**TypeScript**

**O que é?**

È um **superset** para a linguagem JavaScript: adiciona funções ao JavaScript, como a declaração de variável;

Pode ser utilizado com frameworks/libs, como: Express e React;

Precisa ser compilado em JavaScript, ou seja, não executamos TS;

Desenvolvido e mantido pela Microsoft.

**Por que TypeScript?**

Traz **confiabilidade** ao programa(tipos);

Prove novas funcionalidades a JS, como **Interfaces**;

Podem ser **verificados** os **erros** **antes da execução** do código, ou seja, no desenvolvimento;

Deixa JS **mais explícito**, diminuindo a quantidade de bugs;

Por estes e outros motivos perde-se **menos tempo com debug**.

**Instalando TS**

Utiliza-se **npm**(node.js);

Nome do pacote: **typescript**;

Será adicionado de forma global com: **flag -g** (para ter acesso a ele em qualquer local);

A partir da instalação, o TS pode ser executado/compilado em qualquer local da máquina, com o comando **tesc**;

**npm install typescript -g**

**npm tsx -v**

**Primeiro programa com TS**

Intuito: compilar e executar arquivo gerado pelo processo de compilação.

Terminal do cód.: tsc index.ts

**TS com React**

**React com TS**

Adicionar TS no React traz diversas possibilidades;

Seguindo a mesma linha de JS, é uma forma mais padronizada para programar;

Como **tipos** para componentes ou mapeamento de props por meio da interface;

Isso traz confiabilidade ao projeto, fazendo com que TS seja cada vez mais utilizado.

**Instalando React com TS**

No terminal: **create-react-app** + **-template** (contém o valor de typescript);

Um novo projeto é criado com arquivos ‘**.tsx**’;

***npx create-react-app <nomedoprojeto> --template typescript***.

**Estrutura do React com TS**

Não muda muito comparada a estrutura normal com JS;

Existem as pastas clássicas***: node\_modules***, ***src*** e ***public***;

A pasta **src** é onde há algumas mudanças, criação de **arquivos** em ***.tsx***;

São esses **arquivos jsx**, porém, com funcionalidades de TS;

Projeto pode ser executado com ***npm run start***.

**Criação de variáveis em componentes**

Essas variáveis podem receber os tipos já vistos até este momento do curso;

Isso, permite com que trabalhemos com JSX com apoio dessas variáveis e seus tipos.

const name: string = "Matheus";

  const age: number = 30;

  const isWorking: boolean = true;

  return (

    <div className="App">

      <h1>TypeScript com React</h1>

      <h2>Nome: {name}</h2>

      <p>Idade: {age}</p>

      {isWorking && <p>Está trabalhando!</p>}

    </div>

  );

**Criação de funções em componentes**

Essas funções recebem **parâmetros**, que **podem ser tipados**;

Retorno também;

Podem ser aplicados os mesmos conceitos vistos antes em TS.

//2 - funções

  const userGreeting = (name: string): string => {

    return `Olá, ${name}`;

  };

<h3>{userGreeting(name)}</h3>

      {/\* se eu tentar colocar o age por exemplo, a variável será recusada, por ser um number e não uma string \*/}

**Componentes**

**Extensão para React com TS**

Extensão que será utilizada: **TypeScript React code snippets**;

Essa extensão ajuda com atalhos para programar mais rápido, como o **tsrafce (**cria um componente funcional**)**;

**tsrafce**: **ts** – typescript

**r** – react

**af** – arrow function

**ce** – componente

Tornando assim, o dia a dia mais simples.

**Importando Componentes**

Funciona da mesma forma que seria sem o TS;

Porém, precisamos atentar-se aos valores e tipos das props de cada componente;

O TS interage de uma forma mais sucinta na parte da importação.

//4 - importação de components

import FirstComponent from "./components/FirstComponent";

      <FirstComponent />

**Destructuring nas props**

**Destructuring** - Recurso do ES6 que premite separar um array ou objeto em várias variáveis;

Esta técnica é aplicada nas props, para não ser preciso repetir o nome do objeto muitas vezes;

Em TS é possível fazer isso também.

type Props = {

  name: string;

};

const SecondComponent = (props: Props) => {

  return (

    <div>

      <p>Meu segundo componente.</p>

      <p>O nome dele é {props.name}</p>

    </div>

<SecondComponent name="Segundo" />

type Props = {

  title: string;

  content: string;

  commentsQty: number;

  tags: string[];

};

const Descructuring = ({ title, content, commentsQty, tags }: Props) => {

  //ação de desctructuring: {<nomevar>}: type(Props)

<Descructuring

        title="Primeiro post"

        content="Algum conteúdo"

        commentsQty={10}

        tags={["ts", "js"]}

      />

**O hook useState**

Muito utilizado em React;

Serve para consultar e alterar o estado de algum dado;

Atrelando uma função set a um evento como uma mudança de dado em input, é possível alterar o valor da variável base;

Este recurso é adaptável em TS.

import React, { useState, ChangeEvent } from "react"; //chamando hook

const State = () => {

  const [text, setText] = useState<string | null>(null); //declara variável e como valor do useState, colocar como variável começará

                          //é string ou null

  const handleChange = (e: ChangeEvent<HTMLInputElement>) => { //e-evento(valor vindo do input) //declara valor do evento(changeevent) e mostra oq vai ser modificado com esse evento

    setText(e.target.value)//target - input / value - valor do input(texto que será alterado)

  };

**Enum**

Forma interessante de **formatar** um **objeto** com chaves e valores;

Podemos utilizar **como props**;

Passando a chave pela prop, imprimindo valor no componente.

type Props = {

  //8 - enum

  category: Category;

};

export enum Category {

  JS = "JavaScript",

  TS = "TypeScript",

  P = "Phyton",

}

import Destructuring, { Category } from "./components/Destructuring";

<Destructuring

        category={Category.P}

      />

**Types (Padrão da versão atual)**

Além das interfaces, podemos criar estruturas de tipos com o **type**;

Isso nos permite criar dados com tipos de dados fixos.

//8 - type

type textOrNull = string | null;

type fixed = "Isso" | "Ou" | "Aquilo"

 const myText: textOrNull = "Tem algum texto aqui";

  let mySecondText: textOrNull = null

  mySecondText = "opa"

  const testandoFixed: fixed = "Isso"

 { myText && <p>Tem texto na variável</p>}

 { mySecondText && <p>Tem texto na variável</p>}

**Context API**

Uma forma de transmitir dados entre componentes no React;

A ideia principal é que podemos determinar quais componentes recebem estes dados, ou seja, fazem parte do contexto;

Esta funcionalidade pode ser aplicada a TS.

//9 - context

interface IAppContext {//interface - declarando tipos para variáveis

  language: string;

  framework: string;

  projects: number;

}

  const contextValue: IAppContext = {//obj

    language: "JavaScript",

    framework: "Express",

    projects: 5,

  };

//function context

export const AppContext = createContext<IAppContext | null>(null); //precisa ser exportado para poder ser acessado

//criando context que recebe valor de dado suportado

 return (

    <AppContext.Provider value={contextValue}>

 </AppContext.Provider>

  );

}

export default App;

**Consumindo o dado do contexto (consumindo context)**

Será preciso utilizar um hook: useContext;

A partir dele, será possível extrair dados e assim utilizar em um componente.

***App.tsx***

//10 - utilizando contexto

import Context from './components/Context'

 <Context />

import React, { useContext } from "react";

import { AppContext } from "../App";

const Context = () => {

  const details = useContext(AppContext);

  return (

    <>

      {/\*se houver detalhes, será executado\*/}

      {details && (

        <div>

          <h2>Linguagem: {details.language}</h2>

          <h2>FW: {details.framework}</h2>

          <h2>QTD: {details.projects}</h2>

        </div>

      )}

    </>

  );

};