O que são Hooks?

No React antigo, para usar estado e ciclo de vida, você precisava criar um **componente de classe**:

jsx
CopiarEditar

```
class MeuComponente extends React.Component {
  constructor(props) {
    super(props);
    this.state = { contador: 0 };
  }

  incrementar = () => {
    this.setState({ contador: this.state.contador + 1 });
  };

  render() {
    return (
      <div>
        <p>Contador: {this.state.contador}</p>
        <button onClick={this.incrementar}>Aumentar</button>
      </div>
    );
  }
}
```

Com Hooks, podemos fazer isso em **componentes funcionais** de forma muito mais simples:

jsx
CopiarEditar

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

const MeuComponente = () => {
  const [contador, setContador] = useState(0);

  return (
    <div>
      <p>Contador: {contador}</p>
```

```
      <button onClick={() => setContador(contador +
1)}>Aumentar</button>
    </div>
  );
};
```

Agora, vamos ver alguns dos **principais Hooks**!

🔹 1. `useState` → Gerenciar Estado

O `useState` permite que um componente funcional tenha estado interno.

📌 Exemplo:

jsx
CopiarEditar

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

const Contador = () => {
  const [numero, setNumero] = useState(0);

  return (
    <div>
      <p>Valor: {numero}</p>
      <button onClick={() => setNumero(numero +
1)}>Adicionar</button>
    </div>
  );
};

export default Contador;
```

✅ **`numero`** é o estado.
✅ **`setNumero`** altera o estado.
✅ **`useState(0)`** define o valor inicial como `0`.

🔹 2. `useEffect` → Efeitos Colaterais

O `useEffect` permite executar **código** quando algo muda no componente, como:

- Buscar dados de uma API.
- Atualizar o título da página.
- Adicionar/remover eventos do DOM.

📌 Exemplo:

jsx
CopiarEditar

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

const Relogio = () => {
  const [hora, setHora] = useState(new Date().toLocaleTimeString());

  useEffect(() => {
    const intervalo = setInterval(() => {
      setHora(new Date().toLocaleTimeString());
    }, 1000);

    return () => clearInterval(intervalo); // Limpa o intervalo ao
desmontar
  }, []); // [] faz rodar só uma vez ao montar

  return <h2>{hora}</h2>;
};

export default Relogio;
```

✅ O **`useEffect`** executa código **após a renderização**.
✅ O **array vazio `[]`** faz rodar **apenas uma vez**.
✅ O **return** limpa o intervalo quando o componente for desmontado.

🔹 3. `useContext` → Compartilhar Estado Global

O `useContext` permite compartilhar **dados globais** entre componentes sem precisar de **props drilling** (passar `props` de um componente para outro repetidamente).

📌 Exemplo:

jsx
CopiarEditar

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

// Criar contexto
const TemaContext = createContext();

// Componente Provedor (fornece o valor)
const TemaProvider = ({ children }) => {
  const [tema, setTema] = useState("claro");

  return (
    <TemaContext.Provider value={{ tema, setTema }}>
      {children}
    </TemaContext.Provider>
  );
};

// Componente que consome o contexto
const BotaoTrocarTema = () => {
  const { tema, setTema } = useContext(TemaContext);
  return (
    <button onClick={() => setTema(tema === "claro" ? "escuro" :
"claro")}>
      Alternar para {tema === "claro" ? "escuro" : "claro"}
    </button>
  );
};

// App usando o Provider
const App = () => (
  <TemaProvider>
    <BotaoTrocarTema />
  </TemaProvider>
);

export default App;
```

✅ O `createContext()` cria o **contexto global**.
✅ O `useContext(TemaContext)` acessa os **dados do contexto**.

🔹 4. `useRef` → Manipular Elementos DOM e Estados Persistentes

O `useRef` permite acessar elementos do DOM diretamente ou **armazenar valores** sem re-renderizar o componente.

📌 Exemplo: Focar automaticamente em um input

jsx
CopiarEditar

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

const Formulario = () => {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus(); // Foca no input ao carregar
  }, []);

  return <input ref={inputRef} placeholder="Digite algo..." />;
};

export default Formulario;
```

✅ **`useRef(null)`** cria uma referência.
✅ **`inputRef.current.focus()`** foca no input.

🔹 5. `useReducer` → Alternativa ao `useState` para Estados Complexos

O `useReducer` é útil quando temos estados mais complexos, como um carrinho de compras.

📌 Exemplo:

jsx
CopiarEditar

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

const reducer = (state, action) => {
  switch (action.type) {
```

```
    case "incrementar":
      return { contador: state.contador + 1 };
    case "decrementar":
      return { contador: state.contador - 1 };
    default:
      return state;
  }
};

const Contador = () => {
  const [state, dispatch] = useReducer(reducer, { contador: 0 });

  return (
    <div>
      <p>Contador: {state.contador}</p>
      <button onClick={() => dispatch({ type: "incrementar"
})}>➕</button>
      <button onClick={() => dispatch({ type: "decrementar"
})}>➖</button>
    </div>
  );
};

export default Contador;
```

✅ `useReducer(reducer, estadoInicial)`.
✅ `dispatch({ type: "incrementar" })` altera o estado.

🎯 Resumo dos Principais Hooks

Hook	O que faz?
`useState`	Gerencia estados simples.
`useEffect`	Executa efeitos colaterais (ex: buscar API, timers).
`useContext`	Compartilha estado entre componentes sem "props drilling".

`useRef`	Manipula o DOM ou armazena valores sem re-renderizar.
`useReducer`	Gerencia estados complexos, alternativa ao `useState`.