Understanding Memoization in React

useCallback, useMemo, and React.memo Explained

The Confusion

"Why do we need useCallback, useMemo, AND React.memo? They all seem to do similar things. When do I use each one?"

This is one of the most confusing React optimization techniques. Let's break down each one and show exactly when and why to use them.

What is Memoization?

Memoization = Remembering the result of an expensive operation and reusing it instead of recalculating.

Think of it like:

- **Without memoization:** Solving the same math problem from scratch every time
- **With memoization:** Writing down the answer the first time, then just reading it

---

The Core Problem: Unnecessary Re-renders

React's Default Behavior

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

  // ⚠️ NEW function created on EVERY render
  const handleClick = () => {
    console.log('clicked');
  };

  // ⚠️ NEW object created on EVERY render
  const style = { color: 'red' };

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <ExpensiveChild onClick={handleClick} style={style} />
    </div>
  );
}
```

What happens:

1. User clicks button
2. `count` updates (0 → 1)
3. `Parent` re-renders
4. **NEW** `handleClick` function created (different reference!)
5. **NEW** `style` object created (different reference!)
6. React sees `ExpensiveChild` got new props
7. `ExpensiveChild` re-renders **even though nothing actually changed!**

The Three Memoization Tools

Tool	What it Memoizes	When to Use
useCallback	Functions	When passing functions to child components
useMemo	Values/Objects	When computing expensive values
React.memo	Entire Component	When component is expensive to render

---

1. useCallback - Memoizing Functions

The Problem Without useCallback

```
function NutritionProvider({ children }) {
  const [state, dispatch] = useReducer(reducer, initialState);

  // ❌ NEW function created on EVERY render
  const addEntry = async (entry) => {
    // ... implementation
  };

  const removeEntry = async (id) => {
    // ... implementation
  };

  // Child components get NEW functions every time
  // They think props changed → they re-render!
  return (
    <NutritionContext.Provider value={{ state, addEntry, removeEntry }}>
      {children}
    </NutritionContext.Provider>
  );
}
```

Problem:

- Every time `state` updates, **NEW** `addEntry` and `removeEntry` functions are created
- Children using these functions think they got new props
- Unnecessary re-renders cascade through the app

The Solution With useCallback

```
function NutritionProvider({ children }) {
  const [state, dispatch] = useReducer(reducer, initialState);

  // ✅ SAME function reference unless dependencies change
  const addEntry = useCallback(async (entry) => {
    const tempId = `temp-${Date.now()}`;
    dispatch({ type: "ADD_ENTRY", payload: { ...entry, _id: tempId } });

    try {
      const created = await createNutritionEntry(entry);
      dispatch({ type: "UPDATE_ENTRY", payload: created });
    } catch (error) {
      dispatch({ type: "REMOVE_ENTRY", payload: tempId });
      throw error;
    }
  }, []); // ← Dependencies: Empty = never recreate

  const removeEntry = useCallback(async (id) => {
    // ... implementation
  }, [state.entries]); // ← Dependencies: Recreate only when entries change

  return (
    <NutritionContext.Provider value={{ state, addEntry, removeEntry }}>
      {children}
    </NutritionContext.Provider>
  );
}
```

Benefit:

- `addEntry` function is created ONCE
- Same reference every render (unless dependencies change)
- Children don't re-render unnecessarily

---

2. useMemo - Memoizing Values

The Problem Without useMemo

```
function NutritionList() {
  const { state } = useNutrition();

  // ❌ Recalculated on EVERY render (even if entries didn't change!)
  const stats = {
    count: state.entries.length,
    totalCalories: state.entries.reduce((sum, e) => sum + e.calories, 0),
    totalProtein: state.entries.reduce((sum, e) => sum + (e.protein || 0), 0),
    totalCarbs: state.entries.reduce((sum, e) => sum + (e.carbs || 0), 0),
    totalFats: state.entries.reduce((sum, e) => sum + (e.fats || 0), 0),
  };

  return <div>{stats.totalCalories} total calories</div>;
}
```

Problem:

- If `state.isLoading` changes (entries stay same), this component re-renders
- All those `.reduce()` calculations run again
- For 100 entries, that's 400+ operations **for no reason!**

The Solution With useMemo

```
function NutritionList() {
  const { state } = useNutrition();

  // ☑ Only recalculated when entries actually change!
  const stats = useMemo(() => {
    const totalCalories = state.entries.reduce((sum, e) => sum + e.calories, 0);
    const totalProtein = state.entries.reduce((sum, e) => sum + (e.protein || 0),
0);
    const totalCarbs = state.entries.reduce((sum, e) => sum + (e.carbs || 0), 0);
    const totalFats = state.entries.reduce((sum, e) => sum + (e.fats || 0), 0);

    return {
      count: state.entries.length,
      totalCalories: Math.round(totalCalories),
      totalProtein: Math.round(totalProtein),
      totalCarbs: Math.round(totalCarbs),
      totalFats: Math.round(totalFats),
    };
  }, [state.entries]); // ← Only recalculate when entries change

  return <div>{stats.totalCalories} total calories</div>;
}
```

Benefit:

- Expensive calculation runs once
- Result is cached
- Only recalculates when `state.entries` actually changes
- If `state.isLoading` changes, cached result is reused

---

3. React.memo - Memoizing Components

The Problem Without React.memo

```
function NutritionList() {
  const { state } = useNutrition();

  return (
    <div>
      {state.entries.map((entry) => (
        <NutritionCard key={entry._id} entry={entry} />
      ))}
    </div>
  );
}

// ❌ Re-renders ALL cards when ANY entry changes
function NutritionCard({ entry }) {
  console.log('Rendering card for', entry._id);
  return (
    <div>
      <h3>{entry.mealType}</h3>
      <p>{entry.calories} calories</p>
    </div>
  );
}
```

Problem:

- User adds a new entry
- ALL existing cards re-render
- If you have 50 entries, that's 50 unnecessary re-renders!

Scenario:

```
Before: [Entry1, Entry2, Entry3]
After:  [NewEntry, Entry1, Entry2, Entry3]

Without memo:
  ❌ NewEntry renders (correct)
  ❌ Entry1 renders (unnecessary!)
  ❌ Entry2 renders (unnecessary!)
  ❌ Entry3 renders (unnecessary!)
```

The Solution With React.memo

```
// ✅ Only re-renders if entry prop actually changes
const NutritionCard = React.memo(({ entry }) => {
  console.log('Rendering card for', entry._id);
  return (
    <div>
      <h3>{entry.mealType}</h3>
      <p>{entry.calories} calories</p>
    </div>
  );
});
```

Benefit:

- React compares old props vs new props
- If props are the same (same entry), skip re-render
- Only re-renders the cards that actually changed

Scenario with memo:

```
Before: [Entry1, Entry2, Entry3]
After:  [NewEntry, Entry1, Entry2, Entry3]

With memo:
  ✅ NewEntry renders (correct)
  ✅ Entry1 SKIPPED (props same!)
  ✅ Entry2 SKIPPED (props same!)
  ✅ Entry3 SKIPPED (props same!)
```

Advanced: Custom Comparison Function

```
// ✅ Custom comparison for more control
const NutritionCard = React.memo(
  ({ entry }) => {
    return <div>...</div>;
  },
  (prevProps, nextProps) => {
    // Return TRUE to SKIP re-render
    // Return FALSE to DO re-render
    return prevProps.entry._id === nextProps.entry._id;
  }
);
```

How They Work Together

From Your Actual Code:

```
// NutritionProvider.tsx
export default function NutritionProvider({ children, initialEntries }) {
  const [state, dispatch] = useReducer(reducer, { entries: initialEntries });

  // 1️⃣ useCallback - Memoize functions
  const addEntry = useCallback(async (entry) => {
    // ... implementation
  }, []);

  const removeEntry = useCallback(async (id) => {
    // ... implementation
  }, [state.entries]);

  // 2️⃣ useMemo - Memoize the context value object
  const contextValue = useMemo(
    () => ({ state, dispatch, addEntry, removeEntry }),
    [state, addEntry, removeEntry]
  );

  return (
    <NutritionContext.Provider value={contextValue}>
      {children}
    </NutritionContext.Provider>
  );
}

// NutritionList.tsx
export default function NutritionList() {
  const { state } = useNutrition();

  // 3️⃣ useMemo - Memoize expensive calculations
  const stats = useMemo(() => {
    return {
      totalCalories: state.entries.reduce((sum, e) => sum + e.calories, 0),
      // ... more calculations
    };
  }, [state.entries]);

  return (
    <div>
      {state.entries.map((entry) => (
        // 4️⃣ React.memo - Memoize individual cards
        <NutritionCard key={entry._id} entry={entry} />
      ))}
    </div>
  );
}

// NutritionCard.tsx
// 4️⃣ React.memo - Component only re-renders if entry changes
const NutritionCard = React.memo(({ entry }) => {
  return <div>{entry.mealType}</div>;
});
```

Visual Flow

Without Memoization

```
User adds entry
    ↓
State updates
    ↓
Provider re-renders
    ↓
NEW addEntry function created ❌
NEW removeEntry function created ❌
NEW context value object created ❌
    ↓
ALL children re-render ❌
    ↓
NutritionList re-renders
    ↓
Stats recalculated (even though same entries) ❌
    ↓
ALL 50 NutritionCards re-render ❌
    ↓
😰 Performance issues with large lists
```

With Memoization

```
User adds entry
    ↓
State updates
    ↓
Provider re-renders
    ↓
useCallback: SAME addEntry function ✅
useCallback: SAME removeEntry function ✅
useMemo: NEW context value (state changed) ✅
    ↓
Children re-render (context value changed)
    ↓
NutritionList re-renders
    ↓
useMemo: Stats recalculated (entries changed) ✅
    ↓
React.memo: Only NEW card renders ✅
React.memo: 49 existing cards SKIP ✅
    ↓
😊 Great performance!
```

When to Use Each One

useCallback

Use when:

- ✅ Passing functions to child components
- ✅ Functions are dependencies in useEffect/useMemo
- ✅ Optimizing Context API providers

Don't use when:

- ❌ Function is only used in the same component
- ❌ Component doesn't have performance issues
- ❌ Premature optimization

useMemo

Use when:

- ✅ Expensive calculations (loops, filters, reduces)
- ✅ Creating objects/arrays passed to children
- ✅ Preventing referential inequality issues

Don't use when:

- ❌ Simple calculations (addition, string concat)
- ❌ No performance problems
- ❌ Values change every render anyway

React.memo

Use when:

- ✅ Component renders often with same props
- ✅ Component is expensive to render
- ✅ List items
- ✅ Large component trees

Don't use when:

- ❌ Props change frequently anyway
- ❌ Component is simple/fast
- ❌ No measured performance issue

Dependencies Array Explained

Both `useCallback` and `useMemo` have a dependencies array. This is crucial!

```
// ❌ WRONG: Missing dependency
const addEntry = useCallback((entry) => {
  console.log(state.entries); // Using state.entries but not in deps!
  // ...
}, []); // BUG: Will use stale state!

// ✅ CORRECT: Include all dependencies
```

```
const addEntry = useCallback((entry) => {
  console.log(state.entries);
  // ...
}, [state.entries]); // Will update when entries change
```

Rule: Include EVERYTHING used inside the function!

Real Example: Context Value Object

Without useMemo (Problem)

```
function NutritionProvider({ children }) {
  const [state, dispatch] = useReducer(reducer, initialState);

  // ❌ NEW object created every render
  const contextValue = {
    state,
    dispatch,
    addEntry,
    removeEntry,
  };

  // Children think context changed EVERY render!
  return (
    <NutritionContext.Provider value={contextValue}>
      {children}
    </NutritionContext.Provider>
  );
}
```

Why this is bad:

```
// JavaScript reference equality:
{} === {} // false (different objects!)
[] === [] // false (different arrays!)

// So even if contents are same:
{ state: x } === { state: x } // false!
```

With useMemo (Solution)

```
function NutritionProvider({ children }) {
  const [state, dispatch] = useReducer(reducer, initialState);

  // ☑ SAME object reference unless dependencies change
  const contextValue = useMemo(
    () => ({ state, dispatch, addEntry, removeEntry }),
    [state, addEntry, removeEntry]
  );

  // Children only re-render when something actually changed!
  return (
    <NutritionContext.Provider value={contextValue}>
      {children}
    </NutritionContext.Provider>
  );
}
```

Performance Comparison

Scenario: 100 nutrition entries, user adds one entry

Without Memoization	With Memoization
Provider re-renders ✓	Provider re-renders ✓
Creates 3 new functions	Reuses same functions ✅
Creates new context object	Creates new context object ✓
All children re-render	All children re-render ✓
Recalculates stats (400+ ops)	Stats cached ✅
100 cards re-render	1 card renders ✅
~500 operations	**~100 operations**

Result: 5x performance improvement!

Common Mistakes

Mistake 1: Forgetting Dependencies

```
// ❌ BAD
const calculate = useCallback(() => {
  return state.entries.length * multiplier; // Using multiplier!
}, []); // Missing multiplier in deps!

// ✅ GOOD
const calculate = useCallback(() => {
  return state.entries.length * multiplier;
}, [state.entries, multiplier]); // All dependencies included
```

Mistake 2: Overusing Memoization

```
// ❌ BAD: Unnecessary
const sum = useMemo(() => a + b, [a, b]);
// Just do: const sum = a + b;

// ✅ GOOD: Worth it
const expensiveSum = useMemo(() => {
  return hugeArray.reduce((sum, item) => sum + item.value, 0);
}, [hugeArray]);
```

Mistake 3: Using React.memo on Everything

```
// ❌ BAD: Simple component, changes often
const Button = React.memo(({ onClick, label }) => (
  <button onClick={onClick}>{label}</button>
));

// ✅ GOOD: Complex component, rarely changes
const ExpensiveDataTable = React.memo(({ data }) => (
  // Lots of complex rendering logic
));
```

Testing Memoization

See Re-renders in Console

```
const NutritionCard = React.memo(({ entry }) => {
  console.log('Rendering card:', entry._id);
  return <div>...</div>;
});
```

Without memo: See console spam with every state change **With memo:** See console log only for changed cards

React DevTools Profiler

1. Install React DevTools
2. Open Profiler tab
3. Record interaction
4. See which components re-rendered (blue bars)
5. Optimize the expensive ones

The Mental Model

Think of memoization like **caching**:

useCallback = Caching a Recipe

- Don't rewrite the recipe every time
- Keep the same recipe card
- Only update when ingredients change

useMemo = Caching a Cooked Meal

- Don't recook the same meal
- Serve leftovers if ingredients unchanged
- Only recook when ingredients change

React.memo = Caching a Restaurant's Menu Item

- Don't remake dish if order is identical
- Only remake when order changes
- Saves time in busy restaurant (large list)

The Big Takeaway

> **Don't memoize everything!**
>
> Memoization adds complexity and memory usage. Only use it when:
>
> 1. You measure a performance problem
> 2. The optimization makes sense
> 3. The benefit outweighs the cost

Pattern:

1. Build it without memoization
2. Measure performance (React Profiler)
3. Add memoization to bottlenecks
4. Measure again to confirm improvement

Quick Reference

```
// Function that doesn't need to change
const func = useCallback(() => { ... }, [deps]);

// Expensive calculation
const value = useMemo(() => expensiveCalc(), [deps]);

// Component that shouldn't re-render unnecessarily
const Component = React.memo(({ props }) => { ... });

// Combination: Context provider
```

```
const value = useMemo(
  () => ({ state, func1: useCallback(...), func2: useCallback(...) }),
  [state, func1, func2]
);
```

Summary for Your Presentation

Key Points:

1. **useCallback** = Memoize functions (prevent new function creation)
2. **useMemo** = Memoize values (prevent recalculation)
3. **React.memo** = Memoize components (prevent re-renders)
4. All three work together for optimal performance
5. Don't optimize prematurely - measure first!

Demo Opportunity:

- Show console logs with/without React.memo on NutritionCard
- Show React DevTools Profiler
- Add a console.log in stats calculation to show useMemo benefit
- Explain the Provider's useMemo for context value

Further Reading

- React: useCallback
- React: useMemo
- React: memo
- When to useMemo and useCallback