Tailwind CSS

O que é

- Tailwind CSS é um framework de CSS utilitário que permite criar designs modernos sem sair do seu HTML.
- Conceito de utility-first: Classes utilitárias pequenas e compostas que podem ser combinadas para construir qualquer design diretamente no HTML
- Ele pode ser usado em qualquer aplicação Web, não necessariamente só em aplicações React
- https://tailwindcss.com

Instalando

1. Vamos instalar o Tailwind em um projeto React, mas lembrando que poderíamos usar ele até em um projeto de HTML puro. Vamos criar o projeto com o comando npm create vite@lastest e depois rodar o npm install dentro da pasta.

2. Feito isso vamos rodar o comando npm install tailwindcss @tailwindcss/vite. O comando vai instalar 2 bibliotecas, o tailwind e o plugin do tailwind para o vite

Instalando

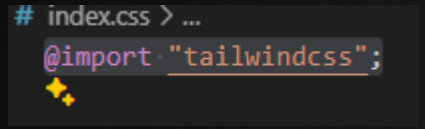
Vamos adicionar o plugin ao nosso vite.config.ts

```ts
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react-swc'
import path from 'path'
import tailwindcss from '@tailwindcss/vite'

// https://vite.dev/config/
export default defineConfig({
  plugins: [react(), tailwindcss()],
  resolve: {
    alias: {
      '@': path.resolve(process.cwd(), './src'),
    },
  },
})

```

Instalando

Agora vamos adicionar as regras de css básicas do Tailwind ao nosso index.css:

@import "tailwindcss";



Instalando

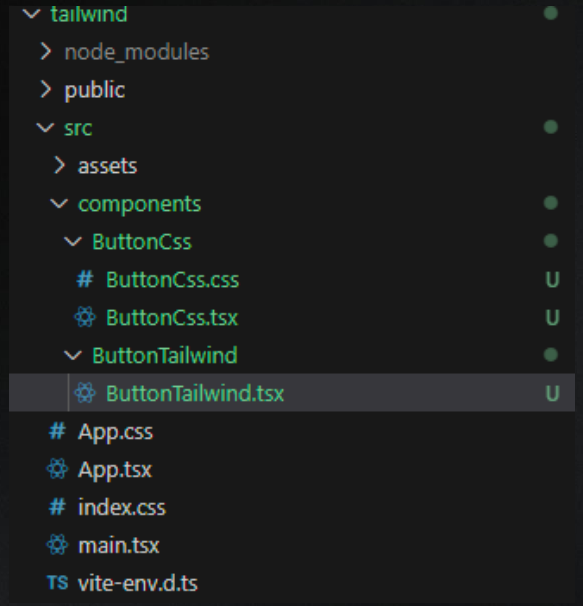
7. Vamos testar no App.tsx, vamos limpar tudo e deixar da seguinte forma:

```
You, 1 second ago | 1 author (You)
1  import './App.css'
2
3  function App() {
4
5    return (
6      <h1 className="text-3xl font-bold underline">
7        Hello world!
8      </h1>
9    )
10 }
11
12 export default App
```

Utility-First Fundamentals

Na prática, vamos ver a diferença de escrever um layout com CSS e outro usando o Tailwind com o Utility-First. Vamos criar um botão qualquer com HTML e CSS dentro do React.

1. Vamos começar criando os nossos arquivos na seguinte estrutura:



Utility-First Fundamentals

2. Agora vamos criar o nosso ButtonCss

```tsx
import { HTMLAttributes, PropsWithChildren } from "react"
import './ButtonCss.css'

export interface ButtonProps extends HTMLAttributes<HTMLButtonElement>, PropsWithChildren {
  icon?: string
}

const ButtonCss = ({ children, icon, ...props }: ButtonProps) => {
  return (
    <button {...props} className={`button ${props.className || ''}`}>
      <div className="button-icon">
        {icon || '👍'}
      </div>
      <div className="button-content">
        {children}
      </div>
    </button>
  )
}

export default ButtonCss
```

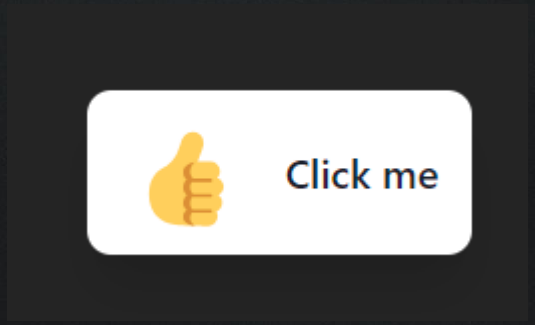
tailwind > src > components > ButtonCss > # ButtonCss.css > .button-content

```css
.button {
  display: flex;
  align-items: center;
  margin: 0 auto;
  padding: 1rem;
  border-radius: 0.75rem;
  background-color: #fff;
  box-shadow: 0 20px 25px -5px rgba(0, 0, 0, 0.1), 0 10px 10px -5px rgba(0, 0, 0, 0.04);
  gap: 1rem;
}

.button-icon {
  flex-shrink: 0;
  font-size: 3rem;
  line-height: 1;
}

.button-content {
  color: #1a202c;
  font-size: 1.25rem;
  line-height: 1.25;
}
```

Utility-First Fundamentals

3. Vamos coloca-lo no App.tsx

```tsx
function App() {

  return (
    <>
      <ButtonCss>Click me</ButtonCss>
    </>
  )
}

export default App
```



Utility-First Fundamentals

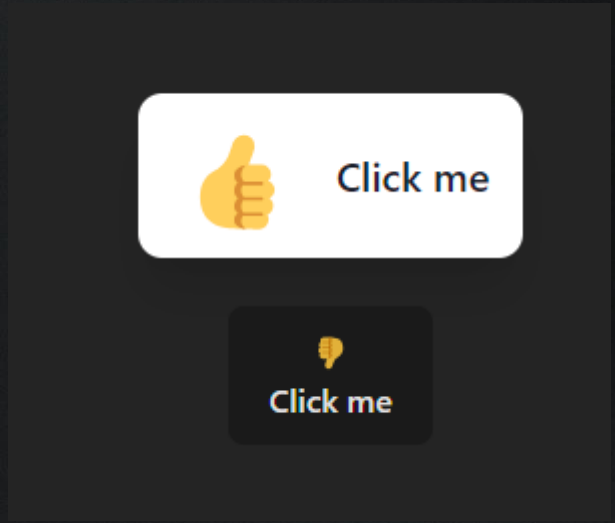
4. Agora vamos trabalhar no mesmo botão só que sem escrever uma única linha de CSS, usando o TailwindCSS. Vamos copiar o conteúdo do arquivo ButtonCss.tsx e colar no arquivo ButtonTailwind.tsx
5. Depois vamos incluir no App.tsx. Vamos passar um ícone diferente só para identificarmos qual é qual

```tsx
function App() {

  return (
    <>
      <ButtonCss>Click me</ButtonCss>
      <br />
      <ButtonTailwind icon='👇'>Click me</ButtonTailwind>
    </>
  )
}

export default App
```

Utility-First Fundamentals

6. Vamos remover todas as classes do botão anterior e pagar a importação do CSS no nosso ButtonTailwind.tsx

```tsx
import { HTMLAttributes, PropsWithChildren } from "react"

export interface ButtonProps extends HTMLAttributes<HTMLButtonElement>, PropsWithChildren {
  icon?: string
}

const ButtonTailwind = ({ children, icon, ...props }: ButtonProps) => {
  return (
    <button {...props} className={` ${props.className || ''}`}>
      <div className="">
        {icon || '👍'}
      </div>
      <div className="">
        {children}
      </div>
    </button>
  )
}

export default ButtonTailwind
```



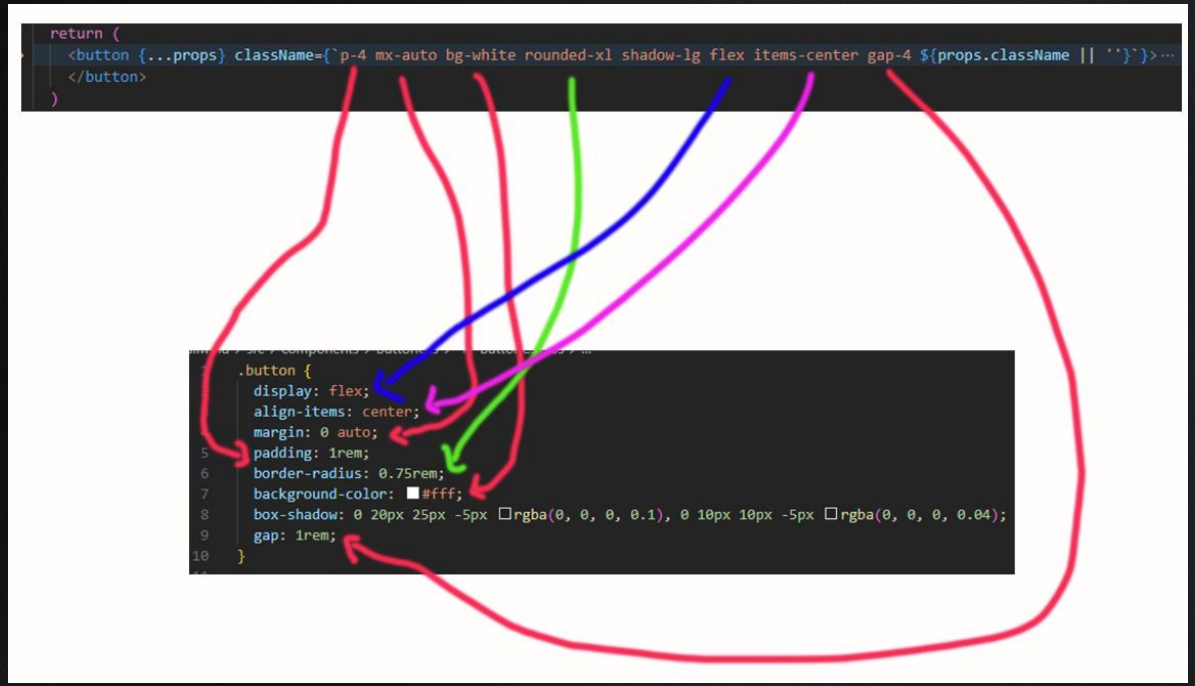
Utility-First Fundamentals

7. Agora vamos deixar o CSS igual no novo botão, apensar usando as classes do Tailwind

```tsx
const ButtonTailwind = ({ children, icon, ...props }: ButtonProps) => {
  return (
    <button {...props} className={`p-4 mx-auto bg-white rounded-xl shadow-lg flex items-center gap-4 ${props.className || ''}`}>
      <div className="shrink-0 text-5xl">
        {icon || '👍'}
      </div>
      <div className="text-xl font-medium text-black">
        {children}
      </div>
    </button>
  )
}

export default ButtonTailwind
```

Utility-First Fundamentals

Comparando as classes que escrevemos no ButtonCss.css com as classes que usamos do Tailwind:



Tailwind CSS

São muitas classes e regras sobre o Tailwind então a orientação é: Sempre procurar na documentação. Quase todas as regras de CSS possuem alguma equivalência no Tailwind, e com a prática isso vai se tornando mais e mais automático dentro da nossa cabeça, assim economizamos tempo e linhas de código sem precisar escrever mais CSS.

Para saber mais:
https://tailwindcss.com/docs/utility-first

Exercicios

1. Volte ao exercício anterior da aula de React Router, da listagem de posts + pagina da publicação, e tente manter o layout usando apenas as classes do Tailwind, removendo todo o CSS escrito. Pode ter adaptações e melhorias no Layout, mas precisa ser parecido com o anterior

2. Volte ao exercício de TODO-LIST e refaça o layout usando o Tailwind

Dúvidas, críticas ou sugestões?


FIAP