**React**

1. **Etapas do curso**

- Introdução

- Fundamentos de React

- Avançando em React

- React e CSS

- Formulários com React

- Projeto: Secret Word

- Requisições HTTP com React (API)

- React Router (biblioteca)

- Context API

- Projeto: MiniBlog(bd)

- Entenda os hooks de React

- Projeto: ReactGram

**Conteúdos Adicionais**

- Javascript Moderno

-Introdução a APIs

- Setup de TypeScript

- React com TypeScript

- Projeto de React com TS

**- Atualizações**

* + 1. **O que é React?**

React é uma biblioteca JS para desenvolvimento de aplicações front-end;

Framework: segue maneira de criar aplicação para ter sucesso, já têm-se a estrutura pronta.

Biblioteca: nós mesmos criamos a base e o fluxo, não tem uma estrutura rígida a ser seguida.

Aplicações são chamadas **SPA** (Single Page Application), um carregamento, reload para diversas páginas;

Arquitetura do React é baseada em componentes;

Com o React pode-se criar a aplicação, ou inserir ele a uma;

É mantido pelo Facebook.

* + 1. **O que é Node.js?**

- O Node.js é uma runtime de JS, ou seja, o js é escrito, um compilador é adicionado e utiliza uma linguagem de mais baixo nível para fazer as execuções. Deixando assim os programas mais rápidos, performáticos;

- Uma biblioteca utilizada por um compilador durante a execução do programa;

- Biblioteca que está contruída na V8 engine, que faz com que JS vira C++, por ser uma linguagem de mais baixo nível;

- Possibilita a criação de softwares em JS no lado do servidor;

- Código JS rodando em C++ garantindo uma alta performance.

* + 1. **O que é npm?**

- Gerenciador de pacotes do Node, serve para que seja possível baixar bibliotecas de terceiros;

- A maioria dos projetos que serão criados em Node, estarão sendo iniciados, gerenciados pelo npm (Node Package Manager) Gerenciador de Pacotes do Node;

- Serve para executar determinados scripts no programa, com isso, podem ser criados setups mais complexos;

- Dificilmente um software em Node.js não utiliza o npm;

- Os módulos externos ficam numa pasta chamada **node\_modules**(padrão de todo projeto Node);

- Ela deve ser descartável e isso não poderá interferir no projeto, ou seja, a cada instalação do projeto são baixados todos os pacotes novamente.

* + 1. **Instalação Node Windows**

**-** O dowload do Node.js é feito no site oficial: nodejs.org;

**-** será baixado um arquivo .msi, o instalador;

**- npm** vem junto com o Node;

**-** Após a instalação, o Node e o npm serão testados em um terminal, para assim a instalação ser validada;

**-** Os binários precisam estar no **PATH**, passo a ser seguido na instalação do Node.

* + 1. **Hello World em React**

- Executor de scripts do Node: npx(x vem de executar);

- Dificilmente cria-se uma aplicação do zero, mas, para gerar uma, basta usar o comando **npx create-react-app<nome>**

- npm start – inicia a aplicação.

npm start

npm run build

npm test

npm run eject

‘cd hello world – abrindo uma pasta

npm start – iniciando a aplicação'

***outra forma de criação de uma aplicação(mais ágil)***

npm create vite@latest

cd <nome>

npm i

npm run dev

* + 1. **Estrutura base do react**

Há algumas pastas e arquivos chaves para o desenvolvimento do react;

**- node\_modules:** onde as dependências do projeto ficam-pode ser removida a qualquer momento, logo, dentro dela não devem ser colocados códigos;

**- public:** assets estáticos e html de inicialização;

**- src:** onde serão programados os apps, principal pasta – components, estruturas responsáveis por fazer requisições assíncrona, configurações de adaptação;

**- src / index.js:** arquivo de inicialização do react;

**- src / App.js:** componente principal da aplicação.

* + 1. **Extensão para React**

Extensão no vs code que ajuda a programar em react: **marketplace**;

Nome: **ES7 + React/Redux/React-Native snippets**;

Pode ser criado muito código com poucos atalhos.

**Preparando o Emmet para o React**

Extensão nativa do VS Code: ajuda a escrever html mais rápido;

Não vem configurada para o React;

File>Settings>Extensions e procurar Emmet;

Incluir: Javascript-javascriptreact;

**Livros que podem auxiliar:**

JS

<https://www.amazon.com.br/gp/product/B016N7G8EK?&linkCode=sl1&tag=horadecodar04-20&linkId=62464e9456c337e90e1471bd13b54ebd&language=pt_BR&ref_=as_li_ss_tl>

React

<https://www.amazon.com.br/React-Praticando-Desenvolva-Aplica%C3%A7%C3%B5es-Biblioteca/dp/6586057396?__mk_pt_BR=%C3%85M%C3%85%C5%BD%C3%95%C3%91&crid=21F52A2MJJQPA&keywords=react&qid=1681470148&sprefix=react,aps,329&sr=8-1&linkCode=sl1&tag=horadecodar04-20&linkId=442f338d162184060295952eee0c6af6&language=pt_BR&ref_=as_li_ss_tl>

1. **Fundamentos do React**
   * 1. **Criando componentes**

Na maioria dos projetos os componentes ficam em uma pasta chamada **componentes**, que devemos criar;

Geralmente nomeados com a **camel case: FirstComponent.js**;

No arquivo **criamos** **uma** **função**, que contém o código desse componente (lógica e template);

Exportar função para reutilizá-lo.

* + 1. **Importando componente**

Importação – maneira de reutilizar componentes;

Sintaxe: **import nomecomponente from ‘./components/nomecomponente’**;

Para colocar o componente importado de outro, coloca-se em forma de tag: **<FirstComponent />**

Finaliza importação, importando FirstComponent em App.

import FirstComponent from './components/FirstComponent';

//importação de FirstComponent para o arquivo JS FirstComponent

Exibindo o elemento no browser

<FirstComponent />

* + 1. **JSX**

HTML do React, porém, nele pode-se colocar instruções dinâmicas, logo, não é só um simples HTML;

Onde serão declaradas as tags HTML que serão exibidas no navegador;

Ficam no **return** do componente;

Existem algumas diferenças do HTML, um exemplo: **class** vira *className*

Isso acontece pelas intruções semelhantes de JS e HTML, pode haver conflitos, então vale ressaltar que JSX é JS, então alguns nomes mudam;

O JSX pode ter apenas um elemento pai;

Os elementos precisam ser encapsulados em uma única tag;

Alguns erros só são visíveis no console.

* + 1. **Comentários no componente**

Essa inserção de comentários pode ser feita de duas maneiras;

Na parte da função onde a lógica é executada, a sintaxe é: **//Algum comente**;

Já no JSX: **{/\*Algum coment\*/}**;

* + 1. **Template Expressions**

É o recurso que nos permite **executar JS e JSX** e também **interpolar varáveis**;

Útil ao longo dos projetos em React;

Síntaxe: **{algumCodigoEmJS}**;

Tudo entre chaves é processado em JS e nos retorna o resultado.

const name = 'Aninha'

    const data = {

        age: 31,

        job: "Programmer",

return (

        <div>

            <h1>Olá {name}, tudo bem?</h1>

            <p>Você atua como: {data.job}</p>

            <p>{4 + 4}</p>

            <p>{console.log("JSX React")}</p>

        </div>

    )

**EXIBIÇÃO:**

**Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente**

* + 1. **Hierarquia de componentes**

Os componentes criados podem ser reutilizados em vários componentes;

Também podem formar uma hierarquia, sendo importados uns dentro dos outros, como foi feito em App.

Hierarquia ex.:

App>FirstComponent.js>MyComponent.js

//arquivo de estilo

import MyComponent from "./MyComponent"

const FirstComponent = () => {

    /\*

    bla

    bla

    mult

    line

    \*/

    return (

        <div>

            {/\*algum coment\*/}

            <h1>Meu primeiro componente</h1>

            <p className="teste">Meu texto</p>

            <MyComponent />

        </div>

        //os elementos precisam ser encapsulados em uma única tag

    )

}

export default FirstComponent

MyComponent.js vira ‘irmão’ dos outros componentes:

function App() {

  return (

    <div className="App">

      <h1>Fundamentos React</h1>

      <FirstComponent />

      <TemplateExpressions />

      <MyComponent />

* + 1. **Eventos**

Os eventos para o front-end são essenciais;

O click será utilizado em vários momentos, como ao **enviar formulários**;

No React os eventos já estão ‘prontos’, o OnClick pode ser utilizado para ativar uma função ao clicar no elemento;

Esta função é criada na própria função do componente;

As funções geralmente tem o padrão **handleAlgumaCoisa;**

AlgumaCoisa - Click, Submit, etc.

* + 1. **Funções no evento**

Quando as funções são simples, pequenas, a lógica pode ser criada no próprio evento;

O **código** se torna **mais** **‘complicado’** pela lógica estar sendo atrelada com HTML;

Mas em algumas situações é aplicado, um exemplo seria um formulário com vários inputs.

* + 1. **Funções de renderização**

Pode-se criar funções que retornam JSX;

Isso serve para criar situações que **dependem de outras condições**, condicionais lógicas;

Sendo assim, o JSX a ser renderizado pode variar por alguma variável, por exemplo;

1. const renderSomething = () => {
2. if ( 'x' ) {
3. return <h1>Renderizando isso!</h1>
4. } else {
5. return <h1>Também posso renderizar isso!</h1>
6. }
7. }
8. {renderSomething(true)}
9. {renderSomething(false)}
10. **Avançando no React**
    * 1. **Imagens no React**

As imagens públicas podem ficar na **pasta public**;

Com isso, elas podem ser chamadas pela tag img diretamente pelo /nomeimg.jpg;

Isso, pois a pasta public é linkada com src das imagens.

* + 1. **Imagens em asset**

A pasta não precisa necessariamente chamar assets, pode ser img por exemplo;

A pasta public pode ser utilizada para colocar as imagens, porém, isso de colocar na pasta assets, em src é um padrão bem utilizado;

Sendo assim, você pode encontrar projetos com as duas abordagens;

Em assets precisa-se fazer a importação da imagem e o src é dinâmico com o nome da importação, como a importação de um componente;

Decide-se oque usa verificando a necessidade do projeto, por exemplo, no caso de mudança de path de um projeto grande, só precisará mudar em um lugar.

import City from './assets/city.jpg'

        <img src={City} alt="Cidade" />

* + 1. **O que são hooks?**

Recursos do React que tem diversas funções;

Um exemplo seria: guardar e alterar o estado de algum dado da nossa aplicação;

Todos os hooks começam com **use**, por exemplo **useState**;

O Hook do exemplo acima, guarda e altera estado. Geralmente a segunda palavra define a funcionalidade, sendo as vezes não tão claro.

Podemos criar os próprios hooks, esta ação chama-se **custom** **hook**;

Os hooks **precisam ser importados**;

Geralmente úteis em todas as aplicações, ao longo do curso serão utilizados diversos do mesmo;

Um exemplo de hook muito utilizado seria o fetch.

**useState hook**

Um dos mais utilizados;

Função:gerenciar estado de algum dado, variáveis não funcionam corretamente, o componente não re-renderiza;

Para guardar um dado, define-se o nome da variável e para alterar utiliza-se **setNome**, onde nome é o nome do dado.

* + 1. **Renderização de listas**

Ação comum: renderizar listas de dados no template;

Faz-se isso com dados do tipo array;

Utiliza-se **método map** para nos auxiliar(Javascript puro, não tem a ver com react);

Além dos dados, pode-se inserir JSX em cada interação.

.map – método nativo de JS, utilizado paraa a manipulação de dados dentro de arrays.

* + 1. **A propriedade key**

Obrigatória na renderização de listas;

Alterar listas sem essa propriedade gera **warning(não é um erro, pois não para a execução do código, mas precisa ser resolvido)** que pode ser verificado no console;

O React precisa de uma chave única em cada um dos itens interados, normalmente é um ID, mas nem sempre existe esse dado;

Serve como ajuda na renderização de componente, por exemplo, quando algum item daquela lista mudar, ele consegue se achar facilmente, fazer a alteração onde é necessário, sem alterar outros nós no HTML , fazendo assim uma re-renderização inteligente do HTML;

Geralmente há um array de objetos, e pode-se colocar key como alguma **chave única**, como o **ID** de algum dado;

Em último caso, devemos utilizar o index do método map, não sendo muito prático. Pois, pode haver exclusão de elementos, alterando assim o índice por ser baseado em array e não em itens únicos. Pode haver diversos problemas, então o ideal é não usá-lo.

<div>

      <ul>

        {list.map((item, i) => (

          <li key={i}>{item}</li>

        ))}

      </ul>

      <ul>

        {users.map((user) => (

          <li key={user.id}>{user.name} - {user.age}</li>

        ))}

      </ul>

    </div>

  )

const ListRender = () => {

  const [list] = useState(["Matheus", "Pedro", "Josias","Maria"])

  const [users] = useState([

    { id: 1, name: "Matheus", age: 31 },

    { id: 72378423, name: "João", age: 28 },

    { id: 8328823, name: "Pedro", age: 44 },

  ])

* + 1. **Previous State**

Um recurso que nos permite pegar o valor original de um dado dentro de um set de dado;

Muito utilizado para modificar listas, pois têm-se o valor antigo e transforma-se em um novo valor;

O primeiro argumento de um set sempre será o previous state.

* + 1. **Renderização condicional**

Quando imprimimos uma parte do template baseado em uma condição(if/else);

Ou seja, utilizando uma checagem com if;

Interessante em algumas situações, um exemplo: usuário autenticado/não autenticado.

**Adicionando um else**

É possível realizar um **if/else** no JSX;

Aqui, usa-se o **if ternário**;

Síntaxe: condição? bloco1 : bloco2.

* + 1. **Props**

Recurso fundamental do React;

Nos permite passar dados de um componente pai para um componente filho, isso acaba gerando a dinamização dos componentes, pois será feito um reaproveitamento baseado em algum dado dinâmico que vem do componente pai, por exemplo: a função de carregamento de um dado;

Será útil quando os dados forem carregados via banco de dados, por exemplo;

As props vem em um objeto no argumento da função do componente: cria-se um componente pra função, até então vazio, ali as props passadas irão.

**App.js**

<ShowUserName name="Matheus" />

{/\*no exemplo acima o component ShowUserName, tem acesso a uma prop chamada name com o

valor de "Matheus\*/}

**ShowUserName**

const ShowUserName = (props) => {

return (

<div>

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

{/\*name foi definido no component pai App.js\*/}

</div>

)

}

**- Quando se declara algo executável no react é sempre entre chaves;**

**- Pode-se criar códigos executáveis no arquivo JS pai e usar nos/no filho.**

Podem ser passados: **valores diretos**, **valores com variável**e **valores de State;**

* + 1. **Destruturando props**

Muito comum a passagem de mais de uma prop em um componente, isso acaba gerando um problema na hora de nomear as mesmas e um excesso de digitação desnecessário;(javascript)

Para falicitar isso o React nos permite fazer a **destruturação das propriedades** que chegam, com o método **destructuring**;

Se tem duas props: **name** e **age**;

Podemos fazer desta maneira: MyComponent({name,age})

Assim **não precisando** mais utilizar **props.AlgumaCoisa**.

Valores numéricos são declarados entre chaves.

* + 1. **Reutilização de componentes**

Com as **props** a **reutilização de componentes** faz mais sentido;

Se temos dados de 1000 carros, podemos reaproveitar o nosso CarDetails 1000 vezes, caso seja necessário fazer uma alteração, ela só precisará ser feita em um lugar;

Muda-se apenas os valores dos componentes, um exemplo, uma checagem não teria que ser feita em todas as divs que tivessem imprimindo carros, um exemplo.

**Componente**

const CarDetails = ({brand, km, color, newCar}) => {

return (

<div>

<h2>Detalhes do carro</h2>

<ul>

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

<li>KM: {km}</li>

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

</ul>

{newCar && <p>Este carro é novo!</p>}

</div>

)

}

**Reaproveitamento**

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

{/\* reaproveitando \*/}

<CarDetails brand="Ford" color="Vermelha" hm={0} newCar={true} />

<CarDetails brand="Fiat" color="Branco" hm={4500} newCar={false} />

* + 1. **Reutilização com loop**

Os arrays de dados podem ter muitos itens também, sendo assim, é correto utilizar uma **estrutura de loop(map)** para a sua exibição;

E com isso, concilia-se os três conceitos: renderização de listas, reaproveitamento de componentes e props.

**Function App**

const cars = [

{id:1, brand:"Ferrari", color:"Amarela", newCar:true, km:0},

{id:2, brand:"KIA", color:"Branco", newCar:false, km:3443},

{id:3, brand:"Renault", color:"Azul", newCar:false, km:234},

]

**return**

{/\* lop em array de objetos \*/}

{cars.map((car) => (

<CarDetails

brand={car.brand}

color={car.color}

km={car.km}

newCar={car.newCar}/>

))}

* + 1. **React Fragments**

São interessantes para quando é necessário ter mais de um elemento pai em um componente;

Criamos uma tag vazia: **<>** ... **</>**, isso não é possível em HTML e dentro dessas tags pode haver diversos elementos, ela **serve como um elemento pai**, não alterando a estrutura HTML;

O Fragment ajuda quando um elemento HTML pai é desnecessário, usando-o aparece apenas oque é de nosso interesse.

**A resolução de warning’s é necessária, por mais que não altere nada no código e nem na execução, para uma melhor performance.**

* + 1. **Children Pop**

Recurso utilizado quando o componente precisa ter JSX dentro dele(como se fosse uma tag de div que abraça outros componentes, HTML ou JSX), se a tag children não for utilizada, o conteúdo html será simplesmente ignorado;

Esse JSX vem do componente pai, há uma diferença na hierarquia pois está sendo importado um componente que está utilizando JSX com outro componente que está no componente pai;

Logo, componente que tem a propriedade **children**, age como **container**, abraçando os elementos;

Children é considerada uma **prop do componente**.

**Container.js**

const Container = ({children, myValue}) => {

return (

<div>

<h2>Este é o título do container</h2>

{children}

<p>O valor é: {myValue}</p>

</div>

)

}

**App.js**

{/\* children \*/}

<Container myValue="testing">

<p><strong>E este é o conteúdo</strong></p>

</Container><Container myValue="testing 2">

<h5>Testando o container</h5>

</Container>

* + 1. **Funções em props**

As funções podem ser passadas para as props normalmente, logo, não há diferenciações de valores nas funções. Basta criar a função no componente pai e **enviar como prop** para o componente;

No componente filho ela pode ser ativada por um evento, por exemplo.

**ExecuteFunction.js**

const ExecuteFunction = ({myFunction}) => {

return (

<div>

<button onClick={myFunction}>Clique aqui para executar a função!</button>

</div>

)

}

**App.js - function**

function showMessage() {

console.log('Evento do componente pai')

}

**App.js - return**

{/\* executar função \*/}

<ExecuteFunction myFunction={showMessage}/>

1. **React e CSS**
   * 1. **CSS global**

É utilizado para estilizar diversos elementos em comum ou fazer um reset no CSS;

Utiliza-se o arquivo **index.css** na pasta **src**.

* + 1. **CSS de Componente**

Utilizado para um componente específico;

Geralmente é criado um arquivo com o mesmo nome do componente e este é importado no componente, um exemplo seria o App.js com o App.css;

Este método não é scoped (com escopo), logo, o CSS vaza para outros componentes se houver uma regra de colisão;

O React cria um exemplo dessa técnica com App.css.

**MyComponent.js**

const MyComponent = () => {

  return (

    <div>

        <h1>CSS de Component</h1>

        <p>Este é o parágrafo do componente</p>

        <p className="my-comp-paragraph">Este tb é do componente</p>

    </div>

  )

}

**MyComponent.css**

p{

    color: green;

    background-color: #000;

}

.my-comp-paragraph{

    color: #000;

    background-color: gold;

}

* + 1. **Inline style**

Igual ao do CSS;

Há uma tag no **atributo style** que permite a aplicação de regras diretamente a um elemento;

Deve-se optar por outras maneiras de css, pois o inline pode dificultar a manutenção ou deixar o código imprevisível em certas situações;

Acaba “poluindo a codificação”.

* + 1. **Inline Style Dinâmico**

O CSS dinâmico inline aplica o estilo baseado em uma condicional;

**Insere-se** no atributo um **if** **ternário**;

Dependendo da condição pode-se mudar que regras de estilo um elemento recebe.

* + 1. **Classes Dinâmicas**

Pode-se aplicar lógica para mudar a classe de um elemento;

Utiliza-se o if ternário;

É mais interessante que o CSS inline, pois as classes estão isoladas no css, resolvendo problemas de organização do código.

**App.js**

  const redTitle = true

{/\* Classe dinâmica \*/}

        <h2 className={redTitle ? "red-title" : "title"}>Este título terá classe dinâmica</h2>

**Index.css**

.red-title{

  background-color: #000;

  color: red;

  padding: 15px;

  border: 5px solid red;

}

.title{

  background-color: #000;

  color: #fff;

}

* + 1. **CSS Modules**

Recurso de CSS scoped, ou seja, será exclusivo do componente;

Nome do arquivo: **Componente.module.css**;

Precisa ser importado no componente.

1. **Formulário**

**5.1.1 Criando Formulários**

Utiliza-se a **tag Form**;

**htmlFor** de label recebe o mesmo valor que o **nome** do input;

Processamento feito de forma assíncriona – action não será utilizado.

* + 1. **Label envolvendo o input**

Um padrão comum dentro do React é a tag label envolvendo o input;

Assim, atributo for se torna opcional, tendo assim, menos linhas de código;

Simplifica estrutura HTML, sem perder a semântica.

* + 1. **Manipulação de valores**

Utiliza-se o **hook** **useState**;

Com, será possível armazenar na variável e utilizar o **set** para alterar o valor;

Criação de função para alterar valor no evento onChange (acontece a cada vez que uma tecla é pressionada no input);

Código ficará mais fácil de trabalhar nas próximas etapas, como: envio de dados para BD e validação.

* + 1. **Simplificando a manipulação dos dados**

Quando há muitos inputs pode ser realizada a manipulação de forma mais simples;

Basicamente cria-se uma **função inline** no **onChange**;

Vai alterar o valor do state, com um método set, da mesma forma e a função isolada.

* + 1. **Envio de formulário**

Enviar um formulário – evento onSubmit;

O mesmo chamará a função e nesta deve-se parar a submissão com o **preventDefault**;

Nesta etapa podemos realizar validações, envio de formulários para o servidor, reset de form entre outras ações que podem ser feitas.

 {/\*onSubmit - pega evento de submissão do form\*/}

{/\*handleSubmit - função para processar envio \*/}

//preventDefault - para o envio, form não recarrega a pág.

* + 1. **Controlled Inputs**

Recurso que nos permite mais flexibilidade nos forms de React;

Precisa igualar o valor ao state(value=”state”);

Uso muito comum: formulários de edição, que dados vem do back-end, preenchimento do input mais facilmente.

**Value < input < MyForm**

value={name}

value={email}

**Component MyForm**

const MyForm = ({ user }) => {

  //6 - controlled inputs

  // 3 - gerenciamento de dados

  const [name, setName] = useState(user ? user.name : '')

  const [email, setEmail] = useState(user ? user.email : '')

* + 1. **Limpando Formulários**

Com o controller fica mais fácil;

Basta atribuir um valor de uma string vazia aos states;

Isso será feito após o envio, em forms que o usuário precisa preencher novamente.

const handleSubmit = (e) => {

    e.preventDefault()

    //preventDefault - para o envio, form não recarrega a pág.

    console.log("Enviando form")

    console.log(name, email)

    //7 - limpar formulário

    setName("")

    setEmail("")

  }

* + 1. **Input de Textarea**

Pode ser considerado um **input de texto** normal;

Alteração do state inicial será feita no value;

Event onChange para modificar valor do state.

  const[bio, setBio] = useState()

    console.log(name, email, bio)

    setBio("")

        <textarea name="bio" placeholder="Descrição do usuário" onChange={(e) => setBio(e.target.value)} value={bio}>

* + 1. **Input de Select**

Também semelhante aos outros inputs;

Quando há alteração de valor o evento onChange capta isso;

O **value** também pode atribuir qual **option** estará selecionada.

1. **Requisições HTTP e React**

**Integração com Back-End**

* + 1. **JSON Server**

JSON Server – pacote npm;

Simula uma API, possibilitando fazer requisições HTTP;

Será ensinado como integrar este recurso com React;

Pode-se entender isso como uma etapa de preparação para APIs reais, atingindo o resultado sem precisar de uma estrutura back-end;

* + 1. **A importância do use Effect**

Faz com que determinada ação seja executada apenas uma vez interessante pois os componentes estão sempre se re-renderizando, logo, precisamos ter ações únicas às vezes;

Possui um array de dependências, que contém os dados que ativam a execução da função de forma automática, para assim termos o controle de por exemplo a entrada de dados, entre tantas outras coisas;

Está presente na maioria das requisições assíncronas.

* + 1. **Resgatando dados com React**

Para trazer os dados serão utilizados diversos recursos:

Ter um local para salvá-lo e futuramente se precisar, alterar (useState);

Renderizar a chamada a API apenas uma vez (useEffect), fazer o request;

Um meio de fazer a requisição assíncrona (Fetch API), deixa a ação de fazer funções assíncronas mais fáceis, por ser tudo nativo e não haver a necessidade de instalar pacotes terceiros.

* + 1. **Adicionando dados**

Resgatas dados do form com useState;

Reunir eles em função após o submit e enviar um request de POST para a nossa API;

Processo parecido com resgate de dados, mas agora, enviando dados.

* + 1. **Carregamento dinâmico de dados**

Se a requisição foi feita com sucesso, pode-se adicionar o item a lista após o request, deixando a aplicação mais performática;

Utiliza-se o **set** do **useState** para isso;

O intuito é melhorar a performance para não ser preciso a utilização do f5 por exemplo, para atualizar, melhorar a usabilidade do site.

    //pega dados próprios que vem da requisição e envia para o front-end, com isso o envio de dados já é automático

* + 1. **Custom Hook para o fetch**

Podemos criar nossos próprios hooks;

Alguns são criados na maioria dos projetos;

Facilitam em algumas ações feitas por nós;

É comum dividir funções que podem ser reaproveitadas em hooks, vai ser criado um para o resgate de dados;

Geralmente ficam na pasta hooks, então, como *‘components’* existe uma pasta para os hooks;

Utiliza-se o padrão do react = useAlgumaCoisa(useName, useState, useEffect);

Basicamente uma função será criada e exportada.

src>**hooks**>arquivo**.js**

* + 1. **Refatorando o POST**

Pode-se utilizar o mesmo Hook para incluir uma etapa de POST;

Cria-se um novo useEffect, que mapeia uma outra mudança de estado, isso que determina se é um POST;

Após ela ocorrer executamos a adição do produto no projeto e a atualização da lista;

Nem sempre reutilizar Hook é a melhor estratégia.

1. **React Router**

Adicionar paginação.

* + 1. **O** **que é?**

Um dos pacotes mais utilizados para criar uma estrutura de rotas em aplicação de React, não está presente no react, é uma aplicação a ser instalada;

Permite que as SPAs tenham múltiplas páginas, não precisa por exemplo, manipular states;

Precisa ser instalado;

A configuração e utilização é simples, mas, trabalhosa;

Há também, outras funções como: **Redirect**, **Nested** **Routes**, **Not** **Found** **Routes** e outros.

**Criando projeto e instalando React Router**

- npx create-react-app react-router

- cd react-router

- npm start

- npm i react-router-dom

- npm i json-server

- npm run server

* + 1. **Configurando o React Router**

Vão precisar serem exportados 3 elementos de **react-router-dom**, como são exportados useState, useEffect, por exemplo;

**BrowserRouter:** Define onde a área do app que vai trocar as páginas;

**Routes:** Define as rotas;

**Route:** Um elemento deste para cada rota, configurar com path e componente da rota.

//elementos fora do routes, estarão sendo exibidos em todas as telas

<BrowserRouter>

        <Routes>

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

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

        </Routes>

      </BrowserRouter>

Geralmente, para as páginas, cria-se uma nova pasta – ‘pages’.

* + 1. **Adicionando links**

Para isso, vamos utilizar um componente do React Router, o **Link**;

No link, o parâmetro to é configurado e recebe a URL/path que será redirecionado quem clicar no link;

Criar componente Navbar.

        <Navbar />{/\*se repete entre as rotas, mas tem elementos no React Router\*/}

**Logo, fica abaixo do <BrowerRouter>**

* + 1. **Carregando dados**

1. Será exercitado novamente o carregamento de dados com o hook useFetch;
2. Depois serão utilizados para o carregamento de dados individuais(privados), como páginas dinâmicas onde dados são carregados pelos Ids;
3. O hook que será utilizado é igual ao da outra seção, os produtos da Home serão imprimidos da mesma forma.

  const { data: items, loading, error } = useFetch(url)

  //chamando propriedados vindas do hook //data - dados

{error && <p>{error}</p>}{/\* se houver erro, será imprimido o mesmo \*/}

 {items && items.map((item => (//checando se itens chegaram, se sim, será feito um .map em cada item

* + 1. **Rota dinâmica**

Um processo feito constantemente;

Para criar uma rota dinâmica, define-se uma **nova Route** em **App.js**;

Que deve conter o padrão de: **/products/:id**, onde **:id** é o dado dinâmico, ou seja, pode-se ter qualquer valor, componente pode ser usado para adicionar diversos valores;

Na página, pode-se utilizar o hook **useParams** para resgatar esta informação.

* + 1. **Carregamento dado individual**

Com o passo dado na aula anterior, o carregamento fica mais fácil;

O id recebido será utilizado para formar uma nova URL;

O useFetch será utilizado para trazer um item;

Para finalizar será feita a validação e impressão do mesmo no JSX.

    <p>ID do produto: {id}</p>{/\* descricao aparecerá na tela de descricao no do produto, parte dinamica feita na Home.js e vinculada com o id e o useParams \*/}

  const { data: product, loading, error } = useFetch(url)//dados vao vir do useFetch dentro de hooks trazendo uma url

**concatenando url e id para exibição de um dado de cada vez**

  const url = "http://localhost:3000/products/" + id

**validação**

 {error && <p>Ocorreu um erro</p>}{/\* <- validação \*/}

    {loading && <p>Carregando...</p>}{/\* <- validação \*/}

* + 1. **Nested Route**

Indicam **URLs** mais **complexas**, como: **/products/id:/something**, /something(algo) – array para mostrar reviews(avaliações), comentários, poderiam ser subprodutos caso produtos fosse uma categoria, etc;

Neste caso, cria-se um componente correspondente ao padrão indicado e a URL em App.js;

Na nested route teremos o acesso ao parâmetro da URL também então, se o id foi colocado dinamicamente, quando componente for acessado na nested route esse id poderá ser acessado.

* + 1. **No match route (404)**

Uma **página 404** pode ser criada facilmente com o React Router;

Basta **criar** o **componente** da página, **imprimindo algo** obviamente;

No arquivo App.js definir um **path** como **\***;

Deste modo, qualquer **rota não existente** cairá **neste componente**.

**App.js**

{/\* 7 - no match route \*/}

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

* + 1. **Link ativo**

Para fácil acesso a uma modificação de links ativos, troca-se o Link pelo **NavLink**;

Neste elemento têm-se acesso ao valor **isActive**, uma **classe** pode ser **ativada** **se** a **rota atual** for a que **está** **no** **atributo to**.

**Navbar.js – importando element NavLink**

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

**Navbar.js**

<NavLink to="/"

        /\*className={({isActive}) => (isActive ? "esta-ativo" : "nao-ativo")}\*///-posso usar caso eu queira usar meu padrao de link ativo, mas nao é necessário

        >Home</NavLink>

        <NavLink to="/about">Sobre</NavLink>

**Navbar.css**

nav a:hover{

    background-color: #000;

}

.active {

    background-color: #000;

    color:#fff

}

* + 1. **Search Params**

Busca de dados através do react;

Recurso que permite obter o que vem na URL em forma de parâmetro, ex.: produtos?q=camisa;

Utilizamos o hook useSearchParams para obtê-los;

Com isso, fica simples fazer uma funcionalidade de busca no sistema.

**SearchForm.js - component**

import { useNavigate } from 'react-router-dom'//para redirecionar usuário dentro do código do componente

  const [query, setQuery] = useState()//busca//manipular estado e utilizar o valor dele para pegar a busca

const handleSubmit =(e) => {

      e.preventDefault()//para página nao ser recarregada qnd botao de submit for clicado

      navigate('/search?q=' + query)//busca

  }

return (

    <form onSubmit={handleSubmit}>

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

      <input type="submit" value="Buscar" />

    </form>

  )

}

**Search.js – page**

import { useSearchParams, Link} from 'react-router-dom'

import { useFetch } from '../hooks/useFetch'//hook para importação de dados

const Search = () => {

  let [searchParams] = useSearchParams()

  const url = "http://localhost:3000/products?" + searchParams

  const { data: items, loading, error} = useFetch(url)

return (

    <div>

      <h1>Resultados disponíveis</h1>

      {error && <p>Erro!!</p>}

        <ul className='products'>

          {items && items.map((item) => (

            <li key={item.id}>

              <h2>{item.name}</h2>

              <p>{item.price}</p>

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

            </li>

          ))}

        </ul>

        {loading && <p>Carregando...</p>}

    </div>

  )

}

* + 1. **Redirecionamento da URL**

Podemos precisar de um **redirecionamento** **de páginas** eventualmente, ex.: uma **página antiga** do sistema **responde** agora uma **nova URL**;

Para isso vamos **criar** uma rota com **Route** normalmente;

Em element vamos utilizar o **componente** **Navigate** com um **to** que vai para a rota correta.

import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom'

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

1. **Context API do React**

Funcionalidade usada para que o valor de algum dado possa ser transportado/transmitido através do componente;

Assunto simples, mas pode se tornar complexo, pois, podem estar sendo manuseados muitos dados.

* + 1. **O que é Context API**

Recurso que facilita o compartilhamento de um estado entre componentes;

Quando precisamos de **dados ‘globais’**, provavelmente precisamos utilizar o Context;

Um exemplo: um usuário faz um cadastro, eu consigo guardar aqueles dados de cadastro e usar em quantos componentes forem necessários;

O **Context encapsula componentes** que receberão seus valores, geralmente fica no App.js ou index.js;

Context vira praticamente global, pode-se acessar ele de qualquer lugar;

Os contexts geralmente ficam na pasta context;

Basicamente, é uma forma de consumir e alterar dados que vão se tornar globais na aplicação e ter um setup para fazer isso.

* + 1. **Criando Contexto**

Criar Context;

O arquivo deve ter a primeira letra maiúscula e geralmente termina em Context, por exemplo: SomeContext.js;

A convenção é deixar na pasta **context** **em src**;

Onde vamos utilizar o valor do contexto, o arquivo precisa ser importado.

// 1 - criar contexto

import { createContext, useState } from 'react'

export default CounterContext = createContext()

* + 1. **Criando o provider**

Vai prover o contexto;

**Delimita** onde o contexto é utilizado;

Criar espécie de componente com a **prop children**;

Este provider deve encapsular os demais componentes em que é preciso consultar ou alterar valor;

Geralmente fica em App.js ou index.js;

Agora é possível compartilhar o valor do contexto em todos os componentes.

***CounterContext.js***

// 2 - criar provider

export const CounterContextProvider = ({children}) =>  {

//o nome do provider é algo relativo ao context mas com provider no fim

//contexto a ser compartilhado entre elementos

    const [counter, setCounter] = useState(5)

    return(

        <CounterContext.Provider value={{counter, setCounter}}>

        {/\* NomeContext.Provider value={{valorConsumo valorAlteracao}} \*/}

            {children}

        {/\*children - encapsular elementos e poder imprimir conteudo deles dentro de outro component\*/}

        </CounterContext.Provider>

    )

}

***Home.js***

import { useContext  } from "react"//importando forma de se usar o contexto

import { CounterContext } from "../context/CounterContext"//importando variavel com elementos a serem utilizados

const Home = () => {

  const { counter } = useContext(CounterContext)//desustruturando counter do useContext-CounterContext

  return (

    <div>

      <h1>Home</h1>

      <p>Valor do contador: {counter}</p>

    </div>

  )

* + 1. **Alterando contexto**

Vamos criar um componente que utilize a função de mudança do contexto(isso nem sempre é necessário, pode-se fazer isso na própria página);

Esta mudança ocorrerá no Context e poderá ser consumida por todos os componentes que recebem o contexto;

Assim é finalizado o ContextAPI.

**Alteração – ChangeCounter.js**

<div>

        <button onClick={() =>{setCounter(counter+1)}}>Add value to counter</button>{/\* valor de contador é incrementado a cada vez que o botao é clicado \*/}

        {/\* arrow function pois se uma funcao é colocada diretamente ela é executada ao renderizar o component \*/}

    </div>

**Home.js, About.js, Products.js**

**Dados colocados para haver a alteração em todas as páginas que contém os mesmos:**

 const { counter } = useContext(CounterContext)

  return (

    <div>

      <h1>Products</h1>

      <p>Valor do contador: {counter}</p>

    </div>

* + 1. **Refatorando contexto com hook**

Pode-se criar um hook para utilizar o contexto, isso nos dá vantagens, no caso isso seria uma boa prática, estamos avançando o nível do código;

Não precisamos importar o useContext em todos os lugares que vamos usar o contexto, só o hook, utilização fica simples;

Temos espaço para fazer uma validação de contexto.

***useCounterContext.js - hook***

import { useContext } from 'react'

import { CounterContext } from '../context/CounterContext'

export const useCounterContext = () => {//criação do hook

    const context = useContext(CounterContext)//chamando hook na variável context

***Home.js – consumindo elemento do hook***

refatorando com hook

import { useCounterContext } from "../hooks/useCounterContext"//importando hook que contém o context

const { counter } = useCounterContext()//invocando a função do hook

* + 1. **Contexto mais complexo**

Podem ter variações no comportamento;

Para isso será utilizado um hook chamado **useReducer**, que é como um useState, mas para controle de dados complexos, com mais de um dado;

No reducer tem diferentes **ações com** um **switch**;

Na aplicação será consumido o estado atual do dado que está no reducer.

***TitleColorContext.js***

    const [state, dispatch] = useReducer(titleColorReducer, {color:"purple"})//pode-se colocar um valor que já vai ser alterado ao iniciar(nesse caso {color: "purple"})

    //esta sendo passado quem altera o estado e quem é o estado inicial(basicamente state entrega uma maneira de alterar o estado e dispatch altera)

    ///state - estado do hook atual - o que ele esta gerenciando

    ////dispatch - função/o que vai alterar o contexto depois

<TitleColorContext.Provider value={{ ...state }}>{/\* esse é o state a ser consumido na aplicação \*/}

            {children}

        </TitleColorContext.Provider>

***Index.html***

    <CounterContextProvider>

      <TitleColorContextProvider>

        <App />

      </TitleColorContextProvider>

    </CounterContextProvider>

* + 1. **Alterando contexto complexo**

Vamos utilizar uma função chamada **dispatch**, que está no **reducer** também;

E deve conter todas as informações necessárias para a alteração do valor do contexto;

Ou seja, o **switch age** e retorna um novo contexto.

***TitleColorContext.js***

 switch (action.type) {//switch é feito em cima dessa action.type

        case "RED"://os tipos de ação são todos em letra maiúscula e string

            return { ...state, color: "red" }

        case "BLUE":

            return { ...state, color: "blue" }//retorna todos os dados do state( ...state) e muda propriedade(color:"blue")

        default://quando não aocntece nada, retornará o state()cor roxa, definita no hook

            return state

    }

***Home.js***

// 6 - alterando state complexo

const setTitleColor = (color) => {

  dispatch({ type: color })//tem um dispatch e a cor seria a acao

}

{/\* 6 - alterando contexto complexo \*/}

      <div>

      <button onClick={() => setTitleColor("RED")}>Vermelho</button>

      <button onClick={() => setTitleColor("BLUE")}>Azul</button>

      </div>

***About.js***

//5 - context mais completo

  const {color} = useTitleColorContext()

<h1 style={{color:color}}>About</h1>{/\* consumindo context que foi consumido inicialmente na home \*/}

* + 1. **Projeto: MiniBlog**

**Instalação FireBase**

**Cria-se:**

**Pasta firebase>config.js**

**config.js**

**- colar configs do projeto firebase**

**- criar banco de dados**

**- exporta banco de dados**

import { getFirestore } from "firebase/firebase"

const db = getFirestore(app)//Firesore - serviço de banco de dados dentro do firebase

export{db}

**tela de registro *(pages>Registro.js)***

  const handleSubmit = (e) => {//o submit serve para reunir todos os dados e ajudar na hora de enviar no formulário

value={displayName}//para que fique com controle de inputs

            onChange={(e) => setDisplayName(e.target.value)} />{/\* preencher o valor dos views dos states \*/}

**dentro da const handleSubmit**

setError("")

    //validação de senha

    const user = {

      displayName,

      email,

      password

    }

    if(password !== confirmPassword){

      setError("As senhas precisam ser iguais!")

      return

**Authenticação**

**hooks>useAuthentication.js**

**funções a serem importadas**

import {

    getAuth,

    createUserWithEmailAndPassword,

    sigInWithEmailAndPassword,

    updateProfile,

    signOut

} from 'firebase/auth'

import { useEffect, useState } from 'react'//funcao do react necessarias

//db de configs do firebase não será usado para salvar autenticacao de usuario

**Função hook**

export const useAuthentication = () => {//funcao hook

    const [error, setError] = useState(null)

    const [loading, setLoading] = useState(null)

    //cleanup - como haverá muitas mudanças de componentes entre páginas, 'resquícios' de funções n podem ficar sendo executados

    //deal with memory leak - lidar com vazamento de memória

    const [cancelled, setCancelled] = useState(false)//as ações futuras do componente serão canceladas(assim que tudo der certo e houver a mudança de false para true)

    const auth = getAuth()//nao quer dizer que usuario esteja autenticado, mas sim que funções de autenticação poderão ser usadas a partir desta função

    function checkIfIsCancelled() {

        if(cancelled){

            return

        }

    }

}

**Contexto de autenticação**

export function AuthProvider({ children, value }) {//valor para poder compartilhar no contexto

    return <AuthContext.Provider value={value}>{children}</AuthContext.Provider>

}

export function useAuthValue() {

    return useContext(AuthContext)

}

**Import**

mport { useAuthentication } from '../hooks/useAuthentication'//será utilizado para a barra de navegação

import { useAuthValue } from '../context/AuthContext'//pegar valor do contexto

{!user && (//se nao tiver usuario, serao exibidos os seguintes links:

**Aula – função logout**

**useAuthentication**

//funções são retornadas para que assim seja possível utilizá-las em outro lugar

//logout - sign out

const logout = () => {

        checkIfIsCancelled()

        signOut(auth)//passando quem está autenticado

    }

**Navbar.js**

const { logout } = useAuthentication()

{user && (<li><button onClick={logout}>Sair</button></li>)}{/\* se o usuário estiver logado,

        ]exiba o botao "Sair", que contém a função {logout} criada no hook(useAuthentication) \*/}

**App.js**

 useEffect(() => {

    onAuthStateChanged(auth, (user) => {//auth - basicamente uma classe que possui funcionalidades para gerenciar autenticação

    //user foi colocado, pois é o que eu quero passar na variável setUser

      setUser(user)

    })

  }, [auth])//smp que mudar a autenticação, o useEffect será executado

  //aula logout - auth muda, executa função novamente e entrega novo user, contexto compartilha usuário que

  //não existe mais e automaticamente não temos mais o usuário