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

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

Responding to events

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

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

```
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>
  );
}
```

State: a component’s memory

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

```
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.' }];

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

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

index is a state variable and setIndex is the setter function.

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

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

```
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}
```

```
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>`	`    <p>`
`  </>`	`      {sculpture.description}`
`);`	`    </p>`
`}`	`  </>`
	`);`
	`}`

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 .

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.

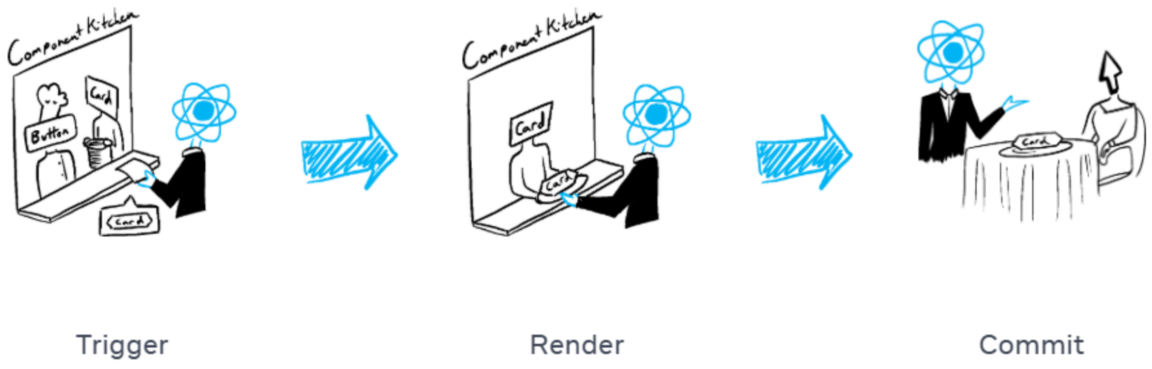
Hooks are functions, but it’s helpful to think of them as unconditional declarations about your component’s needs. You “use” React features at the top of your component similar to how you “import” modules at the top of your file.

Render and commit

Before components are displayed on the screen, they must be rendered by React.

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)



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:

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

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:

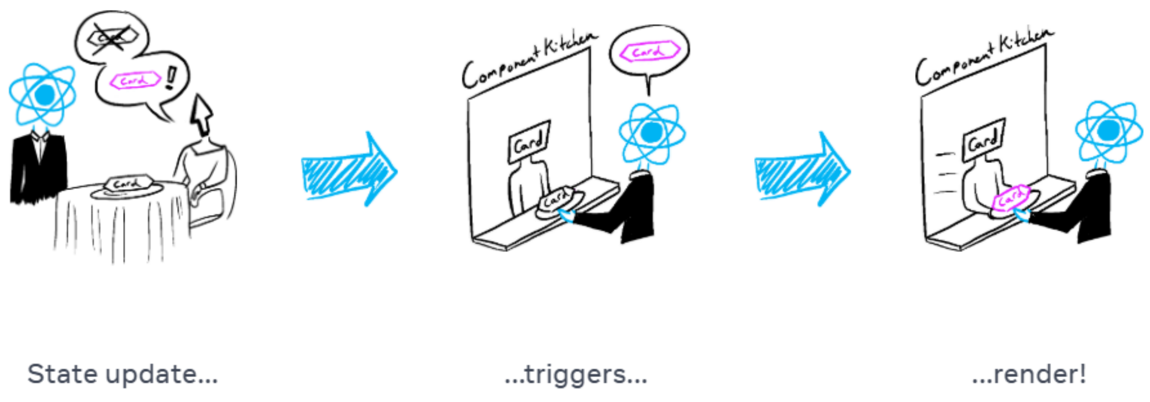
```
import Image from './Image.js';
import { createRoot } from 'react-dom/client';

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

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

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

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.

```
export default function Gallery() {
  return (
    <section>
      <h1>Inspiring Sculptures</h1>
      <Image />
      <Image />
      <Image />
    </section>
  );
}

function Image() {
  return (
    <img
```

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

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

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 />
    </>
  );
}
```

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!

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!
```

State as a snapshot

```
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>
  );
}
```

Functional Components:

- **Structure:** Functional components are simple JavaScript functions that take props as input and return JSX.
- **State (with Hooks):** They can manage state using React Hooks (e.g., useState, useEffect).
- **Lifecycle (with Hooks):** They can handle lifecycle events using Hooks (e.g., useEffect).
- **Example**

JavaScript

```
import React from 'react';

function MyComponent(props) {
  return (
    <div>

      <h1>Hello, {props.name}!</h1>

    </div>
  );
}

export default MyComponent;
```

Class Components:

- **Structure:** Class components are ES6 classes that extend React.Component.
- **State:** They have a dedicated state object to manage component-specific data.
- **Lifecycle Methods:** They can use lifecycle methods (e.g., componentDidMount, componentDidUpdate) to manage side effects and interactions with the DOM.
- **Example**

JavaScript

```
import React from 'react';

class MyComponent extends React.Component {
  render() {
    return (
      <div>
```

```
        <h1>Hello, {this.props.name}!</h1>
      </div>
    );
  }
}

export default MyComponent;
```

Updating array of objects in state

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.

Setting state triggers renders

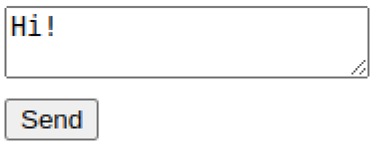
You might think of your user interface as changing directly in response to the user event like a click. In React, it works a little differently from this mental model. On the previous page, you saw that setting state requests a re-render from React. This means that for an interface to react to the event, 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) {
  // ...
}
```



what happens when you click the button:

The onSubmit event handler executes.

setIsSent(true) sets isSent to true and queues a new render.

React re-renders the component according to the new isSent value.

Let’s take a closer look at the relationship between state and rendering.

Rendering takes a snapshot in time

“Rendering” means that React is calling your component, which is a function. The JSX you return from that function is like a snapshot of the UI in time. Its props, event handlers, and local variables were all calculated using its state at the time of the render.

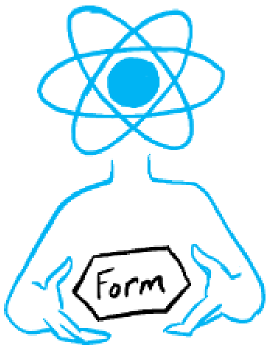
Unlike a photograph or a movie frame, the UI “snapshot” you return is interactive. It includes logic like event handlers that specify what happens in response to inputs. React updates the screen to match this snapshot and connects the event handlers. As a result, pressing a button will trigger the click handler from your JSX.

When React re-renders a component:

React calls your function again.

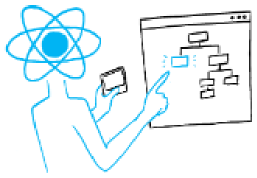
Your function returns a new JSX snapshot.

React then updates the screen to match the snapshot your function returned.



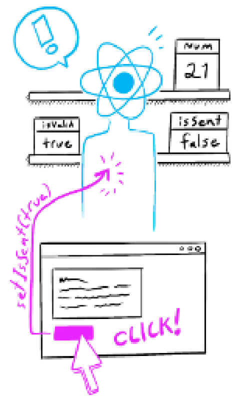
React executing the function

Calculating the snapshot

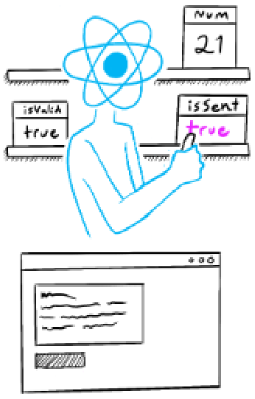


Updating the DOM tree

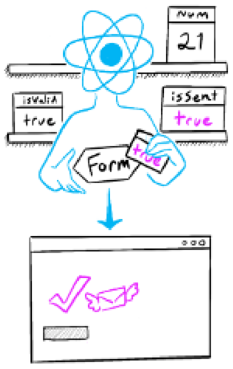
As a component's memory, state is not like a regular variable that disappears after your function returns. State actually "lives" in React itself—as if on a shelf!—outside of your function. When React calls your component, it gives you a snapshot of the state for that particular render. Your component returns a snapshot of the UI with a fresh set of props and event handlers in its JSX, all calculated using the state values from that render!\



You tell React to update the state



React updates the state value



React passes a snapshot of the state value into the component

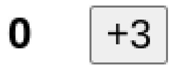
Here’s a little experiment to show you how this works. In this example, you might expect that clicking the “+3” button would increment the counter three times because it calls `setNumber(number + 1)` three times.

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>
    </>
  )
}
```



Notice that number only increments once per click!

Setting state only changes it for the next render. During the first render, number was 0. This is why, in that render’s onClick handler, the value of number is still 0 even after setNumber(number + 1) was called:

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

Here is what this button’s click handler tells React to do:

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

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

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

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

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

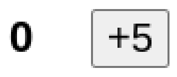
React prepares to change number to 1 on the next render.

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:

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

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

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



```
      }}>+5</button>
    </>
  )
}
```

If you use the substitution method from before, you can guess that the alert shows “0”:

```
setNumber(0 + 5);
alert(0);
```

```
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>

    </>

  )

}
```

f you use the substitution method, you can see the “snapshot” of the state passed to the alert.

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:

Queueing a series of state updates

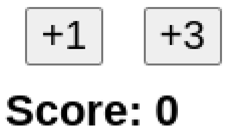
This component is buggy: clicking “+3” increments the score only once.

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



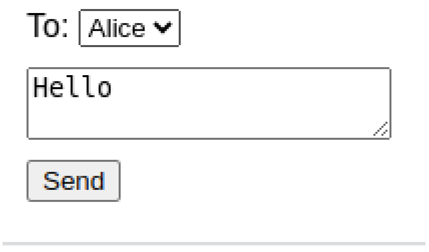
State as a Snapshot explains why this is happening. Setting state requests a new re-render, but does not change it in the already running code. So score continues to be 0 right after you call setScore(score + 1).

```
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>
  );
}
```



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:

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


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


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


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

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

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

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

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

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.

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	`push`, `unshift`	`concat`, `[...arr]` spread syntax

avoid (mutates the array)	prefer (returns a new array)	
		(example)
removing	`pop`, `shift`, `splice`	`filter`, `slice` (example)
replacing	`splice`, `arr[i] = ...` assignment	`map` (example)
sorting	`reverse`, `sort`	copy the array first (example)

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({
          id: nextId++,
```

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

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

Instead, create a new array which contains the existing items and a new item at the end. There are multiple ways to do this, but the easiest one is to use the ... array spread syntax:

```
setArtists( // Replace the state
  [ // with a new array
    ...artists, // that contains all the old items
    { id: nextId++, name: name } // and one new item at the end
  ]
);
```

Now it works correctly:

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

Removing from an array

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

App.js

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

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

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

  return (
    <>
```

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

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

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.

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

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():

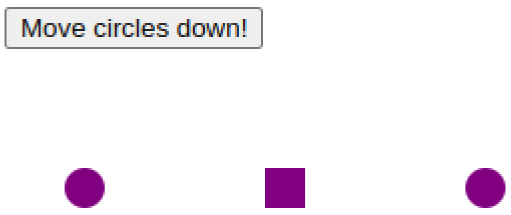
```
import { useState } from 'react';

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

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

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

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



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:

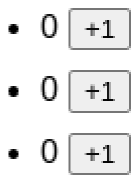
```
import { useState } from 'react';

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);
          }}>+1</button>
        </li>
      ))}
    </ul>
  );
}
```



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:

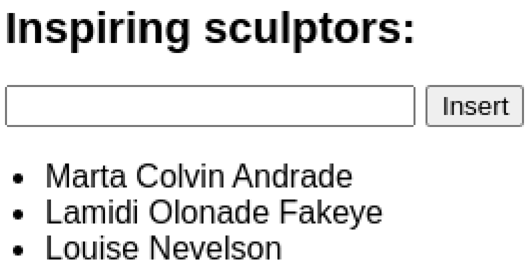
```
import { useState } from 'react';

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

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

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

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


Inspiring sculptors:

- Marta Colvin Andrade
- Lamidi Olonade Fakeye
- Louise Nevelson

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';

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

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

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

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

}

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

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

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

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

Updating objects inside arrays

Objects are not really located "inside" arrays. They might appear to be "inside" in code, but each object in an array is a separate value, to which the array "points". This is why you need to be careful when changing nested fields like list[0]. Another person's artwork list may point to the same element of the array!

When updating nested state, you need to create copies from the point where you want to update, and all the way up to the top level. Let's see how this works.

In this example, two separate artwork lists have the same initial state. They are supposed to be isolated, but because of a mutation, their state is accidentally shared, and checking a box in one list affects the other list:

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


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


export default function BucketList() {
```

```
const [myList, setMyList] = useState(initialList);
const [yourList, setYourList] = useState(
  initialList
);

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

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

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

```
      <ItemList
        artworks={yourList}
        onToggle={handleToggleYourList} />
    </>
  );
}

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

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 = [

 { id: 0, title: 'Big Bellies', seen: false },

 { id: 1, title: 'Lunar Landscape', seen: false },

 { id: 2, title: 'Terracotta Army', seen: true },

];

export default function BucketList() {

 const [myList, setMyList] = useState(initialList);

```
const [yourList, setYourList] = useState(
  initialList
);

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

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

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

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

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

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

Write concise update logic with Immer

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

- Generally, you shouldn't need to update state more than a couple of levels deep. If your state objects are very deep, you might want to restructure them differently so that they are flat.
- If you don't want to change your state structure, you might prefer to use Immer, which lets you write using the convenient but mutating syntax and takes care of producing the copies for you.

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

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

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

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

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

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

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

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.

```
{
  "dependencies": {
    "immer": "1.7.3",
    "react": "latest",
    "react-dom": "latest",
    "react-scripts": "latest",
    "use-immer": "0.5.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  },
  "devDependencies": {}
}
```