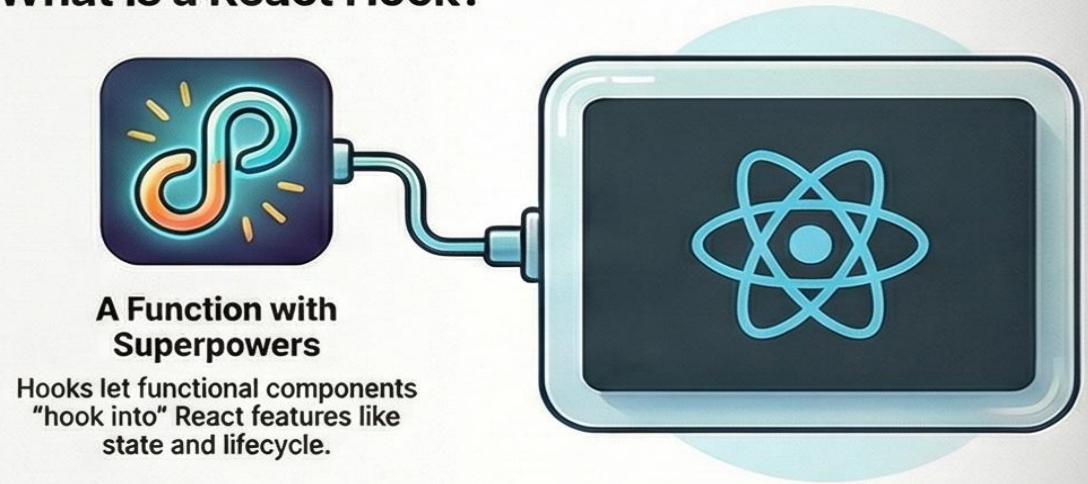
A Quick Guide to React Hooks

React Hooks are special functions that allow you to use state and other React features in functional components, eliminating the need for class components and resulting in simpler, more readable, and reusable code.

What is a React Hook?

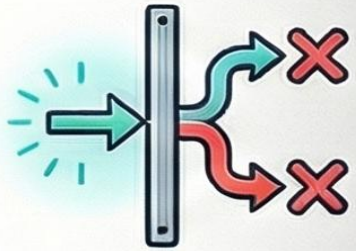


A Function with Superpowers

Hooks let functional components "hook into" React features like state and lifecycle.

No More Class Components

Use state, side effects, and more without writing a single class.



The Rules of Hooks

Only call Hooks at the top level of your component, not in loops or conditions.

The Most Common Hooks

useState: Manage Component State

Creates a state variable that triggers a UI re-render when its value changes.

useEffect: Handle Side Effects

Runs after rendering to perform actions like API calls, timers, or DOM manipulations.

useRef: Reference Values Without Re-renders

Access DOM elements directly or store mutable data that doesn't trigger a UI update.

useContext: Share Data Globally

Avoids passing props down through many nested components, known as "prop drilling".

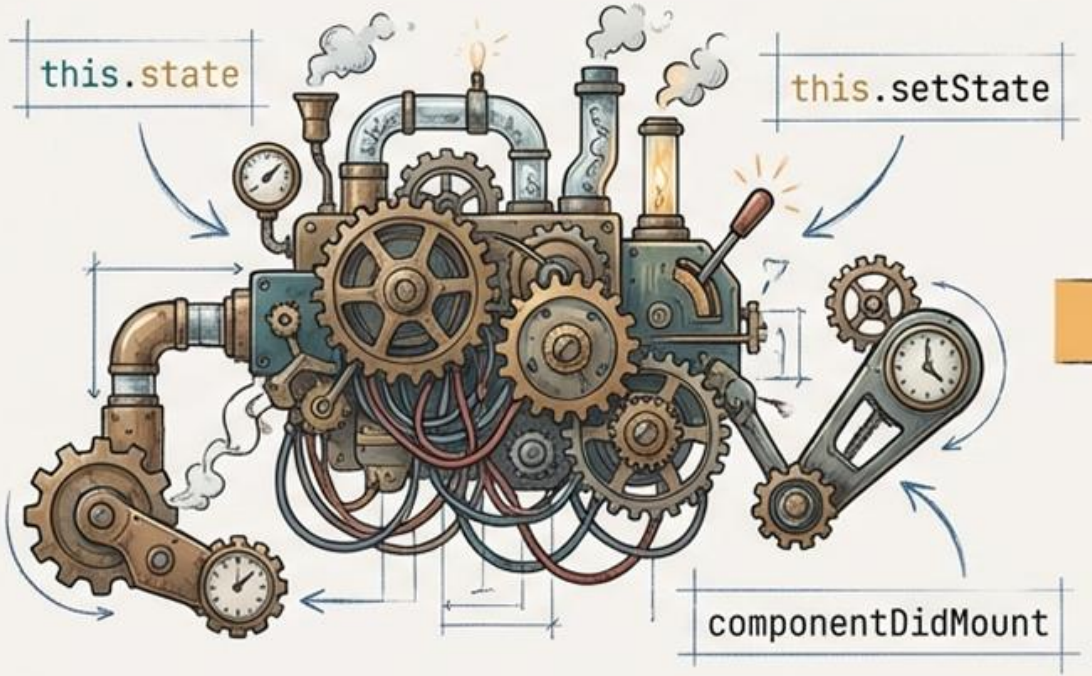
useMemo & useCallback: Optimize Performance

Cache expensive calculated values (useMomo) or function definitions (useCallback) to prevent unnecessary work.

From Classes to Functions: The Rise of React Hooks

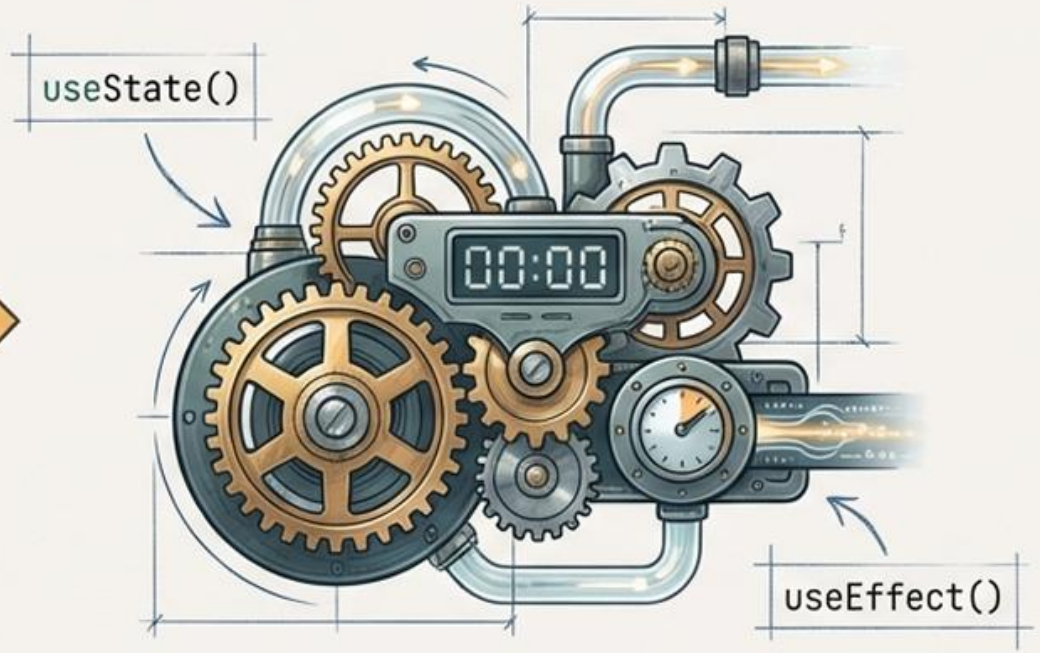
Hooks are special functions that let you “hook into” React state and lifecycle features from function components.

Class Components



- Required verbose class syntax.
- Logic was often split across different lifecycle methods, making components hard to follow.
- Simple functional components couldn't manage state or lifecycle.

Functional Components with Hooks



- Use state and other React features without writing a class.
- Code is simpler, more readable, reusable, and performant.

The Fundamental Challenge: Making the UI React to Change

```
function ColorChanger() {
  let color = 'red';

  function changeColor() {
    color = 'blue';
    console.log(color); // 'blue'
  }

  return (
    <>
      <h1>My favorite color is {color}</h1>
      <button onClick={changeColor}>Blue</button>
    </>
  );
}
```

What Happens

When the button is clicked, the `color` variable is updated to 'blue' and this is confirmed in the console.

The Problem

The UI does not update. The `<h1>` still displays 'My favorite color is red.'

The 'Why'

React doesn't know it needs to re-render the component when a plain variable changes. It needs a way to 'subscribe' to data changes.

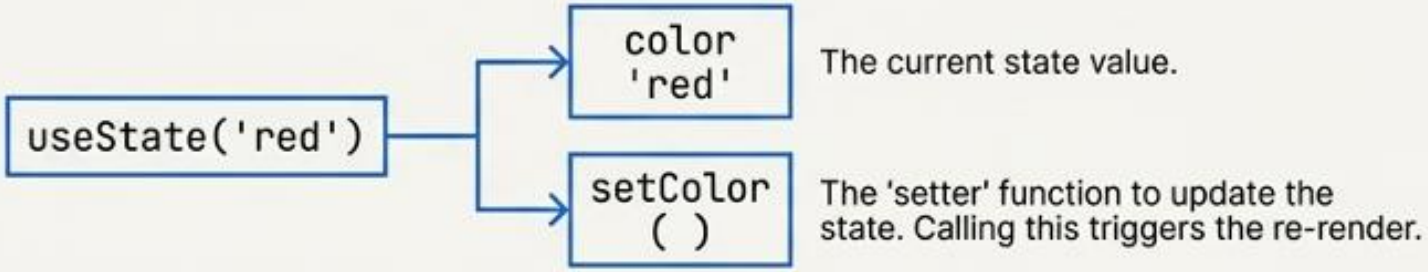
Solution 1: Tracking State with `useState`

Why: We need a way to declare a variable that React *knows* to watch, triggering a re-render whenever its value changes.

What: `useState` is a hook that lets you declare a "state variable." It returns a pair: the current state value and a function that lets you update it.

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

function ColorChanger() {
  const [color, setColor] = useState('red');

  function changeColor() {
    setColor('blue'); // This tells React to re-render!
  }

  return (
    <>
      <h1>My favorite color is {color}</h1>
      <button onClick={changeColor}>Blue</button>
    </>
  );
}
```

Key Takeaway



Advanced `useState`: Handling Objects and Previous State

Pattern 1: Updating Objects & Arrays (Immutability)

State is considered read-only. Mutating an object directly won't trigger a re-render because the object reference doesn't change.

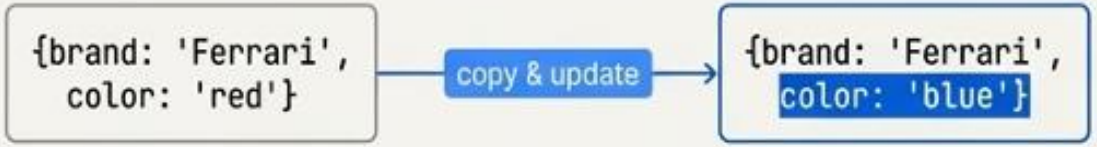
Incorrect Code (Don't do this!)

```
// ► Don't mutate state directly
const [car, setCar] = useState({ brand: 'Ferrari', color: 'red' });

car.color = 'blue';
setCar(car); // React sees the same object reference, may skip re-render
```

Always create a *new* object or array using the spread syntax.

Correct Code (The Solution)

```
// ✅ Replace state with a new object
setCar(prevCar => ({ ...prevCar, color: 'blue' }));
```



Pattern 2: Updating from Previous State (Updater Function)

Multiple state updates in one event handler can be unreliable if they rely on the state's value at the time of the event.

Incorrect Code

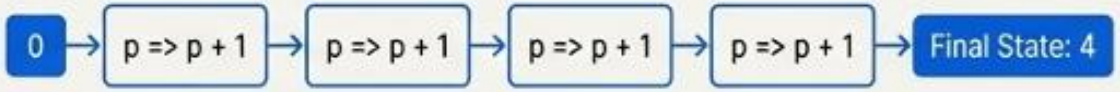
This only increments the count by 1, not 4. Each call uses the same initial `count` value.

```
// ► Unreliable for multiple updates
setCount(count + 1);
setCount(count + 1);
setCount(count + 1);
setCount(count + 1);
```

Pass an "updater function" to guarantee you are working with the latest state.

Correct Code

```
// ✅ Reliable: uses the pending state
setCount(prevCount => prevCount + 1);
setCount(prevCount => prevCount + 1);
// ... (this now works as expected)
```



For Complex Logic: Managing State with `useReducer`

Why: When you have complex state logic involving multiple sub-values or when the next state depends on the previous one in intricate ways, `useState` can become cumbersome and lead to scattered update logic.

What: `useReducer` centralizes your update logic. It is an alternative to `useState` that provides a state and a `dispatch` function. It's often preferred for managing state with multiple actions.

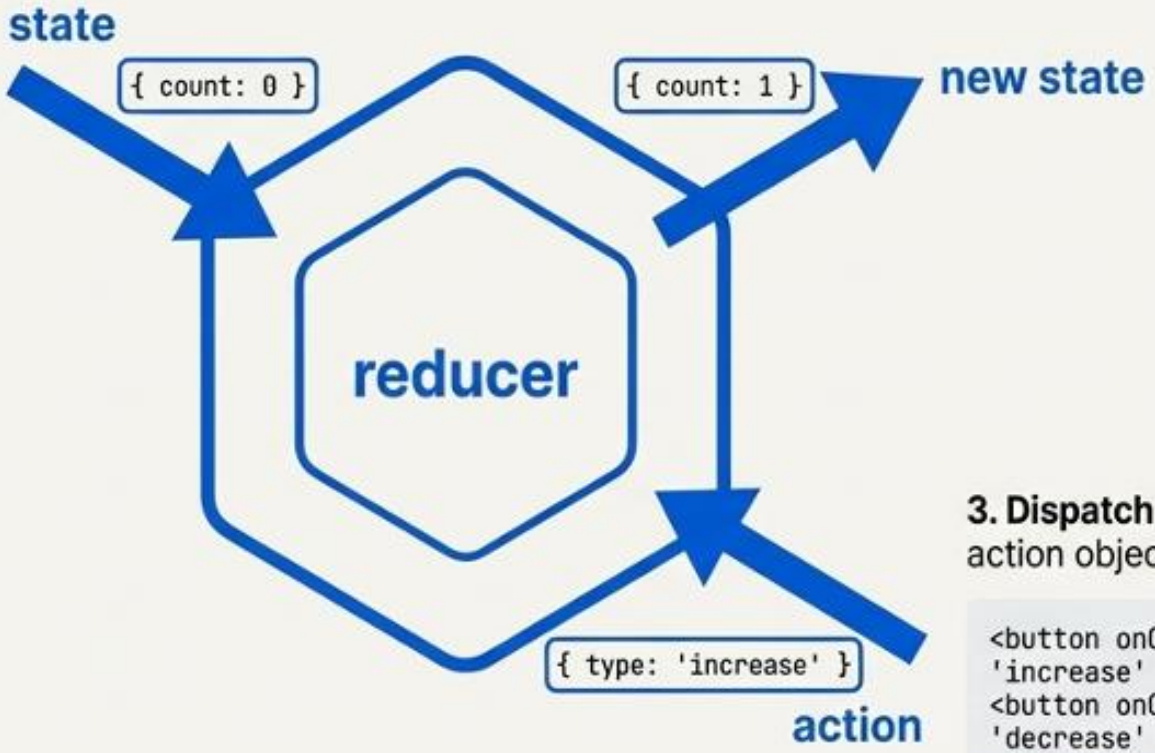
How (The `useReducer` Pattern)

2. Initialize the hook: Pass the reducer and initial state.

```
const initialState = { count: 0 };
const [state, dispatch] = useReducer(redu
cer, initialState);
```

1. Define a `reducer` function: This function specifies how the state gets updated in response to `actions`.

```
function reducer(state, action) {
  switch (action.type) {
    case 'increase':
      return { count: state.count + 1 };
    case 'decrease':
      return { count: state.count - 1 };
    default:
      return state;
  }
}
```



3. Dispatch actions: Call `dispatch` with an action object from your event handlers.

```
<button onClick={() => dispatch({ type:
'increase' })}> + </button>
<button onClick={() => dispatch({ type:
'decrease' })}> - </button>
```

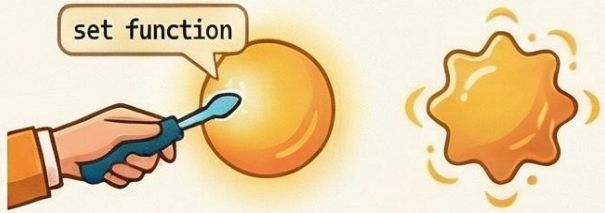
React Hooks: useState vs. useRef

Both allow a component to 'remember' information between renders, but they differ fundamentally in their interaction with React's rendering cycle, making them suitable for very different tasks.

`useState` - For Data That Renders

Manages state variables that are part of the UI.

Updating state triggers a re-render.



Schedule a new render.

State updates are not immediate in the current render.

`useRef` - For Data That Doesn't Render

References a value that is not needed for rendering.

Changing the ref does NOT trigger a re-render.

Freely mutate `.current` without rendering side effects.

Do not read or write `ref.current` during rendering.

Unpredictable behavior; only change in event handlers or effects.

Feature	`useState` (UI Data)	`useRef` (Backstage Data)
Triggers Re-render?	**Yes,** on state change	**No,** change is silent
Value is...	The state variable itself	The `.current` property of the ref object
Primary Use Case	Managing component data for the UI	Accessing DOM nodes or storing mutable values

Handling Side Effects: `useEffect`

Why
Your components often need to interact with systems outside of React's control. This is called a "side effect."

What is `useEffect`?
A hook that lets you synchronize a component with an external system. It runs a function (your "effect") *after* React has committed the changes to the screen.

Core Syntax

```
useEffect(() => {
  // Your side effect logic goes here.
  // This runs after every render by default.
});
```

Mastering `useEffect`: The Dependency Array

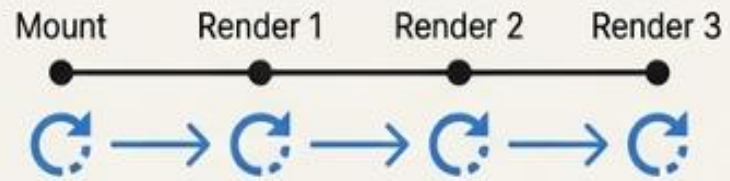
The Dependency Array controls *when* your effect re-runs.

Runs on Every Render

```
useEffect(() => { ... })
```

Behavior: The effect runs after the initial render and *every subsequent re-render*.

Use Case: Rare. Often leads to infinite loops if the effect itself triggers a state update.

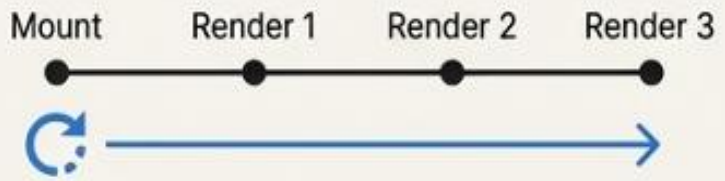


Runs Only Once

```
useEffect(() => { ... }, [])
```

Behavior: The effect runs only once, after the component first mounts. It's the equivalent of `componentDidMount`.

Use Case: Initial data fetching, setting up subscriptions or timers that don't depend on props or state.

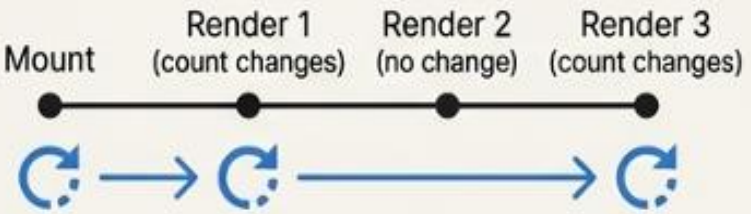


Runs When Dependencies Change

```
useEffect(() => { ... }, [count])
```

Behavior: The effect runs after the initial mount, and then *only if* any of the values in the dependency array have changed since the last render.

Use Case: The most common pattern. Used to re-run an effect in response to changes in props or state.



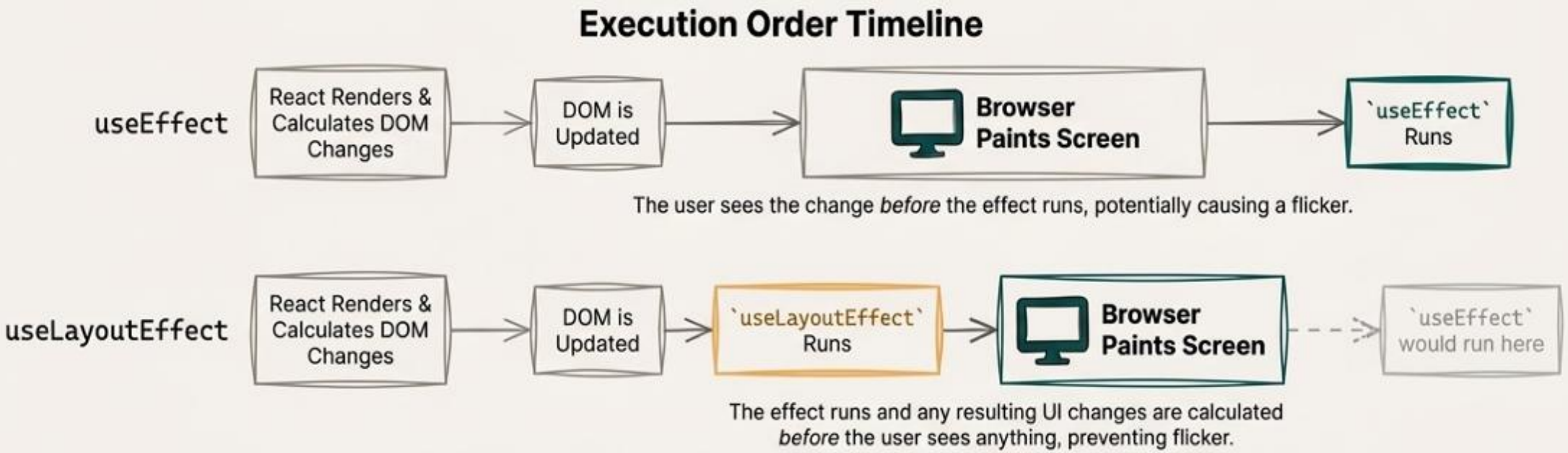
The Specialist Tool: `useLayoutEffect` for Synchronous Updates

The Tool

Works identically to `useEffect` but fires *synchronously* after all DOM mutations, but *before* the browser has painted the changes to the screen.

When to Use It (And When Not To)

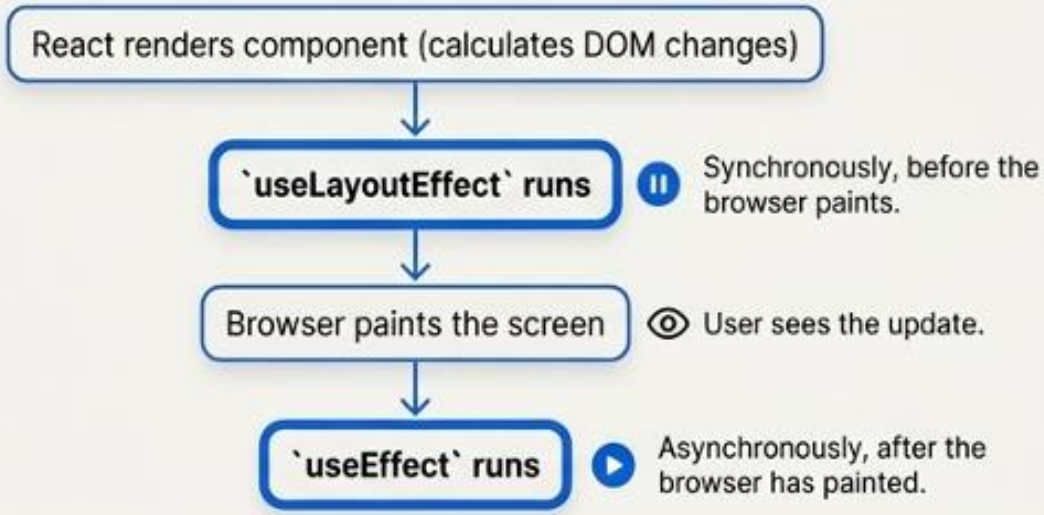
- **Use Case**: To read layout from the DOM and synchronously re-render. Useful for measuring a DOM element's size or position and then changing something before the user sees it.
- **The Problem It Solves**: Prevents a visual "flicker" or "blink" where the user briefly sees an intermediate state before your effect code runs.



`useLayoutEffect` can hurt performance because it blocks browser painting. The React docs recommend: **"It is recommended to use `useEffect` when possible."**

A Matter of Timing: `useEffect` vs. `useLayoutEffect`

The Core Difference is Execution Timing.



Left Column: `useEffect`

When: After the browser has painted.

Impact: Doesn't block painting. The user sees the UI update before the effect runs.

Best For: Most side effects, especially data fetching, where a slight delay is acceptable. **This is the one you should use by default.**

Right Column: `useLayoutEffect`

When: Before the browser has painted.

Impact: Blocks painting. The effect must complete before the user sees the UI update.

Best For: Rare cases where you need to measure the DOM (e.g., get a div's height) and then synchronously re-render to change something *before* the user sees a flicker.

Pro-Tip: `useLayoutEffect` can hurt performance because it's blocking. The React team recommends starting with `useEffect` and only using `useLayoutEffect` if you encounter a specific issue like visual flickering.

Practice useState and useEffect

1. You're building a React component and want to manage a user's profile information, which includes their name, age, occupation and email. If you use a single useState hook to store this information as an object, how should you update only the user's email without losing the existing name and age when the user changes their email in an input field?

2. Imagine you're building a simple status display for an online service. You want to show whether the service is "Online" or "Offline" and update this status every few seconds, simulating a check.

Your task:

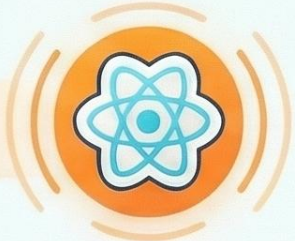
Create a React functional component called ServiceStatus. Use useState to manage a status variable, initialized to "Checking...".
Use useEffect to toggle the status between "Online" and "Offline" every 3000 milliseconds (3 seconds). It should start as "Online" after the initial "Checking...". Display the current status in your component.
This will involve using setInterval inside useEffect and clearing it up when the component unmounts to prevent memory leaks (a common and important pattern with useEffect and timers!).

A Quick Guide to React's useRef Hook

CORE CONCEPTS & PROPERTIES

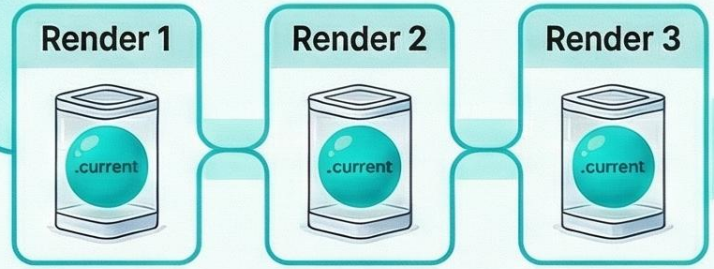
A Mutable Reference Object

useRef returns an object with a single `.current` property for your value.

useState (Triggers Re-render)	useRef (NO Re-render)
	
⚠ Re-render	✓ Changing a Ref

Does NOT Trigger a Re-render

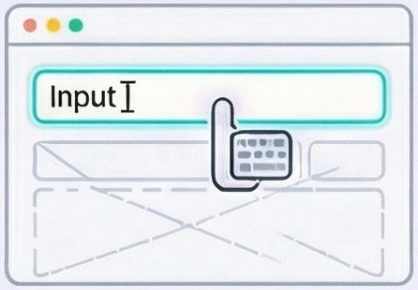
This is the key difference from useState; refs are for values outside rendering logic.



Persists Across Renders

Unlike regular variables, the ref object remains the same on every component render.

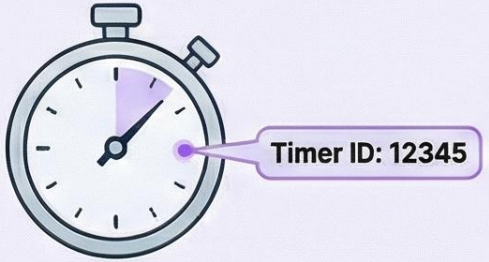
USE CASES & CRITICAL RULES



Use Case: Manipulating the DOM

Call methods like **.focus()** on an input element.

Directly access DOM nodes.



Use Case: Storing a Non-Visual Value

Keep track of values like timer IDs that don't affect the visual output.

Critical Rule: Do NOT Read or Write `.current` During Component Rendering

Mutate `.current` only in event handlers or effects to keep components pure.

✗ Component's Rendering

✓ Button or {u}

Event Handler or Effect
Mutate **.current** ONLY here

Persisting Values Without Re-Renders: `useRef`

Why: Sometimes you need to store a value that persists across renders, but changing it should *not* trigger a re-render. Examples include timer IDs, counters for analytics, or any mutable value not directly used in the JSX.

What: `useRef` returns a mutable ref object whose `.current` property is initialized to the passed argument. This object persists for the full lifetime of the component.

How:

Declaration

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

⟶

Mutation

```
intervalRef.current = newIntervalId;
```

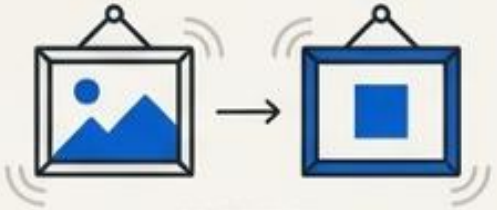
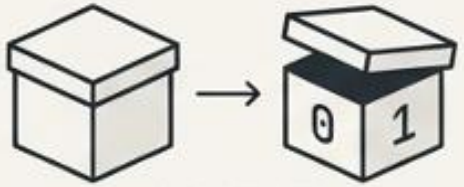
⟶

Reading

```
const id = intervalRef.current;
```

⟶ id: newIntervalId

Juxtaposition: `useState` vs. `useRef`

`useState`	`useRef`
 RE-RENDER	 NO RE-RENDER
Updating the value **triggers a re-render**.	Updating `.current` **does NOT trigger a re-render**.
The value is read directly (e.g., `count`).	The value is accessed via `.current` (e.g., `ref.current`).
Use for anything the user sees on the screen.	Use for "under the hood" values that don't affect the visual output.

Accessing the Imperative World: Manipulating the DOM with `useRef`

Why: While React handles most DOM updates, sometimes you need to imperatively access a DOM element to call methods like `.focus()`, `.play()`, or to measure its size and position.

What: `useRef` can hold a reference to a DOM node.

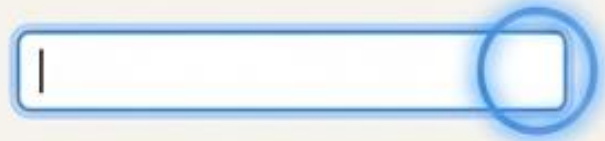
How (The 3-Step Process)

1. Declare the ref with an initial value of `null`.

```
const inputRef = useRef(null);
```

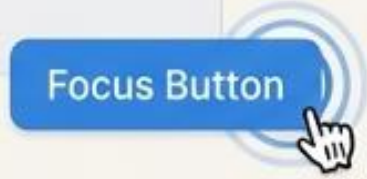
2. Attach the ref to a DOM element in your JSX.

```
<input ref={inputRef} />
```



3. Access the DOM node via `ref.current` in event handlers or effects.

```
function handleFocusClick() {
  inputRef.current.focus();
}
```

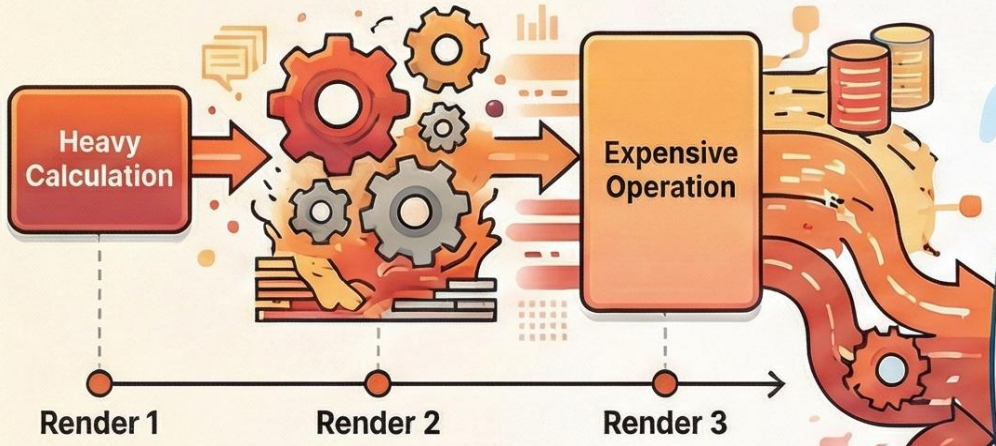


⚠ CRITICAL CAVEAT

Do not write *or read* `ref.current` during the rendering phase. It makes component behavior unpredictable. Only access `.current` inside event handlers or `useEffect`.

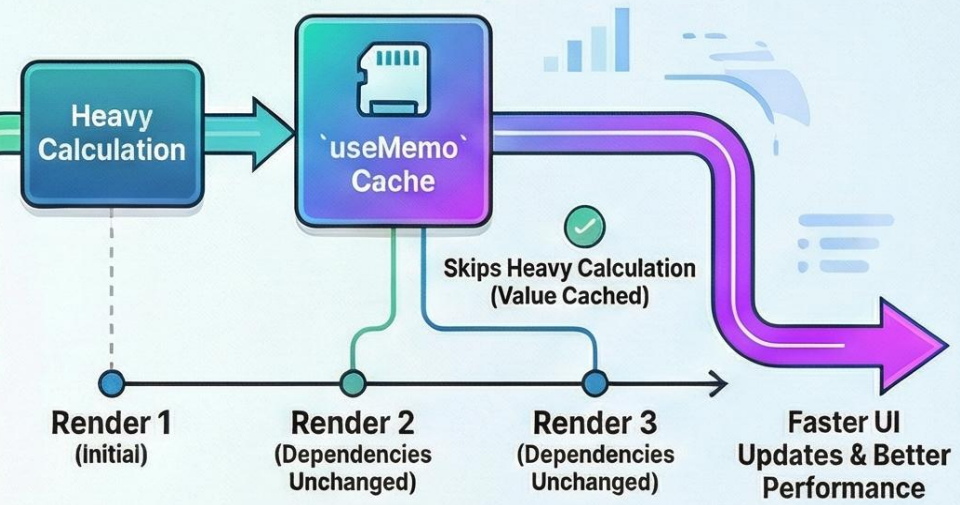
Optimise Your React App with `useMemo`

Expensive calculations run on every single render.



Skips heavy calculations.

Wraps an expensive function so it only runs when needed, making UI updates faster.

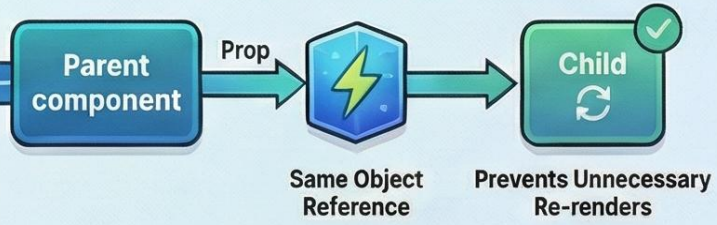


THE SOLUTION:

Smart Caching with `useMemo`

Optimised Child Components Re-render Unnecessarily

Stabilises Props and Dependencies



`useMemo` caches a value between renders
It only re-runs the calculation if a value in its "dependency array" changes

Important Considerations

Use Only for Performance Optimisation
Your application's logic should not depend on `useMemo` to function correctly.

For functions, prefer `useCallback`
`useCallback` is the dedicated hook for memoizing functions themselves.

Problem:
When your component re-renders, *everything inside it runs again*, including heavy calculations.

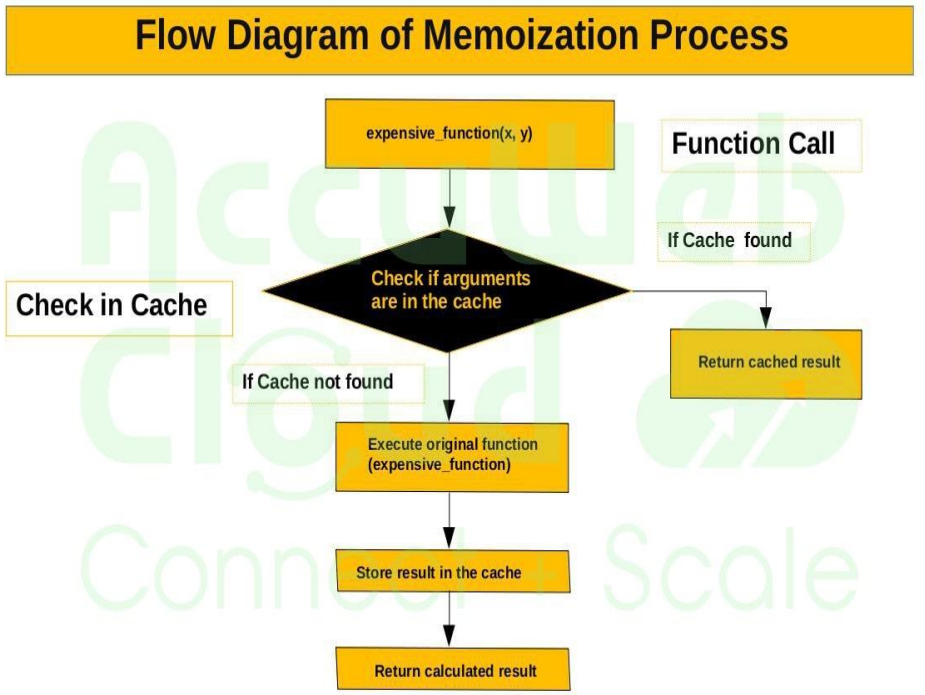
Solution:
useMemo remembers the result of a calculation.

How it works:
It runs the function ONLY when the dependency changes.
Otherwise, it reuses the old saved value (cached value).

Example from your code:
```
const result = useMemo(() =>
cubeCalculation(number), [number]);
```

This means:
- If **number changes** → calculate cube again
- If **count changes** → DO NOT calculate cube again
- Saves performance and prevents unnecessary work

Think of useMemo like:
"Don't calculate again unless it's really needed."

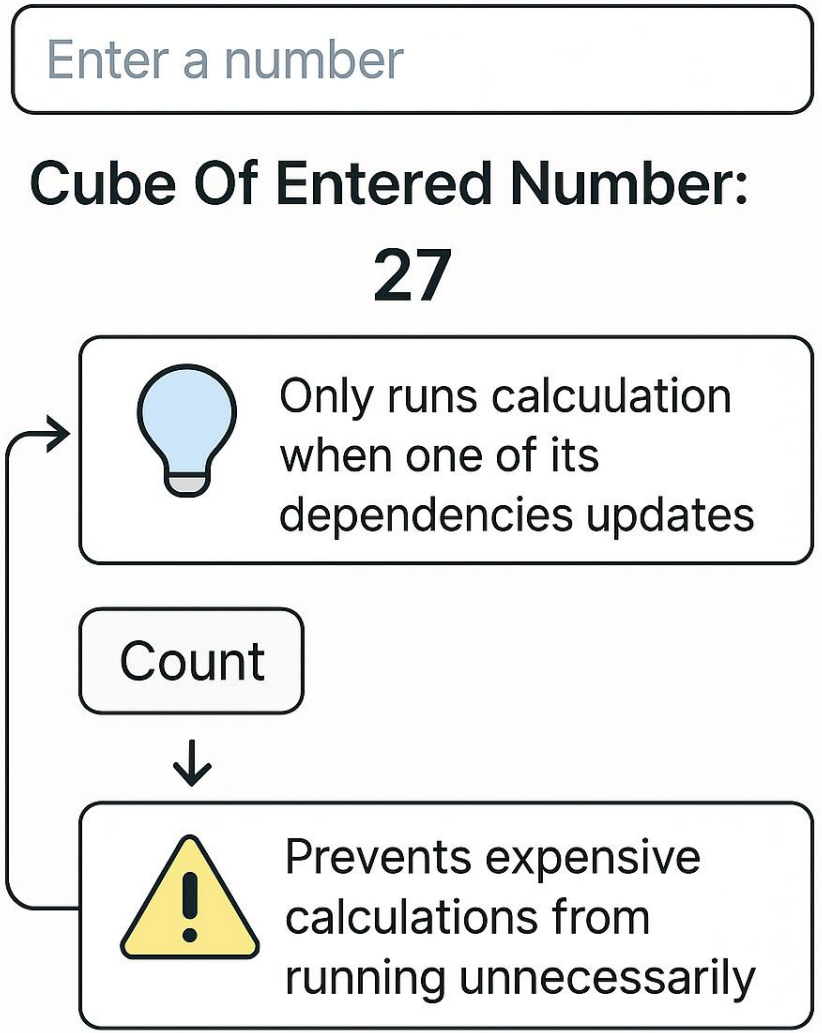


```jsx
import React, { useMemo, useState } from "react";

export default const UseMemo = () => {
 const [number, setNumber] = useState(0);
 const [count, setCount] = useState(0);

 const cubeCalculation = (num) => {
  console.log("Calculating cube...");
  return Math.pow(num, 3);
 };

 // const result = cubeCalculation(number);
 const result = useMemo(() => {
  return cubeCalculation(number);
 }, [number]);

 return (
  <>
   <input type="text“  value={number}
    onChange={(e) => setNumber(e.target.value)}
    placeholder="Enter a number“ />
   <p>Cube Of Entered Number: {result}</p>
   <button onClick={() => setCount(count + 1)}>Count</button>
   <p>{count}</p>
  </>
 );};
```

Example

useMemo

This hook returns a memoized value

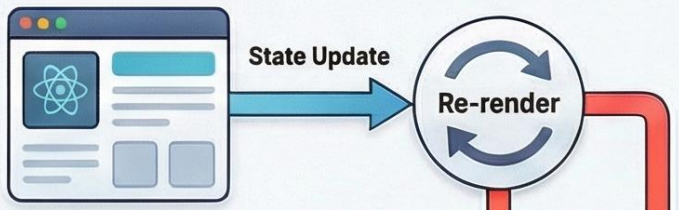


React's useMemo Hook: The Performance Optimizer

Explaining how useMemo optimizes component performance by caching the results of expensive calculations, preventing them from running on every re-render.

The Problem: Wasted Renders

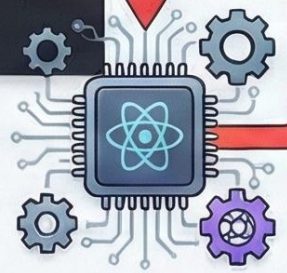
Every State Change Causes a Re-render



An Expensive Calculation

```
const cubeCalculation = (number) => {
  ...expensive...
};
```

number | count



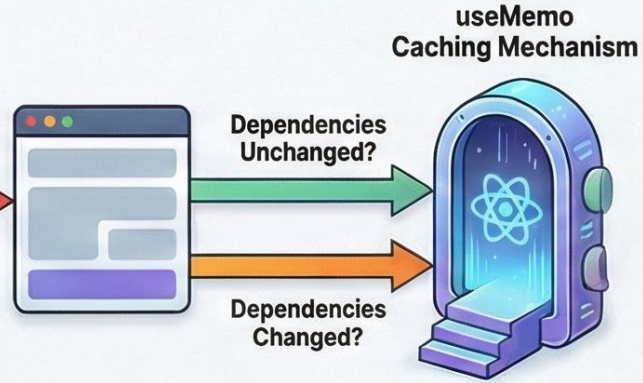
Unnecessary Work is Performed

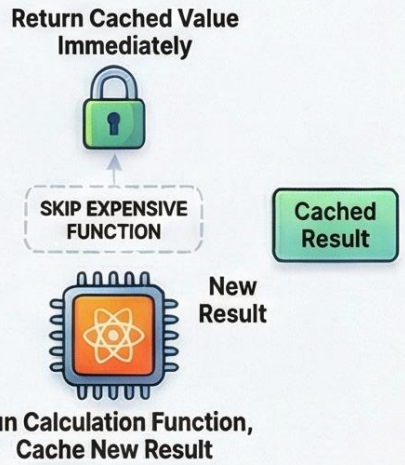
Without optimization, every time the count button is clicked, the component re-renders and **cubeCalculation** runs again, even though the number it depends on hasn't changed.

The Solution: Memoization with useMemo

useMemo Caches Calculation Results

useMemo is a hook that takes a calculation function and a dependency array. it stores the result and only re-calculates it when a dependency changes.





The Syntax

```
const result = useMemo(() => {
  return cubeCalculation(number);
}, [number]]; // Only re-calculates when 'number' changes
```

Don't Forget Dependencies

If you provide an empty [] or omit the dependency array, the value will never be recalculated after the initial render, leading to stale data.

Top 3 Use Cases for useMemo

1. Skipping Expensive Calculations

Perfect for heavy operations like filtering, sorting, or transforming large arrays of data that shouldn't run on every minor UI update.

2. Preventing Child Component Re-renders

When passing an object or array as a prop, useMemo ensures the child component receives the exact same object reference between renders (if dependencies are stable), allowing React.memo to work correctly.

3. Stabilizing Dependencies for Other Hooks

If an object or array is a dependency of another hook like **useEffect**, wrapping its creation in **useMemo** prevents the effect from running on every render.

Key Considerations

It's a Performance Optimization

Only use useMemo to improve speed. Your application logic should work correctly even without it.

For Functions, Use useCallback

While useMemo can memoize a function, useCallback is the specialized hook for that purpose with a cleaner syntax.

The Future is Automatic

The upcoming React Compiler aims to handle memnization automatically, which may reduce the need for manual useMemo calls in the future.

Optimizing Performance with Memoization

Core Concept: Memoization

A form of **caching**. It saves the result of a function call and returns the cached result when the same inputs occur again, avoiding re-computation.

The Tools

`useMemo` and `useCallback` are similar. They only re-run when one of their dependencies has updated.

useMemo: Memoizing a Value

Job

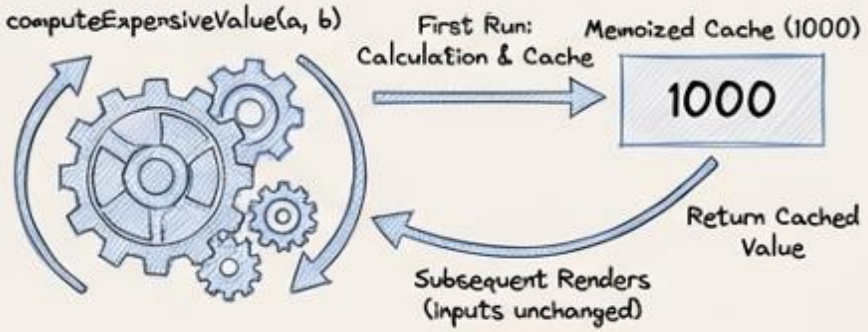
Caches the **return value** of an expensive calculation.

Syntax

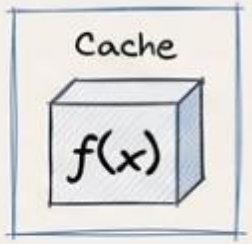
```
const memoizedValue = useMemo(
  () => computeExpensiveValue(a, b),
  [a, b]
);
```

Use Case

Prevents re-running a heavy function on every render, especially when its inputs (`a`, `b`) haven't changed.



useCallback: Memoizing a Function



Job

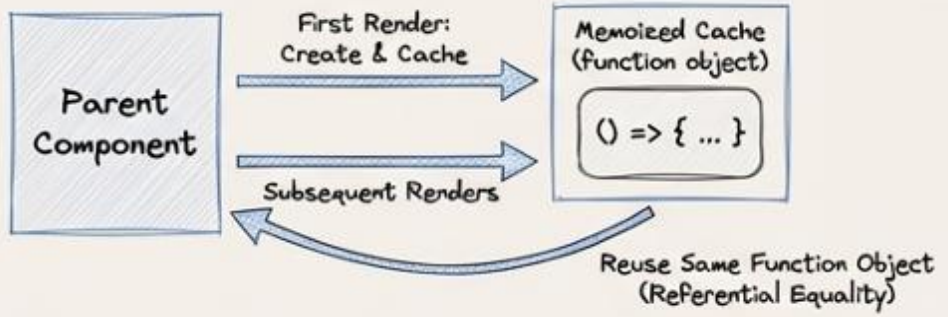
Caches the **function definition itself**.

Syntax

```
const memoizedCallback = useCallback(
  () => { doSomething(a, b); },
  [a, b]
);
```

Use Case

Prevents re-creating a function on every render. Critical when passing callbacks to optimized child components that rely on referential equality.



useMemo returns a memoized **value**. **useCallback** returns a memoized **function**.

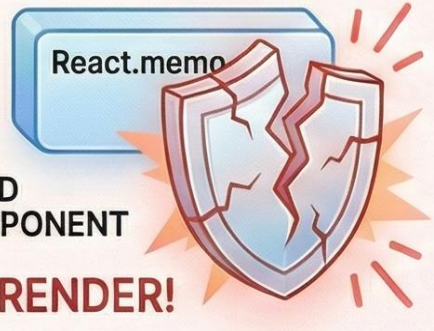
Mastering React's **useCallback**: A Performance Guide

PARENT COMPONENT

THE PROBLEM:
Unnecessary Re-renders

JavaScript creates a new function on every component render.

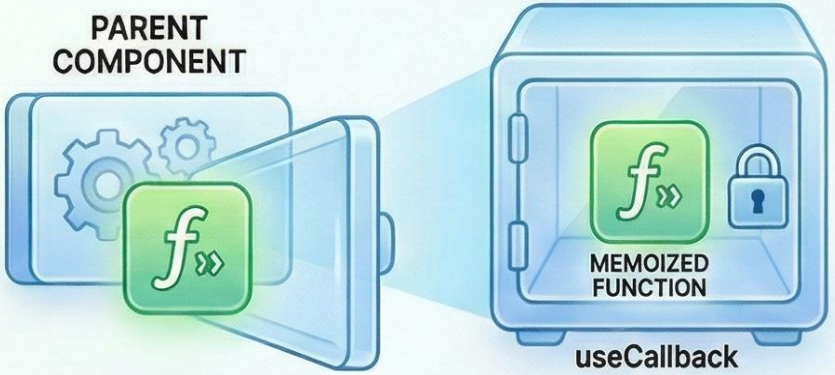
NEW FUNCTION



Causes performance issues as child wrapped in **React.memo** re-renders.

Child receives a "new" function prop each time parent re-renders, breaking memoization.

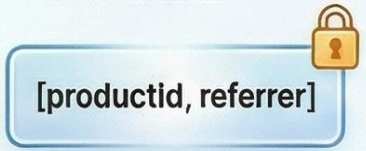
PARENT COMPONENT



THE SOLUTION:
Caching with useCallback

useCallback caches, or "memoizes," your function.

MEMOIZED FUNCTION



The cached function is only recreated when its dependencies change. Provide a dependency array.

Returns the exact same function instance, enabling **React.memo** to work correctly.

BEST PRACTICES & KEY DISTINCTIONS

BEST PRACTICE

Use for **performance optimization only.** If code doesn't work without it, fix the underlying problem first.

useCallback vs. useMemo

`() => {}`	`=> VALUE`
Caches the function **ITSELF**	Caches a function's **RETURN VALUE**

BEST PRACTICE

Don't **overuse** it. it can make code less readable and is often unnecessary.

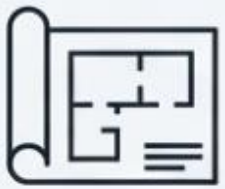
The Culprit: A Case of Mistaken Identity

The problem is **Referential Equality**.

In JavaScript, a function declaration (`function() {}` or `() => {}`) creates a **new, different function every time it's executed**. When `ProductPage` re-renders, it creates a brand new `handleSubmit` function.

Function Blueprint

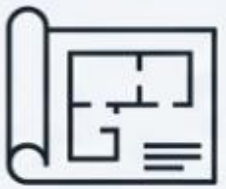
Render 1



handleSubmit

≠

Render 2



handleSubmit

`React.memo` sees these as two different blueprints,
even though the instructions are the same.
The shallow prop comparison fails.

```
function ProductPage({ theme }) {
  // Every time ProductPage re-renders (e.g., due to 'theme' change)...
  function handleSubmit(orderDetails) { /* ... */ } <— ...a NEW function is created here.
  }
  return <ShippingForm onSubmit={handleSubmit} />;
}
```

The Case Solved: `useCallback` and `memo` in Action

By wrapping `handleSubmit` in `useCallback`, we ensure it's the *same function* between re-renders, as long as its dependencies (`productId`, `referrer`) don't change. Now, `React.memo` works as expected.

```
function ProductPage({ productId, referrer, theme }) {
-  function handleSubmit(orderDetails) {
-    post('/product/' + productId + '/buy', { referrer, orderDetails });
-  }
+  const handleSubmit = useCallback((orderDetails) => {
+    post('/product/' + productId + '/buy', { referrer, orderDetails });
+  }, [productId, referrer]);

  return (
    <div className={theme}>
      {/* Now, ShippingForm receives the same prop and can skip re-rendering. */}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

<-- Cache depends on these values.

When the `theme` changes, `handleSubmit`'s identity is preserved. `ShippingForm`'s props are now equal, and the unnecessary re-render is prevented.

Deep Dive: Why `useCallback` Matters for Child Components

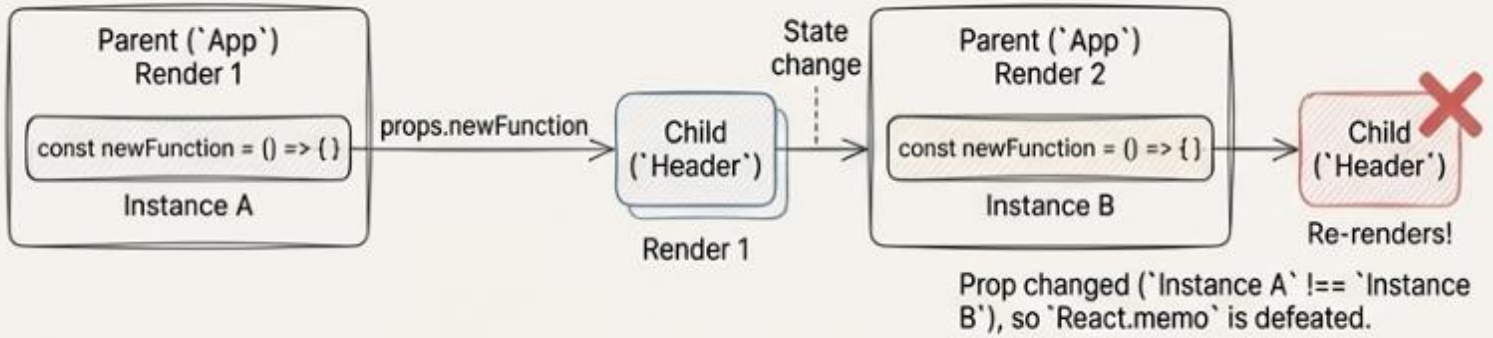
The Problem: Referential Equality

In JavaScript, `() => {}` is not equal to `() => {}`. Every time a component renders, functions defined inside it are brand new instances.

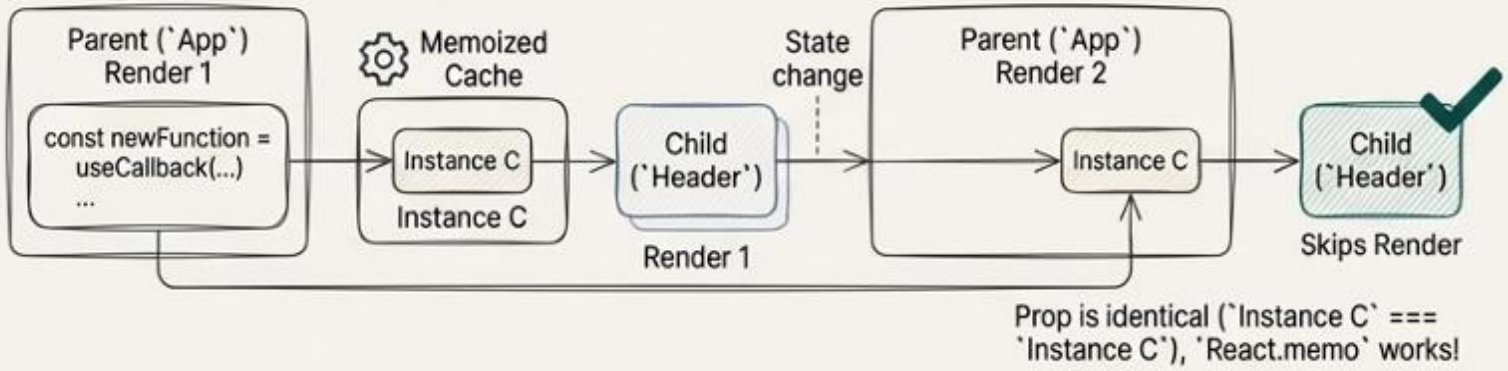
The Impact

If you pass a function as a prop to a child component wrapped in `React.memo`, the child will re-render every time, because from its perspective, it's receiving a "new" prop.

Without `useCallback`



With `useCallback`



The Solution Code

```
// Wrap the function definition in useCallback
const newFunction = useCallback(() => {
  // ... function body
}, []); // Empty dependency array means the function
is created only once
```

Example

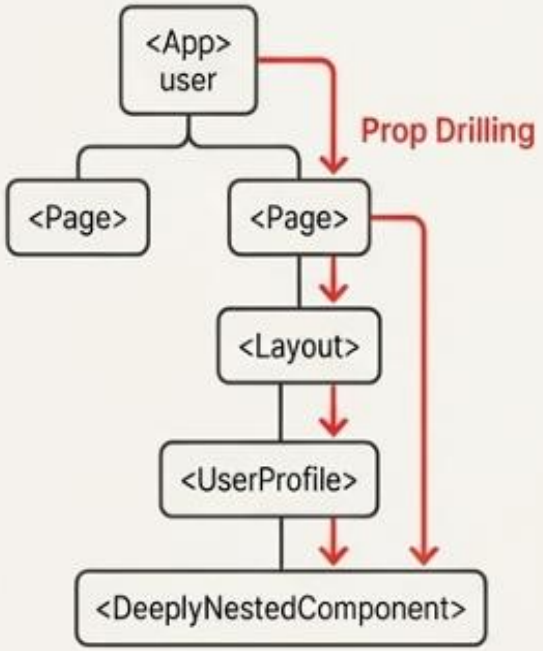
useCallback is a React Hook that memoizes (caches) a function so that it does not get recreated on every render — unless its dependencies change.

This is useful when you're passing functions down to child components that rely on reference equality (e.g., via React.memo), because in React, functions are objects, and a new function reference gets created on every render unless it's memoized.

```
import React, { useCallback, useState } from "react";
import Header from "./Header";

const UseCallback = () => {
  const [count, setCount] = useState(0);

  // const newFn = () => {};
  const newFn = useCallback(() => {}, []);

  return (
    <div>
      <Header newFn={newFn} />
      <p>Count: {count}</p>
      <button onClick={() => setCount((prev) => prev + 1)}>
        Increment Count
      </button>
    </div>
  );
};

export default UseCallback;
```

Sharing Data Across Components: `useContext`

Why (The Problem of "Prop Drilling")

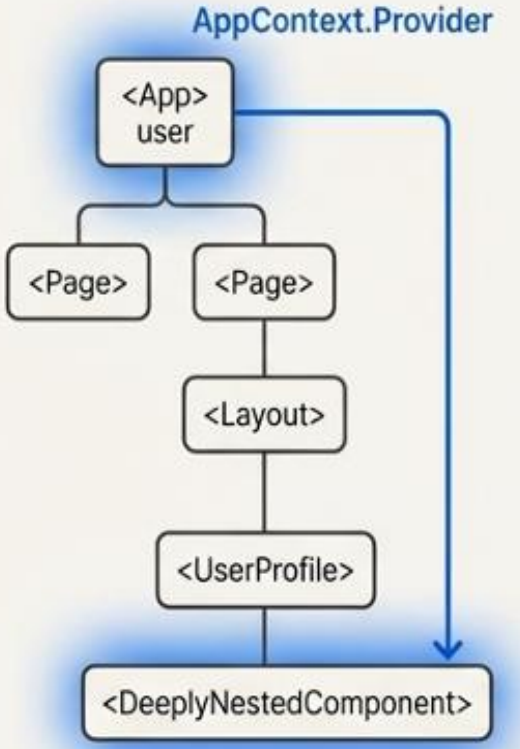


Imagine you have data in a top-level component that a deeply nested child component needs.

You would have to pass that data down as props through every intermediate component.

This is tedious and makes refactoring difficult.

How (The 3-Step Process)



1. Create a Context

```
// AppContext.js
import { createContext } from 'react';

export const AppContext = createContext(null);
```

2. Provide the Context

```
// App.js
import { AppContext } from './AppContext';
import Page from './Page';

<AppContext.Provider value={{ user: 'Jane' }}>
  <Page />
</AppContext.Provider>
```

3. Consume the Context

```
// DeeplyNestedComponent.js
import { useContext } from 'react';
import { AppContext } from './AppContext';

const { user } = useContext(AppContext); // No
prop drilling!
return <h1>Welcome, {user}!</h1>;
```

What

The Context API provides a way to pass data through the component tree without having to pass props down manually at every level. `useContext` is the hook that lets a component read from a context.

useContext in Action: Implementing a Theme

Defining the Context

```
// ThemeContext.js
import { createContext } from 'react';

export const ThemeContext =
  createContext('light');
```

First, we create a context with a default value. This is used if a component isn't inside a provider.

Providing the Context

```
// App.js
import { ThemeContext } from './ThemeContext';

function App() {
  const [theme, setTheme] = useState('dark');

  return (
    <ThemeContext.Provider value={theme}>
      <Form />
      {/* ... button to setTheme ... */}
    </ThemeContext.Provider>
  );
}
```

Next, wrap the application tree in the Provider, passing the current state as the `value`.

Consuming the Context

```
// Button.js
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

function Button({ children }) {
  const theme = useContext(ThemeContext);
  const className = 'button-' + theme;

  return <button className={className}>
>{children}</button>;
}
```

Finally, any nested component can access the `theme` value with a single hook call.

useContext simplifies data access and declutters our component tree.

AppContex.jsx

You, 3 minutes ago | 1 author (You)

```jsx
import { createContext } from "react";
export const AppContext = createContext();

const ContextProvider = (props) => {
  const phone = "123-456-7890";
  const address = "123 Main St, Anytown, USA";

  return (
    <AppContext.Provider value={{ phone, address }}>
      {props.children}
    </AppContext.Provider>
  );
};

export default ContextProvider;


//Step 1: Creating/Defining The Contex inside a context folder in src.
```

main.jsx

You, 1 second ago | 1 author (You)

```jsx
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import "./index.css";
import App from "./App.jsx";

import ContextProvider from "./context/AppContex.jsx";

createRoot(document.getElementById("root")).render(
  <ContextProvider>
    <StrictMode>
      <App />
    </StrictMode>
  </ContextProvider>
);

//Step 2: Use The Context in the main.jsx
```

Contact.jsx

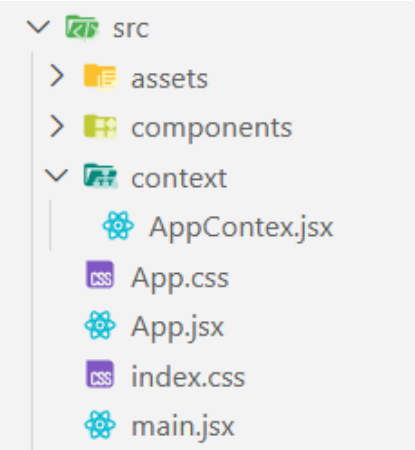
You, 49 seconds ago | 1 author (You)

```jsx
import React, { useContext } from "react";
import { AppContext } from "../context/AppContex";

const Contact = () => {
  const {phone, address} = useContext(AppContext)
  return (
    <div>
      <h4>This is Contact Component</h4>
      <p>Phone Number: {phone}</p>
      <p>
        <i>Address: {address}</i>
      </p>
    </div>
  );
};
export default Contact;

//Step 3: Consuming The Contex
```

App.jsx

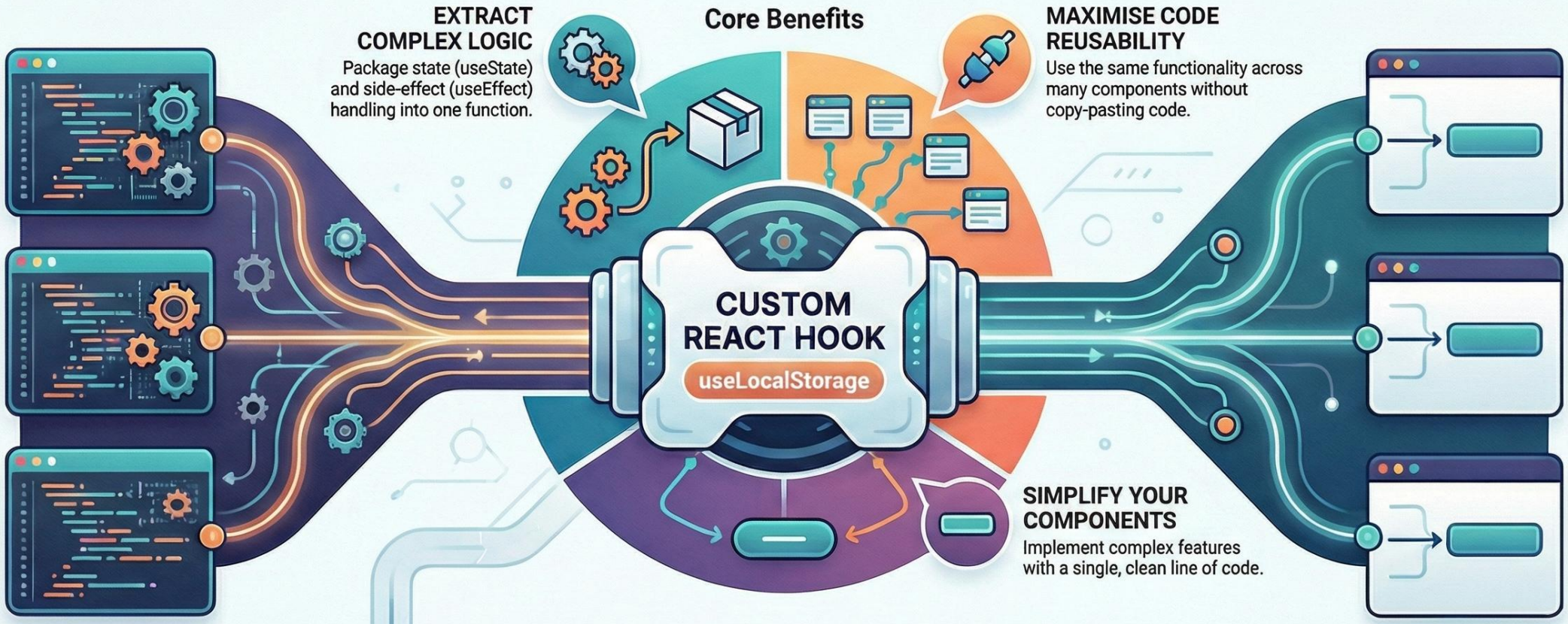
You, 13 seconds ago | 1 author (Yo

```jsx
import Header from "./components/Header";
import Footer from "./components/Footer";
import Profile from "./components/Profile";

const App = () => {
  return (
    <>
      <Header />
      <Profile />
      <Footer />
    </>
  );
};

export default App;
```



The Power of Custom React Hooks

Custom React Hooks are functions that allow developers to extract and reuse component logic, such as state management and side effects. By following a use naming convention, they help simplify complex components and reduce code repetition across an application.



How They Work: The useLocalStorage Example

Naming Convention: Must Start with **use**

NAMING CONVENTION:
allows React to enforce rules for hooks (e.g., **useLocalStorage**)

1. INITIALISE AND MANAGE STATE
Uses **useState** internally to get data from local storage or an initial value.

2. PERSIST CHANGES AUTOMATICALLY
Uses **useEffect** to save the state back to local storage when it changes.

We've all written this logic. And then written it again.

The core problem in large React applications is not just complexity, but repetition. Logic for fetching data, managing local storage, or subscribing to browser events often gets copied and pasted across multiple components, leading to code that is difficult to maintain and debug.

UserProfile.js

```
function UserProfile() {
  const [name, setName] = useState(() => {
    const saved = localStorage.getItem('username');
    return saved || 'Guest';
  });

  useEffect(() => {
    localStorage.setItem('username', name);
  }, [name]);

  // ... component render logic
}
```

SettingsPanel.js

```
function SettingsPanel() {
  const [theme, setTheme] = useState(() => {
    const saved = localStorage.getItem('theme');
    return saved || 'light';
  });

  useEffect(() => {
    localStorage.setItem('theme', theme);
  }, [theme]);

  // ... component render logic
}
```

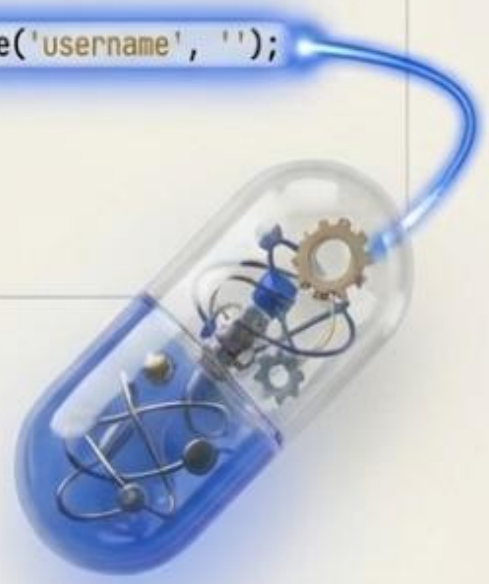
The goal is to elevate logic, not just repeat it.

Before: Logic Inside the Component

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

function UserProfile() {
  const [name, setName] = useState(() => {
    try {
      const item = window.localStorage.getItem('username');
      return item ? JSON.parse(item) : '';
    } catch (error) {
      return '';
    }
  });

  useEffect(() => {
    try {
      window.localStorage.setItem('username', JSON.stringify(name));
    } catch (error) {
      console.log(error);
    }
  }, [name]);

  return (
    // ... JSX to render input
  );
}
```

After: Logic Extracted to a Hook

```
import { useLocalStorage } from './hooks';

function UserProfile() {
  const [name, setName] = useLocalStorage('username', '');

  return (
    // ... JSX to render input
  );
}
```



Let's Build a Practical Example: `useLocalStorage`

To understand how custom hooks work, we will build one from scratch. Our goal is to create a `useLocalStorage` hook that provides a `useState`-like interface but automatically persists **the state to the browser's local storage**.

Inputs

- `key` (for the storage item)
- `initialValue`

Outputs

- `[value, setValue]` array, just like `useState`

Step 1: The Function Shell and Internal State

```
function useLocalStorage(key, initialValue) {
  // 1. The state to store our value
  const [storedValue, setStoredValue] = useState(() => {
    // 2. Logic to get initial value from localStorage
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.log(error);
      return initialValue;
    }
  });
  // ... more to come
}
```

1. `useState(() => ...)`:
We use the 'lazy initial state' function to ensure we only read from `localStorage` once on the initial render, which is a performance optimisation.

2. `getItem(key)`:
We attempt to retrieve and parse the existing value from local storage. If it fails or doesn't exist, we fall back to the `initialValue`.

Step 2: Persisting State Changes with a Side Effect

```
function useLocalStorage(key, initialValue) {
  // 1. The state to store our value
  const [storedValue, setStoredValue] = useState(() => {
    // ... (useState logic from previous slide)
  });

  // 1. The side effect to update localStorage
  useEffect(() => {
    try {
      window.localStorage.setItem(key, JSON.stringify(storedValue));
    } catch (error) {
      console.log(error);
    }
  }, [key, storedValue]); // 2. The dependency array
}
```

1. `useEffect(...)`: This hook runs after every render. It synchronises our component's state *back* to local storage.

2. `[key, storedValue]`: This is critical. The effect will only re-run if the `key` or the `storedValue` changes, preventing unnecessary writes to local storage.

Step 3: Returning the `useState`-like API

```
import { ao, hook } from 'react';

  const {storedValue } = require('storedValue');
  const {storedValue }, = [setStoredValue];

  useEffect(() => {
    useEffect(..) => {
      const [storedValue, setStoredValue];
      return cont[storedValue, setStoredValue];
    },
  }, ());

  // ... (useState and useEffect from previous slides)

  // 1. Return the value and a setter function
  return [storedValue, setStoredValue];
```

1. **`return [storedValue, setStoredValue]`**: By returning an array with the current value and its setter function, we mirror the familiar and intuitive API of the built-in `useState` hook. This makes our custom hook easy for any React developer to adopt.

The Complete Building Block: `useLocalStorage`

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

function useLocalStorage(key, initialValue) {
  const [storedValue, setStoredValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      return initialValue;
    }
  });

  useEffect(() => {
    try {
      window.localStorage.setItem(key, JSON.stringify(storedValue));
    } catch (error) {
      console.log(error);
    }
  }, [key, storedValue]);

  return [storedValue, setStoredValue];
}
```

Consumption: Clean, Declarative, and Powerful

With the complex logic encapsulated, our component becomes trivial to read and write. We simply call the hook and use the state it provides.

```
function UserProfile() {
  // All the complexity is hidden in this single line
  const [name, setName] = useLocalStorage('username', 'Guest');
  return (
    <div>
      <label>Name:</label>
      <input type="text" value={name} onChange={(e) => setName(e.target.value)} />
    </div>
  );
}
```

The component doesn't need to know *how* the state is persisted, only that it *is*.

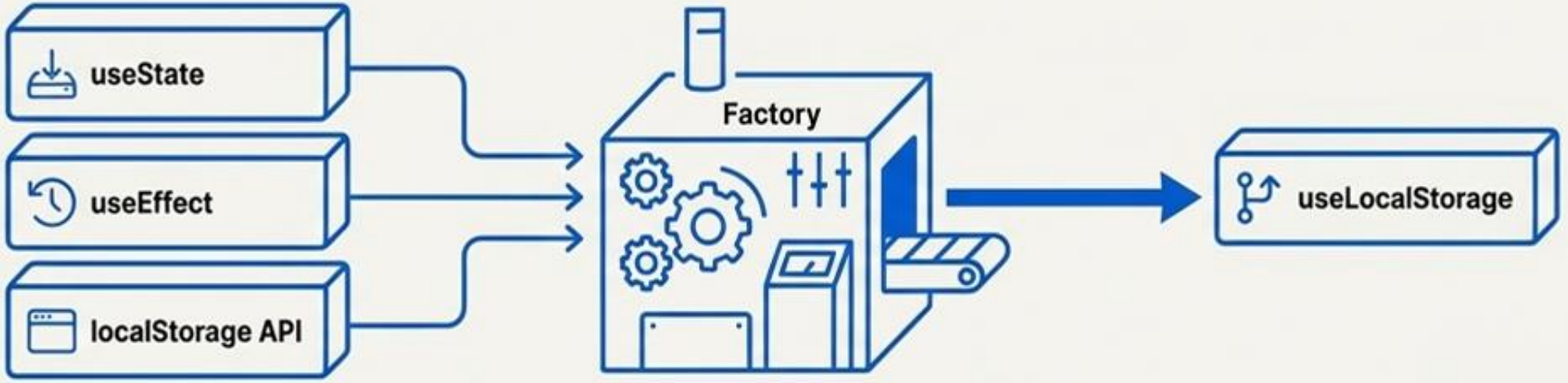
Building Your Own Tools: Custom Hooks

Why

You may find yourself writing the same logic across multiple components (e.g., fetching data, subscribing to events, managing form state). Custom hooks let you extract this component logic into a reusable function.

What

A custom hook is a JavaScript function whose name starts with 'use' and that can call other hooks. It's a convention that lets you share stateful logic, not just UI.



The Custom Hook's Code (`useLocalStorage.js`)

This custom hook abstracts away the logic of using `useState` to store a value and `useEffect` to sync it with local storage.

```
function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    // Logic to get initial value from localStorage
  });

  useEffect(() => {
    // Logic to set value in localStorage when `value` changes
  }, [key, value]);

  return [value, setValue];
}
```

How to Use It (In any component)

The complex logic is hidden behind a simple, declarative API, just like a built-in hook.

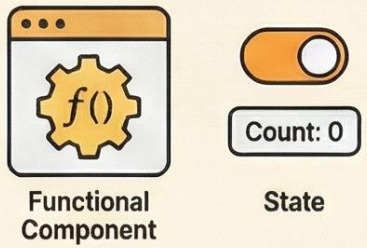
```
function UserSettings() {
  const [name, setName] = useLocalStorage('username', '');
  const [theme, setTheme] = useLocalStorage('theme', 'light');
  // ...
}
```

Hooks are not just tools; they are building blocks. Master them, and you can create your own powerful, reusable abstractions to write cleaner and more maintainable.

Choosing Your React Hook: A Quick Guide

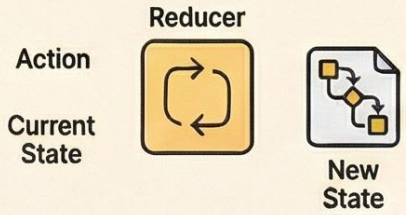
React Hooks are special functions that allow you to use state, lifecycle methods, and other React features within functional components. They simplify code, improve readability and reusability, and are the preferred way to build modern React applications.

State Management



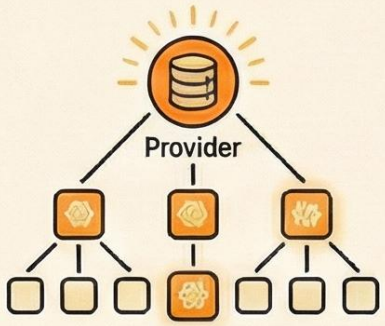
useState: For simple, local component state.

Returns a state variable and a function to update it, triggering re-renders.



useReducer: For complex state logic with multiple sub-values.

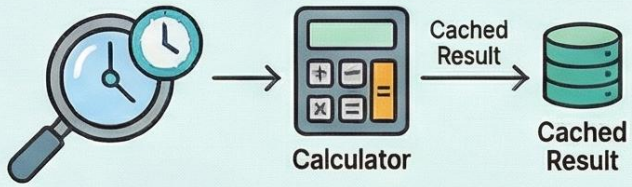
An alternative to useState that uses a reducer for predictable state transitions.



useContext: For sharing "global" data without "prop drilling".

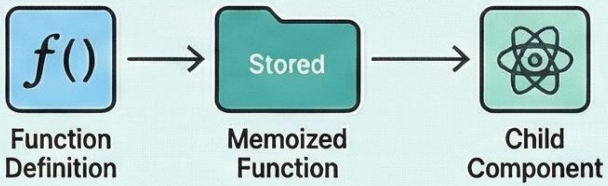
Accesses data from a provider anywhere in the component tree below it.

Performance Optimization



useMemo: Caches the result of an expensive calculation.

it only re-calculates the value when one of its dependencies has changed.



useCallback: Caches a function definition between renders.

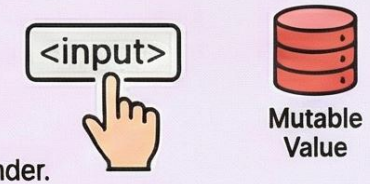
Prevents re-creating functions, optimizing child components that receive callbacks as props.

Side Effects & DOM

useEffect: For handling side effects after the component renders.

Ideal for data fetching, subscriptions, or manually changing the DOM.

useRef: Accesses DOM elements or stores a mutable value.



Updating a ref's value does not trigger a component re-render.

useLayoutEffect: Fires synchronously *before* the browser repaints the screen.

Use for DOM measurements to prevent visual flickering; **useEffect** is preferred otherwise.

React Hooks Blueprint:

State Management

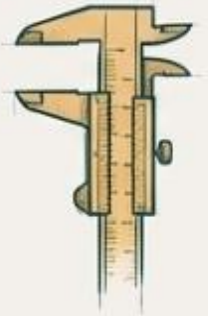
useState
For basic, local component state.

useReducer
For complex state logic with multiple actions.

Side Effects

useEffect
For interacting with the outside world (APIs, timers). Runs *after* browser paint.

useLayoutEffect
For synchronous DOM updates *before* browser paint to prevent flickering.



References

useRef
To access DOM nodes directly or to store mutable values that don't cause re-renders.

Performance

useMemo
To cache the *result* of an expensive calculation.

useCallback
To cache a *function definition* to prevent unnecessary child re-renders.

Global State

useContext
To provide and consume data throughout the app without prop drilling.

The Master Craft

Custom Hooks
Combine any of these tools to create your own reusable logic.

🎉 Thanks for Reaching Till the End!

Your journey into React Hooks doesn't stop here. Keep learning, keep building!

🔗 Connect With Me

🌐 **Portfolio** 💼 **LinkedIn** 🐦 **Twitter (X)** 💻 **GitHub**

📘 Source Code for This Hooks Presentation

👉 **GitHub Repo: https://github.com/itsindrajput/ReactProjects/tree/main/Hooks**

Your time and attention mean a lot.

If you found this helpful, feel free to share or reach out —**Let's build and learn together!** 🚀