Custom Hooks with Context Provider Pattern

Understanding Why and How

The Confusion

"Why do we need a custom hook (`useNutrition`) when we're already using Context API? Isn't the Provider enough?"

This is one of the most confusing aspects of the Context + Custom Hook pattern. Let's break it down step by step.

The Two-Part System

Part 1: The Provider (Creates & Stores State)

```
// NutritionProvider.tsx
"use client";

const NutritionContext = createContext(null);

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

  // 🎯 Provider CREATES and MANAGES the state
  return (
    <NutritionContext.Provider value={{ state, dispatch }}>
      {children}
    </NutritionContext.Provider>
  );
}
```

Part 2: The Custom Hook (Accesses State)

```
// Also in NutritionProvider.tsx
export function useNutrition() {
  const context = useContext(NutritionContext);

  if (!context) {
    throw new Error("useNutrition must be used within NutritionProvider");
  }

  // 🎯 Custom hook SAFELY ACCESSES the state
  return context;
}
```

Why Do We Need Both?

Without Custom Hook (Direct useContext)

```
// ❌ In any component - MESSY AND RISKY
import { useContext } from 'react';
import { NutritionContext } from './NutritionProvider';

function AddNutritionEntry() {
  // Problem 1: Have to import useContext AND NutritionContext
  // Problem 2: No safety check
  // Problem 3: Could be null and crash!
  const context = useContext(NutritionContext);

  // Problem 4: Have to check every time
  if (!context) {
    // What do we do here? Return null? Throw error? Inconsistent!
    return null;
  }

  const { state, addEntry } = context;
  // ... rest of component
}
```

With Custom Hook (useNutrition)

```
// ☑ In any component - CLEAN AND SAFE
import { useNutrition } from './NutritionProvider';

function AddNutritionEntry() {
  // ONE import, ONE line, ALWAYS safe!
  const { state, addEntry } = useNutrition();

  // No null checks needed - hook handles it
  // No importing useContext
  // No importing NutritionContext
  // ... rest of component
}
```

The Problems Custom Hooks Solve

Problem 1: Safety

Without custom hook:

```
const context = useContext(NutritionContext);
// context could be null if used outside Provider
// App could crash!
```

With custom hook:

```
const { state } = useNutrition();
// Throws clear error if used outside Provider
// Developer knows exactly what went wrong
```

Problem 2: Developer Experience

Without custom hook:

```
// Every component needs these imports:
import { useContext } from 'react';
import { NutritionContext } from './NutritionProvider';

const context = useContext(NutritionContext);
```

With custom hook:

```
// Just one import:
import { useNutrition } from './NutritionProvider';

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

Problem 3: Type Safety (TypeScript)

Without custom hook:

```
// Context could be null
const context = useContext(NutritionContext);
// TypeScript: context is NutritionContextType | null
// Have to check everywhere!
if (context) {
  context.state.entries // Only safe here
}
```

With custom hook:

```

```

```
// Custom hook guarantees non-null
const { state } = useNutrition();
// TypeScript: state is definitely NutritionState
state.entries // Always safe!
```

Problem 4: Consistency

Without custom hook:

```
// Developer A does this:
if (!context) return null;

// Developer B does this:
if (!context) throw new Error("No context!");

// Developer C forgets to check:
const { state } = context; // CRASH!
```

With custom hook:

```
// Everyone uses the same hook
// Same error handling everywhere
// Consistent behavior across the app
```

How It Works (Step by Step)

Step 1: Create Context

```
// This is the "tunnel" that will carry data
const NutritionContext = createContext<NutritionContextType | null>(null);
```

Step 2: Provider Puts Data in the Tunnel

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

  const contextValue = { state, dispatch, addEntry, removeEntry };

  // ⬇️ Put data into the tunnel
  return (
    <NutritionContext.Provider value={contextValue}>
      {children}
```

```
    </NutritionContext.Provider>
  );
}
```

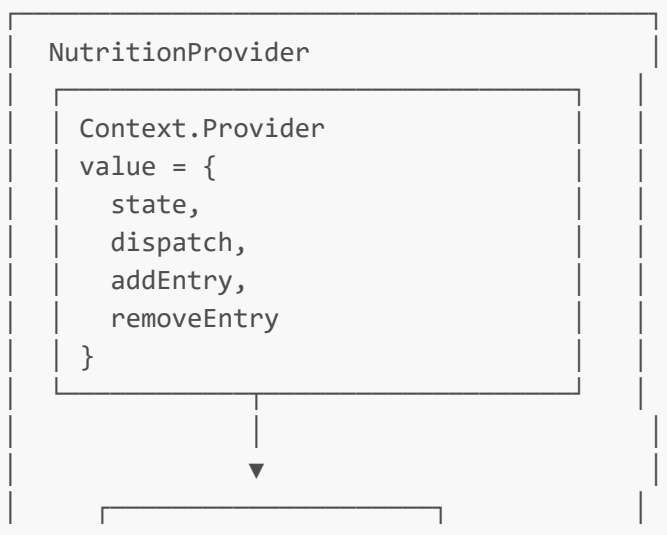
Step 3: Custom Hook Safely Retrieves from Tunnel

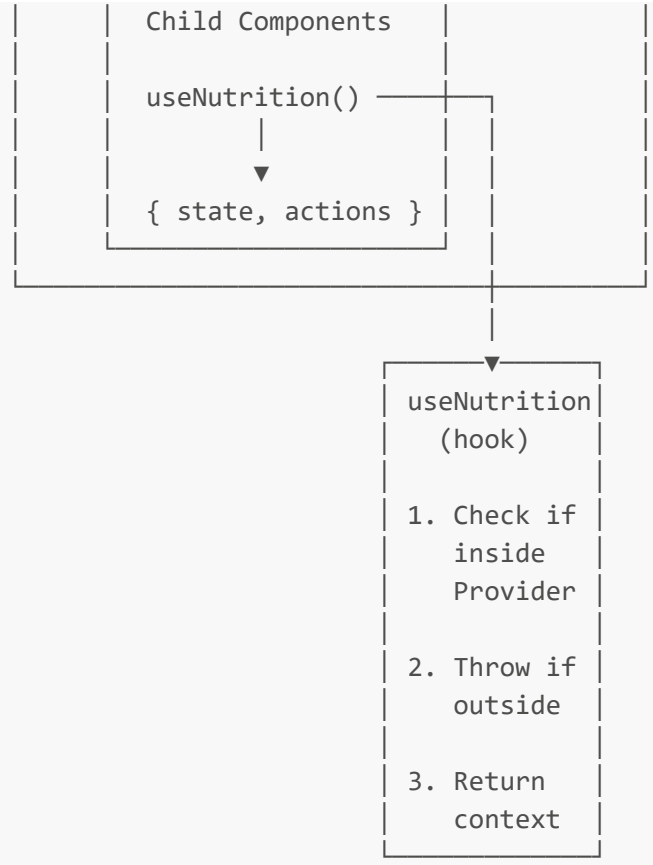
```
export function useNutrition() {
  // ⬇️ Try to get data from tunnel
  const context = useContext(NutritionContext);

  // ⬇️ Check if we're actually inside the tunnel
  if (!context) {
    throw new Error("useNutrition must be used within NutritionProvider");
  }

  // ⬇️ Return the data (guaranteed to exist)
  return context;
}
```

Step 4: Components Use the Hook

```
function AddNutritionEntry() {
  // ⬇️ Get data safely from tunnel
  const { state, addEntry } = useNutrition();

  // ⬇️ Use it!
  return <form onSubmit={() => addEntry(data)}>...</form>;
}
```

Visual Flow





Real Code from Your App

The Provider (Creates State)

```
// NutritionProvider.tsx
export default function NutritionProvider({ children, initialEntries }) {
  // 🏗️ BUILD the state management
  const [state, dispatch] = useReducer(nutritionReducer, {
    entries: initialEntries ?? [],
    isLoading: false,
    error: null,
  });

  // 🔨 CREATE helper functions
  const addEntry = useCallback(async (entry) => {
    // ... implementation
  }, []);

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

  // 📦 PACKAGE everything together
  const contextValue = useMemo(
    () => ({ state, dispatch, addEntry, removeEntry }),
    [state, addEntry, removeEntry]
```

```
  );

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

The Custom Hook (Safe Access)

```
// Also in NutritionProvider.tsx
export function useNutrition() {
  // 🔍 TRY to access context
  const context = useContext(NutritionContext);

  // ⚠️ VALIDATE we're inside provider
  if (!context) {
    throw new Error("useNutrition must be used within NutritionProvider");
  }

  // ☑️ RETURN safe, guaranteed-to-exist context
  return context;
}
```

Usage in Components (Simple!)

```
// AddNutritionEntry.tsx
function AddNutritionEntry() {
  // 🎯 ONE LINE to get everything!
  const { state, addEntry } = useNutrition();

  // ✨ Use it immediately - no null checks!
  const handleSubmit = async (data) => {
    await addEntry(data);
  };

  return <form onSubmit={handleSubmit}>...</form>;
}
```

Common Questions

Q: "Why not just export the Context and use useContext directly?"

A: You *could*, but then:

- Every component has to import `useContext` AND `NutritionContext`
- Every component has to check for null
- Inconsistent error handling across your app
- More code in every component
- TypeScript thinks it could be null everywhere

Q: "Is the custom hook just for convenience?"

A: No! It's for:

- **Safety** - Prevents crashes from null context
- **Consistency** - Same error message everywhere
- **Type Safety** - TypeScript knows it's never null
- **Developer Experience** - Less code, clearer intent
- **Maintainability** - Change logic once, affects all components

Q: "Could I use the custom hook without a Provider?"

A: Try it and see what happens:

```
// Without Provider
function SomeComponent() {
  const { state } = useNutrition();
  // ❌ Error: "useNutrition must be used within NutritionProvider"
}
```

This is **good**! It tells you exactly what's wrong.

Q: "What if I forget to use the hook and use useContext directly?"

A: Nothing stops you, but:

```
// Bypassing the custom hook
const context = useContext(NutritionContext);
// You lose all the safety checks
// You have to handle null yourself
// You lose the nice error message
```

The Pattern in Action

File Structure

```
NutritionProvider.tsx
├─ NutritionContext (private - not exported)
├─ NutritionProvider (exported - wrap components)
└─ useNutrition (exported - access in components)
```

Usage Flow

```
// 1️⃣ Page wraps with Provider
<NutritionProvider initialEntries={data}>
  <AddNutritionEntry />
  <NutritionList />
</NutritionProvider>

// 2️⃣ Components use custom hook
function AddNutritionEntry() {
  const { addEntry } = useNutrition();
  // ...
}

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

Benefits Recap

☑ For Developers

- One import instead of two
- One line instead of multiple
- No null checks needed
- Clear error messages
- Better autocomplete

☑ For Code Quality

- Consistent error handling
- Type safety
- Less boilerplate
- Single source of truth
- Easier to refactor

☑ For Users

- Fewer bugs (can't forget null checks)
- Better error messages in console
- More reliable app

Anti-Pattern vs. Best Practice

❌ ANTI-PATTERN: Export Context, Use Directly

```
// Provider.tsx
export const NutritionContext = createContext(null);

// Component.tsx
import { useContext } from 'react';
import { NutritionContext } from './Provider';

function MyComponent() {
  const context = useContext(NutritionContext);
  if (!context) {
    // Every component does this differently!
    return <div>Error!</div>;
  }
  // ...
}
```

☑ BEST PRACTICE: Custom Hook

```
// Provider.tsx
const NutritionContext = createContext(null); // Private!

export function useNutrition() {
  const context = useContext(NutritionContext);
  if (!context) throw new Error("...");
  return context;
}

// Component.tsx
import { useNutrition } from './Provider';

function MyComponent() {
  const { state } = useNutrition();
  // Clean, safe, simple!
}
```

TypeScript Perspective

Without Custom Hook

```
type NutritionContextType = {
  state: NutritionState;
  addEntry: (entry: Entry) => void;
};
```

```
const NutritionContext = createContext<NutritionContextType | null>(null);

// In component:
const context = useContext(NutritionContext);
// Type: NutritionContextType | null  ⚠

// Have to narrow the type:
if (context) {
  context.state.entries; // OK here
}
```

With Custom Hook

```
export function useNutrition(): NutritionContextType {
  const context = useContext(NutritionContext);
  if (!context) throw new Error("...");
  return context; // Type: NutritionContextType ☑
}

// In component:
const { state } = useNutrition();
// Type: NutritionContextType (never null!) ☑
state.entries; // Always safe!
```

Mental Model

Think of it like a bank:

The Provider = The Bank Vault

- Stores the money (state)
- Manages deposits/withdrawals (actions)
- Only one per branch (one per route)

The Custom Hook = The ATM

- Safe, standardized way to access your money
- Checks if you're a valid customer (inside Provider)
- Clear error if something's wrong
- Same interface everywhere

Direct useContext = Breaking into the vault

- You *could* do it
- But why bypass the safety systems?
- More dangerous
- More work

- Not recommended!

The Big Takeaway

> **The custom hook is NOT just convenience.**
>
> It's a safety layer that:
>
> 1. Validates you're using context correctly
> 2. Provides consistent error handling
> 3. Improves type safety
> 4. Creates a better developer experience
>
> **Pattern: Provider manages state, Custom hook accesses it safely.**

Summary for Your Presentation

Key Points:

1. **Provider** creates and manages state
2. **Custom hook** provides safe, validated access to that state
3. **Without custom hook:** Every component handles errors differently
4. **With custom hook:** Consistent, safe, clean access everywhere
5. Think: "Provider = vault, Custom hook = ATM"

Demo Opportunity:

- Show NutritionProvider with useNutrition hook
- Show component using the hook (1 line!)
- Show what happens if you use it outside Provider (clear error!)
- Compare with hypothetical direct useContext approach (messy!)

Further Reading

- React: Writing Custom Hooks
- Kent C. Dodds: How to use React Context effectively
- React TypeScript Cheatsheet: Context