

useContext: Data Without Prop Drilling

The notice board for your app: createContext, the Provider, useContext, a live theme toggle, and the honest caveat about re-renders.

Introduction

Your app's theme, the logged-in user, the language — some data is needed by components scattered all over the tree. Passing it down level by level is **prop drilling**: five components touching a value only the sixth one uses. **Context** is React's notice board: pin the value at the top, and any component below can read it directly with `useContext`. (Roman Urdu: Context aik notice board hai — value upar lagao, neeche jis ko chahiye woh khud parh lega.)

Key points

- **Three steps, always the same:** (1) `createContext()` makes the board, (2) the **Provider** pins a value on it around your app, (3) any child reads it with `useContext(Board)`.
- **The Provider wraps, it does not replace:** `<ThemeContext.Provider value={theme}>` goes around the part of the tree that needs the value — usually the whole `App`.
- **No prop chains:** components between the Provider and the reader do not receive, pass or even know about the value. Their code stays about their own job.
(Roman Urdu: Beech walay components ko kuch pass nahi karna — value seedha neeche pahunchti hai.)
- **The value can be anything:** a string, an object, state + its setter, a whole bundle like `{ user, login, logout }`. Passing state through context is how global state begins.
- **The honest caveat:** when the context value changes, **every** component reading that context re-renders. For a theme or a user that is fine; for data changing sixty times a second, it is the wrong tool (that is what the state libraries in lessons 5.20–5.22 manage better).
- **Context is for shared, app-wide data** — theme, user, language, cart count. Local data (one form's fields) stays in local state; reaching for context for everything is its own mistake.
- **Custom hook wrapper is the pro habit:** a small `useTheme()` hook that calls `useContext` inside — you will build wrappers like this in lesson 5.9.

Code examples

Example 1 — The theme context, start to finish

```
import { createContext, useContext, useState } from "react";

const ThemeContext = createContext("light");

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button className={theme === "dark" ? "btn-dark" : "btn-light"}>Theme: {theme}</button>;
}

function App() {
  const [theme, setTheme] = useState("light");
  return (
    <ThemeContext.Provider value={theme}>
      <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>Toggle theme</button>
      <ThemedButton />
    </ThemeContext.Provider>
  );
}
```

What it does: The board is created once, the Provider pins the current theme state on it, and `ThemedButton` — which received **no props at all** — reads the theme straight from context. When the toggle changes the state, the Provider's value changes and the button re-renders in the other style.

What you will see: a button that flips between its light and dark styling on every toggle click — and `ThemedButton` never received a single prop.

Example 2 — The user context: login state everywhere

```
import { createContext, useContext, useState } from "react";

const UserContext = createContext(null);

function Navbar() {
  const user = useContext(UserContext);
  return <nav>{user ? `Hello, ${user.name}` : "Please log in"}</nav>;
}

function App() {
  const [user, setUser] = useState(null);
  return (
    <UserContext.Provider value={user}>
      <Navbar />
      <button onClick={() => setUser({ name: "Ayesha" })}>Log in as Ayesha</button>
      <button onClick={() => setUser(null)}>Log out</button>
    </UserContext.Provider>
  );
}
```

What it does: The whole app reads one shared truth — who is logged in. The navbar asks context directly; the buttons change the state at the Provider level, and every reader updates at once. No prop was drilled through a single middleman.

What you will see: the navbar starts at "Please log in", switches to "Hello, Ayesha" on login, and back again on log out.

Example 3 — Prop drilling vs context, side by side

```
// Drilling: Layout and Sidebar only carry the parcel, never open it
function AppDrilled() {
  const [lang] = useState("en");
  return <Layout lang={lang} />;
}
function Layout({ lang }) {
  return <Sidebar lang={lang} />;
}
function Sidebar({ lang }) {
  return <p>Language: {lang}</p>;
}

// Context: the middle layers vanish from the chain
function AppContext() {
  const [lang] = useState("en");
  return (
    <LangContext.Provider value={lang}>
      <Layout2 />
    </LangContext.Provider>
  );
}
function Layout2() {
  return <Sidebar2 />;
}
function Sidebar2() {
  const lang = useContext(LangContext);
  return <p>Language: {lang}</p>;
}
```

What it does: The same language value reaches the same sidebar two ways. In the drilled version, `Layout` and `Sidebar` exist in the chain only as couriers. In the context version they go back to their real jobs, and the reader asks the board itself.

What you will see: identical output — "Language: en" — but the second version stays clean when ten more components join the tree.

Quick recap

- **Context = Provider pins a value;** `useContext` **reads it anywhere below** — no couriers, no prop drilling.
- The value can carry state and setters, making context the first form of global state (theme, user, language).
- The price: value changes re-render all readers — perfect for calm shared data, wrong for rapidly changing data.

Common mistakes

1. **Reading context outside the Provider** — a component above (or outside) the `<Provider>` gets the context's default value, silently. The fix: the Provider must wrap every reader; when a value looks stuck at the default, check the wrapping first.
2. **Creating the context inside a component** — `createContext()` written inside `App` builds a NEW board on every render, and readers lose the value. The fix: create contexts at the top of the file, outside all components, once.
3. **Using context for everything** — every local detail pinned on a board means every change re-renders half the app. The fix: local state for local data; context only for what many corners of the tree genuinely share.

Practice

1. Build a `LanguageContext` with values "en" and "ur". A header reads it to show a greeting ("Welcome" / "Khush Aamdeed"), and two buttons inside the Provider switch the language. No component between the header and the buttons may receive the language as a prop.
2. Extend the Example 1 theme: put both `theme` and `setTheme` into the context value as an object, and let a deeply nested `ThemeToggleButton` (inside two middle components that take no props) flip the theme by reading the setter from context.

Watch the Code Scope channel: <https://www.youtube.com/@CodeScopePK> — this lesson's own video joins the channel when it is recorded.

Read online: <https://toolscope.netlify.app/notes/react-usecontext> · More free tools and notes at <https://toolscope.netlify.app>