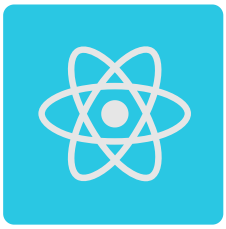

Componentes no React

Faaala Dev!

Nesse documento iremos falar sobre componentes no React.

Ultimamente os componentes com o React têm sido feitos em formato de funções, onde usamos o hooks para realizar cálculos ou até mesmo chamadas assíncronas quando um componente é renderizado ou quando algum estado muda (um exemplo disso é o uso do `useEffect`), mas você sabia que podemos escrever componentes também em classes? você sabe a diferença entre escrever um componente com classes e funções?

Aqui eu te mostrarei um pouco disso. Afinal, é sempre bom saber quais aplicabilidades uma ferramenta nos dá mesmo que não formos usar alguma *feature* no nosso dia-a-dia.

Class Component

Antes da chegada dos hooks na versão 16.8 do React, os componentes eram escritos em formato de classe. O React nos dá uma pseudo-classe com métodos para gerenciar o ciclo de vida do componente onde estendemos a nossa classe usando a pseudo-classe citada.

Se você não entendeu muito bem, é basicamente o seguinte:

```
import { Component } from 'react'; class App extends Component { render()
{ return ( <h1>Hello World</h1> ) } } export default App;
```

Percebe que usamos um método chamado `render` para mostrar o componente? Esse é um dos métodos que a classe `Component` importada do React nos dá.

Quando precisamos executar algo assim que o componente é renderizado, antes de ser renderizado ou até mesmo quando as propriedades recebidas por ele mudam, temos um método específico que será usado para cada caso.

Em resumo, componentes escritos com classes **definem funções** que são executadas durante o **ciclo de vida** de um componente e é por isso que usamos o `extends` com o `Component` que é exportado do React.

Ciclo de vida de um componente

Se você ficou um pouco confuso quando falei sobre ciclo de vida, eu me referi a tudo que é executado antes, durante e depois da renderização de um componente e isso inclui mudanças de estado, chamadas à APIs, etc.

Em componentes com classes, quando precisamos executar chamadas à APIs ou mudanças de estado de acordo com o ciclo de vida do componente (após ele ser montado, por exemplo), a classe `Component` nos permite usar alguns métodos para isso como por exemplo o `componentDidMount` que é executado sempre após a renderização de um componente.

Vamos observar um simples código onde um componente recebe uma propriedade e mostra isso em tela:

```
import { Component } from 'react'; class HelloWorld extends Component {
render() { return ( <h1>{this.state.message}</h1> ) } } class App extends
Component { render() { return ( <HelloWorld message="Hello world" /> ) }
} export default App;
```

Um ponto a ressaltar é que, se você já usou componentes em formato de função e viu esse aí acima, vai perceber que o código está um pouco "verboso", com muito código para pouca coisa. Esse é um dos motivos pelo qual preferimos usar funções em vez de classes.

Functional Component

Após a atualização 16.8 citada no início desse documento, foram introduzidos os *hooks*. Com eles é possível escrever componentes como você provavelmente já usou ou irá usar, usando state e outros recursos do React sem escrever uma classe e sim uma função!

Se formos reescrever os dois componentes mostrados acima mas em formato de função, teríamos o seguinte código:

```
import { useState } from 'react'; function HelloWorld(props) { return ( <h1>{props.message}</h1> ) } function App() { return ( <HelloWorld message="Hello world" /> ) } export default App;
```

Você pode estar se perguntando "Tá, mas e como usamos os métodos que executam com o ciclo de vida do componente que antes eram estendidos da classe `Component`?".

Todos eles continuam acessíveis mas agora usando os hooks!

Por exemplo, o hook `useEffect` possui, dentre outras, a funcionalidade do método `componentDidMount`.

Sempre que precisamos executar uma ação durante a renderização do componente, basta importar e declarar ele dentro do componente. Se precisamos re-executar essa mesma ação quando algum comportamento no seu componente acontece (como uma mudança de estado, por exemplo) basta colocar a dependência dentro do array de dependências que o useEffect recebe. E por último mas não menos importante, se precisamos executar alguma coisa quando o componente deixa de existir (como remover um listener, por exemplo), podemos fazer isso retornando uma função de dentro do useEffect realizando esta ação dentro dessa função:

```
import { useEffect } from 'react' function App() { useEffect(() => { // Aqui executamos uma ação quando o componente é montado // ou quando uma dependência muda return () => { // Aqui executamos uma ação quando o componente é desmontado } }, [message]) // Aqui a nossa dependência é 'message' return ( <HelloWorld message={message} /> ) }
```

Não se preocupe muito com a sintaxe se você não estiver acostumado. Iremos utilizar bastante os hooks durante o Ignite para fixar bem a aplicabilidade de cada um.

Ainda sobre componentes funcionais, existe uma outra forma de escrevermos um componente que é usando *arrow functions*. Nesse caso temos um código ainda mais curto mas nem por isso ele deixa de ser facilmente entendível.

Aqui vemos um exemplo de um componente que retorna apenas um `h1`:

```
const App = (props) => { return <h1>{props.title}</h1> }
```

E que pode ficar ainda mais simples:

```
const App = (props) => <h1>{props.title}</h1>
```

Bem bacana, né? Veja que nesse caso, não possuímos nenhum gerenciamento de estado ou efeitos e por isso há apenas o retorno do conteúdo HTML.

Espero que você tenha entendido um pouco da diferença entre componentes escritos com funções e classes!