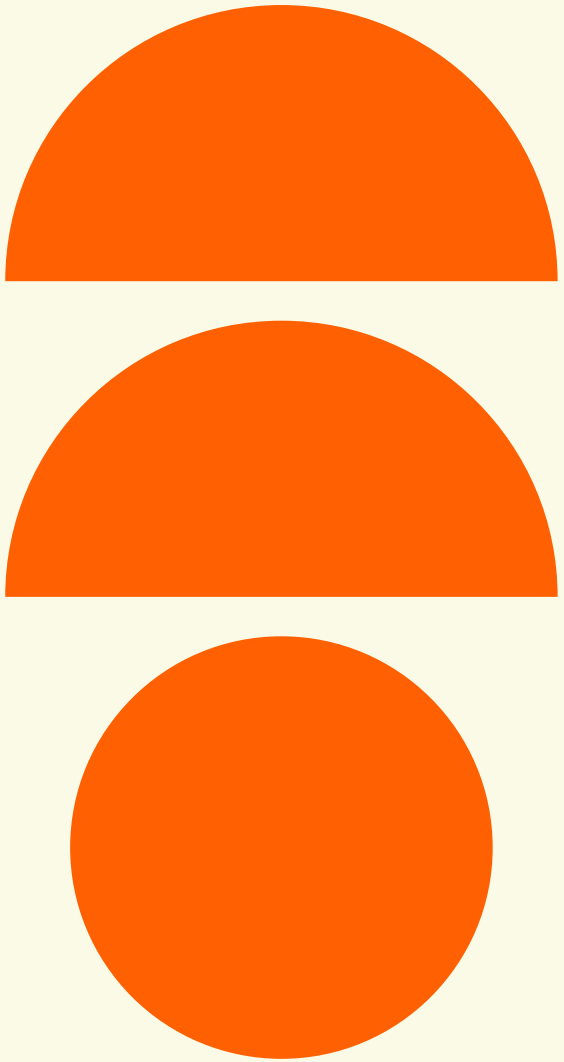
C.E.S.A.R school

NEXT

JavaScript e Front-End básico

Aula 09 - Introdução ao React Hooks



JavaScript e Front-end básico.

Aula 9

Olá!

Eu sou Lucas Marques

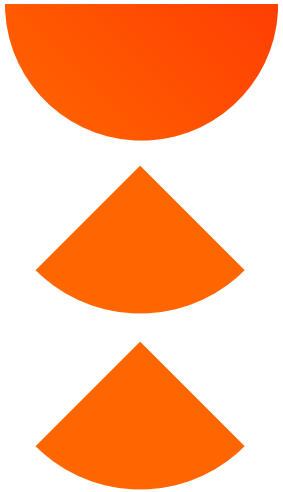
Estou aqui porque sou entusiasta de tecnologia e apaixonado por compartilhar conhecimento

Adoro facilitar o processo de ensino-aprendizagem, tornando conceitos complexos mais simples e acessíveis.

O que veremos hoje?

- Introdução aos Hooks;
- Principais Hooks;
- Custom Hooks

Introdução aos Hooks



O que são hooks?

Hooks são funções especiais do React que permitem adicionar funcionalidades como state (estado) e efeitos colaterais aos componentes funcionais. Eles foram introduzidos na versão 16.8 do React, lançada em 2019.

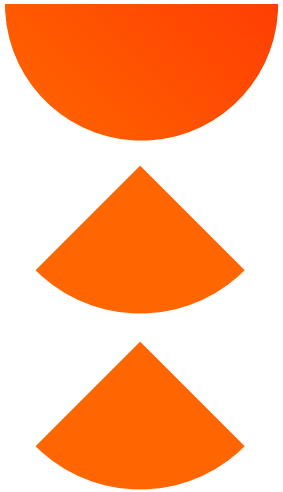
Antes dos hooks, essas funcionalidades só estavam disponíveis em componentes de classe. Com os hooks, ficou mais fácil criar componentes usando apenas funções.

- Três características importantes que precisamos saber sobre os hooks são:
  - Eles só funcionam em componentes funcionais
  - Devem ser chamados diretamente no código do componente, sem estar dentro de laços, condições ou funções internas.
  - Não podem ser executados de forma condicional; eles precisam ser chamados sempre na mesma ordem.

Reutilização de lógica

Antes dos hooks, compartilhar lógica entre componentes podia ser complicado. Para isso, usávamos técnicas como **High Order Components** (HOCs) ou **Render Props**.

O problema é que essas técnicas frequentemente exigiam que re-organizássemos o código do componente para funcionar com elas. Em situações mais complexas, isso criava muitos "aninhamentos" de componentes, algo que chamamos de "Wrapper Hell" — basicamente, muitos componentes dentro de outros, deixando o código confuso e difícil de entender. Ficaria mais ou menos assim seu HTML 👀:



Componentes muito grandes

Antes dos hooks, a lógica de uma funcionalidade muitas vezes ficava espalhada em várias partes do componente, especialmente em componentes grandes. Por exemplo, se quiséssemos adicionar event listeners (como escutar eventos do teclado ou da janela):

- Precisávamos configurá-los no método componentDidMount (quando o componente é Montado).
- Depois, verificar e ajustar no componentDidUpdate (quando o componente era atualizado).
- E, por fim, remover esses listeners no componentWillUnmount (quando o componente era removido da tela).

Tudo isso tornava o código difícil de entender e de manter, especialmente se o componente tivesse várias funcionalidades diferentes.

Com os hooks, ficou muito mais fácil agrupar toda a lógica de uma funcionalidade em um único lugar, deixando o código mais organizado, legível e simples de atualizar.

Enfim, os Hooks!! 🎣 🪝

Importação

Bom, agora que já discorremos acerca dos problemas que enfrentávamos no React antes dos hooks, hora de vê-los funcionando na prática!

[🚨 Importante!] Todos estes hooks são importados de dentro do pacote do React, desta forma:

```
import { useState, useEffect } from 'react';
// ou
import React, { useState, useEffect } from 'react';
```

useState

Permite adicionar um estado ao componente funcional.

Sempre que você precisar armazenar e atualizar valores que mudam ao longo do tempo (como inputs ou contadores).

NeXT JavaScript

```javascript
import React, { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Você clicou {count} vezes</p>
      <button onClick={() => setCount(count + 1)}>Clique aqui</button>
    </div>
  );
}
```

```javascript
function Profile() {
  const [name, setName] = useState('Mateus');

  return (
    <input
      value={name}
      onChange={e => setName(e.target.value)}
    />
  );
}
```

state

função responsável por alterar o state

useState - Regrinha

Quando atualizamos o estado com useState, o React pode agrupar múltiplas atualizações e nem sempre refletir o valor mais recente imediatamente. Para garantir que estamos sempre usando o estado atualizado, usamos a função de atualização, que recebe o estado anterior (prevState).

💡 Regra: Sempre que a nova atualização depender do valor anterior do estado, use a função de atualização em vez de setState(valorDireto).

Untitled-1 TSX

```tsx
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount((prevCount) => prevCount + 1); // Garante atualização correta
  };

  return (
    <div>
      <h2>Contador: {count}</h2>
      <button onClick={increment}>Incrementar</button>
    </div>
  );
}

export default Counter;
```

useEffect

Permite lidar com efeitos colaterais como buscar dados de uma API, configurar event listeners, ou manipular o DOM diretamente.

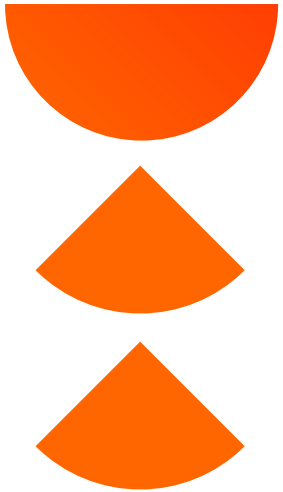
Sempre que precisar executar algo após o componente ser renderizado ou quando valores específicos mudarem.

NeXT — JavaScript

```javascript
import React, { useEffect, useState } from "react";

function SimpleEffect() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("O componente foi renderizado ou o contador mudou!");
  }, [count]); // O useEffect será executado toda vez que 'count' mudar.

  return (
    <div>
      <p>Você clicou {count} vezes</p>
      <button onClick={() => setCount(count + 1)}>Clique aqui</button>
    </div>
  );
}
```

O useEffect executa a função dentro dele toda vez que a variável count mudar.

O console exibirá a mensagem sempre que o botão for clicado, pois o estado do contador (count) está na lista de dependências do useEffect.



useEffect - Regrinhas (1)

Quando o array de dependências é **omitido**:

- O que acontece?
  - O efeito será executado toda vez que o componente renderizar (ou seja, na montagem e em todas as atualizações).
- Quando usar?
  - Geralmente, não recomendado, porque pode causar problemas de desempenho. Útil em casos raros, como para depuração ou quando você deseja executar algo em todas as renderizações.

NeXT JavaScript

```
useEffect(() => {
  console.log("Componente renderizou");
});
```

useEffect - Regrinhas (2)

Quando o array de dependências é **vazio** ([ ]):

- O que acontece?
  - O efeito será executado apenas uma vez, quando o componente é montado.
- Quando usar?
  - Para efeitos que só precisam rodar na montagem do componente, como chamadas iniciais de API ou configuração de listeners.

NeXT JavaScript

```
useEffect(() => {
  console.log("Componente montado");
}, []);
```

useEffect - Regrinhas (3)

Quando o array de dependências **contém** variáveis:

- O que acontece?
  - O efeito será executado na montagem do componente e sempre que as variáveis no array mudarem.
- Quando usar?
  - Para efeitos que dependem de mudanças específicas, como atualizar algo baseado em valores ou props.

NeXT JavaScript

```
useEffect(() => {
  console.log(`O contador mudou para: ${count}`);
}, [count]);
```

useContext

Permite acessar valores de um Contexto sem precisar passar props manualmente por vários níveis.

Para compartilhar dados como tema, idioma ou informações de usuário entre vários componentes.

NeXT JavaScript

```javascript
import React, { useContext, createContext } from "react";

const ThemeContext = createContext();

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button style={{ backgroundColor: theme }}>Clique aqui</button>;
}

function App() {
  return (
    <ThemeContext.Provider value="lightblue">
      <ThemedButton />
    </ThemeContext.Provider>
  );
}
```

useRef

Cria uma referência mutável que persiste entre renderizações. Pode ser usada para acessar elementos DOM diretamente ou armazenar valores que não causam re-renderização.

Para acessar um elemento HTML diretamente ou armazenar valores sem re-renderizar o componente.

NeXT JavaScript

```
import React, { useRef } from "react";

function TextInputFocus() {
  const inputRef = useRef();

  const handleFocus = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleFocus}>Focar no input</button>
    </div>
  );
}
```

NeXT JavaScript

```
import React, { useRef, useState } from "react";

function ClickTracker() {
  const clickCount = useRef(0); // Valor que não dispara re-renderizações
  const [displayCount, setDisplayCount] = useState(0);

  const handleClick = () => {
    clickCount.current += 1; // Incrementamos o valor
    console.log(`Cliques totais: ${clickCount.current}`);
    setDisplayCount(clickCount.current); // Atualizamos apenas o que queremos exibir
  };

  return (
    <div>
      <button onClick={handleClick}>Clique aqui</button>
      <p>Cliques exibidos: {displayCount}</p>
    </div>
  );
}
```

useReducer

useReducer é um hook usado para gerenciar estados mais complexos ou que envolvem múltiplas ações. Ele funciona de maneira parecida com o useState, mas oferece mais controle, especialmente quando temos várias mudanças no estado relacionadas umas às outras.

NeXT JavaScript

```
import React, { useReducer } from "react";

function Counter() {
  // Função redutora
  const reducer = (state, action) => {
    switch (action.type) {
      case "increment":
        return { count: state.count + 1 };
      case "decrement":
        return { count: state.count - 1 };
      default:
        return state;
    }
  };

  // useReducer recebe a função redutora e o estado inicial
  const [state, dispatch] = useReducer(reducer, { count: 0
});

  return (
    <div>
      <p>Contador: {state.count}</p>
      <button onClick={() => dispatch({ type: "increment"
})}>Incrementar</button>
      <button onClick={() => dispatch({ type: "decrement"
})}>Decrementar</button>
    </div>
  );
}
```

O **reducer** controla como o estado muda com base no tipo de ação.

O **dispatch** dispara ações, como { type: "increment" }.

state é o estado atual.

useMemo

useMemo é um hook usado para memorizar valores calculados. Ele evita que cálculos pesados sejam repetidos desnecessariamente quando o componente renderiza novamente.

Quando há cálculos complexos ou demorados que não precisam ser recalculados sempre.

NeXT — JavaScript

```
import React, { useState, useMemo } from "react";

function DoubleCalculator() {
  const [number, setNumber] = useState(0);

  // Memoriza o cálculo do dobro
  const doubled = useMemo(() => {
    console.log("Calculando o dobro...");
    return number * 2;
  }, [number]); // Recalcula apenas se 'number' mudar

  return (
    <div>
      <input
        type="number"
        value={number}
        onChange={(e) => setNumber(Number(e.target.value))}
      />
      <p>O dobro é: {doubled}</p>
    </div>
  );
}
```

O **useMemo** só recalcula o valor do dobro (**number * 2**) quando o número muda.

Evitamos o recálculo desnecessário em renderizações que não alteram o número.

useCallback

useCallback é um hook usado para memorizar funções. Ele evita que funções sejam recriadas em cada renderização, o que pode melhorar a performance.

NeXT JavaScript

```
import React, { useState, useCallback } from "react";

function AlertButton() {
  const [text, setText] = useState("");

  // Memoriza a função para que ela não seja recriada em
cada renderização
  const showAlert = useCallback(() => {
    alert(`Texto: ${text}`);
  }, [text]); // Atualiza apenas se 'text' mudar

  return (
    <div>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <button onClick={showAlert}>Mostrar Alerta</button>
    </div>
  );
}
```

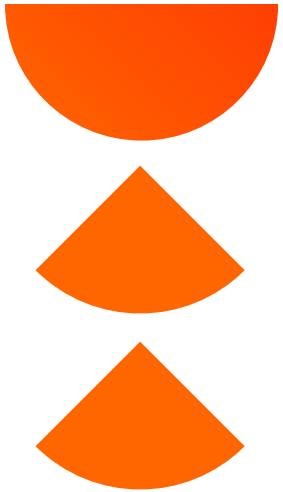
O **showAlert** é uma função que só será recriada se o **text** mudar.

Isso é útil quando o botão (ou outro componente) depende de funções que não precisam ser recalculadas sempre.

INTERVALO!

15 min.

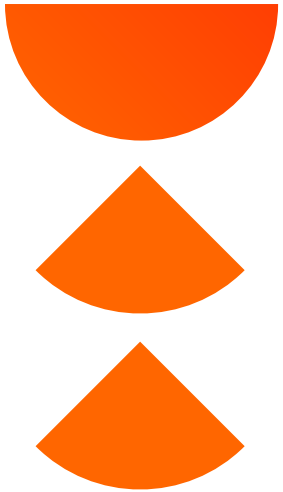
Custom Hooks



O que são?

Custom Hooks são funções personalizadas que permitem reutilizar lógica de estado ou efeitos em componentes funcionais do React. Eles ajudam a organizar e compartilhar lógica entre diferentes partes da aplicação sem precisar duplicar código.

- Por que usar Custom Hooks?
  - Reutilização de lógica: Encapsular lógica comum em uma função que pode ser usada em vários componentes.
  - Melhor organização: Separar lógica específica do componente, tornando o código mais limpo e fácil de manter.
  - Legibilidade: Melhorar a compreensão da lógica de negócios, especialmente em componentes grandes.



Como criar um?

Um Custom Hook nada mais é do que uma função JavaScript que começa com o prefixo use.

Ele pode usar outros Hooks como useState, useEffect, etc., e retornar valores ou funções que podem ser usados em componentes.

São criados geralmente para um contexto específico da aplicação que está sendo desenvolvida.

Exemplo: Hook para Gerenciar Título da Página

NeXT JavaScript

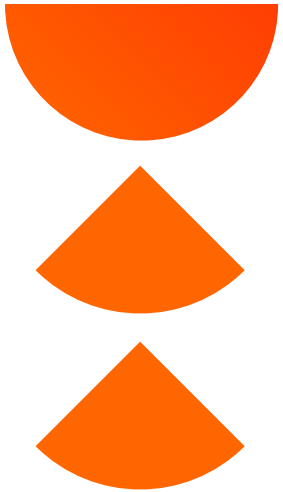
```javascript
import { useEffect } from "react";

export function useDocumentTitle(title) {
  useEffect(() => {
    document.title = title;
  }, [title]);
}
```

- Como criar

NeXT JavaScript

```javascript
import { useDocumentTitle } from "./useDocumentTitle";

function App() {
  useDocumentTitle("Minha Aplicação");

  return <h1>Bem-vindo!</h1>;
}
```

- Como usar



Boas práticas

1. Nomeação: Sempre comece o nome do Hook com use para seguir o padrão do React.
2. Reutilizável: Foque em criar Hooks genéricos que possam ser usados em diferentes cenários.
3. Isolamento de lógica: Mantenha a lógica focada e evite misturar responsabilidades.

DÚVIDAS?! :)