

Day 22: Performance Optimization in React

Topics Covered: `React.memo`, `useCallback`, and `useMemo`

Introduction

As your React applications grow larger, performance can start to slow down. Components may re-render unnecessarily, and expensive calculations can be re-run even when not needed.

React gives us **three powerful tools** to optimize performance: - `React.memo` → Prevents re-rendering of components when props haven't changed. - `useCallback` → Memoizes functions so that they don't get recreated on every render. - `useMemo` → Memoizes computed values to avoid expensive recalculations.

These help make your app faster and smoother.

1. `React.memo()`

What is it?

`React.memo` is a higher-order component that wraps your component and tells React to re-render it **only when its props change**.

Syntax:

```
const MyComponent = React.memo(function MyComponent(props) {  
  return <div>{props.name}</div>;  
});
```

Why use it?

When a parent component re-renders, all child components also re-render by default, even if their props haven't changed. `React.memo` helps stop that.

Example:

```
import React, { useState } from "react";  
  
const Child = React.memo(({ name }) => {  
  console.log("Child rendered");  
  return <h3>Hello, {name}</h3>;  
});  
  
export default function App() {
```

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

return (
  <div>
    <Child name="Rehan" />
    <button onClick={() => setCount(count + 1)}>Increment: {count}</button>
  </div>
);
}
```

✓ Here, clicking the button re-renders the parent, but the `Child` doesn't re-render because its props didn't change.

👉 2. `useCallback()`

🧠 What is it?

`useCallback` is a React Hook that memoizes a function — i.e., it returns the same function reference between renders **unless its dependencies change**.

💚 Syntax:

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

❤️ Why use it?

When you pass functions as props to child components, they're recreated on every render. That causes children to re-render unnecessarily.

`useCallback` helps you avoid that.

🐱 Example:

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

const Child = React.memo(({ onClick }) => {
  console.log("Child rendered");
  return <button onClick={onClick}>Click Child</button>;
});

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

  const handleClick = useCallback(() => {
    alert("Child button clicked!");
  }, []);
```

```

    }, []); // function reference stays same

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

```

✓ Here, even if the parent re-renders, the `Child` doesn't re-render because `onClick` has the same reference.

👎 3. `useMemo()`

🧠 What is it?

`useMemo` caches the **result of an expensive calculation** so that React doesn't re-compute it on every render.

💚 Syntax:

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

❤️ Why use it?

If your component does some heavy computation or filtering, you can memoize the result to boost performance.

🐱 Example:

```

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

export default function App() {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState(["Learn React", "Build Projects"]);

  const expensiveCalculation = (num) => {
    console.log("Calculating...");
    for (let i = 0; i < 1000000000; i++) {} // simulate delay
    return num * 2;
  };

  const calculation = useMemo(() => expensiveCalculation(count), [count]);

  return (

```

```

    <div>
      <h2>Expensive Calculation: {calculation}</h2>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <button onClick={() => setTodos([...todos, "New Todo"])}>Add Todo</
button>

      <ul>
        {todos.map((todo, index) => (
          <li key={index}>{todo}</li>
        ))}
      </ul>
    </div>
  );
}

```

✓ `useMemo` ensures the calculation only runs when `count` changes, not when `todos` change.

👉 Summary Table

Hook	Purpose	Prevents
<code>React.memo</code>	Memoizes entire component	Unnecessary re-render due to unchanged props
<code>useCallback</code>	Memoizes function	Re-creation of function on each render
<code>useMemo</code>	Memoizes computed value	Expensive recalculations

🐱 Exercise (15–20 min)

😓 Task: Optimize a ToDo app

1. Create a ToDo app with these features:
2. Add new tasks
3. Mark tasks as complete
4. Filter tasks (All, Completed, Incomplete)
5. Optimize it using:
6. `React.memo` for task components
7. `useCallback` for event handlers
8. `useMemo` for filtering logic

👉 Bonus Challenge:

Add a button that counts clicks (independent of todos). Ensure that changing the count **does not re-render** the task list.

Key Takeaway

React optimization isn't about making everything memoized — it's about identifying **bottlenecks** and applying these hooks **where re-renders hurt performance**. Overusing them can make your code complex, so use wisely!