React: From Static Blueprint to Interactive App

A visual guide to the core concepts you'll use every day.

NotebookLM

Your Journey Through React's Core

This guide covers 80% of the React concepts you'll use daily. By the end, you will understand:

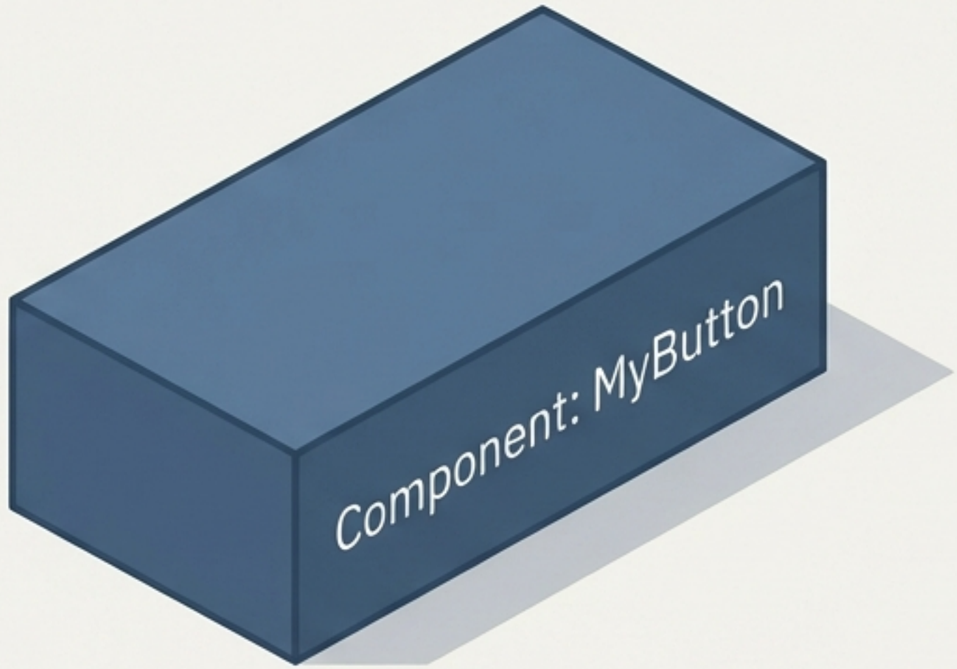
- How to create and nest fundamental UI blocks, called **components.**
- How to add structure with **markup (JSX)** and visual polish with **styles.**
- How to make your components dynamic by **displaying data.**
- How to use logic to **render conditions** and **lists.**
- How to make your UI respond to user input with **events** and **state.**
- How to enable components to communicate by **sharing data.**

NotebookLM

The Building Block: Understanding Components

React apps are made of **components**. A component is a self-contained piece of the UI with its own logic and appearance—from a single button to an entire page.

```
// React components are JavaScript
functions that return markup.
function MyButton() {
  return (
    <button>I'm a button</button>
  );
}
```

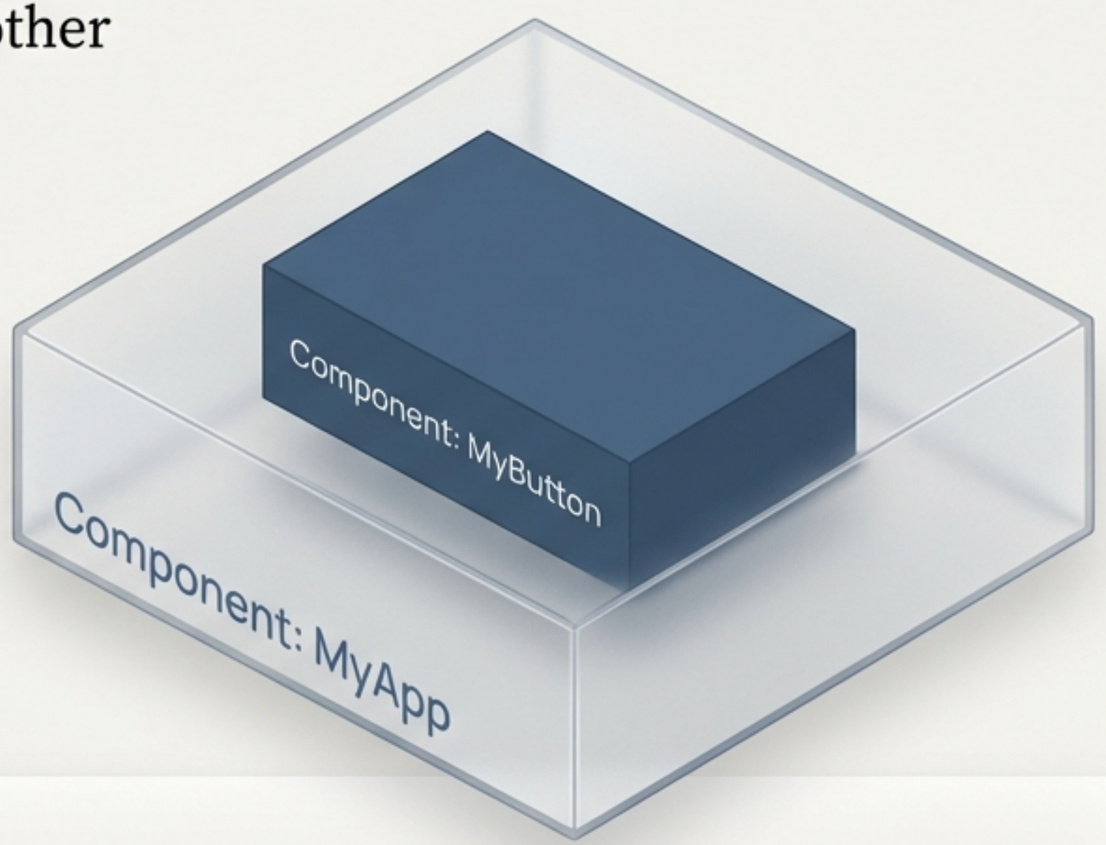


NotebookLM

Assembling the UI: Nesting Components

Once declared, you can nest components inside other components to build complex UIs.

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



A Critical Distinction

Notice `<MyButton />` starts with a capital letter. That's how React identifies components.

- `MyButton` → React Component
- `div`, `h1` → Standard HTML tag (must be lowercase)

NotebookLM

Writing Markup with JSX

JSX is a syntax extension for JavaScript that lets you write HTML-like markup inside your components. While optional, it's a standard in most React projects.

Key Rules

1. **"Close all tags"**: Self-closing tags must be explicitly closed. E.g., `<br />`.
2. **"Return a single root element"**: Your component must return one parent tag. Wrap multiple elements in a `<div>` or an empty `<>..</>` fragment.

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

Empty fragment <>...</> acts as a root element

Self-closing tag

NotebookLM

Adding Styles to Components

In React, you apply CSS classes using the `className` attribute.
It works just like the standard HTML `class` attribute.

In your component file

```
<img className="avatar"
  src={...}
/>
```

In your CSS file

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

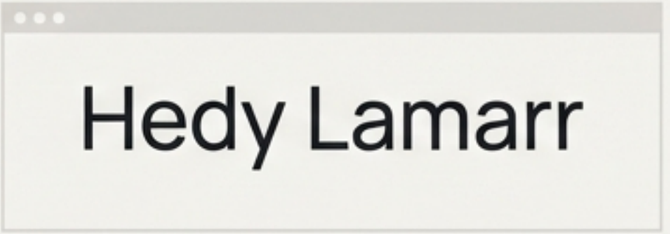
React doesn't prescribe *how* you add CSS files. You can use a `<link>` tag in your HTML or integrate with your build tool.

NotebookLM

Displaying Dynamic Data

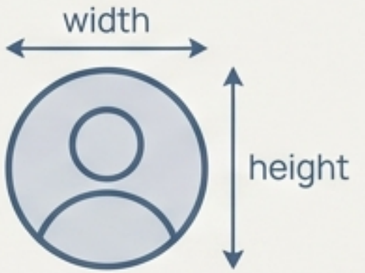
JSX lets you put markup into JavaScript. Curly braces {} let you 'escape back' into JavaScript to embed variables and expressions directly in your UI.

```
const user = { name: 'Hedy Lamarr' };

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




```
const user = { imageUrl: '...' };

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

```
// style={{}} is a regular JS object inside JSX braces.
<img style={ width: user.imageSize, height: user.imageSize } />
```



NotebookLM

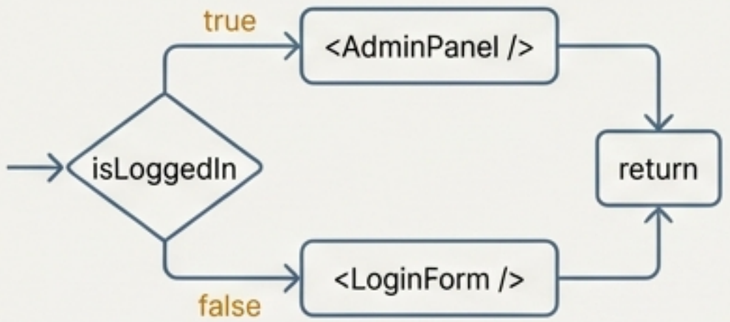
Conditional Rendering: Making Decisions in JSX

React doesn't have special syntax for conditions. You use standard JavaScript logic to decide what to render.

1. if...else (The Classic)

Good for complex logic outside of JSX.

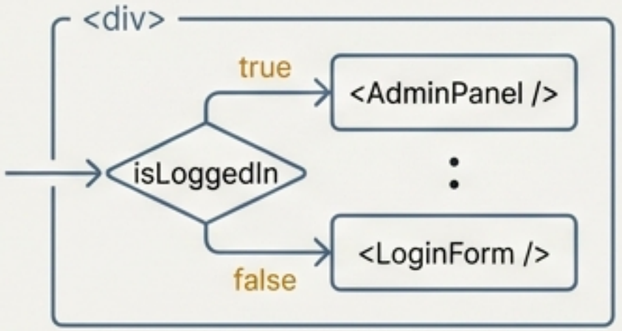
```
let content;
if (isLoggedIn) {
  content = <AdminPanel />;
} else {
  content = <LoginForm />;
}
return <div>{content}</div>;
```



2. Ternary Operator ? : (The Inline Choice)

Perfect for if/else logic directly inside JSX.

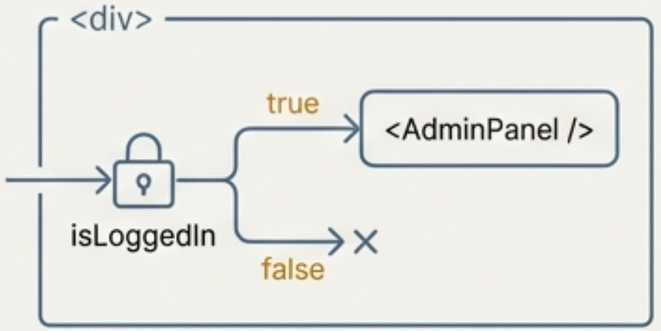
```
<div>
  {isLoggedIn ? <AdminPanel /> :
  <LoginForm />}
</div>
```



3. Logical && (The "If True" Shortcut)

Use when you only want to render something if a condition is true, with no else case.

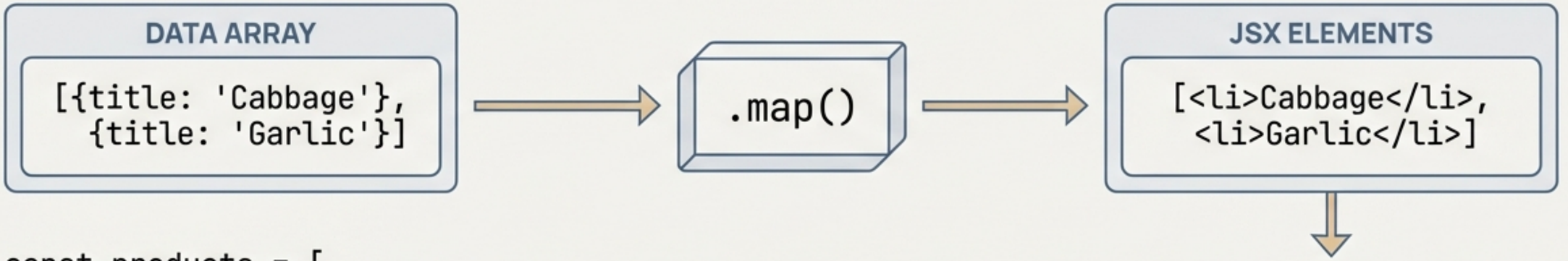
```
<div>
  {isLoggedIn && <AdminPanel />}
</div>
```



NotebookLM

Rendering Lists from Arrays

To render a list of components, you'll rely on standard JavaScript array methods like `.map()` to transform your data into an array of JSX elements.




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

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

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

The `key` Prop is Essential

The `key` is a unique string or number that identifies an item among its siblings. React uses keys to understand what happened if you insert, delete, or reorder items. It should come from your data, like a database ID.

NotebookLM

Responding to Events

You can make your components interactive by declaring event handler functions that respond to user actions like clicks, hover, or form inputs.

```
function MyButton() {
  function handleClick() {
    alert('You clicked me!');
  }

  return (
    <button onClick={handleClick}>
      Click me
    </button>
  );
}
```

Pass, Don't Call!

Notice it's `onClick={handleClick}`, not `onClick={handleClick()}`. You pass the function itself as a prop. React will call it for you when the user clicks.

NotebookLM

Giving Components a Memory with State

To make a component “remember” information that changes over time (like a counter), you add state to it. The `useState` Hook is the primary way to do this.

The `useState` Hook

1. Import it:
   ```
   import { useState } from 'react';
   ```
2. Declare a state variable:
   ```
   function MyButton() {
     const [count, setCount] = useState(0);
     // ...
   }
   ```

Anatomy of `useState`

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

`count`: The current state value. The first time, it will be the initial value you provided (`0`).

`setCount`: The “setter” function. You call this to update the state value and trigger a re-render.

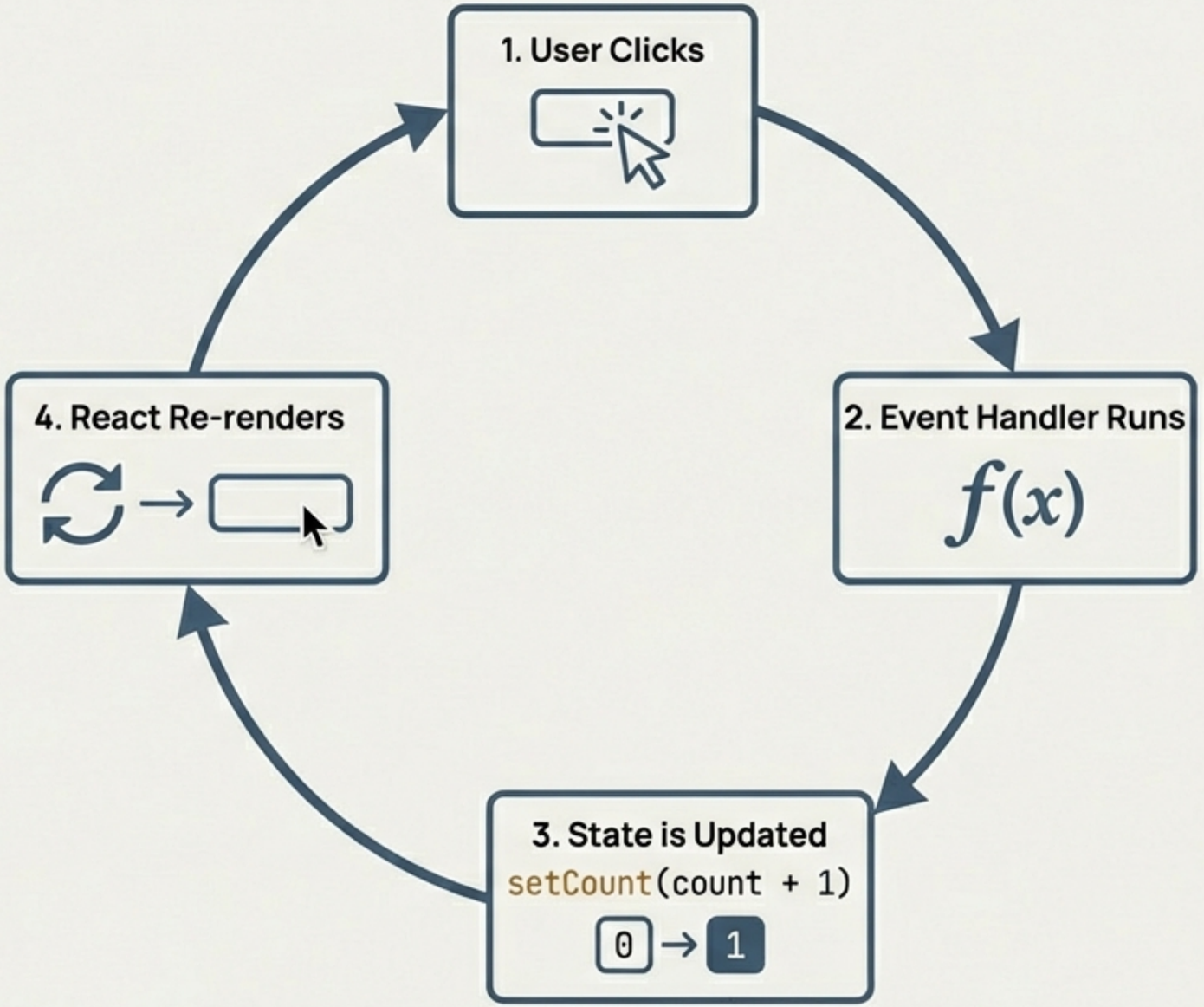
`useState(0)`: The Hook call. The argument (0`) is the initial state.

NotebookLM

The Update Cycle: How State Changes the Screen

When you call the state setter
When you call the state setter function, React automatically re-renders the component with the new state value.

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

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

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

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

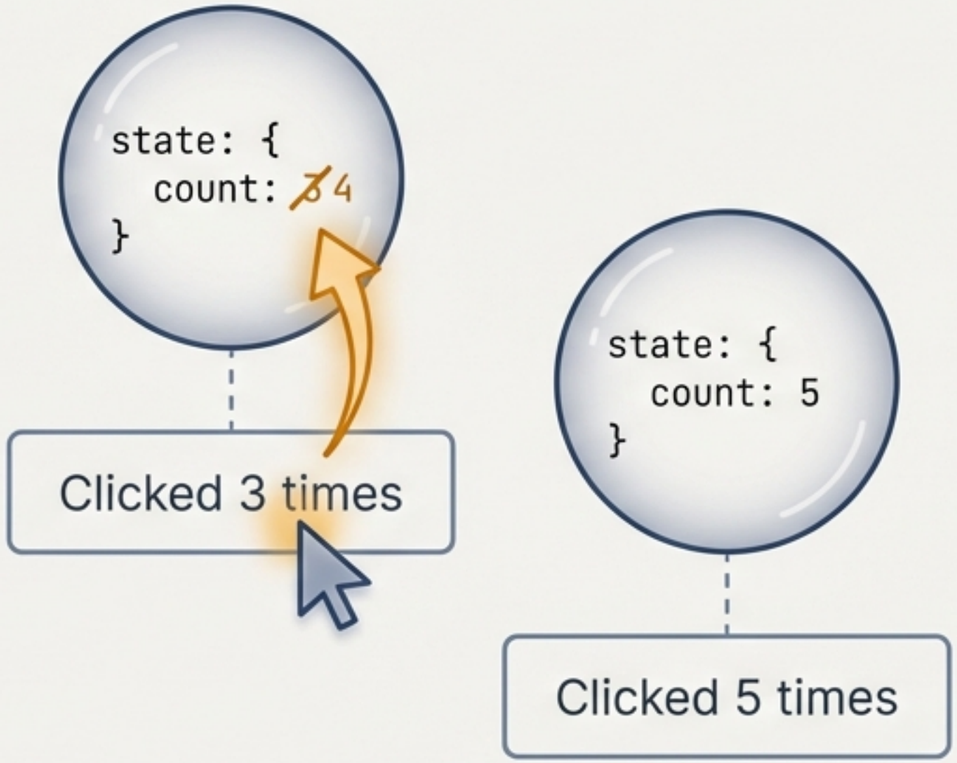


NotebookLM

State is Isolated and Independent

If you render the same component multiple times, each instance gets its own, completely separate state. One component's state does not affect another's.

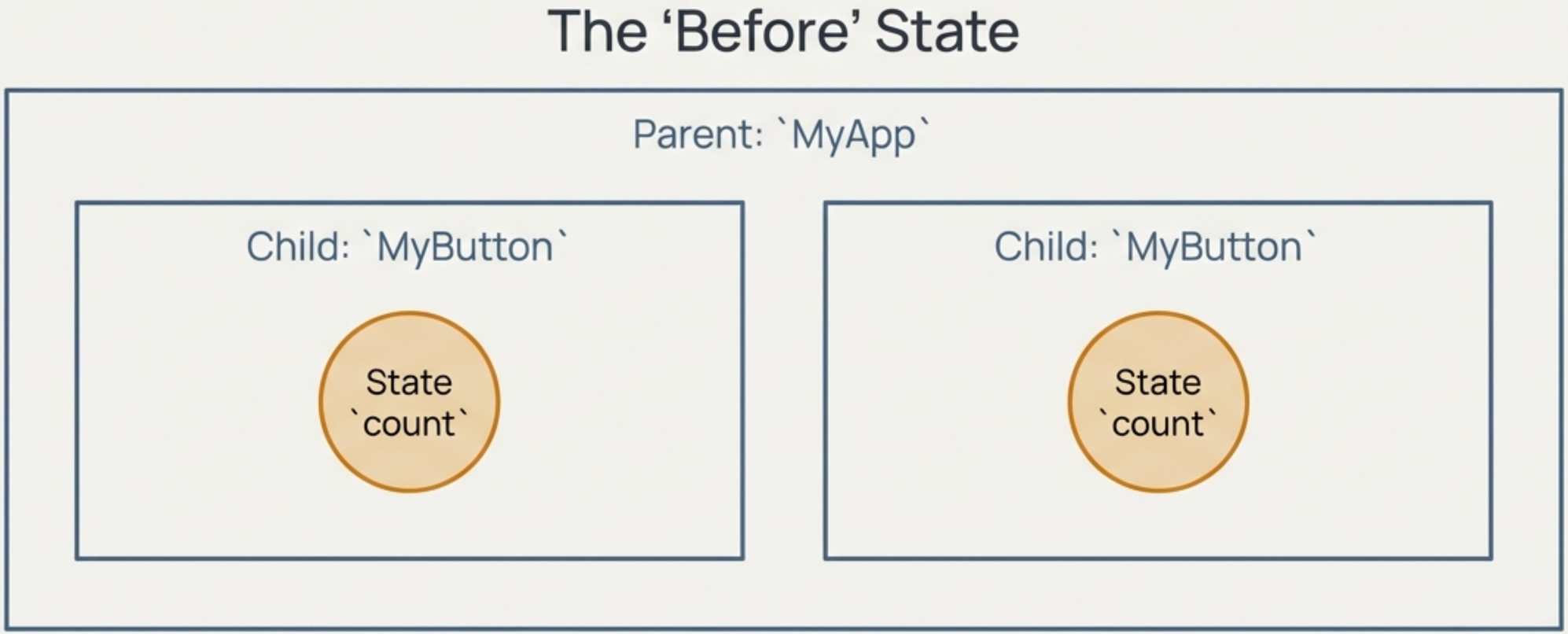
```
export default function MyApp() {
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

// (MyButton component code from previous slide)
```



NotebookLM

The Communication Challenge: Sharing Data

Often, you need multiple components to share the same state and update together. How do you synchronize the `count` for both buttons?

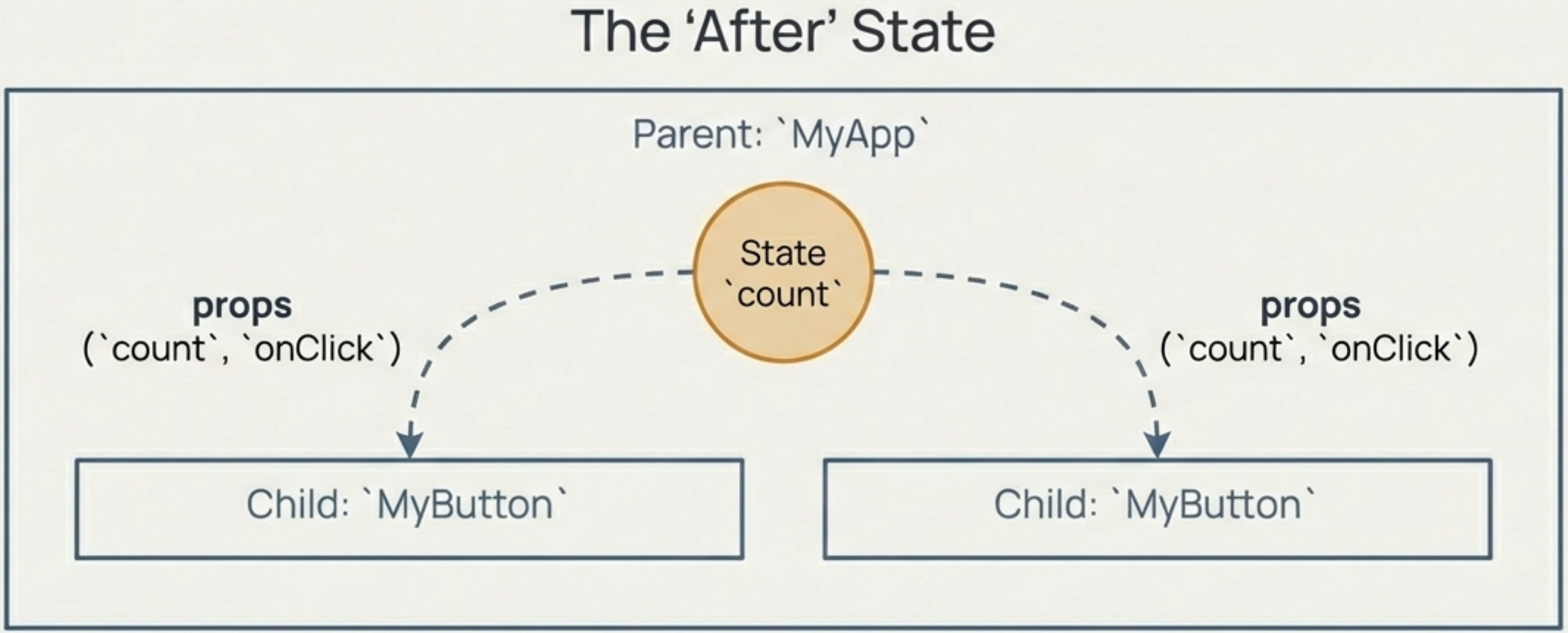


Problem: State is trapped in each child. They can’t see each other’s state.

NotebookLM

The Solution: ‘Lifting State Up’

To share state, move it from the individual children ‘upwards’ to the closest common parent component. Then, pass that state back down to the children as **props**.



Solution: State lives in the parent and is passed down as props.
Now there is a single source of truth.

NotebookLM

Lifting State Up: The Code in 3 Steps

Step 1: Move State Up

Cut the `useState` line and the `handleClick` function from the child (MyButton) and paste them into the parent (MyApp).

```
// In MyApp.js
const [count, setCount] = useState(0);

function handleClick() { ... }
```

Step 2: Pass State Down as Props

In the parent's JSX, pass the state value and the event handler down to each child component as props.

```
// In MyApp.js
<MyButton count={count}
  onClick={handleClick} />
<MyButton count={count}
  onClick={handleClick} />
```

Step 3: Read Props in the Child

Modify the child component to receive and use the props passed from its parent.

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

NotebookLM

The Full Journey, Recapped

Act I: The Blueprint (Static UI)

We built a static structure using **Components**, the fundamental building blocks.

We used **JSX** to write markup and **`className`** to add styles.

Act II: Bringing it to Life (Dynamic Data)

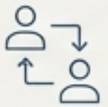
We used **`{}`** to inject data and logic into our markup.

We rendered UI conditionally and created **lists** from data arrays.

Act III: The Spark of Interaction (State & Events)

We responded to user input with **Event Handlers.**

We gave components memory with **`useState`**.



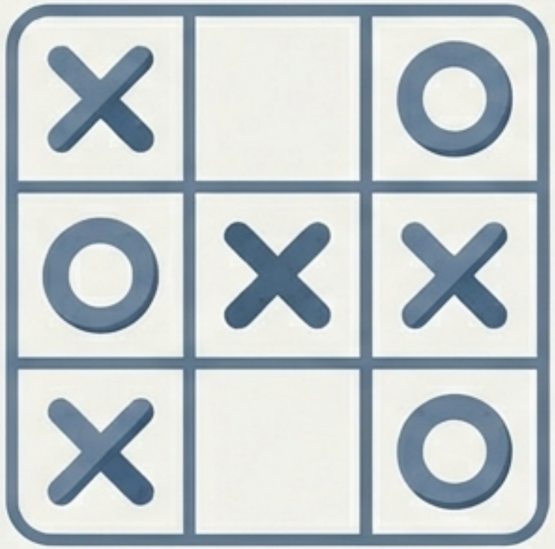
We shared data by **Lifting State Up** and passing **Props**.

NotebookLM

Your Next Step

You now have a solid grasp of the core concepts of React.
The best way to solidify this knowledge is to build something.

Put your skills into practice with the official React Tutorial.
You'll build your first mini-app: an interactive Tic-Tac-Toe game.



Start the Tutorial: Tic-Tac-Toe

NotebookLM