REACT.

MATERIAL DE APOIO

descomplica

React

Context API

Professor: Ayrton Teshima

Tutor: Rodrigo Vasconcelos

Introdução

Como já abordamos anteriormente, o *Context API* vem para resolver um problema que já foi abordado, que é a passagem de propriedades entre componentes. Em um componente super aninhado, como podemos fazer para propagar um valor de uma ponta a outra? A forma é passar de um em um, ou utilizar *children component* para pegar alguns atalhos. Muitas vezes essa passagem não faz sentido, pois a informação que queremos propagar só é utilizada no componente super aninhado.

Objetivos da aula

- Aprofundar o conceito do Context API
- Estabelecer comunicação entre componentes distantes
- Permitir criação de aplicações complexas e sofisticadas através de um recurso da biblioteca fundamental para qualquer desenvolvedor React

Resumo

A API de contexto

Tudo começa quando você importa a função *createContext* do React.

```
import { createContext } from 'react';
```

Fonte: autoral.

Você invoca essa função *createContext* passando o valor inicial do contexto que quer propagar. Ele retorna um objeto que contém dois componentes, que são encarregados de fazer o “teletransporte” das informações.

descomplica

```
import { createContext } from 'react';

const MeuContexto = createContext({});
```

Fonte: autoral.

Esses dois componentes que vem no objeto retornado por *createContext* são: *Provider* e *Consumer*.

Vamos ver primeiro o componente *Provider*, que é o provedor da informação. É nele que você passa a informação que deve ser propagada.

Veja no exemplo abaixo que definimos o componente *Provider*, que é uma propriedade do objeto *MeuContexto*. Nele, passamos a propriedade *value* que é o valor de fato sendo propagado.
Observe que estamos propagando um objeto que tem a propriedade *name* com o valor *'Descomplica'*.

```
import { createContext } from 'react';

const MeuContexto = createContext({});

export const App = () => {
  return (
    <div classname="container">
      <MeuContexto.Provider value={{ name: 'Descomplica' }}>
        <ComponenteA />
      </MeuContexto.Provider>
    </div>
  )
}
```

Fonte: autoral.

Isso significa que todo componente que for descendente do *Provider* consegue acessar a informação propagada. Porém, para acessá-lo, precisamos utilizar o componente *Consumer* do *MeuContexto.*

Vamos definir o componente que vai acessar a informação em outro arquivo (lembre-se de importar ele no arquivo atual). Por causa disso, precisamos exportar o *MeuContexto*, para que nesse novo arquivo, consigamos importar *MeuContexto* e utilizar o componente *Consumer*.

```
export const MeuContexto = createContext({});
```

Fonte: autoral.

Agora vamos definir os componentes e aquele que vai receber os dados passados via Context API.

descomplica

```
const ComponenteZ = () => {
  return (
    <div classname="componente-z">
      <p>O nome é: <span>COLOCAR NOME AQUI</span></p>
    </div>
  )
}

const ComponenteY = () => {
  return (
    <div classname="componente-y">
      <ComponenteZ />
    </div>
  )
}

const ComponenteB = () => {
  return (
    <div classname="componente-b">
      <ComponenteY />
    </div>
  )
}

export const ComponenteA = () => {
  return (
    <div classname="componente-a">
      <ComponenteB />
    </div>
  )
}
```

Fonte: autoral.

Veja que exportamos o *ComponenteA*, que é o utilizado no *App*. Porém, o que queremos é renderizar o valor lá no *ComponenteZ*. Da forma que foi estruturado, a única forma de passar, sem Context API, é passando de componente em componente, do A até o Z, mas, como já sabemos, com Context API conseguimos acessar diretamente.

Para isso, precisamos importar o *MeuContext* e utilizar o componente *Consumer* dele. Vamos focar no *ComponenteZ*, pois os demais não se alteram.

```
import { MeuContexto } from './App'; // Vamos supor que o arquivo App está
na mesma pasta que esse arquivo aqui

const ComponenteZ = () => {
  return (
    <MeuContexto.Consumer>
      {(value) => (
        <div classname="componente-z">
          <p>O nome é: <span>{value.name}</span></p>
        </div>
      )}
    </MeuContexto.Consumer>
  )
}
```

Fonte: autoral.

Nós envolvemos os elementos do *ComponenteZ* no componente *Consumer*. Veja que a sintaxe muda um pouco pois o *Consumer* espera que seu filho seja uma função e não o componente em si.

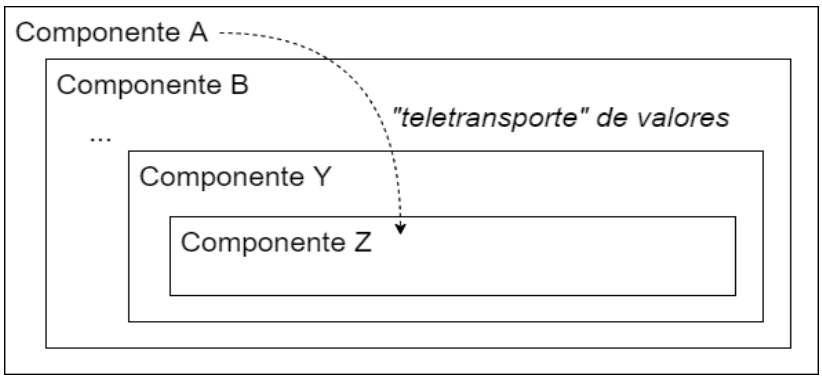
Se você reparar bem, vai notar que o que passamos como filho para o *Consumer* é uma *arrow function.* Como já sabemos, toda expressão JavaScript dentro do JSX fica entre chaves {}. Por isso que iniciamos com chaves. Entre as chaves é uma *arrow function*.

Essa *arrow function* recebe o valor que estamos propagando no *Provider*, que é um objeto. Ou seja, ali definimos o parâmetro chamado *value* e esse *value* é um objeto que tem a propriedade *name*, que é exatamente o que propagamos.

Por isso que na tag *span* nós renderizamos *value.name* e o valor renderizado vai ser “Descomplica”.

Como aplicar na prática o que aprendeu

Se sempre passarmos de um em um, há a chance de acoplarmos informações em componentes de forma desnecessária. Com Context API conseguimos ‘teletransportar’ uma informação de um lugar para o outro, mas sempre de cima para baixo.



Fonte: autoral.

Essas informações podem ser desde valores estáticos, state e funções, basicamente tudo.

Conteúdo bônus

Tópicos avançados

Passando função de estado no contexto

Com o *value* do *Provider* você pode passar praticamente qualquer coisa. No exemplo anterior, passamos um objeto, mas poderia ser outro tipo de valor. Na maioria das vezes você realmente vai passar um objeto pois ele permite passar diversos outros valores contidos.

Vamos fazer algo interessante agora que é propagar uma função que altera o estado do componente atual!

Para isso, vamos usar o *useState* para definir uma informação de estado no componente *App.* Ela será encarregada de definir um *título* para a página.

descomplica

```
import { createContext, useState } from 'react';
import { ComponenteA } from './componentes';

const MeuContexto = createContext({});

export const App = () => {
  const [ titulo, setTitulo ] = useState('React')
  return (
    <div classname="container">
      <h2>{titulo}</h2>
      <MeuContexto.Provider value={{ name: 'Descomplica', setTitulo }}>
        <ComponenteA />
      </MeuContexto.Provider>
    </div>
  )
}
```

Fonte: autoral.

No código acima estamos renderizando o valor de *título* dentro de uma tag *<h2>* e estamos passando a função *setTitulo* no *Provider*. Ou seja, em qualquer lugar que invocar *setTitulo* pelo *Consumer*, deveria alterar o *título* do componente *App.*

Agora basta invocar a função *setTitulo* passada lá no nosso *ComponenteZ.*

```
import { MeuContexto } from './App';

const ComponenteZ = () => {
  return (
    <MeuContexto.Consumer>
      {(value) => (
        <div classname="componente-z">
          <p>O nome é: <span>{value.name}</span></p>
          <button type="button" onClick={() => value.setTitulo('O melhor
conteúdo de React')} />
        </div>
      )}
    </MeuContexto.Consumer>
  )
}
```

Fonte: autoral.

Criamos um botão com evento de clique e passamos para esse evento uma *arrow function* que vai invocar a função *setTitulo* propagada pelo *Provider*. Quando acontecer o clique, o *título* do componente *App* vai ser alterado para "O melhor conteúdo de React".

Hook useContext

Existe um hook no coração do React chamado *useContext.* Ele é um hook que otimiza a sintaxe, deixando quem recebe a informação do *Provider* menos verboso.

Outra grande vantagem desse hook é que você agora também pode ter acesso às informações propagadas pelo *Provider* dentro de outros hooks e não só em componentes.

A implementação do *Provider* se mantém, as mudanças acontecem onde tiver o *Consumer*. Vamos refatorar nosso *ComponenteZ*.

```
import { useContext } from 'react'; // Importe o hook useContext
import { MeuContexto } from './App';

const ComponenteZ = () => {
  const value = useContext(MeuContexto); // Invoque o hook passando o
context
  return ( // Não precisa mais do componente Consumer
    <div classname="componente-z">
      <p>O nome é: <span>{value.name}</span></p>
      <button type="button" onClick={() => value.setTitulo('O melhor
conteúdo de React')} />
    </div>
  )
}
```

Fonte: autoral.

No código acima, removemos o componente *Consumer*, importamos o hook *useContext* e o invocamos passando o contexto em si. O valor retornado é justamente o valor propagado pelo *Provider*.

descomplica

Boa prática com hook useContext

Anteriormente, invocamos o hook *useContext* e o próprio *Context (MeuContexto)* dentro do arquivo que queremos consumir as informações propagadas. Entretanto, como boa prática e convenção de uso do hook *useContext*, o hook é importado já com o contexto embutido, para não precisar fazer duas importações.

Vamos refatorar o código para seguir essa boa prática de context com hook. A primeira etapa é refatorar o local que o *Context* foi criado, que é o nosso arquivo *App.*

```
import { createContext, useState, useContext } from 'react'; // import
useContext
import { ComponenteA } from './componentes';

const MeuContexto = createContext({});

// Exporte função que já retorna o contexto como hook
export const useMeuContexto = () => {
  return useContext(MeuContexto)
}

// Resto continua igual
export const App = () => {
  const [ titulo, setTitulo ] = useState('React')
  return (
    <div classname="container">
      <h2>{titulo}</h2>
      <MeuContexto.Provider value={{ name: 'Descomplica', setTitulo }}>
        <ComponenteA />
      </MeuContexto.Provider>
    </div>
  )
}
```

Fonte: autoral.

Agora já estamos exportando o contexto passando pelo hook *useContext*.

Vamos fazer uma leve refatoração no arquivo do *ComponenteZ* para utilizar o hook com o contexto.

```
import { useMeuContexto } from './App'; // importamos o contexto criado

const ComponenteZ = () => {
  const value = useMeuContexto(); // utilizamos o contexto criado em App
  return (
    <div classname="componente-z">
      <p>O nome é: <span>{value.name}</span></p>
      <button type="button" onClick={() => value.setTitulo('O melhor
conteúdo de React')} />
    </div>
  )
}
```

Fonte: autoral.

A diferença básica é que deixamos de importar o *createContext* do React e passamos a importar o hook já com o contexto, deixando ainda mais enxuto o código.

Múltiplos contextos

Você pode aninhar quantos contextos forem necessários, não é necessário passar tudo de uma só vez em um único contexto.

O interessante é criar os contextos que fazem sentido (quantos forem necessários), até para podermos reutilizar o contexto em outros lugares, pois, se houver um único contexto que propaga tudo junto, ficará difícil de reaproveitá-lo.

Por exemplo, vamos criar um contexto que tem uma única responsabilidade: propagar valores referentes ao estilo da aplicação (como cores, fontes, etc.).

```jsx
import { createContext, useState, useContext } from 'react';
import { ComponenteA } from './componentes';

// Contexto já criado anteriormente
const MeuContexto = createContext({});
export const useMeuContexto = () => {
  return useContext(MeuContexto)
}

// Criando novo contexto de tema (estilo)
const TemaContexto = createContext({});
// Já exportando como hook (sintaxe de retorno imediato)
const useTemaContexto = () =>
  useContext(TemaContexto)
);


export const App = () => {
  // ...
}
```

Fonte: autoral.

Criamos acima o *TemaContexto* e *useTemaContexto*. Agora vamos utilizar o seu *Provider* para oficializar a propagação.

```jsx
export const App = () => {
  const [ titulo, setTitulo ] = useState('React')
  return (
    <div classname="container">
      <h2>{titulo}</h2>
      <TemaContexto.Provider value={{ color: 'green', fontSize: '12px'}}>
        <MeuContexto.Provider value={{ name: 'Descomplica', setTitulo }}>
          <ComponenteA />
        </MeuContexto.Provider>
      </TemaContexto.Provider>
    </div>
  )
}
```

Fonte: autoral.

Veja que aninhamos os *Provider*'s. Nesse caso, a ordem não importa, o *MeuContexto* por exemplo poderia ser o elemento pai do *TemaContexto*.

Agora basta, em um dos componentes descendentes, importar o hook do contexto que queremos utilizar para resgatar o valor.

```
import { useTemaContexto, useMeuContexto } from './App'; // importa
useTemaContexto

const ComponenteZ = () => {
  const value = useMeuContexto();
  const temaContextoValue = useTemaContexto(); // invocando novo contexto

  // utilizando abaixo os valores de temaContextoValue
  return (
    <div classname="componente-z" style={{ color: temaContextoValue.color
}}>
      <p style={{ fontSize: temaContextoValue.fontSize }}>O nome é: <span>
{value.name}</span></p>
      <button type="button" onClick={() => value.setTitulo('O melhor
conteúdo de React')} />
    </div>
  )
}
```

Fonte: autoral.

Seguindo a mesma ideia, invocamos o hook do contexto de tema e no JSX utilizamos seus valores para definir a cor e tamanho da fonte.

Referência Bibliográfica

SILVA, M. S. **React** - Aprenda Praticando: Desenvolva Aplicações web Reais com uso da Biblioteca React e de Seus Módulos Auxiliares. Novatec Editora, 2021.

STEFANOV, S. **Primeiros passos com React:** Construindo aplicações web. Novatec Editora, 2019.

descomplica

Exercícios

1. Quais são formas de resolver o desafio de passagem de propriedades entre componentes aninhados?

a) Prop drilling (passagem um a um), children component e Context API
b) Hooks e prop drilling
c) Componentes de classes e hooks
d) Componentes de classes e hooks
e) Context API, componentes de classes e children component

2. Qual o principal benefício de utilizar Context API?

a) Melhorar performance da renderização dos componentes
b) Invocar funções nas etapas do ciclo de vida do componente
c) Facilitar propagação de valores entre componentes
d) Gerenciar o estado do componente
e) Forma de gerenciar os eventos dos componentes de forma global

3. Quais componentes são gerados pelo createContext para utilizar Context API?

a) Writer e Reader
b) Writable e Consumer
c) Provider e Readable
d) Provider e Reader
e) Provider e Consumer

4. Quais desses é um benefício de utilizar o hook useContext para trabalhar com Context API?

a) Pode acessar o valor propagado pelo Provider mesmo em um componente não aninhado/descendente

b) Forma otimizada e enxuta de passar valores para os componentes descendentes

c) Utilizar valor propagado em um hook customizado

d) Opções A, B e C são todos benefícios do hook useContext

e) Nenhum benefício, apenas uma outra forma de utilizar Context API

Gabarito

1. **Letra A.** Prop drilling é a forma de passar valores componente por componente. Children component é uma forma de passar um valor sem passar antes pelo seu filho diretamente e context api você consegue passar diretamente para qualquer nível de componente aninhado.

2. **Letra D**. Context API vem somente com objetivo de facilitar a propagação de valores entre componentes.

3. **Letra E**. Ao invocar o método createContext, ele retorna dois componentes para trabalhar a propagação de valores: Provider e Consumer.

4. **Letra C**. Graças ao hook useContext podemos acessar o valor propagado pelo Provider do contexto. Antes não era possível, já que o objetivo de um hook customizável não é retornar componentes e sim apenas valores.