at Yorkshire Sculpture Park. Hepworth chose not to create literal  

representations of the world but developed abstract forms inspired by people and  

landscapes.',  

  url: 'https://i.imgur.com/2heNQDcm.jpg',  

  alt: 'A tall sculpture made of three elements stacked on each other reminding of  

a human figure.'  

}, {  

  name: 'Cavaliere',  

  artist: 'Lamidi Olonade Fakeye',  

  description: "Descended from four generations of woodcarvers, Fakeye's work  

blended traditional and contemporary Yoruba themes.",  

  url: 'https://i.imgur.com/wIdGuZwm.png',  

  alt: 'An intricate wood sculpture of a warrior with a focused face on a horse  

adorned with patterns.'  

}, {  

  name: 'Big Bellies',  

  artist: 'Alina Szapocznikow',  

  description: "Szapocznikow is known for her sculptures of the fragmented body as  

a metaphor for the fragility and impermanence of youth and beauty. This sculpture  

depicts two very realistic large bellies stacked on top of each other, each around  

five feet (1,5m) tall.",  

  url: 'https://i.imgur.com/AlHTAdDm.jpg',  

  alt: 'The sculpture reminds a cascade of folds, quite different from bellies in  

classical sculptures.'  

}, {  

  name: 'Terracotta Army',  

  artist: 'Unknown Artist',  

  description: 'The Terracotta Army is a collection of terracotta sculptures  

depicting the armies of Qin Shi Huang, the first Emperor of China. The army  

consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150  

cavalry horses.',  

  url: 'https://i.imgur.com/HMFmH6m.jpg',  

  alt: '12 terracotta sculptures of solemn warriors, each with a unique facial  

expression and armor.'  

}, {  

  name: 'Lunar Landscape',  

  artist: 'Louise Nevelson',  

  description: 'Nevelson was known for scavenging objects from New York City  

debris, which she would later assemble into monumental constructions. In this one,  

she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing  

and gluing them into boxes that reflect the influence of Cubism's geometric  

abstraction of space and form.',  

  url: 'https://i.imgur.com/rN7hY6om.jpg',  

  alt: 'A black matte sculpture where the individual elements are initially  

indistinguishable.'  

}, {  

  name: 'Aureole',  

  artist: 'Ranjani Shettar',  

  description: 'Shettar merges the traditional and the modern, the natural and the  

industrial. Her art focuses on the relationship between man and nature. Her work  

was described as compelling both abstractly and figuratively, gravity defying, and  

a "fine synthesis of unlikely materials."',  

  url: 'https://i.imgur.com/okTpbHhm.jpg',  

  alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the

```

```

    floor. It appears light.'
}, {
  name: 'Hippos',
  artist: 'Taipei Zoo',
  description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged
hippos at play.',
  url: 'https://i.imgur.com/6o5Vuyu.jpg',
  alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if
they were swimming.'
}];

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

```

```

<button id="nextButton">
  Next
</button>
<h3 id="header"></h3>
<button id="moreButton"></button>
<p id="description"></p>
<img id="image">

<style>
* { box-sizing: border-box; }
body { font-family: sans-serif; margin: 20px; padding: 0; }
button { display: block; margin-bottom: 10px; }
</style>

```

button { display: block; margin-bottom: 10px; }

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-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:

```

import Gallery from './Gallery.js';

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

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

```

```

export const 雕塑列表 = [
  {
    name: 'Homenaje a la Neurocirugía',
    artist: 'Marta Colvin Andrade',
    description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.',
    url: 'https://i.imgur.com/Mx7dA2Y.jpg',
    alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'
  },
  {
    name: 'Floralis Genérica',
    artist: 'Eduardo Catalano',
    description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',
    url: 'https://i.imgur.com/ZF6s192m.jpg',
    alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'
  },
  {
    name: 'Eternal Presence',
    artist: 'John Woodrow Wilson',
    description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2.13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',
    url: 'https://i.imgur.com/aTtVpES.jpg',
    alt: 'The sculpture depicting a human head seems ever-present and solemn. It'
  }
]

```

```

    radiates calm and serenity.'
}, {
  name: 'Moai',
  artist: 'Unknown Artist',
  description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',
  url: 'https://i.imgur.com/RCwLEoQm.jpg',
  alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'
}, {
  name: 'Blue Nana',
  artist: 'Niki de Saint Phalle',
  description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',
  url: 'https://i.imgur.com/Sd1AgUOm.jpg',
  alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'
}, {
  name: 'Ultimate Form',
  artist: 'Barbara Hepworth',
  description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',
  url: 'https://i.imgur.com/2heNQDcm.jpg',
  alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'
}, {
  name: 'Cavaliere',
  artist: 'Lamidi Olonade Fakeye',
  description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",
  url: 'https://i.imgur.com/wIdGuZwm.png',
  alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'
}, {
  name: 'Big Bellies',
  artist: 'Alina Szapocznikow',
  description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",
  url: 'https://i.imgur.com/AlHTAdDm.jpg',
  alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'
}, {
  name: 'Terracotta Army',
  artist: 'Unknown Artist',
  description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',
  url: 'https://i.imgur.com/HMFmH6m.jpg',
  alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'
}, {
  name: 'Lunar Landscape',
  artist: 'Louise Nevelson',
  description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one,

```

```

she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing
and gluing them into boxes that reflect the influence of Cubism's geometric
abstraction of space and form.',
  url: 'https://i.imgur.com/rN7hY6om.jpg',
  alt: 'A black matte sculpture where the individual elements are initially
indistinguishable.'
}, {
  name: 'Aureole',
  artist: 'Ranjani Shettar',
  description: 'Shettar merges the traditional and the modern, the natural and the
industrial. Her art focuses on the relationship between man and nature. Her work
was described as compelling both abstractly and figuratively, gravity defying, and
a "fine synthesis of unlikely materials."',
  url: 'https://i.imgur.com/okTpbHhm.jpg',
  alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the
floor. It appears light.'
}, {
  name: 'Hippos',
  artist: 'Taipei Zoo',
  description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged
hippos at play.',
  url: 'https://i.imgur.com/6o5Vuyu.jpg',
  alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if
they were swimming.'
}];

```

```

button { display: block; margin-bottom: 10px; }
.Page > * {
  float: left;
  width: 50%;
  padding: 10px;
}
h2 { margin-top: 10px; margin-bottom: 0; }
h3 {
  margin-top: 5px;
  font-weight: normal;
  font-size: 100%;
}
img { width: 120px; height: 120px; }
button {
  display: block;
  margin-top: 10px;
  margin-bottom: 10px;
}

```

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](#).

- 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.

Complete the gallery `{/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.

```
import { 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}
      />
    </>
  );
}
```

```

export const 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/Sd1AgU0m.jpg',
    alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'
  }, {
    name: 'Ultimate Form',
    artist: 'Barbara Hepworth',
    description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',
    url: 'https://i.imgur.com/2heNQDcm.jpg',
    alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'
  }, {
    name: 'Cavaliere',
    artist: 'Lamidi Olonade Fakeye',
    description: 'Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.',
    url: 'https://i.imgur.com/wIdGuZwm.png',
  }
]

```

```

        alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'
    }, {
        name: 'Big Bellies',
        artist: 'Alina Szapocznikow',
        description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",
        url: 'https://i.imgur.com/AlHTAdDm.jpg',
        alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'
    }, {
        name: 'Terracotta Army',
        artist: 'Unknown Artist',
        description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',
        url: 'https://i.imgur.com/HMFmH6m.jpg',
        alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'
    }, {
        name: 'Lunar Landscape',
        artist: 'Louise Nevelson',
        description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism's geometric abstraction of space and form.',
        url: 'https://i.imgur.com/rN7hY6om.jpg',
        alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'
    }, {
        name: 'Aureole',
        artist: 'Ranjani Shettar',
        description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',
        url: 'https://i.imgur.com/okTpbHhm.jpg',
        alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'
    }, {
        name: 'Hippos',
        artist: 'Taipei Zoo',
        description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',
        url: 'https://i.imgur.com/6o5Vuyu.jpg',
        alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'
    }];

```

```

button { display: block; margin-bottom: 10px; }
.Page > * {
    float: left;
    width: 50%;
    padding: 10px;
}
h2 { margin-top: 10px; margin-bottom: 0; }
h3 {
    margin-top: 5px;
}

```

```

font-weight: normal;
font-size: 100%;

}

img { width: 120px; height: 120px; }

```

This adds a guarding condition inside both event handlers and disables the buttons when needed:

```

import { useState } from 'react';
import { sculptureList } from './data.js';

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

  let hasPrev = index > 0;
  let hasNext = index < sculptureList.length - 1;

  function handlePrevClick() {
    if (hasPrev) {
      setIndex(index - 1);
    }
  }

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

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

  let sculpture = sculptureList[index];
  return (
    <>
      <button
        onClick={handlePrevClick}
        disabled={!hasPrev}>
        >
        Previous
      </button>
      <button
        onClick={handleNextClick}
        disabled={!hasNext}>
        >
        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}
      />
    </>
  );
}

```

```

export const 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/Sd1AgU0m.jpg',
    alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'
  }, {
    name: 'Ultimate Form',
    artist: 'Barbara Hepworth',
    description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',
    url: 'https://i.imgur.com/2heNQDcm.jpg',
    alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'
  }
]

```

```

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

```

```

button { display: block; margin-bottom: 10px; }
.Page > * {
  float: left;

```

```

        width: 50%;
        padding: 10px;
    }
    h2 { margin-top: 10px; margin-bottom: 0; }
    h3 {
        margin-top: 5px;
        font-weight: normal;
        font-size: 100%;
    }
    img { width: 120px; height: 120px; }
}

```

Notice how `hasPrev` and `hasNext` are used *both* for the returned JSX and inside the event handlers! This handy pattern works because event handler functions "close over" any variables declared while rendering.

Fix stuck form inputs {/fix-stuck-form-inputs/}

When you type into the input fields, nothing appears. It's like the input values are "stuck" with empty strings. The `value` of the first `<input>` is set to always match the `firstName` variable, and the `value` for the second `<input>` is set to always match the `lastName` variable. This is correct. Both inputs have `onChange` event handlers, which try to update the variables based on the latest user input (`e.target.value`). However, the variables don't seem to "remember" their values between re-renders. Fix this by using state variables instead.

```

export default function Form() {
    let firstName = '';
    let lastName = '';

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

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

    function handleReset() {
        firstName = '';
        lastName = '';
    }

    return (
        <form onSubmit={e => e.preventDefault()}>
            <input
                placeholder="First name"
                value={firstName}
                onChange={handleFirstNameChange}>
            />
            <input
                placeholder="Last name"
                value={lastName}
                onChange={handleLastNameChange}>
            />
            <h1>Hi, {firstName} {lastName}</h1>
            <button onClick={handleReset}>Reset</button>
        </form>
    );
}

```

```
h1 { margin-top: 10px; }
```

First, import `useState` from React. Then replace `firstName` and `lastName` with state variables declared by calling `useState`. Finally, replace every `firstName = ...` assignment with `setFirstName(...)`, and do the same for `lastName`. Don't forget to update `handleReset` too so that the reset button works.

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

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

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

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

  function handleReset() {
    setFirstName('');
    setLastName('');
  }

  return (
    <form onSubmit={e => e.preventDefault()}>
      <input
        placeholder="First name"
        value={firstName}
        onChange={handleFirstNameChange}>
      />
      <input
        placeholder="Last name"
        value={lastName}
        onChange={handleLastNameChange}>
      />
      <h1>Hi, {firstName} {lastName}</h1>
      <button onClick={handleReset}>Reset</button>
    </form>
  );
}
```

```
h1 { margin-top: 10px; }
```

Fix a crash {/fix-a-crash/}

Here is a small form that is supposed to let the user leave some feedback. When the feedback is submitted, it's supposed to display a thank-you message. However, it crashes with an error message saying "Rendered fewer hooks than expected". Can you spot the mistake and fix it?

Are there any limitations on *where* Hooks may be called? Does this component break any rules? Check if there are any comments disabling the linter checks--this is where the bugs often hide!

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

export default function FeedbackForm() {
```

```

const [isSent, setIsSent] = useState(false);
if (isSent) {
  return <h1>Thank you!</h1>;
} else {
  // eslint-disable-next-line
  const [message, setMessage] = useState('');
  return (
    <form onSubmit={e => {
      e.preventDefault();
      alert(`Sending: "${message}"`);
      setIsSent(true);
    }}>
      <textarea
        placeholder="Message"
        value={message}
        onChange={e => setMessage(e.target.value)}
      />
      <br />
      <button type="submit">Send</button>
    </form>
  );
}
}

```

Hooks can only be called at the top level of the component function. Here, the first `isSent` definition follows this rule, but the `message` definition is nested in a condition.

Move it out of the condition to fix the issue:

```

import { useState } from 'react';

export default function FeedbackForm() {
  const [isSent, setIsSent] = useState(false);
  const [message, setMessage] = useState('');

  if (isSent) {
    return <h1>Thank you!</h1>;
  } else {
    return (
      <form onSubmit={e => {
        e.preventDefault();
        alert(`Sending: "${message}"`);
        setIsSent(true);
      }}>
        <textarea
          placeholder="Message"
          value={message}
          onChange={e => setMessage(e.target.value)}
        />
        <br />
        <button type="submit">Send</button>
      </form>
    );
  }
}

```

Remember, Hooks must be called unconditionally and always in the same order!

You could also remove the unnecessary `else` branch to reduce the nesting. However, it's still important that all calls to Hooks happen *before* the first `return`.

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

export default function FeedbackForm() {
  const [isSent, setIsSent] = useState(false);
  const [message, setMessage] = useState('');

  if (isSent) {
    return <h1>Thank you!</h1>;
  }

  return (
    <form onSubmit={e => {
      e.preventDefault();
      alert(`Sending: "${message}"`);
      setIsSent(true);
    }}>
      <textarea
        placeholder="Message"
        value={message}
        onChange={e => setMessage(e.target.value)}
      />
      <br />
      <button type="submit">Send</button>
    </form>
  );
}
```

Try moving the second `useState` call after the `if` condition and notice how this breaks it again.

If your linter is [configured for React](#), you should see a lint error when you make a mistake like this. If you don't see an error when you try the faulty code locally, you need to set up linting for your project.

Remove unnecessary state [{/remove-unnecessary-state/}](#)

When the button is clicked, this example should ask for the user's name and then display an alert greeting them. You tried to use state to keep the name, but for some reason it always shows "Hello, !".

To fix this code, remove the unnecessary state variable. (We will discuss about [why this didn't work](#) later.)

Can you explain why this state variable was unnecessary?

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

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

  function handleClick() {
    setName(prompt('What is your name?'));
    alert(`Hello, ${name}!`);
  }

  return (
    <button onClick={handleClick}>
      Greet
    </button>
  );
}
```

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

Here is a fixed version that uses a regular `name` variable declared in the function that needs it:

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

export default function FeedbackForm() {
  function handleClick() {
    const name = prompt('What is your name?');
    alert(`Hello, ${name}!`);
  }

  return (
    <button onClick={handleClick}>
      Greet
    </button>
  );
}
```

A state variable is only necessary to keep information between re-renders of a component. Within a single event handler, a regular variable will do fine. Don't introduce state variables when a regular variable works well.

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.

- 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 { /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:

```
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) {
  // ...
}
```

```
label, textarea { margin-bottom: 10px; display: block; }
```

Here's what happens when you click the button:

1. The `onSubmit` event handler executes.
2. `setIsSent(true)` sets `isSent` to `true` and queues a new render.
3. 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-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:

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

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!**

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:

```
import { 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>
    </>
  )
}
```

```

    }]>+3</button>
  </>
}

```

```

button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }

```

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:

1. `setNumber(number + 1)`: `number` is `0` so `setNumber(0 + 1)`.
 - React prepares to change `number` to `1` on the next render.
2. `setNumber(number + 1)`: `number` is `0` so `setNumber(0 + 1)`.
 - React prepares to change `number` to `1` on the next render.
3. `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 { /state-over-time/ }

Well, that was fun. Try to guess what clicking this button will alert:

```
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>
    </>
  )
}

button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

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!

```
import { 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>
    </>
  )
}

button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

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:

1. You press the "Send" button, sending "Hello" to Alice.
2. 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:

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

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

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

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

```
label, textarea { margin-bottom: 10px; display: block; }
```

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!

- 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.

Implement a traffic light [{/implement-a-traffic-light/}](#)

Here is a crosswalk light component that toggles on when the button is pressed:

```
import { 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>
    </>
  );
}
```

```
h1 { margin-top: 20px; }
```

Add 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?

Your `alert` should look like this:

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

export default function TrafficLight() {
```

```
const [walk, setWalk] = useState(true);

function handleClick() {
  setWalk(!walk);
  alert(walk ? 'Stop is next' : 'Walk is next');
}

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

```
h1 { margin-top: 20px; }
```

Whether you put it before or after the `setWalk` call makes no difference. That render's value of `walk` is fixed. Calling `setWalk` will only change it for the *next* render, but will not affect the event handler from the previous render.

This line might seem counter-intuitive at first:

```
alert(walk ? 'Stop is next' : 'Walk is next');
```

But it makes sense if you read it as: "If the traffic light shows 'Walk now', the message should say 'Stop is next.'" The `walk` variable inside your event handler matches that render's value of `walk` and does not change.

You can verify that this is correct by applying the substitution method. When `walk` is `true`, you get:

```
<button onClick={() => {
  setWalk(false);
  alert('Stop is next');
}}>
  Change to Stop
</button>
<h1 style={{color: 'darkgreen'}}>
  Walk
</h1>
```

So clicking "Change to Stop" queues a render with `walk` set to `false`, and alerts "Stop is next".

'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.

- 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? {/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).

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 {/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 {/how-to-write-an-effect/}

To write an Effect, follow these three steps:

1. **Declare an Effect.** By default, your Effect will run after every render.
2. **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*.
3. **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 {/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:

```
import { 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"
      />
    </>
  );
}

button { display: block; margin-bottom: 20px; }
video { width: 250px; }
```

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:

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

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  });

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  return (
    <>
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
        isPlaying={isPlaying}
        src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
      />
    </>
  );
}
```

```

        </>
    );
}

```

```

button { display: block; margin-bottom: 20px; }
video { width: 250px; }

```

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.

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 {/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:

```

import { 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"
      />
    </>
  );
}

input, button { display: block; margin-bottom: 20px; }
video { width: 250px; }

```

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'`:

```

import { 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"
    />
  </>
);
}

input, button { display: block; margin-bottom: 20px; }
video { width: 250px; }

```

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:

```

import { 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"
/>
</>
)
}

```

```

input, button { display: block; margin-bottom: 20px; }
video { width: 250px; }

```

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](#).

The 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.

Why was the ref omitted from the dependency array? {/why-was-the-ref-omitted-from-the-dependency-array/}

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 [{/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:

```
import { 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>;
}
```

```
export function createConnection() {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✅ Connecting...');
    },
    disconnect() {
      console.log('❌ Disconnected.');
    }
  };
}
```

```
input { display: block; margin-bottom: 20px; }
```

This Effect only runs on mount, so you might expect "✅ Connecting..." to be printed once in the console. **However, if you check the console, "✅ Connecting..." gets printed twice. Why does it happen?**

Imagine the `ChatRoom` component is a part of a larger app with many different screens. The user starts their journey on the `ChatRoom` page. The component mounts and calls `connection.connect()`. Then imagine the user navigates to another screen--for example, to the Settings page. The `ChatRoom` component unmounts. Finally, the user clicks Back and `ChatRoom` mounts again. This would set up a second connection--but the first connection was never destroyed! As the user navigates across the app, the connections would keep piling up.

Bugs like this are easy to miss without extensive manual testing. To help you spot them quickly, in development React remounts every component once immediately after its initial mount.

Seeing the "✅ Connecting..." log twice helps you notice the real issue: your code doesn't close the connection when the component unmounts.

To fix the issue, return a *cleanup function* from your Effect:

```
useEffect(() => {
  const connection = createConnection();
  connection.connect();
  return () => {
    connection.disconnect();
  };
}, []);
```

React will call your cleanup function each time before the Effect runs again, and one final time when the component unmounts (gets removed). Let's see what happens when the cleanup function is implemented:

```

import { 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>;
}

export function createConnection() {
  // A real implementation would actually connect to the server
  return {
    connect() {
      console.log('✓ Connecting...');
    },
    disconnect() {
      console.log('✗ Disconnected.');
    }
  };
}

```

`input { display: block; margin-bottom: 20px; }`

Now you get three console logs in development:

1. "✓ Connecting..."
2. "✗ Disconnected."
3. "✓ Connecting..."

This is the correct behavior in development. By remounting your component, React verifies that navigating away and back would not break your code. Disconnecting and then connecting again is exactly what should happen! When you implement the cleanup well, there should be no user-visible difference between running the Effect once vs running it, cleaning it up, and running it again. There's an extra connect/disconnect call pair because React is probing your code for bugs in development. This is normal--don't try to make it go away!

In production, you would only see "✓ Connecting..." printed once. Remounting components only happens in development to help you find Effects that need cleanup. You can turn off [Strict Mode](#) to opt out of the development behavior, but we recommend keeping it on. This lets you find many bugs like the one above.

How to handle the Effect firing twice in development? *{/how-to-handle-the-effect-firing-twice-in-development/}*

React intentionally remounts your components in development to find bugs like in the last example. **The right question isn't "how to run an Effect once", but "how to fix my Effect so that it works after remounting".**

Usually, the answer is to implement the cleanup function. The cleanup function should stop or undo whatever the Effect was doing. The rule of thumb is that the user shouldn't be able to distinguish between the Effect running once (as in production) and a *setup* → *cleanup* → *setup* sequence (as you'd see in development).

Most of the Effects you'll write will fit into one of the common patterns below.

Controlling non-React widgets {/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 {/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 {/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 {/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.

What are good alternatives to data fetching in Effects? *{/what-are-good-alternatives-to-data-fetching-in-effects/}*

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 *{/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 {/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 {/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() {
  // ✅ Buying is an event because it is caused by a particular interaction.
  fetch('/api/buy', { method: 'POST' });
}
```

This illustrates that if remounting breaks the logic of your application, this usually uncovers existing bugs. From the user's perspective, visiting a page shouldn't be different from visiting it, clicking a link, and pressing Back. React verifies that your components abide by this principle by remounting them once in development.

Putting it all together {/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":

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

function Playground() {
  const [text, setText] = useState('a');

  useEffect(() => {
    function onTimeout() {
      console.log('⌚ ' + 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:{" "}
        <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>
    </>
  );
}
```

```

    </button>
    {show && <hr />}
    {show && <Playground />}
  );
}

```

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](#).

Each render has its own Effects [{/each-render-has-its-own-effects/}](#)

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 [{/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 {/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 {/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 {/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 {/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.

- 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.

Focus a field on mount {/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.)

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

```
import { useState } from 'react';
import MyInput from './MyInput.js';

export default function Form() {
  const [show, setShow] = useState(false);
  const [name, setName] = useState('Taylor');
  const [upper, setUpper] = useState(false);
  return (
    <>
      <button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'}</button>
    <br />
    <hr />
    {show && (
      <>
        <label>
          Enter your name:
          <MyInput
            value={name}
            onChange={e => setName(e.target.value)}
          />
        </label>
        <label>
          <input
            type="checkbox"
            checked={upper}
            onChange={e => setUpper(e.target.checked)}
          />
          Make it uppercase
        </label>
        <p>Hello, <b>{upper ? name.toUpperCase() : name}</b></p>
      </>
    )}
  );
}
```

```
label {
  display: block;
  margin-top: 20px;
  margin-bottom: 20px;
}
```

```
body {
  min-height: 150px;
}
```

To 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.

Calling `ref.current.focus()` during render is wrong because it is a *side effect*. Side effects should either be placed inside an event handler or be declared with `useEffect`. In this case, the side effect is *caused by* the component appearing rather than by any specific interaction, so it makes sense to put it in an Effect.

To fix the mistake, wrap the `ref.current.focus()` call into an Effect declaration. Then, to ensure that this Effect runs only on mount rather than after every render, add the empty `[]` dependencies to it.

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

export default function MyInput({ value, onChange }) {
  const ref = useRef(null);

  useEffect(() => {
    ref.current.focus();
  }, []);

  return (
    <input
      ref={ref}
      value={value}
      onChange={onChange}
    />
  );
}
```

```
import { useState } from 'react';
import MyInput from './MyInput.js';

export default function Form() {
  const [show, setShow] = useState(false);
  const [name, setName] = useState('Taylor');
  const [upper, setUpper] = useState(false);
  return (
    <>
      <button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'}

```

```

        />
      </label>
      <label>
        <input
          type="checkbox"
          checked={upper}
          onChange={e => setUpper(e.target.checked)}
        />
        Make it uppercase
      </label>
      <p>Hello, <b>{upper ? name.toUpperCase() : name}</b></p>
    </>
  )}
</>
);
}
}

```

```

label {
  display: block;
  margin-top: 20px;
  margin-bottom: 20px;
}

body {
  min-height: 150px;
}

```

Focus a field conditionally {/focus-a-field-conditionally/}

This form renders two `<MyInput />` components.

Press "Show form" and notice that the second field automatically gets focused. This is because both of the `<MyInput />` components try to focus the field inside. When you call `focus()` for two input fields in a row, the last one always "wins".

Let's say you want to focus the first field. The first `MyInput` component now receives a boolean `shouldFocus` prop set to `true`. Change the logic so that `focus()` is only called if the `shouldFocus` prop received by `MyInput` is `true`.

```

import { useEffect, useRef } from 'react';

export default function MyInput({ shouldFocus, value, onChange }) {
  const ref = useRef(null);

  // TODO: call focus() only if shouldFocus is true.
  useEffect(() => {
    ref.current.focus();
  }, []);

  return (
    <input
      ref={ref}
      value={value}
      onChange={onChange}
    />
  );
}

```

```

import { useState } from 'react';
import MyInput from './MyInput.js';

export default function Form() {
  const [show, setShow] = useState(false);
  const [firstName, setFirstName] = useState('Taylor');
  const [lastName, setLastName] = useState('Swift');
  const [upper, setUpper] = useState(false);
  const name = firstName + ' ' + lastName;
  return (
    <>
      <button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'}</button>
    form</>
    <br />
    <hr />
    {show && (
      <>
        <label>
          Enter your first name:
          <MyInput
            value={firstName}
            onChange={e => setFirstName(e.target.value)}
            shouldFocus={true}
          />
        </label>
        <label>
          Enter your last name:
          <MyInput
            value={lastName}
            onChange={e => setLastName(e.target.value)}
            shouldFocus={false}
          />
        </label>
        <p>Hello, <b>{upper ? name.toUpperCase() : name}</b></p>
      </>
    )}
  );
}

```

```

label {
  display: block;
  margin-top: 20px;
  margin-bottom: 20px;
}

body {
  min-height: 150px;
}

```

To verify your solution, press "Show form" and "Hide form" repeatedly. When the form appears, only the `first` input should get focused. This is because the parent component renders the first input with `shouldFocus={true}` and the second input with `shouldFocus={false}`. Also check that both inputs still work and you can type into both of them.

You can't declare an Effect conditionally, but your Effect can include conditional logic.

Put the conditional logic inside the Effect. You will need to specify `shouldFocus` as a dependency because you are using it inside the Effect. (This means that if some input's `shouldFocus` changes from `false` to `true`, it will focus after mount.)

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

export default function MyInput({ shouldFocus, value, onChange }) {
  const ref = useRef(null);

  useEffect(() => {
    if (shouldFocus) {
      ref.current.focus();
    }
  }, [shouldFocus]);

  return (
    <input
      ref={ref}
      value={value}
      onChange={onChange}
    />
  );
}
```

```
import { useState } from 'react';
import MyInput from './MyInput.js';

export default function Form() {
  const [show, setShow] = useState(false);
  const [firstName, setFirstName] = useState('Taylor');
  const [lastName, setLastName] = useState('Swift');
  const [upper, setUpper] = useState(false);
  const name = firstName + ' ' + lastName;
  return (
    <>
      <button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'}</button>
    form</>
      <br />
      <hr />
      {show && (
        <>
          <label>
            Enter your first name:
            <MyInput
              value={firstName}
              onChange={e => setFirstName(e.target.value)}
              shouldFocus={true}
            />
          </label>
          <label>
            Enter your last name:
            <MyInput
              value={lastName}
              onChange={e => setLastName(e.target.value)}
              shouldFocus={false}
            />
          </label>
          <p>Hello, <b>{upper ? name.toUpperCase() : name}</b></p>
        </>
      )}
    
```

```

        )}
      </>
    );
}

```

```

label {
  display: block;
  margin-top: 20px;
  margin-bottom: 20px;
}

body {
  min-height: 150px;
}

```

Fix an interval that fires twice [{/fix-an-interval-that-fires-twice/}](#)

This `Counter` component displays a counter that should increment every second. On mount, it calls `setInterval`. This causes `onTick` to run every second. The `onTick` function increments the counter.

However, instead of incrementing once per second, it increments twice. Why is that? Find the cause of the bug and fix it.

Keep in mind that `setInterval` returns an interval ID, which you can pass to `clearInterval` to stop the interval.

```

import { useState, useEffect } from 'react';

export default function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    function onTick() {
      setCount(c => c + 1);
    }

    setInterval(onTick, 1000);
  }, []);

  return <h1>{count}</h1>;
}

```

```

import { useState } from 'react';
import Counter from './Counter.js';

export default function Form() {
  const [show, setShow] = useState(false);
  return (
    <>
      <button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'}

```

```

label {
  display: block;
  margin-top: 20px;
  margin-bottom: 20px;
}

body {
  min-height: 150px;
}

```

When [Strict Mode](#) is on (like in the sandboxes on this site), React remounts each component once in development. This causes the interval to be set up twice, and this is why each second the counter increments twice.

However, React's behavior is not the *cause* of the bug: the bug already exists in the code. React's behavior makes the bug more noticeable. The real cause is that this Effect starts a process but doesn't provide a way to clean it up.

To fix this code, save the interval ID returned by `setInterval`, and implement a cleanup function with `clearInterval`:

```

import { useState, useEffect } from 'react';

export default function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    function onTick() {
      setCount(c => c + 1);
    }

    const intervalId = setInterval(onTick, 1000);
    return () => clearInterval(intervalId);
  }, []);

  return <h1>{count}</h1>;
}

```

```

import { useState } from 'react';
import Counter from './Counter.js';

export default function App() {
  const [show, setShow] = useState(false);
  return (
    <>
      <button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'}
      <br />
      <hr />
      {show && <Counter />}
    </>
  );
}

```

```

label {
  display: block;
  margin-top: 20px;
  margin-bottom: 20px;
}

```

```
}
```

```
body {
  min-height: 150px;
}
```

In development, React will still remount your component once to verify that you've implemented cleanup well. So there will be a `setInterval` call, immediately followed by `clearInterval`, and `setInterval` again. In production, there will be only one `setInterval` call. The user-visible behavior in both cases is the same: the counter increments once per second.

Fix fetching inside an Effect [{/fix-fetching-inside-an-effect/}](#)

This component shows the biography for the selected person. It loads the biography by calling an asynchronous function `fetchBio(person)` on mount and whenever `person` changes. That asynchronous function returns a `Promise` which eventually resolves to a string. When fetching is done, it calls `setBio` to display that string under the select box.

```
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(() => {
    setBio(null);
    fetchBio(person).then(result => {
      setBio(result);
    });
  }, [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>
    </>
  );
}
```

```
export async function fetchBio(person) {
  const delay = person === 'Bob' ? 2000 : 200;
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(`This is ${person}'s bio.`);
    }, delay);
  })
}
```

There is a bug in this code. Start by selecting "Alice". Then select "Bob" and then immediately after that select "Taylor". If you do this fast enough, you will notice that bug: Taylor is selected, but the paragraph below says "This is Bob's bio."

Why does this happen? Fix the bug inside this Effect.

If an Effect fetches something asynchronously, it usually needs cleanup.

To trigger the bug, things need to happen in this order:

- Selecting 'Bob' triggers `fetchBio('Bob')`
- Selecting 'Taylor' triggers `fetchBio('Taylor')`
- **Fetching 'Taylor' completes before fetching 'Bob'**
- The Effect from the 'Taylor' render calls `setBio('This is Taylor's bio')`
- Fetching 'Bob' completes
- The Effect from the 'Bob' render calls `setBio('This is Bob's bio')`

This is why you see Bob's bio even though Taylor is selected. Bugs like this are called [race conditions](#) because two asynchronous operations are "racing" with each other, and they might arrive in an unexpected order.

To fix this race condition, add a cleanup function:

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

```

        </>
    );
}

export async function fetchBio(person) {
  const delay = person === 'Bob' ? 2000 : 200;
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(`This is ${person}'s bio.`);
    }, delay);
  })
}

```

Each render's Effect has its own `ignore` variable. Initially, the `ignore` variable is set to `false`. However, if an Effect gets cleaned up (such as when you select a different person), its `ignore` variable becomes `true`. So now it doesn't matter in which order the requests complete. Only the last person's Effect will have `ignore` set to `false`, so it will call `setBio(result)`. Past Effects have been cleaned up, so the `if (!ignore)` check will prevent them from calling `setBio`:

- Selecting '`Bob`' triggers `fetchBio('Bob')`
- Selecting '`Taylor`' triggers `fetchBio('Taylor')` **and cleans up the previous (Bob's) Effect**
- Fetching '`Taylor`' completes *before* fetching '`Bob`'
- The Effect from the '`Taylor`' render calls `setBio('This is Taylor's bio')`
- Fetching '`Bob`' completes
- The Effect from the '`Bob`' render **does not do anything because its `ignore` flag was set to `true`**

In addition to ignoring the result of an outdated API call, you can also use `AbortController` to cancel the requests that are no longer needed. However, by itself this is not enough to protect against race conditions. More asynchronous steps could be chained after the fetch, so using an explicit flag like `ignore` is the most reliable way to fix this type of problems.

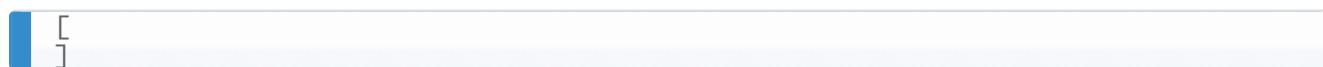
Thinking in React

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 *{/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:



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 *{/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:

```

```

1. `FilterableProductTable` (grey) contains the entire app.

2. `SearchBar` (blue) receives the user input.
3. `ProductTable` (lavender) displays and filters the list according to the user input.
4. `ProductCategoryRow` (green) displays a heading for each category.
5. `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 {/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.

```
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 }) {
  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 = [
];

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

```

```

body {
  padding: 5px
}
label {
  display: block;
  margin-top: 5px;
  margin-bottom: 5px;
}
th {
  padding-top: 10px;
}
td {
  padding: 2px;
  padding-right: 40px;
}

```

(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.

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 [{/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:

1. The original list of products
2. The search text the user has entered
3. The value of the checkbox
4. 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:

1. The original list of products is **passed in as props, so it's not state**.
2. The search text seems to be state since it changes over time and can't be computed from anything.
3. The value of the checkbox seems to be state since it changes over time and can't be computed from anything.
4. 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!

Props vs State {/props-vs-state/}

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 {/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:

1. Identify every component that renders something based on that state.
2. Find their closest common parent component--a component above them all in the hierarchy.
3. Decide where the state should live:
 1. Often, you can put the state directly into their common parent.
 2. You can also put the state into some component above their common parent.
 3. 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:

1. 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).

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

`FilterableProductTable`.

3. 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:

```
import { 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(

```

react

```
<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 = [
];

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

```
body {
  padding: 5px
}
label {
  display: block;
  margin-top: 5px;
  margin-bottom: 5px;
}
th {
  padding-top: 5px;
}
td {
  padding: 2px;
}
```

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

You 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 {/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:

```
<input
  type="text"
  value={filterText}
  placeholder="Search..."
  onChange={(e) => onFilterTextChange(e.target.value)} />
```

Now the application fully works!

```
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()
      ) === 0
    ) {
      if (lastCategory !== product.category) {
        rows.push(<ProductCategoryRow category={product.category} />);
        lastCategory = product.category;
      }
      rows.push(<ProductRow product={product} />);
    }
  });
}
```

```

        ) === -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 = [
];

```

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

body {
  padding: 5px
}
label {
  display: block;
  margin-top: 5px;
  margin-bottom: 5px;
}
th {
  padding: 4px;
}
td {
  padding: 2px;
}
```

You can learn all about handling events and updating state in the [Adding Interactivity](#) section.

Where to go from here {/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.

'Tutorial: Tic-Tac-Toe'

You 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.

This 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? *{/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:

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

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;
}

```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}

.game {
  display: flex;
  flex-direction: row;
}

```

```
.game-info {
  margin-left: 20px;
}
```

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 *{/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.

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

```
* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}
```

```
.status {
  margin-bottom: 10px;
}

.game {
  display: flex;
  flex-direction: row;
}

.game-info {
  margin-left: 20px;
}
```

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

1. Install [Node.js](#)
2. In the CodeSandbox tab you opened earlier, press the top-left corner button to open the menu, and then choose **File > Export to ZIP** in that menu to download an archive of the files locally
3. Unzip the archive, then open a terminal and `cd` to the directory you unzipped
4. Install the dependencies with `npm install`
5. 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 [`{/overview/}`](#)

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

Inspecting the starter code [`{/inspecting-the-starter-code/}`](#)

In CodeSandbox you'll see three main sections:

CodeSandbox with starter code

CodeSandbox with starter code

1. The *Files* section with a list of files like `App.js`, `index.js`, `styles.css` and a folder called `public`
2. The *code editor* where you'll see the source code of your selected file
3. 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:

x-filled square

x-filled square

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

`App.js {/appjs/}`

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 {/stylescss/}`

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 {/indexjs/}`

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 brings 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 *{/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:

`/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:

two x-filled squares

two x-filled squares

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

nine x-filled squares in a line

nine x-filled squares in a line

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 `div`s you have your tic-tac-toe board:

 tic-tac-toe board filled with numbers 1 through 9

tic-tac-toe board filled with numbers 1 through 9

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:

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

```

        </div>
    </>
};

}

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}

.game-info {
  margin-left: 20px;
}

```

Psssst... 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 {/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 `div`s, your own components `Board` and `Square` must start with a capital letter.

Let's take a look:

 one-filled board

one-filled board

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:

value-filled board

value-filled board

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:

empty board

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:

tic-tac-toe board filled with numbers 1 through 9

tic-tac-toe board filled with numbers 1 through 9

Your updated code should look like this:

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

```
* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
}
```

```

    display: table;
}

.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}

.game-info {
  margin-left: 20px;
}

```

Making an interactive component {/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.

If 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:

 adding xes to board

adding xes to board

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:

```
import { 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>
    </>
  );
}
```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}

.game-info {
  margin-left: 20px;
}

```

React Developer Tools {/react-developer-tools/}

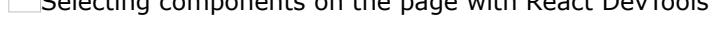
React DevTools let you check the props and the state of your React components. You can find the React DevTools tab at the bottom of the *browser* section in CodeSandbox:



React DevTools in CodeSandbox

React DevTools in CodeSandbox

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



Selecting components on the page with React DevTools

Selecting components on the page with React DevTools

For 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 {/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 {/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:

```
['0', null, 'X', 'X', 'X', '0', '0', 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>
</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:

 empty board

empty board

And your code should look like this:

```
import { 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>
    );
}

```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}
.game-info {
  margin-left: 20px;
}

```

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).

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 hard-coded 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:

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

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>
    </div>
  );
}
```

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

 filling the board with X

filling the board with X

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

This is what your code should look like:

```

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>
    </>
  );
}

```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
}

```

```

    width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}
.game-info {
  margin-left: 20px;
}

```

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:

1. 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`.
2. `handleClick` uses the argument (`0`) to update the first element of the `squares` array from `null` to `x`.
3. 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.

The 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 { /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 {/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:

O overwriting an X

O overwriting an X

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:

```

import { useState } from 'react';

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

```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

```

```

}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}
.game-info {
  margin-left: 20px;
}

```

Declaring a winner {/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;
}

```

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();
  //...
}

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:

```

import { useState } from 'react';

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

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;
}

```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
}

```

```

line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}
.game-info {
  margin-left: 20px;
}

```

Adding time travel {/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 {/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 {/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 `div`s 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 `null`s.

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:

```
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>
      <h1>Tic Tac Toe</h1>
      <p>{status}</p>
      <table border="1">
        <tbody>
          <tr>
            <td><button onClick={handleClick(0)}>{squares[0]}</button></td>
            <td><button onClick={handleClick(1)}>{squares[1]}</button></td>
            <td><button onClick={handleClick(2)}>{squares[2]}</button></td>
          </tr>
          <tr>
            <td><button onClick={handleClick(3)}>{squares[3]}</button></td>
            <td><button onClick={handleClick(4)}>{squares[4]}</button></td>
            <td><button onClick={handleClick(5)}>{squares[5]}</button></td>
          </tr>
          <tr>
            <td><button onClick={handleClick(6)}>{squares[6]}</button></td>
            <td><button onClick={handleClick(7)}>{squares[7]}</button></td>
            <td><button onClick={handleClick(8)}>{squares[8]}</button></td>
          </tr>
        </tbody>
      </table>
      <div>
        <button onClick={onNewGame}>New Game</button>
      </div>
    </div>
  );
}

const 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];
    const aVal = squares[a];
    const bVal = squares[b];
    const cVal = squares[c];
    if (aVal === bVal && bVal === cVal && aVal !== '') {
      return aVal;
    }
  }
  return null;
};

const App = () => {
  const [xIsNext, setXIsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));
  const [history, setHistory] = useState([[]]);
  const [status, setStatus] = useState('');

  const onNewGame = () => {
    setSquares(Array(9).fill(null));
    setHistory([[]]);
    setXIsNext(true);
    setStatus('');
  };

  const onSquareClick = (i) => {
    const nextSquares = squares.slice();
    nextSquares[i] = xIsNext ? 'X' : 'O';
    setSquares(nextSquares);
    setXIsNext(!xIsNext);
    setHistory([...history, nextSquares]);
    const winner = calculateWinner(nextSquares);
    if (winner) {
      setStatus(`Winner: ${winner}`);
    } else {
      setStatus('Next player: ' + (xIsNext ? 'X' : 'O'));
    }
  };

  return (
    <div>
      <h1>Tic Tac Toe</h1>
      <p>{status}</p>
      <table border="1">
        <tbody>
          <tr>
            <td><button onClick={onSquareClick(0)}>{squares[0]}</button></td>
            <td><button onClick={onSquareClick(1)}>{squares[1]}</button></td>
            <td><button onClick={onSquareClick(2)}>{squares[2]}</button></td>
          </tr>
          <tr>
            <td><button onClick={onSquareClick(3)}>{squares[3]}</button></td>
            <td><button onClick={onSquareClick(4)}>{squares[4]}</button></td>
            <td><button onClick={onSquareClick(5)}>{squares[5]}</button></td>
          </tr>
          <tr>
            <td><button onClick={onSquareClick(6)}>{squares[6]}</button></td>
            <td><button onClick={onSquareClick(7)}>{squares[7]}</button></td>
            <td><button onClick={onSquareClick(8)}>{squares[8]}</button></td>
          </tr>
        </tbody>
      </table>
      <div>
        <button onClick={onNewGame}>New Game</button>
      </div>
    </div>
  );
};

export default App;
```

```

        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);
  }

  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;
}

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}
.game-info {
  margin-left: 20px;
}

```

Showing the past moves {/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: `Warning: 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.

```
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;
}

```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
}

```

```

margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}

.game {
  display: flex;
  flex-direction: row;
}

.game-info {
  margin-left: 20px;
}

```

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 {/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

```

<li>Alexa: 7 tasks left</li>
<li>Ben: 5 tasks left</li>

```

to

```

<li>Ben: 9 tasks left</li>
<li>Claudia: 8 tasks left</li>
<li>Alexa: 5 tasks left</li>

```

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 can't 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
</li>
```

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 {/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>
    );
}

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]} 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>
        </>
    );
}

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;
}

```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
}

```

react

```
padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

.status {
  margin-bottom: 10px;
}

.game {
  display: flex;
  flex-direction: row;
}

.game-info {
  margin-left: 20px;
}
```

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.

```
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;
}

```

```

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
  margin: 20px;
  padding: 0;
}

.square {
  background: #fff;
  border: 1px solid #999;
  float: left;
  font-size: 24px;
  font-weight: bold;
  line-height: 34px;
  height: 34px;
  margin-right: -1px;
  margin-top: -1px;
  padding: 0;
  text-align: center;
  width: 34px;
}

.board-row:after {
  clear: both;
  content: '';
  display: table;
}

```

```
.status {
  margin-bottom: 10px;
}
.game {
  display: flex;
  flex-direction: row;
}
.game-info {
  margin-left: 20px;
}
```

Final cleanup {/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 {/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:

```

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]} 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>
    </>
  );
}

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;
}

* {
  box-sizing: border-box;
}

body {
  font-family: sans-serif;
}

```

```

margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}
.game-info {
margin-left: 20px;
}

```

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:

1. For the current move only, show "You are at move #..." instead of a button.
2. Rewrite `Board` to use two loops to make the squares instead of hardcoding them.
3. Add a toggle button that lets you sort the moves in either ascending or descending order.
4. 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).
5. 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 build an app's UI.

Updating Arrays in State

Arrays 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.

- 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 {/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)
adding	<code>push</code> , <code>unshift</code>	<code>concat</code> , <code>[...arr]</code> spread syntax (example)
removing	<code>pop</code> , <code>shift</code> , <code>splice</code>	<code>filter</code> , <code>slice</code> (example)
replacing	<code>splice</code> , <code>arr[i] = ...</code> assignment	<code>map</code> (example)
sorting	<code>reverse</code> , <code>sort</code>	copy the array first (example)

Alternatively, you can [use Immer](#) which lets you use methods from both columns.

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 {/adding-to-an-array/}

`push()` will mutate an array, which you don't want:

```

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

```

```
button { margin-left: 5px; }
```

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
  ]
);

```

Now it works correctly:

```

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={() => {
        setArtists([
          ...artists,

```

```

        ]);
    }]>Add</button>
<ul>
  {artists.map(artist => (
    <li key={artist.id}>{artist.name}</li>
  )));
</ul>
</>
);
}

```

`button { margin-left: 5px; }`

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

```

setArtists([
  ...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 {/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:

```

import { useState } from 'react';

let initialArtists = [
];

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>
    </>
  );
}

```

```

        </>
    );
}

```

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 {/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()`:

```

import { useState } from 'react';

let initialShapes = [
];

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

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

  return (
    <>
      <button onClick={handleClick}>
        Move circles down!
      </button>
      {shapes.map(shape => (
        <div

```

```

        key={shape.id}
        style={{
          background: 'purple',
          position: 'absolute',
          left: shape.x,
          top: shape.y,
          borderRadius:
            shape.type === 'circle'
              ? '50%' : '',
          width: 20,
          height: 20,
        }}/>
      ))}
    );
  }
}

body { height: 300px; }

```

Replacing items in an array {/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:

```

import { 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);
          }}>
            ...
          </button>
        </li>
      ))}
    </ul>
  );
}

```

```

        }]}>+1</button>
      </li>
    )}
</ul>
);
}

```

```
button { margin: 5px; }
```

Inserting into an array {/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`:

```

import { useState } from 'react';

let nextId = 3;
const initialArtists = [];

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:
      // 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>
    </>
  );
}

```

```

        </>
    );
}

```

```
button { margin-left: 5px; }
```

Making other changes to an array {/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:

```

import { useState } from 'react';

let nextId = 3;
const initialList = [
];

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>
    </>
  );
}

```

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

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

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

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

Updating objects inside arrays {/updating-objects-in-side-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:

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

let nextId = 3;
const initialList = [
];

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>
  );
}

```

The problem is in code like this:

```

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

```

import { useState } from 'react';

let nextId = 3;
const initialList = [
];

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>
);
}

```

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 { /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:

```

import { useState } from 'react';
import { useImmer } from 'use-immer';

let nextId = 3;
const initialList = [];

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

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

```

```

        artwork.seen = nextSeen;
    });

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

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

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

```

```
{
  "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"
}
}

```

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.

- 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.

Update an item in the shopping cart [{/update-an-item-in-the-shopping-cart/}](#)

Fill in the `handleIncreaseClick` logic so that pressing "+" increases the corresponding number:

```

import { useState } from 'react';

const initialProducts = [
  {
    name: 'Baklava',
    count: 1,
  },
  {
    name: 'Cheese',
    count: 5,
  },
  {
    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>
);
}

```

```
button { margin: 5px; }
```

You can use the `map` function to create a new array, and then use the `...` object spread syntax to create a copy of the changed object for the new array:

```

import { useState } from 'react';

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

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

  function handleIncreaseClick(productId) {
    setProducts(products.map(product => {
      if (product.id === productId) {
        return {
          ...product,
          count: product.count + 1
        };
      } else {
        return product;
      }
    }));
  }

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

```

```

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

```

```
button { margin: 5px; }
```

Remove an item from the shopping cart [{/remove-an-item-from-the-shopping-cart/}](#)

This shopping cart has a working "+" button, but the "-" button doesn't do anything. You need to add an event handler to it so that pressing it decreases the `count` of the corresponding product. If you press "-" when the count is 1, the product should automatically get removed from the cart. Make sure it never shows 0.

```

import { useState } from 'react';

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

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

  function handleIncreaseClick(productId) {
    setProducts(products.map(product => {
      if (product.id === productId) {
        return {
          ...product,
          count: product.count + 1
        };
      } else {
        return product;
      }
    }));
  }

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

```

```

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

button { margin: 5px; }

```

You can first use `map` to produce a new array, and then `filter` to remove products with a `count` set to 0 :

```

import { useState } from 'react';

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

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

  function handleIncreaseClick(productId) {
    setProducts(products.map(product => {
      if (product.id === productId) {
        return {
          ...product,
          count: product.count + 1
        };
      } else {
        return product;
      }
    }));
  }

  function handleDecreaseClick(productId) {
    let nextProducts = products.map(product => {
      if (product.id === productId) {
        return {
          ...product,
          count: product.count - 1
        };
      } else {
        return product;
      }
    });
    nextProducts = nextProducts.filter(p =>

```

```

    p.count > 0
  );
  setProducts(nextProducts)
}

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

```

```
button { margin: 5px; }
```

Fix the mutations using non-mutative methods *(/fix-the-mutations-using-non-mutative-methods/)*

In this example, all of the event handlers in `App.js` use mutation. As a result, editing and deleting todos doesn't work. Rewrite `handleAddTodo`, `handleChangeTodo`, and `handleDeleteTodo` to use the non-mutative methods:

```

import { useState } from 'react';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;
const initialTodos = [];

export default function TaskApp() {
  const [todos, setTodos] = useState(
    initialTodos
  );

  function handleAddTodo(title) {
    todos.push({
      # title,
      done: false
    });
  }

  function handleChangeTodo(nextTodo) {
    const todo = todos.find(t =>

```

```

    t.id === nextTodo.id
  );
  todo.title = nextTodo.title;
  todo.done = nextTodo.done;
}

function handleDeleteTodo(todoId) {
  const index = todos.findIndex(t =>
    t.id === todoId
  );
  todos.splice(index, 1);
}

return (
  <>
    <AddTodo
      onAddTodo={handleAddTodo}
    />
    <TaskList
      todos={todos}
      onChangeTodo={handleChangeTodo}
      onDeleteTodo={handleDeleteTodo}
    />
  </>
);
}

```

```

import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
  const [title, setTitle] = useState('');
  return (
    <>
      <input
        placeholder="Add todo"
        value={title}
        onChange={e => setTitle(e.target.value)}
      />
      <button onClick={() => {
        setTitle('');
        onAddTodo(title);
      }}>Add</button>
    </>
  )
}

```

```

import { useState } from 'react';

export default function TaskList({
  todos,
  onChangeTodo,
  onDeleteTodo
}) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <Task
            todo={todo}
            onChange={onChangeTodo}
            onDelete={onDeleteTodo}
          />

```

```

        </li>
      )}
    </ul>
  );
}

function Task({ todo, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let todoContent;
  if (isEditing) {
    todoContent = (
      <>
        <input
          value={todo.title}
          onChange={e => {
            onChange({
              ...todo,
              # e.target.value
            });
          }}/>
        <button onClick={() => setIsEditing(false)}>
          Save
        </button>
      </>
    );
  } else {
    todoContent = (
      <>
        {todo.title}
        <button onClick={() => setIsEditing(true)}>
          Edit
        </button>
      </>
    );
  }
  return (
    <label>
      <input
        type="checkbox"
        checked={todo.done}
        onChange={e => {
          onChange({
            ...todo,
            done: e.target.checked
          });
        }}/>
      {todoContent}
      <button onClick={() => onDelete(todo.id)}>
        Delete
      </button>
    </label>
  );
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

In `handleAddTodo`, you can use the array spread syntax. In `handleChangeTodo`, you can create a new array with `map`. In `handleDeleteTodo`, you can create a new array with `filter`. Now the list works correctly:

```
import { useState } from 'react';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;
const initialTodos = [];

export default function TaskApp() {
  const [todos, setTodos] = useState(
    initialTodos
  );

  function handleAddTodo(title) {
    setTodos([
      ...todos,
      {
        # title,
        done: false
      }
    ]);
  }

  function handleChangeTodo(nextTodo) {
    setTodos(todos.map(t => {
      if (t.id === nextTodo.id) {
        return nextTodo;
      } else {
        return t;
      }
    }));
  }

  function handleDeleteTodo(todoId) {
    setTodos(
      todos.filter(t => t.id !== todoId)
    );
  }

  return (
    <>
      <AddTodo
        onAddTodo={handleAddTodo}
      />
      <TaskList
        todos={todos}
        onChangeTodo={handleChangeTodo}
        onDeleteTodo={handleDeleteTodo}
      />
    </>
  );
}
```

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

export default function AddTodo({ onAddTodo }) {
```

```

const [title, setTitle] = useState('');
return (
  <>
    <input
      placeholder="Add todo"
      value={title}
      onChange={e => setTitle(e.target.value)}
    />
    <button onClick={() => {
      setTitle('');
      onAddTodo(title);
    }}>Add</button>
  </>
)
}

```

```

import { useState } from 'react';

export default function TaskList({
  todos,
  onChangeTodo,
  onDeleteTodo
}) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <Task
            todo={todo}
            onChange={onChangeTodo}
            onDelete={onDeleteTodo}
          />
        </li>
      ))}
    </ul>
  );
}

function Task({ todo, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let todoContent;
  if (isEditing) {
    todoContent = (
      <>
        <input
          value={todo.title}
          onChange={e => {
            onChange({
              ...todo,
              # e.target.value
            });
          }} />
        <button onClick={() => setIsEditing(false)}>
          Save
        </button>
      </>
    );
  } else {
    todoContent = (
      <>
        {todo.title}
        <button onClick={() => setIsEditing(true)}>
    
```

```

        Edit
      </button>
    </>
  );
}
return (
  <label>
    <input
      type="checkbox"
      checked={todo.done}
      onChange={e => {
        onChange({
          ...todo,
          done: e.target.checked
        });
      }}
    />
    {todoContent}
    <button onClick={() => onDelete(todo.id)}>
      Delete
    </button>
  </label>
);
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

Fix the mutations using Immer {/fix-the-mutations-using-immer/}

This is the same example as in the previous challenge. This time, fix the mutations by using Immer. For your convenience, `useImmer` is already imported, so you need to change the `todos` state variable to use it.

```

import { useState } from 'react';
import { useImmer } from 'use-immer';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;
const initialTodos = [
];

export default function TaskApp() {
  const [todos, setTodos] = useState(
    initialTodos
  );

  function handleAddTodo(title) {
    todos.push({
      # title,
      done: false
    });
  }

  function handleChangeTodo(nextTodo) {
    const todo = todos.find(t =>
      t.id === nextTodo.id
    );
  }
}

```

```

        todo.title = nextTodo.title;
        todo.done = nextTodo.done;
    }

    function handleDeleteTodo(todoId) {
        const index = todos.findIndex(t =>
            t.id === todoId
        );
        todos.splice(index, 1);
    }

    return (
        <>
            <AddTodo
                onAddTodo={handleAddTodo}
            />
            <TaskList
                todos={todos}
                onChangeTodo={handleChangeTodo}
                onDeleteTodo={handleDeleteTodo}
            />
        </>
    );
}

```

```

import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
    const [title, setTitle] = useState('');
    return (
        <>
            <input
                placeholder="Add todo"
                value={title}
                onChange={e => setTitle(e.target.value)}
            />
            <button onClick={() => {
                setTitle('');
                onAddTodo(title);
            }}>Add</button>
        </>
    )
}

```

```

import { useState } from 'react';

export default function TaskList({
    todos,
    onChangeTodo,
    onDeleteTodo
}) {
    return (
        <ul>
            {todos.map(todo => (
                <li key={todo.id}>
                    <Task
                        todo={todo}
                        onChange={onChangeTodo}
                        onDelete={onDeleteTodo}
                    />
                </li>
            ))}
        </ul>
    )
}

```

```

        </ul>
    );
}

function Task({ todo, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let todoContent;
  if (isEditing) {
    todoContent = (
      <>
        <input
          value={todo.title}
          onChange={e => {
            onChange({
              ...todo,
              # e.target.value
            });
          }} />
        <button onClick={() => setIsEditing(false)}>
          Save
        </button>
      </>
    );
  } else {
    todoContent = (
      <>
        {todo.title}
        <button onClick={() => setIsEditing(true)}>
          Edit
        </button>
      </>
    );
  }
  return (
    <label>
      <input
        type="checkbox"
        checked={todo.done}
        onChange={e => {
          onChange({
            ...todo,
            done: e.target.checked
          });
        }}
      />
      {todoContent}
      <button onClick={() => onDelete(todo.id)}>
        Delete
      </button>
    </label>
  );
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

```
{
  "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"
  }
}

```

With Immer, you can write code in the mutative fashion, as long as you're only mutating parts of the `draft` that Immer gives you. Here, all mutations are performed on the `draft` so the code works:

```

import { useState } from 'react';
import { useImmer } from 'use-immer';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;
const initialTodos = [];

export default function TaskApp() {
  const [todos, updateTodos] = useImmer(
    initialTodos
  );

  function handleAddTodo(title) {
    updateTodos(draft => {
      draft.push({
        # title,
        done: false
      });
    });
  }

  function handleChangeTodo(nextTodo) {
    updateTodos(draft => {
      const todo = draft.find(t =>
        t.id === nextTodo.id
      );
      todo.title = nextTodo.title;
      todo.done = nextTodo.done;
    });
  }

  function handleDeleteTodo(todoId) {
    updateTodos(draft => {
      const index = draft.findIndex(t =>
        t.id === todoId
      );
      draft.splice(index, 1);
    });
  }

  return (
    <>
      <AddTodo
        onAddTodo={handleAddTodo}
      />

```

```

    <TaskList
      todos={todos}
      onChangeTodo={handleChangeTodo}
      onDeleteTodo={handleDeleteTodo}
    />
  );
}

```

```

import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
  const [title, setTitle] = useState('');
  return (
    <>
      <input
        placeholder="Add todo"
        value={title}
        onChange={e => setTitle(e.target.value)}
      />
      <button onClick={() => {
        setTitle('');
        onAddTodo(title);
      }}>Add</button>
    </>
  )
}

```

```

import { useState } from 'react';

export default function TaskList({
  todos,
  onChangeTodo,
  onDeleteTodo
}) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <Task
            todo={todo}
            onChange={onChangeTodo}
            onDelete={onDeleteTodo}
          />
        </li>
      ))}
    </ul>
  );
}

function Task({ todo, onChange, onDelete }) {
  const [isEditing, setIsEditing] = useState(false);
  let todoContent;
  if (isEditing) {
    todoContent = (
      <>
        <input
          value={todo.title}
          onChange={e => {
            onChange({
              ...todo,
              # e.target.value
            });
            setIsEditing(false);
          }}
        />
      </>
    );
  }
  return (
    <li key={todo.id}>
      {isEditing ? todoContent : todo.title}
      <button onClick={onDelete}>Delete</button>
    </li>
  );
}

```

```

        });
    >]} />
    <button onClick={() => setIsEditing(false)}>
      Save
    </button>
  </>
);
} else {
  todoContent = (
    <>
      {todo.title}
      <button onClick={() => setIsEditing(true)}>
        Edit
      </button>
    </>
  );
}
return (
  <label>
    <input
      type="checkbox"
      checked={todo.done}
      onChange={e => {
        onChange({
          ...todo,
          done: e.target.checked
        });
      }}
    />
    {todoContent}
    <button onClick={() => onDelete(todo.id)}>
      Delete
    </button>
  </label>
);
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

```

{
  "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"
  }
}

```

You can also mix and match the mutative and non-mutative approaches with Immer.

For example, in this version `handleAddTodo` is implemented by mutating the Immer `draft`, while `handleChangeTodo` and `handleDeleteTodo` use the non-mutative `map` and `filter` methods:

```
import { useState } from 'react';
import { useImmer } from 'use-immer';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;
const initialTodos = [];

export default function TaskApp() {
  const [todos, updateTodos] = useImmer(
    initialTodos
  );

  function handleAddTodo(title) {
    updateTodos(draft => {
      draft.push({
        # title,
        done: false
      });
    });
  }

  function handleChangeTodo(nextTodo) {
    updateTodos(todos.map(todo => {
      if (todo.id === nextTodo.id) {
        return nextTodo;
      } else {
        return todo;
      }
    }));
  }

  function handleDeleteTodo(todoId) {
    updateTodos(
      todos.filter(t => t.id !== todoId)
    );
  }

  return (
    <>
      <AddTodo
        onAddTodo={handleAddTodo}
      />
      <TaskList
        todos={todos}
        onChangeTodo={handleChangeTodo}
        onDeleteTodo={handleDeleteTodo}
      />
    </>
  );
}

import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
  const [title, setTitle] = useState('');
  return (

```

```

        <>
        <input
            placeholder="Add todo"
            value={title}
            onChange={e => setTitle(e.target.value)}
        />
        <button onClick={() => {
            setTitle('');
            onAddTodo(title);
        }}>Add</button>
    </>
)
}

```

```

import { useState } from 'react';

export default function TaskList({
    todos,
    onChangeTodo,
    onDeleteTodo
}) {
    return (
        <ul>
            {todos.map(todo => (
                <li key={todo.id}>
                    <Task
                        todo={todo}
                        onChange={onChangeTodo}
                        onDelete={onDeleteTodo}
                    />
                </li>
            ))}
        </ul>
    );
}

function Task({ todo, onChange, onDelete }) {
    const [isEditing, setIsEditing] = useState(false);
    let todoContent;
    if (isEditing) {
        todoContent = (
            <>
                <input
                    value={todo.title}
                    onChange={e => {
                        onChange({
                            ...todo,
                            # e.target.value
                        });
                    }}
                />
                <button onClick={() => setIsEditing(false)}>
                    Save
                </button>
            </>
        );
    } else {
        todoContent = (
            <>
                {todo.title}
                <button onClick={() => setIsEditing(true)}>
                    Edit
                </button>
            </>
        );
    }
    return (
        <li key={todo.id}>
            <Task
                todo={todo}
                onChange={onChange}
                onDelete={onDelete}
            />
        </li>
    );
}

```

```

        </>
    );
}

return (
  <label>
    <input
      type="checkbox"
      checked={todo.done}
      onChange={e => {
        onChange({
          ...todo,
          done: e.target.checked
        });
      }}
    />
    {todoContent}
    <button onClick={() => onDelete(todo.id)}>
      Delete
    </button>
  </label>
);
}

```

```

button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }

```

```

{
  "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"
  }
}

```

With Immer, you can pick the style that feels the most natural for each separate case.

Updating Objects in State

State 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.

- 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? {/whats-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 {/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:

```
import { 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>
  );
}

```

```
body { margin: 0; padding: 0; height: 250px; }
```

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:

```
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>
  );
}
```

```
body { margin: 0; padding: 0; height: 250px; }
```

Local mutation is fine *{/local-mutation-is-fine/}*

Code 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 { /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:

```
import { 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>
    </>
  );
}
```

```

        </label>
      <p>
        {person.firstName}{` `}
        {person.lastName}{` `}
        ({person.email})
      </p>
    </>
  );
}

label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }

```

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!

```

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>
    </>
);
}

```

```

label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }

```

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.

Using a single event handler for multiple fields {/using-a-single-event-handler-for-multiple-fields/}

You can also use the `[]` 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:

```

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

```

```

label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }

```

Here, `e.target.name` refers to the `name` property given to the `<input>` DOM element.

Updating a nested object {/updating-a-nested-object/}

Consider a nested object structure like this:

```
const [person, setPerson] = useState({
  name: 'Niki de Saint Phalle',
  artwork: {
    # 'Blue Nana',
    city: 'Hamburg',
    image: 'https://i.imgur.com/Sd1AgU0m.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:

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

export default function Form() {
  const [person, setPerson] = useState({
    name: 'Niki de Saint Phalle',
    artwork: {
      # 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgU0m.jpg',
    }
  });

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

  function handleTitleChange(e) {
    setPerson({
      ...person,
      artwork: {
        ...person.artwork,
```

```
# 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}
    />
  );
}

```

```

label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
img { width: 200px; height: 200px; }

```

Objects are not really nested {/objects-are-not-really-nested/}

An object like this appears "nested" in code:

```

let obj = {
  name: 'Niki de Saint Phalle',
  artwork: {
    # 'Blue Nana',
    city: 'Hamburg',
    image: 'https://i.imgur.com/Sd1AgU0m.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 = {
  # 'Blue Nana',
  city: 'Hamburg',
  image: 'https://i.imgur.com/Sd1AgU0m.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 = {
  # 'Blue Nana',
  city: 'Hamburg',
  image: 'https://i.imgur.com/Sd1AgU0m.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 {/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!

How does Immer work? {/how-does-immer-work/}

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:

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

Here is the above example converted to Immer:

```
import { useImmer } from 'use-immer';

export default function Form() {
  const [person, updatePerson] = useImmer({
    name: 'Niki de Saint Phalle',
    artwork: {
      # 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgU0m.jpg',
    }
  });

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

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

```

}

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

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

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

```

```
{
  "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"
  }
}

label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
img { width: 200px; height: 200px; }

```

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.

Why is mutating state not recommended in React? [{/why-is-mutating-state-not-recommended-in-react/}](#)

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!

- 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.

Fix incorrect state updates `{/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.

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

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

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

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

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

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

```

        onChange={handleLastNameChange}
      />
    </label>
  </>
);
}

label { display: block; margin-bottom: 10px; }
input { margin-left: 5px; margin-bottom: 5px; }

```

Here is a version with both bugs fixed:

```

import { useState } from 'react';

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

  function handlePlusClick() {
    setPlayer({
      ...player,
      score: player.score + 1,
    });
  }

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

  function handleLastNameChange(e) {
    setPlayer({
      ...player,
      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>
  </>
);
}

label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }

```

The problem with `handlePlusClick` was that it mutated the `player` object. As a result, React did not know that there's a reason to re-render, and did not update the score on the screen. This is why, when you edited the first name, the state got updated, triggering a re-render which *also* updated the score on the screen.

The problem with `handleLastNameChange` was that it did not copy the existing `...player` fields into the new object. This is why the score got lost after you edited the last name.

Find and fix the mutation [{/find-and-fix-the-mutation/}](#)

There is a draggable box on a static background. You can change the box's color using the select input.

But there is a bug. If you move the box first, and then change its color, the background (which isn't supposed to move!) will "jump" to the box position. But this should not happen: the `Background`'s `position` prop is set to `initialPosition`, which is `{ x: 0, y: 0 }`. Why is the background moving after the color change?

Find the bug and fix it.

If something unexpected changes, there is a mutation. Find the mutation in `App.js` and fix it.

```

import { useState } from 'react';
import Background from './Background.js';
import Box from './Box.js';

const initialPosition = {
  x: 0,
  y: 0
};

export default function Canvas() {
  const [shape, setShape] = useState({
    color: 'orange',
    position: initialPosition
});

  function handleMove(dx, dy) {
    shape.position.x += dx;
    shape.position.y += dy;
  }

  function handleColorChange(e) {
    setShape({
      ...shape,
      color: e.target.value
    });
  }
}

```

```

        });
    }

    return (
      <>
      <select
        value={shape.color}
        onChange={handleColorChange}>
        </>
        <option value="orange">orange</option>
        <option value="lightpink">lightpink</option>
        <option value="aliceblue">aliceblue</option>
      </select>
      <Background
        position={initialPosition}>
      </>
      <Box
        color={shape.color}
        position={shape.position}
        onMouseMove={handleMove}>
        >
        Drag me!
      </Box>
    </>
  );
}

```

```

import { useState } from 'react';

export default function Box({
  children,
  color,
  position,
  onMouseMove
}) {
  const [
    lastCoordinates,
    setLastCoordinates
  ] = useState(null);

  function handlePointerDown(e) {
    e.target.setPointerCapture(e.pointerId);
    setLastCoordinates({
      x: e.clientX,
      y: e.clientY,
    });
  }

  function handlePointerMove(e) {
    if (lastCoordinates) {
      setLastCoordinates({
        x: e.clientX,
        y: e.clientY,
      });
      const dx = e.clientX - lastCoordinates.x;
      const dy = e.clientY - lastCoordinates.y;
      onMouseMove(dx, dy);
    }
  }

  function handlePointerUp(e) {
    setLastCoordinates(null);
  }
}

```

```

        }

      return (
        <div
          onPointerDown={handlePointerDown}
          onPointerMove={handlePointerMove}
          onPointerUp={handlePointerUp}
          style={{
            width: 100,
            height: 100,
            cursor: 'grab',
            backgroundColor: color,
            position: 'absolute',
            border: '1px solid black',
            display: 'flex',
            justifyContent: 'center',
            alignItems: 'center',
            transform: `translate(
              ${position.x}px,
              ${position.y}px
            )`,
          }}
        >{children}</div>
      );
    }
  
```

```

export default function Background({
  position
}) {
  return (
    <div style={{
      position: 'absolute',
      transform: `translate(
        ${position.x}px,
        ${position.y}px
      )`,
      width: 250,
      height: 250,
      backgroundColor: 'rgba(200, 200, 0, 0.2)',
    }} />
  );
}

body { height: 280px; }
select { margin-bottom: 10px; }

```

The problem was in the mutation inside `handleMove`. It mutated `shape.position`, but that's the same object that `initialPosition` points at. This is why both the shape and the background move. (It's a mutation, so the change doesn't reflect on the screen until an unrelated update--the color change--triggers a re-render.)

The fix is to remove the mutation from `handleMove`, and use the spread syntax to copy the shape. Note that `+=` is a mutation, so you need to rewrite it to use a regular `+` operation.

```

import { useState } from 'react';
import Background from './Background.js';
import Box from './Box.js';

const initialPosition = {

```

```

x: 0,
y: 0
};

export default function Canvas() {
  const [shape, setShape] = useState({
    color: 'orange',
    position: initialPosition
  });

  function handleMove(dx, dy) {
    setShape({
      ...shape,
      position: {
        x: shape.position.x + dx,
        y: shape.position.y + dy,
      }
    });
  }

  function handleColorChange(e) {
    setShape({
      ...shape,
      color: e.target.value
    });
  }

  return (
    <>
      <select
        value={shape.color}
        onChange={handleColorChange}>
        <option value="orange">orange</option>
        <option value="lightpink">lightpink</option>
        <option value="aliceblue">aliceblue</option>
      </select>
      <Background
        position={initialPosition}>
    </>
      <Box
        color={shape.color}
        position={shape.position}
        onMouseMove={handleMove}>
        > Drag me!
      </Box>
    </>
  );
}

```

```

import { useState } from 'react';

export default function Box({
  children,
  color,
  position,
  onMouseMove
}) {
  const [
    lastCoordinates,
    setLastCoordinates
  ] = useState({
    x: 0,
    y: 0
  });

  function handleMove(dx, dy) {
    setLastCoordinates({
      ...lastCoordinates,
      x: lastCoordinates.x + dx,
      y: lastCoordinates.y + dy
    });
  }

  return (
    <>
      <div
        style={{ background: color, width: 100, height: 100, position: 'absolute', left: position.x, top: position.y }}>
        <div
          style={{ position: 'absolute', left: -5, top: -5, width: 10, height: 10, background: 'black', border: '1px solid black' }}>
            <span>X</span>
          </div>
      </div>
      <div
        style={{ position: 'absolute', left: position.x, top: position.y, width: 100, height: 100, background: color }}>
        <div
          style={{ position: 'absolute', left: -5, top: -5, width: 10, height: 10, background: 'black', border: '1px solid black' }}>
            <span>X</span>
          </div>
        <div
          style={{ position: 'absolute', left: 50%, top: 50%, width: 20, height: 20, border: '1px solid black' }}>
            <span>+</span>
          </div>
      </div>
    </>
  );
}

```

```

] = useState(null);

function handlePointerDown(e) {
  e.target.setPointerCapture(e.pointerId);
  setLastCoordinates({
    x: e.clientX,
    y: e.clientY,
  });
}

function handlePointerMove(e) {
  if (lastCoordinates) {
    setLastCoordinates({
      x: e.clientX,
      y: e.clientY,
    });
    const dx = e.clientX - lastCoordinates.x;
    const dy = e.clientY - lastCoordinates.y;
    onMove(dx, dy);
  }
}

function handlePointerUp(e) {
  setLastCoordinates(null);
}

return (
  <div
    onPointerDown={handlePointerDown}
    onPointerMove={handlePointerMove}
    onPointerUp={handlePointerUp}
    style={{
      width: 100,
      height: 100,
      cursor: 'grab',
      backgroundColor: color,
      position: 'absolute',
      border: '1px solid black',
      display: 'flex',
      justifyContent: 'center',
      alignItems: 'center',
      transform: `translate(
        ${position.x}px,
        ${position.y}px
      )`,
    }}
    >{children}</div>
);
}

```

```

export default function Background({
  position
}) {
  return (
    <div style={{
      position: 'absolute',
      transform: `translate(
        ${position.x}px,
        ${position.y}px
      )`,
      width: 250,
      height: 250,
    }}>

```

```

        backgroundColor: 'rgba(200, 200, 0, 0.2)',
    }]} />
);
};

body { height: 280px; }
select { margin-bottom: 10px; }

```

Update an object with Immer {/update-an-object-with-immer/}

This is the same buggy example as in the previous challenge. This time, fix the mutation by using Immer. For your convenience, `useImmer` is already imported, so you need to change the `shape` state variable to use it.

```

import { useState } from 'react';
import { useImmer } from 'use-immer';
import Background from './Background.js';
import Box from './Box.js';

const initialPosition = {
  x: 0,
  y: 0
};

export default function Canvas() {
  const [shape, setShape] = useState({
    color: 'orange',
    position: initialPosition
  });

  function handleMove(dx, dy) {
    shape.position.x += dx;
    shape.position.y += dy;
  }

  function handleColorChange(e) {
    setShape({
      ...shape,
      color: e.target.value
    });
  }

  return (
    <>
      <select
        value={shape.color}
        onChange={handleColorChange}>
        <option value="orange">orange</option>
        <option value="lightpink">lightpink</option>
        <option value="aliceblue">aliceblue</option>
      </select>
      <Background
        position={initialPosition}>
      </Background>
      <Box
        color={shape.color}
        position={shape.position}
        onMouseMove={handleMove}>
    </Box>
  );
}

```

```

    Drag me!
  </Box>
</>
);
}

```

```

import { useState } from 'react';

export default function Box({
  children,
  color,
  position,
  onMove
}) {
  const [
    lastCoordinates,
    setLastCoordinates
  ] = useState(null);

  function handlePointerDown(e) {
    e.target.setPointerCapture(e.pointerId);
    setLastCoordinates({
      x: e.clientX,
      y: e.clientY,
    });
  }

  function handlePointerMove(e) {
    if (lastCoordinates) {
      setLastCoordinates({
        x: e.clientX,
        y: e.clientY,
      });
      const dx = e.clientX - lastCoordinates.x;
      const dy = e.clientY - lastCoordinates.y;
      onMove(dx, dy);
    }
  }

  function handlePointerUp(e) {
    setLastCoordinates(null);
  }

  return (
    <div
      onPointerDown={handlePointerDown}
      onPointerMove={handlePointerMove}
      onPointerUp={handlePointerUp}
      style={{
        width: 100,
        height: 100,
        cursor: 'grab',
        backgroundColor: color,
        position: 'absolute',
        border: '1px solid black',
        display: 'flex',
        justifyContent: 'center',
        alignItems: 'center',
        transform: `translate(
          ${position.x}px,
          ${position.y}px
        )`,
      }}
    >{children}</div>
  );
}

```

```

        }]);
      >{children}</div>
    );
}

export default function Background({
  position
}) {
  return (
    <div style={{
      position: 'absolute',
      transform: `translate(
        ${position.x}px,
        ${position.y}px
      )`,
      width: 250,
      height: 250,
      backgroundColor: 'rgba(200, 200, 0, 0.2)'
    }} />
  );
}

```

```

body { height: 280px; }
select { margin-bottom: 10px; }

```

```

{
  "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"
  }
}

```

This is the solution rewritten with Immer. Notice how the event handlers are written in a mutating fashion, but the bug does not occur. This is because under the hood, Immer never mutates the existing objects.

```

import { useImmer } from 'use-immer';
import Background from './Background.js';
import Box from './Box.js';

const initialPosition = {
  x: 0,
  y: 0
};

export default function Canvas() {
  const [shape, updateShape] = useImmer({
    color: 'orange',
    position: initialPosition
  });

  function handleMove(dx, dy) {

```

```

    updateShape(draft => {
      draft.position.x += dx;
      draft.position.y += dy;
    });
}

function handleColorChange(e) {
  updateShape(draft => {
    draft.color = e.target.value;
  });
}

return (
  <>
  <select
    value={shape.color}
    onChange={handleColorChange}>
    <option value="orange">orange</option>
    <option value="lightpink">lightpink</option>
    <option value="aliceblue">aliceblue</option>
  </select>
  <Background
    position={initialPosition}>
  </>
  <Box
    color={shape.color}
    position={shape.position}
    onMouseMove={handleMove}>
    > Drag me!
  </Box>
  </>
);
}

```

```

import { useState } from 'react';

export default function Box({
  children,
  color,
  position,
  onMouseMove
}) {
  const [
    lastCoordinates,
    setLastCoordinates
  ] = useState(null);

  function handlePointerDown(e) {
    e.target.setPointerCapture(e.pointerId);
    setLastCoordinates({
      x: e.clientX,
      y: e.clientY,
    });
  }

  function handlePointerMove(e) {
    if (lastCoordinates) {
      setLastCoordinates({
        x: e.clientX,
        y: e.clientY,
      });
    }
  }
}

```

react

```
    });
    const dx = e.clientX - lastCoordinates.x;
    const dy = e.clientY - lastCoordinates.y;
    onMove(dx, dy);
}
}

function handlePointerUp(e) {
  setLastCoordinates(null);
}

return (
  <div
    onPointerDown={handlePointerDown}
    onPointerMove={handlePointerMove}
    onPointerUp={handlePointerUp}
    style={{
      width: 100,
      height: 100,
      cursor: 'grab',
      backgroundColor: color,
      position: 'absolute',
      border: '1px solid black',
      display: 'flex',
      justifyContent: 'center',
      alignItems: 'center',
      transform: `translate(
        ${position.x}px,
        ${position.y}px
      )`,
    }}
    >{children}</div>
);
}
```

```
export default function Background({
  position
}) {
  return (
    <div style={{
      position: 'absolute',
      transform: `translate(
        ${position.x}px,
        ${position.y}px
      )`,
      width: 250,
      height: 250,
      backgroundColor: 'rgba(200, 200, 0, 0.2)',
    }} />
  );
};
```

```
body { height: 280px; }
select { margin-bottom: 10px; }
```

```
{
  "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"
}
```

Writing Markup with JSX

JSX 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.

- Why React mixes markup with rendering logic
- How JSX is different from HTML
- How to display information with JSX

JSX: Putting markup into JavaScript *{/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:

HTML

JavaScript

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 component

`Form.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.

JSX 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 *{/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:

```
export 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>
  );
}
```

```
img { height: 90px }
```

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.

Most 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 *{/the-rules-of-jsx/}*

1. Return a single root element *{/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>
  
  <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>
  
  <ul>
    ...
  </ul>
</>
```

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

Why do multiple JSX tags need to be wrapped? {/why-do-multiple-jsx-tags-need-to-be-wrapped/}

JSX 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{/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:

```
<>
  
  <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! {/3-camelcase-salls-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](#).

For historical reasons, `aria-*` and `data-*` attributes are written as in HTML with dashes.

Pro-tip: Use a JSX Converter {/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:

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

```
img { height: 90px }
```

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.

Convert some HTML to JSX [{/convert-some-html-to-jsx/}](#)

This HTML was pasted into a component, but it's not valid JSX. Fix it:

```
export 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</i></b> of scientists!
    </p>
  );
}
```

```
.intro {
  background-image: linear-gradient(to left, violet, indigo, blue, green, yellow, orange, red);
  background-clip: text;
  color: transparent;
  -webkit-background-clip: text;
  -webkit-text-fill-color: transparent;
}

.summary {
  padding: 20px;
  border: 10px solid gold;
}
```

Whether to do it by hand or using the converter is up to you!

```
export default function Bio() {
  return (
    <div>
      <div className="intro">
        <h1>Welcome to my website!</h1>
      </div>
      <p className="summary">
        You can find my thoughts here.
        <br /><br />
        <b>And <i>pictures</i></b> of scientists!
      </p>
    </div>
  );
}
```

```
.intro {
  background-image: linear-gradient(to left, violet, indigo, blue, green, yellow, orange, red);
  background-clip: text;
  color: transparent;
  -webkit-background-clip: text;
  -webkit-text-fill-color: transparent;
}

.summary {
```

```
padding: 20px;  
border: 10px solid gold;  
}
```

'You Might Not Need an Effect'

Effects are an escape hatch from the React paradigm. They let you "step outside" of React and synchronize your components with some external system 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.

- 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 {/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 {/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');

  const [fullName, setFullName] = useState('');
  useEffect(() => {
    setFullName(firstName + ' ' + lastName);
  }, [firstName, lastName]);
  // ...
}
```

This is more complicated than necessary. It is inefficient too: it does an entire render pass with a stale value for `fullName`, then immediately re-renders with the updated value. Remove the state variable and the Effect:

```
function Form() {
  const [firstName, setFirstName] = useState('Taylor');
  const [lastName, setLastName] = useState('Swift');
  // ✅ Good: calculated during rendering
  const fullName = firstName + ' ' + lastName;
  // ...
}
```

When something can be calculated from the existing props or state, don't put it in state. Instead, calculate it during rendering. This makes your code faster (you avoid the extra "cascading" updates), simpler (you remove some code), and less error-prone (you avoid bugs caused by different state variables getting out of sync with each other). If this approach feels new to you, [Thinking in React](#) explains what should go into state.

Caching expensive calculations {/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('');

  const [visibleTodos, setVisibleTodos] = useState([]);
  useEffect(() => {
    setVisibleTodos(getFilteredTodos(todos, filter));
  }, [todos, filter]);
  // ...
}
```

Like in the earlier example, this is both unnecessary and inefficient. First, remove the state and the Effect:

```
function TodoList({ todos, filter }) {
  const [newTodo, setNewTodo] = useState('');
  // ✅ This is fine if getFilteredTodos() is not slow.
  const visibleTodos = getFilteredTodos(todos, filter);
  // ...
}
```

Usually, this code is fine! But maybe `getFilteredTodos()` is slow or you have a lot of `todos`. In that case you don't want to recalculate `getFilteredTodos()` if some unrelated state variable like `newTodo` has changed.

You can cache (or "memoize") an expensive calculation by wrapping it in a `useMemo` Hook:

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

function TodoList({ todos, filter }) {
  const [newTodo, setNewTodo] = useState('');
  const visibleTodos = useMemo(() => {
    // ✅ Does not re-run unless todos or filter change
    return getFilteredTodos(todos, filter);
  }, [todos, filter]);
  // ...
}
```

Or, written as a single line:

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

function TodoList({ todos, filter }) {
  const [newTodo, setNewTodo] = useState('');
  // ✅ Does not re-run getFilteredTodos() unless todos or filter change
  const visibleTodos = useMemo(() => getFilteredTodos(todos, filter), [todos,
filter]);
  // ...
}
```

This tells React that you don't want the inner function to re-run unless either `todos` or `filter` have changed. React will remember the return value of `getFilteredTodos()` during the initial render. During the next renders, it will check if `todos` or `filter` are different. If they're the same as last time, `useMemo` will return the last result it has stored. But if they are different, React will call the inner function again (and store its result).

The function you wrap in `useMemo` runs during rendering, so this only works for [pure calculations](#).

How to tell if a calculation is expensive? [{/how-to-tell-if-a-calculation-is-expensive/}](#)

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 {/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('');

  useEffect(() => {
    setComment('');
  }, [userId]);
  // ...
}
```

This is inefficient because `ProfilePage` and its children will first render with the stale value, and then render again. It is also complicated because you'd need to do this in every component that has some state inside `ProfilePage`. For example, if the comment UI is nested, you'd want to clear out nested comment state too.

Instead, you can tell React that each user's profile is conceptually a *different* profile by giving it an explicit key. Split your component in two and pass a `key` attribute from the outer component to the inner one:

```
export default function ProfilePage({ userId }) {
  return (
    <Profile
      userId={userId}
```

```

    key={userId}
  />
);
}

function Profile({ userId }) {
  // ✅ This and any other state below will reset on key change automatically
  const [comment, setComment] = useState('');
  // ...
}

```

Normally, React preserves the state when the same component is rendered in the same spot. **By passing `userId` as a `key` to the `Profile` component, you're asking React to treat two `Profile` components with different `userId` as two different components that should not share any state.** Whenever the key (which you've set to `userId`) changes, React will recreate the DOM and `reset the state` of the `Profile` component and all of its children. Now the `comment` field will clear out automatically when navigating between profiles.

Note that in this example, only the outer `ProfilePage` component is exported and visible to other files in the project. Components rendering `ProfilePage` don't need to pass the key to it: they pass `userId` as a regular prop. The fact `ProfilePage` passes it as a `key` to the inner `Profile` component is an implementation detail.

Adjusting some state when a prop changes {/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);

  useEffect(() => {
    setSelection(null);
  }, [items]);
  // ...
}

```

This, too, is not ideal. Every time the `items` change, the `List` and its child components will render with a stale `selection` value at first. Then React will update the DOM and run the Effects. Finally, the `setSelection(null)` call will cause another re-render of the `List` and its child components, restarting this whole process again.

Start by deleting the Effect. Instead, adjust the state directly during rendering:

```

function List({ items }) {
  const [isReverse, setIsReverse] = useState(false);
  const [selection, setSelection] = useState(null);

```

```
// Better: Adjust the state while rendering
const [prevItems, setPrevItems] = useState(items);
if (items !== prevItems) {
  setPrevItems(items);
  setSelection(null);
}
// ...
```

Storing information from previous renders like this can be hard to understand, but it's better than updating the same state in an Effect. In the above example, `setSelection` is called directly during a render. React will re-render the `List` immediately after it exits with a `return` statement. React has not rendered the `List` children or updated the DOM yet, so this lets the `List` children skip rendering the stale `selection` value.

When you update a component during rendering, React throws away the returned JSX and immediately retries rendering. To avoid very slow cascading retries, React only lets you update the *same* component's state during a render. If you update another component's state during a render, you'll see an error. A condition like `items !== prevItems` is necessary to avoid loops. You may adjust state like this, but any other side effects (like changing the DOM or setting timeouts) should stay in event handlers or Effects to [keep components pure](#).

Although this pattern is more efficient than an Effect, most components shouldn't need it either. No matter how you do it, adjusting state based on props or other state makes your data flow more difficult to understand and debug. Always check whether you can [reset all state with a key](#) or [calculate everything during rendering](#) instead. For example, instead of storing (and resetting) the selected `item`, you can store the selected `item ID`:

```
function List({ items }) {
  const [isReverse, setIsReverse] = useState(false);
  const [selectedId, setSelectedId] = useState(null);
  // ✅ Best: Calculate everything during rendering
  const selection = items.find(item => item.id === selectedId) ?? null;
  // ...
}
```

Now there is no need to "adjust" the state at all. If the item with the selected ID is in the list, it remains selected. If it's not, the `selection` calculated during rendering will be `null` because no matching item was found. This behavior is different, but arguably better because most changes to `items` preserve the selection.

Sharing logic between event handlers [{/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 }) {
  useEffect(() => {
    if (product isInCart) {
      showNotification(`Added ${product.name} to the shopping cart!`);
    }
  }, [product]);

  function handleBuyClick() {
    addToCart(product);
  }

  function handleCheckoutClick() {
    addToCart(product);
    navigateTo('/checkout');
  }
  // ...
}

```

This Effect is unnecessary. It will also most likely cause bugs. For example, let's say that your app "remembers" the shopping cart between the page reloads. If you add a product to the cart once and refresh the page, the notification will appear again. It will keep appearing every time you refresh that product's page. This is because `product.isInCart` will already be `true` on the page load, so the Effect above will call `showNotification()`.

When you're not sure whether some code should be in an Effect or in an event handler, ask yourself why this code needs to run. Use Effects only for code that should run because the component was displayed to the user. In this example, the notification should appear because the user *pressed the button*, not because the page was displayed! Delete the Effect and put the shared logic into a function called from both event handlers:

```

function ProductPage({ product, addToCart }) {
  // ✅ Good: Event-specific logic is called from event handlers
  function buyProduct() {
    addToCart(product);
    showNotification(`Added ${product.name} to the shopping cart!`);
  }

  function handleBuyClick() {
    buyProduct();
  }

  function handleCheckoutClick() {
    buyProduct();
    navigateTo('/checkout');
  }
  // ...
}

```

This both removes the unnecessary Effect and fixes the bug.

Sending a POST request {/sending-a-post-request/}

This `Form` component sends two kinds of POST requests. It sends an analytics event when it mounts. When you fill in the form and click the Submit button, it will send a POST request to the `/api/register` endpoint:

```

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

  // ✅ Good: This logic should run because the component was displayed
  useEffect(() => {
    post('/analytics/event', { eventName: 'visit_form' });
  }, []);

  const [jsonToSubmit, setJsonToSubmit] = useState(null);
  useEffect(() => {
    if (jsonToSubmit !== null) {
      post('/api/register', jsonToSubmit);
    }
  }, [jsonToSubmit]);

  function handleSubmit(e) {
    e.preventDefault();
    setJsonToSubmit({ firstName, lastName });
  }
  // ...
}

```

Let's apply the same criteria as in the example before.

The analytics POST request should remain in an Effect. This is because the *reason* to send the analytics event is that the form was displayed. (It would fire twice in development, but [see here](#) for how to deal with that.)

However, the `/api/register` POST request is not caused by the form being *displayed*. You only want to send the request at one specific moment in time: when the user presses the button. It should only ever happen *on that particular interaction*. Delete the second Effect and move that POST request into the event handler:

```

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

  // ✅ Good: This logic runs because the component was displayed
  useEffect(() => {
    post('/analytics/event', { eventName: 'visit_form' });
  }, []);

  function handleSubmit(e) {
    e.preventDefault();
    // ✅ Good: Event-specific logic is in the event handler
    post('/api/register', { firstName, lastName });
  }
  // ...
}

```

When you choose whether to put some logic into an event handler or an Effect, the main question you need to answer is *what kind of logic* it is from the user's perspective. If this logic is caused by a particular interaction, keep it in the event handler. If it's caused by the user *seeing* the component on the screen, keep it in the Effect.

Chains of computations {/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);

  useEffect(() => {
    if (card !== null && card.gold) {
      setGoldCardCount(c => c + 1);
    }
  }, [card]);

  useEffect(() => {
    if (goldCardCount > 3) {
      setRound(r => r + 1)
      setGoldCardCount(0);
    }
  }, [goldCardCount]);

  useEffect(() => {
    if (round > 5) {
      setIsGameOver(true);
    }
  }, [round]);

  useEffect(() => {
    alert('Good game!');
  }, [isGameOver]);

  function handlePlaceCard(nextCard) {
    if (isGameOver) {
      throw Error('Game already ended.');
    } else {
      setCard(nextCard);
    }
  }
}

// ...
```

There are two problems with this code.

One problem is that it is very inefficient: the component (and its children) have to re-render between each `set` call in the chain. In the example above, in the worst case (`setCard` → render → `setGoldCardCount` → render → `setRound` → render → `setIsGameOver` → render) there are three unnecessary re-renders of the tree below.

Even if it weren't slow, as your code evolves, you will run into cases where the "chain" you wrote doesn't fit the new requirements. Imagine you are adding a way to step through the history of the game moves. You'd do it by updating each state variable to a value from the past. However, setting the `card` state to a value from the past would trigger the Effect chain again and change the data you're showing. Such code is often rigid and fragile.

In this case, it's better to calculate what you can during rendering, and adjust the state in the event handler:

```
function Game() {
  const [card, setCard] = useState(null);
  const [goldCardCount, setGoldCardCount] = useState(0);
  const [round, setRound] = useState(1);

  // ✅ Calculate what you can during rendering
  const isGameOver = round > 5;

  function handlePlaceCard(nextCard) {
    if (isGameOver) {
      throw Error('Game already ended.');
    }

    // ✅ Calculate all the next state in the event handler
    setCard(nextCard);
    if (nextCard.gold) {
      if (goldCardCount <= 3) {
        setGoldCardCount(goldCardCount + 1);
      } else {
        setGoldCardCount(0);
        setRound(round + 1);
        if (round === 5) {
          alert('Good game!');
        }
      }
    }
  }
}

// ...
```

This is a lot more efficient. Also, if you implement a way to view game history, now you will be able to set each state variable to a move from the past without triggering the Effect chain that adjusts every other value. If you need to reuse logic between several event handlers, you can [extract a function](#) and call it from those handlers.

Remember that inside event handlers, [state behaves like a snapshot](#). For example, even after you call `setRound(round + 1)`, the `round` variable will reflect the value at the time the user clicked the button. If you need to use the next value for calculations, define it manually like `const nextRound = round + 1`.

In some cases, you *can't* calculate the next state directly in the event handler. For example, imagine a form with multiple dropdowns where the options of the next dropdown depend on the selected value of the previous dropdown. Then, a chain of Effects is appropriate because you are synchronizing with network.

Initializing the application { /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() {
  useEffect(() => {
    loadDataFromLocalStorage();
    checkAuthToken();
  });
}
```

```
}, []);
// ...
}
```

However, you'll quickly discover that it [runs twice in development](#). This can cause issues--for example, maybe it invalidates the authentication token because the function wasn't designed to be called twice. In general, your components should be resilient to being remounted. This includes your top-level [App](#) component.

Although it may not ever get remounted in practice in production, following the same constraints in all components makes it easier to move and reuse code. If some logic must run *once per app load* rather than *once per component mount*, add a top-level variable to track whether it has already executed:

```
let didInit = false;

function App() {
  useEffect(() => {
    if (!didInit) {
      didInit = true;
      // ✅ Only runs once per app load
      loadDataFromLocalStorage();
      checkAuthToken();
    }
  }, []);
// ...
}
```

You can also run it during module initialization and before the app renders:

```
if (typeof window !== 'undefined') { // Check if we're running in the browser.
  // ✅ Only runs once per app load
  checkAuthToken();
  loadDataFromLocalStorage();
}

function App() {
  // ...
}
```

Code at the top level runs once when your component is imported--even if it doesn't end up being rendered. To avoid slowdown or surprising behavior when importing arbitrary components, don't overuse this pattern. Keep app-wide initialization logic to root component modules like [App.js](#) or in your application's entry point.

Notifying parent components about state changes {/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);

  useEffect(() => {
    onChange(isOn);
  }, [isOn, onChange])

  function handleClick() {
    setIsOn(!isOn);
  }

  function handleDragEnd(e) {
    if (isCloserToRightEdge(e)) {
      setIsOn(true);
    } else {
      setIsOn(false);
    }
  }

  // ...
}
```

Like earlier, this is not ideal. The `Toggle` updates its state first, and React updates the screen. Then React runs the Effect, which calls the `onChange` function passed from a parent component. Now the parent component will update its own state, starting another render pass. It would be better to do everything in a single pass.

Delete the Effect and instead update the state of *both* components within the same event handler:

```
function Toggle({ onChange }) {
  const [isOn, setIsOn] = useState(false);

  function updateToggle(nextIsOn) {
    // ✅ Good: Perform all updates during the event that caused them
    setIsOn(nextIsOn);
    onChange(nextIsOn);
  }

  function handleClick() {
    updateToggle(!isOn);
  }

  function handleDragEnd(e) {
    if (isCloserToRightEdge(e)) {
      updateToggle(true);
    } else {
      updateToggle(false);
    }
  }

  // ...
}
```

With this approach, both the `Toggle` component and its parent component update their state during the event. React `batches updates` from different components together, so there will only be one render pass.

You might also be able to remove the state altogether, and instead receive `isOn` from the parent component:

```
// ✅ Also good: the component is fully controlled by its 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 [`{/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();
  useEffect(() => {
    if (data) {
      onFetched(data);
    }
  }, [onFetched, data]);
  // ...
}
```

In React, data flows from the parent components to their children. When you see something wrong on the screen, you can trace where the information comes from by going up the component chain until you find which component passes the wrong prop or has the wrong state. When child components update the state of their parent components in Effects, the data flow becomes very difficult to trace. Since both the child and the parent need the same data, let the parent component fetch that data, and *pass it down* to the child instead:

```
function Parent() {
  const data = useSomeAPI();
  // ...
  // ✅ Good: Passing data down to the child
  return <Child data={data} />;
}
```

```
function Child({ data }) {
  // ...
}
```

This is simpler and keeps the data flow predictable: the data flows down from the parent to the child.

Subscribing to an external store { /subscribing-to-an-external-store/ }

Sometimes, your components may need to subscribe to some data outside of the React state. This data could be from a third-party library or a built-in browser API. Since this data can change without React's knowledge, you need to manually subscribe your components to it. This is often done with an Effect, for example:

```
function useOnlineStatus() {
  // Not ideal: Manual store subscription in an Effect
  const [isOnline, setIsOnline] = useState(true);
  useEffect(() => {
    function updateState() {
      setIsOnline(navigator.onLine);
    }

    updateState();

    window.addEventListener('online', updateState);
    window.addEventListener('offline', updateState);
    return () => {
      window.removeEventListener('online', updateState);
      window.removeEventListener('offline', updateState);
    };
  }, []);
  return isOnline;
}

function ChatIndicator() {
  const isOnline = useOnlineStatus();
  // ...
}
```

Here, the component subscribes to an external data store (in this case, the browser `navigator.onLine` API). Since this API does not exist on the server (so it can't be used for the initial HTML), initially the state is set to `true`. Whenever the value of that data store changes in the browser, the component updates its state.

Although it's common to use Effects for this, React has a purpose-built Hook for subscribing to an external store that is preferred instead. Delete the Effect and replace it with a call to `useSyncExternalStore`:

```
function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);
  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
}
```

```

function useOnlineStatus() {
  // ✅ Good: Subscribing to an external store with a built-in Hook
  return useSyncExternalStore(
    subscribe, // React won't resubscribe for as long as you pass the same
    function()
      () => navigator.onLine, // How to get the value on the client
      () => true // How to get the value on the server
    );
}

function ChatIndicator() {
  const isOnline = useOnlineStatus();
  // ...
}

```

This approach is less error-prone than manually syncing mutable data to React state with an Effect. Typically, you'll write a custom Hook like `useOnlineStatus()` above so that you don't need to repeat this code in the individual components. [Read more about subscribing to external stores from React components.](#)

Fetching data {/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(() => {
    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.

- 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.

Transform data without Effects {/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.

```

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>
    </>
  );
}

let nextId = 0;

export function createTodo(text, completed = false) {
  return {
    text,

```

```

    completed
};

export const initialTodos = [
  createTodo('Get apples', true),
  createTodo('Get oranges', true),
  createTodo('Get carrots'),
];

label { display: block; }
input { margin-top: 10px; }

```

If you can calculate something during rendering, you don't need state or an Effect that updates it.

There are only two essential pieces of state in this example: the list of `todos` and the `showActive` state variable which represents whether the checkbox is ticked. All of the other state variables are `redundant` and can be calculated during rendering instead. This includes the `footer` which you can move directly into the surrounding JSX.

Your result should end up looking like this:

```

import { useState } from 'react';
import { initialTodos, createTodo } from './todos.js';

export default function TodoList() {
  const [todos, setTodos] = useState(initialTodos);
  const [showActive, setShowActive] = useState(false);
  const activeTodos = todos.filter(todo => !todo.completed);
  const visibleTodos = showActive ? activeTodos : todos;

  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>
        {activeTodos.length} todos left
      </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>
  </>
);
}

```

```

let nextId = 0;

export function createTodo(text, completed = false) {
  return {
    text,
    completed
  };
}

export const initialTodos = [
  createTodo('Get apples', true),
  createTodo('Get oranges', true),
  createTodo('Get carrots'),
];

```

```

label { display: block; }
input { margin-top: 10px; }

```

Cache a calculation without Effects *{/cache-a-calculation-without-effects/}*

In this example, filtering the todos was extracted into a separate function called `getVisibleTodos()`. This function contains a `console.log()` call inside of it which helps you notice when it's being called. Toggle "Show only active todos" and notice that it causes `getVisibleTodos()` to re-run. This is expected because visible todos change when you toggle which ones to display.

Your task is to remove the Effect that recomputes the `visibleTodos` list in the `TodoList` component. However, you need to make sure that `getVisibleTodos()` does *not* re-run (and so does not print any logs) when you type into the input.

One solution is to add a `useMemo` call to cache the visible todos. There is also another, less obvious solution.

```

import { useState, useEffect } from 'react';
import { initialTodos, createTodo, getVisibleTodos } from './todos.js';

export default function TodoList() {
  const [todos, setTodos] = useState(initialTodos);
  const [showActive, setShowActive] = useState(false);
  const [text, setText] = useState('');
  const [visibleTodos, setVisibleTodos] = useState([]);

  useEffect(() => {

```

react

```
    setVisibleTodos(getVisibleTodos(todos, showActive));
}, [todos, showActive]);

function handleAddClick() {
  setText('');
  setTodos([...todos, createTodo(text)]);
}

return (
  <>
  <label>
    <input
      type="checkbox"
      checked={showActive}
      onChange={e => setShowActive(e.target.checked)}
    />
    Show only active todos
  </label>
  <input value={text} onChange={e => setText(e.target.value)} />
  <button onClick={handleAddClick}>
    Add
  </button>
  <ul>
    {visibleTodos.map(todo => (
      <li key={todo.id}>
        {todo.completed ? <s>{todo.text}</s> : todo.text}
      </li>
    ))}
  </ul>
</>
);
}
```

```
let nextId = 0;
let calls = 0;

export function getVisibleTodos(todos, showActive) {
  console.log(`getVisibleTodos() was called ${++calls} times`);
  const activeTodos = todos.filter(todo => !todo.completed);
  const visibleTodos = showActive ? activeTodos : todos;
  return visibleTodos;
}

export function createTodo(text, completed = false) {
  return {
    text,
    completed
  };
}

export const initialTodos = [
  createTodo('Get apples', true),
  createTodo('Get oranges', true),
  createTodo('Get carrots'),
];

```

```
label { display: block; }
input { margin-top: 10px; }
```

Remove the state variable and the Effect, and instead add a `useMemo` call to cache the result of calling `getVisibleTodos()`:

```

import { useState, useMemo } from 'react';
import { initialTodos, createTodo, getVisibleTodos } from './todos.js';

export default function TodoList() {
  const [todos, setTodos] = useState(initialTodos);
  const [showActive, setShowActive] = useState(false);
  const [text, setText] = useState('');
  const visibleTodos = useMemo(
    () => getVisibleTodos(todos, showActive),
    [todos, showActive]
  );

  function handleAddClick() {
    setText('');
    setTodos([...todos, createTodo(text)]);
  }

  return (
    <>
      <label>
        <input
          type="checkbox"
          checked={showActive}
          onChange={e => setShowActive(e.target.checked)}
        />
        Show only active todos
      </label>
      <input value={text} onChange={e => setText(e.target.value)} />
      <button onClick={handleAddClick}>
        Add
      </button>
      <ul>
        {visibleTodos.map(todo => (
          <li key={todo.id}>
            {todo.completed ? <s>{todo.text}</s> : todo.text}
          </li>
        ))}
      </ul>
    </>
  );
}

```

```

let nextId = 0;
let calls = 0;

export function getVisibleTodos(todos, showActive) {
  console.log(`getVisibleTodos() was called ${++calls} times`);
  const activeTodos = todos.filter(todo => !todo.completed);
  const visibleTodos = showActive ? activeTodos : todos;
  return visibleTodos;
}

export function createTodo(text, completed = false) {
  return {
    text,
    completed
  };
}

export const initialTodos = [
  createTodo('Get apples', true),

```

```
createTodo('Get oranges', true),
createTodo('Get carrots'),
];
```

```
label { display: block; }
input { margin-top: 10px; }
```

With this change, `getVisibleTodos()` will be called only if `todos` or `showActive` change. Typing into the input only changes the `text` state variable, so it does not trigger a call to `getVisibleTodos()`.

There is also another solution which does not need `useMemo`. Since the `text` state variable can't possibly affect the list of todos, you can extract the `NewTodo` form into a separate component, and move the `text` state variable inside of it:

```
import { useState, useMemo } from 'react';
import { initialTodos, createTodo, getVisibleTodos } from './todos.js';

export default function TodoList() {
  const [todos, setTodos] = useState(initialTodos);
  const [showActive, setShowActive] = useState(false);
  const visibleTodos = getVisibleTodos(todos, showActive);

  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>
    </>
  );
}

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>
    </>
  );
}
```

```

        </>
    );
}

let nextId = 0;
let calls = 0;

export function getVisibleTodos(todos, showActive) {
  console.log(`getVisibleTodos() was called ${++calls} times`);
  const activeTodos = todos.filter(todo => !todo.completed);
  const visibleTodos = showActive ? activeTodos : todos;
  return visibleTodos;
}

export function createTodo(text, completed = false) {
  return {
    text,
    completed
  };
}

export const initialTodos = [
  createTodo('Get apples', true),
  createTodo('Get oranges', true),
  createTodo('Get carrots'),
];

```

```

label { display: block; }
input { margin-top: 10px; }

```

This approach satisfies the requirements too. When you type into the input, only the `text` state variable updates. Since the `text` state variable is in the child `NewTodo` component, the parent `TodoList` component won't get re-rendered. This is why `getVisibleTodos()` doesn't get called when you type. (It would still be called if the `TodoList` re-renders for another reason.)

Reset state without Effects {/reset-state-without-effects/}

This `EditContact` component receives a contact object shaped like `{ id, name, email }` as the `savedContact` prop. Try editing the name and email input fields. When you press Save, the contact's button above the form updates to the edited name. When you press Reset, any pending changes in the form are discarded. Play around with this UI to get a feel for it.

When you select a contact with the buttons at the top, the form resets to reflect that contact's details. This is done with an Effect inside `EditContact.js`. Remove this Effect. Find another way to reset the form when `savedContact.id` changes.

```

import { useState } from 'react';
import ContactList from './ContactList.js';
import EditContact from './EditContact.js';

export default function ContactManager() {
  const [
    contacts,
    setContacts
  ] = useState(initialContacts);
  const [

```

```

    selectedId,
    setSelectedId
] = useState(0);
const selectedContact = contacts.find(c =>
  c.id === selectedId
);

function handleSave(updatedData) {
  const nextContacts = contacts.map(c => {
    if (c.id === updatedData.id) {
      return updatedData;
    } else {
      return c;
    }
  });
  setContacts(nextContacts);
}

return (
  <div>
    <ContactList
      contacts={contacts}
      selectedId={selectedId}
      onSelect={id => setSelectedId(id)}
    />
    <hr />
    <EditContact
      savedContact={selectedContact}
      onSave={handleSave}
    />
  </div>
)
};

const initialContacts = [
];

```

```

export default function ContactList({
  contacts,
  selectedId,
  onSelect
}) {
  return (
    <section>
      <ul>
        {contacts.map(contact =>
          <li key={contact.id}>
            <button onClick={() => {
              onSelect(contact.id);
            }}>
              {contact.id === selectedId ? 
                <b>{contact.name}</b> :
                contact.name
              }
            </button>
          </li>
        )}
      </ul>
    </section>
  );
}

```

```

import { useState, useEffect } from 'react';

export default function EditContact({ savedContact, onSave }) {
  const [name, setName] = useState(savedContact.name);
  const [email, setEmail] = useState(savedContact.email);

  useEffect(() => {
    setName(savedContact.name);
    setEmail(savedContact.email);
  }, [savedContact]);

  return (
    <section>
      <label>
        Name:{' '}
        <input
          type="text"
          value={name}
          onChange={e => setName(e.target.value)}
        />
      </label>
      <label>
        Email:{' '}
        <input
          type="email"
          value={email}
          onChange={e => setEmail(e.target.value)}
        />
      </label>
      <button onClick={() => {
        const updatedData = {
          name,
          email
        };
        onSave(updatedData);
      }}>
        Save
      </button>
      <button onClick={() => {
        setName(savedContact.name);
        setEmail(savedContact.email);
      }}>
        Reset
      </button>
    </section>
  );
}

```

```

ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li { display: inline-block; }
li button {
  padding: 10px;
}
label {
  display: block;
  margin: 10px 0;
}
button {

```

```
    margin-right: 10px;
    margin-bottom: 10px;
}
```

It would be nice if there was a way to tell React that when `savedContact.id` is different, the `EditContact` form is conceptually a *different contact's form* and should not preserve state. Do you recall any such way?

Split the `EditContact` component in two. Move all the form state into the inner `EditForm` component. Export the outer `EditContact` component, and make it pass `savedContact.id` as the `key` to the inner `EditContact` component. As a result, the inner `EditForm` component resets all of the form state and recreates the DOM whenever you select a different contact.

```
import { useState } from 'react';
import ContactList from './ContactList.js';
import EditContact from './EditContact.js';

export default function ContactManager() {
  const [
    contacts,
    setContacts
  ] = useState(initialContacts);
  const [
    selectedId,
    setSelectedId
  ] = useState(0);
  const selectedContact = contacts.find(c =>
    c.id === selectedId
  );

  function handleSave(updatedData) {
    const nextContacts = contacts.map(c => {
      if (c.id === updatedData.id) {
        return updatedData;
      } else {
        return c;
      }
    });
    setContacts(nextContacts);
  }

  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedId={selectedId}
        onSelect={id => setSelectedId(id)}
      />
      <hr />
      <EditContact
        savedContact={selectedContact}
        onSave={handleSave}
      />
    </div>
  )
}

const initialContacts = [
```

```
export default function ContactList({
  contacts,
  selectedId,
  onSelect
}) {
  return (
    <section>
      <ul>
        {contacts.map(contact =>
          <li key={contact.id}>
            <button onClick={() => {
              onSelect(contact.id);
            }}>
              {contact.id === selectedId ?
                <b>{contact.name}</b> :
                contact.name
              }
            </button>
          </li>
        )}
      </ul>
    </section>
  );
}
```

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

export default function EditContact(props) {
  return (
    <EditForm
      {...props}
      key={props.savedContact.id}
    />
  );
}

function EditForm({ savedContact, onSave }) {
  const [name, setName] = useState(savedContact.name);
  const [email, setEmail] = useState(savedContact.email);

  return (
    <section>
      <label>
        Name:{' '}
        <input
          type="text"
          value={name}
          onChange={e => setName(e.target.value)}
        />
      </label>
      <label>
        Email:{' '}
        <input
          type="email"
          value={email}
          onChange={e => setEmail(e.target.value)}
        />
      </label>
      <button onClick={() => {
        const updatedData = {
          name: name,
          email: email
        };
        onSave(updatedData);
      }}>
        Save
      </button>
    </section>
  );
}
```

```

    };
    onSave(updatedData);
  }>
  Save
</button>
<button onClick={() => {
  setName(savedContact.name);
  setEmail(savedContact.email);
}}>
  Reset
</button>
</section>
);
}

```

```

ul, li {
  list-style: none;
  margin: 0;
  padding: 0;
}
li { display: inline-block; }
li button {
  padding: 10px;
}
label {
  display: block;
  margin: 10px 0;
}
button {
  margin-right: 10px;
  margin-bottom: 10px;
}

```

Submit a form without Effects [{/submit-a-form-without-effects/}](#)

This `Form` component lets you send a message to a friend. When you submit the form, the `showForm` state variable is set to `false`. This triggers an Effect calling `sendMessage(message)`, which sends the message (you can see it in the console). After the message is sent, you see a "Thank you" dialog with an "Open chat" button that lets you get back to the form.

Your app's users are sending way too many messages. To make chatting a little bit more difficult, you've decided to show the "Thank you" dialog *first* rather than the form. Change the `showForm` state variable to initialize to `false` instead of `true`. As soon as you make that change, the console will show that an empty message was sent. Something in this logic is wrong!

What's the root cause of this problem? And how can you fix it?

Should the message be sent *because* the user saw the "Thank you" dialog? Or is it the other way around?

```

import { useState, useEffect } from 'react';

export default function Form() {
  const [showForm, setShowForm] = useState(true);
  const [message, setMessage] = useState('');

  useEffect(() => {
    if (!showForm) {

```

```

        sendMessage(message);
    },
    [showForm, message]);
}

function handleSubmit(e) {
    e.preventDefault();
    setShowForm(false);
}

if (!showForm) {
    return (
        <>
            <h1>Thanks for using our services!</h1>
            <button onClick={() => {
                setMessage('');
                setShowForm(true);
            }}>
                Open chat
            </button>
        </>
    );
}

return (
    <form onSubmit={handleSubmit}>
        <textarea
            placeholder="Message"
            value={message}
            onChange={e => setMessage(e.target.value)}
        />
        <button type="submit" disabled={message === ''}>
            Send
        </button>
    </form>
);
}

function sendMessage(message) {
    console.log('Sending message: ' + message);
}

```

```
label, textarea { margin-bottom: 10px; display: block; }
```

The `showForm` state variable determines whether to show the form or the "Thank you" dialog. However, you aren't sending the message because the "Thank you" dialog was *displayed*. You want to send the message because the user has *submitted the form*. Delete the misleading Effect and move the `sendMessage` call inside the `handleSubmit` event handler:

```

import { useState, useEffect } from 'react';

export default function Form() {
    const [showForm, setShowForm] = useState(true);
    const [message, setMessage] = useState('');

    function handleSubmit(e) {
        e.preventDefault();
        setShowForm(false);
        sendMessage(message);
    }
}
```

```

if (!showForm) {
  return (
    <>
      <h1>Thanks for using our services!</h1>
      <button onClick={() => {
        setMessage('');
        setShowForm(true);
      }}>
        Open chat
      </button>
    </>
  );
}

return (
  <form onSubmit={handleSubmit}>
    <textarea
      placeholder="Message"
      value={message}
      onChange={e => setMessage(e.target.value)}
    />
    <button type="submit" disabled={message === ''}>
      Send
    </button>
  </form>
);
}

function sendMessage(message) {
  console.log('Sending message: ' + message);
}

```

```
label, textarea { margin-bottom: 10px; display: block; }
```

Notice how in this version, only *submitting the form* (which is an event) causes the message to be sent. It works equally well regardless of whether `showForm` is initially set to `true` or `false`. (Set it to `false` and notice no extra console messages.)

Your First Component

Components 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!

- What a component is
- What role components play in a React application
- How to write your first React component

Components: UI building blocks {/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 *{/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):

```
export default function Profile() {
  return (
    
  )
}



```

And here's how to build a component:

Step 1: Export the 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 *{/step-2-define-the-function/}*

With `function Profile() { }` you define a JavaScript function with the name `Profile`.

React components are regular JavaScript functions, but **their names must start with a capital letter** or they won't work!

Step 3: Add markup *{/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>
);
```

Without parentheses, any code on the lines after `return` will be ignored!

Using a component *{/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:

```
function Profile() {
  return (
    
  );
}

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

```
img { margin: 0 10px 10px 0; height: 90px; }
```

What the browser sees *{/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 {/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.

Components 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() {
  // ...
}

// ✅ 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.

Components all the way down {/components-all-the-way-down/}

Your React application begins at a "root" component. Usually, it is created automatically when you start a new project. For example, if you use [CodeSandbox](#) or [Create React App](#), the root component is defined in `src/App.js`. If you use the framework [Next.js](#), the root component is defined in `pages/index.js`. In these examples, you've been exporting root components.

Most React apps use components all the way down. This means that you won't only use components for reusable pieces like buttons, but also for larger pieces like sidebars, lists, and ultimately, complete pages! Components are a handy way to organize UI code and markup, even if some of them are only used once.

[React-based frameworks](#) take this a step further. Instead of using an empty HTML file and letting React "take over" managing the page with JavaScript, they *also* generate the HTML automatically from your React components. This allows your app to show some content before the JavaScript code loads.

Still, many websites only use React to [add interactivity to existing HTML pages](#). They have many root components instead of a single one for the entire page. You can use as much—or as little—React as you need.

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:
 1. Their names always begin with a capital letter.
 2. They return JSX markup.

Export the component *{/export-the-component/}*

This sandbox doesn't work because the root component is not exported:

```
function Profile() {
  return (
    
  );
}

img { height: 181px; }
```

Try to fix it yourself before looking at the solution!

Add `export default` before the function definition like so:

```
export default function Profile() {
  return (
    
  );
}

img { height: 181px; }
```

You might be wondering why writing `export` alone is not enough to fix this example. You can learn the difference between `export` and `export default` in [Importing and Exporting Components](#).

Fix the return statement *{/fix-the-return-statement/}*

Something isn't right about this `return` statement. Can you fix it?

You may get an "Unexpected token" error while trying to fix this. In that case, check that the semicolon appears *after* the closing parenthesis. Leaving a semicolon inside `return ()` will cause an error.

```
export default function Profile() {
  return
    ;
}
```

```
img { height: 180px; }
```

You can fix this component by moving the return statement to one line like so:

```
export default function Profile() {
  return ;
}
```

```
img { height: 180px; }
```

Or by wrapping the returned JSX markup in parentheses that open right after `return`:

```
export default function Profile() {
  return (
    
  );
}
```

```
img { height: 180px; }
```

Spot the mistake {/spot-the-mistake/}

Something's wrong with how the `Profile` component is declared and used. Can you spot the mistake? (Try to remember how React distinguishes components from the regular HTML tags!)

```
function profile() {
  return (
    
  );
}

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

```
img { margin: 0 10px 10px 0; height: 90px; }
```

React component names must start with a capital letter.

Change `function profile()` to `function Profile()`, and then change every `<profile />` to `<Profile />`:

```
function Profile() {
  return (
    
  );
}

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

```
img { margin: 0 10px 10px 0; }
```

Your own component `{/your-own-component/}`

Write a component from scratch. You can give it any valid name and return any markup. If you're out of ideas, you can write a `Congratulations` component that shows `<h1>Good job!</h1>`. Don't forget to export it!

```
// Write your component below!
```

```
export default function Congratulations() {
  return (
    <h1>Good job!</h1>
  );
}
```

Colophon

This book is created by using the following sources:

- React - English
- GitHub source: [reactjs/react.dev/src/content/learn](https://github.com/reactjs/react.dev/tree/main/src/content/learn)
- Created: 2023-04-29
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>