React Performance

Preventing unnecessary re-renders with React.memo, useMemo, and useCallback, and measuring render times with React's Profiler component -- with simple examples.

Advanced 12 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/react-performance

---

React Performance

This category is our introduction to React's "advanced" topics -- now that you know the fundamentals (state, hooks, forms, routing, fetching data, sharing state), we can focus on using them MORE EFFICIENTLY. First stop: preventing unnecessary re-renders.

Unnecessary Re-renders

When a parent component re-renders, React by default re-renders ALL of its children too -- even if the child's props haven't changed:

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

function ExpensiveList({ items }) {
  // We added this console.log to make every render of this component
  // "expensive" -- in a real app this could be a large list or a complex
  // calculation.
  console.log("ExpensiveList rendered");

  return (
    <ul>
      {items.map((item) => (
        <li key={item}>{item}</li>
      ))}
    </ul>
  );
}

const items = ["Java", "React", "Spring Boot"];

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

  return (
    <div>
      {/* Every time `count` increases, UnnecessaryRerenderExample
          re-renders -- and by default, React also re-renders EVERY child
          whose parent re-rendered. Even though ExpensiveList's props
          (`items`) never CHANGE, it still re-renders unnecessarily. */}
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <ExpensiveList items={items} />
    </div>
  );
}
```

Every time `count` increases, `ExpensiveList` re-renders even though `items` NEVER changed. This isn't a problem for small components, but in large lists or complex calculations it can cause noticeable slowdowns.

Skipping Unnecessary Renders with React.memo

`memo()` wraps a component so it will NOT re-render unless its props change:

```
import { memo, useState } from "react";

// memo() wraps a component in a "memoized" version -- BEFORE re-rendering,
// React checks whether the props are the SAME as last time; if they are,
// it SKIPS the render.
const ExpensiveList = memo(function ExpensiveList({ items }) {
  console.log("ExpensiveList rendered");

  return (
    <ul>
      {items.map((item) => (
        <li key={item}>{item}</li>
      ))}
    </ul>
  );
});

// We define `items` OUTSIDE the component -- if we put it inside, a NEW
// array would be created on every render, its reference would change, and
// memo would never be able to skip the render (see the next lesson,
// "ReactMemoWithCallbackExample").
const items = ["Java", "React", "Spring Boot"];

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

  return (
    <div>
      {/* When `count` increases, ReactMemoExample re-renders, but since the
          CONTENTS of `items` stayed the same (same array elements), memo
          now SKIPS re-rendering ExpensiveList. */}
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <ExpensiveList items={items} />
    </div>
  );
}
```

Before re-rendering a `memo`-wrapped component, React COMPARES the new props with the props from the last render (a shallow comparison); if they're the same, it skips the render.

memo + useCallback: Function Props

`memo` alone isn't enough -- you need to be careful with function props:

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

const SearchButton = memo(function SearchButton({ onSearch }) {
  console.log("SearchButton rendered");
  return <button onClick={onSearch}>Search</button>;
});

function ReactMemoWithCallbackExample() {
  const [count, setCount] = useState(0);
  const [query, setQuery] = useState("");

  // WITHOUT useCallback, `handleSearch` would be a NEW function on every
  // render -- since functions are values too, memo would treat this as
  // "props changed" and re-render SearchButton anyway. useCallback keeps
  // the SAME function reference as long as `query` doesn't change.
  const handleSearch = useCallback(() => {
    console.log("Searching for:", query);
  }, [query]);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <input value={query} onChange={(event) => setQuery(event.target.value)} />
      <SearchButton onSearch={handleSearch} />
    </div>
  );
}
```

WITHOUT the `useCallback` we saw in the Hooks lesson, a NEW function is created on every render -- since functions are values too, `memo` treats this as "the props changed" and re-renders anyway. `useCallback` keeps the SAME function reference as long as its dependencies haven't changed.

Caching Expensive Calculations with useMemo

Let's revisit the `useMemo` we saw in Hooks, this time directly in a performance context:

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

const courses = ["Java", "React", "Spring Boot", "PostgreSQL", "Docker"];

function sortAlphabetically(list) {
  console.log("sortAlphabetically ran (expensive operation)");
  return [...list].sort();
}

function UseMemoForExpensiveCalculationExample() {
  const [query, setQuery] = useState("");
  const [count, setCount] = useState(0);

  // The result of `sortAlphabetically` is only recomputed when `courses`
  // CHANGES -- even if the component re-renders because `count` changed,
  // since `courses` stayed the same, useMemo returns the CACHED result and
  // does NOT run the sort function again.
  const sortedCourses = useMemo(() => sortAlphabetically(courses), [courses]);

  const filtered = sortedCourses.filter((course) =>
    course.toLowerCase().includes(query.toLowerCase()),
  );

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <input value={query} onChange={(event) => setQuery(event.target.value)} />
      <ul>
        {filtered.map((course) => (
          <li key={course}>{course}</li>
        ))}
      </ul>
    </div>
  );
}
```

`sortAlphabetically` only runs when `courses` CHANGES -- the component re-rendering for some other reason (like `count` changing) does NOT re-trigger this expensive operation.

React DevTools Profiler and <Profiler>

To MEASURE how long a component takes to render, React's own `<Profiler>` component can be used:

```
import { Profiler, useState } from "react";

function onRenderCallback(id, phase, actualDuration) {
  // React calls this function after every render -- `id` is the name we
  // gave the Profiler; `phase` is whether it was a "mount" (initial render)
  // or an "update" (re-render); `actualDuration` is how many milliseconds
  // the render took.
  console.log(`${id} (${phase}) took ${actualDuration.toFixed(2)}ms`);
}

function CourseList({ items }) {
  return (
    <ul>
      {items.map((item) => (
        <li key={item}>{item}</li>
      ))}
    </ul>
  );
}

function ProfilerComponentExample() {
  const [count, setCount] = useState(0);
  const items = ["Java", "React", "Spring Boot"];

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      {/* React's built-in <Profiler> component MEASURES the render time of
          the tree it wraps -- this is the mechanism behind the "Profiler"
          tab in React DevTools. In real apps it's usually added
          temporarily while INVESTIGATING a performance issue, not left in
          as permanent code. */}
      <Profiler id="CourseList" onRender={onRenderCallback}>
        <CourseList items={items} />
      </Profiler>
    </div>
  );
}
```

The `onRender` callback is called every time the wrapped tree renders, giving you the render duration in milliseconds. This is the same mechanism behind the "Profiler" tab in the React DevTools browser extension -- in real applications it's usually added TEMPORARILY while investigating a performance issue, not left in as permanent code.

Summary and Glossary

When a parent re-renders, React by default renders all of its children too; `memo()` lets you skip this as long as props haven't changed -- but it needs to be paired with `useCallback` for function props. `useMemo` caches the result of an expensive calculation as long as its dependencies haven't changed. React's `<Profiler>` component or React DevTools can be used to measure a component's render time.

Glossary

Re-render — A component being rendered again as a result of a state or props change.

Memoization — Storing the result of a calculation or component render and reusing that stored result instead of recomputing it, as long as the inputs haven't changed.