**React Design Patterns: Provider Pattern**

A structured way of managing and passing data through your component tree



Photo by [Clark Van Der Beken](https://unsplash.com/@snapsbyclark?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

The Provider Pattern in React is a design pattern that leverages React’s context API to create a structured way of managing and passing data through your component tree. It helps to avoid prop drilling, where you have to pass data down through multiple layers of components, even if some of them don’t need the data.

**The anatomy of the Provider Pattern**

The structure of the Provider Pattern typically consists of three parts.

**Context**

A file that defines the context variable used by the structure. It’s created using React’s createContext function.

import { createContext } from 'react';  
const UserContext = createContext({state: {}, actions: {}});

**Provider**

The main context provider that wraps whatever children it’s given with a dynamic context. It’s a component that uses the context and provides its value to all components underneath it in the component tree.

import { useState } from 'react';  
  
const UserProvider = ({ children }) => {  
 const [name, setName] = useState('World');  
 const value = {  
 state: { name },  
 actions: { setName },  
 };  
 return (  
 <UserContext.Provider value={value}>  
 {children}  
 </UserContext.Provider>  
 )  
}

**Hook**

A custom hook that gives components access to the current context value. It’s used to consume the context in any component that needs the data.

import { useContext } from 'react';  
import UserContext from './context';  
  
const useUser = () => {  
 return useContext(UserContext);  
}

*Be aware that all components that consume the context will re-render when the context value changes. Therefore, it’s advisable to split your data into different providers according to your logic to enhance the performance.*

**Hands-on**

As you can see, the Provider Pattern can be used with the useContext hook in React. The useContext hook is a built-in hook that allows you to use context without wrapping a component in a Context.Consumer component. **It makes the code cleaner and easier to understand.**

Here is an example of how to use the Provider Pattern with the useContext hook:

import { useState, useContext, createContext } from 'react';  
  
// Step 1: Create a Context  
const ThemeContext = createContext("light", () => "light");  
  
// Step 2: Create a Provider Component  
const ThemeProvider = ({ children }) => {  
 const [theme, setTheme] = useState("light");  
  
 return (  
 <ThemeContext.Provider value={{ theme, setTheme }}>  
 {children}  
 </ThemeContext.Provider>  
 );  
}  
  
export { ThemeContext, ThemeProvider };  
  
// Step 3: Create a Consumer Component  
const UserSettings = () => {  
 const { theme, setTheme } = useContext(ThemeContext);  
  
 // ...  
}  
  
// Step 4: Use the Provider to Wrap Components  
const App = () => {  
 return (  
 <ThemeProvider>  
 <UserSettings />  
 {/\* Any other components that need to use context \*/}  
 </ThemeProvider>  
 );  
}  
  
export default App;

In this way, the Provider Pattern can be used with the useContext hook to provide and consume context in a more straightforward and clear manner.

**Conclusion**

One of the main advantages of the Provider Pattern is that it reduces the risk of introducing bugs when refactoring code. With the Provider Pattern, if you need to rename a prop, you only have to change it in one place — the context provider. Without the Provider Pattern, you would have to find and change every instance of that prop throughout your application, which can be time-consuming and error-prone.

The Provider Pattern also makes it easier to manage global state in your application. You can create a context provider that holds your global state, and then any component that needs to access that state can do so through the context. This makes it easy to keep track of and manage your application’s state.

However, it’s important to use the Provider Pattern judiciously. While it’s great for sharing data across multiple components, overusing it can lead to performance issues. All components that consume the context will re-render when the context value changes. Therefore, it’s advisable to split your data into different providers according to your logic to enhance the performance.