Introdução

React, desenvolvido pelo Facebook, é uma biblioteca JavaScript focada na construção de interfaces de usuário (UI). Sua principal característica é a criação de componentes reutilizáveis que permitem desenvolver aplicações web eficientes e escaláveis. Diferentemente de frameworks como Angular, React não é uma solução completa para desenvolvimento de aplicações, mas foca na "view" em uma arquitetura MVC (Model-View-Controller). Isso significa que React lida principalmente com a renderização de interfaces, enquanto outras responsabilidades, como gerenciamento de rotas e estado global, são frequentemente terceirizadas para bibliotecas complementares como React Router e Redux.

React é utilizado tanto para desenvolvimento web quanto mobile (através do React Native) e emprega o conceito de "componentes declarativos", o que simplifica o desenvolvimento e manutenção de interfaces complexas. O processo de desenvolvimento com React envolve:

- **Criação de componentes reutilizáveis**: Componentes são funções ou classes que retornam uma UI em JSX (JavaScript XML), uma extensão de sintaxe que combina JavaScript com HTML.
- **Gerenciamento de estado:** React permite o controle do estado dos componentes, que reflete as mudanças dinâmicas da interface.
- **Utilização de hooks:** Hooks como useState e useEffect são funções que permitem lidar com estados e efeitos colaterais dentro de componentes funcionais.

Nas próximas seções, abordaremos cada um desses elementos de forma mais detalhada.

Principais conceitos do React

De maneira geral, uma aplicação React é composta por pequenos blocos reutilizáveis chamados de **componentes**. Esses componentes podem ser combinados para criar interfaces dinâmicas e interativas.

Componentes

Um componente em React é uma unidade básica de UI que define como uma parte da interface deve ser renderizada. Um componente pode ser uma classe ou uma função que retorna elementos JSX. Abaixo, um exemplo de um componente funcional:

```jsx
function ContaList() {
  const [contas, setContas] = useState([]);
  const [contaSelecionada, setContaSelecionada] = useState(null);

  useEffect(() => {
    fetchContas().then(data => setContas(data));
  }, []);

  const selecionarConta = (conta) => {
    setContaSelecionada(conta);
  };

  return (
    <div>
      <h2>Listagem de Contas</h2>
      <ul>
        {contas.map((conta) => (
          <li key={conta.id} onClick={() => selecionarConta(conta)}>
            {conta.agencia} - {conta.numero}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Neste exemplo, usamos **hooks** como useState para gerenciar o estado e useEffect para realizar efeitos colaterais (como buscar dados da API).

JSX e Templates

React utiliza JSX, que permite escrever elementos HTML dentro do JavaScript. Embora a sintaxe se pareça com HTML, o JSX é na verdade uma representação em JavaScript de como os elementos devem ser renderizados.

```
return (
  <div>
    <h2>Listagem de Contas</h2>
    <ul>
      {contas.map(conta => (
        <li key={conta.id}>
          {conta.agencia} - {conta.numero}
        </li>
      ))}
    </ul>
  </div>
);
```

O código acima mostra como um template JSX pode ser escrito dentro de um componente React para renderizar uma lista de contas.

Estado (State) e Propriedades (Props)

O estado em React é utilizado para controlar dados dinâmicos de um componente. Através do **estado**, podemos refletir mudanças dinâmicas na interface. Já as **props** são propriedades passadas de um componente pai para um componente filho e são imutáveis.
Exemplo de uso de estado:

```
const [contaSelecionada, setContaSelecionada] = useState(null);
```

Props são usadas para passar informações entre componentes:

```
<ContaDetalhe conta={contaSelecionada} />
```

Data Binding em React

Em React, o fluxo de dados é unidirecional (one-way data binding), ou seja, os dados fluem do componente pai para o filho via props e, se necessário, eventos podem ser passados de volta. Isso melhora a previsibilidade e o controle da aplicação.

```
function ContaDetalhe({ conta }) {
  return conta ? (
    <div>
      <h3>Detalhes da Conta</h3>
      <p>Agência: {conta.agencia}</p>
      <p>Número: {conta.numero}</p>
    </div>
  ) : null;
}
```

Hooks

Hooks são funções que permitem "ligar" o estado e outras funcionalidades do React em componentes funcionais. Os dois hooks mais comuns são:

- useState: Para gerenciar estados dentro de componentes funcionais.
- useEffect: Para realizar efeitos colaterais como chamadas de API ou manipulação de DOM.

useState

O hook useState é uma função que permite adicionar e gerenciar estado local em componentes funcionais. Antes da introdução dos hooks, o estado só podia ser manipulado em componentes de classe. Agora, com useState, você pode declarar um estado diretamente dentro de componentes funcionais.

```
const [state, setState] = useState(initialState);
```

Aqui, state é a variável que mantém o valor atual do estado, e setState é a função usada para atualizá-lo. O valor inicial é definido por initialState. Veja a seguir um exemplo prático:

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

function Contador() {
  // Declara uma variável de estado chamada "contador", inicializada com o valor 0
  const [contador, setContador] = useState(0);

  return (
    <div>
      <p>Você clicou {contador} vezes</p>
      <button onClick={() => setContador(contador + 1)}>
        Incrementar
      </button>
    </div>
  );
}
```

Neste exemplo:

- contador é o valor atual do estado, inicialmente 0.
- setContador é a função que atualiza o valor de contador.
- Toda vez que o botão é clicado, setContador(contador + 1) incrementa o valor do estado, e o componente re-renderiza com o novo valor.

useEffect

O hook useEffect permite que você execute efeitos colaterais em componentes funcionais. Efeitos colaterais incluem coisas como:

- Buscar dados de uma API,
- Manipular o DOM diretamente,
- Subscrições ou timers.

O useEffect substitui os métodos de ciclo de vida dos componentes de classe, como componentDidMount, componentDidUpdate, e componentWillUnmount.

```
useEffect(() => {
  // Código executado após o componente ser montado ou atualizado

  return () => {
    // Código executado ao desmontar o componente (limpeza)
  };
}, [dependencias]);
```

A função de efeito é executada:

1. **Após a primeira renderização** (equivalente a componentDidMount).
2. **Após cada atualização** se as dependências mudarem (equivalente a componentDidUpdate).
3. **Quando o componente for desmontado**, a função de retorno (opcional) é chamada para limpar recursos (equivalente a componentWillUnmount).

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

function FetchData() {
  const [dados, setDados] = useState([]);

  useEffect(() => {
    // Fetch de dados de uma API após o componente ser montado
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response => response.json())
      .then(data => setDados(data));

    // Função de limpeza (opcional) ao desmontar o componente
    return () => {
      console.log("Componente desmontado!");
    };
  }, []); // A lista de dependências vazia faz com que o efeito seja executado apenas na

  return (
    <div>
      <h2>Dados buscados</h2>
      <ul>
        {dados.map(item => (
          <li key={item.id}>{item.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

No exemplo acima:

- O useEffect faz uma **chamada para uma API** assim que o componente é montado (equivalente a componentDidMount).
- O array de dependências vazio ([]) garante que o efeito seja executado **apenas uma vez**, logo após a montagem inicial.
- Se a função de retorno for definida, ela será chamada quando o componente for desmontado (útil para limpar subscrições ou timers).

Comparando Ciclo de Vida

Nos componentes de classe, o ciclo de vida envolve métodos específicos como:

- componentDidMount: Executado uma vez, logo após a montagem do componente.
- componentDidUpdate: Executado após cada atualização do componente.
- componentWillUnmount: Executado antes do componente ser desmontado, usado para limpar recursos.

Nos componentes funcionais, o useEffect pode substituir esses métodos:

- Para executar código na **montagem** (equivalente a componentDidMount), basta usar o useEffect com um array de dependências vazio:

```
useEffect(() => {
  // Código que executa após a montagem
}, []);
```

- Para **reagir a atualizações** (equivalente a componentDidUpdate), adicione dependências no array:

```
useEffect(() => {
  // Código que executa após alguma dependência mudar
}, [dependencia]);
```

- Para **limpar recursos** ao desmontar o componente (equivalente a componentWillUnmount), retorne uma função de limpeza dentro do useEffect:

```
useEffect(() => {
  // Código na montagem e atualização

  return () => {
    // Código de limpeza ao desmontar o componente
  };
}, [dependencia]);
```