Day 6: React Hooks - `useEffect` & `useRef`

Today, you'll learn about two essential React hooks:
- **`useEffect`** (for handling side effects like fetching data, event listeners, etc.)
- **`useRef`** (for accessing DOM elements and persisting values without re-renders)

1️⃣ What is `useEffect`?

`useEffect` is used to handle **side effects** in functional components, such as:

✅ Fetching API data
✅ Subscribing to events
✅ Updating the document title
✅ Managing timers

Basic Syntax of `useEffect`

```
useEffect(() => {
  // Code to run
}, [dependencies]);
```

- The **callback function** inside `useEffect` runs when the component **mounts, updates, or unmounts**.

- The **dependency array** `[dependencies]` controls when the effect runs.

2️⃣ Using `useEffect` Without Dependencies

If no dependencies are provided, `useEffect` **runs on every render**.

Example: Logging on Every Render

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

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

  useEffect(() => {
    console.log("Component re-rendered!");
  });

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}

export default Counter;
```

✅ **Every time the state updates, `useEffect` runs.**

3️⃣ Running `useEffect` Only on Mount (`[]` Dependency Array)

If you provide an **empty dependency array (`[]`)**, `useEffect` runs **only once when the component mounts**.

Example: Fetch API Data on Mount

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

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

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos/1")
      .then((response) => response.json())
      .then((json) => setData(json));
  }, []);

  return <pre>{JSON.stringify(data, null, 2)}</pre>;
}

export default FetchData;
```

✅ **Runs once on mount, fetches data, and updates state.**

4️⃣ Running `useEffect` When a Value Changes (`[dependency]`)

If you provide **a variable inside `[dependency]`**, `useEffect` runs when that value changes.

Example: Watching a Counter Value

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

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

  useEffect(() => {
    console.log(`Count changed: ${count}`);
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}

export default CounterWithEffect;
```

✅ **Effect runs only when `count` changes.**

5️⃣ Cleaning Up Side Effects (Unmounting)

If `useEffect` sets up an event listener, interval, or timeout, you should **clean it up** when the component unmounts.

Example: Cleanup with `return` inside `useEffect`

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);

    return () => {
      clearInterval(interval); // Cleanup when unmounted
    };
  }, []);

  return <p>Time: {seconds}s</p>;
}

export default Timer;
```

✅ **Clearing the interval prevents memory leaks.**

---

6 What is `useRef`?

`useRef` is used for **accessing DOM elements directly** and **storing values without causing re-renders**.

Basic Syntax of `useRef`

```
const myRef = useRef(initialValue);
```

- `useRef` **does not trigger re-renders** when its value changes.

---

7 Accessing DOM Elements with `useRef`

`useRef` is often used to interact with **input fields**.

Example: Focusing an Input Field

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

function FocusInput() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus(); // Directly focus input
  }

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Type something" />
      <button onClick={handleClick}>Focus Input</button>
    </div>
  );
}

export default FocusInput;
```

✅ **Button clicks focus the input without re-renders.**

8️⃣ Storing Values Without Re-Renders

Unlike `useState`, `useRef` persists values **without causing component re-renders**.

Example: Tracking Previous Count

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

function PreviousCounter() {
  const [count, setCount] = useState(0);
  const prevCount = useRef(0);

  useEffect(() => {
    prevCount.current = count; // Store previous value
  }, [count]);

  return (
    <div>
      <p>Current: {count}</p>
      <p>Previous: {prevCount.current}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}

export default PreviousCounter;
```

✅ **Tracks the previous count without re-rendering the component.**

📌 Summary of Day 6

- ✔ **`useEffect`** handles side effects like fetching data & event listeners
- ✔ **Runs on every render, mount (`[]`), or when dependencies change**
- ✔ **Clean up effects using the `return` function**
- ✔ **`useRef`** accesses DOM elements and persists values without re-renders

---

🚀 Next Step: Day 7 - React Router (Navigation & Dynamic Pages)