front-end

prof. Lucas Ferreira

engenharia de software

engenharia de computação

"Componentização", Gerenciamento de Estado: Hooks e Context API

parte 1

"Componentização" em React

Acredito já ter falado sobre isso algumas vezes, quase tudo em React.js pode ser considerado um ***Component***.

Com exceção de hooks, controladores puramente lógicos e libs de terceiros, o resto tudo é ***Component***.

Desde uma tela inteira a até um pequeno botão de "ligar e desligar" que está nessa mesma tela. Ou seja, tudo é um componente, e sendo um componente maior *(voltando ao exemplo da tela)*, será formado de diversos outros componentes menores que porventura podem ter outros componentes ainda menores dentro deles *(isso pode tender ao infinito)*.

"Componentização" em React

Um dos últimos grandes desafios do time do React foi focar o futuro da biblioteca em componentes baseados em funções e que também pudessem ter estado próprio e determinadas lógicas/ciclos de vidas mais elaborados. Para ajudar nisso também foram criados os ***HOOKs***.

A ideia por trás dos ***hooks*** *(veremos mais a frente)* foi trazer flexibilidade de estado sempre mantendo a performance dos componentes baseados em funções.

Porém antes de chegarmos nos ***HOOKS*** básicos *(e depois ideias mais avançadas/intermediárias)* quero trocar mais umas ideias sobre componentes no React.

Card

```
function Card() {
  return (
    <div className="card">
      <img src="https://placekitten.com/390/240" className="card-img-top" alt="🐱" />
      <div className="card-body">
        <h5 className="card-title">Card title</h5>
        <p className="card-text">
          Some quick example text to build on the card title.
        </p>
        <a href="#go" className="btn btn-primary">
          Go somewhere
        </a>
      </div>
    </div>
  );
}
```

"Componentização" em React

O componente mostrado anteriormente não tem nada de mais, é praticamente um trecho de HTML "convertido para React", nesse caso específico apenas o atributo "*class*" passou a ser "*className*" devido a uma restrição do JSX.

Sendo que seu estilo/css está baseado em componentes puros de ***Bootstrap v5***. Minha ideia daqui em diante é ir "melhorando" a componentização dessa base de "HTML tradicional" para algo um pouco mais "*Reactzado*" em padrões e possibilidades que eu *(na minha preferência)* acho bacana de trabalhar.

Vamos juntos?

Se você está começando em React aqui junto comigo *(na matéria de Front-end)* acho que este exemplo é bacana de seguir:

1. Abra o endereço https://vite.new/react em seu navegador pra iniciar um novo projeto
2. Vá no arquivo index.html e adicione a linha abaixo dentro da tag **<HEAD>**:
   ```
   <link
   href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css
   " rel="stylesheet" crossorigin="anonymous">
   ```
3. Depois vá no arquivo `**src/App.jsx**` para iniciar nosso trabalho

Card

Nosso próximo passo com esse componente é trabalhar com propriedades "customizadas".

Em React temos algo que chamamos de **props**, diretamente ligado a melhor maneira de injetar uma informação externa ao *Component* dentro dele.

```
function Card(props) {
  return (
    <div className="card">
      <img
        src="https://placekitten.com/390/240"
        className="card-img-top"
        alt="🐱"
      />
      <div className="card-body">
        <h5 className="card-title">{props.title}</h5>
        <p className="card-text">
          {props.text}
        </p>
        <a href="#go" className="btn btn-primary">
          {props.buttonLabel}
        </a>
      </div>
    </div>
  );
}
```

Card

No código anterior nós injetamos algumas variáveis baseadas em propriedades para podermos re-aproveitar o mesmo componente mais de uma vez.

Esse é um dos conceitos fundamentais do React.

```
function App() {
  return (
    <div className="App">
      <div className="d-flex gap-3">
        <Card
          title="Adote um Gatinho!"
          text="Eles são muito fofinhos e destroem todos os seus móveis"
          buttonLabel="Quero Adotar"
        />
        <Card
          title="Gato Fofinho"
          text="Eles são muito fofinhos e destroem todos os seus móveis"
          buttonLabel="Quero Apertar"
        />
      </div>
    </div>
  );
}
```

Card

Utilizando o objeto ***props*** como argumento "catch-all" de um componente apesar de ser "rapidamente implementável" é muito pouco didático e também abre uma possibilidade do componente receber atributos "indesejáveis".

Sendo assim o caminho indicado para melhorar esse cenário seria declararmos de forma evidente quais atributos nós aceitamos...

Card

Desta forma também conseguimos um jeito fácil de "definir" valores padrões para argumentos opcionais, com o que foi feito por exemplo com `*buttonLabel*`.

```
function Card({ title, text, buttonLabel = "Clique Aqui" }) {
  return (
    <div className="card">
      <img
        src="https://placekitten.com/390/240"
        className="card-img-top"
        alt="🐱"
      />
      <div className="card-body">
        <h5 className="card-title">{title}</h5>
        <p className="card-text">{text}</p>
        <a href="#go" className="btn btn-primary">
          {buttonLabel}
        </a>
      </div>
    </div>
  );
}
```

Card

Outra ideia no uso declarativo de atributos aceitos como props de um componente é poder usá-los ao longo de toda a estrutura do componente não só no elemento raiz.

Por exemplo, nós informamos um "title" para o componente ***<Card />***, mas esse title é usado dentro de um filho deste componente, no caso o H5 `***<h5 className="card-title">{title}</h5>***`.

Outra ideia bacana é deixar o componente aberto para receber propriedades não previstas, como por exemplo `style` e outras que uma DIV pode aceitar. Podemos inclusive trabalhar com o conceito de spread params para "agrupar" toda e qualquer propriedade não prevista em um objeto "resto"...

Card

Na alteração ao lado criamos um objeto responsável por acumular qualquer outro parâmetro que não seja title, text ou buttonLabel. Na sequência aplicamos esse objeto na DIV principal.

Agora podemos passar um ***style={...}*** para o component Card, por exemplo.

```
function Card({ title, text, buttonLabel, ...props }) {
  return (
    <div className="card" {...props}>
      <img
        src="https://placekitten.com/390/240"
        className="card-img-top"
        alt="🐱"
      />
      <div className="card-body">
        <h5 className="card-title">{title}</h5>
        <p className="card-text">{text}</p>
        <a href="#go" className="btn btn-primary">
          {buttonLabel}
        </a>
      </div>
    </div>
  );
}
```

Card

Para deixar nosso componente ainda mais versátil, podemos trabalhar com testes opcionais para renderizar OU não um pedaço de nosso card. Por exemplo podemos criar um Card com botão opcional assim:

```
{!!buttonLabel && (
  <a href="#go" className="btn btn-primary">
    {buttonLabel}
  </a>
)}
```

A linha acima só será renderizada se a variável ***`buttonLabel`*** estiver validamente preenchida com algo diferente de *null* ou *undefined*.

Card - Nova Ideia

Continuando nossa exploração do componente Card, podemos tomar um rumo diferente na organização/estrutura de nosso componente e de quebra explorar uma outra propriedade bem interessante, a prop "***children***".

Lembrando que até aqui, o uso de nosso componente atualmente está vinculado a parâmetros/propriedades:

```
<Card
  title="Adote um Gatinho"
  text="Eles são muito fofinhos e destroem todos os seus móveis"
  buttonLabel="Quero Adotar!"
/>
```

Card - Nova Ideia

Na teoria o *component* Card já estava razoavelmente estruturado mas podemos concordar que ele está "fazendo coisas d+", perceba que dentro deles temos uma imagem, um "corpo", um título, um campo de texto de apoio, um botão de ação e etc. E se nós quiséssemos flexibilizar ainda mais a personalização desse componente, quantos atributos teríamos que criar?

```
<Card
    title="Gato Fofinho"
    titleProps={...}
    text="Eles são muito fofinhos e destroem todos os seus móveis"
    textProps={...}
    image="https://placekitten.com/390/240"
    imageProps={...}
    buttonLabel="Vou Apertar!"
    onButtonPress={event => {}}
    buttonProps={...}
    meuDeusQuantasProps=[`ta`, 'louco']
  />
```

Card - Nova Ideia

O código anterior prova que quanto maior a versatilidade e complexidade de um componente, mais complicado fica de gerenciar as propriedades/atributos no mesmo.

Diante dessa situação um dos principais caminhos recomendados é começar a "quebrar" componentes grandes em **componentes menores**.

Card

Começamos "extraindo" componentes usados dentro do atual componente Card em outros *components* com suas próprias *props*.

```
function CardThumb({ src, alt = "", ...props }) {
  return <img src={src} className="card-img-top" alt={alt} {...props} />;
}

function CardTitle({ title, ...props }) {
  return (
    <h5 className="card-title" {...props}>
      {title}
    </h5>
  );
}

function CardText({ text, ...props }) {
  return (
    <p className="card-text" {...props}>
      {text}
    </p>
  );
}

function CardButton({ label, href = "#go", ...props }) {
  return (
    <a href={href} className="btn btn-primary" {...props}>
      {label}
    </a>
  );
}
```

Card - Nova Ideia

Uma vez que tenhamos componentes menores nós já podemos "desonerar" nosso componente principal e deixá-lo mais *"light"*, recebendo outros componentes como parâmetros/props:

```
function Card({ title, text, button, thumb, ...props }) {
  return (
    <div className="card" {...props}>
      {thumb}
      <div className="card-body">
        {title}
        {text}
        {button}
      </div>
    </div>
  );
}
```

Card

Um novo uso para primeira nova "ideia" de reformulação do componente Card é passar os "sub" componentes como props para dentro do componente "esqueleto" a fim de conseguirmos personalizar melhor nosso uso:

```
<Card
  thumb={
    <CardThumb
      style={{ border: "1px solid red" }}
      src="https://placekitten.com/390/240"
      alt="🐱"
    />
  }
  title={
    <CardTitle title="Adote um Gatinho" style={{ color: "red" }} />
  }
  text={
    <CardText text="Eles são muito fofinhos e destroem todos os seus móveis" />
  }
  button={<CardButton label="Quero Adotar!" />}
/>
```

Card - Nova... Nova Ideia

Creio que até aqui o cenário de otimização/flexibilização do nosso componente Card já tenha melhorado bastante, mas acho que podemos deixar a coisa "ainda mais" flexível, explorando a ideia de realmente mantermos o componente principal como um "esqueleto" que só recebe conteúdo.

Usando a propriedade "***children***" iremos proporcionar ao nosso projeto quase que "uma nova tag de HTML".

Card - Nova... Nova Ideia

Ao invés de especificar um atributo para cada elemento, vamos trabalhar com um mínimo estrutural:

```
function Card({ children, thumb, ...props }) {
  return (
    <div className="card" {...props}>
      {thumb}
      <div className="card-body">{children}</div>
    </div>
  );
}
```

Dá para perceber nossa "economia" em estrutura, com o atributo "***children***" localizado no lugar certo abrimos um portal para receber qualquer conteúdo dentro do nosso *component* Card.

Card - Nova... Nova Ideia

E assim uma ideia de uso:

```
<Card>
  <CardTitle title="Adote um Gatinho" style={{ color: "red" }} />
  <CardText text="Eles são muito fofinhos e destroem todos os seus móveis" />
  <CardButton label="Quero Adotar!" />
</Card>
```

Ou por exemplo, sem um botão em nosso Card:

```
<Card>
  <CardTitle title="Adote um Gatinho" style={{ color: "red" }} />
  <CardText text="Eles são muito fofinhos e destroem todos os seus móveis" />
</Card>
```

Introdução aos HOOKs

Mas, o que é um Hook?

Resumidamente, hooks foram lançados a proposta principal do *core-team* React para permitir e utilizar estado, ciclo de vida, entre outras funcionalidades sem a necessidade de escrevermos componentes com classe *(modelo antigo)*. A funcionalidade encontra-se disponível desde a versão 16.8 do React.

Sendo ainda mais básico são funções que permitem ao dev. “ligar-se” aos recursos de state e ciclo de vida do React a partir de componentes funcionais.

State Hook

É o nosso primeiro e mais básico hooks, serve como introdução ao conceito.

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

function Contador() {
  // Declara uma nova variável de state, que chamaremos de "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Você clicou {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Clique aqui
      </button>
    </div>
  );
}
```

State Hook

O state hook (***useState***) foi chamado de dentro de um componente funcional para adicionar algum estado local. O React irá preservar este state entre re-renderizações *(vou demonstrar)*.

O ***useState*** *(e a maioria dos hooks)* retorna um par de elementos: **o valor do state atual e uma função que permite atualizá-lo**.

O único argumento para ***useState*** é o estado inicial. No exemplo anterior é 0, porque nosso contador começa do zero. Por fim, o argumento de state inicial é utilizado apenas durante a primeira renderização.

✌️ Regras dos Hooks

Hooks são funções JavaScript, mas eles impõem duas regras adicionais:

- Apenas chame Hooks **no nível mais alto**. Não chame Hooks dentro de loops, condições ou funções aninhadas.
- Apenas chame Hooks de **componentes funcionais**. Não chame Hooks de funções JavaScript comuns *(apesar de ser válido também chamar hooks de dentro dos seus próprios Hooks customizados)*.

spoilers

Não percam o próximo capítulo dessa aula o qual veremos:

- **useEffect**
- **useReducer**
- **ContextApi** e **useContext**
- **React Router aplicado em projetos simples** ***(troca de telas, 100% front-end)***

obrigado 🚀