Renderização condicional e validações

Link Notion: https://cherry-client-b8f.notion.site/Renderiza-o-condicional-e-valida-es-28dff80de82244b98c2ce340d4ea539b?pvs=73

A renderização condicional no React é uma técnica que permite renderizar componentes ou elementos JSX com base em determinadas condições, usando lógicas de JavaScript como `if` ou o operador ternário, por exemplo. Isso permite criar interfaces de usuário dinâmicas e interativas que se adaptam a mudanças no estado, como a exibição de uma mensagem de erro se os dados falharem ao carregar ou de um menu de usuário se ele estiver logado.. Ele pode ocorrer de duas formas:

If ternário

```
function Greeting({isLoggedIn}) {
  return (
    <div>
      {isLoggedIn ? (
        <h1>Welcome back!</h1>
      ) : (
        <h1>Please sign up.</h1>
      )}
    </div>
  );
}
```

Neste caso, se isLoggedIn for **igual a true**, o usuário receberá a frase **“Welcome back!”**, senão ele receberá a mensagem **“Please sign up”**.

&&

Essa é uma maneira mais simples de renderizar quando não há um else, chamada de **short-circuit**. Ele funciona avaliando o primeiro operando e, se a condição for atendida, a expressão ou elemento após o operador é processado.

```
function Greeting({isLoggedIn}) {
  return (
    <div>
      {isLoggedIn && <h1>Welcome back!</h1>}
    </div>
  );
}
```

Exemplo aplicado à um login simplificado

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

const App = () => {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  const handleLogin = () => {
    setIsLoggedIn(true);
  };

  const handleLogout = () => {
    setIsLoggedIn(false);
  };
```

```
  return (
    <div>
      {isLoggedIn ? (
        <div>
          <h1>Welcome Back!</h1>
          <button onClick={handleLogout}>Logout</button>
        </div>
      ) : (
        <div>
          <h1>Please Login</h1>
          <button onClick={handleLogin}>Login</button>
        </div>
      )}
    </div>
  );
};

export default App;
```

Colocando um spinner

Existem bibliotecas exclusivas para componentes de carregamento que podem ser inseridos enquantos os elementos da página não carregam, como o caso da UI Ball.

https://uiball.com/ldrs/

Validações

Ao trabalhar com dados dinâmicos, muitas vezes lidamos com informações imprevisíveis, como campos que existentes para alguns objetos e outros não e informações inválidas. Para isso há o conceito de **Programação Defensiva**.

A Programação defensiva é uma **técnica de desenvolvimento de software que envolve a antecipação e a prevenção de erros, garantindo que o programa continue funcionando de maneira confiável mesmo em situações inesperadas ou com dados inválidos**. Em vez de assumir que o código sempre será usado corretamente, o desenvolvedor implementa verificações e salvaguardas para lidar com falhas potenciais, como entradas incorretas do usuário ou problemas em sistemas externos. Abaixo estão algumas das principais técnicas:

1. Validação de dados de APIs

- Sempre confira se os dados existem antes de acessar propriedades.
- Use **optional chaining (`?.`)** e **nullish coalescing (`??`)**.

```
const rating = product.rating?.rate ?? 0;
const title = product.title ?? "Título indisponível";
```

- Ajuda a prevenir erros do tipo: `Cannot read property 'rate' of undefined`.

Optional Chaining

Optional chaining no React é um recurso do JavaScript (introduzido no ES2020) que permite acessar propriedades de objetos aninhados de forma segura, evitando erros quando uma propriedade intermediária é `null` ou `undefined`. Ele é usado com o operador `?.` e, em vez de lançar um erro, a expressão retorna `undefined`, o que simplifica a escrita de código para lidar com dados dinâmicos.

2. Verificação de arrays antes de map/filter

```
{products && products.length > 0 ? (
  products.map(p => <ProductCard key={p.id} {...p} />)
) : (
  <p>Nenhum produto encontrado.</p>
)}
```

- Evita tentar mapear `undefined` ou arrays vazios sem feedback para o usuário.

3. Renderização condicional para loading ou erros

```
if (loading) return <p>Carregando...</p>;
if (error) return <p>Erro ao carregar os produtos</p>;
```

- Garante que a interface só tenta renderizar dados **quando eles estão prontos e válidos**.

4. Tratamento de erros de fetch ou promises

```
fetch(API_URL)
  .then(res => res.json())
  .then(setData)
  .catch(err => setError(err))
  .finally(() => setLoading(false));
```

- Evita que falhas na API quebrem o componente.
- Permite mostrar mensagens amigáveis para o usuário.

5. Evitar operações em estado indefinido

- Nunca assuma que o estado já está preenchido; sempre inicialize com valores seguros.

```
const [products, setProducts] = useState([]); // array vazio
```

- Garante que `.map()` e `.filter()` funcionem sem erro desde o início.

6. Separar lógica de renderização da lógica de dados

- Ex.: calcular listas filtradas antes do JSX:

```
const featuredProducts = products.filter(p => p.rating?.rate >= 4);
```

- Reduz o risco de fazer cálculos diretamente no JSX, que podem falhar ou deixar o componente bagunçado.

Referências

https://www.freecodecamp.org/news/react-conditional-rendering/

https://www.dhiwise.com/post/mastering-react-add-class-conditionally-a-comprehensive-guide