

FIAP

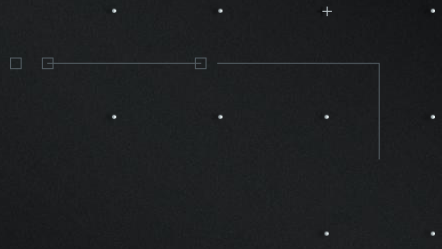
FIAP

SLIDER ▢■◀





React - Props



O que é

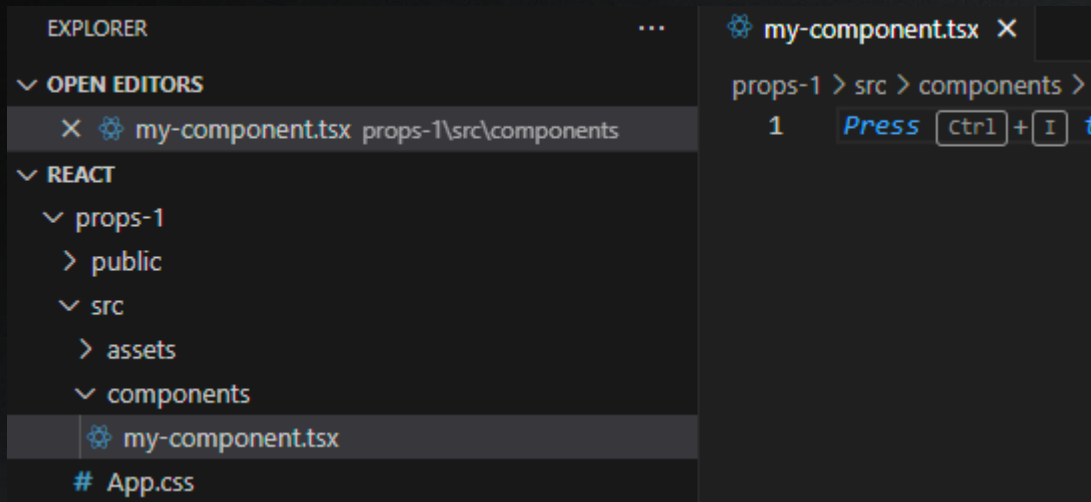
Props são mecanismos que permitem transmitir dados de um componente pai para um componente filho. São uma forma resumida de dizer PROPriedadeS.

As **Props** são o equivalente a argumentos de funções, aceitando variados tipos de dados e funções. O uso de **props** é uma ótima maneira de tornar seus componentes mais flexíveis e reutilizáveis.

Lembrando que o fluxo do React entre os componentes é unidirecional, então só podemos passar props de pai para o filho.

Props

1. Vamos criar uma aplicação nova, com o comando `npm create vite@latest` e `npm install`
2. Dentro da pasta `src`, vamos criar uma pasta chamada `components` e dentro dela um novo arquivo chamado `my-component.tsx`



Props

3. Vamos criar um componente que renderiza um texto qualquer, e vamos chamar de **MyComponent**

```
my-component.tsx X
props-1 > src > components > my-compor
1  const MyComponent = () => {
2    return (
3      <p>
4        Hello, world!
5      </p>
6    )
7  }
8
9  export default MyComponent
```


Props

4. Vamos até o nosso App.tsx e apagar o que vem por padrão

props-1 > src > App.tsx > ...

```
1  import './App.css'
2
3  function App() {
4    return (
5      <>
6
7      </>
8    )
9  }
10
11  export default App
12
```

Props

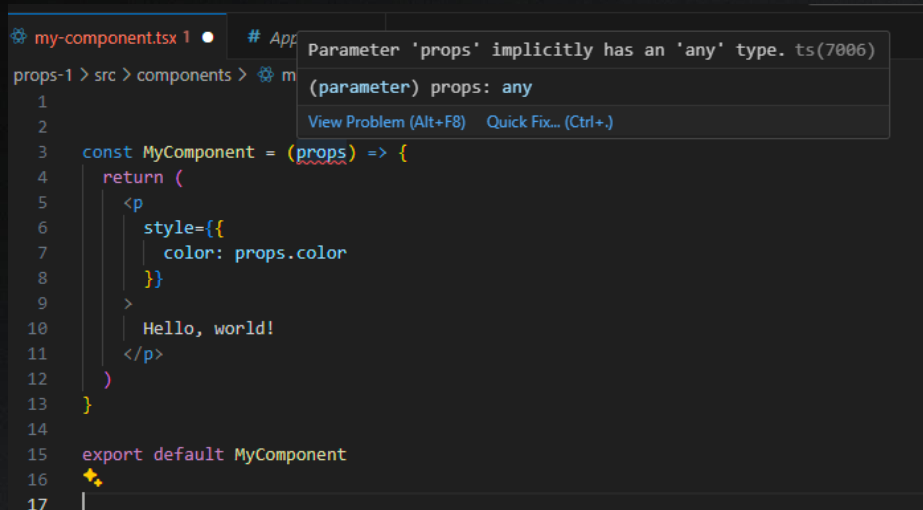
5. Agora vamos incluir o nosso componente

```
props-1 > src > App.tsx > ...  
1  import './App.css'  
2  import MyComponent from './components/my-component'  
3  
4  function App() {  
5    return (  
6      <>  
7        <MyComponent />  
8      </>  
9    )  
10 }  
11  
12 export default App  
13
```

Hello, world!

Props

5. De volta ao nosso componente, como argumento da função, vamos passar as **props** que sempre vai ser um objeto de Javascript. Vamos acessar a propriedade **colors** dentro de props e setar a cor do nosso texto para ser uma propriedade do nosso componente. Porém o nosso editor esta reclamando da typagem. Como estamos usando TypeScript, precisamos criar uma interface para as nossas props



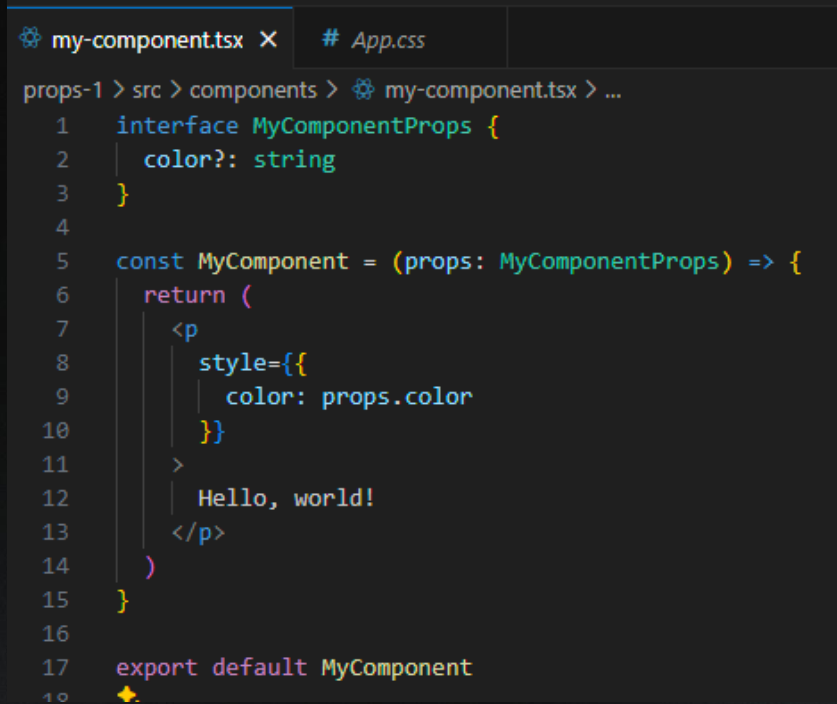
The screenshot shows a code editor with a file named `my-component.tsx`. The editor displays a TypeScript error: "Parameter 'props' implicitly has an 'any' type. ts(7006)". Below the error, a tooltip shows the function signature: `(parameter) props: any`. The code snippet defines a component `MyComponent` that takes `props` as an argument and returns a JSX element. The `style` object is set to `{color: props.color}`. The component is exported as the default export.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
const MyComponent = (props) => {
  return (
    <p
      style={{
        color: props.color
      }}
    >
      Hello, world!
    </p>
  )
}

export default MyComponent
```


Props

6. Vamos criar uma interface no mesmo arquivo e atribuir o tipo a props



The screenshot shows a code editor with two tabs: 'my-component.tsx' and '# App.css'. The active tab is 'my-component.tsx', which displays the following TypeScript code:

```
props-1 > src > components > my-component.tsx > ...
1  interface MyComponentProps {
2    color?: string
3  }
4
5  const MyComponent = (props: MyComponentProps) => {
6    return (
7      <p
8        style={{
9          color: props.color
10        }}
11      >
12        Hello, world!
13      </p>
14    )
15  }
16
17  export default MyComponent
```

Props

7. Vamos voltar ao App.tsx e passar a cor red como atributo de cor do nosso componente

```
App.tsx
props-1 > src > App.tsx > ...
1  import './App.css'
2  import MyComponent from './components/my-component'
3
4  function App() {
5    return (
6      <>
7        <MyComponent color='red' />
8      </>
9    )
10 }
11
```

Hello, world!

Props - Destructuring Objects

Também podemos acessar as informações de props usando o destructuring do Javascript, que é uma expressão que permite descompactar valores de arrays, ou propriedades de objetos, em variáveis distintas. Então vamos mudar um pouco as props que acabamos de criar

```
1 interface MyComponentProps {  
2   color?: string  
3 }  
4  
5 const MyComponent = ({ color }: MyComponentProps) => {  
6   return (  
7     <p  
8       style={{  
9         color: color  
10      }}  
11     >  
12       Hello, world!  
13     </p>  
14   )  
15 }  
16  
17 export default MyComponent
```

Dessa forma o código fica mais limpo e direto ao ponto. Mas ambas as abordagens funcionam!

Props - Children

No React temos uma propriedade especial chamada **children**. Ela é uma forma de passarmos components ou outros elementos HTML para dentro do nosso componente de forma aninhada. Vamos aplicar ao nosso componente para ficar mais claro

1. Primeiro, como estamos usando TypeScript, precisamos avisar que ele tem essa propriedade. O React tem tipos criados dentro da biblioteca que podemos usar:

```
1 import { PropsWithChildren } from 'react'
2
3 interface MyComponentProps extends PropsWithChildren {
4   color?: string
5 }
6
7 const MyComponent = ({ color }: MyComponentProps) => {
8   return /
```

O que fizemos: extendemos uma classe do React para herdarmos o children do React. Dessa forma já vem com o tipo correto sem precisarmos criar.

Props - Children

2. Agora vamos colocar a prop children

```
rops-1 > src > components > my-component.tsx > ...
1  import { PropsWithChildren } from 'react'
2
3  interface MyComponentProps extends PropsWithChildren {
4    color?: string
5  }
6
7  const MyComponent = ({ color, children }: MyComponentProps) => {
8    return (
9      <p
10        style={{
11          color: color
12        }}
13      >
14        Hello, world!
15      </p>
16    )
17  }
18
19  export default MyComponent
20  ✨
```


Props - Children

3. Vamos substituir o texto estático “Hello, world!” pelo children. Vamos usar `{}` (chaves) que no React funcionam como interpolação de variável. Se notarmos nossa aplicação ficou sem texto algum.

```
props-1 > src > components > my-component.tsx > ...
1  import { PropsWithChildren } from 'react'
2
3  interface MyComponentProps extends PropsWithChildren {
4    color?: string
5  }
6
7  const MyComponent = ({ color, children }: MyComponentProps) => {
8    return (
9      <p
10        style={{
11          color: color
12        }}
13      >
14        {children}
15      </p>
16    )
17  }
18
19  export default MyComponent
20
```


Props - Children

4. Agora vamos passar o children para o nosso componente, que é o nosso texto, lá no App.tsx. Assim voltamos ao resultado anterior mas usando o children.

```
function App() {  
  return (  
    <>  
      <MyComponent color='red'>  
        Hello, World!  
      </MyComponent>  
    </>  
  )  
}
```

O Children pode ser qualquer coisa, de texto, a elementos HTML ou Components React.

Props – Valor padrão

Também podemos atribuir valores padrão para as nossas props, de uma forma bem simples, basta igualarmos nossa variavel a um valor específico. Vou definir a cor padrão como **blue**. Assim, na ausência da propriedade quando o componente for usado, a cor padrão vai ser o azul.

```
props-1 > src > components > my-component.tsx > ...  
1 import { PropsWithChildren } from 'react'  
2  
3 interface MyComponentProps extends PropsWithChildren {  
4   color?: string  
5 }  
6  
7 const MyComponent = ({ color = 'blue', children }: MyComponentProps) => {  
8   return (  
9     <p  
10       style={{  
11         color: color  
12       }}  
13     >  
14       {children}  
15     </p>  
16   )  
17 }  
18  
19 export default MyComponent  
20  
21
```

`<MyComponent>`
Hello, World!
`</MyComponent>`

Hello, World!

Props – Passando valor Filho -> Pai

Como dito anteriormente, o fluxo do React é unilateral de Pai pra filho. Mas existem formas de pegar um valor de dentro de um componente filho e passar para um componente pai. Uma dessas formas é através das **Props**.

A abordagem consiste em passar uma função como Prop que será executada em dado momento no componente filho, recebendo o valor a ser transferido para o pai

Props – Passando valor Filho -> Pai

1. Dentro do nosso componente, vamos adicionar um botão.
2. Vamos colocar na nossa interface uma função chamada de identifyClick
3. Vamos receber a função via Props no componente, e no onClick do botão vamos invocar a função

```
1 import { PropsWithChildren } from 'react'
2
3 interface MyComponentProps extends PropsWithChildren {
4   color?: string
5   identifyClick?: () => void
6 }
7
8 const MyComponent = ({ color = 'blue', identifyClick, children }: MyComponentProps) => {
9   return (
10     <>
11       <p
12         style={{
13           color: color
14         }}
15       >
16         {children}
17       </p>
18       <button onClick={() => identifyClick ? identifyClick() : null}>
19         Click
20       </button>
21     </>
22   )
23 }
24
25 export default MyComponent
26
27 ▲
```

Props – Passando valor Filho -> Pai

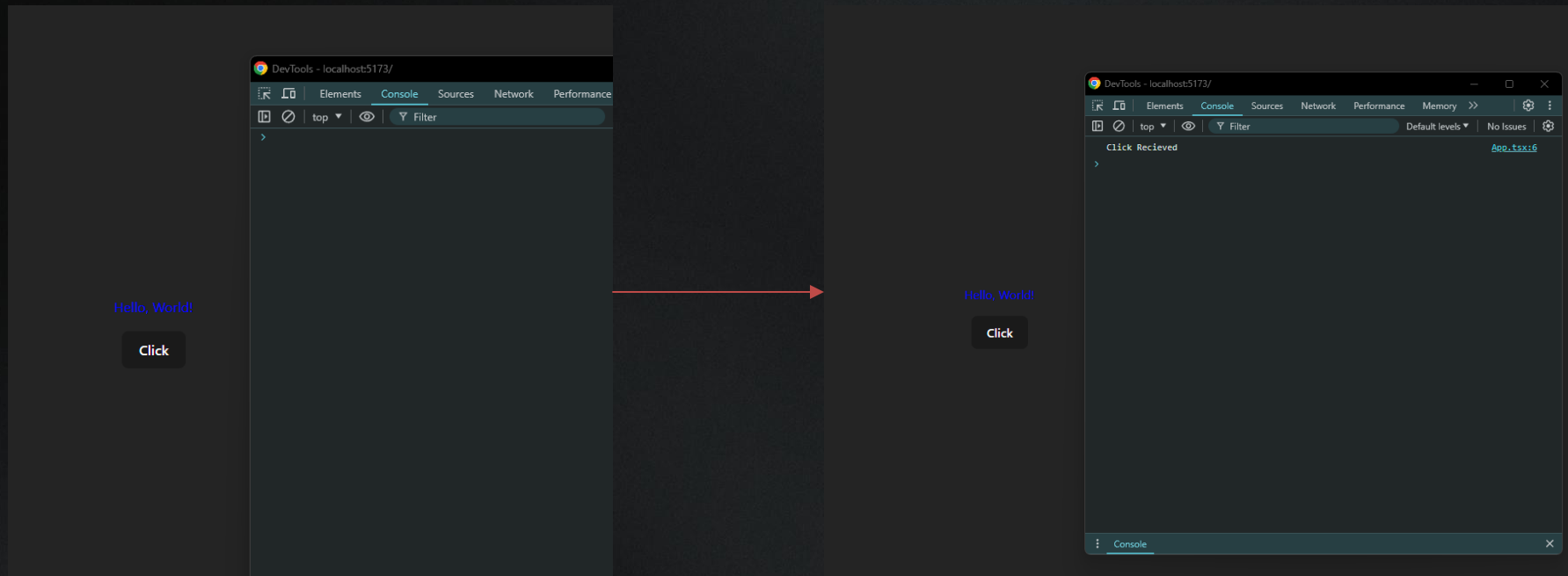
4. Agora no App.tsx onde chamamos o botão, vamos criar uma nova função e passar como parâmetro.

props > src > App.tsx > ...

```
1  import './App.css'
2  import MyComponent from './components/my-component'
3
4  function App() {
5    const clickRecieved = () => {
6      console.log('Click Recieved')
7    }
8
9    return (
10     <>
11       <MyComponent identifyClick={clickRecieved}>
12         Hello, World!
13       </MyComponent>
14     </>
15   )
16 }
17
18 export default App
19
```

Props – Passando valor Filho -> Pai

5. Agora na nossa aplicação, quando clicarmos no botão vamos ver o log sendo disparado. Ou seja, uma ação do componente filho executou um trecho de código no componente pai



Props – Passando valor Filho -> Pai

6. Agora vamos passar um valor que só exista no nosso filho para pai. Isso pode ser qualquer valor aceito pelo Javascript. Vamos passar o `event.currentTarget`, que vai passar nosso HTML do botão para o Pai. Poderia ser uma variável qualquer contendo valores, elementos, components, etc.

```

1  import { PropsWithChildren } from 'react'
2
3  interface MyComponentProps extends PropsWithChildren {
4    color?: string
5    identifyClick?: (button: HTMLButtonElement) => void
6  }
7
8  const MyComponent = ({ color = 'blue', identifyClick, children }: MyComponentProps) => {
9    return (
10      <>
11        <p
12          style={{
13            color: color
14          }}
15        >
16          {children}
17        </p>
18        <button onClick={(e) => identifyClick ? identifyClick(e.currentTarget) : null}>
19          Click
20        </button>
21      </>
22    )
23  }
24
25  export default MyComponent

```

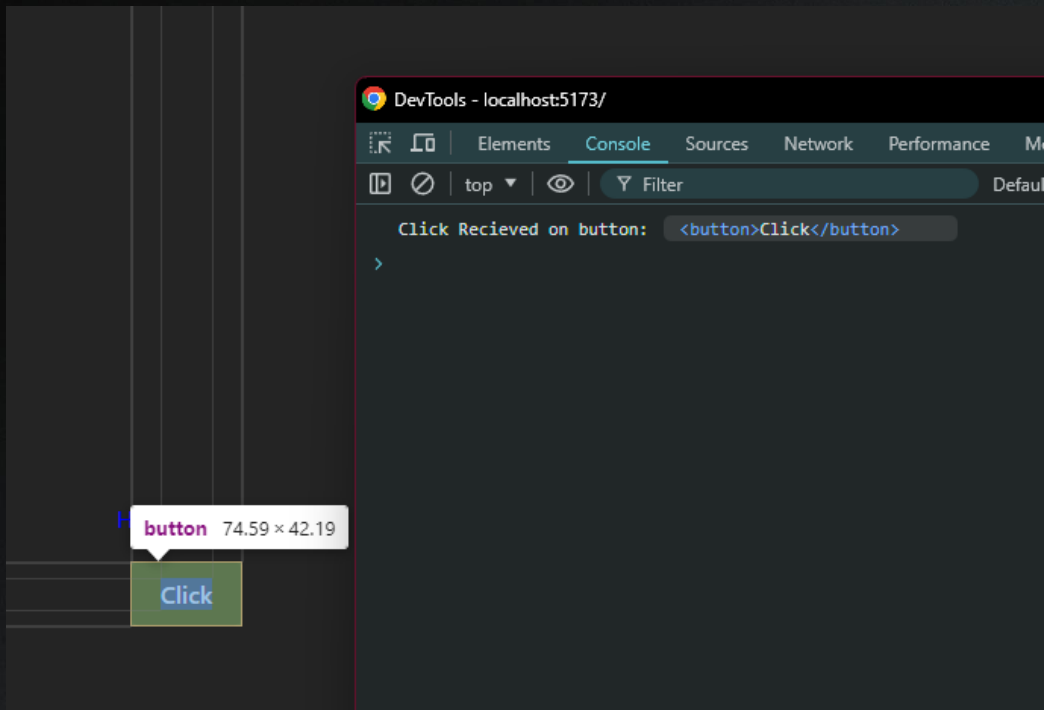
```

props > src > App.tsx > ...
1  import './App.css'
2  import MyComponent from './components/my-component'
3
4  function App() {
5    const clickRecieved = (button: HTMLButtonElement) => {
6      console.log('Click Recieved on button:', button)
7    }
8
9    return (
10      <>
11        <MyComponent identifyClick={clickRecieved}>
12          Hello, World!
13        </MyComponent>
14      </>
15    )
16  }
17
18  export default App
19

```

Props – Passando valor Filho -> Pai

7. Agora podemos ver que recebemos o nosso botão no Pai



Exercícios

1. Continuando o exercício de lista estática de usuários

Agora vamos continuar aquele exercício da aula passada. Porém ao invés de usarmos a lista no próprio componente vamos passar ele como propriedade para um componente filho. Vamos criar um componente que recebe uma lista via props e passar para o componente que criamos anteriormente.

2. Continuando o exercício de card interativo

Vamos criar uma lista de itens e passar como propriedade para um outro componente que liste os itens. Esse componente de listagem deve ter o componente de InteractiveCard que criamos nos exercícios anteriores e deve receber um objeto como parâmetro

Dúvidas, críticas ou sugestões?

FIAP