**Elementos Objetos**

O React usa elementos do DOM virtuais, que se tornam reais mediante um renderizador.

Um elemento DOM é definido com um sim,ples objeto literal.

O renderizador alvo da nossa aplicação é descrito da seguinte forma:

**<div id="app">**

**</div>**

Criamos um elemento para renderização dentro do React usando:

**var reactElement = {}**

Esse pequeno trecho de código é o que receberá nossas informações e posteriormente mostraremos na tela utilizando:

**ReactDOM.render(reactElement, renderTarget)**

Lembrando que estamos inserindo o reactElement que criamos e dentro do renderTarget precisamos colocar o elemento DOM alvo da renderização, da seguinte forma:

**var renderTarget = document.getElementById("app")**

Nossos elementos precisam de atributos, alguns como:

**$$typeof: magicValue,**

**ref: null,**

**type: h1,**

**props:{}**

Dentro de props, definimos algumas propriedades que devem ser passadas para o elemento:

**props:{**

**children: “Hello, world!”,**

**className: “nomeElemento”**

**style: {**

**textAlign: “center”,**

**},**

**}**

Manipuladores de eventos podem ser adicionados como propriedades também:

**props:{**

**children: “Hello, world!”,**

**className: “nomeElemento”**

**style: {**

**textAlign: “center”,**

**},**

**onClick: function(umEventoNãoRegular) { alert(“Click”) },**

**}**

Aqui um elemento com todas as configurações que vimos acima:

**var elementoQualquer = {**

**$$typeof: magicValue,**

**ref: null,**

**type: “p”,**

**props: {**

**children: “Um parágrafo legal”,**

**},**

**}**

**Fábrica de Elementos**

Utilizamos o React.createElement() para simplificar a criação dos nossos objetos ou atributos, com isso não precisamos utilizar $$type/Symbol, o React.createElement() faz isso automaticamente. Além disso, isso torna o código menos verboso.

Lembrar: React.createElement() necessita de tipo, propriedades e filhos.

**var reactElement = React.createElement(“h1”, {**

**className: “abc”,**

**style:{**

**textAlign: “center”,**

**},**

**onClick: function() { alert(“click”) },**

**}, “Hello, world!”)**

De forma simplificada, também fazemos:

**var umElementoQualquer = React.createElement(“p”, null, “Um paragrafo legal.”)**

O segundo argumento é a propriedade objeto(props), e deve ser null caso vazia.

**JSX**

O JSX deve ser convertido para JavaScript, para fazer isso utilizamos Babel. O Babel também converte JSX para ES5.

JSX é um jeito idiomática de se criar elementos, basicamente é XHTML com {} para uso de conteúdos dinâmicos (javascript). Por exemplo, classes voltam a se chamar className.

**var reactElement =**

**<h1**

**className = “abc”**

**style={ {textAlign: “center”}}**

**onClick={function() { alert(“Click”) } }>**

**Hello, world!**

**</h1>**

É o mesmo que:

**reactElement =**

**React.createElement(**

**“h1”,**

**{ className: “abc”, style: { textAlign: “center”},**

**onClick: function() { alert(“click”) } },**

**“Hello, world!”**

**)**

JSX brilha em elementos simples, por exemplo:

**var umElementoQualquer = <p>Um paragrafo legal.</p>**

É o mesmo que:

**umElementoQualquer = React.createElement(“p”, null, “Um paragrafo legal.”)**

**Elementos Aninhados**

Basicamente, o aninhamento de elementos consiste em misturar ambas as tecnologias (HTML, Javascript) e utilizá-las de maneira mais dinâmica.

De forma simples, se temos um JSX escrito assim:

**var reactElement =**

**<div className = “abc”>**

**<h1>Hello</h1>**

**<h2>world</h2>**

**</div>**

Podemos fazer uma mistura muito curiosa com diversas possibilidades se incrementarmos Javascript ao código:

**var minhaClasse = “abc”**

**function meuTexto () {**

**return “world”**

**}**

Inserindo esse Javascript ao JSX original, teríamos:

**reactElement =**

**<div className={minhaClasse}>**

**<h1>Hello {10 \* 10}</h1>**

**<h2>{meuTexto}</h2>**

**</div>**

O Javascript pode conter elementos HTML também:

**var elementoAninhado = <h2>world</h2>**

**reactElement =**

**<div>**

**<h1>Hello</h1>**

**{elementoAninhado}**

**</div>**

Também podemos “espalhar”, por assim dizer um objeto como propriedades

**var properties = {**

**className: “abc”,**

**onClick: function(){ alert(“click”) },**

**}**

**reactElement =**

**<div {...properties}>**

**<h1>Hello</h1>**

**<h2>world</h2>**

**</div>**

**Componentes**

O maior destaque do React é seu sistema de componentes, os componentes são usados para encapsular elementos e seu comportamento.

Aqui vemos os elementos stand alone:

**var data = “world”**

**var reactElement =**

**<div>**

**<h1>Hello</h1>**

**<h2>{data}</h2>**

**</div>**

Abaixo os elementos estão encapsulados em uma função simples. Deve sempre iniciar com letra maiúscula e retornar um elemento raiz com ou sem elementos aninhados: (antes do React 16)

**function MyComponent(){**

**var data = “world”**

**return(**

**<div>**

**<h1>Hello</h1>**

**<h2>{data}</h2>**

**</div>**

**)**

**}**

Depois do React 16 os componentes podem retornar um array de elementos.

Uma advertência importante é que, similar ao que fazemos ao renderizar um array, devemos adicionar uma chave única para cada elemento. \*\*não esquecer as virgulas.

**function MyComponent(){**

**var data = “world”**

**return [**

**<h1 key=”hello”>Hello</h1>,**

**<h2 key=”data”>{data}</h2>**

**]**

**}**

Desde o React 16.2 podemos usar componentes “wrapper” especiais chamados fragmentos, que se comportam da mesma maneira, e removem a necessidade de criar chaves explicitas (os fragmentos fazem isso de forma automática) e as virgulas\*\*

**function MyComponent(){**

**var data = “world”**

**return (**

**<React.Fragment>**

**<h1>Hello</h1>**

**<h2>{data}</h2>**

**</React.Fragment>**

**)**

**}**

Uma função de componente pode ser usada como um elemento também:

**reactElement = <MyComponent/>**

A tradução para uma chamada React.createElement() é: (null significa que nehuma propriedade foi criada)

**reactElement = React.createElement(MyComponent, null)**

Para referenciar um div interna ao react

**var umElementoQualquer = <div/>**

É convertida para:

**umElementoQualquer = React.createElement(“div”, null)**

**Propriedades**

Os components, assim como elementos, podem usar propriedades também

**function MyComponents(props){**

**return(**

**<div className={props.className}>**

**<h1>Hello</h1>**

**<h2>{props.customData}</h2>**

**</div>**

**)**

**}**

**var reacElement = <MyComponent className=”abc” customData=”world”/>**

O que também funciona utilizarmos com o objeto é o operador de espalhamento que vimos antes

**var props = {**

**className: “abc”,**

**customData: “world”,**

**}**

**var reactElement = <MyComponent {...props}/>**

Isso nos permite criar componentes com um conteúdo mais dinâmico utilizando array, por exemplo

**var planets = [“earth”, “mars”, “venus”]**

Um array consiste em uma única variável recebendo inúmeros conteúdos ou filhos, cada filho precisa ter uma chave unida, a chamada abaixo vai fazer essa atribuição no index para nós e retornar as propriedades informadas:

**var elements = planets.map(function(planet, index){**

**return <MyComponent className=”myClass” customData={planet} key={index}/>**

**})**

**var reactElement = <div>{elements}</div>**

**Tipos de Propriedades**

Os componentes são criados para encapsular coisas que deveriam estar juntas em apenas um único lugar ou para reuso.

Fazer o reuso requer que o usuário do componente proveja as mudanças corretas, de forma que possamos definir um tipo para cada propriedade de definir padrões para elas.

**function MyComponents(props){**

**return(**

**<div className={props.className}>**

**<h1>Hello</h1>**

**<h2>{props.customData}</h2>**

**</div>**

**)**

**}**

Podemos fazer isso adicionando a propriedade do propTypes para a função componente para deixar que ela valide suas propriedades. Aqui definimos o tipo:

**MyComponent.propTypes = {**

**customData: PropTypes.string,**

**}**

Aqui definimos o padrão:

**MyComponent.defaultProps = {**

**customData: “default”,**

**className: “default-class”,**

**}**

**var reactElement = <MyComponent/>**

Obs.: a partir do React 16 os PropTypes foram removidos.

**Exemplo de Propriedade**

Um exemplo prático de componente, formata uma data e retorna um span contendo a string formatada

**function DateSpan(props){**

**var date = props.date,**

**day = date.getDate(),**

**month = date.getMonth() +1,**

**year = date.getFullYear()**

**return <span>{day}.{month}.{year} </span>**

**}**

Devemos prover o elemento data e o componente que criamos faz a formatação:

**var reactElement = <DateSpan date={new Date()}/>**

**Tratamento de valor da input**

Esse tópico não se encaixa em um conceito básico, mas se faz extremamente necessário ao realizar entradas de informações numa tela.

**Passo 1: Manipulação de Eventos**

Simular ao JavaScript, porém a função pode ser inserida dentro do elemento do JSX ou como propriedades separadas que podemos chamar dentro do elemento (...propriedade):

**<button type=”button” onClick={() => alert(“Hi”)}>**

**Say Hi!**

**</button>**

Ou

**const propriedades = {**

**type:”button”,**

**onClick: function() => {alert(“Hi”)}**

**}**

**<button {…propriedades}>**

**Say Hi!**

**</button>**

Para lidar com o envio de informações de um formulário, ou campo de dados, vamos criar um componente e adicionar esse componente ao elemento JSX, assim como fizemos com a propriedade, porém utilizando componentes ao invés disso.

**function manipulaEnvio(e){**

**e.preventDefault()**

**}**

**<form onSubmit={manipulaEnvio}>**

**Passo 2: Definindo State e Hooks**

Hooks(ganchos) nos permitem utilizar o estado e outros recursos do React sem necessariamente criar uma classe. Com esses recursos, podemos alterar o estado de um componente ou até mesmo algum elemento do código. Nesse caso usaremos o State (manipulação do estado de um componente).

Para usar o state Hook precisamos importá-lo:

**import React, {useState} from ‘react’**

Qualquer outra função de gancho deverá ser importada.

O useState retorna suas coisas: o estado inicial e uma função que será usada para alterar o estado inicial.

**const [nome, setNome] = useState(“Usando hooks!”)**

Aqui estamos fazendo duas coisas, definindo um valor inicial para nome e definindo a função que usaremos posteriormente. O useState está ligado diretamente ao uso de um array, pois dados serão alterados. Portanto sempre trabalharemos com ele para que os dados sejam armazenados de forma separada.

Ao criar um Hook, precisamos passá-lo ao elemento do código:

**<input**

**type=”text”**

**id=”novo-input”**

**className=”input”**

**value={nome}**

**/>**

O que queremos ao final é alterar o valor inicial para uma string vazia.

**const [nome, setName] = useState(“”)**

Simples assim!

Lembrar que o nome já deve ter sido definido como um dado de array no index.js, estamos manipulando ele aqui como um props.

**Passo 3: Ler a entrada do usuário**

Para alterar o valor do nome, precisamos capturar a sua entrada quando ele digitar, para isso criamos uma outra função para controlar mudanças, mais uma vez vamos ouvir essa mudança e informar dentro do meu elemento.

**function manipulaMudanca(e){**

**setNome(e.target.value)**

**}**

Aqui, estamos setando o valor em setNome com o target (que identifica se um evento foi disparado ou não)

**<input**

**type=”text”**

**id=”novo-input”**

**className=”input”**

**value={nome}**

**onChange={manipulaMudanca}**

**/>**

Nosso elemento recebe a função criada em onChange, uma propriedade que recebe mudanças. Então, quando meu input estiver recebendo uma mudança, estaremos lendo a entrada do usuário.

**Passo 4: Manipulando o envio**

Após fazermos a leitura do estado no campo de entrada, precisamos fazer o envio do dado para onde eu quero que ele seja mostrado, manipulado ou alterado.

Para isso precisamos adicionar o valor informado utilizando uma outra função que adicionará a tarefa executada em um array, ou seja, outro estado.

**function manipulaEnvio(e){**

**e.preventDefault()**

**props.addTask(nome)**

**setNome(“”)** //limpamos o campo após o envio

**}**

Para que a tarefa seja adicionada, que é o que queremos, temos de criar também no App (renderizador) a função para fazer isso.

**function addTask(nome){**

**alert(nome)**

**}**

Essa função deve ser chamada dentro do meu return, senão, não aparecerá no navegador

**<Campos addTask={addTask} />**

**Passo 5: Adicionar a tarefa**

O App.js é o arquivo principal, ele renderiza nossos componentes e os monstra no navegador.

Precisamos importar o useState nele também.

**import React, {useState} from ‘react’**

Precisamos passar as definições do props para um novo estado que criaremos, esse estado armazenará as tarefas.

**const [lista, setLista] = useState(props.nomes)**

Assim podemos chamar essas propriedades dentro da nossa lista de nomes.

Agora nossa lista de nomes deverá ser alterada para receber o que for inserido no campo que recebe esse valor nome, como já manipulamos ele acima, agora vamos coletar esse valor e inserir numa tela. Continuamos a usar o map para percorrer o array.

**const listaNomes = lista.map((nomeObjeto) => (**

**<Lista**

**nome={nomeObjeto.nome}**

**/>**

**))**

Para adicionar o nome na lista alteramos nossa addTask da seguinte forma:

**function addTask(nome){**

**const novoNome = {nome}**

**setLista([...lista, novoNome])**

**}**

\*\*além da parametrização para manuseio da lista, precisamos definir um id para cada entrada.

Necessário instalação de uma biblioteca do próprio Javascript, é o nanoid:

**npm install nanoid**

Importar no App.js:

**Import {nanoid} from “nanoid”**

Adicionamos no index.js:

**const DADOS = [**

**{id: “nome-0”, nome: “Cassia”},**

**{id: “nome-1”, nome:”João”}**

**]**

Agora, na função de adicionar o nome referenciaremos o id utilizando a biblioteca.

**const novoNome = { id: `nome-${nanoid()}`, nome}**

**Passo 6: Concluindo uma tarefa e sincronizando o navegador com nossos dados**

Ao alterar uma caixa de seleção, ele não altera o estado em nosso aplicativo react. Isso significa que que o navegador e o aplicativo estão fora de sincronia.

A propriedade de controla as conclusões de uma tarefa é a propriedade completed, por isso a adicionaremos nas nossas funções.

Em index.js:

**const DADOS = [**

**{id:”nome-0”, nome:”Cassia”, completed: true},**

**{id:”nome-1”, nome:”João”, completed: false}**

**]**

Configurar o checkbox na Lista:

**<input**

**id:={props.id}**

**type:”checkbox”**

**defautChecked={props.completed}**

**/>**

**<label>**

**{props.nome}**

**</label>**

E no nosso App.js:

**const novoNome = {id: `nome-${nanoid()}`, nome, completed: true}**

Agora com nosso completed todo configuradinho, vamos criar uma função, que irá controlar o estado da nossa checkbox.

function alternaNomesCompleted(id){

}