useMemo & useCallback

Memoization nedir, useMemo ile hesaplama sonucunu ve nesne referansını, useCallback ile fonksiyon referansını önbelleğe almak, ve ne zaman kullanılmaması gerektiği -- basit örneklerle.

Orta 12 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/use-memo-use-callback

---

useMemo & useCallback

Şimdiye kadar `useState`, `useEffect` ve `useRef`'i gördük. Bu ders, iki performans hook'unu -- `useMemo` ve `useCallback` -- anlatıyor. Bu ikisi diğerlerinden biraz farklı: kullanmayı bilmek kadar, NE ZAMAN kullanmaman gerektiğini bilmek de önemli.

Memoization Nedir?

Memoization, bir hesaplamanın sonucunu hafızada tutup, aynı girdilerle tekrar istenirse hesaplamayı TEKRARLAMADAN o sonucu döndürme tekniğidir. React'te component'ler sık sık yeniden render olur; her render'da aynı pahalı hesaplamayı ya da aynı fonksiyonu yeniden oluşturmak, bazı durumlarda gereksiz bir maliyet olabilir -- `useMemo` ve `useCallback` bu maliyeti önlemek için var.

useMemo ile Hesaplama Sonucunu Önbelleğe Almak

`useMemo`, bir hesaplamanın SONUCUNU önbelleğe alır:

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

function slowSquare(n) {
  // Deliberately a SLOW computation -- to simulate a real "expensive
  // calculation".
  let result = 0;
  for (let i = 0; i < 100000000; i++) {
    result = n * n;
  }
  return result;
}

function UseMemoBasicExample() {
  const [number, setNumber] = useState(5);
  const [theme, setTheme] = useState("light");

  // useMemo does NOT re-run slowSquare as long as `number` hasn't changed --
  // it keeps the previous result in memory (memoizes it) and returns that.
  const squared = useMemo(() => slowSquare(number), [number]);

  return (
    <div>
      <p>
        Square of {number}: {squared}
      </p>
      <button onClick={() => setNumber(number + 1)}>Increase Number</button>
      <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
        Toggle Theme ({theme})
      </button>
    </div>
  );
}
```

`useMemo(() => slowSquare(number), [number])`, `number` değişmediği sürece `slowSquare`'i TEKRAR ÇALIŞTIRMAZ -- önceki sonucu döndürür. Temayı değiştirmek component'i yeniden render eder, ama `number` aynı kaldığı için yavaş hesaplama tekrar çalışmaz.

useMemo ile Nesne Referansını Sabit Tutmak

`useMemo`, yalnızca "pahalı" hesaplamalar için değil, bir nesnenin/dizinin REFERANSINI (kimliğini) render'lar arasında sabit tutmak için de kullanılır:

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

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

  // Without useMemo, a NEW object is created on every render -- even if the
  // content is the same, objects from two different renders are NOT
  // considered "equal" (===).
  const config = useMemo(
    () => ({ theme: "dark", fontSize: 16 }),
    [], // No dependencies -- the object is only created on the FIRST render.
  );

  return (
    <div>
      <p>Count: {count}</p>
      <p>
        Theme: {config.theme}, Font Size: {config.fontSize}
      </p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

`useMemo` olmadan, her render'da `{ theme: "dark", fontSize: 16 }` YENİ bir nesne olarak oluşur -- içeriği aynı olsa bile, JavaScript'te iki ayrı nesne `===` ile karşılaştırıldığında eşit sayılmaz. `useMemo`, dependency array değişmediği sürece AYNI nesneyi döndürür.

useCallback ile Fonksiyon Referansını Sabit Tutmak

`useCallback`, `useMemo`'ya çok benzer, ama bir DEĞER yerine bir FONKSİYONU önbelleğe alır:

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

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

  // Without useCallback, handleClick would be created as a NEW function on
  // every render -- functions from two different renders are not considered
  // "equal". useCallback keeps the SAME function reference as long as the
  // dependency array doesn't change.
  const handleClick = useCallback(() => {
    console.log("Clicked, count:", count);
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
      <button onClick={handleClick}>Log to Console</button>
    </div>
  );
}
```

`useCallback` olmadan, her render'da `handleClick` YENİ bir fonksiyon olarak oluşturulur. `useCallback`, dependency array (`[count]`) değişmediği sürece AYNI fonksiyon referansını korur.

Ne Zaman KULLANILMAMALI?

`useMemo` ve `useCallback`'in kendi maliyeti var -- dependency array'i karşılaştırmak ve sonucu hafızada tutmak. Basit, hızlı işlemler için bu maliyet, kazandırdığından daha pahalı olabilir:

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

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

  // UNNECESSARY: This addition is already very fast -- useMemo's own
  // overhead (doing the comparison, keeping the result in memory) can be
  // more expensive than what it saves.
  // const doubled = useMemo(() => count * 2, [count]);

  // BETTER: useMemo isn't needed for simple, fast calculations.
  const doubled = count * 2;

  return (
    <div>
      <p>
        {count} x 2 = {doubled}
      </p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

`count * 2` gibi basit bir işlemi `useMemo` ile sarmalamanın hiçbir faydası yok -- işlem zaten neredeyse anlık. `useMemo` / `useCallback`'i yalnızca gerçekten pahalı bir hesaplama varsa, ya da bir değerin/ fonksiyonun referansının sabit kalması (örneğin başka bir hook'un dependency array'inde kullanılacağı için) gerektiğinde kullan. Aksi halde kodun daha karmaşık, ama daha hızlı OLMAYAN bir hâli ortaya çıkar -- buna "erken optimizasyon" (premature optimization) denir.

Özet ve Terimler Sözlüğü

`useMemo`, bir hesaplamanın sonucunu ya da bir nesnenin referansını önbelleğe alır; `useCallback` aynı şeyi bir fonksiyon için yapar. İkisi de dependency array değişmediği sürece önceki sonucu/referansı döndürür. Basit, hızlı işlemler için bu hook'lara gerek yok -- kendi maliyetleri, çözdükleri sorundan daha büyük olabilir.

Terimler Sözlüğü

Memoization — Bir hesaplamanın sonucunu hafızada tutup, aynı girdilerle tekrar istendiğinde hesaplamayı tekrarlamadan döndürme tekniği.

`useMemo` — Bir hesaplamanın sonucunu ya da bir nesne/dizi referansını memoize eden hook.

`useCallback` — Bir fonksiyon referansını memoize eden hook.

Premature Optimization (Erken Optimizasyon) — Gerçekten gerekmeden, performans kazancı sağlamayan ama kodu karmaşıklaştıran bir optimizasyon yapmak.