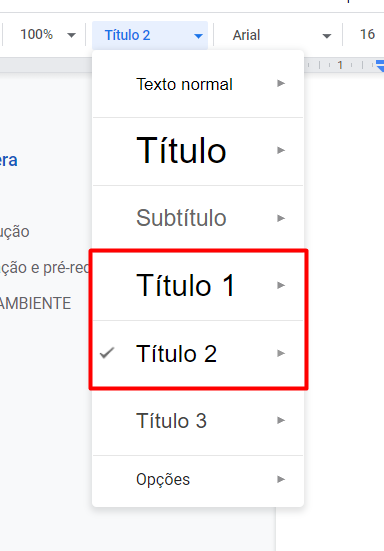
Olá! Este é o documento onde você vai escrever as atividades do curso. Seguindo as regras abaixo, garantimos que o upload dos exercícios e materiais seja feito corretamente, da maneira que nós planejamos!

Materiais para consulta

* [Guia para criação de exercícios](https://drive.google.com/file/d/12dN3KXAdtW0ZEzV7ysJmKaA6HbghFyxc/view?usp=sharing)
* [Exemplo de documento completo de atividades](https://docs.google.com/document/d/1BmRybf2ejo0G0C0JuPzl-hjFlZ0RJ2EURDb6OvrgKhE/edit?usp=sharing)

Títulos e subtítulos

Para que os conteúdos sejam posicionados corretamente, indicamos o início da aula como **Título 1**, e os nomes das atividades (vídeos e exercícios), como **Título 2**.



Além de posicionar as aulas e atividades como indicado acima, o markdown também pode ser utilizado para destacar os títulos, com # (H1) para as aulas e ## (H2) para as atividades:

Organização

Ordene as atividades e marque também onde estão os vídeos de cada aula, assim cada exercício vai para seu devido lugar, sem a necessidade de editarmos na plataforma. Veja no print abaixo como essa organização aparecerá na plataforma:

Indique sempre o tipo de atividade, lembrando que a plataforma aceita os seguintes tipos:   
   
- Vídeo   
- Atividade de única escolha   
- Atividade de múltipla escolha   
- Atividade sem resposta do aluno   
- Explicação

Atividades de alternativa

Neste tipo de atividade, as respostas devem estar organizadas da seguinte forma:   
   
a) Texto da alternativa

**Justificativa**   
**Alternativa correta!** Texto da justificativa

b) Texto da alternativa

**Justificativa**

Texto da justificativa   
   
As justificativas das alternativas corretas precisam começar com a expressão “**Alternativa correta!”**, assim elas já serão identificadas e marcadas corretamente na plataforma.

Arquivos e links de projeto/stages

Os links dos arquivos de projeto e stages precisam estar corretos e compartilhados via **GitHub** com o usuário **romulohenrique** (Se tiver alguma dúvida a respeito dos arquivos, consulte a revisão didática e técnica!)

Imagens

As imagens são recursos importantes nas atividades, ao usá-las, lembre-se de sempre nomeá-las conforme a ordem em que aparecem, indicando a aula correspondente, como no exemplo abaixo:

![descrição da imagem](Aula2-img4.png)

No caso, os prints das imagens não são necessários, apenas o markdown, como no exemplo acima, contendo descrição (alt, ou [texto alternativo](https://cta.ifrs.edu.br/boas-praticas-para-descricao-de-imagens/)), importante tanto para acessibilidade quanto SEO. Os arquivos com as imagens precisam estar em uma pasta dentro da pasta do curso chamada **ID-imgs-para-atividades**.

Markdown

Para embutir links, usar negrito, itálico ou mostrar um trecho de código, ao invés de usar os recursos do Google Docs, utilize a linguagem markdown. Alguns exemplos de uso:

* \*Itálico\* => *Itálico*
* \*\*Negrito\*\* => **Negrito**
* \*\*\*Itálico e Negrito\*\*\* => ***Itálico e Negrito***

Em alguns momentos se fará necessário o uso de citações utilizando o **>** para dar destaque a um parágrafo, como no caso de dicas, observações, citações propriamente ditas, textos de briefings e afins, como no exemplo a seguir:

**>** Atenção:evite criar slides ou utilizar imagens. Na review é preferível mostrar o software funcionando.

O texto será destacado com um fundo cinza na plataforma:

Para links, a formatação em markdown é **[texto que ficará com o link](link)**, como em:

Este é o link da [Alura]([https://www.alura.com.br](https://www.alura.com.br/))

# Aula 1 – Estados e efeitos

## **Preparando o ambiente: comece por aqui**

## Olá, dev!

Boas-vindas ao curso de React 18: hooks essenciais! \*\***Este novo formato de curso foi pensado e organizado para que você consiga aprofundar seus conhecimentos teóricos e técnicos\*\*** nos hooks essenciais do React. Então você poderá se aprofundar em hooks como useState, useEffect, useContext, useRef e muito mais.

E não só isso, você ainda vai poder pôr a mão na massa e praticar por si mesmo em um projeto incrível que testará os seus conhecimentos. Em cada aula você terá uma missão que é aplicar os conhecimentos vistos na aula no projeto. Então sua tarefa será usar o que aprendeu sobre os hooks e toda a bagagem de JavaScript que você tem, para adicionar comportamentos e funcionalidades na aplicação. Mas não se preocupe, você não estará sozinho nessa. Deixarei o gabarito de cada exercício para que você compare suas respostas, tire suas dúvidas e aprenda muito mais!

Então antes de começarmos, vamos preparar o ambiente:

## ## 1. Ferramentas

Para prosseguirmos, usaremos o editor de código [Visual Studio Code](<https://code.visualstudio.com/download>), que pode ser baixado para Windows, Linux e Mac. Além disso, você pode usar o navegador que quiser, mas recomendo usar o [Chrome](<https://www.google.com/intl/en_us/chrome/>) como navegador principal; é importante tê-lo instalado em seu computador.

### ## 2. Download do projeto

O projeto que você irá praticar e desenvolver [está no GitHub](https://github.com/NeiltonSeguins/4487-react-hooks). Você deve baixá-lo na sua máquina, mas se preferir, [baixe o arquivo zip do projeto](https://github.com/NeiltonSeguins/4487-react-hooks/archive/refs/heads/projeto-base.zip).

Indico que você rode seu projeto na seguinte versão:

- React:18.3.1.

Isso garantirá que estaremos na mesma página versões quando você conferir o gabarito com as soluções das atividades propostas.

### ## 3. Figma

Você receberá um projeto base, com toda a interface do usuário desenvolvida, estilos e componentes. Mas isso não impede de você se desafiar um pouco mais e tentar fazer ele do zero, não é mesmo?

Então, caso queira fazer isso, você pode conferir o layout do projeto, e se sinta desafiado a criá-lo do zero, [acesse o figma](https://www.figma.com/community/file/1468989433664551328/react-hooks-e-context-api-poupapp). Vá com tudo!

Caso tenha dúvidas na instalação ou ao longo do curso, fique à vontade para se juntar ao fórum!

Vamos começar?

## ✅ Vídeo 1.1 - Apresentação

## ✅ Vídeo 1.2 - Por que usamos hooks?

## ✅ Atividade de alternativa: Otimizando a gestão de preferências pessoais (Feito com IA)

## Mão na massa:Implementando a adição de tarefas com useState

Nesta aula, aprendemos como utilizar o Hook `useState` para gerenciar o estado de componentes no React. Como prometido, esta atividade é para que você coloque em prática os conhecimentos adquiridos na aula teórica.

Agora é sua vez de praticar! Sua missão é a seguinte:

1. **Gerencie o estado das tarefas e do formulário:**
   1. Utilize o hook `useState` para controlar a lista de tarefas e a visibilidade do formulário de adição.
2. **Implemente a função de adicionar tarefas:**
   1. Crie uma função que adicione uma nova tarefa à lista, garantindo que tarefas vazias não sejam adicionadas.
3. **Controle a exibição do formulário:**
   1. Configure a interface para exibir ou ocultar o formulário de adição de tarefas com base no estado.

Tente fazer por si mesmo antes de olhar o gabarito. Abaixo está o passo a passo detalhado de como você pode realizar a atividade!

**Opinião da pessoa instrutora:**

Você conseguiu fazer? Eu espero que sim! Se não, eu vou te dar “uma colher de chá” e te mostrar como eu faria. Se você fez e sua solução está diferente da minha não tem problema, você pode comparar as duas soluções e tirar aprendizados importantes. Vamos lá!

## **## Criando um estado para tarefas**

No componente principal `AppTarefas`, importe o hook useState e defina dois estados: um para a lista de tarefas e outro para controlar a visibilidade do formulário.

```jsx

import { useState } from "react";  
  
const AppTarefas = () => {  
 const [tarefas, setTarefas] = useState([]);  
 const [exibirFormulario, setExibirFormulario] = useState(false);  
  
 // ...  
};

```

## **## Adicionando tarefas**

Dentro do mesmo componente, crie a função `adicionarTarefa` que recebe o texto da nova tarefa. Verifique se o texto não está vazio antes de adicionar a tarefa à lista.

```jsx

function adicionarTarefa(texto) {  
 if (!texto.trim()) return;  
  
 const novaTarefa = {  
 id: Date.now(),  
 texto,  
 concluida: false,  
 };  
  
 setTarefas([...tarefas, novaTarefa]);  
 setExibirFormulario(false);  
}

```

## **## Controlando a exibição do formulário:**

Utilize o estado `exibirFormulario` para determinar quando o formulário deve ser exibido. Adicione um botão que altera esse estado para mostrar ou ocultar o formulário. Você também precisa passar via props a função de `adicionarTarefa` e `setExibirFormulario` para o componente de Formulário. Não esqueça de passar as tarefas para o componente de ListaTarefas também.

```jsx

return (  
 <div>

{/\*...\*/}  
 <ListaTarefas tarefas={tarefas} />  
 {exibirFormulario && (  
 <FormularioTarefas  
 aoSalvar={adicionarTarefa}  
 aoCancelar={() => setExibirFormulario(false)}  
 />  
 )}  
 <Botao  
 onClick={() => setExibirFormulario(true)}  
 icone={add\_circleImg}  
 className={styles["tasks\_\_button--add-task"]}  
 >  
 Adicionar nova tarefa  
 </Botao>  
 {/\* ... \*/}  
 </div>  
);

```

O componente AppTarefas terá essa cara:

```jsx

import styles from "./styles.module.css";  
import add\_circleImg from "/src/assets/imgs/add\_circle.png";  
import Botao from "./Botao";  
import CabecalhoTarefas from "./CabecalhoTarefas";  
import FormularioTarefas from "./FormularioTarefas";  
import ListaTarefas from "./ListaTarefas";  
import { useState } from "react";  
  
const AppTarefas = () => {  
 const [tarefas, setTarefas] = useState([]);  
 const [exibirFormulario, setExibirFormulario] = useState(false);  
  
 function adicionarTarefa(texto) {  
 if (!texto.trim()) return;  
  
 const novaTarefa = {  
 id: Date.now(),  
 texto,  
 concluida: false,  
 };  
  
 setTarefas([...tarefas, novaTarefa]);  
 setExibirFormulario(false);  
 }  
  
 return (  
 <section className={styles["tasks"]}>  
 <div className={styles["tasks\_\_task-content"]}>  
 <CabecalhoTarefas />  
 <ListaTarefas tarefas={tarefas} />  
 {exibirFormulario && (  
 <FormularioTarefas  
 estaVisivel={exibirFormulario}  
 aoSalvar={adicionarTarefa}  
 aoCancelar={() => {  
 setExibirFormulario(false);  
 }}  
 />  
 )}  
 <Botao  
 onClick={() => setExibirFormulario(true)}  
 icone={add\_circleImg}  
 className={styles["tasks\_\_button--add-task"]}  
 >  
 Adicionar nova tarefa  
 </Botao>  
 </div>  
 </section>  
 );  
};  
  
export default AppTarefas;

```

## **## Passando estados via props**

Como você passou via props as tarefas para o componente de ListaTarefas, você precisa ajustar seu componente também:

```jsx

import Botao from "../Botao";  
import styles from "../styles.module.css";  
import checkImg from "/src/assets/imgs/check-task.png";  
import trashImg from "/src/assets/imgs/trash.svg";  
  
const ListaTarefas = ({ tarefas }) => {  
 return (  
 <ul className={styles["tasks\_\_task-list"]}>  
 {tarefas.map((tarefa) => (  
 <li  
 key={tarefa.id}  
 className={`${styles["tasks\_\_task-list-item"]} ${tarefa.concluida ? styles["tasks\_\_task-list-item-complete"] : ""}`}  
 >  
 <Botao  
 icone={checkImg}  
 id="marcar-tarefa-como-concluida"  
 className={`${styles["check-task\_\_button"]} ${tarefa.concluida ? styles["check-task\_\_button--complete"] : ""}`}  
 />  
 <p className={styles["tasks\_\_task-list-item-description"]}>{tarefa.texto}</p>  
 <Botao  
 icone={trashImg}  
 id="deletar-tarefa"  
 className={`${styles["delete-task\_\_button"]}`}  
 />  
 </li>  
 ))}  
 </ul>  
 );  
};  
  
export default ListaTarefas;

```

A mesma coisa você deve fazer para o FormularioTarefas:

```jsx

import { useState } from "react";  
import CampoTexto from "../CampoTexto";  
import RodapeTarefas from "../RodapeTarefas";  
import styles from "../styles.module.css";  
  
const FormularioTarefas = ({ estaVisivel, aoSalvar, aoCancelar }) => {  
 const [texto, setTexto] = useState("");  
  
 return (  
 <form  
 className={`${styles["form-add-task"]} ${!estaVisivel ? styles["form-add-task--hidden"] : ""}`}  
 aria-hidden={!estaVisivel}  
 >  
 <CampoTexto  
 value={texto}  
 onChange={(e) => setTexto(e.target.value)}  
 etiqueta="Adicionando tarefa"  
 placeholder="No que você está trabalhando?"  
 />  
 <RodapeTarefas aoSalvar={() => aoSalvar(texto)} aoCancelar={aoCancelar} />  
 </form>  
 );  
};  
  
export default FormularioTarefas;

```

Neste componente, precisamos de um outro estado para salvar o que o usuário digita. Chamamos este estado de “texto” e ele é alterado sempre que o usuário digita no campo textarea para adicionar uma nova tarefa.

E com isso você já pode testar sua aplicação e ser capaz de adicionar uma tarefa e visualizá-la em tela. Faça o teste. Se você quiser conferir o código com todas as alterações que fizemos, só [clicar aqui neste link](https://github.com/NeiltonSeguins/4487-react-hooks/commit/3d4b4d3740da2d7617f341c71c590d07a442431d).

Grande abraço!

## ✅ Vídeo 1.3 - Adicionando efeitos colaterais

## Para saber mais: React e o modo estrito

Sabe quando você está debugando seu código React e coloca um console.log no componente para ver algum comportamento e nota que no navegador seu console.log foi chamado duas vezes? Isso não é um bug do seu aplicativo nem tampouco do React. Isto é uma forma bem inteligente que o React tem de saber se você aplica boas práticas no seu código, e é chamado de “Modo Estrito”.

O Modo Estrito (*Strict Mode*) do React é uma ferramenta que ajuda as pessoas desenvolvedoras a identificar potenciais problemas em suas aplicações. Ele não altera o comportamento do seu aplicativo, mas adiciona verificações extras e avisa sobre práticas que podem levar a bugs no futuro. Pense nele como um professor que aponta onde você pode melhorar seu código para evitar problemas mais tarde.

## **## Por que usar o modo estrito?**

* **Identificação de problemas**: o Modo Estrito ajuda a identificar componentes que não estão sendo usados corretamente ou que podem causar problemas de desempenho.
* **Melhoria de código**: ele incentiva o uso de práticas recomendadas, ajudando a manter o código melhor possível.

## **## Como ativar o modo estrito?**

Para usar o Modo Estrito, você só precisa envolver a parte do seu aplicativo que deseja verificar com o componente `<React.StrictMode>`. Veja um exemplo:

```jsx

import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App';  
  
ReactDOM.render(  
 <React.StrictMode>  
 <App />  
 </React.StrictMode>,  
 document.getElementById('root')  
);  
```

O melhor disso é que este código já vem por padrão quando você criar uma aplicação React. Se você não quiser o comportamento do modo estrito, só remover este componente.

## **## Impacto no desenvolvimento**

**1. Verificações de ciclo de vida**

O Modo Estrito verifica se você está usando corretamente os métodos de ciclo de vida dos componentes. Isto é, quando seu componente é montado, desmontado ou atualizado. Se você quiser saber mais sobre [o ciclo de vida de efeitos reativos, só clicar aqui](https://react.dev/learn/lifecycle-of-reactive-effects).

**2. Efeitos colaterais**

Efeitos colaterais são mudanças que ocorrem fora do escopo de uma função, como modificar uma variável global ou alterar o DOM diretamente. O Modo Estrito ajuda a identificar e limpar esses efeitos corretamente.

Imagine que você tem um componente que altera o título da página:

```jsx

import React, { useEffect } from 'react';  
  
function MyComponent() {  
 useEffect(() => {  
 document.title = 'Novo Título';  
  
 // Limpeza do efeito colateral  
 return () => {  
 document.title = 'Título Original';  
 };  
 }, []);  
  
 return <div>Meu Componente</div>;  
}

```

No exemplo acima, o useEffect altera o título da página. O Modo Estrito garante que você também forneça uma função de limpeza para restaurar o título original quando o componente for desmontado. Se você não usar uma função de limpeza sabe o que vai acontecer? O seu título original sumirá pra sempre.

Então dito isso, podemos adotar algumas práticas ao usar o useEffect:

**Retorne uma função de limpeza**: quando usar useEffect, retorne uma função que limpe qualquer efeito colateral sempre quando for o caso. Exemplo:

```jsx

useEffect(() => {  
 const timer = setInterval(() => {  
 console.log('Intervalo ativo');  
 }, 1000);  
  
 return () => clearInterval(timer);  
}, []);

```

**Evite dependências omissas**: certifique-se de incluir todas as dependências necessárias no *array de dependências* do useEffect. Isso garante que o efeito seja atualizado corretamente quando as dependências mudarem.

**Teste em modo estrito**: sempre teste seu aplicativo com o Modo Estrito ativado para capturar problemas cedo.

## **## Para mergulhos mais profundos**

Se você deseja se aprofundar mais no Modo Estrito do React, aqui estão alguns tópicos que podem ser interessantes:

1. **Documentação oficial do React**: a documentação do React é um ótimo recurso para [entender melhor o Modo Estrito](https://react.dev/reference/react/StrictMode) e outras funcionalidades.
2. **Hooks do React**: aprender sobre hooks, e entender como eles funcionam é muito importante. Então você pode ler este [artigo](https://www.alura.com.br/artigos/react-hooks) que te ensina de forma rápida o que são, como funcionam e a importância dos hooks.

Bons estudos!

## ✅ Atividade de alternativa: Gerenciamento de efeitos colaterais em atualizações de tarefas (Feito com IA)

## Mão na massa: Adicionando efeitos colaterais no cronômetro

Nesta aula, exploramos o Hook useEffect do React, entendendo seu funcionamento, sua utilidade, a importância de termos uma função de efeito quando necessário e como utilizá-lo com ou sem array de dependências.

Agora é sua vez de praticar e desenvolver a lógica do cronometro, seja a contagem regressiva ou os modos do botão. Você deve implementar essas funcionalidades aplicando principalmente o useEffect e o useState vistos até aqui. Manda ver!

Siga o passo a passo:

1. **Configure o estado do cronômetro:**
   1. Utilize o Hook useState para criar estados que representem o tempo restante do cronômetro e se ele está em execução.
2. **Implemente o efeito para decrementar o tempo:**
   1. Use o Hook useEffect para configurar um intervalo que decremente o tempo a cada segundo quando o cronômetro estiver em execução.
3. **Adicione controles para iniciar e pausar o cronômetro:**
   1. Implemente funcionalidades que permitam iniciar e pausar o cronômetro conforme necessário.

Abaixo está o passo a passo detalhado de como você pode realizar a atividade, com comentários explicativos no código!

**Opinião da pessoa instrutora**

Você conseguiu superar mais este desafio? Sei que é realmente complicado quando parece que você está sozinho, mas estou aqui para compartilhar com você minha resposta. Assim você pode ver em quais pontos deixou passar algo. Ah, e se quiser pode sempre usar o fórum, compartilhando suas dúvidas lá. Ficaremos felizes em te ajudar com quaisquer dúvidas.

Vamos à solução!

## **## Estados do cronômetro**

Dentro do seu componente Cronometro, você precisa definir os estados que permitirão que controlemos não só o tempo em contagem regressiva, mas os modos do pomodoro, os modos de foco, pausa curta e pausa longa. Vamos lá.

```jsx

import styles from "./styles.module.css";

import BotoesModos from "./BotoesModos";

import Timer from "./Timer";

import SwitchMusica from "./SwitchMusica";

import BotaoCronometro from "./BotaoCronometro";

import { useEffect } from "react";

import { useState } from "react";

const modos = [

{

id: "foco",

nome: "Foco",

duracao: 25, // em segundos

},

{

id: "pausa\_curta",

nome: "Pausa curta",

duracao: 5, // em segundos

},

{

id: "pausa\_longa",

nome: "Pausa longa",

duracao: 15, // em segundos

},

];

export default function Cronometro() {

const [modoAtual, setModoAtual] = useState(modos[0]);

const [tempoRestante, setTempoRestante] = useState(modoAtual.duracao);

const [emExecucao, setEmExecucao] = useState(false);

return (

<div className={styles["cronometer"]}>

<BotoesModos />

<Timer />

<SwitchMusica />

<BotaoCronometro />

</div>

);

}

```

Trouxe a constante “modos” para este componente. Ele servirá para definirmos o estado inicial do modoAtual. O modo atual será muito importante para sabermos se estamos no foco, na pausa curta ou pausa longa. Já o estado de tempo restante servirá para nós decrementarmos dependendo da quantidade de minutos.

## **## Decrementando o tempo restante com useEffect**

Agora vamos usar o useEffect para decrementar o tempo restante, bem como para controlar o modoAtual e definir se nosso cronometro está em execução ou não.

```jsx

import styles from "./styles.module.css";

import BotoesModos from "./BotoesModos";

import Timer from "./Timer";

import SwitchMusica from "./SwitchMusica";

import BotaoCronometro from "./BotaoCronometro";

import { useEffect } from "react";

import { useState } from "react";

const modos = [

{

id: "foco",

nome: "Foco",

duracao: 25, // em minutos

},

{

id: "pausa\_curta",

nome: "Pausa curta",

duracao: 5, // em minutos

},

{

id: "pausa\_longa",

nome: "Pausa longa",

duracao: 15, // em minutos

},

];

export default function Cronometro() {

const [modoAtual, setModoAtual] = useState(modos[0]);

const [tempoRestante, setTempoRestante] = useState(modoAtual.duracao);

const [emExecucao, setEmExecucao] = useState(false);

useEffect(() => {

let intervalo;

if (emExecucao && tempoRestante > 0) {

intervalo = setInterval(() => {

setTempoRestante((prev) => prev - 1);

}, 1000);

} else if (tempoRestante === 0) {

setModoAtual((prev) => {

const modoAtualIndex = modos.findIndex((modo) => modo.id === prev.id);

const proximoModo = modos[(modoAtualIndex + 1) % modos.length];

setTempoRestante(proximoModo.duracao);

return proximoModo;

});

setEmExecucao(false);

}

return () => clearInterval(intervalo);

}, [emExecucao, tempoRestante, modoAtual]);

return (

<div className={styles["cronometer"]}>

<BotoesModos />

<Timer />

<SwitchMusica />

<BotaoCronometro />

</div>

);

}

```

Usamos o useEffect para iniciar um intervalo que decrementa o estado tempoRestante a cada segundo quando `emExecucao` é verdadeiro e `tempoRestante` é maior que 0. Quando `tempoRestante` chega a 0, ele avança para o próximo modo na lista de modos, que pode ser o próximo ou o primeiro se chegarmos no último modo na lista de modos. Então a sequência do useEffect define o `tempoRestante` para a duração desse novo modo e pausa o temporizador definindo `emExecucao` como falso.

A função de limpeza dentro do useEffect garante que o intervalo seja limpo quando o componente é desmontado ou quando as dependências (emExecucao, tempoRestante, modoAtual) mudam, evitando vazamentos de memória. Olha aqui a importância da função de limpeza. Mas ainda não acabou, precisamos criar funções para controlar quando o cron}ometro começa e quando ele pausa. Vamos lá!

**## Funções do cronômetro**

```jsx

import styles from "./styles.module.css";

import BotoesModos from "./BotoesModos";

import Timer from "./Timer";

import SwitchMusica from "./SwitchMusica";

import BotaoCronometro from "./BotaoCronometro";

import { useEffect } from "react";

import { useState } from "react";

const modos = [

{

id: "foco",

nome: "Foco",

duracao: 25, // em minutos

},

{

id: "pausa\_curta",

nome: "Pausa curta",

duracao: 5, // em minutos

},

{

id: "pausa\_longa",

nome: "Pausa longa",

duracao: 15, // em minutos

},

];

export default function Cronometro() {

const [modoAtual, setModoAtual] = useState(modos[0]);

const [tempoRestante, setTempoRestante] = useState(modoAtual.duracao);

const [emExecucao, setEmExecucao] = useState(false); // Para saber se o cronômetro está rodando

useEffect(() => {

let intervalo;

if (emExecucao && tempoRestante > 0) {

intervalo = setInterval(() => {

setTempoRestante((prev) => prev - 1);

}, 1000);

} else if (tempoRestante === 0) {

setModoAtual((prev) => {

const modoAtualIndex = modos.findIndex((modo) => modo.id === prev.id);

const proximoModo = modos[(modoAtualIndex + 1) % modos.length];

setTempoRestante(proximoModo.duracao);

return proximoModo;

});

setEmExecucao(false);

}

return () => clearInterval(intervalo);

}, [emExecucao, tempoRestante, modoAtual]);

const alternarCronometro = () => {

setEmExecucao((prev) => !prev);

};

const selecionarModo = (novoModo) => {

setModoAtual(novoModo);

setTempoRestante(novoModo.duracao);

setEmExecucao(false);

};

return (

<div className={styles["cronometer"]}>

<BotoesModos modos={modos} modoAtual={modoAtual} onModoSelecionado={selecionarModo} />

<Timer tempoRestante={tempoRestante} />

<SwitchMusica />

<BotaoCronometro emExecucao={emExecucao} onClick={alternarCronometro} />

</div>

);

}

```

As funções `alternarCronometro` inicia/pausa um cronômetro, e `selecionarModo` muda o modo do cronômetro, definindo sua duração e pausando-o. E como temos componentes que precisam desses estados, vamos passar como props para eles, como em BotoesModo, Timer e BotaoCronometro.

E já que passamos essas props para os componentes, precisamos atualizá-los. Vem comigo.

**## Atualizando os componentes do cronômetro**

Vamos atualizar os componentes dentro do `< Cronometro />`, já que passamos algumas props para eles. Começando pelo `< Timer />`.

```jsx

import styles from "./styles.module.css";  
  
export default function Timer({ tempoRestante }) {  
 const minutos = Math.floor(tempoRestante / 60);  
 const segundos = tempoRestante % 60;  
  
 return (  
 <div className={styles["cronometer-timer"]}>  
 {minutos}:{segundos < 10 ? `0${segundos}` : segundos}  
 </div>  
 );  
}

```

Agora o componente de `< BotoesModo />`

```jsx

import styles from "./styles.module.css";

import BotaoModo from "./BotaoModo";

const BotoesModos = ({ modos, modoAtual, onModoSelecionado }) => {

return (

<ul className={styles["cronometer-modes"]}>

{modos.map((m) => (

<li key={m.id}>

<BotaoModo ativo={m.id === modoAtual.id} onClick={() => onModoSelecionado(m)}>

{m.nome}

</BotaoModo>

</li>

))}

</ul>

);

};

export default BotoesModos;

```

E por fim, no componente de `< BotoesCronometro />`

```jsx

import styles from "./styles.module.css";  
import play\_arrowImg from "/src/assets/imgs/play\_arrow.png";  
import pauseImg from "/src/assets/imgs/pause.png";  
  
const BotaoCronometro = ({ emExecucao, onClick }) => {  
 return (  
 <div className={styles["cronometer\_\_primary-button-wrapper"]}>  
 <button onClick={onClick} className={styles["cronometer\_\_primary-button"]}>  
 <img  
 className={styles["cronometer\_\_primary-button-icon"]}  
 src={emExecucao ? pauseImg : play\_arrowImg}  
 />  
 <span>{emExecucao ? "Pausar" : "Começar"}</span>  
 </button>  
 </div>  
 );  
};  
  
export default BotaoCronometro;

```

Prontinho! Agora você já deve ser capaz de clicar no botão de Começar na aplicação e o cronômetro disparar automaticamente. Faça o teste e me conta como foi.

Se quiser conferir o [diff com todas as alterações desta parte da aula, clique aqui neste link](https://github.com/NeiltonSeguins/4487-react-hooks/commit/abfe9cd85f9149dac7881baae8bc98b4545919aa) e acesse o github.

Grande abraço!

## ✅ O que aprendemos (Feito com IA)

# Aula 2 – Expondo referências e métodos

## Projeto da aula anterior

Você pode ir acompanhando o passo a passo do desenvolvimento do nosso projeto e, caso deseje, \*\*[você pode acessar o site do Github](https://github.com/NeiltonSeguins/4487-react-hooks/tree/aula-01)\*\* ou \*\*[baixar o projeto da aula anterior](https://github.com/NeiltonSeguins/4487-react-hooks/archive/refs/heads/aula-01.zip)\*\*.

Bons estudos!

## ✅ Vídeo 2.1 - Referenciando elementos com useRef

## ✅ Atividade de alternativa: Exposição de referências em componentes customizados (Feito com IA)

## Mão na massa: Controlando reprodução de música com useRef

Nesta aula, exploramos o hook useRef do React, entendendo seu funcionamento, sua utilidade e como utilizá-lo para manter referências a elementos ou valores mutáveis que persistem entre renderizações sem causar novas renderizações. Como exemplo prático, implementamos a funcionalidade de controlar a reprodução de uma música utilizando useRef.

Agora é sua vez de praticar, fazendo o mesmo no seu projeto! Siga o passo a passo:

1. **Configure a referência para o objeto de áudio:**
   1. Utilize o hook useRef para criar uma referência que armazenará o objeto de áudio.
2. **Inicialize o objeto de áudio dentro de um efeito:**
   1. Use o hook useEffect para inicializar o objeto de áudio quando o componente for montado e limpá-lo quando o componente for desmontado.
3. **Implemente a função para alternar a reprodução da música:**
   1. Crie uma função que utilize a referência do objeto de áudio para iniciar ou pausar a reprodução da música.

Abaixo está o passo a passo detalhado de como você pode realizar a atividade, com comentários explicativos no código! Mas é muito importante que você tente fazer antes de olhar o gabarito. Isto irá te desafiar e mesmo que você não consiga ainda será um(a) desenvolvedor(a) melhor por ter tentado.

**Opinião da pessoa instrutora:**

Pulando o papo motivacional ali em cima, me conta: você conseguiu fazer? Se sim, ótimo! Mas se não, não se preocupe. Aqui ninguém solta a mão de ninguém e preparei um gabarito com a forma que eu fiz, aplicando tudo que vimos até aqui como o useState, useEffect e o useRef. Vamos lá!

## ## Configurando a referência para o objeto de áudio:

No seu componente, importe os Hooks useRef e useEffect do React. Em seguida, defina uma referência utilizando useRef para armazenar o objeto de áudio.

```jsx

import { useRef, useEffect } from "react";  
import musicaSom from "/src/assets/sons/luna-rise-part-one.mp3";  
  
export default function SwitchMusica() {  
 // Cria uma referência para armazenar o objeto de áudio  
 const musicaRef = useRef(null);  
  
 // ...  
}

```  
Usamoso useRef para termos a referência da música. O objetivo aqui é inicializar um objeto de áudio direto no DOM, para que possamos controlar a música sem um estado. Isso é muito bom porque podemos também limpar o efeito da música tocando, isto é, se o usuário for para outra página da aplicação o áudio será interrompido, porque o componente foi desmontado, ou seja, não está mais em tela. Maneiro né?

## ## Inicializando o objeto de áudio dentro de um efeito:

Agora utilize o hook useEffect para criar uma instância do objeto Audio e atribua à referência `musicaRef` quando o componente for montado. Além disso, implemente uma função de limpeza para pausar a música e remover a referência quando o componente for desmontado.

```jsx

useEffect(() => {  
 // Inicializa o objeto de áudio e armazena na referência  
 musicaRef.current = new Audio(musicaSom);  
  
 // Função de limpeza que pausa a música e remove a referência  
 return () => {  
 musicaRef.current?.pause();  
 musicaRef.current = null;  
 };  
}, []);

```  
Sabe me dizer agora porque o array de dependências está vazio? Não vale colar, hein... (risos). Bom, array de dependências vazio significa que este efeito só vai acontecer uma vez, como vimos.

Agora precisamos implementar uma função para alternar a reprodução da música.

## ## Implementando a função para alternar a reprodução da música:

Crie uma função que verifica o estado atual da reprodução e alterna entre reproduzir e pausar a música utilizando a referência musicaRef.

```jsx

function alternarMusica() {  
 // Verifica se o objeto de áudio está inicializado  
 if (!musicaRef.current) return;  
  
 // Se a música estiver pausada, inicia a reprodução; caso contrário, pausa  
 if (musicaRef.current.paused) {  
 musicaRef.current.play();  
 } else {  
 musicaRef.current.pause();  
 }  
}

```

Essa função alterna a reprodução da música. Se ela está tocando a função a pausa e vice-versa. Tudo isso controlando diretamente pela nossa referência.

## ## Integrando a função ao componente:

Adicione um botão ao componente que chama a função `alternarMusica` ao ser clicado.

```jsx

return (  
 <div>  
 {/\* Botão que alterna a reprodução da música \*/}  
 <button onClick={alternarMusica}>Alternar Música</button>  
 </div>  
);

```  
 E o nosso componente de `< SwitchMusica />` fica assim:

```jsx

import { useRef } from "react";  
import styles from "./styles.module.css";  
import { useEffect } from "react";  
import musicaSom from "/src/assets/sons/luna-rise-part-one.mp3";  
  
export default function SwitchMusica() {  
 const musicaRef = useRef(null);  
  
 useEffect(() => {  
 musicaRef.current = new Audio(musicaSom);  
  
 return () => {  
 musicaRef.current?.pause();  
 musicaRef.current = null;  
 };  
 }, []);  
  
 function alternarMusica() {  
 if (!musicaRef.current) return;  
  
 if (musicaRef.current.paused) {  
 musicaRef.current.play();  
 } else {  
 musicaRef.current.pause();  
 }  
 }  
  
 return (  
 <label className={styles["toggle"]}>  
 <input  
 onChange={alternarMusica}  
 className={styles["toggle\_\_checkbox"]}  
 type="checkbox"  
 id="alternar-musica"  
 />  
 <div className={styles["toggle\_\_switch"]}></div>  
 <span className={styles["toggle\_\_span"]} htmlFor="alternar-musica">  
 Música  
 </span>  
 </label>  
 );  
}

```

Legal né? Agora você pode optar por reproduzir ou não uma música para se concentrar durante a execução das suas tarefas aplicando o método pomodoro.

Se quiser [conferir o código completo desta atividade clica aqui neste link](https://github.com/NeiltonSeguins/4487-react-hooks/commit/b5db210eeca4186c88de489bc3916772da746cfd). Ah, e qualquer dúvida não deixe de usar nosso fórum!

Bons estudos!

## Para saber mais: React 19 e o uso de referências

O [React 19](https://react.dev/blog/2024/12/05/react-19#ref-as-a-prop) introduziu mudanças significativas na maneira como referências (refs) são utilizadas em componentes funcionais. Anteriormente, para que um componente funcional pudesse receber uma ref, era necessário envolvê-lo com a função [forwardRef](https://react.dev/reference/react/forwardRef). A partir do React 19, isso não é mais necessário, pois a ref pode ser passada diretamente como uma propriedade (prop) para o componente.

## **## Uso de refs antes do React 19**

Antes do React 19, para que um componente funcional aceitasse uma ref, era necessário utilizar a função forwardRef. Isso permitia que o componente recebesse a ref como um segundo argumento na função de renderização:

```jsx

import { forwardRef } from 'react';  
  
const MeuInput = forwardRef(function MeuInput(props, ref) {  
 return <input {...props} ref={ref} />;  
});

```jsx

Nesse exemplo, MeuInput é um componente funcional que, graças ao uso de forwardRef, pode receber uma ref passada pelo componente pai e associá-la ao elemento <input>.

## **## Uso de refs no React 19**

Com a chegada do React 19, os componentes funcionais podem receber a ref diretamente como uma prop, eliminando a necessidade de envolver o componente com forwardRef:

```jsx

function MeuInput({ placeholder, ref }) {  
 return <input placeholder={placeholder} ref={ref} />;  
}

```

Agora, ao utilizar MeuInput, a ref pode ser passada diretamente:

```jsx

const ref = useRef();  
  
<MeuInput ref={ref} placeholder="Digite algo" />;  
```jsx

Essa abordagem simplifica o código e torna a utilização de refs muito mais simples.

## **## Depreciação do forwardRef**

Com essa mudança, o uso de forwardRef se torna desnecessário para novos componentes funcionais (isto apenas para versões 19+). O React planeja depreciar e, eventualmente, remover forwardRef em versões futuras. Para facilitar a transição, será disponibilizado um codemod que atualizará automaticamente os componentes para utilizar a nova abordagem com a ref como prop.

## **## Para ir mais fundo**

Se você quiser se aprofundar nessas e outras mudanças que chegam com a nova versão do React, a versão 19, recomendo que você [leia este artigo do blog do React com as mudanças](https://react.dev/blog/2024/12/05/react-19) e todas as novidades.

Bons estudos!

## ✅ Vídeo 2.2 - Expondo métodos com o useImperativeHandler

## ✅ Atividade de alternativa: Otimizando a comunicação entre componentes (Feito com IA)

## Mão na massa: Controlando a reprodução de música com useImperativeHandler e useRef

Nesta aula, exploramos o hook `useImperativeHandle` do React e como ele pode ser utilizado em conjunto com `useRef` para permitir que um componente pai controle funções específicas de um componente filho. Vimos o exemplo de controlar o componente de VideoPlayer.

Mas agora é sua vez de praticar, fazendo o mesmo no seu projeto! Use o useImperativeHandler para expor os métodos de um componente filho para seu pai. E aí? Mão na massa, ou melhor, no código?

Siga o passo a passo:

1. **Crie o componente filho que reproduz a música:**
   1. Utilize useRef para criar uma referência ao elemento de áudio.
   2. Use useImperativeHandle para expor métodos específicos (play e pause) ao componente pai.
2. **Implemente o componente pai que controla o componente filho:**
   1. Utilize useRef para criar uma referência ao componente filho.
   2. Implemente botões que acionam os métodos play e pause do componente filho.

Abaixo está o passo a passo detalhado de como você pode realizar a atividade, com comentários explicativos no código!

**Opinião da pessoa instrutora:**

Espero que você não tenha espiado aqui antes de tentar fazer, pois isso não seria eficiente para o seu aprendizado. Então se fez isso, volta lá e tenta fazer, depois volta aqui. Agora se você tentou fazer e não conseguiu, ou conseguiu fazer e quer ver uma outra solução, continuemos aqui mesmo.

## **## Refatorando o componente**

Para começar, o componente que vamos expor os métodos é o componente de `<SwitchMusica />`. No momento ele está assim:

```jsx

import { useRef } from "react";  
import styles from "./styles.module.css";  
import { useEffect } from "react";  
import musicaSom from "/src/assets/sons/luna-rise-part-one.mp3";  
  
export default function SwitchMusica() {  
 const musicaRef = useRef(null);  
  
 useEffect(() => {  
 musicaRef.current = new Audio(musicaSom);  
  
 return () => {  
 musicaRef.current?.pause();  
 musicaRef.current = null;  
 };  
 }, []);  
  
 function alternarMusica() {  
 if (!musicaRef.current) return;  
  
 if (musicaRef.current.paused) {  
 musicaRef.current.play();  
 } else {  
 musicaRef.current.pause();  
 }  
 }  
  
 return (  
 <label className={styles["toggle"]}>  
 <input  
 onChange={alternarMusica}  
 className={styles["toggle\_\_checkbox"]}  
 type="checkbox"  
 id="alternar-musica"  
 />  
 <div className={styles["toggle\_\_switch"]}></div>  
 <span className={styles["toggle\_\_span"]} htmlFor="alternar-musica">  
 Música  
 </span>  
 </label>  
 );  
}

```

E como estamos na versão 18.3.1 do React, precisamos do `forwardRef` para permitir que nosso componente aceite receber uma ref. Então vamos modificar o SwithMusica.

```jsx

/\* eslint-disable react/display-name \*/  
import { forwardRef, useImperativeHandle, useRef } from "react";  
import styles from "./styles.module.css";  
import { useEffect } from "react";  
import musicaSom from "/src/assets/sons/luna-rise-part-one.mp3";  
  
const SwitchMusica = forwardRef((props, ref) => {  
 const musicaRef = useRef(null);  
  
 useEffect(() => {  
 musicaRef.current = new Audio(musicaSom);  
  
 return () => {  
 musicaRef.current?.pause();  
 musicaRef.current = null;  
 };  
 }, []);  
  
 ‘  
 function alternarMusica() {  
 if (!musicaRef.current) return;  
  
 if (musicaRef.current.paused) {  
 musicaRef.current.play();  
 } else {  
 musicaRef.current.pause();  
 }  
 }  
  
 return (  
 <label className={styles["toggle"]}>  
 <input  
 onChange={alternarMusica}  
 className={styles["toggle\_\_checkbox"]}  
 type="checkbox"  
  
 id="alternar-musica"  
 />  
 <div className={styles["toggle\_\_switch"]}></div>  
 <span className={styles["toggle\_\_span"]} htmlFor="alternar-musica">  
 Música  
 </span>  
 </label>  
 );  
});  
export default SwitchMusica;

```

As principais diferenças aí são:

1. Agora o componente usa o forwardRef para aceitar uma referência do componente pai
2. Devido à isso, modifiquei o modo de exportação do componente, criando ele como uma arrow function e exportando ele por padrão ao fim do arquivo.

## **## Adicionando o useImperativeHandler**

Agora vamos adicionar o hook useImperativeHandler para expôr os métodos de play e pause ao componente pai.

```jsx

useImperativeHandle(ref, () => ({  
 play: () => {  
 if (musicaRef.current) {  
 musicaRef.current.play();  
 }  
 },  
 pause: () => {  
 if (musicaRef.current) {  
 musicaRef.current.pause();  
 }  
 },  
 }));

```

Dentro do hook, da função de callback dele, a gente retorna um objeto com os métodos de play e pause. Estes métodos foram criados usando um pouco de programação defensiva, onde antes eu verifico se a gente tem a referência e logo depois acesso o método que preciso.

E para finalizar eu adiciono um atributo `checked` no input, para podermos alterar visualmente o Switch na aplicação. O código completo do SwitchMusica fica assim:

```jsx

/\* eslint-disable react/display-name \*/  
import { forwardRef, useImperativeHandle, useRef } from "react";  
import styles from "./styles.module.css";  
import { useEffect } from "react";  
import musicaSom from "/src/assets/sons/luna-rise-part-one.mp3";  
  
const SwitchMusica = forwardRef((props, ref) => {  
 const musicaRef = useRef(null);  
  
 useEffect(() => {  
 musicaRef.current = new Audio(musicaSom);  
  
 return () => {  
 musicaRef.current?.pause();  
 musicaRef.current = null;  
 };  
 }, []);  
  
 useImperativeHandle(ref, () => ({  
 play: () => {  
 if (musicaRef.current) {  
 musicaRef.current.play();  
 }  
 },  
 pause: () => {  
 if (musicaRef.current) {  
 musicaRef.current.pause();  
 }  
 },  
 }));  
  
 function alternarMusica() {  
 if (!musicaRef.current) return;  
  
 if (musicaRef.current.paused) {  
 musicaRef.current.play();  
 } else {  
 musicaRef.current.pause();  
 }  
 }  
  
 return (  
 <label className={styles["toggle"]}>  
 <input  
 onChange={alternarMusica}  
 className={styles["toggle\_\_checkbox"]}  
 type="checkbox"  
 checked={!musicaRef.current?.paused}  
 id="alternar-musica"  
 />  
 <div className={styles["toggle\_\_switch"]}></div>  
 <span className={styles["toggle\_\_span"]} htmlFor="alternar-musica">  
 Música  
 </span>  
 </label>  
 );  
});  
export default SwitchMusica;

```

## Atualizando o componente Cronometro

Agora que estamos expondo os métodos do componente filho ao pai, vamos usar estes métodos no componente pai. Vamos lá!

O seu componente de Cronometro está com essa cara:

```jsx

import styles from "./styles.module.css";

import BotoesModos from "./BotoesModos";

import Timer from "./Timer";

import SwitchMusica from "./SwitchMusica";

import BotaoCronometro from "./BotaoCronometro";

import { useEffect } from "react";

import { useState } from "react";

const modos = [

{

id: "foco",

nome: "Foco",

duracao: 25, // em minutos

},

{

id: "pausa\_curta",

nome: "Pausa curta",

duracao: 5, // em minutos

},

{

id: "pausa\_longa",

nome: "Pausa longa",

duracao: 15, // em minutos

},

];

export default function Cronometro() {

const [modoAtual, setModoAtual] = useState(modos[0]);

const [tempoRestante, setTempoRestante] = useState(modoAtual.duracao);

const [emExecucao, setEmExecucao] = useState(false); // Para saber se o cronômetro está rodando

useEffect(() => {

let intervalo;

if (emExecucao && tempoRestante > 0) {

intervalo = setInterval(() => {

setTempoRestante((prev) => prev - 1);

}, 1000);

} else if (tempoRestante === 0) {

setModoAtual((prev) => {

const modoAtualIndex = modos.findIndex((modo) => modo.id === prev.id);

const proximoModo = modos[(modoAtualIndex + 1) % modos.length];

setTempoRestante(proximoModo.duracao);

return proximoModo;

});

setEmExecucao(false);

}

return () => clearInterval(intervalo);

}, [emExecucao, tempoRestante, modoAtual]);

const alternarCronometro = () => {

setEmExecucao((prev) => !prev);

};

const selecionarModo = (novoModo) => {

setModoAtual(novoModo);

setTempoRestante(novoModo.duracao);

setEmExecucao(false);

};

return (

<div className={styles["cronometer"]}>

<BotoesModos modos={modos} modoAtual={modoAtual} onModoSelecionado={selecionarModo} />

<Timer tempoRestante={tempoRestante} />

<SwitchMusica />

<BotaoCronometro emExecucao={emExecucao} onClick={alternarCronometro} />

</div>

);

}

```

Você vai precisar adicionar uma referência e passar para o componente SwitchMusica. Além disso, precisa criar uma função para controlar a música. Outra coisa legal que pode fazer é iniciar a música quando clicar no botão de “Começar”. Vamos fazer isso:

```jsx  
import { useEffect, useRef } from "react"; // Importe o useRef  
  
export default function Cronometro() {  
 const musicaRef = useRef(null); // cria uma referência  
  
// cria uma função para controlar a música

const controlarMusica = () => {  
 if (musicaRef.current) {  
 musicaRef.current.play();  
 }  
 };  
  
 const alternarCronometro = () => {  
 setEmExecucao((prev) => !prev);  
  
// Deixe a música reproduzindo mesmo alternando o cronometro, o modo.

if (!emExecucao) {  
 controlarMusica();  
 } else {  
 musicaRef.current.pause();  
 }  
 };  
  
 const selecionarModo = (novoModo) => {  
 setModoAtual(novoModo);  
 setTempoRestante(novoModo.duracao);  
 setEmExecucao(false);

// pause a música ao mudar de modo  
 musicaRef.current.pause();  
 };  
  
 return (  
 <div className={styles["cronometer"]}>  
 <BotoesModos modos={modos} modoAtual={modoAtual} onModoSelecionado={selecionarModo} />  
 <Timer tempoRestante={tempoRestante} />  
 <SwitchMusica ref={musicaRef} /> // passe a ref para o componente de SwitchMusica  
 <BotaoCronometro emExecucao={emExecucao} onClick={alternarCronometro} />  
 </div>  
 );  
}

```

Eu omiti algumas partes repetidas do código e destaquei com comentários onde modificamos o componente de cronômetro. Desta forma, fica mais claro onde adicionamos coisas novas, mas o código completo do componente de Cronometro está mostrado abaixo:

```jsx

import styles from "./styles.module.css";  
  
import BotoesModos from "./BotoesModos";  
import Timer from "./Timer";  
import SwitchMusica from "./SwitchMusica";  
import BotaoCronometro from "./BotaoCronometro";  
import { useEffect, useRef } from "react";  
import { useState } from "react";  
  
const modos = [  
 {  
 id: "foco",  
 nome: "Foco",  
 duracao: 25, // em minutos  
 },  
 {  
 id: "pausa\_curta",  
 nome: "Pausa curta",  
 duracao: 5, // em minutos  
 },  
 {  
 id: "pausa\_longa",  
 nome: "Pausa longa",  
 duracao: 15, // em minutos  
 },  
];  
  
export default function Cronometro() {  
 const musicaRef = useRef(null);  
  
 const [modoAtual, setModoAtual] = useState(modos[0]);  
 const [tempoRestante, setTempoRestante] = useState(modoAtual.duracao);  
 const [emExecucao, setEmExecucao] = useState(false); // Para saber se o cronômetro está rodando  
  
 useEffect(() => {  
 let intervalo;  
  
 if (emExecucao && tempoRestante > 0) {  
 intervalo = setInterval(() => {  
 setTempoRestante((prev) => prev - 1);  
 }, 1000);  
 } else if (tempoRestante === 0) {  
 setModoAtual((prev) => {  
 const modoAtualIndex = modos.findIndex((modo) => modo.id === prev.id);  
 const proximoModo = modos[(modoAtualIndex + 1) % modos.length];  
 setTempoRestante(proximoModo.duracao);  
 return proximoModo;  
 });  
 setEmExecucao(false);  
 }  
  
 return () => clearInterval(intervalo);  
 }, [emExecucao, tempoRestante, modoAtual]);  
  
 const controlarMusica = () => {  
 if (musicaRef.current) {  
 musicaRef.current.play();  
 }  
 };  
  
 const alternarCronometro = () => {  
 setEmExecucao((prev) => !prev);  
  
 if (!emExecucao) {  
 controlarMusica();  
 } else {  
 musicaRef.current.pause();  
 }  
 };  
  
 const selecionarModo = (novoModo) => {  
 setModoAtual(novoModo);  
 setTempoRestante(novoModo.duracao);  
 setEmExecucao(false);  
 musicaRef.current.pause();  
 };  
  
 return (  
 <div className={styles["cronometer"]}>  
 <BotoesModos modos={modos} modoAtual={modoAtual} onModoSelecionado={selecionarModo} />  
 <Timer tempoRestante={tempoRestante} />  
 <SwitchMusica ref={musicaRef} />  
 <BotaoCronometro emExecucao={emExecucao} onClick={alternarCronometro} />  
 </div>  
 );  
}

```

Tranquilo? Notou alguma nesse código? Um ponto de melhoria, alguma dificuldade? Eu notei, ele tá com responsabilidades demais. O componente de cronômetro tem que lidar com muitas coisas, como pintar a tela, controlar o cronômetro, controlar a música. Precisamos resolver isso e temos uma forma nativa no React que iremos conhecer na próxima aula.

Se você quiser [ver o código completo desta aula](https://github.com/NeiltonSeguins/4487-react-hooks/commit/9f765394e5f91901ed961dc1d71ac4ef04be762e), acesse este link e veja no github.

Bons estudos!

## ✅ O que aprendemos (Feito com IA)

# Aula 3 - Contexto e estados complexos

## Projeto da aula anterior

Você pode ir acompanhando o passo a passo do desenvolvimento do nosso projeto e, caso deseje, \*\*[você pode acessar o site do Github](https://github.com/NeiltonSeguins/4487-react-hooks/tree/aula-02)\*\* ou \*\*[baixar o projeto da aula anterior](https://github.com/NeiltonSeguins/4487-react-hooks/archive/refs/heads/aula-02.zip)\*\*.

Bons estudos!

## ✅ Vídeo 3.1 - Acessando o contexto com useContext

## ✅ Atividade de alternativa: Gerenciando promoções (Feito com IA)

## Mão na massa: Gerenciando o estado global de tarefas

Nesta aula, exploramos o hook useContext do React, que permite acessar o contexto global da aplicação, evitando a necessidade de *prop drilling* (passagem de props por múltiplos níveis). Até aqui já deu para notar que alguns componentes estão ficando grandes demais, com muitas responsabilidades e precisando passar props a níveis profundos de componentes muito acima na árvore de componentes.

Então agora é sua vez de praticar! Começando pelo componente `< AppTarefas />`. Crie as funções para adicionar tarefas, deletar tarefas, marcar uma tarefa como concluída, deletar todas as tarefas e deletar apenas as concluídas.

Siga o passo a passo:

1. **Crie o contexto para gerenciar o estado das tarefas:**
   1. Utilize createContext para criar um contexto que armazenará o estado global das tarefas e as funções para manipulá-lo.
2. **Forneça o contexto aos componentes filhos:**
   1. Utilize o componente TarefasProvider para encapsular os componentes que necessitam acessar o contexto, fornecendo o estado e as funções através da propriedade value.
3. **Consuma o contexto nos componentes necessários:**
   1. Utilize o Hook useContext nos componentes que precisam acessar ou modificar o estado global das tarefas.

Esta atividade envolve muita refatoração de componentes, então você irá precisar ter bastante atenção. Tente fazer, mas não se preocupe, abaixo está o passo a passo detalhado de como você pode realizar a atividade, com comentários explicativos no código!

**Opinião da pessoa instrutora:**

## Criando o contexto de Tarefas

Então para resolver mais este desafio, precisamos usar a Context API do React. Então para manter nosso projeto organizado, vamos criar um arquivo dentro da pasta `src/context/TarefasContext.jsx`. Neste arquivo iremos criar o contexto:

```jsx

import { createContext, useState } from "react";

export const TarefasContext = createContext();

const TarefasProvider = ({ children }) => {

const [tarefas, setTarefas] = useState([]);

const [exibirFormulario, setExibirFormulario] = useState(false);

function adicionarTarefa(texto) {

if (!texto.trim()) return;

const novaTarefa = {

id: Math.floor(Math.random() \* 100000) + Date.now(),

texto,

concluida: false,

};

setTarefas([...tarefas, novaTarefa]);

setExibirFormulario(false);

}

const aoExibirFormulario = (valor) => {

setExibirFormulario(valor);

};

function deletarTarefa(id) {

setTarefas((prev) => prev.filter((tarefa) => tarefa.id !== id));

}

function deletarTodasTarefas() {

setTarefas([]);

}

function deletarTarefasConcluidas() {

setTarefas((prev) => prev.filter((tarefa) => !tarefa.concluida));

}

function marcarComoConcluida(id) {

setTarefas((prev) =>

prev.map((tarefa) =>

tarefa.id === id ? { ...tarefa, concluida: !tarefa.concluida } : tarefa,

),

);

}

return (

<TarefasContext.Provider

value={{

tarefas,

adicionarTarefa,

deletarTarefa,

deletarTodasTarefas,

deletarTarefasConcluidas,

marcarComoConcluida,

exibirFormulario,

aoExibirFormulario,

}}

>

{children}

</TarefasContext.Provider>

);

};

export default TarefasProvider;

```

Os estaodos de tarefas e exibirFormulario, bem como a função de adicionar tarefas a gente fez lá no início, lembra? Agora estamos tirando esta responsabilidade do componente e usando a Context API para isso.

Aproveitei e criei as novas funções de deletarTarefa, deletarTodasTarefas, deletarTarefasConcluidas, marcarComoConcluida e aoExibirFormulario. Esta última só fiz pois não queria expôr diretamente o setState de exibir o formulário.

## Envolvendo o App no Contexto de Tarefas

Bom, antes de usar o hook useContext para recuperar o contexto e consequêntemente estados e funções dentro dele, a gente precisa envolver a aplicação em um Provider.

```jsx

import { StrictMode } from "react";

import { createRoot } from "react-dom/client";

import App from "./App.jsx";

import "./index.css";

import TarefasProvider from "./context/TarefasContext.jsx";

createRoot(document.getElementById("root")).render(

<StrictMode>

<TarefasProvider>

<App />

</TarefasProvider>

</StrictMode>,

);

```

## Atualizando os componentes

E agora vamos atualizar os componentes que dependiam do estado de tarefas. Começando pelo AppTarefas:

```jsx

import styles from "./styles.module.css";  
import add\_circleImg from "/src/assets/imgs/add\_circle.png";  
import Botao from "./Botao";  
import CabecalhoTarefas from "./CabecalhoTarefas";  
import FormularioTarefas from "./FormularioTarefas";  
import ListaTarefas from "./ListaTarefas";  
import { useContext } from "react";  
import { TarefasContext } from "../../context/TarefasContext";  
  
const AppTarefas = () => {  
 const { exibirFormulario, aoExibirFormulario } = useContext(TarefasContext);  
  
 return (  
 <section className={styles["tasks"]}>  
 <div className={styles["tasks\_\_task-content"]}>  
 <CabecalhoTarefas />  
 <ListaTarefas />  
 {exibirFormulario && <FormularioTarefas />}  
 <Botao  
 onClick={() => aoExibirFormulario(true)}  
 icone={add\_circleImg}  
 className={styles["tasks\_\_button--add-task"]}  
 >  
 Adicionar nova tarefa  
 </Botao>  
 </div>  
 </section>  
 );  
};  
  
export default AppTarefas;

```

Percebeu que removi as props dos componentes de FormularioTarefas e ListaTarefas? É porque com a Context API a gente consegue acessar o estado diretamente dentro destes componentes com o useContext. Vamos lá!

### FormularioTarefas

```jsx

import { useContext, useState } from "react";

import CampoTexto from "../CampoTexto";

import RodapeTarefas from "../RodapeTarefas";

import { TarefasContext } from "../../../context/TarefasContext";

import styles from "../styles.module.css";

const FormularioTarefas = () => {

const { exibirFormulario, adicionarTarefa, aoExibirFormulario } = useContext(TarefasContext);

const [texto, setTexto] = useState("");

return (

<form

className={`${styles["form-add-task"]} ${!exibirFormulario ? styles["form-add-task--hidden"] : ""}`}

aria-hidden={!exibirFormulario}

>

<CampoTexto

value={texto}

onChange={(e) => setTexto(e.target.value)}

etiqueta="Adicionando tarefa"

placeholder="No que você está trabalhando?"

/>

<RodapeTarefas aoSalvar={() => adicionarTarefa(texto)} aoCancelar={aoExibirFormulario} />

</form>

);

};

export default FormularioTarefas;

```

### ListaTarefas

```jsx

import { useContext } from "react";  
import Botao from "../Botao";  
import styles from "../styles.module.css";  
import checkImg from "/src/assets/imgs/check-task.png";  
import trashImg from "/src/assets/imgs/trash.svg";  
import { TarefasContext } from "../../../context/TarefasContext";  
  
const ListaTarefas = () => {  
 const { tarefas, deletarTarefa, marcarComoConcluida } = useContext(TarefasContext);  
 return (  
 <ul className={styles["tasks\_\_task-list"]}>  
 {tarefas.map((tarefa) => (  
 <li  
 key={tarefa.id}  
 className={`${styles["tasks\_\_task-list-item"]} ${tarefa.concluida ? styles["tasks\_\_task-list-item-complete"] : ""}`}  
 >  
 <Botao  
 icone={checkImg}  
 id="marcar-tarefa-como-concluida"  
 className={`${styles["check-task\_\_button"]} ${tarefa.concluida ? styles["check-task\_\_button--complete"] : ""}`}  
 onClick={() => marcarComoConcluida(tarefa.id)}  
 />  
 <p className={styles["tasks\_\_task-list-item-description"]}>{tarefa.texto}</p>  
 <Botao  
 icone={trashImg}  
 id="deletar-tarefa"  
 className={`${styles["delete-task\_\_button"]}`}  
 onClick={() => deletarTarefa(tarefa.id)}  
 />  
 </li>  
 ))}  
 </ul>  
 );  
};  
  
export default ListaTarefas;

```

### MenuOpcoes

```jsx

import styles from "../styles.module.css";  
import moreImg from "/src/assets/imgs/more.svg";  
import checkImg from "/src/assets/imgs/check.svg";  
import trashImg from "/src/assets/imgs/trash.svg";  
import Botao from "../Botao";  
import { TarefasContext } from "../../../context/TarefasContext";  
import { useContext } from "react";  
  
const MenuOpcoes = () => {  
 const { deletarTodasTarefas, deletarTarefasConcluidas } = useContext(TarefasContext);  
 return (  
 <div className={styles["dropdown"]}>  
 <Botao icone={moreImg} className={styles["dropdown\_\_button-more"]} />  
  
 <ul className={styles["dropdown\_\_ul"]}>  
 <li className={styles["dropdown\_\_li"]}>  
 <Botao  
 icone={checkImg}  
 className={styles["dropdown\_\_button"]}  
 onClick={deletarTarefasConcluidas}  
 id="btn-remover-concluidas"  
 >  
 Limpar tarefas concluídas  
 </Botao>  
 </li>  
 <li className={styles["dropdown\_\_li"]}>  
 <Botao  
 icone={trashImg}  
 className={styles["dropdown\_\_button"]}  
 onClick={deletarTodasTarefas}  
 id="btn-remover-concluidas"  
 >  
 Limpar todas as tarefas  
 </Botao>  
 </li>  
 </ul>  
 </div>  
 );  
};  
  
export default MenuOpcoes;

```

E com isso agora os nossos componentes estão recuperando estados e funções diretamente da Context API. O useContext acessa o contexto específico, e o React o procura em nossa aplicação. Daí, temos acesso a esses estados e funções que agora são globais. Poderíamos inclusive usar o Provider no App, envolvendo o componente App que também funcionaria:

```jsx

function App() {

const modoCronometro = {

id: "foco",

nome: "Foco",

frase: ["Otimize sua produtividade,", "mergulhe no que importa."],

tempoInicialEmSegundos: 30,

};

return (

<div className={styles[`app--${modoCronometro.id}`]}>

<Cabecalho />

<main>

<Cronometro />

<TarefasProvider >

<AppTarefas />

<TarefasProvider />

</main>

<Rodape />

</div>

);

}

```

Belezinha? Com isso, agora nossos componentes do AppTarefas não precisam conhecer lógica de adicionar tarefas, excluir tarefas, etc. Nem precisam conhecer o estado, tudo está encapsulado no TarefasContext. Isso reduziu responsabilidades dos componentes, deixando nossa aplicação mais escalável, segura e seguindo padrões recomendados.

Se quiser conferir o código completo desta atividade, [clique aqui](https://github.com/NeiltonSeguins/4487-react-hooks/commit/9553874ec6fc0e0e792bf03b5efce2f0d8b6c4e7#diff-d9293b831981800272cde0a10a8c40900ddd0d520ed92854cb1840ce9bd57302R4) e veja no github.

Abraços!

## Para saber mais: Bibliotecas de gerenciamento de estados

Você sabia que, no React, gerenciar o estado da aplicação pode rapidamente se tornar um caos conforme o projeto cresce? Manter tudo organizado entre componentes pode ser desafiador, e é aí que entram bibliotecas como Redux, Zustand, Recoil e MobX! Cada uma tem sua abordagem para facilitar a vida dos desenvolvedores, tornando o código escalável, seguindo boas práticas de gerenciamento de estados e muito mais.

Quer entender melhor qual delas se encaixa no seu projeto? Confira [esta formação](https://cursos.alura.com.br/formacao-react-gerenciamento-estados) completa que traz as principais bibliotecas de gerenciamento de estado no React.

## ✅ Vídeo 3.2 - Gerenciando estados complexos com useReducer

## ✅ Atividade de alternativa: Personalizando a experiência de compra (Feito com IA)

## Mão na massa: Gerenciando os estados do Cronometro

Já que criamos um contexto para as tarefas, não podemos deixar de fazer o mesmo para os estados do Cronometro, concorda? Você deve ter imaginado que iríamos fazer algo com o useReducer, não é mesmo? Mas vamos primeiro organizar nosso app e depois usamos o useReducer.

Então agora é sua vez de praticar! Crie um contexto para gerenciar os estados do Cronometro.

Siga o passo a passo:

1. **Crie o contexto para gerenciar o estado do cronômetro:**
   1. Utilize createContext para criar um contexto que armazenará o estado global do cronômetro e as funções para manipulá-lo.
2. **Forneça o contexto aos componentes filhos:**
   1. Utilize o componente CronometroProvider para encapsular os componentes que necessitam acessar o contexto, fornecendo o estado e as funções através da propriedade value.
3. **Consuma o contexto nos componentes necessários:**
   1. Utilize o Hook useContext nos componentes que precisam acessar ou modificar o estado global do cronômetro.

Abaixo está o passo a passo detalhado de como você pode realizar a atividade, com comentários explicativos no código!

**Opinião da pessoa instrutora:**

Mais uma atividade para você praticar e se desafiar. Esta é bem parecida com o mão na massa anterior, onde criamos o contexto para as tarefas. Vamos fazer o mesmo só que para o Cronometro.

## ## Criando o contexto do Cronometro

Vamos criar um arquivo `src/context/CronometroContext.jsx`:

```jsx

import { createContext, useState, useEffect, useRef } from "react";  
import beepSound from "../assets/sons/beep.mp3";  
  
export const CronometroContext = createContext();  
  
const modos = [  
 {  
 id: "foco",  
 nome: "Foco",  
 duracao: 25, // em segundos  
 },  
 {  
 id: "pausa\_curta",  
 nome: "Pausa curta",  
 duracao: 5, // em segundos  
 },  
 {  
 id: "pausa\_longa",  
 nome: "Pausa longa",  
 duracao: 15, // em segundos  
 },  
];  
  
const CronometroProvider = ({ children }) => {  
 const musicaRef = useRef(null);  
 const beepRef = useRef(new Audio(beepSound));  
  
 const [modoAtual, setModoAtual] = useState(modos[0]);  
 const [tempoRestante, setTempoRestante] = useState(modoAtual.duracao);  
 const [emExecucao, setEmExecucao] = useState(false);  
 const [musicaTocando, setMusicaTocando] = useState(false);  
  
 useEffect(() => {  
 let intervalo;  
  
 if (emExecucao && tempoRestante > 0) {  
 intervalo = setInterval(() => {  
 setTempoRestante((prev) => prev - 1);  
 }, 1000);  
 } else if (tempoRestante === 0) {  
 beepRef.current.play();  
 setModoAtual((prev) => {  
 const modoAtualIndex = modos.findIndex((modo) => modo.id === prev.id);  
 const proximoModo = modos[(modoAtualIndex + 1) % modos.length];  
 setTempoRestante(proximoModo.duracao);  
 return proximoModo;  
 });  
 setEmExecucao(false);  
 }  
  
 return () => clearInterval(intervalo);  
 }, [emExecucao, tempoRestante]);  
  
 const alternarCronometro = () => {  
 setEmExecucao((prev) => !prev);  
 if (!emExecucao) {  
 controlarMusica();  
 } else {  
 musicaRef.current.pause();  
 }  
 };  
  
 const selecionarModo = (novoModo) => {  
 setModoAtual(novoModo);  
 setTempoRestante(novoModo.duracao);  
 setEmExecucao(false);  
 musicaRef.current.pause();  
 };  
  
 const controlarMusica = () => {  
 if (musicaRef.current) {  
 musicaRef.current.play();  
 setMusicaTocando(true);  
 }  
 };  
  
 const pausarMusica = () => {  
 if (musicaRef.current) {  
 musicaRef.current.pause();  
 setMusicaTocando(false);  
 }  
 };  
  
 return (  
 <CronometroContext.Provider  
 value={{  
 modos,  
 modoAtual,  
 tempoRestante,  
 emExecucao,  
 musicaTocando,  
 musicaRef,  
 alternarCronometro,  
 selecionarModo,  
 controlarMusica,  
 pausarMusica,  
 }}  
 >  
 {children}  
 </CronometroContext.Provider>  
 );  
};  
  
export default CronometroProvider;

```

Boa! Então o próximo passo é adicionar o provider em volta da nossa aplicação. Aqui tanto faz no `main.jsx` ou no App.jsx` em volta do Cronometro. Você decide. Eu optei pelo `main.jsx`:

```jsx

import { StrictMode } from "react";

import { createRoot } from "react-dom/client";

import App from "./App.jsx";

import TarefasProvider from "./context/TarefasContext.jsx";

import CronometroProvider from "./context/CronometroContext.jsx";

import "./index.css";

createRoot(document.getElementById("root")).render(

<StrictMode>

<TarefasProvider>

<CronometroProvider>

<App />

</CronometroProvider>

</TarefasProvider>

</StrictMode>,

);

```

## ## Atualizando os componentes

Agora começa a caça às bruxas, digo, aos componentes que usam estados e funções que agora são compartilhados pelo contexto. Vamos lá!

## Cronometro

```jsx

import styles from "./styles.module.css";

import BotoesModos from "./BotoesModos";

import Timer from "./Timer";

import SwitchMusica from "./SwitchMusica";

import BotaoCronometro from "./BotaoCronometro";

import { CronometroContext } from "../../context/CronometroContext";

import { useContext } from "react";

export default function Cronometro() {

const { musicaRef } = useContext(CronometroContext);

return (

<div className={styles["cronometer"]}>

<BotoesModos />

<Timer />

<SwitchMusica ref={musicaRef} />

<BotaoCronometro />

</div>

);

}

```

## Timer

```jsx

import { useContext } from "react";

import { CronometroContext } from "../../../context/CronometroContext";

import styles from "./styles.module.css";

export default function Timer() {

const { tempoRestante } = useContext(CronometroContext);

const minutos = Math.floor(tempoRestante / 60);

const segundos = tempoRestante % 60;

return (

<div className={styles["cronometer-timer"]}>

{minutos}:{segundos < 10 ? `0${segundos}` : segundos}

</div>

);

}

```

## BotoesModo

```jsx

import styles from "./styles.module.css";

import BotaoModo from "./BotaoModo";

import { useContext } from "react";

import { CronometroContext } from "../../../context/CronometroContext";

const BotoesModos = () => {

const { modos, modoAtual, selecionarModo } = useContext(CronometroContext);

return (

<ul className={styles["cronometer-modes"]}>

{modos.map((m) => (

<li key={m.id}>

<BotaoModo ativo={m.id === modoAtual.id} onClick={() => selecionarModo(m)}>

{m.nome}

</BotaoModo>

</li>

))}

</ul>

);

};

export default BotoesModos;

```

## BotaoCronometro

```jsx

import styles from "./styles.module.css";  
import play\_arrowImg from "/src/assets/imgs/play\_arrow.png";  
import pauseImg from "/src/assets/imgs/pause.png";  
import { useContext } from "react";  
import { CronometroContext } from "../../../context/CronometroContext";  
  
const BotaoCronometro = () => {  
 const { emExecucao, alternarCronometro } = useContext(CronometroContext);  
 return (  
 <div className={styles["cronometer\_\_primary-button-wrapper"]}>  
 <button onClick={alternarCronometro} className={styles["cronometer\_\_primary-button"]}>  
 <img  
 className={styles["cronometer\_\_primary-button-icon"]}  
 src={emExecucao ? pauseImg : play\_arrowImg}  
 />  
 <span>{emExecucao ? "Pausar" : "Começar"}</span>  
 </button>  
 </div>  
 );  
};  
  
export default BotaoCronometro;

```

E finalizamos tudo! Agora nossa aplicação possui contexto para gerenciar a lista de tarefas e o cronometro. Tudo isso nos fez deixar a aplicação mais limpa, escalável, com responsabilidades bem organizadas e separadas. Basta olhar os componentes como estavam antes e agora.

E falando em diferenças, se você quiser conferir o que fizemos nesta aula, [acesse o link no github clicando aqui](https://github.com/NeiltonSeguins/4487-react-hooks/commit/b5246d0b9762bbaa5b6a19895440be1655252cc4).

Bons estudos!

## Para saber mais: reduce, useReducer e Redux

Se você já usou [Redux](https://redux.js.org/tutorials/index) ou ouviu falar dele, deve saber que ele gira em torno de um conceito fundamental: o **dispatch de ações**. Mas você sabia que o useReducer funciona de forma muito semelhante? E mais: ambos seguem a mesma lógica por trás do método reduce do JavaScript. Vamos explorar essas semelhanças de um jeito descomplicado!

## **## O que é o useReducer e como ele funciona?**

Imagine que você tem um carrinho de compras em um e-commerce. Conforme a pessoa usuária adiciona e remove produtos, o estado do carrinho precisa ser atualizado. O useReducer é uma forma mais organizada e previsível de lidar com esse tipo de lógica.

Ele funciona assim:

* Você tem um estado inicial.
* Define um reducer, que é uma função pura responsável por dizer como o estado deve mudar com base em uma ação.
* Usa o dispatch para enviar ações que vão modificar o estado.

Exemplo:

```jsx

import { useReducer } from "react";  
  
const initialState = { count: 0 };  
  
function reducer(state, action) {  
 switch (action.type) {  
 case "increment":  
 return { count: state.count + 1 };  
 case "decrement":  
 return { count: state.count - 1 };  
 default:  
 return state;  
 }  
}  
  
function Contador() {  
 const [state, dispatch] = useReducer(reducer, initialState);  
  
 return (  
 <div>  
 <p>Contagem: {state.count}</p>  
 <button onClick={() => dispatch({ type: "increment" })}>+</button>  
 <button onClick={() => dispatch({ type: "decrement" })}>-</button>  
 </div>  
 );  
}

```

Reparou no padrão? Temos um **estado inicial**, um **reducer que define como o estado muda** e um **dispatch que dispara ações** para modificar esse estado.

## **## E o Redux? Como ele é parecido?**

No Redux, seguimos a mesma lógica:

* Existe um **estado global** (store).
* Temos um **reducer** que determina como o estado muda.
* Usamos um **dispatch** para enviar ações e atualizar o estado.

Aqui está um exemplo equivalente usando Redux:

```jsx

// counterSlice.js

import { createSlice } from '@reduxjs/toolkit';

export const counterSlice = createSlice({

name: 'counter',

initialState: {

value: 0,

},

reducers: {

increment: (state) => {

state.value += 1;

},

decrement: (state) => {

state.value -= 1;

},

incrementByAmount: (state, action) => {

state.value += action.payload;

},

},

});

export const { increment, decrement, incrementByAmount } = counterSlice.actions;

export default counterSlice.reducer;

````

E na store, fazemos o seguinte:

```jsx

// store.js

import { configureStore } from '@reduxjs/toolkit';

import counterReducer from './counterSlice';

const store = configureStore({

reducer: {

counter: counterReducer,

},

});

export default store;

```

A estrutura é muito parecida com o useReducer! A grande diferença é que no Redux o estado é global, enquanto no useReducer ele é local ao componente. Mas nada impede de você combinar o useReducer a Context API para gerenciar estados globalmente em suas aplicações. Fica a dica!

## **## Mas e o reduce do JavaScript? Onde entra nessa história?**

Se você já usou o [método **reduce**](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\_Objects/Array/reduce)do JavaScript, pode notar uma semelhança: ele também recebe um **acumulador (estado)** e uma função que diz como esse acumulador deve ser atualizado.

Exemplo:

```js

const numeros = [1, 2, 3, 4];  
  
const somaTotal = numeros.reduce((acumulador, numero) => acumulador + numero, 0);  
  
console.log(somaTotal); // 10

```

A ideia por trás é a mesma do useReducer e do Redux! Você tem um estado (acumulador), uma função (reducer) que define como esse estado muda e um valor inicial.

Maneiro, né? Se você precisa gerenciar estado local dentro de um componente e quer mais controle sobre as atualizações, useReducer é uma ótima escolha. E você pode combiná-lo com a Context API em aplicações grandes para gerenciar o estado global em alguns casos. Agora, se você tiver uma aplicação muito grande, com muitos estados que precisam ser compartilhados globalmente entre vários componentes, Redux pode ser a melhor alternativa.

No fim das contas, ambos seguem o mesmo princípio e foram inspirados na lógica do reduce do JavaScript. Entender um ajuda a entender o outro! Então, da próxima vez que você ver um dispatch, já sabe que está lidando com um padrão poderoso e familiar.

## **## Para ir mais fundo**

Quer aprender sobre o Redux e arquitetura flux? [Este curso](https://cursos.alura.com.br/course/redux-toolkit-explorando-arquitetura-flux) é o mais indicado para você aprender os principais conceitos e já sair gerenciando os estados de suas aplicações com o Redux Toolkit. Vá com tudo!

Bons estudos!

## ✅ O que aprendemos (Feito com IA)

# Aula 4 - Otimização e performance

## Projeto da aula anterior

Você pode ir acompanhando o passo a passo do desenvolvimento do nosso projeto e, caso deseje, \*\*[você pode acessar o site do Github](https://github.com/NeiltonSeguins/4487-react-hooks/tree/aula-03)\*\* ou \*\*[baixar o projeto da aula anterior](https://github.com/NeiltonSeguins/4487-react-hooks/archive/refs/heads/aula-03.zip)\*\*.

Bons estudos!

## ✅ Vídeo 4.1 - Memoizando funções com useCallback

## ✅ Atividade de alternativa: Otimizando a busca de livros (Feito com IA)

## ✅ Vídeo 4.2 - Realizando cálculos pesados com o useMemo

## ✅ Atividade de alternativa: Gerenciamento eficiente de organogramas (Feito com IA)

## Para saber mais: Performance em aplicações React

A performance em aplicações React pode ser impactada por re-renderizações desnecessárias e carregamentos ineficientes. Vimos que hooks como useCallback e useMemo ajudam a otimizar cálculos e funções, evitando recriações desnecessárias.

Mas existem outras estratégias para melhorar a performance de suas aplicações. O [SSR (Server-Side Rendering)](https://www.alura.com.br/artigos/vite-ssr-react) melhora o tempo de carregamento inicial e a experiência do usuário, entregando conteúdo mais rapidamente usando menos JavaScript do lado do cliente. Além disso, ferramentas como o [**New Relic**](https://newrelic.com/pt) permitem monitorar e identificar gargalos de desempenho.

Quer aprender como aplicar essas estratégias e deixar seu React voando? Confira [esta formação completa sobre performance em aplicações React](https://cursos.alura.com.br/formacao-react-maximizando-performance-aplicacoes) e me agradece depois (risos).

## **## Para ir mais fundo**

Se quiser entender sobre SSR, CSR e SSG que são siglas que basicamente se referem a alguma forma de entregar o conteúdo de páginas na web para o usuário, eu recomendo a leitura deste [artigo](https://www.alura.com.br/artigos/csr-ssr-ssg) maravilho escrito pela Nayanne. Vai lá conferir!  
  
Bons estudos!

## Mão na massa: useReducer e Context API

Nesta aula, aprendemos sobre os hooks useCallback e useMemo. Mas como não temos motivos reais para usarmos eles em nosso projeto, vamos praticar um outro hook que vimos na aula passada, o useReducer. O useReducer nos permite definir um *reducer* que lida com as diferentes ações que podem ser executadas sobre o estado. Essas ações são disparadas por uma função retornada pelo hook chamada de dispatch.

Agora é sua vez de praticar! Use o poder do hook useReducer para gerenciar e lidar com o estado no contexto de Tarefas. Temos muitas funções que alteram o estado e esta é uma ótima oportunidade de você botar a mão na massa e entender como o useReducer funciona.

Siga o passo a passo:

1. **Defina o *reducer* para gerenciar as ações relacionadas às tarefas:**
   1. Implemente uma função que receba o estado atual e uma ação, retornando o novo estado com base no tipo de ação.
2. **Atualize o contexto para fornecer o estado e o *dispatch* aos componentes:**
   1. Utilize o createContext para criar um contexto que armazenará o estado das tarefas e a função dispatch.
3. **Implemente os componentes que consumirão o contexto e despacharão ações:**
   1. Desenvolva componentes como o formulário de adição de tarefas e a lista de tarefas, que utilizarão o estado e as funções fornecidas pelo contexto.

Abaixo está o passo a passo detalhado de como você pode realizar a atividade, com comentários explicativos no código!

**Opinião da pessoa instrutora:**

Já que nossa aplicação não necessita de um controle rigoroso de performance, com memoizações de funções ou cálculos pesados, vamos praticar o useReducer, hook que vimos na aula passada e ainda não praticamos em nosso projeto. Vem comigo!

## ## Definindo o reducer

O primeiro passo é criar uma função redutora, o reducer. Esta função vou colocar em um arquivo em `src/reducer/tarefasReducer.js`:

```js

export const estadoInicial = {

lista: [],

};

export const tarefasReducer = (estado, acao) => {

switch (acao.tipo) {

case "ADICIONAR\_TAREFA":

return {

...estado,

lista: [

...estado.lista,

{

id: Math.floor(Math.random() \* 100000) + Date.now(),

texto: acao.payload,

concluida: false,

},

],

};

case "DELETAR\_TAREFA":

return {

...estado,

lista: estado.lista.filter((tarefa) => tarefa.id !== acao.payload),

};

case "DELETAR\_TODAS":

return { ...estado, lista: [] };

case "DELETAR\_CONCLUIDAS":

return { ...estado, lista: estado.lista.filter((t) => !t.concluida) };

case "MARCAR\_CONCLUIDA":

return {

...estado,

lista: estado.lista.map((tarefa) =>

tarefa.id === acao.payload ? { ...tarefa, concluida: !tarefa.concluida } : tarefa,

),

};

default:

return estado;

}

};

```

Aproveitei e já defini o estado inicial que iremos usar, neste caso, o estado é uma lista que sabemos que é uma lista de tarefas. Toda aquela lógica de funções para adicionar, remover, deletar e marcar tarefas como concluídas está aqui nesta função reducer.

E só recapitulando o que essa função faz: ela recebe um estado e uma ação. Esta ação normalmente é um objeto e tem o tipo de ação a ser executada e pode vir com um payload, que é um dado que nos permitirá atualizar o estado de forma específica. O tipo é usado para definir que ação será executada, e neste caso deve ser exatamente igual ao que tem em cada case do switch. Tranquilo?

Agora vamos atualizar o contexto.

## ## Atualizando o contexto de Tarefas

Já que toda a lógica de atualização do estado de tarefas está na função redutora, o reducer, o contexto fica bem mais simplificado. Vejamos:

```jsx

import { createContext, useReducer, useState } from "react";

import { estadoInicial, tarefasReducer } from "../reducers/tarefasReducer";

export const TarefasContext = createContext();

const TarefasProvider = ({ children }) => {

const [estado, dispatch] = useReducer(tarefasReducer, estadoInicial);

const [exibirFormulario, setExibirFormulario] = useState(false);

const aoExibirFormulario = (valor) => {

setExibirFormulario(valor);

};

return (

<TarefasContext.Provider value={{ estado, dispatch, exibirFormulario, aoExibirFormulario }}>

{children}

</TarefasContext.Provider>

);

};

export default TarefasProvider;

```

Agora passamos ao value do provider o estado, o dispatch e formas de controlar a exibição do formulário de adicionar tarefas. Estamos diante de um contexto mais enxuto, que não precisa saber como alterar o seu estado, pois agora isso é problema do reducer. Mas isso é bom, nosso código fica mais limpo e organizado. Imagine que você tenha que adicionar uma funcionalidade de editar tarefa, você precisaria mexer apenas na função redutora e só. Isso é maravilhoso do ponto de vista de manutenção.

## Atualizando os componentes

Muito bem! Com o contexto organizado precisamos fazer algumas modificações em nossos componentes que dependem desse estado, como o ListaTarefas, FormularioTarefas e MenuOpcoes. Começando pelo ListaTarefas:

### ListaTarefas

```jsx

import { useContext } from "react";  
import Botao from "../Botao";  
import styles from "../styles.module.css";  
import checkImg from "/src/assets/imgs/check-task.png";  
import trashImg from "/src/assets/imgs/trash.svg";  
import { TarefasContext } from "../../../context/TarefasContext";  
  
const ListaTarefas = () => {  
 const { estado, dispatch } = useContext(TarefasContext);  
 return (  
 <ul className={styles["tasks\_\_task-list"]}>  
 {estado.lista.map((tarefa) => (  
 <li  
 key={tarefa.id}  
 className={`${styles["tasks\_\_task-list-item"]} ${tarefa.concluida ? styles["tasks\_\_task-list-item-complete"] : ""}`}  
 >  
 <Botao  
 icone={checkImg}  
 id="marcar-tarefa-como-concluida"  
 className={`${styles["check-task\_\_button"]} ${tarefa.concluida ? styles["check-task\_\_button--complete"] : ""}`}  
 onClick={() => dispatch({ tipo: "MARCAR\_CONCLUIDA", payload: tarefa.id })}  
 />  
 <p className={styles["tasks\_\_task-list-item-description"]}>{tarefa.texto}</p>  
 <Botao  
 icone={trashImg}  
 id="deletar-tarefa"  
 className={`${styles["delete-task\_\_button"]}`}  
 onClick={() => dispatch({ tipo: "DELETAR\_TAREFA", payload: tarefa.id })}  
 />  
 </li>  
 ))}  
 </ul>  
 );  
};  
  
export default ListaTarefas;

```

O grande lance aqui é que só precisamos do estado e do dispatch do contexto. O estado contém a lista de tarefas que usamos para fazer um `.map()` e exibir a lista de tarefas. Já o dispatch precisamos chamar como retorno de uma função de callback dentro do onClick dos botões.

Chamamos o dispatch, passando um objeto com o tipo e o payload, que em ambos os casos é uma `tarefa.id`.

### FormularioTarefas

```jsx

import { useContext, useState } from "react";

import CampoTexto from "../CampoTexto";

import RodapeTarefas from "../RodapeTarefas";

import { TarefasContext } from "../../../context/TarefasContext";

import styles from "../styles.module.css";

const FormularioTarefas = () => {

const { exibirFormulario, aoExibirFormulario, dispatch } = useContext(TarefasContext);

const [texto, setTexto] = useState("");

return (

<form

className={`${styles["form-add-task"]} ${!exibirFormulario ? styles["form-add-task--hidden"] : ""}`}

aria-hidden={!exibirFormulario}

>

<CampoTexto

value={texto}

onChange={(e) => setTexto(e.target.value)}

etiqueta="Adicionando tarefa"

placeholder="No que você está trabalhando?"

/>

<RodapeTarefas

aoSalvar={() => {

dispatch({ tipo: "ADICIONAR\_TAREFA", payload: texto });

setTexto("");

aoExibirFormulario(false);

}}

aoCancelar={aoExibirFormulario}

/>

</form>

);

};

export default FormularioTarefas;

```

Neste exemplo, o payload é um texto. O estado de texto onde o usuário digita a descrição da tarefa que quer adicionar. Ao chamar a função aoSalvar no RodapeTarefas, também limpamos o campo de texto da tarefa e fechamos o form. Você também pode abstrair isso para uma função e chamar ela aqui para ficar mais organizado. Fica a dica!

### MenuOpcoes

```jsx

import styles from "../styles.module.css";  
import moreImg from "/src/assets/imgs/more.svg";  
import checkImg from "/src/assets/imgs/check.svg";  
import trashImg from "/src/assets/imgs/trash.svg";  
import Botao from "../Botao";  
import { TarefasContext } from "../../../context/TarefasContext";  
import { useContext } from "react";  
  
const MenuOpcoes = () => {  
 const { dispatch } = useContext(TarefasContext);  
 return (  
 <div className={styles["dropdown"]}>  
 <Botao icone={moreImg} className={styles["dropdown\_\_button-more"]} />  
  
 <ul className={styles["dropdown\_\_ul"]}>  
 <li className={styles["dropdown\_\_li"]}>  
 <Botao  
 icone={checkImg}  
 className={styles["dropdown\_\_button"]}  
 onClick={() => dispatch({ tipo: "DELETAR\_CONCLUIDAS" })}  
 id="btn-remover-concluidas"  
 >  
 Limpar tarefas concluídas  
 </Botao>  
 </li>  
 <li className={styles["dropdown\_\_li"]}>  
 <Botao  
 icone={trashImg}  
 className={styles["dropdown\_\_button"]}  
 onClick={() => dispatch({ tipo: "DELETAR\_TODAS" })}  
 id="btn-remover-concluidas"  
 >  
 Limpar todas as tarefas  
 </Botao>  
 </li>  
 </ul>  
 </div>  
 );  
};  
  
export default MenuOpcoes;

```

Neste componente só precisamos do dispatch. Com ele podemos chamar as funções para deletar todas as tarefas e deletar todas as tarefas concluídas.

Tranquilo? Com isso, aplicamos o useReducer para gerenciar melhor o estado de tarefas. Abstraímos a lógica de adição, deleção, marcar tarefa como concluída para a função redutora, mantendo-a aberta para novas ações deixando a aplicação mais escalável.

Se quiser conferiro o código completo do que fizemos nesta atividade, só [acessar este link](https://github.com/NeiltonSeguins/4487-react-hooks/commit/bafd9bbb59eff919b82d629bcc8f0d934a4c79bf) e conferir no github.

Bons estudos!

## ✅ O que aprendemos (Feito com IA)

# Aula 5 - Criando os próprios hooks

## Projeto da aula anterior

Você pode ir acompanhando o passo a passo do desenvolvimento do nosso projeto e, caso deseje, \*\*[você pode acessar o site do Github](https://github.com/NeiltonSeguins/4487-react-hooks/tree/aula-04)\*\* ou \*\*[baixar o projeto da aula anterior](https://github.com/NeiltonSeguins/4487-react-hooks/archive/refs/heads/aula-04.zip)\*\*.

Bons estudos!

## ✅ Vídeo 5.1 - Reaproveitando lógica com hooks customizados

## ✅ Atividade de alternativa: Vantagens hooks customizados (Feito com IA)

## Mão na massa: Utilizando hooks customizados

Nesta aula, aprendemos como criar um hook customizado para centralizar e reutilizar a lógica em nossos componentes. Podemos ter um hook para recuperar o contexto e eliminar a tarefa de ter que importar o TarefasContext ou o CronometroContext toda vez. Podemos criar um hook para salvar as tarefas no localStorage e assim não perder elas quando recarregar a página sem querer. São muitas possibilidades.

E agora é com você! Implemente hooks customizados na aplicação para reaproveitar lógica ou executar uma tarefa crucial como salvar os dados no localStorage.

Siga o passo a passo:

1. **Crie o hook customizado para gerenciar o estado**
   1. Crie um hook que encapsula a lógica de retornar o contexto.
2. **Substitua o uso do useContext pelo seu hook customizado**
   1. Importe o hook onde necessário e utilize-o para acessar o contexto.
3. **Crie um hook para armazenar as tarefas no localStorage**
   1. Crie um hook useLocalStorage para armazenar as tarefas no navegador.

Agora é com você, tente fazer antes de olhar o gabarito abaixo. É muito importante que você pratique o que aprendeu em vídeo, e esta é uma ótima oportunidade.

**Opinião da pessoa instrutora**

Vamos começar criando os hooks para encapsular a lógica de recuperar o contexto. Estes hooks tem apenas esta função, recuperam o contexto específico e retornam eles dentro dos componentes. Dentro de `src/hooks` vamos criar o :

## useTarefasContext.jsx

```jsx

import { useContext } from "react";

import { TarefasContext } from "../context/TarefasContext";

const useTarefasContext = () => {

const contexto = useContext(TarefasContext);

if (!contexto) {

throw new Error("useTarefasContext deve ser usado dentro de um TarefasProvider");

}

return contexto;

};

export default useTarefasContext;

```

E o `useCronometroContext.jsx`:

```jsx

import { useContext } from "react";

import { CronometroContext } from "../context/CronometroContext";

const useCronometroContext = () => {

const contexto = useContext(CronometroContext);

if (!contexto) {

throw new Error("useCronometroContext deve ser usado dentro de um CronometroProvider");

}

return contexto;

};

export default useCronometroContext;

```

Estes dois hooks são responsáveis por recuperar o e devolver o contexto de Tarefas e Cronometro, respectivamente.

Agora precisamos importá-los nos componentes que os utilizam:

### AppTarefas

```jsx

import useTarefasContext from "../../hooks/useTarefasContext";

const AppTarefas = () => {

const { exibirFormulario, aoExibirFormulario } = useTarefasContext();  
// ... resto do componente

```

### ListaTarefas

```jsx

import useTarefasContext from "../../../hooks/useTarefasContext";

const ListaTarefas = () => {

const { estado, dispatch } = useTarefasContext();

// ... resto do componente

```

## FormularioTarefas

```jsx

import useTarefasContext from "../../../hooks/useTarefasContext";

const FormularioTarefas = () => {

const { exibirFormulario, aoExibirFormulario, dispatch } = useTarefasContext();

// ... resto do componente

```

## MenuOpcoes

```jsx

import useTarefasContext from "../../../hooks/useTarefasContext";

const MenuOpcoes = () => {

const { dispatch } = useTarefasContext();

// ... resto do componente

```

Agora vamos utilizar o hook `useCronometroContext()` para recuperar o contexto de Cronometro.

## Cronometro

```jsx

import useCronometroContext from "../../hooks/useCronometroContext";

export default function Cronometro() {

const { musicaRef } = useCronometroContext();

// ... resto do componente

```

## Timer

```jsx

import useCronometroContext from "../../../hooks/useCronometroContext";

export default function Timer() {

const { tempoRestante } = useCronometroContext();

// ... resto do componente

```

## BotoesModo

```jsx

import useCronometroContext from "../../../hooks/useCronometroContext";

const BotoesModos = () => {

const { modos, modoAtual, selecionarModo } = useCronometroContext();

// ... resto do componente

```

## BotaoCronometro

```jsx

import useCronometroContext from "../../../hooks/useCronometroContext";

const BotaoCronometro = () => {

const { emExecucao, alternarCronometro } = useCronometroContext();

// ... resto do componente

```

E com isso a gente alterou em todos os lugares onde antes usávamos diretamente o useContext.

## Criando o hook useLocalStorage

Muito bem, e para finalizar a nossa aplicação vamos criar um hook que irá armazenar as tarefas no localStorage, para evitar que ao recarregar a página sem querer a gente perca as tarefas que já tinhamos adicionado antes.

Então em `src/hooks` eu crio o arquivo:

## useLocalStorage

```jsx

import { useState, useEffect } from "react";  
  
const useLocalStorage = (chave, valorInicial) => {  
 const [valor, setValor] = useState(() => {  
 const valorSalvo = localStorage.getItem(chave);  
 return valorSalvo ? JSON.parse(valorSalvo) : valorInicial;  
 });  
  
 useEffect(() => {  
 localStorage.setItem(chave, JSON.stringify(valor));  
 }, [chave, valor]);  
  
 return [valor, setValor];  
};  
  
export default useLocalStorage;

```

Este hook useLocalStorage permite armazenar e recuperar dados do localStorage do navegador. Ele recebe uma chave e um valor inicial, e retorna o valor armazenado (ou inicial) e uma função para atualizar o valor, que também atualiza o localStorage.

Então uma lógica que você poderia ter espalhada nos seus componentes ou contexto, agora é encapsulada e compartilhada através deste hook.

E para usar este hook precisamos ir no TarefasContext e fazer a seguinte alteração:

```jsx

import { createContext, useEffect, useReducer, useState } from "react";  
import { estadoInicial, tarefasReducer } from "../reducers/tarefasReducer";  
import useLocalStorage from "../hooks/useLocalStorage";  
  
export const TarefasContext = createContext();  
  
const TarefasProvider = ({ children }) => {  
 const [tarefasSalvas, setTarefasSalvas] = useLocalStorage("tarefas", estadoInicial);  
 const [estadoTarefas, dispatch] = useReducer(tarefasReducer, tarefasSalvas);  
 const [exibirFormulario, setExibirFormulario] = useState(false);  
  
 useEffect(() => {  
 setTarefasSalvas(estadoTarefas);  
 }, [estadoTarefas, setTarefasSalvas]);  
  
 const aoExibirFormulario = (valor) => {  
 setExibirFormulario(valor);  
 };  
  
 return (  
 <TarefasContext.Provider  
 value={{ estado: estadoTarefas, dispatch, exibirFormulario, aoExibirFormulario }}  
 >  
 {children}  
 </TarefasContext.Provider>  
 );  
};  
  
export default TarefasProvider;

```

O useLocalStorage inicializa o estado das tarefas com dados do localStorage ou um estado inicial padrão. Ele mantém o estado da aplicação sincronizado com o localStorage, garantindo que as alterações nas tarefas sejam persistidas e carregadas mesmo após o navegador ser fechado e reaberto. Isso permite que as tarefas permaneçam disponíveis para o usuário.

E para que sua aplicação fique com os valores de minutos certinhos do pomodoro padrão, atualize a constante `modos` dentro do CronometroContext.jsx:

```jsx

const modos = [

{

id: "foco",

nome: "Foco",

duracao: 25 \* 60, // em minutos

},

{

id: "pausa\_curta",

nome: "Pausa curta",

duracao: 5 \* 60, // em minutos

},

{

id: "pausa\_longa",

nome: "Pausa longa",

duracao: 15 \* 60, // em minutos

},

];

```

E é isso! Finalmente sua aplicação está com todas as funcionalidades completas. Você consegue adicionar e deletar tarefas, marcar tarefas como concluídas, deletar todas as tarefas. Consegue iniciar o cronometro do pomodoro, navegar entre os modos de foco, pausa curta e pausa longa, além de controlar a música.

Tudo isso aplicando o conhecimento sobre os hooks essenciais aprendidos aqui como: useState, useEffect, useContext, useRef, useImperativeHandler, useReducer e hooks customizados. Maneiro, né?

Isso mostra como é importante saber realmente como usar os hooks, como eles funcionam e quando faz sentido usar um hook ou outro, ou até mesmo criar seu próprio hook customizado.

Bem, já que finalizamos, se você quiser conferir o código completo desta etapa, [acesse este link do github](https://github.com/NeiltonSeguins/4487-react-hooks/commit/5d9c13cf6d129f39de076314a4446a21221bf050) e aproveite.

Grande abraço e bons estudos!

## Para saber mais: Explorando hooks customizados

Os hooks como useState, useEffect e useContext são muito usados no React mas servem para coisas específicas. Algum você estava desenvolvendo suas aplicações e pensou: “Puxa, seria legal ter uma forma de compartilhar essa lógica com todos os componentes...”. Isto é legal de se pensar, pois quando precisamos de lógica reutilizável em vários componentes podemos pensar em **hooks customizados**!

## **## Quando usar hooks customizados?**

Já conversamos sobre isso, então vamos dar uma breve revisada:

1. Precisa reutilizar lógica em vários componentes.
2. Quer organizar melhor efeitos colaterais (useEffect).
3. Deseja encapsular e compartilhar lógica complexa com vários componentes.

Agora, vamos ver alguns exemplos de hooks customizados que podem ser úteis!

## **## Exemplos de hooks customizados**

Agora vamos ver alguns hooks úteis que você pode precisar no algum dia.

### **1. Salvando Dados no localStorage**

O localStorage permite armazenar dados no navegador de forma persistente. Vamos criar um hook para gerenciar valores no localStorage:

```jsx

import { useState, useEffect } from "react";  
  
function useLocalStorage(key, initialValue) {  
 const [value, setValue] = useState(() => {  
 const storedValue = localStorage.getItem(key);  
 return storedValue ? JSON.parse(storedValue) : initialValue;  
 });  
  
 useEffect(() => {  
 localStorage.setItem(key, JSON.stringify(value));  
 }, [key, value]);  
  
 return [value, setValue];  
}  
  
export default useLocalStorage;  
```

Uso do hook:

```jsx

function App() {  
 const [name, setName] = useLocalStorage("user\_name", "");  
  
 return (  
 <div>  
 <input value={name} onChange={(e) => setName(e.target.value)} />  
 <p>Nome salvo: {name}</p>  
 </div>  
 );  
}  
```

Esse hook mantém os dados salvos mesmo após atualizar a página!

### **2. Obtendo a localização do usuário**

Se você precisa acessar a localização do usuário via API do navegador, um hook customizado é a melhor forma de encapsular essa lógica:

```jsx

import { useState, useEffect } from "react";  
  
function useGeolocation() {  
 const [location, setLocation] = useState({ lat: null, lon: null, error: null });  
  
 useEffect(() => {  
 if (!navigator.geolocation) {  
 setLocation((prev) => ({ ...prev, error: "Geolocalização não suportada" }));  
 return;  
 }  
  
 const onSuccess = (position) => {  
 setLocation({  
 lat: position.coords.latitude,  
 lon: position.coords.longitude,  
 error: null,  
 });  
 };  
  
 const onError = (error) => {  
 setLocation((prev) => ({ ...prev, error: error.message }));  
 };  
  
 navigator.geolocation.getCurrentPosition(onSuccess, onError);  
 }, []);  
  
 return location;  
}  
  
export default useGeolocation;

```

Uso do hook:

```

function LocationComponent() {  
 const { lat, lon, error } = useGeolocation();  
  
 return (  
 <div>  
 {error ? <p>Erro: {error}</p> : <p>Latitude: {lat}, Longitude: {lon}</p>}  
 </div>  
 );  
}

```

Agora o componente consegue acessar a localização do usuário sem precisar repetir a lógica em vários lugares. Isto pode ser útil em aplicações de georeferenciamento ou uma alternativa mais simples ao uso das API’s de mapas do Google!

### **3. Fazendo fetch de API**

Esse hook encapsula a lógica de chamadas de API:

```jsx

import { useState, useEffect } from "react";  
function useFetch(url, options = {}) {  
 const [data, setData] = useState(null);  
 const [loading, setLoading] = useState(true);  
 const [error, setError] = useState(null);  
  
 useEffect(() => {  
 let isMounted = true;  
  
 const fetchData = async () => {  
 try {  
 setLoading(true);  
 const response = await fetch(url, options);  
 if (!response.ok) throw new Error("Erro ao buscar os dados");  
 const result = await response.json();  
 if (isMounted) setData(result);  
 } catch (err) {  
 if (isMounted) setError(err.message);  
 } finally {  
 if (isMounted) setLoading(false);  
 }  
 };  
  
 fetchData();  
  
 return () => {  
 isMounted = false; // Evita atualização em componentes desmontados  
 };  
 }, [url, options]);  
  
 return { data, loading, error };  
}  
  
export default useFetch;

```

Uso do useFetch:

```jsx

function UsersList() {  
 const { data: users, loading, error } = useFetch("https://jsonplaceholder.typicode.com/users");  
  
 if (loading) return <p>Carregando...</p>;  
 if (error) return <p>Erro: {error}</p>;  
  
 return (  
 <ul>  
 {users.map((user) => (  
 <li key={user.id}>{user.name}</li>  
 ))}  
 </ul>  
 );  
}  
```

### **4. Validação de formulários**

Esse hook ajuda a gerenciar inputs e validar campos em formulários:

```jsx

import { useState } from "react";  
function useFormValidation(initialState, validate) {  
 const [values, setValues] = useState(initialState);  
 const [errors, setErrors] = useState({});  
 const [isSubmitting, setIsSubmitting] = useState(false);  
  
 const handleChange = (e) => {  
 const { name, value } = e.target;  
 setValues({ ...values, [name]: value });  
 };  
  
 const handleSubmit = (e) => {  
 e.preventDefault();  
 const validationErrors = validate(values);  
 setErrors(validationErrors);  
 setIsSubmitting(true);  
 };  
  
 return { values, errors, isSubmitting, handleChange, handleSubmit };  
}  
  
export default useFormValidation;

```

Uso do useFormValidation:

```jsx

function validateForm(values) {  
 let errors = {};  
 if (!values.email.includes("@")) errors.email = "E-mail inválido";  
 if (values.password.length < 6) errors.password = "Senha deve ter pelo menos 6 caracteres";  
 return errors;  
}  
  
function LoginForm() {  
 const { values, errors, handleChange, handleSubmit } = useFormValidation(  
 { email: "", password: "" },  
 validateForm  
 );  
  
 return (  
 <form onSubmit={handleSubmit}>  
 <input type="email" name="email" value={values.email} onChange={handleChange} placeholder="E-mail" />  
 {errors.email && <p>{errors.email}</p>}  
  
 <input type="password" name="password" value={values.password} onChange={handleChange} placeholder="Senha" />  
 {errors.password && <p>{errors.password}</p>}  
  
 <button type="submit">Entrar</button>  
 </form>  
 );  
}

```

Você pode usar validações simples como essa do exemplo, ou usar bibliotecas de validação ou criação de schemas em formulários React, como [zod](https://zod.dev/) ou [React Hook Form](https://react-hook-form.com/).

## **## Para ir mais fundo**

Inclusive, se você deseja se aprofundar em validação de formulários em React, saiba que temos uma formação completinha que aborda as principais bibliotecas de validação de formulários e criação de schemas para aplicações React. Se quiser [conferir a formação completa, acesse este link](https://cursos.alura.com.br/formacao-react-bibliotecas-desenvolver-formularios).

Bons estudos!

## ✅ O que aprendemos (Feito com IA)

## Projeto final e portfólio

Parabéns! Você concluiu o curso sobre TypeScript com React! Agora, é hora de consolidar seu aprendizado e turbinar seu portfólio.

Você pode [baixar](https://github.com/NeiltonSeguins/4487-react-hooks/archive/refs/heads/aula-05.zip) ou [acessar o código-fonte](https://github.com/NeiltonSeguins/4487-react-hooks/tree/aula-05) do projeto final para referência.

## ✅ Vídeo 5.2 - Conclusão