React Performance

React.memo, useMemo ve useCallback ile gereksiz yeniden render'ları önlemek, ve React'in Profiler component'i ile render sürelerini ölçmek -- basit örneklerle.

İleri 12 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/react-performance

---

React Performance

Bu kategori, React'in "ileri seviye" konularına giriş yapıyor -- artık temel kavramları (state, hooks, forms, routing, veri çekme, state paylaşımı) bildiğin için, bunları NASIL DAHA VERİMLİ kullanacağına odaklanabiliriz. İlk durak: gereksiz yeniden render'ları önlemek.

Gereksiz Yeniden Render'lar

Bir parent component yeniden render olduğunda, React varsayılan olarak TÜM child'larını da yeniden render eder -- child'ın props'u değişmemiş olsa bile:

```
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>
  );
}
```

`count` her arttığında, `items` HİÇ DEĞİŞMEDİĞİ halde `ExpensiveList` de yeniden render olur. Küçük component'lerde bu sorun olmaz, ama büyük listelerde ya da karmaşık hesaplamalarda fark edilir bir yavaşlığa yol açabilir.

React.memo ile Gereksiz Render'ı Atlamak

`memo()`, bir component'i, props'u değişmediği sürece yeniden render ETMEYECEK şekilde sarmalar:

```
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>
  );
}
```

React, `memo` ile sarmalanmış bir component'i yeniden render etmeden ÖNCE, yeni props'ları bir önceki render'daki props'larla KARŞILAŞTIRIR (sığ/shallow karşılaştırma); aynılarsa render'ı atlar.

memo + useCallback: Fonksiyon Prop'ları

`memo` tek başına yeterli değil -- fonksiyon prop'larıyla dikkatli olmak gerekir:

```
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>
  );
}
```

Hooks dersinde gördüğümüz `useCallback` OLMADAN, her render'da YENİ bir fonksiyon oluşur -- fonksiyonlar da birer değer olduğu için, `memo` bunu "props değişti" sayar ve yine de yeniden render eder. `useCallback`, bağımlılıkları değişmediği sürece AYNI fonksiyon referansını korur.

useMemo ile Pahalı Hesaplamaları Önbelleklemek

Hooks dersinde gördüğümüz `useMemo`'yu, şimdi doğrudan performans bağlamında tekrar görelim:

```
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`, yalnızca `courses` DEĞİŞTİĞİNDE çalışır -- component'in başka bir nedenle (örneğin `count` değiştiği için) yeniden render olması, bu pahalı işlemi TEKRAR TETİKLEMEZ.

React DevTools Profiler ve `<Profiler>`

Bir component'in ne kadar sürede render edildiğini ÖLÇMEK için, React'in kendi `<Profiler>` component'i kullanılabilir:

```
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>
  );
}
```

`onRender` callback'i, sarmalanan ağaç her render olduğunda çağrılır ve render süresini milisaniye cinsinden verir. Bu, React DevTools tarayıcı eklentisindeki "Profiler" sekmesinin arkasındaki mekanizmayla aynıdır -- gerçek uygulamalarda genellikle kalıcı kod olarak değil, bir performans sorununu araştırırken GEÇİCİ olarak eklenir.

Özet ve Terimler Sözlüğü

Bir parent yeniden render olduğunda, React varsayılan olarak tüm child'larını da render eder; `memo()`, props değişmediği sürece bunu atlamayı sağlar -- ama fonksiyon prop'ları için `useCallback` ile birlikte kullanılması gerekir. `useMemo`, pahalı bir hesaplamanın sonucunu, bağımlılıkları değişmediği sürece önbellekte tutar. Bir component'in render süresini ölçmek için React'in `<Profiler>` component'i ya da React DevTools kullanılabilir.

Terimler Sözlüğü

Re-render (Yeniden Render) — Bir component'in, state ya da props değişikliği sonucunda tekrar render edilmesi.

Memoization (Önbellekleme) — Bir hesaplamanın ya da component render'ının sonucunu saklayıp, girdiler değişmediği sürece tekrar hesaplamak yerine bu saklanan sonucu kullanmak.