

<LAB365>

Hooks

AGENDA | M1S09 - A1

- O que são Hooks?
- useState
- Efeitos colaterais

HOOKS

Os Hooks revolucionaram o React ao permitir o uso de estado e outros recursos em componentes funcionais.

Hooks são **funções** que permitem "conectar" funcionalidades do React (como estado e ciclo de vida) a componentes funcionais. Eles foram introduzidos no React 16.8 para **simplificar** o desenvolvimento e **reduzir a complexidade** de componentes de classe.

Por que usar Hooks?

- Simplificação: Eliminam a necessidade de classes
- Reutilização: Facilitam a criação de lógica compartilhada
- Organização: Agrupam código relacionado logicamente

HOOKS

Principais Hooks e Seus Usos

- **useState:** Gerenciamento de Estado: Gerencia estado local em componentes funcionais.
- **useEffect:** Efeitos Colaterais: Executa código em diferentes fases do ciclo de vida.
- **useContext:** Acesso ao Contexto: Acessa valores de um Context sem precisar de Consumer.
- **useRef:** Referências Mutáveis: Cria referências que persistem entre renders.
- **useReducer:** Estado Complexo: Alternativa ao useState para lógica de estado mais complexa.



USESTATE

O useState é um Hook fundamental que permite **adicionar estado local a componentes funcionais** no React. Antes dos Hooks, apenas componentes de classe podiam ter estado.

Como funciona?

- Recebe um valor inicial como argumento
- Retorna um array com duas posições:
 - O valor atual do estado
 - Uma função para atualizar esse estado

A primeira utilização desse hook, já podemos ver no nosso exemplo na criação do projeto, mas vamos criar nosso próprio código para ver a utilização do mesmo.

USESTATE

Para utilizar o useState, temos de inicialmente fazer a importação do mesmo: `import { useState } from 'react';`

Na sequência, dentro do nosso componente temos de declarar o mesmo e crias as funções de incremento e assim podemos utilizar o mesmo quando tivermos necessidade.

```
import { useState } from 'react';
```

```
const [count, setCount] = useState(0);
```

USESTATE

```
import { useState } from 'react';

function Contagem() {
  // Declaração do estado: count é o valor atual,
  // setCount é a função de atualização
  const [count, setCount] = useState(0);

  // Função de incremento
  const increment = () => {
    setCount(count + 1); // Atualização direta
  };

  // Função de incremento com valor anterior (forma mais segura)
  const safeIncrement = () => {
    setCount(prevCount => prevCount + 1); // Usando função de atualização
  };
}
```

USESTATE

```
return (  
  <div>  
    <h2>Contador: {count}</h2>  
    <button onClick={increment}>Incrementar (+1)</button>  
    <button onClick={safeIncrement}>Incrementar com segurança (+1)</button>  
    <button onClick={() => setCount(0)}>Resetar</button>  
  </div>  
);  
}  
  
export default Contagem
```


USESTATE

Tenho de trabalhar com somente um único useState por componente?

Não, nós podemos ter quantos useState precisar em um único componente React.

Não há limite para o número de useState que podemos declarar, e essa é uma das vantagens dos Hooks: você pode dividir seu estado em múltiplas variáveis para melhor organização.

Por que usar múltiplos useState?

- Separação lógica: Cada useState pode gerenciar um aspecto diferente do estado.
- Facilidade de manutenção: Fica mais claro qual parte do estado está sendo atualizada.
- Evita objetos complexos: Em vez de um único estado com vários campos, você pode ter estados independentes.

USESTATE

```
import { useState } from 'react';
import type { FormEvent } from 'react'; //só para visualização do useState

function FormularioAceite() {
  // Estado para o nome (string)
  const [nome, setNome] = useState('');

  // Estado para a idade (number)
  const [idade, setIdade] = useState(18);

  // Estado para aceitar termos (boolean)
  const [aceite, setAceite] = useState(false);

  //só para visualização do useState
  const verUseState = (event : FormEvent) => {
    event.preventDefault()
    alert(`Estado atual:\n\nNome: ${nome}\nIdade: ${idade}\n
    Aceitou termos: ${aceite ? 'Sim' : 'Não'}`);
  };
}
```

USESTATE

```
return (  
  <form>  
    <input  
      type="text"  
      value={nome}  
      onChange={(e) => setNome(e.target.value)}  
      placeholder="Nome"  
    />  
  
    <input  
      type="number"  
      value={idade}  
      onChange={(e) => setIdade(Number(e.target.value))}  
      placeholder="Idade"  
    />  
  )
```

USESTATE

```
    <label>
      <input
        type="checkbox"
        checked={aceite}
        onChange={(e) => setAceite(e.target.checked)}
      />
      Aceito os termos
    </label>

    <button onClick={verUseState}> Mostrar UseState no Console</button>
  </form>
);
}

export default FormularioAceite;
```

USESTATE

Podemos também usar um único useState com objeto, isso é indicado se os dados estão fortemente relacionados.

```
const [dadosAceite, setDadosAceite] = useState(  
  {  
    nome: '',  
    idade: 18,  
    aceite: false  
  });
```

```
<input  
  type="number"  
  value={dadosAceite.idade}  
  onChange={(e) => setDadosAceite( {...dadosAceite, idade: parseInt(e.target.value)} )}  
  placeholder="Idade"  
>
```

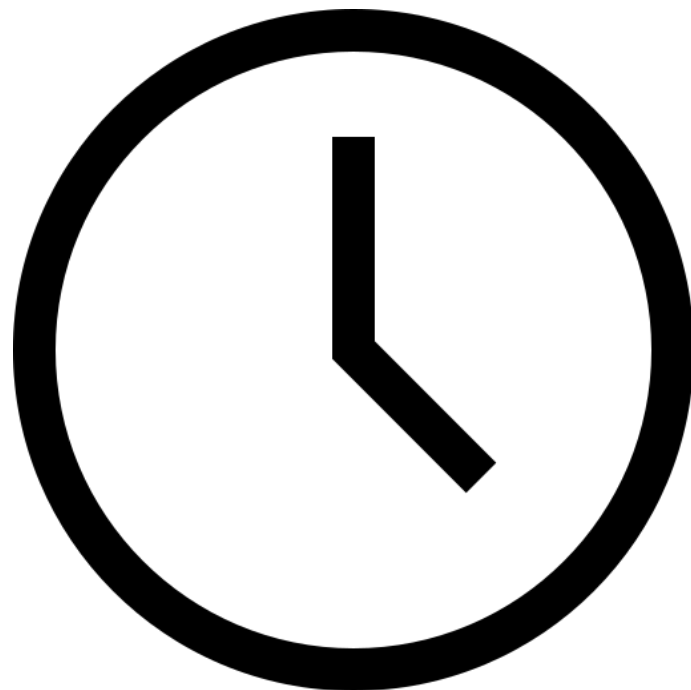
INTERVALO!

Finalizamos o nosso primeiro período de hoje. Que tal descansar um pouco?!

Nos vemos em 20 minutos.

Início: 20:20

Retorno: 20:40



EFEITOS COLATERAIS

O Que São Efeitos Colaterais?

Em programação, um efeito colateral ocorre quando uma função:

- Interage com sistemas externos além do seu escopo local
- Modifica algo fora do ambiente da função
- Produz resultados observáveis além do seu retorno

Pense em um componente React como uma máquina de sucos:

- Entrada (props/estado) → Frutas e água
- Processamento → Extrair o suco (renderização)
- Efeito colateral → Ligar para o fornecedor quando faltam laranjas (ação externa)



EFEITOS COLATERAIS

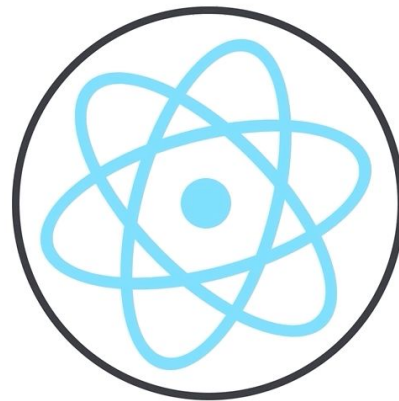
Efeitos Colaterais Comuns no React

- Interação com APIs/Bancos de Dados
- Manipulação Direta do DOM
- Subscriptions/Event Listeners
- Timers e Intervalos

E aqui chegamos a questão de utilizar o `useEffect`.

O React exige que efeitos colaterais ocorram:

- Fora do fluxo principal de renderização
- No momento correto do ciclo de vida do componente
- Com controle sobre execução e limpeza



Fluxo Ideal

- Renderização: Componente calcula o que deve aparecer na tela
- Commit: React atualiza o DOM
- Efeitos: Executa ações secundárias após a renderização

TREINANDO NOSSAS HABILIDADES!

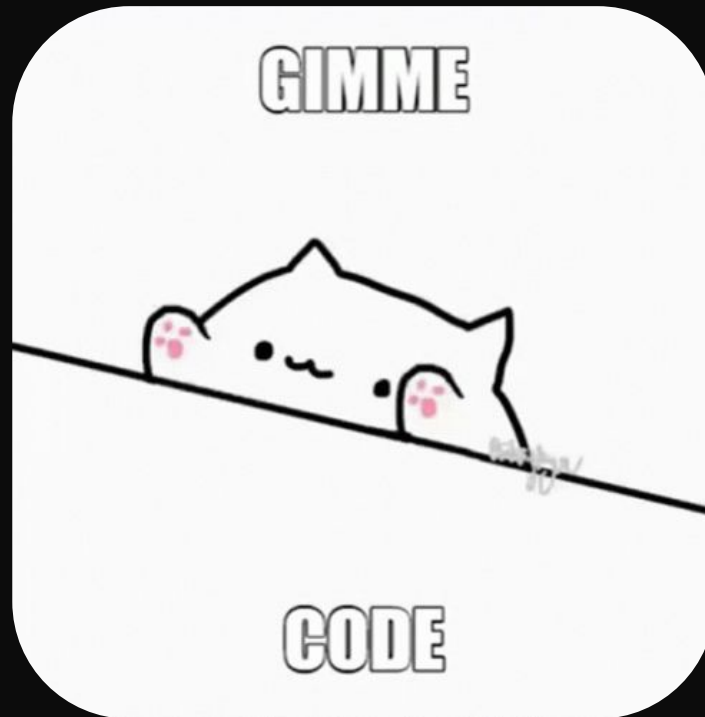
Fomos incumbidos de desenvolver uma lista de tarefas.

Nesse componente podemos adicionar uma nova tarefa por um campo textual.

A lista de tarefa deve ser exibida abaixo do campo.

Precisaremos de um contador, para saber quantas tarefas temos na lista.

As tarefas serão armazenadas em nosso `useState`!



VAMOS CODAR!



<LAB365>

<LAB365>

SENAI

<LAB365>

USEEFFECT

<LAB365>

SENAI

AGENDA | M1S09 - A2

- useEffect
- Exercício em Squads

USEEFFECT

O useEffect é um dos Hooks mais importantes e versáteis do React, usado para **gerenciar efeitos colaterais** em componentes funcionais. Vamos explorar seu funcionamento de forma completa, com exemplos práticos e dicas profissionais.

Fluxo de Execução

- Renderização: Componente é renderizado
- Commit: React atualiza o DOM
- Efeitos:
 - Executa a função principal
 - Se houver, retorna uma função de limpeza (que roda antes do próximo efeito ou na desmontagem)

```
useEffect(() => {  
  // Código do efeito (executado após a renderização)  
  
  return () => {  
    // Função de limpeza (opcional)  
  };  
}, [dependencias]); // Array de dependências
```

USEEFFECT

```
useEffect(() => {  
  console.log('Componente montado!');  
  fetch('/api/data').then(response => response.json());  
}, []); // Array vazio = executa apenas na montagem
```

```
const [user, setUser] = useState({name: ''});  
  
useEffect(() => {  
  if (user) {  
    document.title = `Perfil de ${user.name}`;  
  }  
}, [user]); // Executa sempre que 'user' mudar
```

USEEFFECT

```
useEffect(() => {  
  const timer = setTimeout(() => alert('Tempo esgotado!'), 3000);  
  
  return () => clearTimeout(timer); // Limpa o timer se o componente desmontar  
}, []);
```

```
useEffect(() => {  
  // Só executa se 'count' for maior que 10  
  if (count > 10) {  
    alert('Contagem alta!');  
  }  
}, [count]);
```


**VAMOS VER UM EXEMPLO
BUSCANDO NOSSOS DADOS
EM UMA API EXTERNA?**

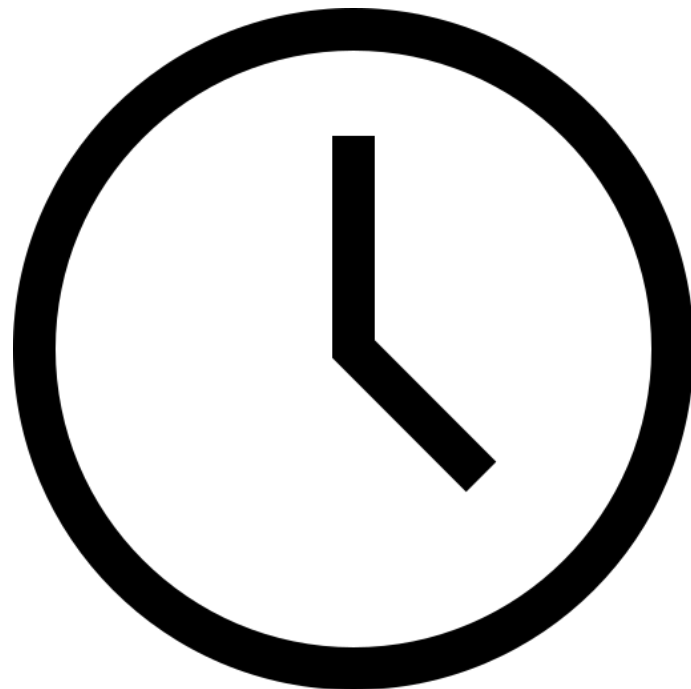
INTERVALO!

Finalizamos o nosso primeiro período de hoje. Que tal descansar um pouco?!

Nos vemos em 20 minutos.

Início: 20:40

Retorno: 21:00



TREINANDO NOSSAS HABILIDADES!

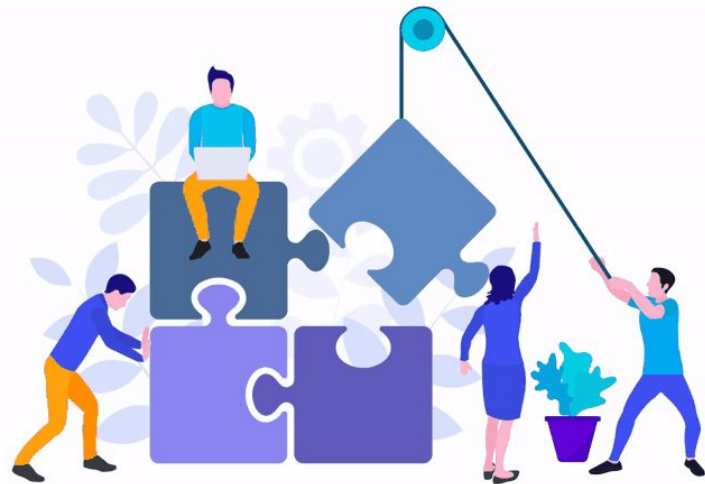
Agora vamos construir um componente de cronômetro!

Esse exercício será feito em squads para o debate de como fazer o desenvolvimento do mesmo.

Precisaremos:

- Botões de iniciar, pausar e zerar
- Mostre o tempo decorrido em segundos
- Atualizar o título da página na passagem de segundos.

Utilize os seus conhecimentos adquiridos em front-end até o momento.



<LAB365>

<LAB365>

**COMPONENTES E
HOOKS CUSTOMIZADOS**

<LAB365>

SENAI

AGENDA | M1S09 - A3

- Componentes controlados
- Componentes não controlados
- Hooks customizados
- Revisão

COMPONENTES CONTROLADOS E NÃO CONTROLADOS

No React, existem duas abordagens principais para **gerenciar formulários e inputs**: componentes controlados e não controlados.

Componentes controlados são componentes onde o **React controla o estado** do elemento de formulário através de estados (useState).

Já os não controlados são casos de componentes onde o **DOM mantém o estado**, e o React acessa o valor via ref.

COMPONENTES CONTROLADOS

Características

- Fonte da verdade: O valor é armazenado no estado do React
- Atualizações: Manipuladas via onChange
- Renderização: O componente é rerenderizado a cada mudança

Vantagens

- Controle total sobre o valor
- Validação em tempo real
- Integração fácil com outros estados
- Melhor para formulários complexos

Desvantagens

- Mais rerenders (impacto em performance para muitos inputs)
- Boilerplate de código maior

Casos de Uso Ideais

- Formulários com validação complexa
- Quando você precisa:
 - Formatação em tempo real (ex: máscaras)
 - Feedback instantâneo (ex: busca live)
 - Controlar múltiplos inputs interconectados

COMPONENTES NÃO CONTROLADOS

Características

- Fonte da verdade: O valor fica no DOM
- Atualizações: Acessadas via `ref.current.value`
- Renderização: Não causa rerenders desnecessários

Vantagens

- Melhor performance (menos rerenders)
- Código mais simples para casos básicos
- Ideal para integração com bibliotecas não-React

Desvantagens

- Menos controle sobre o valor
- Validação só no submit
- Dificuldade com valores derivados

Casos de Uso Ideais

- Formulários com validação simples
- Quando você precisa:
 - Formulários simples
 - Quando performance é crítica
 - Para inputs especiais

COMPONENTES CONTROLADOS VS COMPONENTES NÃO CONTROLADOS

```
import { useState } from "react";
import type { ChangeEvent } from "react";

function Controlados(){
  const [value, setValue] = useState('');

  const handleChange = (e: ChangeEvent<HTMLInputElement>) => {
    setValue(e.target.value); // Atualiza o estado
  };

  return(
    <input
      type="text"
      value={value}
      onChange={handleChange}
    />
  );
}
```

COMPONENTES CONTROLADOS VS COMPONENTES NÃO CONTROLADOS

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

function Controlados() {
  const inputRef = useRef<HTMLInputElement>(null); // Ref tipada

  const handleClick = () => {
    alert(`Valor digitado: ${inputRef.current?.value}`); // Acessa valor via ref
  };

  return (
    <div>
      <input
        type="text"
        ref={inputRef}
        placeholder="Digite algo (não controlado)"
      />
      <button onClick={handleClick}>Mostrar valor</button>
    </div>
  );
}
```

TIPAGENS AVANÇADAS

Vamos falar um pouco sobre algumas tipagens que vimos nos últimos exemplos.

ChangeEvent<HTMLInputElement>

- É a tipagem para eventos de mudança em elementos de formulário, autocompletar e verificação de tipos.
- ChangeEvent: Tipo genérico para eventos de mudança
- <HTMLInputElement>: Especifica o tipo de elemento que disparou o evento

Devemos usar sempre que precisar tipar o parâmetro e em um onChange.

Elemento

Tipo do Evento

```
<input type="text">
```

```
ChangeEvent<HTMLInputElement>
```

```
<textarea>
```

```
ChangeEvent<HTMLTextAreaElement>
```

```
<select>
```

```
ChangeEvent<HTMLSelectElement>
```

```
<input type="file">
```

```
ChangeEvent<HTMLInputElement>
```

```
<input type="checkbox">
```

```
ChangeEvent<HTMLInputElement>
```

TIPAGENS AVANÇADAS

useRef<HTMLInputElement>(null)

- É a tipagem para referências de elementos DOM, permitindo acesso seguro às propriedades nativas.
- HTMLInputElement: Interface TypeScript para elementos <input>
- null: Valor inicial obrigatório para refs de DOM

Elemento	Tipo da Ref	Propriedades Acessíveis
<input>	HTMLInputElement	value, focus(), checked
<div>	HTMLDivElement	innerText, clientHeight
<button>	HTMLButtonElement	disabled, click()
<video>	HTMLVideoElement	play(), pause()
<form>	HTMLFormElement	submit(), reset()

TIPAGENS AVANÇADAS

Evento	Tipo TypeScript	Uso Típico
<code>onClick</code>	<code>MouseEvent<HTMLButtonElement></code>	Cliques em botões
<code>onSubmit</code>	<code>FormEvent<HTMLFormElement></code>	Submissão de formulários
<code>onKeyDown</code>	<code>KeyboardEvent<HTMLInputElement></code>	Teclas pressionadas
<code>onChange</code>	<code>ChangeEvent<HTMLInputElement></code>	Mudanças em inputs
<code>onFocus</code>	<code>FocusEvent<HTMLInputElement></code>	Foco em elementos

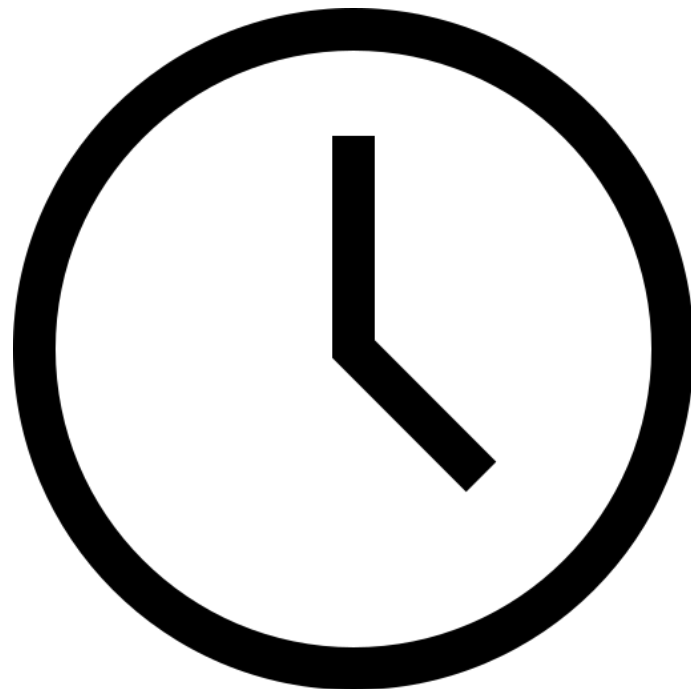
INTERVALO!

Finalizamos o nosso primeiro período de hoje. Que tal descansar um pouco?!

Nos vemos em 20 minutos.

Início: 20:20

Retorno: 20:40



HOOKS CUSTOMIZADOS

Hooks customizados são funções JavaScript/TypeScript que **encapsulam lógica reutilizável** usando **outros hooks nativos** (como useState, useEffect, etc.). Eles seguem a convenção de nomes começando com use (ex: useFetch, useLocalStorage).

Por que usar?

- Reutilização de lógica entre componentes
- Organização do código (separar lógica complexa)
- Facilidade de teste (lógica isolada)
- Compartilhamento entre projetos

Regras Básicas

- Sempre comece com use (ex: useTimer)
- Pode chamar outros hooks
- Deve seguir as Regras dos Hooks: <https://pt-br.legacy.reactjs.org/docs/hooks-rules.html>

HOOKS CUSTOMIZADOS

```
import { useState, useEffect } from 'react';

function useLocalStorage<T>(key: string, initialValue: T) {
  // 1. Recupera do localStorage ou usa valor inicial
  const [storedValue, setStoredValue] = useState<T>(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.error(error);
      return initialValue;
    }
  });

  // 2. Atualiza localStorage quando o estado muda
  useEffect(() => {
    try {
      window.localStorage.setItem(key, JSON.stringify(storedValue));
    } catch (error) {
      console.error(error);
    }
  }, [key, storedValue]);
}
```

```
    return [storedValue, setStoredValue] as const;
  }

  export default useLocalStorage;
```

HOOKS CUSTOMIZADOS



```
import useLocalStorage from "../hooks/useLocalStorage";

function Component() {
  const [name, setName] = useLocalStorage('username', 'Anônimo');

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

export default Component
```

HOOKS CUSTOMIZADOS: BOAS PRÁTICAS

	
Nomeie hooks com use para seguir convenções	Não use hooks condicionalmente
Teste hooks isoladamente	Não crie hooks para lógica muito específica
Compartilhe via pacotes npm (se útil para outros)	Não esqueça de limpar efeitos (ex: clearInterval)

REVISÃO

O que já estamos craques!!!

- HTML
 - Tags
 - Estruturação
- CSS
 - Seletores
 - Referências
- JS
 - Lógica de Programação
 - Manipulação de DOM
 - Arrays
 - POO

REVISÃO

O que estamos aprendendo no React!

- Criação de Projetos
- Componentes Funcionais
- Props
- Tipagem com TypeScript
- Hooks
 - useState
 - useEffect
- Componentes controlados e não controlados
- Hooks Customizados

VAMOS CODAR!!!



AVALIAÇÃO DOCENTE

O que você está achando das minhas aulas neste conteúdo?

Clique [aqui](#) ou escaneie o QRCode ao lado para avaliar minha aula.

Sinta-se à vontade para fornecer uma avaliação sempre que achar necessário.



<LAB365>

<LAB365>

SENAI