ScopeTreasure Courses

REACT COURSES · LESSON 5.23 · CODE SCOPE SERIES

React Performance

Find the slowness before fixing it: profiling with React DevTools, memo in the right places, lazy-loading heavy screens, and the render habits that waste work.

Introduction

"My app feels slow" is not a bug report — it is a mystery. This lesson is the detective work: **measure** where renders and milliseconds actually go, then apply the smallest fix that removes the waste — with the tools from 5.17 now aimed by evidence instead of guesses. *(Roman Urdu: Pehle naapo, phir theek karo — performance ka kaam andazay se nahi, saboot se hota hai.)*

Key points

- **The Profiler finds the truth:** React DevTools' Profiler tab records an interaction and shows a flame chart of every render with its duration — and its "Why did this render?" note. The slowest bar is your first suspect, not the component you *feel* is guilty.
- **Wasted renders are the usual crime:** a parent re-renders, so every child re-renders — even children whose props never change. The fixes are the 5.17 trio, applied to the profiler's evidence: `memo` on the innocent children, `useCallback` / `useMemo` to keep their inputs stable.
- **State placement is a performance tool:** state held high re-renders everything below it. Pushing state *down* to the components that use it (or out to a store with selectors, 5.21) shrinks every update's blast radius — often the biggest win, and it is free.
  (Roman Urdu: State jitna upar, blast utna bara — state ko wahan rakho jahan woh istemal hota hai.)
- **Code-splitting ships less JavaScript:** `const Admin = lazy(() => import("./Admin"))` + `<Suspense fallback={…}>` splits a heavy screen into its own chunk, downloaded only when visited. First load stops paying for pages most users never open.
- **Lists have their own rules:** thousands of rows rendered at once cost real time; render windows (virtualization) are the advanced answer — but the honest first steps are pagination and asking whether 5,000 rows on one screen serve anyone.
- **Never optimize blind:** every "optimization" costs code complexity. The lesson's law, repeated from 5.17: **profile → fix the measured bottleneck → profile again** to prove the fix worked. An unverified optimization is a rumour.

Code examples

Example 1 — The Profiler session, step by step

```
1. Install "React Developer Tools" (browser extension) and open
   your running app → DevTools → ⚛ Profiler tab.
2. Press Record, do the slow thing (type in the search box,
   click the heavy button), press Stop.
3. Read the flame chart: each bar is one component's render;
   its LENGTH is the time. Find the widest bars.
4. Click a suspicious bar → right panel: "Why did this render?"
   (props changed / parent rendered / hooks changed).
5. Fix from the evidence, then Record the same interaction
   again and compare the totals.
```

What it does: This is the workflow the video performs live on a deliberately slow demo app: typing in one input re-rendered a 2,000-row list on every keystroke. The Profiler did not suggest the fix — it *located* the cost, which is the half of performance work beginners skip.

What you will see: a flame chart where one innocent-looking input's render fans out into the whole list — the smoking gun, in milliseconds, before a single line of code changes.

Example 2 — lazy + Suspense: split the heavy screen out

```
import { lazy, Suspense } from "react";
import { Routes, Route, Link } from "react-router-dom";

const Home = () => <h2>Home</h2>;
// Admin is heavy (charts, editors) — load it only when visited
const Admin = lazy(() => import("./Admin"));

function App() {
  return (
    <div>
      <nav>
        <Link to="/">Home</Link> | <Link to="/admin">Admin</Link>
      </nav>
      <Suspense fallback={<p>Loading screen…</p>}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/admin" element={<Admin />} />
        </Routes>
      </Suspense>
    </div>
  );
}
```

What it does: `lazy` turns the Admin screen into a separate bundle chunk. The initial download — what every visitor pays — excludes it entirely; clicking Admin shows the `Suspense` fallback for the brief moment the chunk travels. Same app, smaller first load, honest fallback.

What you will see: in the network tab, the first load has no Admin code; click Admin and a new chunk (`Admin-….js`) downloads once, the "Loading screen…" flashes, and the page appears — later visits are instant.

Example 3 — The memo fix, aimed by the profiler

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

const Row = memo(function Row({ name, onPick }) {
  return <button onClick={() => onPick(name)}>{name}</button>;
});

function Picker({ names }) {
  const [picked, setPicked] = useState("");
  const [ticks, setTicks] = useState(0);

  // Stable identity (5.17) is what lets memo protect the rows
  const pick = useCallback((name) => setPicked(name), []);

  return (
    <div>
      <button onClick={() => setTicks(ticks + 1)}>Tick {ticks}</button>
      <p>Picked: {picked}</p>
      {names.map((n) => (
        <Row key={n} name={n} onPick={pick} />
      ))}
    </div>
  );
}
```

What it does: The profiler's story for this shape: an unrelated counter (`ticks`) re-rendered every row. The evidence-based fix is exactly 5.17's pair — `memo` on the row plus a `useCallback`-stable handler — and a second profile proves the rows now sit out the ticks entirely.

What you will see: before the fix, render logs per row on every tick; after it, silence from the rows and a profiler total a fraction of the original. Fix and proof, both measured.

Quick recap

- **Profile first:** React DevTools' Profiler (flame chart + "why did this render?") turns "feels slow" into a located cost.
- The common fixes, in order of frequency: **push state down**, **memo + stable props** for innocent children, **lazy/Suspense** to split heavy screens out of first load.
- Every optimization ends with a **second profile** — unmeasured fixes are rumours, and complexity is a real cost.

Common mistakes

1. **Optimizing the wrong component** — polishing the component you *suspect* while the profiler's widest bar sits elsewhere. The fix: the flame chart chooses the target, never vibes.
2. `memo` **everywhere as a superstition** — blanketing an app in memo/useMemo adds comparisons and clutter and can hide the real problem (state placed too high). The fix: apply surgically where the profiler shows unchanged props re-rendering.
3. **Lazy-loading everything** — splitting tiny components creates dozens of waterfall downloads and fallback flashes, slower than one bundle. The fix: split at *route/heavy-feature* boundaries, where the chunk is large and rarely needed on first load.

Practice

1. Profile a real screen from your projects: record one slow interaction, write down (a) the widest bar, (b) its "why did this render" reason, and (c) the total render time. Apply the matching fix from this lesson and record the same interaction again — your homework passes only if the second number is in writing too.
2. Split one heavy page of any earlier project (the full-stack notes app's editor page works well) behind `lazy()` + `Suspense`, and compare the initial bundle: use the build output's chunk list before and after, and quote both sizes.

Practice quiz

1. How do you find the truth about a slow React app?

- By guessing which component feels guilty
- With React DevTools' Profiler: a flame chart of every render with its duration and a "Why did this render?" note; the slowest bar is the first suspect
- By adding memo everywhere
- By counting lines of code

Answer: With React DevTools' Profiler: a flame chart of every render with its duration and a "Why did this render?" note; the slowest bar is the first suspect

2. Why is state placement a performance tool?

- It is not, placement never matters
- State held high re-renders everything below it; pushing state down to the components that use it shrinks every update's blast radius, often the biggest win, and it is free
- High state is always faster
- State placement only affects memory

Answer: State held high re-renders everything below it; pushing state down to the components that use it shrinks every update's blast radius, often the biggest win, and it is free

3. What do lazy and Suspense do for first load?

- They make everything load faster by magic
- They split a heavy screen into its own chunk, downloaded only when visited, so first load stops paying for pages most users never open
- They delete unused components
- They only work for images

Answer: They split a heavy screen into its own chunk, downloaded only when visited, so first load stops paying for pages most users never open

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/react-performance · More free tools and notes at https://scopetreasure.com