



The correct way to use React Context API

An Overview of The `useContext()` hook

in 5 steps.



techmate.shubham

The **React Context API** is a way for a React app to effectively produce shared state that can be passed around. This is the alternative to “prop drilling” or moving props from grandparent to child to parent, and so on. Context is also touted as an easier, lighter approach to state management using Redux.

1- Create a Context

First, create a context to hold the shared state. This can be done using the `React.createContext` function.

```
// ThemeContext.js

import { createContext } from 'react';

const ThemeContext = createContext();

export default ThemeContext;
```

2- Create a Provider Component

Create a provider component that will wrap the components needing access to the shared state. This component will provide the state and any necessary functions to update it.

```
// ThemeProvider.js

import React, { useState } from 'react';
import ThemeContext from './ThemeContext';

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

  const toggleTheme = () => {
    setTheme(prevTheme => (prevTheme === 'light' ? 'dark' : 'light'));
  };

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

export default ThemeProvider;
```

3- Wrap your App with the Provider

Wrap your entire application (or a part of it) with the **ThemeProvider** to make the context available to all components within the wrapped section.

```
// App.js
import React from 'react';
import ThemeProvider from './ThemeProvider';
import Header from './Header';
import Content from './Content';

const App = () => {
  return (
    <ThemeProvider>
      <Header />
      <Content />
    </ThemeProvider>
  );
};

export default App;
```

4- Consume the Context in Components

Use the `useContext` hook in the components where you want to access the shared state.

```
// Header.js
import React, { useContext } from 'react';
import ThemeContext from './ThemeContext';

const Header = () => {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <header>
      <h1>My App</h1>
      <button onClick={toggleTheme}>
        Toggle Theme ({theme})
      </button>
    </header>
  );
};

export default Header;
```

5- Repeat Step 4 for Other Components

Repeat step 4 for other components that need access to the shared state. Use the `useContext` hook to consume the context and access the state and functions.

By following these steps, you'll replace prop drilling with the useContext hook, making it easier to manage shared state across your React components.

Remember to have a clear understanding of when and where to use context, as it might not be the best solution for every scenario.



Few More Things!!

Overuse

Using Context for every piece of shared state, even if it's only relevant to a small part of the application, can lead to unnecessary complexity. In such cases, it might be better to use local component state.

Performance Concerns

Context can cause re-renders in components that consume the context value whenever the context value changes. If the context is updated frequently, it might lead to performance issues. In such cases, optimizing with memoization techniques or using more specialized state management solutions might be better.

Complex Data Dependencies

If the shared state and its updates involve complex data dependencies or require advanced logic, a more powerful state management library like Redux might be more appropriate.

Readability and Predictability

Overusing context or using it without a clear organizational structure can lead to code that is harder to read and maintain. It's important to strike a balance and ensure that the context is used where it enhances code clarity rather than complicating it.

API Misuse

Using context incorrectly or not following best practices can lead to bugs and unexpected behavior.

