Context API

React's built-in solution to props drilling -- createContext, useContext, Provider, and wrapping context in a custom hook, with simple examples.

Intermediate 13 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/context-api

---

Context API

In Sharing State, we saw how props drilling becomes a problem in a deep component tree. This lesson covers React's built-in solution to that problem -- the **Context API**.

Creating a Context with createContext

Context is a "box" that carries a value readable from anywhere in the tree, without passing props:

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

// createContext() creates a "box" -- it carries a value that can be read
// from anywhere in the tree, without passing props. The "light" in the
// parentheses is the DEFAULT value used when no Provider exists.
const ThemeContext = createContext("light");

function ThemedButton() {
  // useContext(ThemeContext) reads the `value` of the nearest
  // ThemeContext.Provider -- ThemedButton reads this value directly,
  // WITHOUT receiving it as a prop.
  const theme = useContext(ThemeContext);

  return <button className={theme}>Click me</button>;
}

function CreateContextExample() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedButton />
    </ThemeContext.Provider>
  );
}
```

`createContext("light")` creates a `ThemeContext` -- the value in the parentheses is the DEFAULT value used when there's no Provider. `<ThemeContext.Provider value="dark">` OVERRIDES this value to `"dark"` for the entire tree inside it. `useContext(ThemeContext)` reads the value from the nearest Provider.

Without a Provider: The Default Value

A Provider isn't always required -- without one, `useContext` returns the default value given to `createContext`:

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

const ThemeContext = createContext("light");

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>Click me</button>;
}

function DefaultValueExample() {
  // There's NO Provider here at all -- when ThemedButton calls
  // useContext(ThemeContext), it gets the DEFAULT value ("light") given
  // by createContext("light"). A Provider is only needed when we want to
  // OVERRIDE that value.
  return <ThemedButton />;
}
```

There's no `Provider` at all in this example; `ThemedButton` gets the `"light"` value given via `createContext("light")`. A Provider is only needed when you want to CHANGE that default.

Solving Props Drilling with Context

Let's rewrite the deep tree example from Sharing State, this time with Context:

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

const UserContext = createContext(null);

function Level1() {
  return <Level2 />;
}

function Level2() {
  return <Level3 />;
}

function Level3() {
  return <Level4 />;
}

function Level4() {
  // Unlike the example in the Sharing State lesson, NONE of Level1/Level2/
  // Level3 even knows about the `user` prop -- only Level4, at the bottom,
  // reads DIRECTLY from UserContext.
  const user = useContext(UserContext);
  return <p>Logged in as {user}</p>;
}

function AvoidingPropsDrillingExample() {
  return (
    <UserContext.Provider value="Ada">
      <Level1 />
    </UserContext.Provider>
  );
}
```

NONE of `Level1`, `Level2`, `Level3` even know about the `user` prop anymore -- only `Level4`, at the very bottom, reads it DIRECTLY from `UserContext`. We didn't need to pass anything through the intermediate layers.

Carrying State Inside Context

Context doesn't just carry a fixed value -- it can carry the state itself AND the function that updates it:

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

const CartContext = createContext(null);

function AddItemButton() {
  const { addItem } = useContext(CartContext);
  return <button onClick={() => addItem("Course")}>Add to Cart</button>;
}

function CartCount() {
  const { items } = useContext(CartContext);
  return <p>Items in cart: {items.length}</p>;
}

function ContextWithStateExample() {
  // Context doesn't only carry a FIXED value -- here we're giving `value`
  // an object: both the state itself (`items`) and the function that
  // UPDATES it (`addItem`). This is the most common way Context is used
  // in real applications.
  const [items, setItems] = useState([]);

  function addItem(item) {
    setItems([...items, item]);
  }

  return (
    <CartContext.Provider value={{ items, addItem }}>
      <AddItemButton />
      <CartCount />
    </CartContext.Provider>
  );
}
```

`value={{ items, addItem }}` gives the Provider an OBJECT -- both the current `items` list and the `addItem` function that updates it. This is the most common way Context is used in real applications.

Wrapping Context in a Custom Hook

The custom hook pattern from the Hooks lesson pairs very naturally with Context:

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

const ThemeContext = createContext(null);

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

// The pattern from the Custom Hooks lesson continues here: instead of
// spreading useContext across components, we wrap it in a separate hook
// (`useTheme`). This way, the consuming component just calls `useTheme()`
// without ever dealing with the "context" concept -- just like `useState`.
function useTheme() {
  const context = useContext(ThemeContext);
  if (!context) {
    throw new Error("useTheme must be used within a ThemeProvider");
  }
  return context;
}

function ThemeToggleButton() {
  const { theme, setTheme } = useTheme();

  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Current theme: {theme}
    </button>
  );
}

function CustomContextHookExample() {
  return (
    <ThemeProvider>
      <ThemeToggleButton />
    </ThemeProvider>
  );
}
```

`useTheme()` WRAPS `useContext(ThemeContext)` to expose a cleaner API -- the component that uses it calls `useTheme()` without ever dealing with the concept of "context," just like a built-in hook. Throwing an error when used outside a Provider catches misuse early.

Summary and Glossary

`createContext()` creates a value readable from anywhere in the tree; a `Provider` OVERRIDES that value for a specific subtree; `useContext()` reads the value from the nearest Provider. Without a Provider, the default value given to `createContext` is used. Context can

carry a fixed value, or state together with the functions that update it -- combined with wrapping `useContext` in a custom hook (like `useTheme()`), this is the most common way to manage what could be called "global" state in React.

Glossary

Context — A React structure that carries a value readable from anywhere in the tree, without passing props.

Provider — The component that sets a Context's value for the entire subtree beneath it.

Practical Project

There's a real, runnable example project that brings together the concepts from this category (Sharing State, Context API): **State Management Demo** -- a course list that can be filtered by search and have courses added to favorites.

It shows keeping the search text in a shared parent (lifting state up) and passing it via props to both the search box and results list, and managing favorite courses with a `FavoritesContext` (createContext + Provider + custom hook) read from different parts of the tree (the course list AND a favorites counter in the header) with NO props drilling at all, all working together. You can download it and run it yourself, and read through the code line by line:

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects
npm install
cd projects/state-management
npm run dev
```

The `react-course-projects` repo uses **npm workspaces** -- `npm install` only needs to run once, at the repo root, and every project folder shares the same dependencies (no separate `node_modules` per folder). If you've already run `npm install` at the root, you can just `cd react-course-projects/projects/state-management` and run `npm run dev`.