useState

`useState` is a React Hook that lets you add a state variable to your component.

```
const [state, setState] = useState(initialState);
```


- Reference
  - `useState(initialState)`
  - `set` functions, like `setSomething(nextState)`
- Usage
  - Adding state to a component
  - Updating state based on the previous state
  - Updating objects and arrays in state
  - Avoiding recreating the initial state
  - Resetting state with a key
  - Storing information from previous renders
- Troubleshooting
  - I've updated the state, but logging gives me the old value
  - I've updated the state, but the screen doesn't update
  - I'm getting an error: "Too many re-renders"
  - My initializer or updater function runs twice
  - I'm trying to set state to a function, but it gets called instead

Reference

useState(initialState)

Call `useState` at the top level of your component to declare a state variable.

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

function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

The convention is to name state variables like `[something, setSomething]` using array destructuring.

See more examples below.

Parameters

- `initialState` : The value you want the state to be initially. It can be a value of any type, but there is a special behavior for functions. This argument is ignored after the initial render.
  - If you pass a function as `initialState` , it will be treated as an *initializer function*. It should be pure, should take no arguments, and should return a value of any type. React will call your initializer function when initializing the component, and store its return value as the initial state. See an example below.

Returns

`useState` returns an array with exactly two values:

1. The current state. During the first render, it will match the `initialState` you have passed.
2. The `set` function that lets you update the state to a different value and trigger a re-render.

Caveats

- `useState` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.
- In Strict Mode, React will **call your initializer function twice** in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your initializer function is pure (as it should be), this should not affect the behavior. The result from one of the calls will be ignored.

`set` functions, like `setSomething(nextState)`

The `set` function returned by `useState` lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

```
const [name, setName] = useState('Edward');

function handleClick() {
  setName('Taylor');
  setAge(a => a + 1);
  // ...
```

Parameters

- `nextState` : The value that you want the state to be. It can be a value of any type, but there is a special behavior for functions.

- If you pass a function as `nextState`, it will be treated as an *updater function*. It must be pure, should take the pending state as its only argument, and should return the next state. React will put your updater function in a queue and re-render your component. During the next render, React will calculate the next state by applying all of the queued updaters to the previous state. See an example below.

Returns

`set` functions do not have a return value.

Caveats

- The `set` function **only updates the state variable for the *next* render**. If you read the state variable after calling the `set` function, you will still get the old value that was on the screen before your call.

- If the new value you provide is identical to the current `state`, as determined by an `Object.is` comparison, React will **skip re-rendering the component and its children.** This is an optimization. Although in some cases React may still need to call your component before skipping the children, it shouldn't affect your code.

- React batches state updates. It updates the screen **after all the event handlers have run** and have called their `set` functions. This prevents multiple re-renders during a single event. In the rare case that you need to force React to update the screen earlier, for example to access the DOM, you can use `flushSync`.

- Calling the `set` function *during rendering* is only allowed from within the currently rendering component. React will discard its output and immediately attempt to render it again with the new state. This pattern is rarely needed, but you can use it to **store information from the previous renders**. See an example below.

- In Strict Mode, React will **call your updater function twice** in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your updater function is pure (as it should be), this

should not affect the behavior. The result from one of the calls will be ignored.

Usage

Adding state to a component

Call `useState` at the top level of your component to declare one or more state variables.

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

function MyComponent() {
  const [age, setAge] = useState(42);
  const [name, setName] = useState('Taylor');
  // ...
```

The convention is to name state variables like `[something, setSomething]` using array destructuring.

`useState` returns an array with exactly two items:

1. The current state of this state variable, initially set to the initial state you provided.
2. The `set` function that lets you change it to any other value in response to interaction.

To update what's on the screen, call the `set` function with some next state:

```
function handleClick() {
  setName('Robin');
}
```

React will store the next state, render your component again with the new values, and update the UI.

Pitfall

Calling the `set` function **does not** change the current state in the already executing code:

```
function handleClick() {
  setName('Robin');
  console.log(name); // Still "Taylor"!
}
```

It only affects what `useState` will return starting from the *next* render.

Basic useState examples

1. Counter (number) 2. Text field (string) 3. Checkbox (boolean) 4.

Example 1 of 4:

Counter (number)

In this example, the `count` state variable holds a number. Clicking the button increments it.

App.js Download Reset

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

export default function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      You pressed me {count} times
```

Next Example

Updating state based on the previous state

Suppose the `age` is `42`. This handler calls `setAge(age + 1)` three times:

```
function handleClick() {
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
}
```

However, after one click, `age` will only be `43` rather than `45`! This is because calling the `set` function does not update the `age` state variable in the already running code. So each `setAge(age + 1)` call becomes `setAge(43)`.

To solve this problem, **you may pass an *updater function*** to `setAge` instead of the next state:

```
function handleClick() {
  setAge(a => a + 1); // setAge(42 => 43)
  setAge(a => a + 1); // setAge(43 => 44)
  setAge(a => a + 1); // setAge(44 => 45)
}
```

Here, `a => a + 1` is your updater function. It takes the pending state and calculates the next state from it.

React puts your updater functions in a queue. Then, during the next render, it will call them in the same order:

1. `a => a + 1` will receive `42` as the pending state and return `43` as the next state.
2. `a => a + 1` will receive `43` as the pending state and return `44` as the next state.
3. `a => a + 1` will receive `44` as the pending state and return `45` as the next state.

There are no other queued updates, so React will store `45` as the current state in the end.

By convention, it's common to name the pending state argument for the first letter of the state variable name, like `a` for `age`. However, you may also call it like `prevAge` or something else that you find clearer.

React may call your updaters twice in development to verify that they are pure.

DEEP DIVE

Is using an updater always preferred?

Show Details

The difference between passing an updater and passing the next state directly

1. Passing the updater function
2. Passing the next state directly

Example 1 of 2:

Passing the updater function

This example passes the updater function, so the "+3" button works.

App.js Download Reset

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

export default function Counter() {
  const [age, setAge] = useState(42);
```

```
function increment() {
  setAge(a => a + 1);
}

return (
  <>
    <h1>Your age: {age}</h1>
```

Show more

Next Example

Updating objects and arrays in state

You can put objects and arrays into state. In React, state is considered read-only, so **you should *replace* it rather than *mutate* your existing objects**. For example, if you have a `form` object in state, don't mutate it:

```
// 🚩 Don't mutate an object in state like this:
form.firstName = 'Taylor';
```

Instead, replace the whole object by creating a new one:

```
// ✅ Replace state with a new object
setForm({
  ...form,
  firstName: 'Taylor'
});
```

Read updating objects in state and updating arrays in state to learn more.

Examples of objects and arrays in state

1. Form (object) 2. Form (nested object) 3. List (array) 4. Writing c

Example 1 of 4:

Form (object)

In this example, the `form` state variable holds an object. Each input has a change handler that calls `setForm` with the next state of the entire form. The `{ ...form }` spread syntax ensures that the state object is replaced rather than mutated.

App.js Download Reset

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

export default function Form() {
  const [form, setForm] = useState({
```

```
    firstName: 'Barbara',
    lastName: 'Hepworth',
    email: 'bhepworth@sculpture.com',
  });

  return (
    <>
      <label>
```

Show more

Next Example

Avoiding recreating the initial state

React saves the initial state once and ignores it on the next renders.

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos());
```

```
  // ...
```

Although the result of `createInitialTodos()` is only used for the initial render, you're still calling this function on every render. This can be wasteful if it's creating large arrays or performing expensive calculations.

To solve this, you may **pass it as an *initializer* function** to `useState` instead:

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos);
  // ...
```

Notice that you're passing `createInitialTodos`, which is the *function itself*, and not `createInitialTodos()`, which is the result of calling it. If you pass a function to `useState`, React will only call it during initialization.

React may call your initializers twice in development to verify that they are pure.

The difference between passing an initializer and passing the initial state directly

1. Passing the initializer function
2. Passing the initial state directly

Example 1 of 2:

Passing the initializer function

This example passes the initializer function, so the `createInitialTodos` function only runs during initialization. It does not run when component re-renders, such as when you type into the input.

App.js Download Reset

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

function createInitialTodos() {
  const initialTodos = [];
  for (let i = 0; i < 50; i++) {
    initialTodos.push({
      id: i,
      text: 'Item ' + (i + 1)
    });
  }
  return initialTodos;
}
```

Show more

Next Example

Resetting state with a key

You'll often encounter the `key` attribute when rendering lists. However, it also serves another purpose.

You can **reset a component's state by passing a different `key` to a component.** In this example, the Reset button changes the `version` state variable, which we pass as a `key` to the `Form`. When the `key` changes, React re-creates the `Form` component (and all of its children) from scratch, so its state gets reset.

Read preserving and resetting state to learn more.

App.js Download Reset

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

export default function App() {
  const [version, setVersion] = useState(0);

  function handleReset() {
    setVersion(version + 1);
  }

  return (
    <>
      <button onClick={handleReset}>Reset</button>
```

Show more

Storing information from previous renders

Usually, you will update state in event handlers. However, in rare cases you might want to adjust state in response to rendering — for example, you might want to change a state variable when a prop changes.

In most cases, you don't need this:

- **If the value you need can be computed entirely from the current props or other state, remove that redundant state altogether.** If you're worried about recomputing too often, the `useMemo` Hook can help.
- If you want to reset the entire component tree's state, pass a different `key` to your component.
- If you can, update all the relevant state in the event handlers.

In the rare case that none of these apply, there is a pattern you can use to update state based on the values that have been rendered so far, by calling a `set` function while your component is rendering.

Here's an example. This `CountLabel` component displays the `count` prop passed to it:

```
export default function CountLabel({ count }) {
  return <h1>{count}</h1>
}
```

Say you want to show whether the counter has *increased or decreased* since the last change. The `count` prop doesn't tell you this — you need to keep track of its previous value. Add the `prevCount` state variable to track it. Add another

state variable called `trend` to hold whether the count has increased or decreased. Compare `prevCount` with `count`, and if they're not equal, update both `prevCount` and `trend`. Now you can show both the current count prop and *how it has changed since the last render*.

App.js CountLabel.js Reset

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

export default function CountLabel({ count }) {
  const [prevCount, setPrevCount] = useState(count);
  const [trend, setTrend] = useState(null);
  if (prevCount !== count) {
    setPrevCount(count);
    setTrend(count > prevCount ? 'increasing' : 'decreasing');
  }
  return (
    <>
      <h1>{count}</h1>
```

Show more

Note that if you call a `set` function while rendering, it must be inside a condition like `prevCount !== count`, and there must be a call like `setPrevCount(count)` inside of the condition. Otherwise, your component would re-render in a loop until it crashes. Also, you can only update the state of the *currently rendering* component like this. Calling the `set` function of *another* component during rendering is an error. Finally, your `set` call should still update state without mutation — this doesn't mean you can break other rules of pure functions.

This pattern can be hard to understand and is usually best avoided. However, it's better than updating state in an effect. When you call the `set` function during render, React will re-render that component immediately after your component exits with a `return` statement, and before rendering the children. This way, children don't need to render twice. The rest of your component function will still execute (and the result will be thrown away). If your condition is below all the Hook calls, you may add an early `return;` to restart rendering earlier.

Troubleshooting

I've updated the state, but logging gives me the old value

Calling the `set` function **does not change state in the running code**:

```
function handleClick() {
  console.log(count);  // 0

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

  setTimeout(() => {
```

```
    console.log(count); // Also 0!
  }, 5000);
}
```

This is because states behaves like a snapshot. Updating state requests another render with the new state value, but does not affect the `count` JavaScript variable in your already-running event handler.

If you need to use the next state, you can save it in a variable before passing it to the `set` function:

```
const nextCount = count + 1;
setCount(nextCount);

console.log(count);     // 0
console.log(nextCount); // 1
```

I've updated the state, but the screen doesn't update

React will **ignore your update if the next state is equal to the previous state,** as determined by an `Object.is` comparison. This usually happens when you change an object or an array in state directly:

```
obj.x = 10;  // 🚩 Wrong: mutating existing object
setObj(obj); // 🚩 Doesn't do anything
```

You mutated an existing `obj` object and passed it back to `setObj`, so React ignored the update. To fix this, you need to ensure that you're always *replacing* objects and arrays in state instead of *mutating* them:

```
// ✅ Correct: creating a new object
setObj({
  ...obj,
  x: 10
});
```

I'm getting an error: "Too many re-renders"

You might get an error that says: `Too many re-renders. React limits the number of renders to prevent an infinite loop.` Typically, this means that you're unconditionally setting state *during render*, so your component enters a loop: render, set state (which causes a render), render, set state (which causes a render), and so on. Very often, this is caused by a mistake in specifying an event handler:

```
// 🚩 Wrong: calls the handler during render
return <button onClick={handleClick()}>Click me</button>

// ✅ Correct: passes down the event handler
return <button onClick={handleClick}>Click me</button>

// ✅ Correct: passes down an inline function
return <button onClick={(e) => handleClick(e)}>Click me</button>
```

If you can't find the cause of this error, click on the arrow next to the error in the console and look through the JavaScript stack to find the specific `set` function call responsible for the error.

My initializer or updater function runs twice

In Strict Mode, React will call some of your functions twice instead of once:

```
function TodoList() {
  // This component function will run twice for every render.

  const [todos, setTodos] = useState(() => {
    // This initializer function will run twice during initialization.
    return createTodos();
  });

  function handleClick() {
    setTodos(prevTodos => {
      // This updater function will run twice for every click.
      return [...prevTodos, createTodo()];
    });
  }
  // ...
```

This is expected and shouldn't break your code.

This **development-only** behavior helps you keep components pure. React uses the result of one of the calls, and ignores the result of the other call. As long as your component, initializer, and updater functions are pure, this shouldn't affect your logic. However, if they are accidentally impure, this helps you notice the mistakes.

For example, this impure updater function mutates an array in state:

```
setTodos(prevTodos => {
  // 🚩 Mistake: mutating state
  prevTodos.push(createTodo());
});
```

Because React calls your updater function twice, you'll see the todo was added twice, so you'll know that there is a mistake. In this example, you can fix the mistake by replacing the array instead of mutating it:

```
setTodos(prevTodos => {
  // ✅ Correct: replacing with new state
  return [...prevTodos, createTodo()];
});
```

Now that this updater function is pure, calling it an extra time doesn't make a difference in behavior. This is why React calling it twice helps you find mistakes. **Only component, initializer, and updater functions need to be pure.** Event handlers don't need to be pure, so React will never call your event handlers twice.

Read keeping components pure to learn more.

I'm trying to set state to a function, but it gets called instead

You can't put a function into state like this:

```
const [fn, setFn] = useState(someFunction);

function handleClick() {
  setFn(someOtherFunction);
}
```

Because you're passing a function, React assumes that `someFunction` is an initializer function, and that `someOtherFunction` is an updater function, so it tries to call them and store the result. To actually *store* a function, you have to put `() =>` before them in both cases. Then React will store the functions you pass.

```
const [fn, setFn] = useState(() => someFunction);

function handleClick() {
  setFn(() => someOtherFunction);
}
```

PREVIOUS
useRef

NEXT
useSyncExternalStore

How do you like these docs?

Take our survey!

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

API Reference

React APIs

React DOM APIs

Escape Hatches

Community

Code of Conduct

Meet the Team

Docs Contributors

Acknowledgements

More

Blog

React Native

Privacy

Terms