React Hooks: useEffect

The **useEffect** hook allows you to perform side effects in React components. Side effects are tasks that run outside the normal component rendering process, such as fetching data, updating the document title, or setting up subscriptions.

What is a Side Effect?

Examples of side effects include:

- Fetching data from an API
- Directly updating the DOM
- Setting up timers or intervals
- Subscribing to events

Syntax

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

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

  useEffect(() => {
    document.title = `Clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
    </div>
  );
}
```

How it works

- `useEffect` runs after every render by default.
- In the above example, whenever `count` changes, the document title is updated.

Using Dependencies

You can control when the effect runs by passing a dependency array as the second argument.

```
useEffect(() => {
  console.log("Effect ran!");
}, [count]);
```

- The effect will only run when `count` changes.
- If you pass an empty array `[]`, the effect runs only once after the component mounts.

Cleanup with useEffect

Some effects require cleanup, like removing event listeners or clearing intervals. You can return a function from `useEffect` for cleanup.

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log("Timer running");
  }, 1000);

  return () => clearInterval(timer);
}, []);
```

- The cleanup function runs before the component unmounts or before the effect re-runs.

Key Points

- `useEffect` replaces lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
- Always specify dependencies clearly to avoid unnecessary re-renders.
- Use cleanup to prevent memory leaks.