Hooks React

É possível usar os hooks apenas dentro de componentes funcionais.

O numero de vezes que um hook é executado / chamado, nunca pode mudar, ou seja, não pode colocar o hook dentro de um if ou função.

---

useState

Armazena valores (estados). Permite armazenar estado a componentes funcionais. Possui dois valores, o primeiro é uma variável que guarda o valor e o segundo é uma função que seta (altera) o state (variável). Dentro do useState é passado o valor inicial, uma string, numero, lista, objeto, etc.

O jeito correto de pegar um valor anterior e setar um novo valor baseado no antigo é colocar uma função dentro do setValor e utilizar como parametro um prevState e alterar ele.

É possível colocar um objeto com mais de um valor dentro do useState só que vai ser preciso retornar ele inteiro quando for alterar porque se a alteração ocorrer apenas em um valor ele não vai pegar o outro automaticamente. Normalmente não é utilizado desse jeito, se for preciso de dois ou mais valores guardados, será utilizado vários states separados para isso.

Importar: `import { useState } from “react”`

Declarar: const [valor, setValor] = useState(valorInicial)

Utilização: `onClick={setValor((prevState) => {prevStateAlterado})}`

Objeto Como Valor:

```
const [obj, setObj] = useState({count: 0, theme: “dark”})

setValor((prevState) => {
    return {
        ...prevState, count: prevState + 1, theme: “light”
    }
})
```

Exemplo:

```
import { useState } from 'react'

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

    return (
        <>
            <h1>{count}</h1>
            <button onClick={() => {setCount((prevState) => prevState +
            1)}}>Aumentar Valor</button>
            <button onClick={() => {setCount((prevState) => prevState -
            1)}}>Diminuir Valor</button>
        </>
    )
}

export default UseState
```

---

useEffect

Permite executar efeitos colaterais em componentes funcionais, ou seja, é usado quando é preciso executar algo quando alguma coisa acontece (muda).

Recebe dois parâmetros, uma função e uma lista opcional. A lista é o que o useEffect vai ficar de olho, sempre que essa lista for alterada a função vai ser executada. Se a lista estiver vazia, ele será executado apenas quando o componente iniciar, ou seja, quando a pagina é carregada. Se não tiver lista, ele será executado toda vez que o componente atualizar.

Basicamente o useEffect é executado sempre que o valor passado na lista mudar. É possível passar mais de um valor e fazer mais de um useEffect.

O useEffect não pode ser assíncrono, por isso, quando for necessário criar um fetch async o certo é criar uma função dentro dele e essa função poderá conter o fetch.

Com o useEffect é possível puxar valores de API diferente sempre que um botão for clicado, por exemplo, quando um botão post ser clicado o fetch será direcionado pro post, quando um botão comment ser clicado o fetch vai para os comentários, fazendo com que o valor mostrado mude.

Importar: `import { useEffect } from “react”`

Utilização: `useEffect(( ) => {}, [valorObservado])`

Utilização Com Async:

```
useEffect(( ) => {
    const fetchAsync = async () => {
        const response = await fetch(`url/${valorDinamico}`)
      const responseJSON = await response.json()
    }
    fetchAsync()
}, [valorDinamico])
```

Exemplo 1:

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

function UseEffect() {
    const [resourceType, setResourceType] = useState("Posts")

    useEffect(() => {
        console.log("resource type mudou")
    }, [resourceType])

    const changeResourceType = (resourceType) => {
        setResourceType(resourceType)
    }

  return (
    <div>
        <h1>{resourceType}</h1>
        <button onClick={() =>
        {changeResourceType("Posts")}}>Posts</button>
        <button onClick={() =>
```

```
        {changeResourceType("Comments")}}>Comments</button>
        <button onClick={() =>
        {changeResourceType("Todos")}}>Todos</button>
    </div>
  )
}

export default UseEffect
```

Exemplo 2:

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

function UseEffect() {
    const [resourceType, setResourceType] = useState("Posts")
    const [items, setItems] = useState([])

    useEffect(() => {
        fetch(`https://jsonplaceholder.typicode.com/${resourceType}`)
        .then(resp => resp.json())
        .then((data) => {
            setItems(data)
        })
    }, [resourceType])

    const changeResourceType = (resourceType) => {
        setResourceType(resourceType)
    }

  return (
    <div>
        <button onClick={() =>
        {changeResourceType("Posts")}}>Posts</button>
```

```
        <button onClick={() =>
        {changeResourceType("Comments")}}>Comments</button>
        <button onClick={() =>
        {changeResourceType("Todos")}}>Todos</button>

        {
           items.map((item, i) => (
              <p key={i}>
                 {item.title ? "Possui titulo: " + item.title : "Não Possui titulo:
                  " + item.id}
              </p>
           ))
        }
     </div>
  )
}

export default UseEffect
```

---

useRef

O useRef possui 3 formas de ser usado.

Importar: `import { useRef } from “react”`

Forma 1: Guarda quantas vezes um componente foi renderizado. Ele guarda um valor, mas quando é atualizado o componente não é renderizado novamente.

Recebe um valor inicial e o valor atual dele fica guardado no .current.

Vai ser útil para quando um valor precisar persistir durante todo o ciclo de vida do componente, mas não permitir que quando esse valor for alterado o componente seja renderizado novamente.

Utilizar:

```
const renders = useRef(0)

useEffect(( ) => {renders.current = renders.current + 1})

<p>Renders: {renders.current}</p>
```

Forma 2: A forma mais usada do useRef.

É possível usá-lo para referenciar elemento HTML, ou seja, também serve para armazenar um elemento HTML, esse elemento fica na propriedade .current do useRef.

Essa é a principal utilidade do useRef, pegar uma referencia do elemento e manipulá-lo, pegar propriedades, valores, etc.

Para selecionar o elemento, basta colocar um ref={} como atributo e dentro do ref ficará a variável que esta com o useRef.

Utilizar:

```
const paragrafoRef = useRef( )  ← O valor .current dele é o <p>
console.log(paragrafoRef.current)

<p ref={elementRef}></p>
```

Exemplo:

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


function UseRef() {
    const [name, setName] = useState("")
    const inputRef = useRef()

    function focusInput() {inputRef.current.focus()}

    return (
        <div>
            <input ref={inputRef} value={name} onChange={(e) =>
            setName(e.target.value)} />
            <p>oi meu nome é {name}</p>
            <button onClick={focusInput}>Focus Input</button>
        </div>
    )
}

export default UseRef
```

Forma 3: Também é usado para guardar o valor anterior de um state e guardá-lo.

Utilizar:

```
const previusState = useRef( )
useEffect(( ) => {previusState.current = state}, [state])
```

---

useReducer

É usado para gerenciar estado dentro dos componentes assim como o useState só que de uma maneira diferente

Ele retorna uma lista e recebe como primeiro valor dessa lista o state e o dispatch: `const [state, dispatch] = useReducer()`

O primeiro parametro é a função reducer, que faz a alteração do estado, recebe como parametro o state e o action. O state.valor é o valor recebido para alteração. O action.type é a condição para essa alteração:
`function reducer(state, action) {console.log(state.variável, action.type}`
`const [state, dispatch] = useReducer(reducer, {variável: valor})`

O dispatch vai no elemento e recebe um objeto com type como propriedade que se liga ao action.type para fazer a alteração. O dispatch é a função reducer:
`<button onClick={() => dispatch({type: "increment"})}>Aumentar Valor</button>`

Basicamente o type do dispatch é o action.type do reducer que é a condição de alteração, se o action.type for um valor ele faz uma alteração, se é outro ele faz uma alteração diferente.

É possível usar o usereducer e usestate juntos
o dispacth pode receber mais que o type, qualquer coisa que quiser
o action armazena todos os valor do dispatch: action.type, action.payload. é uma convenção chamar o segundo parametro do dispatch de payload:
dispacth({type: “valor1”, payload: valorDinamico})

Quando usar o useReducer e o useState: Use o useReducer quando o state for muito complexo, grande ou quando o valor do state depender de outros valores, ou seja, quando varias propriedades do state dependerem uma da outra

importar: `import { useReducer } from “react”`

Utilizar:

```
function reducer(state, action) {console.log(state.variável, action.type)}

const [state, dispatch] = useReducer(reducer, {variável: valor})

<button onClick={() => dispatch({type: “increment”})}>Aumentar Counter</button>
```

Exemplo 1:

```
function reducer(state, action) {
   switch(action.type) {
      case ‘increment’:
         return {counter: state.counter + 1}
       case ‘decrement’:
          return {counter: state.counter -  1}
      default:
         return state
   }
}
```

```
const [state, dispatch] = useReducer(reducer, {counter: 0})

<p>{state.counter}</p>
<button onClick={() => dispatch({type: “increment”})}>Aumentar Counter</button>
<button onClick={() => dispatch({type: “decrement”})}>Diminuir Counter</button>
```

Exemplo 2: Com ele tambem é possível guardar varias tarefas dentro do state e mostrar elas na tela.

```
import { useReducer, useState } from "react"

const reducer = (state, action) => {
    switch (action.type) {
        case 'add-task':
        return {
            tasks: [...state.tasks, {name: action.payload, isCompleted: false}]
        }
        default:
            return state
    }
}

function UseReducer() {
    const [state, dispatch] = useReducer(reducer, {tasks: []})

    const [inputValue, setInputValue] = useState('')

    return (
        <div>
```

```
            <input value={inputValue} onChange={e =>
            setInputValue(e.target.value)} />
            <button onClick={() => {
                dispatch({type: "add-task", payload: inputValue})
                setInputValue("")
                }}>
                Adicionar
            </button>

            { state.tasks.map(task => <p>{task.name}</p>)}
        </div>
    )
}

export default UseReducer
```

Exemplo 3:

```
import { useReducer, useState } from "react"

const reducer = (state, action) => {
    switch (action.type) {
        case 'add-task':
        return {
            state,
            tasks: [...state.tasks, {name: action.payload, isCompleted: false}]
        }
        case 'toggle-task':
            return {
                ...state,
                tasks: state.tasks.map((item, index) => index ===
action.payload ? {...item, isCompleted: !item.isCompleted} : item)
            }
```

```
        default:
            return state
    }
}

function UseReducer() {
    const [state, dispatch] = useReducer(reducer, {tasks: []})

    const [inputValue, setInputValue] = useState('')

    return (
        <div>
            <input value={inputValue} onChange={e =>
setInputValue(e.target.value)} />
            <button onClick={() => {
                dispatch({type: "add-task", payload: inputValue})
                setInputValue("")
                }}>
                Adicionar
            </button>

            {
                state.tasks.map((task, index) => <p onClick={() =>
dispatch({type: 'toggle-task', payload: index})}
                style={{textDecoration: task.isCompleted ? 'Line-through' :
'none'}}>{task.name}</p>)
            }
        </div>
    )
}

export default UseReducer
```

---

useContext

É usado em conjunto com o context API.

Context API: Com o context API é possível passar props para uma arvore inteira de componentes.

Importar: `import { createContext } from “react”`

Exemplo:

theme-context.jsx

```
import { createContext, useState } from "react"

export const ThemeContext = createContext({theme: 'light',
toggleTheme: () => {}})

const ThemeContextProvider = ({ children }) => {
    const [theme, setTheme] = useState('light')

    const toggleTheme = () => {
        if(theme === 'light') {
            return setTheme('dark')
        }

        return setTheme('light')
    }

    return (
```

```
        <ThemeContext.Provider value={{ theme, toggleTheme }}>
            {children}
        </ThemeContext.Provider>
    )
}

export default ThemeContextProvider
```

App.jsx

```
<ThemeContextProvider>
      <UseState />
</ThemeContextProvider>
```

useContext: O useContext usa um objeto de variavel que recebe todos os valores passados por propriedade pelo ThemeContext.provider.

Importar: `import { useContext } from “react”`

Utilizar: `const {variavel1, variavel2} = useContext(context)`

Exemplo:

App.jsx

```
<ThemeContextProvider>
      <UseContext />
</ThemeContextProvider>
```

theme-context.jsx

```
import { createContext, useState } from "react"

export const ThemeContext = createContext({theme: 'light',
toggleTheme: () => {}})
```

```
const ThemeContextProvider = ({ children }) => {
    const [theme, setTheme] = useState('light')

    const toggleTheme = () => {
        if(theme === 'light') {return setTheme('dark')}

        return setTheme('light')
    }

    return (
        <ThemeContext.Provider value={{ theme, toggleTheme }}>
            {children}
        </ThemeContext.Provider>
    )
}

export default ThemeContextProvider
```

UseContext.jsx

```
import { useContext } from "react"
import { ThemeContext } from "./theme-context"

function UseContext() {
    const {theme, toggleTheme} = useContext(ThemeContext)

    return (
        <div
            style={{padding:20, borderRadius: 10, backgroundColor: theme
            === 'light' ? "#eee" : "#333", color: theme === 'dark' ?
            "#eee" : "#333"}}
        >
```

```
            <h1>Current Theme: {theme}</h1>

            <button onClick={() => toggleTheme()}>Toggle
            Theme</button>

        </div>
    )
}

export default UseContext
```

---

useMemo

O useMemo é utilizado quando é preciso otimizar a performance da aplicação, quando um problema de performance esta sendo notável no projeto. Quando uma função esta dando problema.

Recebe dois argumentos, uma função e uma lista de dependências. A lista de dependências é todos os argumentos que a função (que esta dentro do primeiro argumento, que é uma função também) receber.

Não é indicado usar o useMemo para tudo, porque ele vai guardar o valor da função em memoria, se muitas coisas forem guardadas, a memoria vai ficar cheia de coisas desnecessarias.

importar: `import { useMemo } from “react”`

Utilizar: `const função = useMemo(() => {return funçãoLenta(arg)}, [arg])`

---

useCallback

Igual ao useMemo é usado para resolver problemas de performances. A diferença é que o useCallback guarda em memoria a função em si, enquanto o useMemo guarda o retorno da função.

Recebe dois parametros, uma função e uma lista de dependências.
A lista de dependências recebe tudo o que é necessário ficar de olho.
Dentro da função fica o código que esta dando problemas.
Só quando o que estiver na lista de dependências mudar que o código dentro da função va iser executado, não permitindo fazer execuções desnecesarias.

Com ele é possível guardar uma função em memoria que só é recriada / executada quando o que esta dentro da lista de dependências mudar.

O callback será utilizado quando uma função tiver um custo grande de performance (tipo um fetch), tiver sendo executada desnecessariamente e tiver sendo passada como prop para outro componente.

Importar: `import { useCallback } from “react”`

Utilizar: const função = useCallback(() => {}, [])

---

useLayoutEffect

Funciona do mesmo jeito que o useEffect, só que ao contrario do useEffect que é executado depois do DOM ser montado, ele é executado antes.

O useLayoutEffect é usado quando a alteração é feita no DOM se baseando no DOM. Por exemplo, alterar o top de um pop up se baseando

no bottom de um button. Também se for preciso que um código / alteração / lógica seja feito antes que o DOM (return do componente) seja montado.

É um hook usado poucas vezes.

Importar: `import { useLayoutEffect } from “react”`

Utilizar: `useLayoutEffect(( ) => {}, [])`

Link Do Vídeo

https://encr.pw/wK4nR

---

useImperativeHandle

É um hook utilizado para customizar a exposição de uma referencia de um componente filho para seu componente pai, ou seja, permite personalizar os valores expostos por um ref quando usado com o forwardRef.

É preciso utilizar em conjunto o forwardRef, que expõem as referencias de um componente filho para um componente pai.

Quando é usado o forwardRef, o react expõe o DOM ou o componente inteiro e todas as referencias. Com o useImperativeHandle é possível controlar exatamente o que será exposto, como métodos específicos que o componente pai pode chamar diretamente.

Utilizando o forwardRef:

App.jsx

```
const ref = useRef(null)
<Input ref={ref} />
```

Input.jsx

```
const Input = forwardRef(function Input(props, ref)) {
    return <input ref={ref} />
}
```

Sem o forwardRef não é possível manipular a referencia no componente pai.

Importar: `import { useImperativeHandle } from “react`

Utilizar:

```
useImperativeHandle(ref, () => ({
    metodo1: () => {refVar.current.metodo1()},
    metodo2: () => {refVar.current.metodo2()}
}))

useImperativeHandle(ref, () => {
    return {
        metodo() {refVar.current.metodo()},
    }
})

<button onClick={() => varRef.current.método()}>
    Ativar Função
</button>
```

```
const ref = useRef()
function  handleClick() {
    ref.current.metodo()
}
```

```
<button onClick={handleClick}>
   Ativar Função
</button>
```

Exemplo 1:

Input.jsx

```
import { useImperativeHandle, useRef, forwardRef } from “react”

const myInput = forwardRef((props, ref) => {
   const inputRef = useRef()

   useImperativeHandle(ref, () => {
      focus: () => {inputRef.current.focus()}
   })

   return <input ref={inputRef} />
})
```

App.jsx

```
const inputRef = useRef()

return (
   <>
      <Input ref={inputRef} />
       <button onClick={() => inputRef.current.focus()}>
          Focus Input
      </button>
   </>
)
```

Exemplo 2:

App.jsx

```
import { forwardRef, useRef, useState } from “react”

function App() {
const [customerName, setCustomerName] = useState(“”)
const [customers, setCustomers]  = useState([])

const ref = useRef(null)

function handleClick() {
   setCustomerName(“”)
   setCustomers([… customers, customerName])
   ref.current.focus()
}

   return (
      <form>
         <Input
            ref={ref}
            onChange={(e) => setCustomerName(e.target.value)}
            value={customerName}
         />

         <button onClick={handleClick}>Add</button>
      </form>

      {customers.map((customer, idx) => {
         <p key={idx}>{customer}</p>})
      }
   )
```

```
}
```

Input.jsx

```
const Input = forwardRef(function Input(props, ref) {
    const inputRef = useRef(null)

    useImperativeHandle(ref, () => {
        return {focus() {inputRef.current.focus()}}
    }, [])

    return <input { ...props } ref={inputRef} />
})

export default Input
```

---

useDebugValue

É um hook usado para depuração e é útil principalmente na criação de hooks personalizados, pois permite exibir informações no React DevTools que ajuda a entender o estado interno dele.

Não retorna nada e é usado para exibir uma label em um hook customizado na extensão do react DevTools.

Importar: `import { useDebugValue } from “react”`

Utilizar:

```
useDebugValue(valor)
useDebugValue(valor, (valor1) => valor1 ? valor2 : valor3)
```

Exemplo:

App.jsx

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

function useOnlineStatus() {
  const [isOnline, setIsOnline] = useState(true);

  useDebugValue(isOnline ? 'Online' : 'Offline');

  useEffect(() => {
    const handleStatusChange = () => setIsOnline(navigator.onLine);
    window.addEventListener('online', handleStatusChange);
    window.addEventListener('offline', handleStatusChange);
    return () => {
      window.removeEventListener('online', handleStatusChange);
      window.removeEventListener('offline', handleStatusChange);
    };
  }, []);

  return isOnline;
}
```

---

useTransition

É usado quando é preciso evitar travamentos de UI durante operações pesadas e dar prioridade para interações imediatas, como digitação, empurrando o processamento mais pesado para segundo plano.

Retorna dois parâmetros:

isPending - É um booleano que indica se uma transição esta em andamento, ou seja, é true ou false.

startTransition - É uma função usada para marcar um bloco de código como transição, ou seja, é uma função que dentro dela vai ficar um código que pode demorar (como filtrar ou pegar uma lista gigante), enquanto esse código estiver executando, o isPending recebe true, quando esse código acaba de executar ele vira false. Esse código também vai ser marcado como não urgente, ou seja, outras coisas vão continuar executando enquanto ele estiver sendo executado.

Importar: `import { useTransition } from “react”`

Utilizar:

```
const [isPending, startTransition] = useTransition()

startTransition(() => {Código que pode demorar})

{isPending && <p>Loading...</p>}
```

Exemplo: Um fetch para uma lista gigante, enquanto esse fetch estiver acontecendo o isPending vai ser true, possibilitando uma verificação e se true um <Loading /> vai aparecer na tela.

---

useDeferredValue

Permite adiar a atualização de um valor para que a interface permaneça responsiva durante renderizações mais pesadas.

É usado quando tem um valor que muda rapidamente (como um input de texto), mas quer evitar que partes pesadas da interface sejam recalculadas a cada tecla digitada.

Ele retorna uma versão atrasada do valor original, que o react atualiza em segundo plano quando tiver tempo. Isso ajuda a evitar travamentos em atualizações pesadas.

O valor retornado não é garantido estar sempre sincronizado com o valor original e é ideal para melhorar a performance de renderizações pesadas, não para lógica critica.

Importar: `import { useDeferredValue } from “react”`

Utilizar: `const valorAdiado = useDeferredValue(valor)`

Exemplo:

App.jsx

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

function Busca() {
  const [input, setInput] = useState('');
  const inputAdiado = useDeferredValue(input);

  const resultados = pesquiseAlgoPesado(inputAdiado);

  return (
    <>
      <input value={input} onChange={e => setInput(e.target.value)} />
      <Resultados resultados={resultados} />
    </>
  );
}
```
