

React State Management

Course: Java

S1



React State Management

Most software bugs are the result of bad state. More accurately, they're the result of bad assumptions about state. We assume that some bit of data could never happen or that some sequence of state changes is impossible.

Then we write code with those assumptions baked in.

Even if our code is perfect, troubles can arise.

If our code isn't the only code managing state, another process or step can modify state into something that our code didn't anticipate.

That's a bug.



React State Management

React state management protects us from a lot of that.

Science hasn't figured out how to create truly bug-free code, but React state management gets us a little closer.



Learning Outcomes

By the end of this lesson, you will be able to:

01

Describe one of the benefits of React's state management.

02

Use the `useState` hook inside a component to track single values.

03

Use the `useState` hook inside a component to track objects and arrays.

04


Add an event handler to a component.

05

Create components that follow the hook rules.

State and Hooks

State



State is the data that is currently in memory—including numbers, strings, objects, etc.

At rest, state doesn't change.

State changes through events—such as UI interactions, application starts and stops, and background requests.

React's State Flow

At its core, React is conceptually simple.



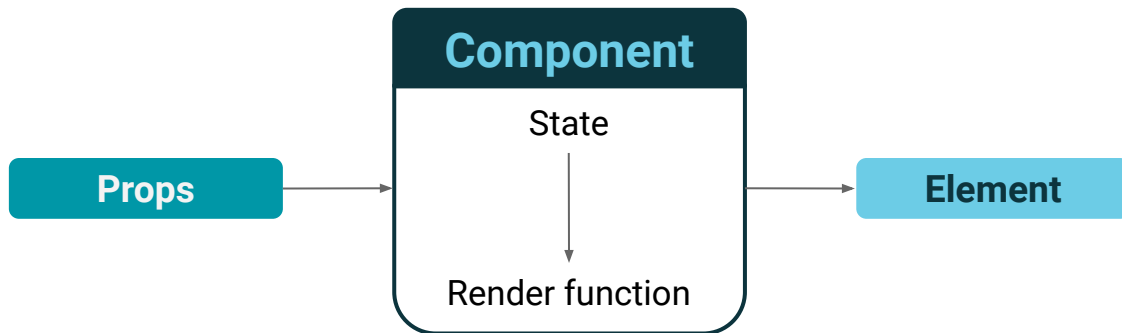
Component instances track their own private state.



Components may cautiously share their state with a child via props.



Any change to state (and by proxy, props) triggers a re-render.



React's State Flow

This happens over and over in the following process:

01

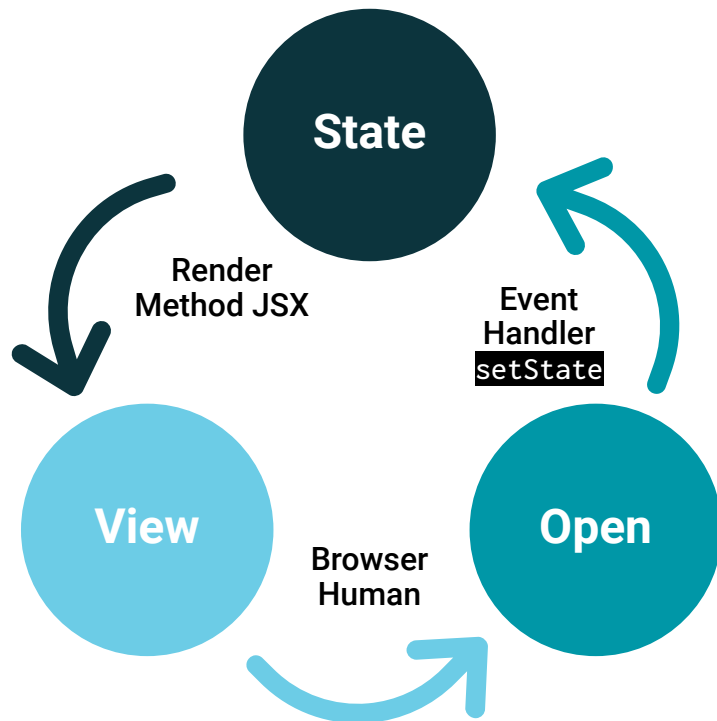
State drives UI rendering.
React reacts to state.

02

Events modify state.

03

Return to Step 1.



But, the devil is in the details.

Hooks



Functional components use **hooks** for state management.



A hook is a function.



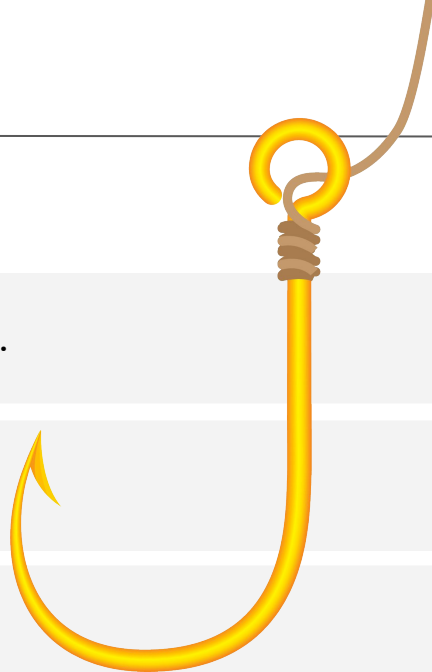
Hook names always start with the **use** prefix.



Hooks allow us to tap into larger React plumbing. We “hook” into a React process.



The **useState** hook manages state.



The useState Hook



Accepts a single parameter: an initial value. This can be any value.



Returns an array with two elements.



The first is the current value.



The second is a setter function. It's the only way to change the state's value.

```
// current value
//      |   function to set value
//      |           |           initial value (0
words)
//      |           |           |
//      v           v           v
const [wordCount, setWordCount] = useState(0);
const [name, setName] = useState("Ringo");
const [todos, setTodos] = useState([]);

// alternatively
const nameElements = useState("Ringo");
const name = nameElements[0];
const setName = nameElements[1];
```

Using useState

01

Import the `useState` function.

02

Set initial values; receive the current value and a setter.

03

Read state anywhere after the hook.

04

Write (mutate) state anywhere after the hook.

```
import { useState } from 'react'; // 1

function Clicker() {

  const [clicks, setClicks] = useState(0); // 2

  return (
    /*4*/
    <button onClick={() => setClicks(clicks + 1)}>
      Clicks: {clicks} /*3*/
    </button>
  );
}

export default Clicker;
```

Track Multiple Values



We can track as many values as required.



Code that depends on state should be written after state setup.

```
function Clicker() {
  const [clicks, setClicks] = useState(0);
  const [lastUpdated, setLastUpdated] = useState("No Updates");

  const update = () => {
    setClicks(clicks + 1);
    setLastUpdated(new Date().toLocaleString());
  };

  return (
    <div>
      <button onClick={update}>
        Clicks: {clicks}
      </button>
      <p>
        Last Updated: {lastUpdated}
      </p>
    </div>
  );
}
```

Track Arrays and Objects



Sometimes, state is best tracked at the object or array level, rather than as individual values.



Never change/mutate an existing object.



Instead, make a copy. Then change the copy, and replace the original with the updated copy.



The `...spread` syntax saves typing.

```
function Values() {  
  
  const [values, setValues] = useState([]);  
  
  const handleEnter = (evt) => {  
    if (evt.key === "Enter") {  
      setValues([...values, evt.target.value]);  
    }  
  };  
  
  return (  
    <div>  
      <p>Enter a value and press <kbd>Enter</kbd>.</p>  
      <input type="text" onKeyUp={handleEnter} />  
      <ul>  
        {values.map(n => <li>{n}</li>)}  
      </ul>  
    </div>  
  );  
}
```

Hook Rules

We must comply with the three rules of hooks:

01

Hooks must execute.

Hooks must execute in the same order.

02

Only call hooks from top-level components.

Never call hooks from within loops, conditionals, or nested functions.

03

Hooks may only be called from React components.

Never call a hook from a regular JavaScript function.

This makes it easier for the developer to find all stateful logic.



If we vary from these rules, our components will not work properly. These rules apply to all hooks, not just to `useState`.

```
// Good
const [wordCount, setWordCount] = useState(0);
const [name, setName] = useState("Ringo");

// BAD, conditional
let toDos, setToDos;
if (props.value > 0) {
  [toDos, setToDos] = useState([]);
}

// BAD, varying order
let a, b, setA, setB;
if (Math.random() < 0.5) {
  [a, setA] = useState(0);
  [b, setB] = useState(0);
} else {
  [b, setB] = useState(0);
  [a, setA] = useState(0);
}

// BAD, conditional
const stateArray = [];
for (let i = 0; i < props.array.length; i++) {
  stateArray.push(useState(i));
}
```




A close-up photograph of a computer keyboard. The central focus is a large, white, rectangular key with rounded corners. On this key, there is a dark blue icon of a coffee cup with three wavy lines above it representing steam. Below the icon, the word "Break" is printed in a dark blue, serif font. The key is slightly raised from the keyboard's base. Surrounding this key are other keys, including one with a double quote symbol to the left and one with a dash/slash symbol to the right. The keyboard has a light-colored, possibly wood-grain, textured surface.

Break

State Activities



Time to Code

More React State

Suggested Time:

20 Minutes



Activity: More React State

Suggested Time:

25 Minutes



Recap



What is state?



How is state modified?



What problems do hooks solve?



What specific problem does the `useState` hook solve?



What is the argument passed to `useState`? What does it return?



How do event handlers differ in React compared to HTML and JavaScript?

Learning Outcomes

By the end of this lesson, you will be able to:

01

Describe one of the benefits of React's state management.

02

Use the `useState` hook inside a component to track single values.

03

Use the `useState` hook inside a component to track objects and arrays.

04

Add an event handler to a component.

05

Create components that follow the hook rules.

*The
End*