**REACT HORA DE CODAR**

**16/09 – O QUE É REACT?**

O React é uma biblioteca baseada em JavaScript e mantida pelo Facebook. Essa biblioteca é muito utilizada para desenvolver aplicações front-end, principalmente SPA (Single Page Application)\*.

A arquitetura do React é baseada em componentes. Cada pequeno pedaço de uma aplicação representa um componente (botão, cabeçalho, rodapé, etc). Quando esses componentes se unem, eles formam a aplicação completo. Essa arquitetura é excelente, uma vez que esses componentes podem ser reusados, facilitando a manutenção do código.

\* Uma SPA é uma “aplicação de página única”, cujo carregamento é necessário apenas uma vez. Depois, quando o usuário navega entre as páginas da aplicação, não é feito o recarregamento. Isso otimiza o site e garante a performance dele.

**CRIANDO UMA APLICAÇÃO REACT**

Existem duas maneiras bem comuns de criar uma aplicação em React, as quais estão listadas a seguir:

**npx:** O npx é um executor de scripts que vem junto com o Node. Esse é o jeito mais fácil de criar uma aplicação React, mas também o mais demorado. Para criar um projeto React com npx, basta digitar no terminal o seguinte comando:

*npx create-react-app <nome-do-projeto>*

Com isso, o npx criará uma pasta com o nome fornecido, e dentro dessa pasta, ele fornecerá os arquivos em uma estrutura básica para iniciar o projeto.

**Vite:** essa maneira é mais rápida do que usando o npx. Basta digitar:

*npm create vite@latest*

Com isso, basta selecionar as opções desejadas (nome do projeto, framework, etc) e seguir os procedimentos.

**ESTRUTURA BÁSICA DE UM PROJETO REACT**

**node\_modules:** pasta descartável, aonde as dependências do nosso projeto ficam.

**public:** arquivos estáticos e HTML de inicialização.

**src:** onde os nossos componentes e as regras de negócio deles serão programadas.

**src/index.js:** arquivo de inicialização do React.

**src/App.jsx:** arquivo do componente principal do projeto.

**17/09 - ESTRUTURA DE UM COMPONENTE REACT**

Ao criarmos um componente, o nome de seu arquivo deve seguir o padrão de digitação Pascal Case (letras inicias maiúsculas, inclusive a primeira). Por convenção, é comum criarmos uma pasta chamada **components** dentro de **./src**, e armazenarmos nossos componentes lá.

É importante importar o React antes de inicializar o componente, usando **import React from ‘react’.**

**Exportando um componente:** Todo componente em React é uma função. Como os componentes são reutilizados em outros componentes, nós devemos exportá-lo, ou seja, permitir que outros componentes o usem. Para isso, usamos a palavra **export default** (exportar por padrão).

export default function FirstComponent(){

    return(

        <div>

            <h1>Oi, esse é meu primeiro componente!</h1>

        </div>

    )

}

**Importando um componente:** para usar um componente em outro arquivo, basta usar o comando **import X from ‘./components/x’,** onde X é o nome do componente.

import FirstComponent from './components/FirstComponent';

**Instanciando um componente:** para de fato utilizar um componente, você precisa instanciar ele dentro de um return de outro componente. Basta tratar o componente como uma tag de auto-fechamento, como no caso a seguir:

function App() {

  return (

    <div *className*="App">

**<FirstComponent />**

    </div>

  );

}

**ENTENDENDO O JSX**

JSX, abreviação de "JavaScript XML," é uma extensão de sintaxe usada no React. O JSX permite que você escreva código parecido com HTML dentro do seu código JavaScript, o que torna a criação de componentes de interface do usuário mais intuitiva e fácil de ler.

**Principais características do JSX:**

**Tag englobadora:** em expressões JSX (que sempre ficam dentro do **return** de um componente), é permitido haver apenas um elemento pai, ou seja, não se pode retornar duas tags irmãs. Uma tag maior tem que englobar as outras tags dentro dela, e geralmente essa tag é uma <div>.

**Sintaxe similar ao HTML:** O JSX se assemelha à sintaxe do HTML, o que facilita a criação e a manutenção de componentes de interface do usuário.

**Expressivo:** Permite a incorporação de expressões JavaScript diretamente dentro do código JSX, dentro de chaves {}. Isso é útil para renderizar dinamicamente conteúdo.

**Componentes:** Você pode usar o JSX para criar componentes reutilizáveis, que encapsulam parte da interface do usuário e podem ser compostos em estruturas mais complexas.

**Compilação:** O código JSX deve ser transpilado para JavaScript padrão antes de ser executado no navegador. Isso geralmente é feito com ferramentas como o Babel em projetos React.

**COMENTÁRIOS**

**Dentro de uma função no React:**

// linha única

/\*

múltiplas linhas

\*/

**Dentro de uma expressão JSX:**

{ /\*

múltiplas linhas (dentro de um par de chaves)

\*/}

**TEMPLATE EXPRESSIONS**

Template expressions é um recurso que nos permite interpretar código JavaScript dentro de uma expressão JSX. Para isso, basta utilizar um par de chaves { }.

<h1>Me chamo {nome}. Eu tenho {idade} anos!</h1>

**EVENTOS**

Para dispararmos eventos em algum elemento JSX, devemos aplicado o atributo **onEvent** nesse elemento, onde event se refere ao evento específico.

Quando um evento acontece, uma função é chamada. Geralmente, o nome dessas funções são constituídas por **handleNomeEvento().** Essa função pode ser referenciada usando template expression, através do par de chaves. Veja um exemplo:

<button *onClick*={handleClick}>Clique aqui!</button>

**FUNÇÕES DE RENDERIZAÇÃO**

Funções de renderização retornam expressões JSX e são capazes de renderizar conteúdos dependendo de uma determinada condição. Ou seja, o JSX pode ser renderizado com conteúdos diferentes a depender do valor de uma variável, por exemplo. Isso é muito útil para exibir conteúdos diferentes para usuários autenticados e não autenticados.

Criando a função:

const renderSomething = (*boolean*) => {

     if (*boolean*){

         return <h1>Renderizando true</h1>

     } else {

         return <h1>Renderizando false</h1>

    }

}

Chamando a função:

return(

    <div>

        {renderSomething(true)}

        {renderSomething(false)}

    </div>

)

**18/09 - MANUSEIO DE IMAGENS EM PROJETO REACT**

Podemos trabalhar com imagens em nosso projeto React de duas maneiras diferentes:

**1: pasta public:** Nesta maneira, as imagens ficam salvas na pasta public do nosso projeto. Com isso, podemos usar tags <img> com o atributo src sendo **‘./nome.jpg’**. O React já conecta as imagens da pasta public direto na raiz do projeto, não sendo necessário digitar o caminho dessa pasta no endereço do arquivo.

**2: pasta assets:** Nesse outro jeito, a imagem fica numa pasta chamada **assets** dentro da pasta **src.** Podemos importar essa imagem normalmente (como se estivéssemos importando um componente) e usar ela dentro de uma template expression. no atributo src de uma tag <img> (exemplo: src={Gato})

**HOOKS**

Hooks são funções especiais do React que realizam tarefas específicas, como guardar e alterar o estado de algum dado, por exemplo (useState). Além dos hooks já presentes no React, podemos criar os nossos próprios hooks. Esses hooks são chamados de **custom hooks.**

Todos os hooks começam com a palavra **use** (useState, useEffect, useContext, useRef, useMemo, useCallback, etc). Quando formos usar algum hook do React, devemos importá-lo dessa maneira:

import { useState } from 'react';

**useState()**

O useState é um hook do React que permite que você adicione e gerencie o estado de alguma variável em componentes. Em termos simples, ele é usado para manter variáveis que podem mudar ao longo do tempo e fazer com que o componente seja renderizado novamente quando essas variáveis mudam.

A sintaxe é formada da seguinte maneira:

const [banana, setBanana] = useState(false)

A variável em que desejamos gerenciar o estado (ou seja, uma variável que pode ser modificada) é a constante **banana.** Já a função que **vai** alterar e gerenciar o estado dessa variável é a **setBanana.**

Atribuímos esses dois itens desestruturados de um array do hook useState(), e o parâmetro que esse hook recebe é o valor inicial da variável banana (false).

Quando formos mudar o valor da variável banana, devemos chamar a função setBanana, e passar como parâmetro o novo valor da variável banana. Dessa maneira, as variáveis poderão ter seu valor alterado e o componente vai respeitar isso sempre que for renderizado.

**RENDERIZAÇÃO DE LISTAS**

Uma tarefa muito comum realizada em ambientes profissionais é a renderização de listas, ou seja, utilizar o método **map()** em cima de um array de tal forma que, para cada registro desse array, um item de lista seja criado utilizando expressões JSX.

No React, é bem simples renderizarmos listas, como se pode ver a seguir:

const [list, setList] = useState(['Matheus', 'Pedro', 'Josias'])

    return(

        <div>

            <ul>

                {list.map((*item, index)* => (

                    <li key={index}>{*item*}</li>

                ))}

            </ul>

        </div>

    )

Há um adendo no código acima: a arrow function dentro de map é estabelecida por meio de parênteses (), não chaves.

**PROPRIEDADE KEY**

A propriedade key é um requisito que o React precisa para renderizar listas. Key representa uma chave única para cada registro em um array ou objeto, e serve para ajudar na renderização do componente. Podemos usar como key o próprio id que vem de registros em bancos de dados (no caso de objetos) ou então usarmos o index (no caso de arrays). Veja um exemplo a seguir:

const [users] = useState([

        {id: 1, name: 'Matheus', age: 32},

        {id: 2, name: 'João', age: 37},

        {id: 3, name: 'Edcarlos', age: 47}

    ])

    return(

        <div>

            {users.map(*item* => {

                <ul *key*={*item*.id}>

                    <li>{*item*.name}</li>

                    <li>{*item*.age}</li>

                </ul>

            })}

        </div>

    )

**PREVIOUS STATE**

"Previous state" (estado anterior) refere-se ao valor do estado antes de uma atualização. Em React, você pode acessar o estado anterior usando uma função no método useState, que recebe o estado anterior como argumento.

Veja o exemplo a seguir:

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

    const handleIncrement = () => {

        setCount((*prevCount*) => *prevCount* + 1);

    };

**prevCount** é o estado anterior e é usado para calcular o novo valor do count. Usar o estado anterior é importante em situações onde várias atualizações de estado podem ocorrer em rápida sucessão, garantindo que você esteja atualizando o estado com base no valor mais recente. Isso ajuda a evitar problemas de concorrência e garantir que as atualizações de estado sejam consistentes.

**PROPS**

As props são usadas para transmitir dados de um componente pai para um componente filho de forma que os componentes possam ser reutilizados com diferentes dados. Isso é excelente para deixar nossos componentes dinâmicos, reusáveis e personalizáveis.

As props se encontram nos parâmetros da função do componente. Veja o exemplo a seguir:

export default function ShowUsername(*props*){

    // componente filho

    return(

        <h2>O nome do usuário é {*props*.name}</h2>

    )

}

function App() {

// componente pai

  return (

      <*ShowUsername* *name*="Gabriel" />

  );

}

**DESTRUCTURING DE PROPS**

É comum passar mais de um prop em um componente. Para facilitar isso, o React nos permite desestruturas as props, usando o recurso Destructuring do JavaScript. Podemos fazer isso da seguinte maneira:

export default function CarDetails({*brand*, *km*, *color*}){

    return(

        <div>

            <h2>Detalhes do carro</h2>

            <ul>

                <li>Marca: {*brand*}</li>

                <li>Quilometragem: {*km*}</li>

                <li>Cor: {*color*}</li>

            </ul>

        </div>

    )

}

<*CarDetails* *brand*="VW" *km*={100000} *color*="Azul"/>

Dessa maneira, temos mais controle sobre as props que são passadas entre os componentes e não precisamos mais usar a notação ponto para se referir a props (props.brand, props.km, props.color).

Com as props, a reutilização de componentes começa a fazer muito sentido. Se temos dados de 1000 carros, não precisamos copiar tags <ul> e <li> passando os dados desses carros. Nós podemos **reaproveitar** nosso componente CarDetails, passando a informação de cada carro como props. Dessa maneira, nosso código fica bem mais padronizado, personalizável e fácil de dar manutenção.

**RENDERIZAÇÃO DE LISTAS + REAPROVEITAMENTO DE COMPONENTES**

Veja como podemos unir esses dois conceitos no código a seguir:

const cars = [

    {id: 1, brand: 'Ferrari', color: 'Amarelo', newCar: true, km: 0},

    {id: 2, brand: 'Kia', color: 'Roxo', newCar: false, km: 250000},

    {id: 3, brand: 'Lamborghini', color: 'Verde', newCar: false, km: 4000000},

  ]

Primeiro, temos um array de objetos com informações de vários carros.

{cars.map((*car*) => (

        <*CarDetails* *brand*={*car*.brand} *km*={*car*.km} *color*={*car*.color} *newCar*={*car*.newCar} />

))}

Logo em seguida, usamos o método **map** para renderizar um único componente (CarDetails) para cada item do array de carros. Com isso, exibimos na tela os registros de cada carro reaproveitando esse componente CarDetails.

**REACT FRAGMENT**

Um Fragment é uma maneira de renderizar vários elementos filhos em um único nó, sem a necessidade de criar um elemento HTML adicional para envolvê-los. Ele é representado por tags vazias <> e </>. Essas tags serão representadas como elemento pai, não alterando a estrutura do HTML com uma <div>, por exemplo.

return (

    <>

      <h1>Irmão 1</h1>

      <h2>Irmão 2</h2>

    </>

  );

**PROPS.CHILDREN**

Props.children é uma propriedade especial do React que permite que você passe elementos JSX como filhos para um componente pai. Isso é útil quando você deseja compor componentes e criar layouts mais flexíveis e reutilizáveis.

Quando você utiliza props.children, você permite que os componentes pais enviem qualquer conteúdo, incluindo outros componentes, texto, elementos HTML e muito mais, como filhos (children).

Veja o exemplo a seguir:

export default function Container(*props*){

    return(

        <div>

            {*props*.children}

            <h2>Meu valor é: {*props*.myValue}</h2>

        </div>

    )

}

A função do componente acima exibe props.children dentro dele. Vejamos o que vai ser esse props.children no componente em que ele é instanciado:

<*Container* *myValue*={17}>

    <h2>Titulo container</h2>

</*Container*>

Como vemos, o props.children se refere ao elemento <h2> que está dentro do componente quando ele é instanciando no componente pai.

**Lembre-se: props.children se refere ao que vem DENTRO do componente filho quando ele é instanciado no componente pai.**

**FUNÇÕES EM PROPS**

As funções podem ser passadas para as props normalmente. Para isso, basta criar a função no componente pai e enviar como prop para o componente filho. No componente filho, ela pode ser ativada por um evento, por exemplo.

Confira o exemplo a seguir:

export default function ExecuteFunction(*props*){

    return(

        <>

            <button *onClick*={*props*.func}>Clique aqui</button>

        </>

    )

}

Aqui, determinamos que será passada um props com o nome de **func,** que levará a função a ser executada.

function showMessage(){

    console.log('Evento do componente pai!')

}

return (

    <div *className*="App">

      <*ExecuteFunction* *func*={showMessage} />

    </div>

  )

Agora, criamos a função showMessage() e passamos em função como prop para o componente filho ExecuteFunction. Podemos ver então que, quando o botão do componente for clicado, a função showMessage será executada, que foi a função passada com prop.

**STATE LIFTING**

"State lifting" (ou "levantamento de estado") é um conceito no React que se refere à prática de mover o estado de um componente filho para um componente pai. Isso é feito para compartilhar e gerenciar o estado comum entre vários componentes filhos a partir de um componente pai comum. Essa técnica é usada para manter o estado sincronizado entre os componentes e tornar o código mais organizado e eficiente.

Vamos explicar isso com um exemplo:

Imagine que você tem um aplicativo de lista de tarefas em React. Você tem um componente pai chamado App e dois componentes filhos chamados ListaDeTarefas e FormularioDeTarefa. O FormularioDeTarefa é responsável por adicionar novas tarefas à lista, enquanto o ListaDeTarefas exibe as tarefas existentes.

Aqui está um exemplo simplificado de como o "state lifting" pode ser aplicado:

import React, { useState } from 'react';

function App() {

  const [tarefas, setTarefas] = useState([]);

  const adicionarTarefa = (*novaTarefa*) => {

    setTarefas([...tarefas, *novaTarefa*]);

  };

  return (

    <div>

      <*FormularioDeTarefa* *onAdicionarTarefa*={adicionarTarefa} />

      <*ListaDeTarefas* *tarefas*={tarefas} />

    </div>

  );

}

function FormularioDeTarefa(*props*) {

  const [novaTarefa, setNovaTarefa] = useState('');

  const handleSubmit = (*event*) => {

*event*.preventDefault();

*props*.onAdicionarTarefa(novaTarefa);

    setNovaTarefa('');

  };

  return (

    <form *onSubmit*={handleSubmit}>

      <input

*type*="text"

*value*={novaTarefa}

*onChange*={(*e*) => setNovaTarefa(*e*.target.value)}

*placeholder*="Adicionar tarefa"

      />

      <button *type*="submit">Adicionar</button>

    </form>

  );

}

function ListaDeTarefas(*props*) {

  return (

    <ul>

      {*props*.tarefas.map((*tarefa*, *index*) => (

        <li *key*={*index*}>{*tarefa*}</li>

      ))}

    </ul>

  );

}

export default App;

Neste exemplo:

* O estado ‘tarefas’ é mantido no componente App. O FormularioDeTarefa recebe uma função onAdicionarTarefa como uma prop que permite adicionar uma nova tarefa ao estado ‘tarefas’ do componente App.
* Quando um novo item é adicionado usando o FormularioDeTarefa, ele chama a função onAdicionarTarefa, que é definida no componente App. Isso atualiza o estado ‘tarefas’ no componente App.
* O componente ListaDeTarefas recebe o estado ‘tarefas’ como uma prop e exibe a lista de tarefas.

Esse é um exemplo simples de "state lifting", onde o estado é mantido em um componente pai e compartilhado com componentes filhos para garantir que eles estejam sincronizados. Isso torna o código mais organizado e facilita o gerenciamento de estados compartilhados em aplicativos React.

**REACT E CSS**

**CSS GLOBAL:** é utilizado para estilizar diversos elementos em comum (padronizar um mesmo estilo para todos os componentes) ou fazer um reset no CSS. Para criar estilos globalmente, alteramos o arquivo **index.css**, que fica na pasta **/src.**

**CSS DE COMPONENTE:** é utilizado para estilizar um componente em específico. Geralmente, é criado um arquivo CSS com o mesmo nome do componente, e esse arquivo é importado no próprio componente.

O React já cria um exemplo dessa técnica com o App.jsx (componente) e App.css (folha de estilo para o componente).

Mas é importante notar o seguinte: esse método não é **scoped,** ou seja, o CSS vaza para outros elementos fora do componente. Ao usar esse método, é recomendado que se use classes, para garantir que os estilos aplicados ali se limitem apenas àquele componente.

**INLINE STYLE:** técnica onde, por meio do atributo **style**, conseguimos aplicar regras de estilo diretamente no elemento. Porém, essa técnica é fortemente **desencorajada,** uma vez que dificulta a manutenção do código e deixa ele imprevisível.

**INLINE STYLE DINÂMICO:** mesma técnica do inline style, porém aplica estilo baseado em uma condicional, por meio de um if ternário.

**CLASSES DINÂMICAS:** aplica lógica para mudar a classe CSS de um elemento. Essa lógica é aplicada por meio de um if ternário.

Essa abordagem é mais interessante que o CSS inline, uma vez que as classes estarão isoladas no arquivo de CSS, resolvendo o problema de organização do código.

**CSS MODULES:** técnica **scoped**, cujo estilo aplicado por meio dessa técnica pertencerá exclusivamente a um componente em especificado. O nome do arquivo CSS dessa técnica é **Componente.module.css.** Depois, nós importamos esse arquivo no componente.

**20/09 – FORMULÁRIOS EM REACT**

Para criar formulários em React, também iremos utilizar a tag <form> do HTML e seguir o mesmo padrão de criação de formulários. Porém, algumas diferenças serão aplicadas, as quais estão listadas a seguir:

* Na tag <label>, ao invés de usarmos a propriedade for, iremos utilizar **htmlFor.** Isso é algo imposto pelo React, necessário para evitar conflitos com a estrutura de repetição **for** do JavaScript.
* Não utilizaremos a propriedade **action** no formulário, pois ele será enviado de maneira assíncrona via AJAX.
* É possível utilizar a tag <label> de maneira que ela envolva o input a que se refere. Isso simplifica a estrutura HTML, não perde a semântica e faz com que digitemos menos código.

**CONTROLLED INPUTS**

Para gerenciarmos da melhor forma os dados dos inputs, podemos armazenar o estado deles. Para isso, usamos o hook **useState().**

Como propriedade **value** do input, nós referenciamos a variável do useState que criamos. Além disso, podemos aplicar o evento **onChange** no input e usar a função do useState para alterar o estado da variável para o valor atual do input, toda vez que ele mudar (referenciado por **e.target.value**). Veja tudo isso na prática.

Const [name, setName] = useState(‘’)

<input  *type*=”text”

*value*={name}

*onChange*={€ => setName(*e*.target.value)} />

**ENVIO DO FORMULÁRIO**

Aplicamos o evento onSubmit ao formulário, disparado quando o usuário o envia. Chamamos uma função que vai tratar de validar o formulário, enviá-lo ao servidor, resetar o formulário, dentre outras ações. Nessa função, usamos o **e.preventDefault()** para prevenir o comportamento padrão do navegador e impedir o page reload.

<form *onSubmit*={handleSubmit}>

**RESET DO FORMULÁRIO**

Para limpar os campos do formulário, basta atribuir uma string vazia aos states e pronto! Geralmente, fazemos isso nas funções de submit do formulário, como no exemplo a seguir:

function handleSubmit(*e*){

*e*.preventDefault()

    setName('')

    setEmail('')

}

**25/09 – useEffect()**

O useEffect() é um hook do React usado para lidar com efeitos colaterais do componente. Em termos mais técnicos, o useEffect funciona como um **observador,** e é usado para realizar tarefas em seu componente quando certas condições ocorrem.

Você configura o useEffect passando uma função que contém as ações que deseja executar e uma lista de dependências (coisas que o useEffect deve observar). Quando as condições nas dependências mudam, o useEffect executa as ações que você especificou.

Usamos muito o useEffect para buscar dados em uma API, manipular o DOM, entre outros.

**RESGATANDO DADOS DE UMA API**

Para resgatar dados de uma API, precisamos utilizar vários recursos, os quais estão listados a seguir:

* Ter um local para salvá-los (useState)
* Renderizar a chamada a API apenas uma vez (useEffect)
* Um meio de fazer a requisição assíncrona (Fetch)

const baseUrl = 'http://localhost:3001/products'

function App() {

  const [products, setProducts] = useState([])

  useEffect(() => {

    async function fetchData(){

      const res = await fetch(baseUrl)

      const data = await res.json()

      setProducts(data)

  }

    fetchData()

  }, [])

}

return (

      <ul>

        {products.map(*product* => (

          <li *key*={*product*.id}>{*product*.name} - R${*product*.price}</li>

        ))}

      </ul>

);

**ENVIANDO DADOS PARA UMA API**

* Para adicionar um dado, vamos precisar resgatá-lo de um formulário com o useState.
* Após isso, reunir eles em uma função após o submit e enviar um request de POST para a nossa API.
* O processo é bem parecido com o de resgate de dados, mas agora estamos **enviando dados.**

function App() {

  const [name, setName] = useState('')

  const [price, setPrice] = useState('')

  const handleSubmit = async (*e*) => {

*e*.preventDefault()

    const product = {

      name,

      price

    }

    const res = await fetch(baseUrl, {

      method: 'POST',

      headers: {

         "Content-Type" : "application/json"

      },

      body: JSON.stringify(product)

    })

  }

return(

<form *onSubmit*={handleSubmit}>

    <label>

      <span>Nome</span>

      <input *type*="text" *value*={name} *name*="name" *onChange*={*e* => setName(*e*.target.value)} />

    </label>

    <label>

      <span>Preço</span>

      <input *type*="number" *value*={price} *name*="price" *onChange*={*e* => setPrice(*e*.target.value)} />

    </label>

    <button *type*="submit">Criar produto</button>

  </form>

)

**CUSTOM HOOKS**

Custom Hooks são funções personalizadas que criamos para serem reaproveitadas em nossos componentes. Geralmente, esses hooks ficam na pasta **hooks/**, e os nomes desses hooks devem seguir o padrão **use**AlgumaCoisa. Criamos essa função, sempre lembrando de exportá-la para usar em nossos componentes.

No curso, fizemos um custom hook para fazer o resgate de dados vindo de uma API (**useFetch**).

**ESTADO DE LOADING**

Em nossos hooks, podemos identificar quando o estado de loading começa e termina, ou seja, identificar o estado de loading entre a requisição e o recebimento da resposta.

Para isso, usamos o **useState().** Assim que o estado de loading começa, podemos configurar o estado da variável loading para **true.** Quando esse processo de loading terminar, mudamos o estado para **false.**

Com esse toggle de estado, podemos ter acesso ao estado dessa variável em nossos componentes e, a partir disso, gerar uma animação enquanto loading está como **true,** por exemplo. Além disso, podemos desabilitar botões e inputs enquanto a requisição é processada.

useEffect(() => {

      const fetchData = async () => {

          setLoading(true)

          const res = await fetch(*url*)

          const data = await res.json()

          setData(data)

          setLoading(false)

      }

      fetchData()

}, [*url*, callFetch])

**26/09 – REACT ROUTER**

O React Router é uma biblioteca do React que nos permite que criar e gerenciar uma estrutura de **rotas** em nossas aplicações. Ou seja, ele nos permite criar aplicações de página única (SPAs) com várias rotas, onde cada rota corresponde a um componente React específico.

Isso facilita a criação de interfaces de usuário com navegação fluida entre diferentes “páginas” sem precisar recarregar a página inteira.

**\* Rotas** definem o comportamento da nossa aplicação em resposta a diferentes URLs ou caminhos.

**TRABALHANDO COM REACT ROUTER**

Para configurar o React Router, vamos ter que instalar o **react-router-dom** do npm e, em seguida, importar três elementos:

* **BrowserRouter:** define que ali, dentro dele, haverá o roteamento de componentes. Geralmente, fica entre o cabeçalho e o rodapé do site.
* **Routes/Switch:** define as rotas, envolvendo vários componentes Route.
* **Route:** usado para definir uma rota em nossa aplicação. Nele, configuramos o path (caminho) e element (componente a ser renderizado).

return (

    <div *className*="App">

      <h1>React Router</h1>

      <*BrowserRouter*>

        <*Routes*>

          <*Route* *path*="/" *element*={<*Home* />} />

          <*Route* *path*="/about" *element*={<*About* />} />

        </*Routes*>

      </*BrowserRouter*>

    </div>

);

Na rota raiz (/), será exibido o componente Home. Já na rota “/about”, será exibido o componente About. Esses componentes devem ser previamente importados.

**COMPONENTE LINK**

O componente Link é usado para criar links de navegação em nossa aplicação, permitindo que os usuários cliquem em um link para acessar diferentes rotas. Ele possui o atributo **to,** que especifica para qual rota o link deve direcionar.

O Link é muito semelhante a uma tag <a> no HTML. Mas então porque não se usa <a>? Não usamos a tag <a> porque essa tag faz o reload da página, enquanto o <Link /> apenas troca o componente. Isso deixa a navegação muito mais rápida e fluida.

import { Link } from "react-router-dom"

export default function Navbar(){

    return(

        <nav>

            <*Link* *to*="/">Home</*Link*>

            <*Link* *to*="/about">Sobre</*Link*>

        </nav>

    )

}

**ROTAS DINÂMICAS**

Com o React Router, podemos criar rotas dinâmicas, ou seja, rotas que podem possuir diferentes parâmetros em sua URL. Veja um pequeno exemplo de rota dinâmica:

<*Route* *path*="/products/:id" *element*={<*Product* />}/>

Essa rota é considerada dinâmica, pois **id** é um argumento que varia, identificado por suceder dois-pontos (:).

Ao renderizarmos uma grande lista de produtos, podemos usar o componente Link para conectar a rota de cada produto com um componente possuindo suas informações:

<*Link* *to*={`products/${*item*.id}`}>Detalhes</*Link*>

Por fim, no componente que irá exibir a informação de cada produto de maneira dinâmica, podemos utilizar o hook useParams() para obter o **id** que o usuário acessou na rota, e então ter acesso a informações do produto e carrega-lo individualmente..

**PÁGINA NOT FOUND (404)**

Podemos criar uma página 404 facilmente com o React Router. Basta criarmos o componente da página, e no arquivo onde as rotas são definidas, definir o atributo **path** de Route como **\*.** Dessa maneira, qualquer rota que não exista cairá nesse componente.

<*Route* *path*="\*" *element*={<*NotFound* />} />

**COMPONENTE NAVLINK**

O componente NavLink é importado do React Router e tem a mesma função do componente Link: criar links de navegação para as rotas da nossa aplicação. A diferença é que o NavLink é mais usado para barras de navegação, onde você tem vários links em uma lista.

Com o NavLink, temos acesso ao atributo **isActive**, que identifica se aquela rota está sendo acessada nesse momento. Com base nesse booleano, podemos atribuir uma classe dinâmica para caso a rota do elemento esteja ativo (ou para caso não esteja).

<*NavLink* *to*="/" *className*={({ *isActive* }) => (*isActive* ? 'esta-ativo' : 'nao-ativo')}>Home</*NavLink*>

**useParams() e useSearchParams()**

Esses dois recursos do React, também importados do React Router, nos permitem obter o valor do parâmetro de uma URL.

**useParams:** usado para obter parâmetros de ID que vem na URL. (ex1)

**useSearchParams:** usado para obter parâmetros de query que vem na URL. (ex2)

Com esses recursos, obtemos os dados de algum item em específico da nossa API e podemos fazer uma busca no sistema, para exibir alguma informação.

Exemplo 1 (query): /produtos/**q?camisa**

Exemplo 2 (id): /produtos/**1**

ID PARAMS

const { id } = useParams()

const url = `http://localhost:3002/products/${id}`

QUERY PARAMS

const [searchParams] = useSearchParams()

const url = `http://localhost:3002/products?${searchParams}`

**REDIRECIONAMENTO DE URLs COM NAVIGATE**

Quando queremos que uma rota aponte para outra rota, nós podemos usar o componente **Navigate** que vem no React Router. Isso irá redirecionar o usuário da rota que ele digitou para a rota apontada. Veja um exemplo:

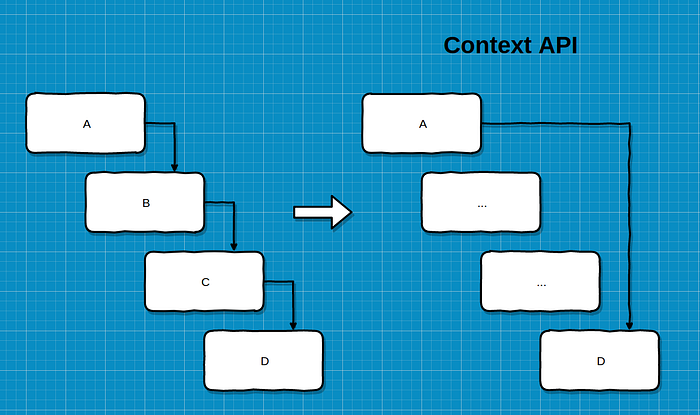
<*Route* *path*="/about" *element*={<*About* />} />

<*Route* *path*="/company" *element*={<*Navigate* *to*="/about" />} />

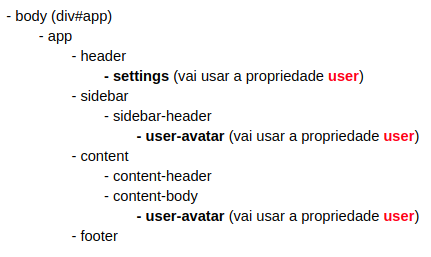
Dessa maneira, quando o usuário acessa a rota ‘/company’, é como se ele estivesse acessando a rota ‘/about’, e o componente <About /> será renderizado.

**CONTEXT API**

Context API **é um recurso do React que fornece uma maneira de passar dados entre componentes sem ter que passar props manualmente para eles.** É muito útil quando você quer passar dados de um componente A para um componente D, por exemplo. Ela foi introduzida no React para lidar com a chamada "prop drilling" (quando você precisa passar uma prop por vários níveis de componentes).



Usaremos Context API quando queremos que certos dados sejam “globais”, ou seja, que todos os componentes (até os mais baixos na hierarquia) acessem eles. Veja um exemplo a seguir:



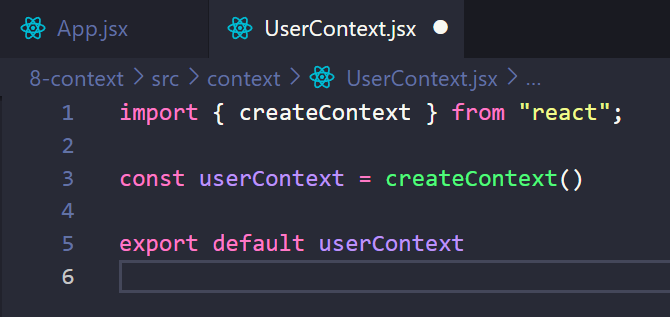
Perceba que os dados do usuário são utilizados em diferentes componentes. Desta maneira, deveríamos passar os dados entre os componentes, até atingirmos os interessados (settings e user-avatar). Porém, quando fizermos isso, notamos que não tem por que alguns componentes terem acesso a propriedade user. Além do mais, para evitar passar user para todos os componentes, deveríamos armazenar esse dado no componente de maior hierarquia (no nosso caso, app) o que gera uma manutenção trabalhosa, e logo não é uma boa prática deixar esse dado ali.

Com a context API nesse cenário, podemos criar um contexto que armazene os dados do usuário, e a partir disso, os componentes interessados nesse dado “consumirão” os dados do contexto!

**Criando o contexto:**

ETAPA 1: CRIANDO COMPONENTE DE CONTEXTO

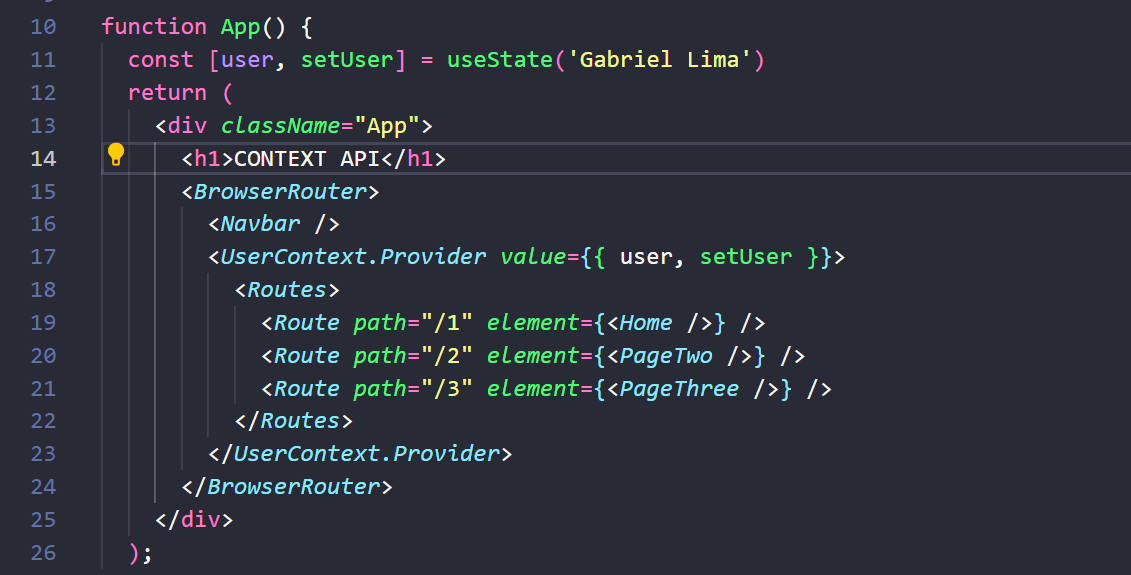
* 1. Criamos uma pasta dentro de /src chamada **context,** onde iremos criar todos os nossos contextos.
* 2. Importamos a função **createContext()** do React e armazenamos a chamada dela dentro de uma variável. Como parâmetro dessa função, pode-se incluir um valor padrão.
* 3. **Exportamos por padrão** a variável que criamos com a chamada da função.



**Criando o provider:** é a instância de um contexto. É um componente que provê os valores de um contexto para seus componentes filhos. Ele define um contexto e fornece valores que podem ser acessados por componentes dentro desse contexto.

ETAPA 2: CRIANDO PROVIDER

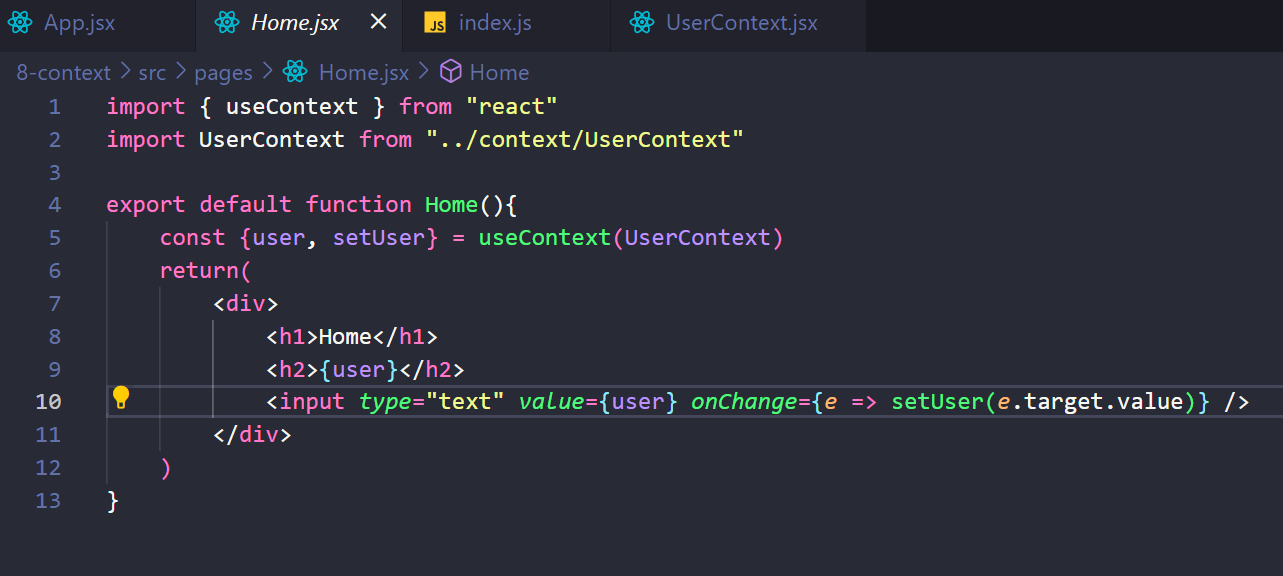
* 1. No componente principal **App.jsx**, envolvemos todos os componentes presentes ali com componente **UserContext.Provider**. Ele vai abraçar todo mundo.
* 2. Esse componente deve possuir um atributo chamado value, que armazena os dados de um contexto para serem usados dentro de todos os componentes que ele abrange. (Nesse exemplo, armazenamos a variável user com useState() e passamos isso como value).



**Usando os dados vindo do contexto nos componentes filhos:** para que algum componente envolvido em nosso Provider faça uso dos valores passados no contexto, fazemos o seguinte:

ETAPA 3: CONSUMINDO UM CONTEXTO (USANDO OS DADOS)

* 1. Importamos o hook **useContext()** do React.
* 2. Importamos o componente de contexto que criamos na etapa 1 (no caso desse exemplo, o UserContext)
* 3. Desestruturamos os dados que queremos, atribuindo eles á chamada da função **useContext()**, passando como parâmetro o componente que armazena o contexto (no caso desse exemplo, o UserContext).
* 4. Pronto! Agora, é só usar os dados nos componentes filhos.

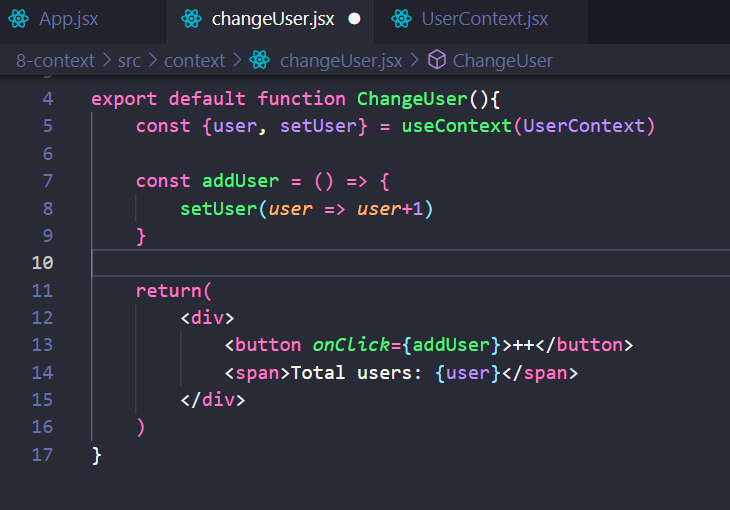


**Alterando os dados de um contexto:** para alterar o valor de um contexto, precisamos criar um componente que utilize a função de mudança do contexto. Esta mudança ocorrerá no Context e poderá ser consumida por todos os componentes que recebem o contexto.

Dessa forma, o ciclo da Context API será finalizado.

ETAPA 4: ALTERANDO O CONTEXTO

* 1. Em um novo componente chamado **ChangeUser (**só para exemplo**)**, importamos o **useContext** do React e o componente do nosso contexto.
* 2. Desestruturamos os dados que queremos alterar, atribuindo eles á chamada da função useContext(), passando como parâmetro o componente que armazena o contexto (no caso desse exemplo, o UserContext).
* 3. Fazemos alguma alteração nesses dados (no caso desse exemplo, usamos alteração de estado com useState()).
* 4. Bem simples e parecido com o consumo: a alteração de contexto reflete em todos os componentes que o consomem.



**04/10 – HOOKS DO REACT**

**useState()**

Esse é o principal hook do React. Seu objetivo é **gerenciar dados.** Com useState(), nós podemos consultar ou alterar algum dado, e isso faz com que o componente seja re-renderizado, algo que não acontece com a manipulação de variáveis tradicional.

A sintaxe do useState é essa:

const [name, setName] = useState('Pedro')

Nós usamos desestruturação de array e obtemos dois elementos: a variável (**name**) e a função que vai alterar o valor dessa variável (**setName**). Eles são retornados após a chamada do hook useState(). Além idsso, podemos definir um valor inicial para a variável criada (**name**)**,** esse valor inicial poderá ser definido como parâmetro

**\* useState() e controlled inputs:** podemos combinar o useState com inputs de formulário. Dessa forma, poderemos ter melhor controle dos dados e enviá-los de maneira fácil para o backend, por exemplo. Veja como fazer isso:

**1.** Primeiro, criamos variáveis com useState() para cada input do formulário. Exemplo: se há um input para que o usuário digita a idade, criamos uma variável com useState chamada **age** e sua função setter **setAge.**

**2.** Atribuímos como valor do input o valor da variável age.

**3.** Atribuímos um evento ao input chamado **onChange**, que é disparado toda vez que o valor do input é modificado. Como resposta a esse evento, chamamos a função **setAge** para alterar o valor de **age** para o que o usuário acabou de digitar (**e.target.value**). Dessa maneira, o estado da variável age será atualizado, e isso refletirá no valor do input.

**4.** Ao final, atribuímos mais um evento, dessa vez ao formulário. O evento **onSubmit** é disparado quando o formulário é enviado, e como resposta a esse evento, chamamos a função **handleSubmit()**, que irá tratar, validar e enviar os dados para o backend.

**useReducer()**

Esse hook tem o mesmo objetivo do useState: **gerenciar dados.**

A diferença é que, com useReducer, nós podemos executar uma função na hora da alteração do dado. Essa função pode criar uma lógica customizada para alterar o estado do dado.

Esse hook é utilizado em lógicas de estado mais complexas e avançadas.

Considerando como exemplo o desenvolvimento de uma lista de tarefas, a sintaxe do useReducer é a seguinte:

const [tasks, dispatchTasks] = useReducer(taskReducer, initialTasks)

**Tasks:** variável que vai armazenar todas as tarefas (array de objetos).

**dispatchTasks:** função que vai alterar algum valor no array de tarefas.

**taskReducer:** função que irá criar uma estrutura de escolha com **switch** e determinar o que será alterado no array de tarefas (adicionar, editar ou remover tarefa).

**initialTasks:** array inicial das tarefas.

Para ficar mais fácil de entender o useReducer, veja essa ilustração:

Imagine que você tem um brinquedo, como um carro de controle remoto, e você quer controlar esse brinquedo. Agora, vamos dizer que existem apenas dois botões que você pode pressionar: um botão "+1" e um botão "-1".

O useReducer é como uma pessoa que está observando esses botões e o brinquedo. Quando você pressiona o botão "+1", você envia uma mensagem para essa pessoa dizendo "aumente o número em 1". Quando você pressiona o botão "-1", você envia uma mensagem dizendo "diminua o número em 1".

A pessoa é muito boa em seguir essas mensagens. Ela tem uma contagem atual, começa com zero, e sempre faz o que você pede. Se você disser para aumentar, ela adiciona 1 à contagem. Se você disser para diminuir, ela subtrai 1 da contagem.

Então, o useReducer é como essa pessoa que ajuda você a controlar o número no seu brinquedo. Você envia ações (como pressionar os botões) para o useReducer, e ele atualiza o número para você com base nessas ações. Isso facilita o controle do número e garante que ele seja sempre atualizado corretamente.

Agora, vamos ver passo a passo do exemplo a seguir

1. Criamos nosso array inicial de tarefas.

const initialTasks = [

    {id: 1, text: 'Fazer alguma coisa'},

    {id: 2, text: 'Fazer outra coisa'}

]

**2.** Criamos a função taskReducer(), que vai determinar o que será alterado no array de tarefas (adicionar, por exemplo). Essa função recebe dois parâmetros: o **state**, que representa o primeiro elemento do array desestruturado de useReducer (tasks), e o **action** (um objeto passado como parâmetro do segundo elemento do array desestruturado de useReducer (função dispatch), que irá possuir um atributo chamado **type**, que determina o tipo de ação a ser tomada).

const taskReducer = (*state*, *action*) => {

   switch(*action*.type){

       case "ADD":

           const newTask = {

               id: Math.random(),

               text: taskText (variável do useState vinda de input)

           }

           setTaskText('')

           return [...*state*, newTask]

       default:

            return *state*

    }

}

**3.** Inicializamos o useReducer, passando os devidos parâmetros.

const [tasks, dispatchTasks] = useReducer(taskReducer, initialTasks)

**4.** Chamamos a função **dispatchTasks**, passando o objeto **action** como parâmetro e seu atributo **type**, para que uma tarefa seja adicionada.

dispatchTasks({ type: "ADD" })

**useEffect()**

O hook useEffect() é usado para lidar com efeitos colaterais do nosso código. Ele funciona como um observador dentro do componente. Ele “observa” as coisas acontecerem e pode realizar ações quando essas coisas mudarem.

Em termos mais técnicos, o useEffect é usado para **determinar quantas vezes e em quais condições uma tarefa será realizada no componente**.

useEffect(() => {

    // tarefa

}, [deps])

**\* Tarefa executada toda vez que o componente é re-renderizado**

Para isso, basta configurar o useEffect passando uma função dentro dele. Nessa função, se encontrará a respectiva tarefa, que será executada toda vez que o componente for re-renderizado.

**\* Tarefa executada apenas uma vez**

Para isso, basta configurar o useEffect passando como primeiro parâmetro uma função dentro dele (contendo a tarefa a ser executada), e como segundo parâmetro, um array de dependências vazio.

**\* Tarefa executada toda vez que um dado mudar**

Para isso, basta configura o useEffect passando como primeiro parâmetro uma função que contém as ações que deseja executar, e como segundo parâmetro, um array de dependências com o dado que, caso mude, o useEffect será chamado novamente. Isso nos fornece um maior controle de quando a função é executada ou não.

**\*\* Cleanup do useEffect**

O cleanup é uma limpeza de efeitos quando um componente é desmontado. Essa prática é importante para evitar vazamentos de momória e garantir que o componente seja eficiente e se comporte corretamente.

Num useEffect de setTimeout ou setInterval, por exemplo, nós retornamos uma função clearInterval ou clearTimeout, para interromper essa execução quando outro componente for renderizado.

Usamos mais o useEffect quando vamos fazer requisições assíncronas a alguma API ou então para manipular o DOM.

**05/10 – useContext()**

O hook useContext() é utilizado para **consumir o valor de um contexto na Context API.**

Para usá-lo, precisamos primeiro criar o contexto (**createContext**) e também o Provider desse contexto. Após isso, nós envolvemos com esse Provider, lá no componente App, os componentes que receberão os valores compartilhados, e então podemos executar o useContext() em cada componente para desestruturar um valor específico lá do contexto.

**ARQUIVO DO CONTEXTO**

// criação do contexto

export const SomeContext = createContext()

// criação do provider

export function SomeProvider({*value*, *children*}){

    return

<*SomeContext.Provider* *value*={*value*}>

{*children*}

</*SomeContext.Provider*>

}

// useContext para consumir o valor do contexto

export function useSomeContext(){

    return useContext(SomeContext)

}

**COMPONENTE APP**

<*SomeProvider* *value*={{fruit, setFruit}}>

  <*BrowserRouter*>

    <*Routes*>

      <*Route* *path*=”/” *element*={<*Home* />} />

      <*Route* *path*=”/about” *element*={<*About* />} />

    </*Routes*>

  </*BrowserRouter*>

</*SomeProvider*>

**COMPONENTE HOME**

const { fruit, setFruit } = useSomeContext()

**useRef()**

O hook useRef() tem como objetivo **gerenciar valores** e **criar referências para variáveis ou elementos do DOM, de maneira que essas variáveis ou elementos não re-renderizem o componente ao serem modificados.** É útil quando você precisa acessar diretamente elementos HTML ou quando deseja manter uma variável que não causa a re-renderização do componente quando é modificada.

O useRef é um objeto e seu valor está na propriedade **current.** Quando alguma referência criada pelo useRef é alterada, o componente não é re-renderizado, sendo esse o grande diferencial dele.

**USEREF COM VARIÁVEIS**

const numberRef = useRef(0)

useEffect(() => {

    numberRef.current = numberRef.current + 1

})

Criamos uma variável chamada numberRef, e usamos o useRef para referenciá-la. Quando essa variável é modificada com o useEffect(), o componente não é re-renderizado.

**USEREF COM ELEMENTOS DO DOM**

    const inputRef = useRef() 1

    return(

        <div>

           <input *type*="text" *ref*={inputRef} /> 2

           <button *onClick*={() => inputRef.current.focus()}> 3

Clique aqui para focar no input

</button>

        </div>

    )

}

1. Criamos uma variável para armazenar a referência de um input, chamada **inputRef.**

2. Conectamos essa variável de referência com o nosso input por meio do atributo ref.

3. Toda vez que o botão é clicado, conseguimos dar foco ao input obtendo a referência dele e chamando a função **focus.**

**useCallback()**

O hook useCallback() é usado para **memorizar uma função, fazendo com que ela NÃO seja executada a cada renderização do componente.** Em simples palavras, o useCallback visa a **performance e o desempenho** da nossa aplicação.

Esse hook é útil quando a função é passada como prop para um componente filho, pois garante que o componente filho não seja re-renderizado desnecessariamente quando a função muda.

Geralmente, usamos esse hook em dois casos:

1. Quando prezamos pela performance e queremos que uma função muito complexa não seja reconstruída toda vez.

2. Quando o próprio React nos indica que uma função deveria ser envolvida no useCallback para evitar problemas de performance.

useCallback(() => {

    return ['a', 'b', 'c']

}, [])

// Essa função será memorizada pelo React e não será executada de novo quando o componente for re-renderizado.

**useMemo()**

useMemo é um hook do React que permite **memorizar o resultado de um cálculo e armazená-lo para reutilização em outros lugares**. Ele é útil quando você tem um cálculo, que exige muito recurso computacional em um componente, e que não precisa ser recalculado sempre que o componente renderizar.

A ideia principal do useMemo é **economizar recursos computacionais**. Em vez de fazer cálculos repetidos a cada vez que o componente é renderizado, o useMemo calcula o valor apenas quando as coisas nas quais ele depende mudam.

Podemos dizer que o useMemo é semelhante ao useCallback, mas para variáveis.

A sintaxe do useMemo é essa:

const [number] = useState(0)

    const square = useMemo(() => {

        return number \* number

    }, [number])

    return(

        <div>

          <p>Número: {number}</p>

          <p>Quadrado: {square}</p>

        </div>

    )

}

Dessa maneira, o useMemo faz com que a função **square** (que calcula o valor do quadrado de um número) só seja executada quando o número (**number**, que está no array de dependências) for mudado.

**useLayoutEffect()**

O hook useLayoutEffect() é bem semelhante ao useEffect().

A grande diferença é que este usamos este hook quando queremos que **uma função** **seja executada antes do componente ser renderizado.**

Isso o torna adequado para efeitos que precisam ser sincronizados com a atualização do layout, como medição de elementos, cálculos de geometria ou qualquer manipulação do DOM que afete diretamente o layout.

A sintaxe do useLayoutEffect é exatamente a mesma do useEffect, veja só essa situação:

useEffect(() => {

  console.log('2')

  setName('Mudou de nome.')

}, [])

useLayoutEffect(() => {

    console.log('1')

    setName('Outro nome')

}, [])

Nesse caso, mesmo que o useLayoutEffect() se encontre depois do useEffect(), ele será executado primeiro, pois ele é executado **antes** do componente ser renderizado, enquanto o useEffect é executado **depois** do componente ser renderizado.

**useImperativeHandle()**

O hook useImperativeHandle() **permite que um componente filho exponha funções específicas para o componente pai através de uma referência**. Isso é útil para manter uma separação clara de responsabilidades entre os componentes e tornar a comunicação entre eles mais flexível.

Tudo começa no componente filho, nós usamos a função **forwardRef()** para criar essa interface personalizada entre pai e filho. Essa função recebe dois parâmetros: **props** e **ref.**

COMPONENTE FILHO

const ComponenteFilho = forwardRef((*props*, *ref*) => {

    const inputRef = useRef();

    // Função que será exposta para o componente pai

    const focusInput = () => {

      inputRef.current.focus();

    };

    // Usando useImperativeHandle para expor a função focusInput

    useImperativeHandle(*ref*, () => ({

      focusInput,

    }));

    return <input *ref*={inputRef} />;

  });

  export default ComponenteFilho;

COMPONENTE PAI

export function ComponentePai() {

    const componenteFilhoRef = useRef();

    const handleClick = () => {

      // Chama a função focusInput do componente filho através da referência

      componenteFilhoRef.current.focusInput();

    };

    return (

      <div>

        <button *onClick*={handleClick}>Focar no input</button>

        <*ComponenteFilho* *ref*={componenteFilhoRef} />

      </div>

    );

  }

**CUSTOM HOOKS**

Custom Hooks são os hooks que nós mesmos criamos. Muitas vezes fazemos isso para **abstrair funções complexas do componente** ou simplesmente **reaproveitar o código.**

Todos os custom hooks devem ser funções que comecem com a palavra **use**. Com eles, podemos ter a capacidade de reutilizar lógica em diferentes componentes.

Essa técnica é amplamente utilizada em projetos profissionais. Veja um exemplo de custom hook de contador a seguir:

export const useContador = () => {

    const [contador, setContador] = useState(0);

    const incrementar = () => {

      setContador(contador + 1);

    };

    const decrementar = () => {

      setContador(contador - 1);

    };

    return {

      contador,

      incrementar,

      decrementar,

    };

  }

Agora, podemos usar esse hook em algum componente, como é visto a seguir:

import { useContador } from './useContador';

function ComponenteA() {

  const { contador, incrementar, decrementar } = useContador();

  return (

    <div>

      <p>Contador em Componente A: {contador}</p>

      <button *onClick*={incrementar}>Incrementar</button>

      <button *onClick*={decrementar}>Decrementar</button>

    </div>

  );

}

**useDebugValue()**

Como o próprio nome sugere, o hook useDebugValue() nos permite **debugar o nosso código.**

Na documentação do React, é fortemente aconselhado que se faça uso desse hook dentro de um custom hook.

Quando fazemos uso desse hook, é adicionado uma área no React Dev Tools, para melhorar a experiência de debug do desenvolvedor.

**REACT DEVELOPER TOOLS**

O React Dev Tools é uma extensão do Google Chrome que nos ajuda a inspecionar e depurar o código de aplicações React diretamente do navegador.

Com essa valiosa ferramenta, podemos ter acesso á:

**1. Árvore de componentes:** nos permite ver como eles se relacionam uns com os outros.

**2. Inspeção de props e estados:** nos permite verificar como os dados estão sendo passados e atualizados entre os componentes.

**3. Detecção de atualizações:** nos permite identificar quais componentes foram renderizados novamente e o motivo disso.

**4. Manipulação de estado:** podemos alterar o estado de um componente em tempo real para testar diferentes cenários e verificar como a aplicação reage à mudanças de estado.

**5. Desempenho:** usando o Profiler, podemos analisar o desempenho da aplicação e identificar quais componentes estão causando lentidão.