Objetivo:

I. Ciclo de vida de um componente React.

I. Ciclo de vida de um componente React

O ciclo de vida de um componente React descreve as fases pelas quais um componente passa desde a sua criação até ser removido do DOM. Esse ciclo está vinculado às fases de montagem (mounting – o componente é criado e inserido no DOM), atualização (updating – o componente é atualizado devido a alterações no state ou props) e desmontagem (unmounting – o componente é removido do DOM).

Podemos acessar essas fases usando o Hook de efeito colateral `useEffect`.

Fases do ciclo de vida do componente:

1. Montagem (mounting): o componente é criado e inserido no DOM. Para executar ações específicas durante essa fase, utilizamos o Hook `useEffect` com os seguintes parâmetros:

```
useEffect(() => {}, [])
```

- 1º parâmetro: uma função callback que será executada após o componente ser criado e inserido no DOM;
- 2º parâmetro: um array vazio, indicando que o `useEffect` será executado apenas uma vez, no momento da montagem.

Exemplo de montagem com useEffect:

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

export default function App(){
  useEffect(
    () => { console.log("useEffect: componente App montado"); },
    []
  );

  return <div>Boa noite</div>;
}
```

Explicação do código:

- O `useEffect` é executado apenas uma vez, quando o componente é montado;
- A mensagem "useEffect: componente App montado" será exibida no console assim que o componente for inserido no DOM.

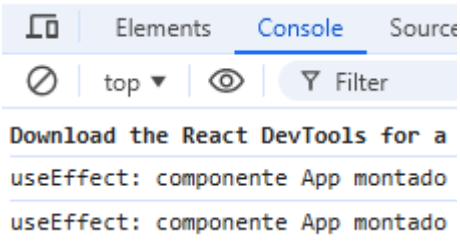
Observações:

- O `useEffect` é executado duas vezes devido ao uso do `StrictMode` no arquivo main.tsx. O StrictMode é uma ferramenta do React que ajuda a identificar potenciais

Resultado do código no console:

problemas na aplicação durante o desenvolvimento, como efeitos colaterais inesperados ou más práticas no código;

- O `StrictMode` não afeta o comportamento da aplicação em produção. Ele é ativado apenas em ambiente de desenvolvimento para fornecer avisos úteis;
- Ao testar o comportamento do `useEffect`, recomenda-se remover temporariamente o `StrictMode` para evitar chamadas duplicadas desnecessárias.




```
createRoot(document.getElementById('root')!).render(
  <StrictMode>
    <App />
  </StrictMode>,
);
```

2. Atualização (updating): a fase de atualização ocorre quando há uma mudança no estado (state) ou nas propriedades (props) de um componente. Nessa fase, o React re-renderiza o componente para refletir as alterações. Para acessar essa fase, usamos o Hook `useEffect`, passando os seguintes parâmetros:

   `useEffect(() => {}, [dependências])`

   - 1º parâmetro: uma função callback que será executada sempre que alguma das dependências monitoradas for alterada;
   - 2º parâmetro: um array que contém as dependências monitoradas. Sempre que o valor de uma dessas dependências mudar, a função de callback será executada novamente. No exemplo a seguir, está sendo monitorado apenas o estado `age`.

   Explicação do código:

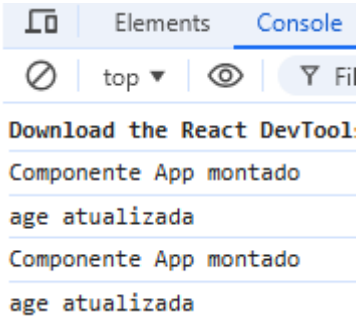
   - O primeiro `useEffect ([])` é chamado apenas na montagem do componente;
   - O segundo `useEffect ([age])` é chamado sempre que o estado age for atualizado.

   Observação: ao inicializar o estado com `useState`, uma atualização inicial pode ocorrer, chamando o `useEffect` relacionado.

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

export default function App() {
  const [name, setName] = useState("");
  const [age, setAge] = useState("");

  useEffect(() => {
    console.log("Componente App montado");
  }, []);
```

Resultado do código no console ao carregar:

```
  useEffect(
    () => {  console.log("age atualizada");  },
    [age]
  );

  return (
    <>
      <div>
        <label htmlFor="name">Nome</label>
        <input
          id="name"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </div>
      <div>
        <label htmlFor="name">Idade</label>
        <input id="age"
        value={age}
        onChange={(e) => setAge(e.target.value)}
        />
      </div>
    </>
  );
}
```



No exemplo a seguir está sendo monitorada a propriedade `name` no componente `Result`, ou seja, temos um componente filho `Result` que monitora a propriedade `name` passada pelo componente pai.

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

export default function App() {
  const [name, setName] = useState("");
  const [age, setAge] = useState("");

  return (
    <>
      <div>
        <label htmlFor="name">Nome</label>
        <input
          id="name"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </div>
      <div>
        <label htmlFor="name">Idade</label>
        <input id="age" value={age} onChange={(e) => setAge(e.target.value)} />
      </div>
```

```
      <Result name={name} age={age} />
    </>
  );
}

interface Person {
  name: string;
  age: string;
}


function Result(props:Person) {
  useEffect(() => {
    console.log("name atualizado");
  }, [props.name]);

  return (
    <>
      <div>Nome: {props.name}</div>
      <div>Idade: {props.age}</div>
    </>
  );
}
```

Explicação do código:

- O `useEffect` no componente `Result` é executado sempre que a propriedade `name` mudar;
- Alterar o campo de entrada `name` no componente pai faz com que o `useEffect` do componente filho `Result` seja disparado.

3. Desmontagem (unmounting): a fase de desmontagem ocorre quando um componente é removido do DOM. Para acessar essa fase, usamos o Hook `useEffect`, passando os seguintes parâmetros:

```
useEffect( () => { return () => {} }, [])
```

- 1º parâmetro: a função passada para o `useEffect` (em amarelo) deve retornar outra função (em ciano), que será executada antes do componente ser desmontado;
- 2º parâmetro: um array vazio garante que o `useEffect` será executado apenas na desmontagem do componente.

No exemplo a seguir, o componente `Result` é renderizado apenas quando o campo `name` contém algum valor. Caso o valor seja apagado, o componente é removido do DOM, disparando a fase de desmontagem.

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

export default function App() {
  const [name, setName] = useState("");
  const [age, setAge] = useState("");
```

```
  return (
    <>
      <div>
        <label htmlFor="name">Nome</label>
        <input
          id="name"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </div>
      <div>
        <label htmlFor="name">Idade</label>
        <input id="age" value={age} onChange={(e) => setAge(e.target.value)} />
      </div>
      {name !== "" && <Result name={name} age={age} />}
    </>
  );
}

interface Person {
  name: string;
  age: string;
}

function Result(props: Person) {
  useEffect(() => {
    console.log("Componente Result montado");

    return () => {
      console.log("componente desmontado");
    };
  }, []);

  return (
    <>
      <div>Nome: {props.name}</div>
      <div>Idade: {props.age}</div>
    </>
  );
}
```

II. Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Exercícios 1 e 2 - https://youtu.be/kw-jGocCrf8

Exercício 3 - https://youtu.be/KCicjtuZZDA

Exercício 1 – Complete o código a seguir para salvar, no localStorage do navegador, os valores dos estados name e age ao serem alterados, e para carregar esses valores quando o componente é montado.

Dicas:

- Use `localStorage.setItem("name", name)` para escrever no localStorage do navegador a chave (key) de nome `name`;
- Use `localStorage.getItem("name")` para ler o valor da chave `name` no localStorage.

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

export default function App() {
  const [name, setName] = useState("");
  const [age, setAge] = useState("");

  return (
    <>
      <div>
        <label htmlFor="name">Nome</label>
        <input
          id="name"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </div>
      <div>
        <label htmlFor="age">Idade</label>
        <input id="age" value={age} onChange={(e) => setAge(e.target.value)} />
      </div>
      <div>Nome: {name}</div>
      <div>Idade: {age}</div>
    </>
  );
}
```

Exercício 2 – Complete o código a seguir para salvar, no localStorage do navegador, os valores fornecidos pelo usuário ao clicar no botão Salvar e que esses valores sejam lidos do localStorage ao inicializar a aplicação.

Dicas:

- Ao montar o UserProvider, leia o localStorage e carregue no estado users;
- Na função add do UserProvider, escreva no localStorage;
- Será necessário usar JSON.stringfy(users) ao escrever o array e JSON.parse ao carregar no estado users o valor lido do localStorage.

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

export default function App() {
```

```
  return (
    <UserProvider>
      <Form />
      <List />
    </UserProvider>
  );
}

interface ContextProps {
  users: User[];
  add: (user: User) => void;
}

interface User {
  name: string;
  age: string;
}

interface ChildrenProps {
  children: React.ReactNode;
}

const UserContext = createContext<ContextProps | null>(null);

function UserProvider({ children }: ChildrenProps) {
  const [users, setUsers] = useState<User[]>([]);

  function add(user: User) {
    setUsers((prev) => [...prev, user]);
  }

  return (
    <UserContext.Provider value={{ users, add }}>
      {children}
    </UserContext.Provider>
  );
}

// Hook customizado para consumir o contexto
function useUserContext() {
  const context = useContext(UserContext);
  if (!context) {
    throw new Error("useUserContext deve ser usado dentro de um UserProvider");
  }
  return context;
}

function Form() {
  const { add } = useUserContext();
```

```
  const [name, setName] = useState("");
  const [age, setAge] = useState("");

  return (
    <div>
      <div>
        <label htmlFor="name">Nome</label>
        <input
          id="name"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </div>
      <div>
        <label htmlFor="age">Idade</label>
        <input id="age" value={age} onChange={(e) => setAge(e.target.value)} />
      </div>
      <div>
        <button onClick={() => add({ name, age })}>Salvar</button>
      </div>
    </div>
  );
}

function List() {
  const { users } = useUserContext();

  return (
    <ol>
      {users.map((item, index) => (
        <li key={index}>
          {item.name} - {item.age}
        </li>
      ))}
    </ol>
  );
}
```

Exercício 3 – Complete o código a seguir para salvar, no localStorage do navegador, os botões clicados pelo usuário e que esses valores sejam lidos do localStorage ao inicializar a aplicação.

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

export default function App() {
  return (
    <SourceProvider>
      <div>
        <Button label="A" />
```

```
        <Button label="B" />
        <Button label="C" />
        <Button label="D" />
      </div>
      <List />
    </SourceProvider>
  );
}

interface ButtonProps {
  label: string;
}

interface ContextProps {
  sources: string[];
  add: (value: string) => void;
}

interface ChildrenProps {
  children: React.ReactNode;
}

const SourceContext = createContext<ContextProps | null>(null);

function SourceProvider({ children }: ChildrenProps) {
  const [sources, setSources] = useState<string[]>([]);

  function add(value:string){
    setSources((prev) => [...prev, value]);
  }

  return (
    <SourceContext.Provider value={{ sources, add }}>
      {children}
    </SourceContext.Provider>
  );
}

// Hook customizado para consumir o contexto
function useSourceContext() {
  const context = useContext(SourceContext);
  if (!context) {
    throw new Error("useSourceContext deve ser usado dentro de um SourceProvider");
  }
  return context;
}

function Button({ label }: ButtonProps) {
  const { add } = useSourceContext();
```

```
  return (
    <button style={buttonStyle} onClick={() => add(label)}>
      {label}
    </button>
  );
}

function List() {
  const { sources } = useSourceContext();

  return (
    <ol>
      {sources.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ol>
  );
}

const buttonStyle: CSSProperties = {
  backgroundColor: "#567196",
  color: "white",
  padding: "10px 20px",
  marginRight: "10px",
  fontSize: "16px",
  fontWeight: "bold",
  border: "none",
  borderRadius: "5px",
  cursor: "pointer",
};
```