**useContext:**

🔗 Connect with Me:  
🌐 LinkedIn : https://www.linkedin.com/in/priya-bagde  
📂 GitHub : https://github.com/priya42bagde  
💻 LeetCode : https://leetcode.com/priya42bagde/  
🎥 YouTube Channel : https://youtube.com/channel/UCK1\_Op30\_pZ1zBs9l3HNyBw (Priya Frontend Vlogz)

**What is Prop Drilling:**

When managing data between parent and child components, React gives us the ability to use something known **as props to pass data down from parent to child**. Props can only flow **in one direction**, from parent components to child components (and further down). **Prop Drilling** is a situation where data is passed **down from a parent component to multiple child components** until it has reached its final destination. Prop Drilling not only **complicates our code** but also **consumes a lot of space** and can sometimes be **the root of re-rendering. To avoid this, the concept of Context API was introduced** in **both class and functional components, but consuming code is messy, so best way to use the useContext**.

When **state changes occur on parent elements**, React **will re-render components that depend on those values.** Using props works well in most cases. However, when **working in large applications with a large number of components in the component tree, props can become hard to maintain since props need to be declared in each and every component in the component tree. Context**, in React, **can help make things easier for us in situations**.

**What’s context and Context API(Callback hell):**

Context is a way **to manage state globally**. Context is particularly useful when dealing with **data that is considered “global” or needs to be accessible by many components** within the application. Context in React provides a way to pass data through a component tree **without the need to prop-drill (i.e., pass props down manually at every level).**

**Context API uses Context.Provider and Context.Consumer Components** pass down the data but it is very **cumbersome to write the long functional code** to use this Context API and it will create a callback hell. **So useContext hook helps** to make the code **more readable, less verbose and removes the need to introduce Consumer Component**.

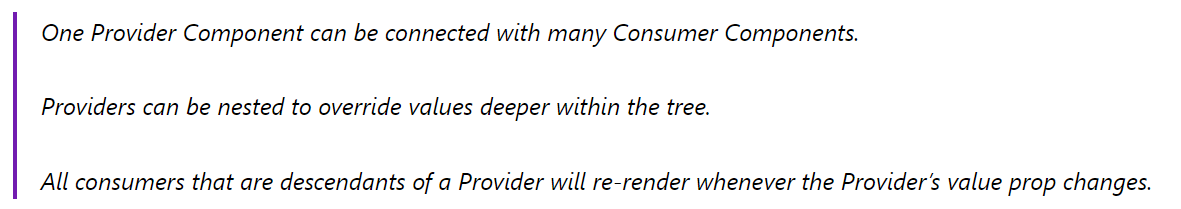
**Why useContext is Important:**

1.Sharing Data Across Components

2.Avoiding Prop Drilling

3.Avoiding callback hell using consumers.

**Key Points:**



**Common Use Cases for useContext:**

1.**User Authentication**: To manage user authentication status and provide user-specific data to components that need it.

2.**Language Localization**: Context is useful for implementing language localization by providing translated strings to components based on the user’s language preference.

3.**Theme Customization**: To apply custom themes to an application, allowing users to personalize the appearance of app.

**Limitations of useContext:**

One of the limitations of useContext is that it **does not have built-in performance optimizations**. When the **value provided by a context provider changes**, all components that **consume the context will re-render, regardless of whether the change is relevant to them**. This can lead to **unnecessary re-renders and negatively impact performance**, especially in large applications with frequent state updates. Developers need to implement their own performance optimizations, such as **using React.memo to prevent unnecessary re-renders of child** **components**.

**Pitfall:**

useContext() always **looks for the closest provider above the component that calls it**. It searches upwards and **does not consider providers** in the component **from which you’re calling useContext().**

**What’s useContext:**

* **As the name suggests, the useContext hook lets us use the Context without a Consumer.**
* useContext **returns the context value** for the context you passed. **To determine the context value, React searches the component tree and finds the closest context provider above for that particular context.**
* The useContext Hook provides functional components access to the context value for a context object. It:

1.Takes the context object (i.e., value returned from React.createContext) as the one argument it accepts.

2.And returns the current context value as given by the nearest context provider.

* A component calling useContext will always re-render when the context value changes**. If re-rendering the component is expensive, you can optimize it by using memoization.**
* **In Class Components, useContext(MyContext) is equivalent to <MyContext.Consumer>.**
* Now compare the two scenarios, one in which we were using Consumer Component to get the data that we were passing using the Provider Component. In the case of the **Consumer Component, we were using render props that were causing a callback hell.**
* But when we are using the useContext hook in place of Consumer Component sharing and receiving of data becomes much **simpler** than it usually was, to begin with, and our code becomes a lot **less messy and complex**.
* The **working of the useContext hook is the same as Consumer Component**. The only difference is in the Consumer component, we have **to create a callback function to capture the value of the prop of the Provider component** and use it in our React application. But in the case of the useContext hook, we no longer have to create that function, we simply **have to pass the context object in the useContext hook** **which will return a value** that will be equal to **the value we have sent to the context using the value prop of the nearest Provider Component** and save it in a new variable which will be used in our app dynamically.
* When the nearest Provider Component updates, this Hook will trigger a re-render with the latest context value passed to that Provider. **Even if an ancestor uses React.memo or shouldComponentUpdate, a re-render will still happen to start at the component itself using useContext.**

**How to use the context:** Using the context in React requires 3 simple steps:

**1.creating the context, 2.providing the context, and 3.consuming the context.**

**1.Creating a Context:**

To create a context in React, we use **the React.createContext method**. This method **returns a context object** that can be **used to provide and consume values** within the component tree. The createContext() function which **converts the local state and props into global state and props** and **stores them inside a separate container globally** so that any child component can access it without actually passing down the line.

**import React, { createContext } from "react";**

**const Context = createContext();**

**2.Providing the context:**

Context.Provider component is **used to provide the context to its child components, no matter how deep they are.** Again, what's important here is that all the components that'd like later to consume the context **have to be wrapped inside the provider component.**

To set the value of context use the **value** prop available on the **<Context.Provider value={value} />.** This is how we’ll provide some initial data. If you want to change the context value, simply update the value prop.

**3.Consuming the context:**

**You can have as many consumers as you want for a single context.** If the context value changes (by changing the value prop of the provider <Context.Provider value={value} />), then all consumers are immediately notified and re-rendered.

**If the consumer isn't wrapped inside the provider**, but still tries to access the context value (using useContext(Context) or <Context.Consumer>), then the **value of the context would be the default value** argument supplied to createContext(defaultValue) factory function that had created the context.

Consuming the context can be performed **in 2 ways.**

* The first way, to use the **useContext() hook**, returns the value of the context**: value = useContext(Context)**. The hook also makes sure **to re-render the component** when the **context value changes**.
* The second way, by using a **render function supplied as a child to Context.Consumer** special component available on the context instance.



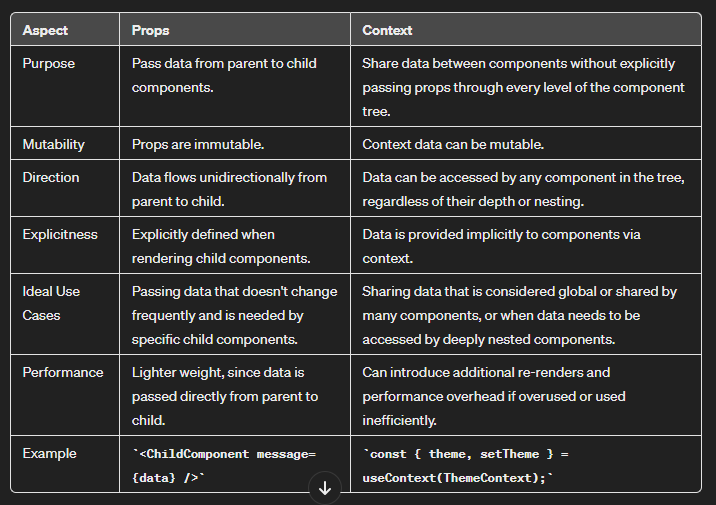
**What is context vs useContext?**

Context is a feature in React that **allows data to be passed down the component tree** **without having to pass props explicitly at every level**. It’s a way to share data between components that are not in a parent-child relationship. **Context is created using the createContext() method**, **which returns an object with a Provider component and a Consumer component.** On the other hand, useContext is a React Hook that provides a way **to consume data from a Provider component in the context API**. It is a more convenient and efficient way **to access data from the Provider component than using the Consumer component**. By using the useContext Hook, a component can subscribe to changes in the context and access the context value without having to wrap itself in a Consumer component.

**What is the difference useContext hook and useState hook?**

* useContext allows you to **access data that is stored in a global context and pass it down through components**, while useState allows you **to store and manage state within a single component**.
* UseContext is most useful when you **need to access global data**, while useState is best **used within the component.**
* useContext allows you **to keep your global state in one place** and have it **update automatically** whenever that state changes, while useState requires that you **manually update** the state within the component whenever the data changes.

**What are the differences between Props and Context:**



**What are the differences between useContext and Redux:**



**When to Use Each(useContext/Redux):**

**useContext:** Consider using useContext **for simple state management** needs or for sharing **state within a specific part of the application**. It is a good choice for **small to medium-sized applications** where state management requirements are straightforward.

**Redux:** Consider using Redux for **complex state management needs**, especially **in large applications with frequent state updates and interactions**. Redux is also a good choice when you need advanced features like **middleware, time-travel debugging, and a centralized store for global state.**

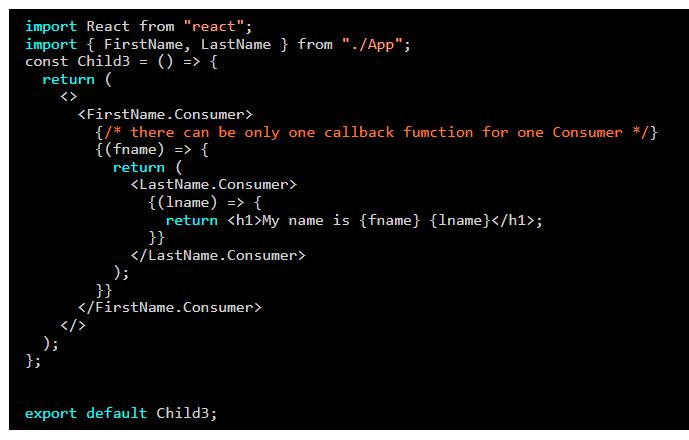
**React Context can also be messy sometimes……………..:**

Trying to find the solution for **prop drilling** so that we can make our code **less complex and messy** and for that, we are **using Context API**. **Let’s say we have to pass another value irrespective of the previous value or object**. We again have to follow the **same steps as in creating a Context with the help of the createContext() function followed by the Provider Component and Consumer Component.**

Let’s understand this concept with the help of the example:



In the above code, we can clearly see that we have **to create a new Provider Component for every CreateContext() function**. **Until now there is no problem, but the complexity starts from the child component as there can only be one callback function, and for every Provider, we have to create a Consumer, then only we will be able to use that passed value.**

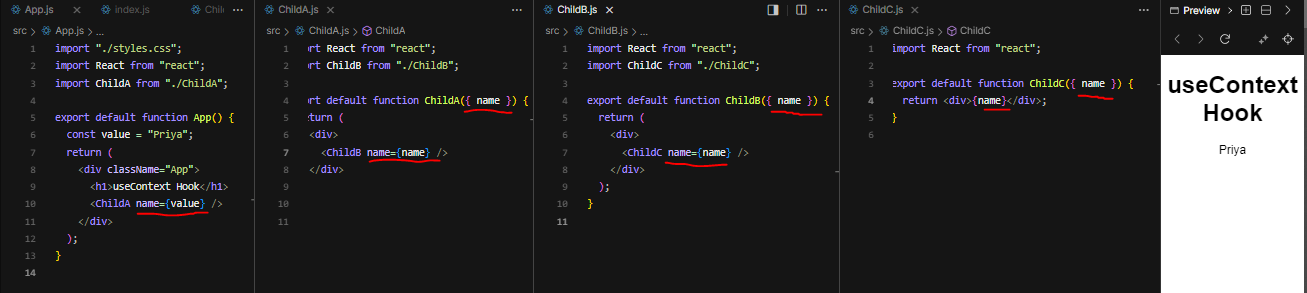


It should be obvious from the above code the **level of complexity we will be facing if this goes on and somehow we have to deal with four or five consumers**. It is as complex as it is messy and can be termed the callback hell in programming terms.

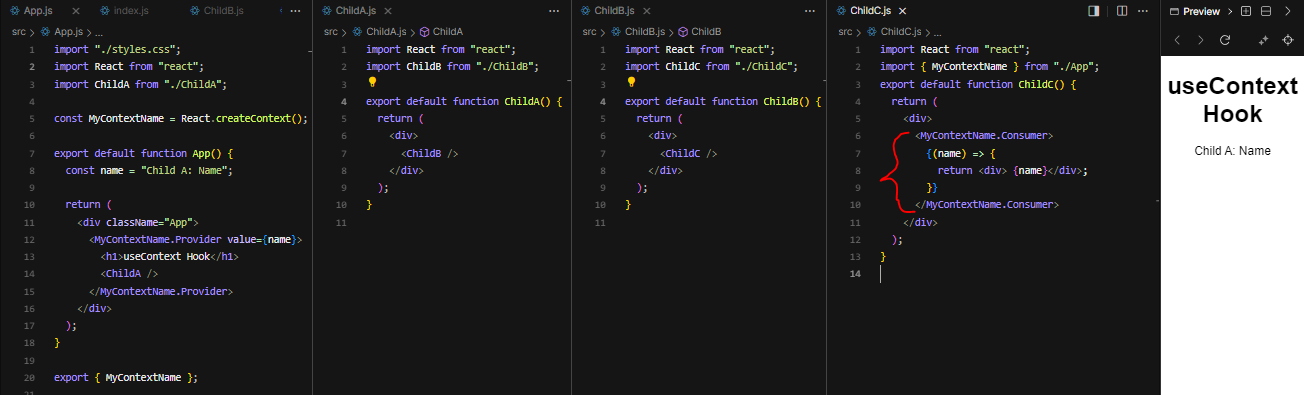
Well, React developers have found a way to use all the functionalities of React Context API while **not falling into this callback hell trap**. **That way is to use the useContext hook** which was introduced in React version 16.8 with other hooks.

**Example:**

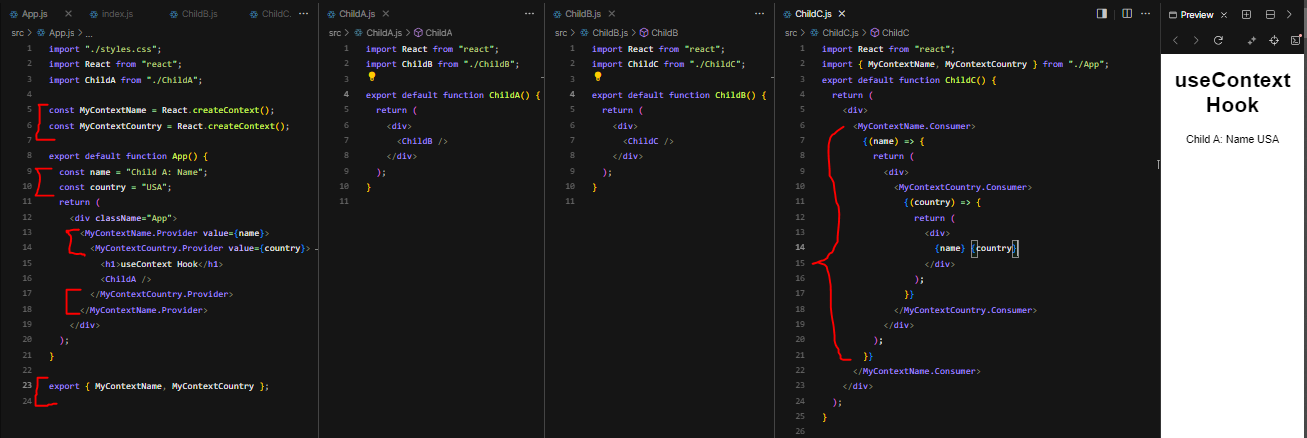
**Prop Drilling:**



**Context API with single context:**



**Context API with multiple context(i.e, callback hell):**



**useContext implementation:**

