

Teach me useContext hook in react

The useContext hook in React is used to access the value of a context within a component. It's a simpler way to consume context compared to using the Context.Consumer component. Here's how it works:

Step-by-step usage:

1. Create a Context:

First, you need to create a context using <code>React.createContext()</code> . This creates a Context object that can hold a value shared across components.

```
import React, { createContext, useState } from 'react';
```

2. Provide Context Value:

Next, you'll use the MyContext.Provider to wrap your component tree or any part of the tree where the context should be available. You pass the value you want to share as a prop to the

```
const App = () => {
 const [value, setValue] = useState("Hello, Context!");
   <MyContext.Provider value={value}>
   </MyContext.Provider>
```

3. Consume the Context:

In any child component that needs access to the ontext, use the useContext hook. Pass the context object (in this case, MyContext) as an argument to useContext .

```
const ComponentA = () => {
 const contextValue = useContext(MyContext);
 return <div>{contextValue}</div>; // Output: Hello, Context!
```

Full Example:

```
import React, { createContext, useState, useContext } from 'react';
 const [value, setValue] = useState("Hello, Vcext!");
    <MyContext.Provider value={value}>
      <ComponentA />
    </MyContext.Provider>
const ComponentA = () => {
  const contextValue = useContext(MyContext);
  return <div>{contextValue}</div>; // Output: Hello, Context!
```

Key Points:

createContext creates a context object.

- useContext allows you to consume context values in functional components.
- The value passed to Provider is the shared data.
- useContext | helps avoid prop drilling, especially when data is needed in deeply nested components.
- ው የ የ የ የ የ C v

does it helps resolving props drilling?

Yes, useContext helps in resolving prop drilling. Prop drilling happens when you pass data through multiple levels of components just to reach a deeply nested child.

Example of Prop Drilling (Without Context)

Here, message is passed through multiple components unnecessarily:

```
const App = () => {
    const message = "Hello, Context!";
    return <Parent message={message} />;
};

const Parent = ({ message }) => <Child message={message} />;

const Child = ({ message }) => <GrandChild message={message} />;

const GrandChild = ({ message }) => <div>{message} </div>;

export default App;
```

The message prop is passed through Parent → Child → GrandChild , even though only GrandChild needs it.

Using useContext (Avoiding Prop Drilling):

With useContext, we can directly access the value from the context without passing it down manually.

Why useContext is Better?

- ✓ Avoids passing props manually through every intermediate component.
- Cleaner and more maintainable code.
- Easy to update context value without modifying multiple components.

So yes, useContext is an effective way to eliminate prop drilling when dealing with global or shared state in your app. ${\mathscr I}$

```
< 2/2
```

Easy to update context value without modifying multiple components

if we are using useContext to reduce passing of props to the component that $\langle \, \, \psi \, \, |$ leeply drilled

then ultimately we are making the process of passing props to the inner components easier.

if we are passing prope and prope are immutable then how we can

```
in we are passing props and props are infiniteable then now we can update context value?

explain this part
```

Good question! You're right that props in React are **immutable**, meaning you cannot directly modify them. However, **context values are not immutable**—they can be updated when stored in a <code>useState</code> or <code>useReducer</code> inside the **Provider**.

How Context Values Are Updated?

Even though props are immutable, we usually pass a state and a setter function inside the Context Provider. The state can be updated using useState or useReducer, which allows the context value to change dynamically.

Example: Updating Context Value with useState

Here's an example where a theme (light/dark) is managed using <code>useContext</code> and updated from a nested component.

1 Create Context & Provider with State

2 Consume Context and Update the Value

Now, in a deeply nested component, we can access and update the theme.

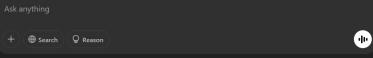
Wrap Your App with the Provider

- The setter function setTheme is passed in context.
- Any component using useContext(ThemeContext) can read (theme) and update (setTheme) the
 value.
- When setTheme is called, React re-renders only the components consuming this context, avoiding unnecessary prop drilling.

Key Takeaways

- 1 Props are immutable, but context values can be managed using useState or useReducer.
- We pass a state and its updater function in the context, making values dynamically changeable.
- This avoids prop drilling and makes the state accessible from deeply nested components.
- Only the components consuming the context re-render, making it efficient.
- Now, instead of manually passing the theme and setTheme props down multiple levels, any child component can directly access and modify the value using useContext(ThemeContext).





ChatGPT can make mistakes. Check important inf