useMemo & useCallback

What memoization is, caching a calculation's result and an object reference with useMemo, caching a function reference with useCallback, and when NOT to use them -- with simple examples.

Intermediate 12 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/use-memo-use-callback

---

useMemo & useCallback

So far we've seen `useState`, `useEffect`, and `useRef`. This lesson covers two performance hooks -- `useMemo` and `useCallback`. These two are a bit different from the others: knowing how to use them matters, but knowing WHEN NOT to use them matters just as much.

What Is Memoization?

Memoization is a technique for storing the result of a calculation so that, if it's requested again with the same inputs, you get that result back WITHOUT repeating the calculation. React components often re-render frequently; redoing the same expensive calculation or recreating the same function on every render can sometimes be an unnecessary cost -- `useMemo` and `useCallback` exist to avoid that cost.

Caching a Calculation's Result with useMemo

`useMemo` caches the RESULT of a calculation:

```
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])` does NOT re-run `slowSquare` as long as `number` hasn't changed -- it returns the previous result. Changing the theme re-renders the component, but since `number` stays the same, the slow calculation doesn't run again.

Keeping an Object Reference Stable with useMemo

`useMemo` isn't just for "expensive" calculations -- it's also used to keep an object's/array's REFERENCE (identity) stable across renders:

```
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>
  );
}
```

Without `useMemo`, `{ theme: "dark", fontSize: 16 }` would be a NEW object on every render -- even with identical contents, two separate objects aren't considered equal when compared with `===` in JavaScript. `useMemo` returns the SAME object as long as the dependency array hasn't changed.

Keeping a Function Reference Stable with useCallback

`useCallback` is very similar to `useMemo`, but it memoizes a FUNCTION instead of a value:

```
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>
  );
}
```

Without `useCallback`, `handleClick` would be a NEW function on every render. `useCallback` keeps the SAME function reference as long as the dependency array (`[count]`) hasn't changed.

When NOT to Use Them

`useMemo` and `useCallback` have their own cost -- comparing the dependency array and holding onto the result. For simple, fast operations, that cost can be more expensive than what it saves:

```
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>
  );
}
```

Wrapping something as simple as `count * 2` in `useMemo` has no real benefit -- the operation is already nearly instant. Use `useMemo` / `useCallback` only when there's a genuinely expensive calculation, or when a value's/function's reference needs to stay stable (for example, because it's used in another hook's dependency array). Otherwise you end up with code that's more complex, but not actually faster -- this is called "premature optimization."

Summary and Glossary

`useMemo` caches the result of a calculation or an object's reference; `useCallback` does the same thing for a function. Both return the previous result/reference as long as the dependency array hasn't changed. For simple, fast operations you don't need these hooks -- their own cost can be bigger than the problem they solve.

Glossary

Memoization — A technique for caching a calculation's result and returning it, without repeating the calculation, when requested again with the same inputs.

`useMemo` — The hook that memoizes a calculation's result or an object/array reference.

`useCallback` — The hook that memoizes a function reference.

Premature Optimization — Making an optimization that isn't really needed, doesn't provide a real performance gain, but adds complexity to the code.