React Hooks

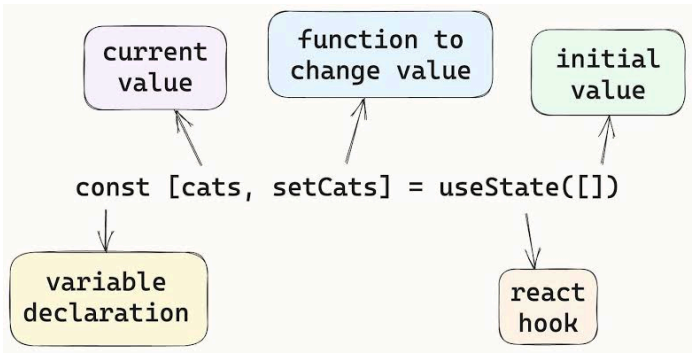
Link Notion: https://cherry-client-b8f.notion.site/React-Hooks-267911d84e0d80108950ff2f0d65b37a?pvs=73

React Hooks

No React, **hooks** são ganchos que você usa para “puxar” funcionalidades extras dentro de um componente.

Sem eles, um componente React seria só uma função que devolve HTML (JSX).

UseState



Quando começamos a trabalhar com React, precisamos de uma forma de guardar informações que mudam durante a execução do programa. Essas informações são chamadas de **estado**.

O **useState** é o hook (função especial do React) que nos permite criar e controlar esse estado dentro de componentes funcionais.

O que é o useState?

O useState é como uma caixa de memória que o React cria dentro do componente, guardando um valor e também fornecendo uma função para atualizar esse valor.

Sempre que o valor muda, o React desenha novamente o componente na tela para mostrar o valor atualizado.

Como é a estrutura do useState?

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

const [valor, setValor] = useState(valorInicial);
```

Para declarar um estado, utilizamos:

- valor → o estado que utilizaremos.
- setValor → é a função que usamos para atualizar esse estado.
- useState(valorInicial) → define qual será o valor inicial do estado.

Importante: O React sempre redesenha o componente quando usamos a função de atualização (setValor).

O exemplo mais utilizado em tutoriais é do contador, que permite aumentar um número ao clicar no botão, mostrando a diferença entre uma variável comum(que não gera uma atualização na tela) e um estado, que gera uma atualização completa do componente.

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

function Contador() {
  // Criamos um estado chamado "contador" que começa em 0
  const [contador, setContador] = useState(0);

  return (
    <div>
```

```
      <p>Valor atual: {contador}</p>
      <button onClick={() => setContador(contador + 1)}>+1</button>
      <button onClick={() => setContador(contador - 1)}>-1</button>
    </div>
  );
}
```

Como funciona:

1. O useState(0) cria um estado chamado contador e inicia ele em 0.
2. Quando clicamos em +1, chamamos setContador(contador + 1).
3. O React atualiza o valor e re-renderiza o componente.
4. O texto <p> é atualizado automaticamente.

Indo além

Você pode criar diversos estados, para todos as variáveis que você desejar:

```
const App = () => {
  const [name, setName] = useState("John Doe");
  const [age, setAge] = useState(20);
  const [hobby, setHobby] = useState("reading");

  return (
    // ...
  );
};
```

Você também poderia combinar os estados dentro de um objeto e acessá-los posteriormente:

```
const App = () => {
  const [userDetails, setUserDetails] = useState({
    name: 'John Doe',
    age: 20,
    hobby: 'Reading',
  });

  return (
    <div>
      <h1>{userDetails.name}</h1>
      <p>
        {userDetails.age} || {userDetails.hobby}
      </p>
    </div>
  );
```

Você também pode iniciar um estado como uma função, por exemplo quando você precisa fazer alguns cálculos para determinar como ele deve iniciar:

```
const expensiveComputation = () => {
  let number = 50;
  let newNumber = (50 % 10) * 10 - number;
  return newNumber;
};

const App = () => {
  const [calc, setCalc] = useState(() => expensiveComputation());
  return (
```

```
      <div>
        <p>{calc}</p>
      </div>
    );
```

Regra para utilizar o useState

Você só pode chamar o useState() diretamente dentro do componente. Você não pode utilizá-lo dentro de uma função, loop, funções dentro do seu componente ou condicionais.

```
// Nunca faça isso ❌❌❌❌

if(condition){
    const [count, setCount] = useState()(0);
}

for (let index = 0; index < 25; index++) {
    let [count, setCount] = useState()(0);
}

const nestedFn = () => () => {
    const [count, setCount] = useState()(0);
}
```

useEffect

O **useEffect** é usado para lidar com efeitos colaterais, que são ações que acontecem fora do fluxo normal da renderização, como:

- Buscar dados em uma API
- Criar um timer (relógio, cronômetro)
- Escutar eventos do navegador (scroll, resize, etc.)

```
useEffect(() => {
  // código que roda depois da tela ser desenhada
}, [dependencias]);
```

Ele recebe uma função de callback, que será executada somente depois da tela ter sido toda montada na tela. Ele também recebe um parâmetro que controla quantas vezes será executado, chamado **array de dependências**. Ele pode ser:

- [] → roda só na montagem (primeira vez que aparece na tela).
- [variável] → roda sempre que a variável ou estado mudarem.
- sem nada → roda toda vez que o componente renderiza (quase nunca usamos assim).

useContext

Imagine que você tem um valor (ex: tema claro/escuro) que precisa ser usado em vários componentes. Em vez de passar props de pai para filho em cadeia, usamos o **Context API**. O **useContext** serve para pegar esse valor facilmente.

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

const TemaContext = createContext("claro");

function Botao() {
  const tema = useContext(TemaContext);
  return <button style={{ background: tema === "claro" ? "#fff" : "#333" }}>Clique</button>;
}
```

```jsx
function App() {
  return (
    <TemaContext.Provider value="escuro">
      <Botao />
    </TemaContext.Provider>
  );
}
```

useRef

O useRef cria uma caixinha mutável que não causa re-render quando muda. Muito usado para:

- Acessar elementos do DOM (ex: dar foco em um input).
- Guardar valores que não precisam atualizar a tela.

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

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

  function focar() {
    inputRef.current.focus(); // pega o elemento DOM
  }

  return (
    <div>
      <input ref={inputRef} placeholder="Digite algo" />
      <button onClick={focar}>Focar no input</button>
    </div>
  );
}
```

useMemo

Às vezes temos cálculos pesados (como somar muitos números ou processar dados). Se o componente renderiza várias vezes, o cálculo pode ser refeito sem necessidade.

O **useMemo** memoriza o resultado e só recalcula se alguma dependência mudar.

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

function Fatorial({ numero }) {
  const resultado = useMemo(() => {
    console.log("Calculando...");
    return numero <= 1 ? 1 : numero * (numero - 1);
  }, [numero]);

  return <p>Fatorial de {numero} = {resultado}</p>;
}
```

Ainda existem outros como **useCallback** e **useReducer**, que você pode conferir como funcionam nos links de referências.

Referências

https://www.alura.com.br/artigos/react-hooks

https://dev.to/pratham10/all-you-need-to-know-about-react-hooks-54p0

https://ui.dev/why-react-renders