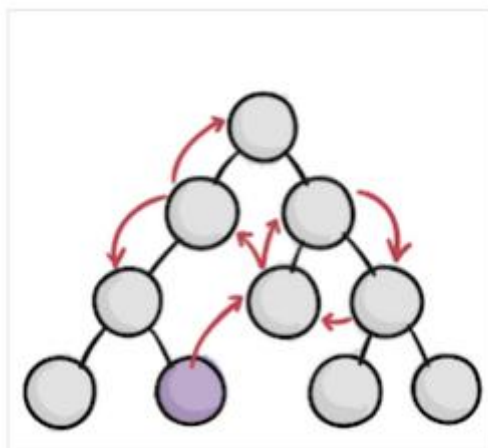


SUMÁRIO

1. TRANSMISSÃO DE ESTADOS ENTRE COMPONENTES.....	1
2. STATES REUNIDOS EM UM ÚNICO LUGAR	1
2.1. UM COMPONENTE PROVEDOR DE ESTADOS.....	2
2.2. CASOS DE USO	2
2.3. COMO CRIAR UM CONTEXT COMPONENT, OU “STATE PROVIDER”	2
2.4. ERRO COMUM: OBJECT UNDEFINED.....	4
3. FONTES.....	6

1. TRANSMISSÃO DE ESTADOS ENTRE COMPONENTES

Como já se sabe, no React existem estados, ou States, e Props, que são dados quaisquer que podem ser transmitidos entre componentes. Entre esses “dados” que podem ser transmitidos entre componentes, estão inclusos os States e suas funções de atualização. A transmissão de States via Props se dá da mesma maneira que quaisquer outros dados. Na invocação do componente a variável de estado e/ou sua função, podem ser passados como atributos que serão recuperados, no componente de destino, por meio do objeto Props. O fluxo, portanto, se dá dessa maneira:



Utilizando Props, dados são passados por meio da invocação, e sempre de um para outro, o que é um processo repetitivo, e que torna a compreensão do fluxo de dados mais complicada.

2. STATES REUNIDOS EM UM ÚNICO LUGAR

Então a prática de transmitir estados entre componentes, um a um, criando um fluxo de dados muito diverso, é problemática. Seria melhor se, ao invés disso, existisse uma possibilidade de **juntar estados “semelhantes” em um só lugar**, para que fossem capturados a partir dele, diretamente.

Pois bem: isso é possível. Trocando estados “semelhantes”, por estados de um mesmo “contexto”, com o hook `useContext`, estados podem ser reunidos em componentes de “contexto de estado”, e a partir deles serem utilizados. Dessa forma, a transmissão de componentes para componente se torna desnecessária.

2.1. UM COMPONENTE PROVEDOR DE ESTADOS

Antes de explicar o passo a passo, **vamos imaginar uma situação em que caiba o `useContext`**. Certo: vamos imaginar que existe um state que varia de um em um, com um clique em um botão. Um clássico contador. Considere que esse state, “count”, foi criado no componente “CounterComponent”, é impresso nele, em uma `<div>`, mas que também deve ser impresso em outra `<div>` e de outro componente, o “CounterMirrorComponent”.

Uma solução “feita nas coxas” para esse caso, seria, ao invés de criar o state do contador no primeiro componente, criar ele no “App.js”, e passa-lo via Props, na invocação de cada componente. Dessa forma, com o state sendo atualizado no “App.js”, e passado via Props, a impressão do seu valor no “CounterComponent”, e no “CounterMirrorComponent”, poderia ser realizada. Realmente funcionaria, mas queremos prezar pelas boas práticas e pela organização da aplicação.

A melhor solução, ao invés dessa, **seria criar um outro componente** que teria esse state, e outros, se necessário, e considera-lo o “**provedor**”, ou “**servidor**” de estados.

2.2. CASOS DE USO

Alguns casos de uso são recorrentes. Por exemplo, um usuário autenticado pode ter o seu nome de usuário utilizado em diversos componentes ao longo da aplicação, e em mais de uma página, com esse hook. O comum estado do modo de visualização da aplicação, ou “tema” alternável, faz sentido no escopo global. Também o idioma da aplicação, se for alterável, é necessário que seja um estado global.

Mas, e deve ser enfatizado isso, na [documentação](#) **é recomendado o uso moderado do `useContext`**, “é usado principalmente quando algum dado precisa ser acessado por muitos componentes em diferentes níveis”.

Ainda, na documentação, outra alternativa é recomendada em casos mais enxutos, e de herança, mas que se quer, ainda assim, evitar o uso de Props, que é a [composição por contenção](#).

2.3. COMO CRIAR UM CONTEXT COMPONENT, OU “STATE PROVIDER”

O primeiro passo consiste em criar uma pasta, em src, chamada “context”, que terá componentes de contexto, isto é, componentes que irão prover acesso global a estados.

O segundo passo consiste, obviamente, em criar um componente de contexto. Considerando aquele problema imaginado, o caminho poderia ser assim: `src/context/CounterContext.js`.

O **terceiro passo**, agora, é sobre a codificação do componente de contexto, e da sua lógica de funcionamento na aplicação. A primeira parte é esta:

```
import React, {useState, useContext, createContext} from 'react';
import { Counter } from '../components/Counter/Counter';

//Cria um objeto Contexto (context)
const CounterContext = createContext();

//Esse é o componente Wrapper, ou "embrulhador", que envolve os componentes que utilizarão estados do contexto
//Por isso ele recebe o objeto Children //Children é o objeto que contém os componentes filhos/embrulhados //Ver no App.js
export function CounterProvider({children}){

  //O state é inicializado aqui, no contexto
  const [Counter, setCounter] = useState(0);

  return(

    //Componentes de um contexto consideram o valor/value do seu provider
    <CounterContext.Provider value = {{Counter, setCounter}}>

      /* Dentro desse objeto, children, estão, então, os componentes <Counter /> e <MirrorCounter /> */
      {children}

    </CounterContext.Provider>

  )
}
```

Vendo esse todo, o importante é notar, após a leitura, a exportação de um **componente de nome “CounterProvider”**, que é o componente que irá embrulhar aqueles que na aplicação irão utilizar os states do contexto, o que explica o porquê de ter o parâmetro “children”. Veja abaixo.

```
function App() {
  return (
    <div className="App">

      /* Esse é o componente "wrapper" do contexto Counter */
      <CounterProvider>

        /* Esses são os componentes filhos do "wrapper" */
        /* Eles são enviados para o wrapper dentro do objeto Children */
        <Counter />
        <MirrorCounter />

      </CounterProvider>

    </div>
  );
}
```

Além disso, é nesse componente, de nome [state_name]Provider(), no caso “CounterProvider()”, que o state é declarado e inicializado, e também, igualmente importante, o seu retorno é a estrutura com os valores do state no contexto.

Por último, é criado um Hook “useCounter()”, personalizado, para acessar os valores do state Counter, criado dentro do componente “CounterProvider()”, e retorna-los na forma de um objeto {state, setState}.

```
//Esse é o hook personalizado, criado para utilizar os valores do contexto //O nome de um hook é sempre use[name]
export function useCounter(){
  //Context recebe o atual valor do contexto CounterContext
  //Esse "atual valor", seria o objeto {Counter, setCounter}, que pode estar com o valor do state diferente
  const context = useContext(CounterContext);

  //O retorno de context é o retornar de um objeto {Counter, setCounter} com valores atualizados
  return context;
}
```

Os componentes embrulhados pelo CounterProvider, que são os que terão acesso aos seus states, irão importar esse Hook criado, e utiliza-lo como se fossem utilizar o useState, mas ao invés de um array [state, setState], será um objeto constante {state, setState}. Veja abaixo o em um dos componentes embrulhados:

```
export function MirrorCounter(){
  //A chamada de useCounter retorna o próprio objeto {Counter, setCounter} com os valores atualizados
  //Para entender o retorno dessa função, veja o componente CounterContext

  //Resgate do state Counter do ContextCounter
  const {Counter, setCounter} = useCounter();
}
```

2.4. ERRO COMUM: OBJECT UNDEFINED

Esse erro vai ocorrer se tentarmos utilizar o Hook personalizado em um componente que não está dentro do wrapper ou provider.

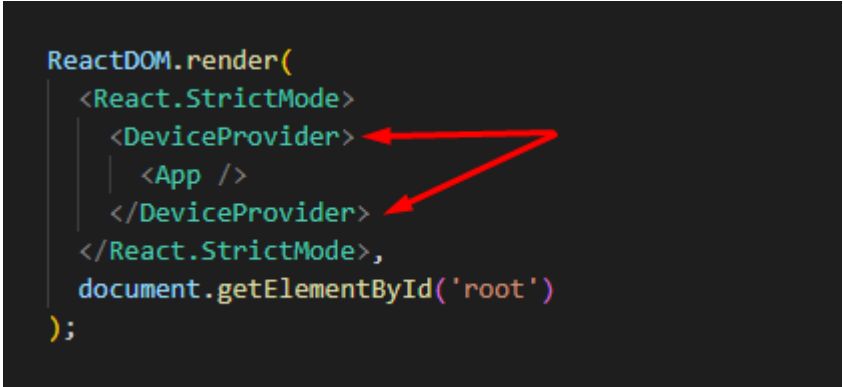
Uma vez tentei utilizar, por exemplo, no App.js um Hook de um Context, ao mesmo tempo que estava utilizando o provider dele no arquivo. Veja abaixo o que quero dizer:

```
const {stateX, setX} = useX();

<StateProvider>
  <Header/>
  <Main state = {stateX} />
</StateProvider>
```

O Hook do state criado em um contexto, está sendo utilizado fora do escopo do wrapper `<StateProvider>`. Isso não é possível, e vai produzir o erro “object undefined”. O uso do Hook criado deve sempre ocorrer dentro do escopo do seu provider.

Assim, uma solução possível para esse caso, em que um contexto deve ser utilizado no `App.js`, por exemplo, é pôr o seu provider no `index.js`, fazendo com que o componente `<App />` exista dentro do seu escopo. Fazendo isso, o próprio `App.js`, como um todo, estará necessariamente dentro do escopo do provider. Veja abaixo uma amostra dessa possibilidade:



```
ReactDOM.render(  
  <React.StrictMode>  
    <DeviceProvider>  
      <App />  
    </DeviceProvider>  
  </React.StrictMode>,  
  document.getElementById('root')  
)
```

The image shows a code snippet in a dark-themed editor. The code is a React render call. It starts with `ReactDOM.render(`, followed by a JSX element. The JSX element has a root tag `<React.StrictMode>`. Inside this tag is another tag `<DeviceProvider>`. Inside `<DeviceProvider>` is the `<App />` component. After `<App />` is the closing tag `</DeviceProvider>`. After the JSX element is a second argument `document.getElementById('root')`. The render call ends with `)`. Two red arrows point from the right side of the code to the `<DeviceProvider>` and `</DeviceProvider>` tags, highlighting the scope of the provider.

3. FONTES

<https://pt-br.reactjs.org/docs/hooks-reference.html#usecontext>

<https://www.youtube.com/watch?v=FsCBw9X9U84&t=623s>