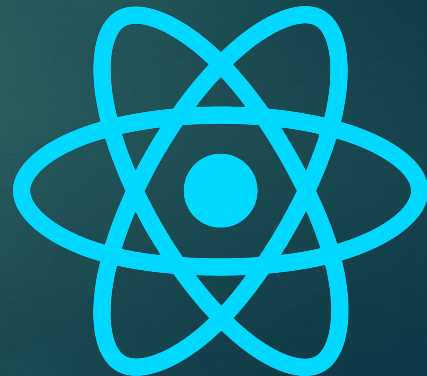


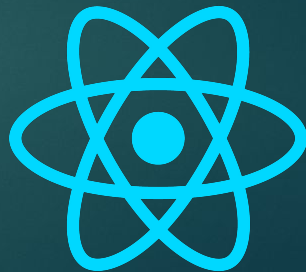
# Biblioteca do Javascript: React

Feito por Alvaro Carvalho de Lima



# O que é React?

- ▶ React é uma biblioteca javascript desenvolvida pelo Facebook com o objetivo de melhorar a criação de interfaces do usuário.
- ▶ Ele pode ser usado para criar aplicativos que possuem uma única página (como a netflix, você não vê uma barra de navegação dizendo as páginas que pode ir nela, ficando sempre na mesma página).
- ▶ Uma de suas vantagens é permitir a criação de componentes que podem ser reutilizados sempre que quiser.



# Como funciona o React?

Em vez de manipular o DOM do navegador diretamente, o React cria um DOM virtual na memória, onde faz toda a manipulação necessária, antes de fazer as alterações no DOM do navegador.

Ele consegue trabalhar com componentes, permitindo que ele divida a página em várias partes e trabalhe nelas de forma independente, e assim, mudando apenas o que precisa mudar.

# Usando o React

Para usar o React de forma completa, você precisará ter o 'node.js' instalado no computador. Caso já tenha e apenas queira verificar a versão:

Digite 'cmd' ou 'terminal' na barra de pesquisa do windows. Após isso, abra o "prompt de comando" e escreva 'node -v' para olhar a versão, caso ele apareça como não reconhecido, o instale no site original (apenas jogando node.js no google).

**OBS:** Para criar um arquivo React, seu 'node' deverá ter a versão igual ou superior a 14.0.0 e o 'npm(vem instalado com o node)' precisa ter uma versão igual ou superior a 5.6.

# Usando o React

Agora que já baixou/verificou a versão de seu node, abra o editor de código de sua preferência e também abra o terminal dele, logo após, escreva o seguinte código:

```
npx create-react-app my-react-app
```

Enquanto a parte do 'npx create-react-app' é obrigatória para dizer que deseja baixar todos os arquivos, criando este novo projeto do react. A parte 'my-react-app' trata-se apenas do nome da pasta, então pode escrever qualquer coisa contanto que não tenha espaços nem letra maiúscula.



# Usando o React

Após baixar todo o código e arquivos, você precisará entrar dentro da pasta do React para poder trabalhar com a mesma. A forma mais fácil de fazer isso é escrever o seguinte comando no terminal do editor:

```
cd my-react-app
```

Pois o comando 'cd' serve para entrar na pasta cujo nome for especificado logo em seguida.

Por fim, para exibir sua página no navegador, basta escrever no terminal o comando "npm start".

```
npm start
```

# Modificando o arquivo React

Certo, você conseguiu ver seu aplicativo rodando no navegador. Mas deve estar se perguntando como fazer para estar modificando ele, para isso, deve ir até a pasta 'src' e achar o arquivo 'App.js' onde verá este código:

Nele você pode retirar tudo que estiver dentro do 'return' e escrever o código que quiser, tipo esse:

```
function App() {  
  return (  
    <div className="App">  
      <h1>Hello World!</h1>  
    </div>  
  );  
}  
  
export default App;
```

```
import logo from './logo.svg';  
import './App.css';  
  
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <p>  
          Edit <code>src/App.js</code> and save to reload.  
        </p>  
        <a  
          className="App-link"  
          href="https://reactjs.org"  
          target="_blank"  
          rel="noopener noreferrer"  
        >  
          Learn React  
        </a>  
      </header>  
    </div>  
  );  
}  
  
export default App;
```

# Modificando o arquivo React

Você também pode remover os códigos e arquivos que não irá usar como medida para liberar memória, tendo como exemplo o arquivo CSS, ou o index.js, etc.



# Variáveis no React

Antigamente, só havia uma maneira de declarar variáveis, que era utilizando a palavra-chave 'var'. Caso você não declare elas, seria informado um erro de variável indefinida se você estivesse no modo estrito.

No entanto, agora existem 3 formas diferentes de declarar variáveis, sendo elas: 'var', 'let' e 'const'.

Há também um novo escopo, que é o 'escopo de função' específico para a mesma.

Por exemplo, caso você declare uma variável com 'var' dentro de uma função, será declarado um 'escopo de função' mas caso a declare em um bloco, será declarada como escopo global.

# Variáveis no React

O `let` possui um escopo de bloco, logo diferente do `'var'` ele é só pode ser usado no bloco em que for criado, seja um loop, etc.

Já a palavra `'const'` não define um valor constante, mas sim faz uma referência a ele

Com um `'const'` você NÃO pode:

- ▶ Reatribuir um valor constante.
- ▶ Reatribuir uma matriz constante.
- ▶ Reatribuir um objeto constante.

Mas você pode:

- ▶ Alterar os elementos em que forem declaradas com ele.

# A função ReactDOM.render()

Esta função é de extrema importância, pois como o nome dela já diz, ela renderiza o código em uma página na Web.

Esta função recebe dois argumentos, o primeiro é o texto que será renderizado, já o segundo é o local que irá renderizar esse texto, por exemplo em uma div com o id chamado 'root':

```
ReactDOM.render(<p>Hello</p>, document.getElementById('root'));
```

```
<body>  
  <div id="root"></div>  
</body>
```

# A função React.DOM.render(): Root

Root é o 'div' que será usado para exibir o conteúdo escrito como primeiro argumento da função 'render()'. Mas precisa saber de algumas coisas. Como por exemplo o fato de que não é obrigatório que o 'id' desta 'div' seja 'root', assim como não é obrigatório ser uma 'div'. Por exemplo:

```
ReactDOM.render(<p>Hallo</p>, document.getElementById('sandy'));
```

```
<body>  
  
  <header id="sandy"></header>  
  
</body>
```

# JSX no React

JSX significa Javascript e XML, ele nos permite escrever códigos HTML no React junto com o Javascript sem quaisquer problemas. Por exemplo:

```
const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



# Código HTML no React

O react consegue estar reconhecendo qualquer elemento HTML sem precisar de uma tag específica (como o `<script>`), além disso ele consegue rodar ambos javascript e html na mesma folha de código. Por exemplo:

```
const myelement = (  
  <table>  
    <tr>  
      <th>Name</th>  
    </tr>  
    <tr>  
      <td>John</td>  
    </tr>  
    <tr>  
      <td>Elsa</td>  
    </tr>  
  </table>  
)  
;  
  
ReactDOM.render(myelement, document.getElementById('root'));
```

# Expressões em JSX

Com o JSX é possível escrever expressões (operações) dentro de chaves {} para evitar concatená-las ao texto como ocorria no javascript. Por exemplo:

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

# Elemento superior no JSX

Caso você tenha colocado mais de um parágrafo dentro da mesma variável, acabou obtendo uma mensagem de erro ao invés do resultado esperado. Isto se deve pelo fato de que só pode ser passado 'um elemento superior' por variável. Por exemplo, uma solução para o caso acima é cobrir os dois parágrafos com uma `<div>`, pois ela será o "único" elemento chefe a ser passado, não importando quantos elementos filhos existam dentro dela. Por exemplo:

```
const myElement = (  
  <div>  
    <p>Eu sou um parágrafo.</p>  
    <p>Eu também sou um parágrafo.</p>  
  </div>  
);
```

# Fragmento no JSX

O fragmento é muito utilizado para encapsular elementos e ser uma alternativa para a `<div>` no caso anterior. Por exemplo:

```
const myElement = (  
  <>  
    <p>Eu sou um parágrafo.</p>  
    <p>Eu também sou um parágrafo.</p>  
  </>  
)
```

# Elementos fechados no JSX

Todos elementos que forem iniciados no JSX devem ser fechados, sem exceções, até mesmo para os elementos vazios que não possuem “tag de fechamento”, como medida você pode colocar uma barra ( / ) do lado do sinal de maior ( > ). Por exemplo:

```
const myElement = <input type="text" />;
```

Caso o elemento vazio não seja fechado, a página irá disparar um erro.



# Atributo class no React

O React não permite estar utilizando o atributo 'class' para definir determinada classe, uma vez que seu código é renderizado como Javascript, fazendo com que a palavra 'class' se torne uma palavra reservada. No entanto, como medida preventiva, foi criado o atributo 'className' que faz as mesmas coisas que sua versão anterior.

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

# Condições no JSX

O JSX não oferece suporte com as condições de 'if' e 'else'. Então você não consegue armazenar o resultado da verificação diretamente, a menos que utilize o operador ternário. Por exemplo:

```
const x = 5;  
let text = "Goodbye";  
if (x < 10) {  
  text = "Hello";  
}  
  
const myElement = <h1>{text}</h1>;
```

# Operador ternário no JSX

Ao contrário da condição 'if' e 'else', você consegue armazenar diretamente o resultado da verificação caso utilize o operador ternário. Por exemplo:

```
const x = 5;  
  
const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

O código após o ' ? ' é usado para caso a condição seja verdadeira, já o código depois do ' : ' é usado caso a condição seja falsa.

# Componentes no React

Os componentes são pedaços de código independentes e reutilizáveis. Eles servem para o mesmo propósito que as funções no Javascript, mas trabalham de forma isolada e retornam elementos HTML.

Existem dois tipos de componentes, sendo os componentes de classe e os componentes de função.

Como regra número um: O nome de um componente **DEVE** começar com letra maiúscula.

# Componentes no React:

## componente de classe

Um componente de classe deve incluir a instrução 'extends'. Essa instrução cria uma herança para `React.Component` (informando que se trata de um componente do react) e dá ao componente acesso às funções do `React.Component`.

O componente também requer um método `render()`, esse método pode retornar código HTML. Por exemplo:

```
class Car extends React.Component {  
  render() {  
    return <h2>Hi, I am a Car!</h2>;  
  }  
}
```



# Componentes no React:

## componente de função

Um componente de função também retorna código HTML e se comporta da mesma maneira que um componente de classe, mas os componentes da função podem ser escritos usando muito menos código, além de serem mais fáceis de entender. Como podemos ver no mesmo exemplo que o componente de classe:

```
function Car() {  
  return <h2>Hi, I am a Car!</h2>;  
}
```

# Renderizando os componentes

É bem similar a forma de renderizar os componentes com a forma de renderizar o código HTML no Javascript puro, a diferença é usando a função `createRoot`. Por exemplo:

```
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Car />);
```

# Props nos componentes

Os componentes também podem ser passados como 'props', que significa propriedade.

Props são como parâmetros das funções, você pode usar eles para receber valores do usuário ou quando for exibir este código mais tarde. Por exemplo:

```
function Car(props) {  
  return <h2>I am a {props.color} Car!</h2>;  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Car color="red"/>);
```

# Componentes em componentes

É possível chamar um componente dentro de outro componente, pode ser usado para intercalar informações. Por exemplo:

```
function Car() {  
  return <h2>Eu sou um carro!</h2>;  
}  
  
function Garage() {  
  return (  
    <>  
      <h1>Quem mora na minha Garagem?</h1>  
      <Car />  
    </>  
  );  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Garage />);
```

# Eventos no React:

Assim como no Javascript, o React também pode disparar algumas ações a partir de eventos do usuário. Inclusive possuindo os mesmos eventos que o javascript, porém existem algumas diferenças, como as seguintes:

- ▶ Os eventos no React são escritos em 'camelCase'. Por exemplo, se escreve 'onClick' ao invés de 'onclick'.
- ▶ Os manipuladores de eventos são gravados através de chaves. Por exemplo: onClick={correr} em vez de onclick='correr()'.

```
<button onClick={shoot}>Take the Shot!</button>
```

```
<button onclick="shoot()">Take the Shot!</button>
```



# Condicionais no React

Assim como mostrado anteriormente, é possível estar utilizando do desvio condicional (if) para realizar verificações. Além disso, outra coisa que pode ser usada com essas validações é o operador lógico ( && | | ! ), onde os dois primeiros servem para realizar mais de uma verificação ao mesmo tempo, já o ! (não) nega o resultado de uma verificação. Por exemplo:

```
function Garage(props) {  
  const cars = props.cars;  
  return (  
    <>  
      <h1>Garage</h1>  
      {cars.length > 0 &&  
        <h2>  
          You have {cars.length} cars in your garage.  
        </h2>  
      }  
    </>  
  );  
}  
  
const cars = ['Ford', 'BMW', 'Audi'];  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Garage cars={cars} />);
```

# Formulários no React

Você também consegue estar adicionando formulários livremente dentro dos componentes, como neste caso:

```
function MyForm() {  
  return (  
    <form>  
      <label>Enter your name:  
        <input type="text" />  
      </label>  
    </form>  
  )  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<MyForm />);
```

O formulário irá funcionar perfeitamente nele.

# Formulários no React

No entanto, o React tem sua própria forma de manipular formulários, que é sua forma de manipular os dados enviados pelo usuário.

Você pode usar o atributo 'onChange' dentro do input para "capturar" o valor inserido pelo usuário, assim como pode usar o gancho 'useState()' (será aprendido sobre ganchos mais pra frente) para acompanhar o primeiro valor do formulário, com o que será recebido. Por exemplo:

```
function MyForm() {  
  const [name, setName] = useState("");  
  
  return (  
    <form>  
      <label>Enter your name:  
        <input  
          type="text"  
          value={name}  
          onChange={(e) => setName(e.target.value)}  
        />  
      </label>  
    </form>  
  )  
}
```

# Formulários no React: enviando formulários

Você pode enviar formulários para algum outro componente ou página caso adicione o atributo 'onSubmit' no elemento <form>. Por exemplo:

Nesse caso foi usado o atributo 'onSubmit' para redirecionar o nome informado até a 'arrow function' 'handleSubmit', que irá retornar a informação passada em um 'alert'.

```
function MyForm() {  
  const [name, setName] = useState("");  
  
  const handleSubmit = (event) => {  
    event.preventDefault();  
    alert(`The name you entered was: ${name}`)  
  }  
  
  return (  
    <form onSubmit={handleSubmit}>  
      <label>Enter your name:  
        <input  
          type="text"  
          value={name}  
          onChange={(e) => setName(e.target.value)}  
        />  
      </label>  
      <input type="submit" />  
    </form>  
  )  
}
```

# Formulários no React: textarea

Também é possível trabalhar com o elemento 'textarea' no React. Tendo esta como uma das formas:

```
<textarea>  
  Content of the textarea.  
</textarea>
```

Além disso, ele também pode ser tratado da mesma forma que o input com o 'useState'. Por exemplo:

```
function MyForm() {  
  const [textarea, setTextarea] = useState(  
    "O conteúdo de uma área de texto vai no atributo value"  
  );  
  
  const handleChange = (event) => {  
    setTextarea(event.target.value)  
  }  
  
  return (  
    <form>  
      <textarea value={textarea} onChange={handleChange} />  
    </form>  
  )  
}
```



# Formulários no React: select

Assim como é possível estar usando um textarea ou formulário no React, também é possível usar o 'select' que atua como uma "lista suspensa". Desde a forma HTML, até a com o React manipulando:

```
<select>
  <option value="Ford">Ford</option>
  <option value="Volvo" selected>Volvo</option>
  <option value="Fiat">Fiat</option>
</select>
```

```
function MyForm() {
  const [myCar, setMyCar] = useState("Volvo");

  const handleChange = (event) => {
    setMyCar(event.target.value)
  }

  return (
    <form>
      <select value={myCar} onChange={handleChange}>
        <option value="Ford">Ford</option>
        <option value="Volvo">Volvo</option>
        <option value="Fiat">Fiat</option>
      </select>
    </form>
  )
}
```

# Formulários no React: importando o useState

Sempre que for utilizar algum formulário, área de texto ou lista suspensa no React. Lembre-se de importar o 'useState' através da linha de código.

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

Do contrário, ele não irá funcionar.

# Memo no React

O uso deste comando faz com que o React ignore a renderização de um componente caso seus adereços (props ou os valores do useState) não tenham sido alterados. Isso permite um melhor desempenho em sua página, uma vez que todos os componentes utilizados (podem acabar sendo muitos ou pesados) sejam renderizados apenas quando os dados forem atualizados.

Ele funciona com todos e quaisquer componentes, para utilizá-lo basta importá-lo com a linha de código:

```
import { memo } from "react";
```

E o exportando junto com o componente:

```
export default memo(Todos);
```

# CSS no React

Há três formas de se estilizar o React através do CSS. Sendo elas:

- ▶ Inline
- ▶ Objetos CSS
- ▶ Folha de Estilo Externo.
- ▶ Módulos CSS

# CSS no React: inline

A forma inline já deve ser conhecida por todos, que é onde se estiliza um elemento específico através do atributo 'style'. Por exemplo:

```
<h1 style={{color: "red"}}>Hello Style!</h1>
```

Sua principal diferença de quando aplicado no HTML é que seu estilo é coberto por 2 chaves {{}}.

Outra diferença é que ele separa palavras longas pela forma 'camelCase' ao invés de colocar um hífen (traço - ) no meio. Por exemplo:

```
<h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1>
```



# CSS no React: objetos CSS

É possível também utilizar propriedades do CSS como se fossem palavras de um objeto do Javascript. Por exemplo:

```
const Header = () => {  
  const myStyle = {  
    color: "white",  
    backgroundColor: "DodgerBlue",  
    padding: "10px",  
    fontFamily: "Sans-Serif"  
  };  
  return (  
    <>  
      <h1 style={myStyle}>Hello Style!</h1>  
      <p>Add a little style!</p>  
    </>  
  );  
}
```

# CSS no React: folhas de estilo CSS

A forma que todos estão mais acostumados em utilizar certamente é a do CSS externo, onde nesse caso a única coisa de diferente do CSS aplicado no HTML é a forma que importa o arquivo com a extensão '.css'. Por exemplo:

```
import './App.css';
```

# CSS no React: módulos CSS

Os módulos CSS são convenientes para componentes, uma vez que ele só funciona no componente que ele for utilizado, dessa forma é possível evitar conflitos de nomes iguais. Para usar o módulo CSS, basta aplicar a classe "style.nomeClasse". Por exemplo:

```
return <h1 className={styles.bigblue}>Hello Car!</h1>;
```

```
.bigblue {  
  color: DodgerBlue;  
  padding: 40px;  
  font-family: Sans-Serif;  
  text-align: center;  
}
```

# CSS no React: módulo CSS

E também, basta importá-lo da seguinte maneira:

```
import styles from './my-style.module.css';
```

Basta importá-lo uma vez e então utilizá-lo em todo o código.

# SASS no React

Sass é um pré-processador CSS.

Os arquivos Sass são executados no lado do servidor e enviam CSS para o navegador.

É um pouco diferente do CSS para usá-los no React. Você precisa baixá-lo no terminal de seu navegador com a seguinte linha de código:

```
>npm i sass
```

Após isso, basta importá-lo como no arquivo CSS:

```
import './my-sass.scss';
```



# Ganchos no React

Os ganchos nos permitem "conectar" aos recursos do React, como métodos de estado e ciclo de vida.

Para usá-los, primeiro você precisa importá-los do 'react'. Por exemplo:

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

```
const [color, setColor] = useState("red");
```

# Ganchos no React: regras

Existem 3 regras para ganchos, sendo elas:

- ▶ Os ganchos só podem ser chamados dentro de componentes de função.
- ▶ Os ganchos só podem ser chamados no topo dos componentes em React.
- ▶ Os ganchos não podem ser condicionais (tipo quando armazena um valor de condicional dentro de uma variável).

# Ganchos no React: useState

O gancho 'useState' nos permite rastrear o estado em um componente de função. Estado geralmente se refere a dados ou propriedades que precisam ser rastreados em um aplicativo, como por exemplo um input, onde você define o valor vazio e o valor que será recebido.

Para usá-lo, primeiro precisa importá-lo no começo da página.

```
import { useState } from "react";
```

Em seguida, nós precisamos iniciar ele no começo de uma função.

```
function FavoriteColor() {  
  const [color, setColor] = useState("");  
}
```

# Ganchos no React: useState

O primeiro valor é o estado atual do elemento, já o segundo valor vai armazenar a atualização deste elemento.

Um exemplo é quando você usa o primeiro valor em um input e depois marca o segundo valor para receber o que o usuário enviar no input.

O parâmetro passado dentro do 'useState()' será marcado como no primeiro valor dele.

```
const [color, setColor] = useState("red");
```

No entanto, podemos mudar a cor ao colocar um botão com o atributo 'onClick' que chama o nome do segundo valor:

```
onClick={() => setColor("blue")}
```

# Ganchos no React: useState

Também é possível colocar uma propriedade dentro dos parênteses do 'useState()'. Por exemplo:

```
function Car() {  
  const [car, setCar] = useState({  
    brand: "Ford",  
    model: "Mustang",  
    year: "1964",  
    color: "red"  
  });  
  
  return (  
    <>  
      <h1>My {car.brand}</h1>  
      <p>  
        It is a {car.color} {car.model} from {car.year}.  
      </p>  
    </>  
  )  
}
```



# Ganchos no React: useEffect

Este gancho permite que você aplique efeitos em seus componentes. Alguns exemplos dele são: temporizadores, atualização direta do DOM e busca de dados.

O 'useEffect' aceita dois argumentos, tendo o segundo como opcional.

Usando um código de temporizador como exemplo:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I've rendered {count} times!</h1>;
}
```

# Explicando o código do temporizador

```
import { useState, useEffect } from "react";  
import ReactDOM from "react-dom/client";
```

Nesta parte nós definimos importamos os ganchos do 'useState' e do 'useEffect'.

Na linha abaixo nós importamos o ReactDOM para podermos usá-lo no método que está na última linha do código.

# Explicando o código do temporizador

```
function Timer() {  
  const [count, setCount] = useState(0);
```

Criando o componente de função com o nome 'Timer'.

Dentro do componente nós criamos uma const que receberá o gancho 'useState'.

Definimos o valor inicial do 'useState' como 0 através do argumento passado entre parênteses.

# Explicando o código do temporizador

```
useEffect(() => {  
  setTimeout(() => {  
    setCount((count) => count + 1);  
  }, 1000);  
});
```

Após isso o gancho 'useEffect' é usado, tendo uma "arrow function" como seu primeiro argumento.

Dentro da arrow function é definido a função 'setTimeout' tendo como argumento outra 'arrow function'.

Dentro da outra 'arrow function' é definido a função 'setCount' (segundo valor do 'useState').

# Explicando o código do temporizador

```
setCount((count) => count + 1);  
  }, 1000);  
});
```

No valor 'setCount' do 'useState', é definido uma 'arrow function' onde é enviado o valor inicial do useState (count) como parâmetro, e diz que sempre que a função carregar, 'count' terá seu valor incrementado.

Na linha abaixo, é definido o segundo argumento do 'setTimeout', que é o tempo que a função irá levar para ser iniciada novamente.

Então o gancho 'useEffect' é encerrado logo abaixo.



# Explicando o código do temporizador

```
return <h1>I've rendered {count} times!</h1>;  
}
```

Então no final do componente de função 'Timer', é definido que ele irá retornar um <h1> dizendo que foi renderizado 'count' vezes. Onde o valor do 'count' será atualizado por segundo.

Na linha abaixo o componente de função 'Timer' é encerrado.

# Explicando o código do temporizador

```
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Timer />);
```

Então é usado o método do 'ReactDOM' que foi importado após os dois ganchos.

Este método envia como primeiro argumento que irá exibir o código dentro da div 'root'.

Então é usado o método render do root (criado pelo ReactDOM) que ele irá renderizar o componente de função 'Timer'.

# Ganchos no React: useEffect

No entanto, caso o segundo argumento dele esteja vazio, mostra que este gancho sempre será renderizado de acordo com a contagem do 'setTimeout'.

Isso pode acabar acarretando em alguns problemas de desempenho, caso seu código seja muito grande.

No entanto, para fazer este gancho rodar apenas uma vez, basta fazer com que seu segundo argumento seja preenchido com colchetes vazios. Por exemplo:

```
useEffect(() => {  
  Executa somente na primeira renderização  
}, []);
```

# Ganchos no React: useEffect

Para não ter o gancho sendo renderizado a todo instante, nem ele sendo renderizado uma única vez. Basta fazer com que ele dependa de uma variável, um botão, etc. Por exemplo:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

function Counter() {
  const [count, setCount] = useState(0);
  const [calculation, setCalculation] = useState(0);

  useEffect(() => {
    setCalculation(() => count * 2);
  }, [count]); // <- add the count variable here

  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount((c) => c + 1)}>+</button>
      <p>Calculation: {calculation}</p>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Counter />);
```

# Explicando o segundo código do useEffect

```
function Counter() {  
  const [count, setCount] = useState(0);  
  const [calculation, setCalculation] = useState(0);
```

Para adiantar, vamos pular as partes repetitivas como importação ou a exibição através do 'ReactDOM'.

É criado um componente de função chamado de 'Counter'.

Logo abaixo é definido dois ganchos do 'useState', o primeiro sendo o contador, já o segundo sendo a operação (que irá depender do contador para ser renderizado).



# Explicando o segundo código do useEffect

```
useEffect(() => {  
  setCalculation(() => count * 2);  
}, [count]); // <- add the count variable here
```

Depois disso, é aplicado o gancho do 'useEffect' onde ele recebe uma 'arrow function' como primeiro argumento.

Dentro desta 'arrow function' é definido o valor que será atualizado do calculador, com uma 'arrow function' dizendo que seu valor será o mesmo que o valor do 'count' multiplicado por 2.

O 'useEffect' recebe como segundo argumento o 'count' entre colchetes para definir a quantidade de vezes que ele será renderizado.

# Explicando segundo código do useEffect

```
return (  
  <>  
    <p>Count: {count}</p>  
    <button onClick={() => setCount((c) => c + 1)}>+</button>  
    <p>Calculation: {calculation}</p>  
  </>  
>);
```

No final a função irá retornar um parágrafo que terá o valor do 'count'.

Irá retornar também um botão que quando clicado irá atualizar o valor do count, somando ele com 1.

E então exibe o valor do calculador.

Por fim ele fecha o return, e a função.

# Ganchos no React: useRef

Em geral, queremos deixar o React lidar com toda a manipulação do DOM. Uma das formas de se obter isso, é utilizando o 'useRef'.

No React, podemos adicionar um atributo 'ref' a um elemento para acessá-lo diretamente no DOM. Por exemplo:

```
import { useRef } from "react";
import ReactDOM from "react-dom/client";

function App() {
  const inputElement = useRef();

  const focusInput = () => {
    inputElement.current.focus();
  };

  return (
    <>
      <input type="text" ref={inputElement} />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

# Explicando o código do useRef

```
function App() {  
  const inputElement = useRef();
```

Após importar o gancho e o 'ReactDOM', é criado um componente de função com o nome de 'App'.

Na primeira linha de código dentro dele é definido uma const com o nome de 'inputElement', que recebe o valor do gancho 'useRef'.

# Explicando o código do useRef

```
const focusInput = () => {  
  inputElement.current.focus();  
};
```

Após isso, é criada outra constante que irá receber uma 'arrow function' como valor.

Dentro dela é escrito que o "atual(current)" inputElement receberá o estado focado (quando um input está selecionado) através da função 'focus()'.

Na linha abaixo ele encerra a 'arrow function'.



# Explicando o código do useRef

```
return (  
  <>  
    <input type="text" ref={inputElement} />  
    <button onClick={focusInput}>Focus Input</button>  
  </>  
);
```

Então será retornar um input do tipo texto com o atributo 'ref' para selecioná-lo como o 'inputElement' definido no começo.

Então é colocado um botão que ao clicar, irá chamar a função que irá ativar o foco do input, sem precisar clicar nele.

Então é encerrado o parênteses do 'return' e logo abaixo é encerrado o componente de função.

# Listas no React

As listas são usadas para exibir dados em um formato ordenado e usadas principalmente para exibir menus em sites. No React, as Listas podem ser criadas de maneira semelhante à que criamos listas em JavaScript.

Já a função `map()` é usada para percorrer as listas. No exemplo abaixo, a função `map()` usa a matriz informada e multiplica seus valores por 5. Então nós atribuímos os resultados da operação na nova matriz.

```
var numbers = [1, 2, 3, 4, 5];  
  
const multiplyNums = numbers.map((number)=>{  
  return (number * 5);  
});  
  
console.log(multiplyNums);
```

```
[5, 10, 15, 20, 25]
```

# Listas no React

Outra forma de exibir uma lista no React usando uma matriz é através da `<ul>`. Como neste caso:

```
import React from 'react';
import ReactDOM from 'react-dom';

const myList = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
const listItems = myList.map((myList) => {
  return <li>{myList}</li>;
});

ReactDOM.render(
  <ul> {listItems} </ul>,
  document.getElementById('app')
);

export default App;
```

# Explicando o código das listas

```
const myList = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];  
const listItems = myList.map((myList)=>{  
  return <li>{myList}</li>;  
});
```

Após realizar as devidas importações, é criado uma 'const' com o nome 'myList', cuja mesma abriga 5 valores de string.

Depois disso, é criado outra const com o nome de 'listItems' que irá usar a função 'map()' para percorrer todos os valores do 'myList'. Então é usado uma 'arrow function' dentro do map para retornar os mesmo valores dentro de uma <li>.

# Explicando o código das listas

```
ReactDOM.render(  
  <ul> {listItems} </ul>,  
  document.getElementById('app')  
);
```

Em seguida, é usado o método 'ReactDOM' junto com a função 'render' para exibir o código, e como primeiro argumento desta função ele recebe a 'listItems' dentro de uma <ul>, onde todos seus valores terão <li>.

Ele recebe como segundo argumento o local que irá exibir este código, ressaltando novamente de que não precisa ser uma <div>, nem ter o 'id' com o nome "root".



# Atributo key no React

Uma chave é um identificador exclusivo. No React, ele é usado para identificar quais itens foram alterados, atualizados ou excluídos das listas. É útil quando criamos componentes dinâmicos ou quando os usuários alteram as listas. Também ajuda a determinar quais componentes de uma coleção precisam ser renderizados novamente em vez de renderizar repetidamente todo o conjunto de componentes todas as vezes.

Será mostrado uma das formas de aplicar índices nos arrays.

```
const stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];

const updatedLists = stringLists.map((strList)=>{
  <li key={strList.id}> {strList} </li>;
});
```

# Explicando o código do atributo key

```
const updatedLists = stringLists.map((strList)=>{  
  <li key={strList.id}> {strList} </li>;  
});
```

Após declarar uma lista, é declarado outro array, dessa vez com o nome da lista e usando a função 'map()' para percorrer o array interior.

A palavra passada entre parênteses é um nome dado como parâmetro para receber os valores da outra lista e, também exibir o id delas através do atributo 'key' na exibição.

# Atributo key no React:

Caso sua lista não possua índices estáveis, você pode atribuir o argumento 'index' junto ao parâmetro que irá percorrer a lista, e então exibi-lo no lugar do '.id'. Por exemplo:

```
const stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];

const updatedLists = stringLists.map((strList, index)=>{
  <li key={index}> {strList} </li>;
});
```

**OBS:** O uso do atributo 'key' só é recomendado caso seja utilizado a função map.

# Atributo key no React

Apesar de que possa não parecer, o parâmetro passado dentro dos parênteses da função 'map()' tem um escopo de bloco. Logo, você pode utilizar o mesmo parâmetro em listas diferentes que vai funcionar do mesmo jeito.

```
const titlebar = (  
  <ol>  
    {props.data.map((show) =>  
      <li key={show.id}>  
        {show.title}  
      </li>  
    )}  
  </ol>  
)  
;  
  
const content = props.data.map((show) =>  
  <div key={show.id}>  
    <h3>{show.title}: {show.content}</h3>  
  </div>  
)  
;
```

# Fragmentos no React

Já foi ensinado que você precisa cobrir todos elementos de um componente em uma `<div>` caso você tenha mais de um elemento no mesmo.

No entanto, cobrir os elementos com uma `div` para exibí-los poderá acarretar em alguns problemas como formatação, baixo desempenho, etc.

Para resolver os problemas mencionados, foi adicionado o “fragmento” na versão 16.2 do React. Eles permitem agrupar uma lista de filhos sem adicionar nós extras ao documento.



# Fragmentos no React

```
<React.Fragment>  
  <h2> Hello World! </h2>  
  <p> Welcome to the JavaTpoint. </p>  
</React.Fragment>
```

Motivos para usar o fragment:

- ▶ Ele torna a execução do código mais rápida em comparação com a tag.
- ▶ Utiliza menos memória.

# Fragmentos no React: sintaxe curta

Uma forma mais rápida de declarar fragmentos no React é através da tag vazia, que aplica os mesmos efeitos.

```
<>
  <h2> Hello World! </h2>
  <p> Welcome to the JavaTpoint </p>
</>
```

No entanto, há algumas dificuldades com elas, como por exemplo o fato de que não é possível utilizar atributos neles. Nesse caso, você precisa declarar o fragmento completo, que é o 'React.Fragment'.

# Método map() no React

O 'map()' é uma espécie de coletor de dados onde os dados são armazenados na forma de pares, e cada um contém uma chave exclusiva (famoso índice ou id, que foram mencionados anteriormente).

Não é possível armazenar dois tipos de valores iguais no 'map()'.

Ele é usado principalmente para pesquisa rápida e pesquisas de dados.

Neste exemplo, a função 'map()' usa um array de números e dobra seus valores. Atribuímos o novo array retornada por 'map()' à variável doubleValue e a exibimos.

```
var numbers = [1, 2, 3, 4, 5];  
  
const doubleValue = numbers.map((number)=>{  
  return (number * 2);  
});  
  
console.log(doubleValue);
```

# Tabelas no React

Uma tabela é um arranjo que organiza as informações em linhas e colunas, serve justamente para armazenar e exibir dados.

O 'react-table' é um datagrid (elemento de interface gráfica que permite mostrar os dados e alguns widgets em forma de tabela com linhas e colunas.) leve, rápida e totalmente personalizável (JSX, modelos, estilos, etc). Ele precisa ser instalado junto com o React.

# Exemplo de utilização do datagrid

Primeiro, será importado o 'react-table' através do terminal de seu editor de código através do seguinte comando:

```
npm instalar react-table
```

Uma vez que tenhamos instalado o 'react-table', precisamos **importar** ele para o componente. Para fazer isso, abra o arquivo **src/App.js** e adicione o seguinte trecho.

```
import ReactTable from "react-table";
```



# Exemplo de utilização do datagrid

Vamos supor que temos os seguintes dados que precisam ser renderizados:

```
const data = [{  
  name: 'Ayaan',  
  age: 26  
},{  
  name: 'Ahana',  
  age: 22  
},{  
  name: 'Peter',  
  age: 40  
},{  
  name: 'Virat',  
  age: 30  
},{  
  name: 'Rohit',  
  age: 32  
},{  
  name: 'Dhoni',  
  age: 37  
}]
```

# Exemplo de utilização de datagrid

Junto com os dados, também precisamos especificar as informações da coluna e seus atributos.

```
const columns = [{  
  Header: 'Name',  
  accessor: 'name'  
},{  
  Header: 'Age',  
  accessor: 'age'  
}]
```

# Exemplo de utilização de datagrid

Dentro do método de renderização, precisamos vincular esses dados com o 'react-table' e, em seguida, retornar o mesmo dentro do componente de classe.

```
return (  
  <div>  
    <ReactTable  
      data={data}  
      columns={columns}  
      defaultPageSize = {2}  
      pageSizeOptions = {[2,4, 6]}  
    />  
  </div>  
)
```

# Exemplo de utilização do datagrid

Agora o código do **src/App.js** irá se parecer com isso:

```
import React, { Component } from 'react';
import ReactTable from "react-table";
import "react-table/react-table.css";

class App extends Component {
  render() {
    const data = [
      {
        name: 'Ayaan',
        age: 26
      },
      {
        name: 'Ahana',
        age: 22
      },
      {
        name: 'Peter',
        age: 40
      },
      {
        name: 'Virat',
        age: 30
      },
    ]
```

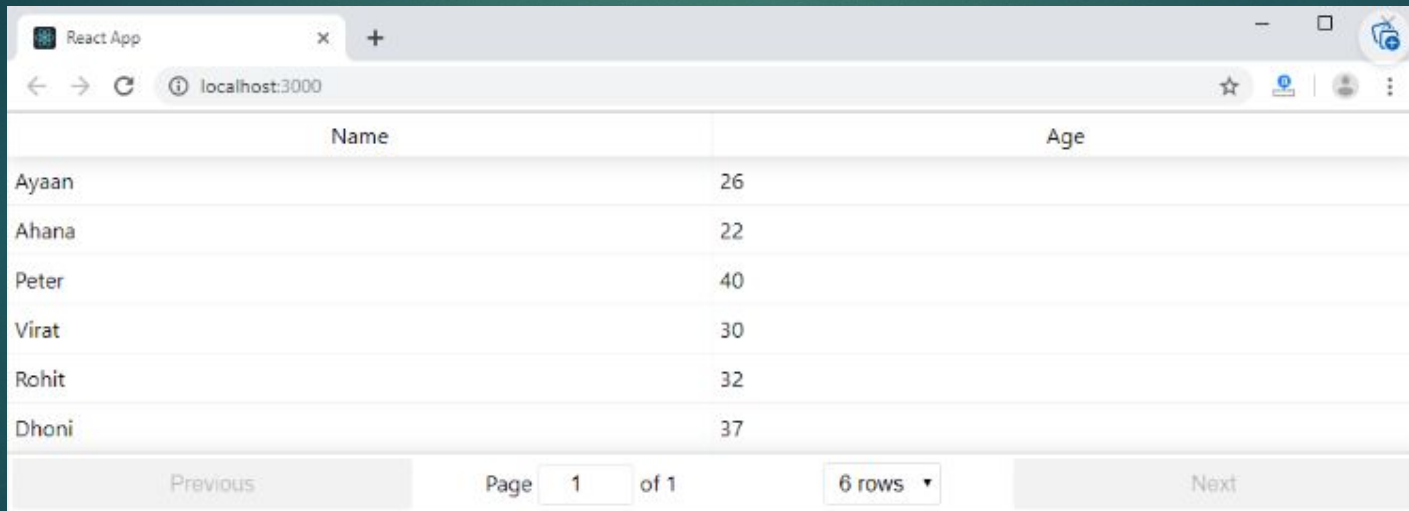
```
      name: 'Rohit',
      age: 32
    },
    {
      name: 'Dhoni',
      age: 37
    }
  ]
  const columns = [
    {
      Header: 'Name',
      accessor: 'name'
    },
    {
      Header: 'Age',
      accessor: 'age'
    }
  ]
```

```
    return (
      <div>
        <ReactTable
          data={data}
          columns={columns}
          defaultPageSize = {2}
          pageSizeOptions = {[2,4, 6]}
        />
      </div>
    )
  }
}

export default App;
```

# Exemplo de utilização de datagrid

O resultado será esse:



The screenshot shows a web browser window with the title 'React App' and the address bar displaying 'localhost:3000'. The browser window contains a data grid with two columns: 'Name' and 'Age'. The grid displays six rows of data. Below the grid, there is a pagination bar with the following controls: 'Previous', 'Page 1 of 1', '6 rows' (with a dropdown arrow), and 'Next'.

Name	Age
Ayaan	26
Ahana	22
Peter	40
Virat	30
Rohit	32
Dhoni	37

Previous Page 1 of 1 6 rows Next



# Componentes de ordem superior no React

Também conhecido como HOC. Trata-se de uma técnica avançada para reutilizar a lógica de componentes. É uma função que pega um componente e retorna em um novo.

O principal objetivo dele é decompor a lógica do componente em funções mais simples e menores que podem ser reutilizadas conforme necessário.

Sua sintaxe é a seguinte:

```
'const novoComponente = HigherOrderComponent(outroComponente).'
```

# Exemplo de uso do HOC

Será explicado em um exemplo uma das utilidades do HOC.

Criação de Função

```
function add (a, b) {  
  return a + b  
}  
  
function higherOrder(a, addReference) {  
  return addReference(a, 20)  
}
```

Chamada de função

```
higherOrder(30, add) // 50
```

# Explicando o exemplo do HOC

No exemplo anterior, criamos duas funções `add()` e `higherOrder()`. Agora, oferecemos a função `add()` como um argumento para a função `higherOrder()`. Para invocá-la, renomeie-o como `addReference` na função `higherOrder()` e invoque-o.

Regras do HOC:

- ▶ Não deve usá-lo dentro do método `'render()'` de um componente.
- ▶ HOCs não funciona para `'refs'` pois ele não passa um parâmetro ou argumento.

# Portais no React

Um portal fornece uma maneira de renderizar um elemento fora de sua hierarquia de componentes, ou seja, em um componente separado.

Antes da versão do React 16.0, era muito complicado renderizar o componente filho fora de seu componente pai. Caso fizemos isso, ele quebra a convenção em que um componente precisa ser renderizado como um novo elemento e seguir uma hierarquia pai-filho. No React, o componente pai sempre quer ir para onde seu componente filho vai. É por isso que foi criado o conceito do portal no React.

# Portais no React

Sua sintaxe é a seguinte:

`ReactDOM.createPortal(filho, contêiner).`

Aqui, o primeiro argumento (filho) é o componente, que pode ser um elemento, cadeia de caracteres ou fragmento, e o segundo argumento (contêiner) é um elemento DOM.

Às vezes, queremos inserir um componente filho em um local diferente no DOM. Isso significa que o React não quer criar uma nova `<div>`. Podemos fazer isso criando o portal React.

```
render() {  
  return ReactDOM.createPortal(  
    this.props.children,  
    myNode,  
  );  
}
```



# Explicando o exemplo do portal

```
render() {  
  return ReactDOM.createPortal(  
    this.props.children,  
    myNode,  
  );  
}
```

Este código fala que o componente está retornando o método do 'createPortal', onde envia como primeiro parâmetro o componente filho através da palavra-chave 'this'. Além disso, ele envia como segundo parâmetro (depois da vírgula) o local onde será exibido através do id deste elemento.

# Usando o react portal

Para conseguir usá-lo, será necessário instalar o mesmo no terminal do seu editor de código com o seguinte comando:

```
npm install react-portal --save
```

Depois disso, abra o arquivo 'App.js' e insira o seguinte trecho de código:

```
import React, {Component} from 'react';
import './App.css'
import PortalDemo from './PortalDemo.js';

class App extends Component {
  render () {
    return (
      <div className='App'>
        <PortalDemo />
      </div>
    );
  }
}

export default App;
```

# Usando o react portal

A próxima etapa é criar um componente de portal e importá-lo no arquivo App.js, através do arquivo importado anteriormente: 'PortalDemo.js'.

```
import React from 'react'
import ReactDOM from 'react-dom'

function PortalDemo(){
  return ReactDOM.createPortal(
    <h1>Portals Demo</h1>,
    document.getElementById('portal-root')
  )
}

export default PortalDemo
```

# Usando o react portal

Agora, abra o arquivo 'Index.html' e adicione um elemento `<div id="portal-root"></div>` para acessar o componente filho fora do nó raiz.

```
<div id="root"></div>  
<div id="portal-root"></div>
```

E pronto, agora poderá acessar partes específicas de seu componente no elemento em que quiser sem precisar todo o código.

# Limite de erro no React

No passado, caso nosso código tivesse algum erro, o React iria exibir uma mensagem de erro por toda a tela, que impediria a visualização dos usuários e até mesmo dos desenvolvedores. Pensando nisso foi implementado um limite de erros, para que esta mesma mensagem só seja exibida quando houver atingido este determinado limite.

Os limites de erro são componentes do React que capturam erros JavaScript em qualquer lugar do nosso aplicativo, registram esses erros e exibem uma interface do usuário de fallback. Ele não quebra toda a árvore de componentes do aplicativo e só renderiza a interface do usuário de fallback sempre que ocorrer um erro em um componente. Os limites de erro detectam erros durante a renderização em métodos de ciclo de vida de componentes e construtores de toda a árvore abaixo deles.



# Limite de erro no React

Para um aplicativo simples, podemos declarar um limite de erro uma vez e usá-lo para todo o aplicativo. Para um aplicativo complexo que tem vários componentes, podemos declarar vários limites de erro para recuperar cada parte de todo o aplicativo.

Para implementar o limite de erros você precisa:

- ▶ **Passo-1:** Crie uma classe que estenda o componente React e passe os adereços dentro dele.
- ▶ **Passo-2:** Agora, adicione o método `componentDidCatch()` que permite capturar erros nos componentes abaixo deles na árvore.
- ▶ **Passo-3:** Em seguida, adicione o método `render()`, que é responsável por como o componente deve ser renderizado. Por exemplo, ele exibirá a mensagem de erro como "Algo está errado".

# Exemplo de implementação do limite de erros

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    Ele atualizará o estado para que a próxima renderização mostre a interface do usuário de fallback.
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    Ele vai pegar erro em qualquer componente abaixo. Também podemos registrar o erro em um serviço de relatório de erros.
    logErrorToMyService(error, info);
  }

  render() {
    if (this.state.hasError) {
      return (
        <div>Something is wrong.</div>;
      );
    }
    return this.props.children;
  }
}
```

# Implementando o limite de erros

**Passo-4:** Agora, podemos usá-lo como um componente regular. Adicione o novo componente em HTML, que você deseja incluir no limite de erro. Neste exemplo, estamos adicionando um limite de erro em torno de um componente com o nome 'MyWidgetCounter'.

```
<ErrorBoundary>  
  <MyWidgetCounter/>  
</ErrorBoundary>
```

# Usando loops em arrays

Quando queremos construir qualquer aplicativo Web, é muito importante ter o conhecimento de lidar com uma matriz de dados. será mostrado dois exemplos de loops com arrays:

```
import React from 'react';

function App() {

  const myArray = ['Jack', 'Mary', 'John', 'Krish', 'Navin'];

  return (
    <div className="container">
      <h1> Example of React Map Loop </h1>

      {myArray.map(name => (
        <li>
          {name}
        </li>
      ))}

    </div>
  );
}

export default App;
```

# Explicando o primeiro exemplo de loops em um array

```
function App() {  
  const myArray = ['Jack', 'Mary', 'John', 'Krish', 'Navin'];
```

Nesta parte nós criamos uma função de nome 'App', dentro criamos um array com o nome 'myArray'.

```
  return (  
    <div className="container">  
      <h1> Example of React Map Loop </h1>
```

Após isso, é informado que esta função irá retornar uma div com a classe 'container', e um h1



# Explicando o primeiro exemplo de loops em um array

```
{myArray.map(name => (  
  <li>  
    {name}  
  </li>  
  )))  
</div>
```

Então é criado dentro de chaves {} o método 'map()', que irá percorrer todos os valores do array 'myArray'.

Para cada valor percorrido ele irá atribuí-lo a uma <li>.

Após isso ele encerra a div 'container'.

# Explicando o segundo exemplo de loops em um array

```
function App() {  
  
  const students = [  
    {  
      'id': 1,  
      'name': 'Jack',  
      'email': 'jack@gmail.com'  
    },  
    {  
      'id': 2,  
      'name': 'Mary',  
      'email': 'mary@gmail.com'  
    },  
    {  
      'id': 3,  
      'name': 'John',  
      'email': 'john@gmail.com'  
    },  
  ],  
};
```

```
return (  
  <div className="container">  
    <h1> Example of React Map Loop </h1>  
  
    <table className="table table-bordered">  
      <tr>  
        <th>ID</th>  
        <th>Name</th>  
        <th>Email</th>  
      </tr>  
  
      {students.map((student, index) => (  
        <tr data-index={index}>  
          <td>{student.id}</td>  
          <td>{student.name}</td>  
          <td>{student.email}</td>  
        </tr>  
      ))}  
  
    </table>  
  
  </div>
```

# Explicando o segundo exemplo de loops em um array

Primeiro, é criada uma função com o nome de 'App'. Depois disso, é criado dentro dela um objeto com o nome 'students' que recebe como propriedade o 'id', 'name' e 'email'.

Ela preenche todas estas propriedades com os valores.

**OBS:** o ? foi um erro no código mesmo.

```
function App() {  
  
  const students = [  
    {  
      'id': 1,  
      'name': 'Jack',  
      'email': 'jack@gmail.com'  
    },  
    {  
      'id': 2,  
      'name': 'Mary',  
      'email': '?mary@gmail.com'  
    },  
    {  
      'id': 3,  
      'name': 'John',  
      'email': 'john@gmail.com'  
    },  
  ]  
}
```

# Explicando o segundo exemplo de loops em um array

Após isso, a função irá retornar uma div de nome 'container' que dentro irá conter um `<h1>` e uma tabela.

```
return (  
  <div className="container">  
    <h1> Example of React Map Loop </h1>  
  
    <table className="table table-bordered">  
      <tr>  
        <th>ID</th>  
        <th>Name</th>  
        <th>Email</th>  
      </tr>
```

# Explicando o segundo exemplo de loops em um array

Então é usado o método 'map()' para percorrer o objeto 'students', o método 'map' envia como primeiro argumento 2 parâmetros (students e index). e então usa uma 'arrow function' para dizer que cada valor do objeto será anexado em um <td> através da propriedade passada junto com o objeto.

```
{students.map((student, index) => (  
  <tr data-index={index}>  
    <td>{student.id}</td>  
    <td>{student.name}</td>  
    <td>{student.email}</td>  
  </tr>  
))}
```

Após isso, é encerrado o método 'map', a div e a função.



# Caixas de seleção no React

Às vezes, temos que definir várias caixas de seleção com base nos requisitos do usuário. Podemos definir as opções para selecionar frutas, e os usuários podem selecioná-las de acordo com suas escolhas. Se os usuários quiserem selecionar mais de uma opção ou várias opções na lista, eles também poderão fazer isso. Nesse caso, somos obrigados a colocar várias caixas de seleção no ReactJS. Para isso, o exemplo a seguir nos ajudará a entender o uso de várias caixas de seleção no react.

# Exemplo das caixas de seleção

```
import React, { Component } from 'react';
import { render } from 'react-dom';
```

```
class App extends Component {
  constructor() {
    super();
    this.state = {
      categories: [
        {id: 1, value: "Angular"},
        {id: 2, value: "React"},
        {id: 3, value: "PHP"},
        {id: 4, value: "Laravel"}
      ],
      checkedItems: new Map()
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
}
```

```
handleChange(event) {
  var isChecked = event.target.checked;
  var item = event.target.value;

  this.setState(prevState => ({ checkedItems: prevState.checkedItems.set(item, isChecked) }));
}

handleSubmit(event) {
  console.log(this.state);
  event.preventDefault();
}

render() {
  return (
    <div>
      <h1> Examples of Multiple Checkbox in React </h1>
```

```
<form onSubmit={this.handleSubmit}>
  {
    this.state.categories.map(item => (
      <li>
        <label>
          <input
            type="checkbox"
            value={item.id}
            onChange={this.handleChange}
          /> {item.value}
        </label>
      </li>
    ))
  }

  <br/>
  <input type="submit" value="Submit" />
</form>
</div>
```

# Explicando o código das caixas de seleção

É criado um componente de classe onde é definido um objeto de nome 'categories' pelo 'this.state'.

Este objeto possui 2 propriedades, sendo 'id' e 'value'.

Todo item deste objeto é envolto de chaves.

```
class App extends Component {  
  constructor() {  
    super();  
    this.state = {  
      categories: [  
        {id: 1, value: "Angular"},  
        {id: 2, value: "React"},  
        {id: 3, value: "PHP"},  
        {id: 4, value: "Laravel"}  
      ],  
    }  
  }  
}
```

# Explicando o código das caixas de seleção

```
},  
  checkedItems: new Map()  
};  
  
this.handleChange = this.handleChange.bind(this);  
this.handleSubmit = this.handleSubmit.bind(this);
```

Então, é definido um objeto Mapa de nome 'checkedItems' e é encerrado o 'this.state'.

Após isso, é acessado duas funções **posteriores** ( que serão definidas logo abaixo). que elas irão acessar o objeto 'categories' pelo 'this' que foi enviado como parâmetro de ambas funções.

Então é encerrado o construtor.

# Explicando o código das caixas de seleção

```
handleChange(event) {  
  var isChecked = event.target.checked;  
  var item = event.target.value;  
  
  this.setState(prevState => ({ checkedItems: prevState.checkedItems.set(item, isChecked) }));  
}
```

Após encerrar o construtor, é definida a função 'handleChange', onde é enviado como parâmetro o 'event'.

Dentro dela é criado duas variáveis, a 'isChecked' que recebe o 'event' como parâmetro em seu estado de checado. Outra variável é o 'item' que dará um valor a esta caixa de seleção.

Então elas são ligadas ao objeto 'checkedItems' através do 'this.setState'.



# Explicando o código das caixas de seleção

```
handleSubmit(event) {  
  console.log(this.state);  
  event.preventDefault();  
}
```

Após isso, é criada a outra função, que receberá o 'event' como parâmetro também, ela irá apenas exibir o objeto no console, e será aquela que definirá o estado padrão do formulário.

# Explicando o código das caixas de seleção

Então, depois de definir as duas funções, é declarado o 'render' para renderizar na página o código que estiver dentro dele. Dentro do render, será retornado uma div, um <h1> e um formulário, que quando interagido irá ativar a função 'handleSubmit'.

Dentro do formulário é usado o método 'map' para percorrer o objeto 'categories' e colocar cada um de seus valores em um input do tipo 'checkbox' onde receberá seus valores como parâmetro.

```
render() {  
  return (  
    <div>  
      <h1> Examples of Multiple Checkbox in React </h1>  
  
      <form onSubmit={this.handleSubmit}>  
  
        {  
          this.state.categories.map(item => (  
            <li>  
              <label>  
                <input  
                  type="checkbox"  
                  value={item.id}  
                  onChange={this.handleChange}  
                /> {item.value}  
              </label>  
            </li>  
          ))  
        }  
      </form>  
    )  
  )  
}
```

# Explicando o código das caixas de seleção

```
<br/>  
<input type="submit" value="Submit" />  
</form>  
</div>
```

E no final é criado um input do tipo 'submit' que irá ativar o estado 'onSubmit' definido no formulário.

Após isso é encerrado o formulário e o componente de classe.

# Projetos práticos para fazer usando React (obrigatórios)

1. Form Validation in React Js

Link: <https://www.youtube.com/watch?v=N3815jc3ur8>

2. React Project: Photo Voting App

Link: <https://www.youtube.com/watch?v=EVTu97uww0Y>

3. ReactJs CRUD APP | React Tutorial | Full Stack Tutorial

Link: <https://www.youtube.com/watch?v=-quobUzNmuA>

4. Step Progress Widget | React JS | UI Components

Link: <https://www.youtube.com/watch?v=Ha5LORTxiKQ>

5. Food Recipe React App In Telugu | React JS Projects In Telugu

Link: <https://www.youtube.com/watch?v=JbNkCwFDPVA>

6. ReactJS Calculator App | Simple ReactJs Calculator

Link: <https://www.youtube.com/watch?v=hpfDRnijdPE>

7. Build A Todo List App with React from Scratch in 2022 | | CRUD App

Link: <https://www.youtube.com/watch?v=dD0MdMRVHoo>

# Projetos práticos para fazer usando React (obrigatórios)

8. Build a CALCULATOR APP in REACT JS | A React JS Beginner Tutorial

Link: <https://www.youtube.com/watch?v=oiX-6Y2oGjI>

9. ReactJS Multi-Step Form Tutorial - React Hooks Tutorial

Link: <https://www.youtube.com/watch?v=wOxP4k9f5rk>



# Projetos práticos para fazer usando React (opcionais)

1. React js project from scratch in Hindi

Link: [https://www.youtube.com/playlist?list=PL8p2l9GklV46FX2Uik\\_rVlbN\\_nSx0Wc43](https://www.youtube.com/playlist?list=PL8p2l9GklV46FX2Uik_rVlbN_nSx0Wc43)

2. React Project Tutorial: Build a Responsive Portfolio Website

Link: <https://www.youtube.com/watch?v=hYv6BM2fWd8>

3. React js Shopping Cart for beginner | Easy Way to Add to cart reactjs

Link: <https://www.youtube.com/watch?v=B0E2esA5nQo>

4. Crie um Quiz com React.js - Projeto de React para iniciantes

Link: <https://www.youtube.com/watch?v=HlkbeikH8cs>

5. Shopping Cart | React.js Project with explanation | Beginner

Link: <https://www.youtube.com/watch?v=P9-zbdMTwiM>

6. Create a Digital Clock using React JS in Hindi | React Mini Project #2

Link: <https://www.youtube.com/watch?v=AaIEKyqaaqSk>