Hooks

React Hooks are functions that allow you to use state and other React features in functional components.

1. \*\*useState:\*\*

`useState` is used to manage state in functional components. It returns an array with two elements: the current state value and a function to update it.

Eg

import React, { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

```

2. \*\*useEffect:\*\*

`useEffect` is used to perform side effects in functional components. It's similar to lifecycle methods in class components. You can use it to fetch data, set up subscriptions, or perform other side effects.

Eg

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

function DataFetcher() {

const [data, setData] = useState(null);

useEffect(() => {

// Fetch data from an API

fetch('https://api.example.com/data')

.then((response) => response.json())

.then((data) => setData(data));

}, []); // Empty dependency array means this effect runs once on component mount

return (

<div>

{data ? <p>Data: {data}</p> : <p>Loading...</p>}

</div>

);

}

```

3. \*\*useContext:\*\*

`useContext` allows you to access the context of a parent component. It's useful for passing data or functions down the component tree without having to manually pass props through each level of nesting.

Eg

import React, { useContext } from 'react';

// Create a context

const MyContext = React.createContext();

function ParentComponent() {

return (

<MyContext.Provider value={'Hello from context'}>

<ChildComponent />

</MyContext.Provider>

);

}

function ChildComponent() {

const contextData = useContext(MyContext);

return <p>{contextData}</p>;

}

```

4. \*\*useReducer:\*\*

`useReducer` is a hook for managing complex state logic. It's similar to `useState`, but it's designed for cases where the state has more complex updates based on the previous state.

Eg

import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

return state;

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, initialState);

return (

<div>

<p>Count: {state.count}</p>

<button onClick={() => dispatch({ type: 'increment' })}>Increment</button>

<button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>

</div>

);

}

These are just a few examples of React Hooks. React provides several other hooks, such as `useRef`, `useMemo`, and `useCallback`, to cover various use cases when building functional components. React Hooks have become a fundamental part of modern React development, and they offer a more concise and organized way to manage component logic.