**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**.

Excluindo um nome aleatório de acordo com o click no botão

<ul>

        {users.map((user) => (

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

        ))}

      </ul>

      <button onClick={deleteRandom}>Delete random user</button>

const [users, setUsers] = useState([

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

    { id: 2, name: "João", age: 28 },

    { id: 3, name: "Pedro", age: 44 },

  ])

  const deleteRandom = () => {

    const randomNumber = Math.floor(Math.random() \* 4 )

    setUsers((prevUsers)=>{

      console.log(prevUsers)

      return prevUsers.filter((user) => randomNumber !== user.id)

    })}

* + 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.

const ConditionalRender = () => {

    const [x] = useState(true)

  return (

    <div>

        <h1>Isso será exibido?</h1>

        {x  && <p>Se x for true, sim!</p>}

        {!x && <p>Agora x é falso</p>}

    </div>

  )

}

* + 1. **Adicionando um else**

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

Aqui, usa-se o **if ternário**;

Síntaxe: condição? bloco1 : bloco2.

        <button onClick={setName=("João")}>Clica aqui!</button>

{name === "João" ? (

            <div>

                <p>O nome é João</p>

            </div>

        ) : (

            <div>

                <p>Nome não encontrado</p>

            </div>

        )}

* + 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;**

  const name = "Joaquim"

  const [userName] = useState("Maria")

<ShowUserName name={userName}/>

  const name = "Joaquim"

<ShowUserName name={name}/>

* + 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

import React from 'react'

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

    return (

        <div>

            <h2>Detalhes do carro</h2>

            <ul>

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

                <li>KM: {km}</li>

                <li>Cor: {color}</li>

            </ul>

        </div>

    )

}

* + 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 prop**

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>