**React Context API**

Managing state has always been a critical aspect of making web applications with React. The most basic way to do this is prop drilling. In prop drilling, you pass props around from the parent component to other components that need it, no matter how deeply nested they are.

The problem with prop drilling is that, as the application grows in complexity, passing data through multiple levels of components can become messy, cumbersome, and error-prone.

The React Context API was released in 2018 to avoid prop drilling by simplifying state management and making sharing data across the component tree more efficient and error-free.

This article will explore the Context API, starting from understanding the need for it in React applications, to setting it up and using it effectively. We will also look at common use cases, compare it with other state management solutions, and discuss best practices to ensure you use the Context API to its full potential.

Understanding the Need for Context in React

Let's look at a basic example in which we have a ParentComponent that holds some count state, and you need to pass that state down to a deeply nested GrandchildComponent.

Here's the ParentComponent that holds the state and setState but not using them:

'use client';

**import { useState } from 'react';**

**import ChildComponent from './ChildComponent';**

**const ParentComponent = () => {**

**const [count, setCount] = useState(0);**

**return (**

**<>**

**<div className="text-center mt-3">**

**<h2 className="text-3xl">Parent Component</h2>**

**<small>Not using the count state</small>**

**</div>**

**<ChildComponent count={count} setCount={setCount} />**

**</>**

**);**

**};**

**export default ParentComponent;**

**This is the ChildComponent that does not use the state and the setState too, but still has to take them from the ParentComponent and pass them to the GrandChildComponent that needs them:**

**'use client';**

**import GrandChildComponent from './GrandChildComponent';**

**const ChildComponent = ({ count, setCount }) => {**

**return (**

**<>**

**<div className="text-center mt-3">**

**<h2 className="text-3xl">Child Component</h2>**

**<small>Not Using the count state too</small>**

**</div>**

**<GrandChildComponent count={count} setCount={setCount} />**

**</>**

**);**

**};**

**export default ChildComponent;**

**This is the GrandChildComponent that needs the state and setState, and uses them:**

**'use client';**

**const GrandChildComponent = ({ count, setCount }) => {**

**return (**

**<div>**

**<div className="text-center mt-3">**

**<h2 className="text-3xl">Grandchild Component</h2>**

**<small>Using the count state</small>**

**</div>**

**<div className="text-center">**

**<h3 className="text-2xl">Count is: {count}</h3>**

**<button**

**onClick={() => setCount(count + 1)}**

**className="bg-pink-600 p-2 rounded text-white"**

**>**

**Increase Count**

**</button>**

**</div>**

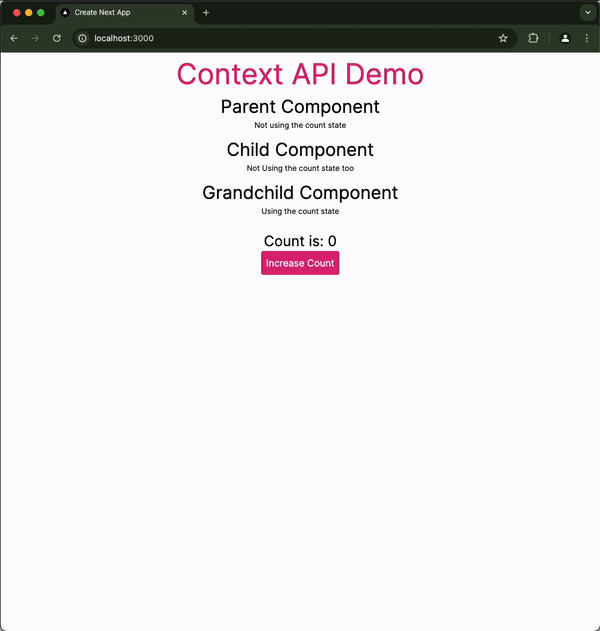
**</div>**

**);**

**};**

**export default GrandChildComponent;**

**And here's what things look like in the browser once the ParentComponent is imported into the Home component of a Next JS project:**

***Prop drilling in React***

**That is prop drilling in action. You can see the ChildComponent that does not use the count state and the setCount still has to absorb both because they'll be used in the GrandChildComponent.**

**That is how you'll continue to pass the state around in the app. So what if you have components that are still deeper in the tree? Like GreatGrandChild and even GreatGreatGrandChild? Why does a parent have to bother their younger generation with their problems?**

**In larger applications, prop drilling can make the code harder to maintain and understand. Each intermediary component must be aware of the props it needs to pass down, even if it does not use them.**

**This is why the Context API exists to prevent this cumbersome prop drilling and make using state in deeply nested components less cumbersome and more straightforward.**

**How Does the Context API Work?**

**The Context API provides a means to share values like state, functions, or any data across the component tree without passing props down manually at every level. This is particularly useful for global data that many components need to access.**

**To start using the Context API, the first thing you need to do is to create a context using the createContext() method. This function returns a context object with two components – a Provider and a Consumer.**

**The Provider is used to wrap the part of your component tree where you want the context to be available. It accepts a compulsory value prop that holds the data you want to share across other components. When the value prop of the Provider changes, all descendants that consume the context will re-render.**

**The Consumer allows any descendant component to use the context. It takes a function as a child, where the function argument is the current context value. In modern React, the useContext hook is often used instead of Consumer for better readability and simplicity.**

**How to Set Up a Context Provider**

**To show you how to set up a context provider, I will use the count state and setCount function from the prop drilling example.**

**Remember the first thing to do is to create a context using the createContext method. I'll do that in a context/counterContext.js file. This is the convention for naming a context file – functionalityContext.js or .ts.**

**These are the full steps for setting up a context provider:**

* **Import createContext and useState from React**
* **Create a CounterContext constant and set it to createContext**
* **Pass in the default values to createContext**
* **Create the CounterProvider component that'll take in children**
* **Define your state and setState**
* **Return a CounterContext.Provider that'll take in the count and setCount as the values of the value prop**
* **Pass in children – it represents everything to be nested when the Context is consumed**

**Export CounterContext and CounterProvider**

**import { createContext, useState } from 'react';**

**const CounterContext = createContext({**

**count: 0,**

**setCount: () => {},**

**});**

**const CounterProvider = ({ children }) => {**

**const [count, setCount] = useState(0);**

**return (**

**<CounterContext.Provider value={{ count, setCount }}>**

**{children}**

**</CounterContext.Provider>**

**);**

**};**

**export { CounterContext, CounterProvider };**

**This same process applies the same way to any context you want to create.**

**How to Consume Context in React Components**

**To consume a context, the first thing you need to do is to import it and wrap it around the app.**

**For our small counter app, you can do that inside the layout file of a Next JS 14 project by importing CounterProvider from the counterContext file and wrapping it around {children} inside the body tag:**

**import { CounterProvider } from '@/context/counterContext';**

**export default function RootLayout({**

**children,**

**}: Readonly<{ children: React.ReactNode }>) {**

**return (**

**<html lang="en">**

**<body className={inter.className}>**

**{/\* Wrap the CounterProvider around the childre \*/}**

**<CounterProvider>{children}</CounterProvider>**

**</body>**

**</html>**

**);**

**}**

**Now, all pages and components will have access to the count state and setCount function.**

**Now, inside the GrandChild component where the count state and setCount function are being used, import useContext from 'react' and CounterContext from the counterContext file, then remove the props.**

**Also, pull out the count state and setCount from the CounterContext you imported like this:**

**const { count, setCount } = useContext(CounterContext);**

**You can leave the count and setCount as they are and things will work fine:**

**'use client';**

**import { useContext } from 'react';**

**import { CounterContext } from '@/context/counterContext';**

**const GrandChildComponent = () => {**

**const { count, setCount } = useContext(CounterContext);**

**return (**

**<div>**

**<div className="text-center mt-3">**

**<h2 className="text-3xl">Grandchild Component</h2>**

**<small>Using the count state</small>**

**</div>**

**<div className="text-center">**

**<h3 className="text-2xl">Count is: {count}</h3>**

**<button**

**onClick={() => setCount(count + 1)}**

**className="bg-pink-600 p-2 rounded text-white"**

**>**

**Increase Count**

**</button>**

**</div>**

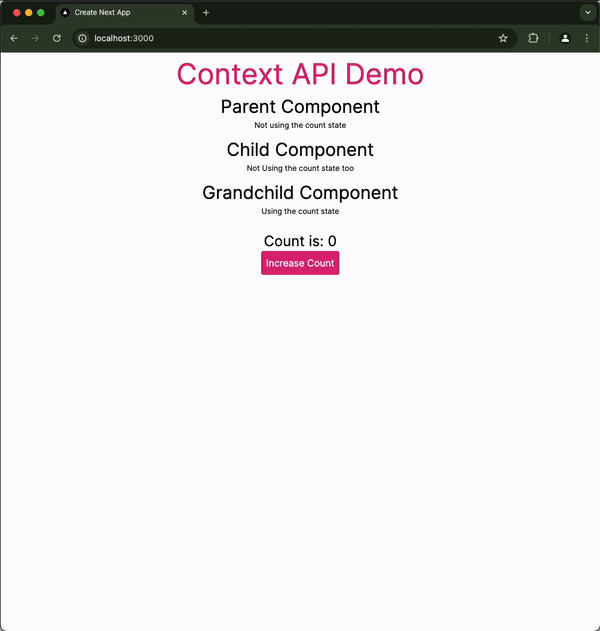
**</div>**

**);**

**};**

**export default GrandChildComponent;**

**Everything still works fine:**

***Context API to mitigate prop drilling in React***

**Common Use Cases for the Context API**

**The Context API is versatile and can be used in various scenarios where managing state and sharing data across multiple components is necessary. Here are some common use cases:**

* **Global state management in medium to large apps: the Context API can handle global state management like cart items in an e-commerce app or the currently playing song in a music app.**
* **Authentication management: using the Context API and other solutions like it to manage the auth state is a common use case for it. States like the current user and auth tokens can be shared across the application using the Context API. This allows any component to access the user authentication status and perform actions like login and logout, or display certain items based on the state.**
* **Theme Management: Another popular use case for the Context API is theme management (dark mode and light mode toggles). You can do this by storing the theme state in a context, and then accessing and updating the theme in any component without having to pass props through multiple layers.**

**Other use cases are localization, user preferences like notification settings, open, close, and toggle states of a modal, API request management, breadcrumbs navigation, step progress, and any other point where the state is involved.**

[**https://v1.scrimba.com/articles/react-context-api/**](https://v1.scrimba.com/articles/react-context-api/)

[**https://www.scaler.com/topics/react/contex-api/**](https://www.scaler.com/topics/react/contex-api/)

**https://www.freecodecamp.org/news/react-context-api-explained-with-examples/**