Internship Report – Frontend Dev

Week 6: React.js Basics

Name:	**Zainab**
Father Name:	**Assad Qayyum**
Date:	**31st July, 2025**
Internship Domain:	**Front-end Intern**
Task:	**useEffect Hook (basic usage)**

Task Overview: (Day4)

Today's task focused on understanding the **useEffect Hook in React.js.** The goal was to learn how to perform side effects in functional components, such as fetching data, updating the page title, and running certain code when components load or update.

Content Covered:

The content covered today is:

- Introduction to useEffect Hook
- Purpose of using useEffect
- Syntax and basic usage
- Dependency array in useEffect
- Cleanup function (basic understanding)

useEffect Hook (Basic Usage)

A The useEffect Hook in React allows you to **run code automatically after your component renders** or when **some data inside it changes**. This is useful for tasks like:

- Fetching data from an API
- Updating the document title
- Starting or stopping timers

Purpose:

1. To perform **side effects in functional components** (e.g., fetching data, timers, logging).
2. To **run code automatically** after rendering without manual triggers.
3. To update external resources like the document title or browser storage based on state changes.
4. To control when code runs using a dependency array, avoiding unnecessary re-runs.
5. To **clean up resources** (e.g., stopping timers, unsubscribe from events) when a component is removed.

Syntax:

```
useEffect(() => {

  // Code to run after rendering

}, [dependencies]);
```

The first part is a function that contains the code you want to run. The dependency array [] tells React when to run the effect:

- **Empty []** → Runs only once when the component first appears (mounts).
- **With variables** → Runs every time those variables change.

Adding a Dependency Array:

You can control when your effect runs by adding a dependency array:

```
useEffect(() => {

  console.log("Runs only once when the component mounts");

}, []);
```

Empty array [] → Runs only once (when the component first loads).

```
useEffect(() => {

  console.log("Runs every time `count` changes");

}, [count]);
```

With [count] → Runs only when the count variable changes.

Cleanup Function

Sometimes you need to **stop something when the component is removed** (like clearing a timer or unsubscribing).

You do this by returning a cleanup function:

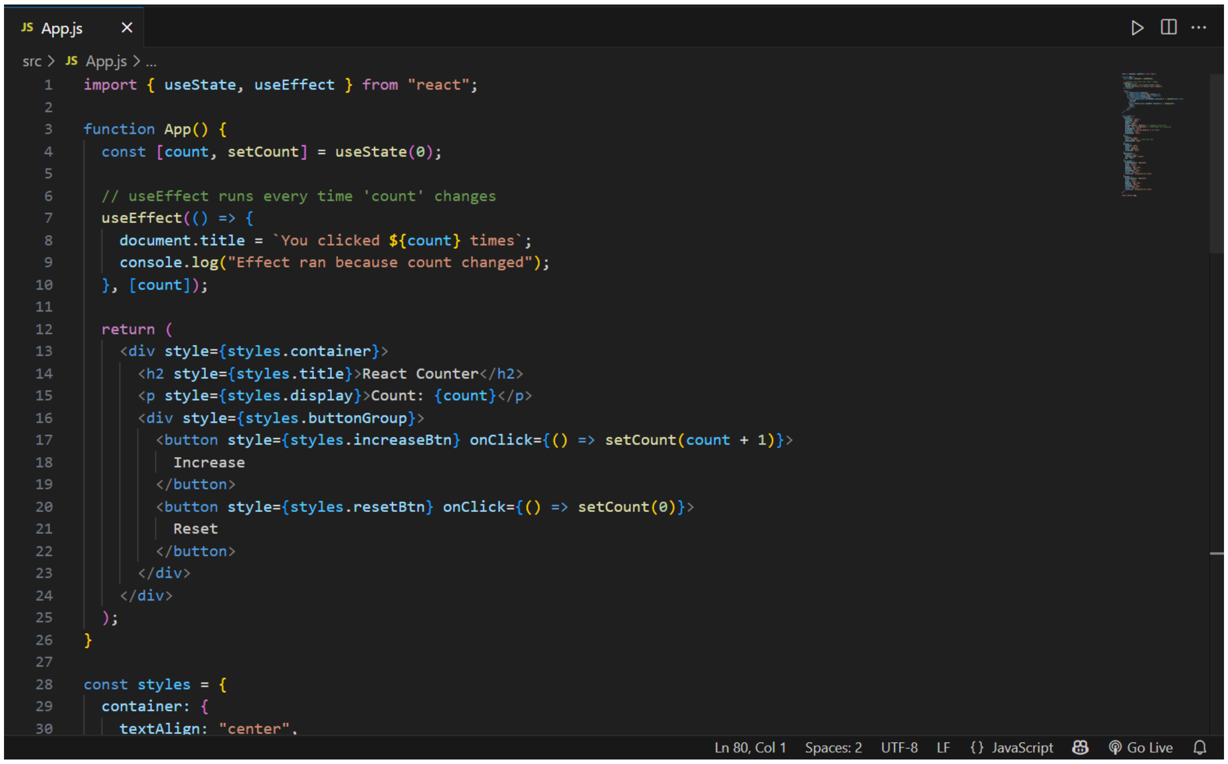
```
useEffect(() => {
  const timer = setInterval(() => {
    console.log("Timer running...");
  }, 1000);

  return () => clearInterval(timer); // Cleanup when component unmounts
}, []);
```

Example:

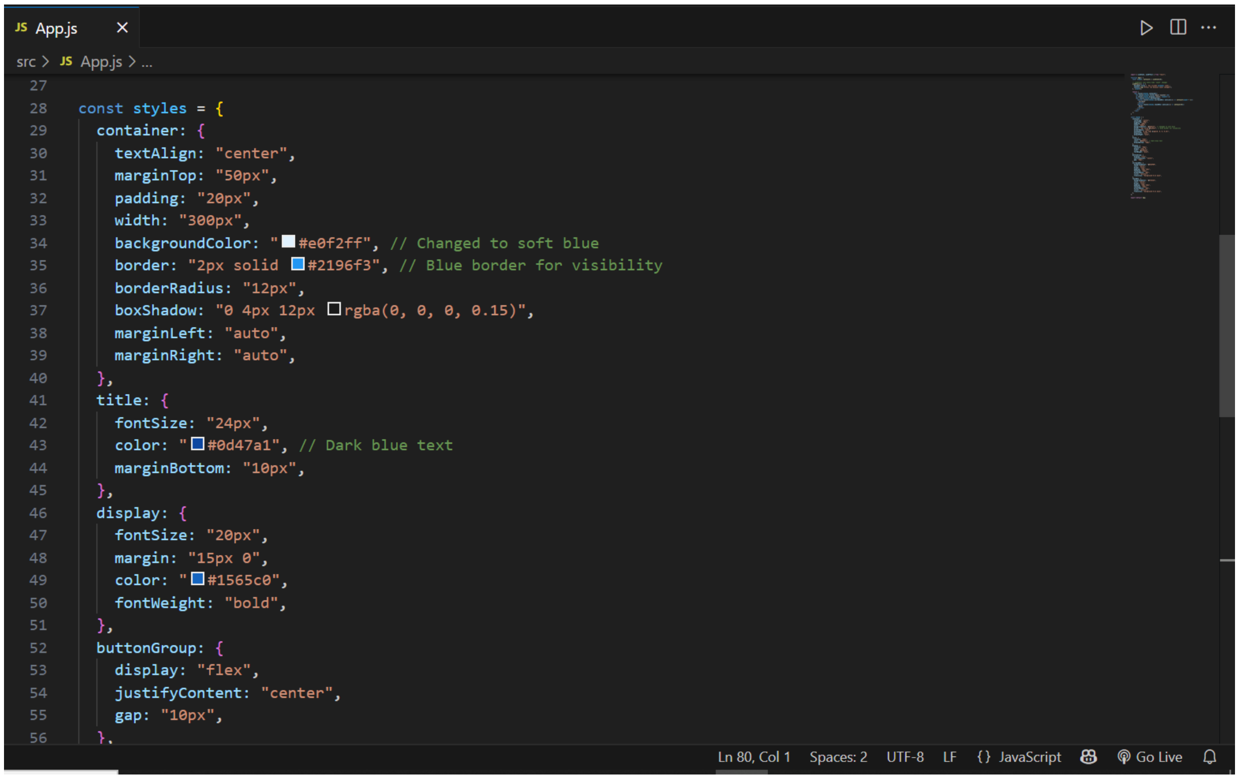
```
import { useState, useEffect } from "react";

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

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

  return (
    <button onClick={() => setCount(count + 1)}>
      Click Me
    </button>
  );
}
```

Practice Code:

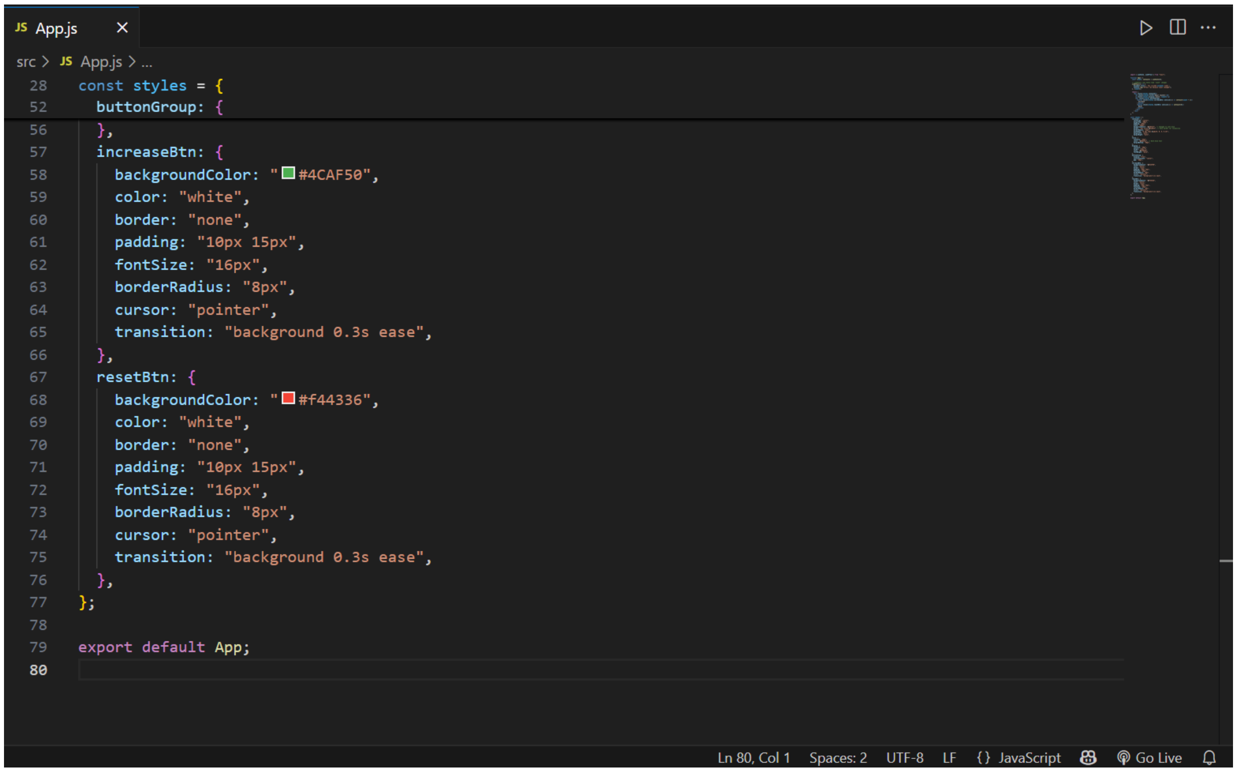
In this code, we used the **useEffect hook** to automatically **update the page title** whenever the **count changes.** This shows the basic usage of useEffect for handling side effects that happen outside the UI (like changing the browser title). It runs after every render when count updates, making the app more dynamic and interactive.



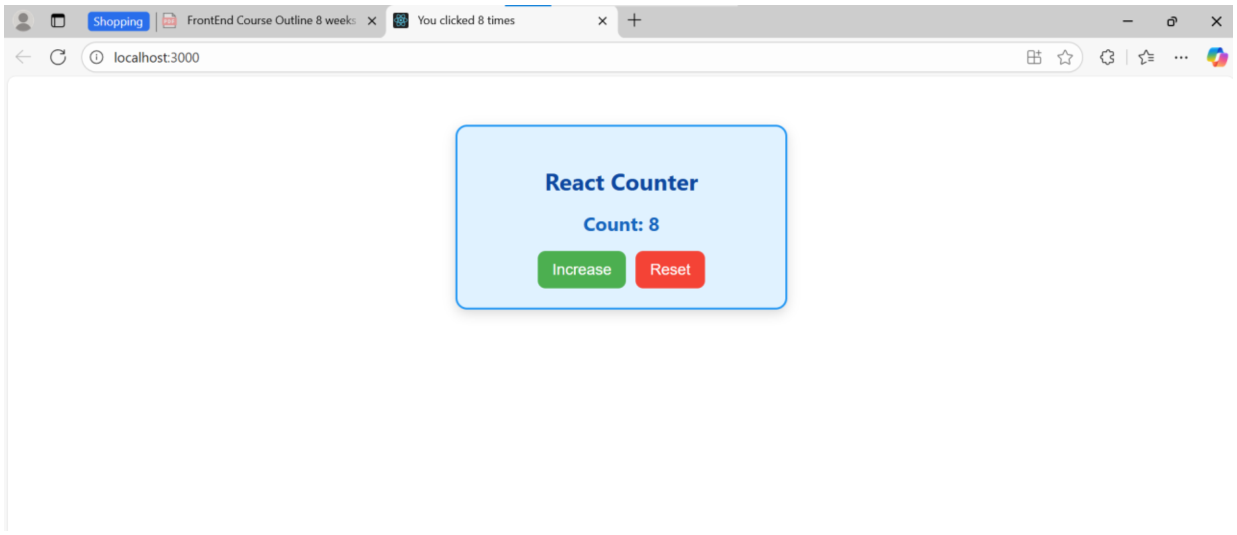
```javascript
import { useState, useEffect } from "react";

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

  // useEffect runs every time 'count' changes
  useEffect(() => {
    document.title = `You clicked ${count} times`;
    console.log("Effect ran because count changed");
  }, [count]);

  return (
    <div style={styles.container}>
      <h2 style={styles.title}>React Counter</h2>
      <p style={styles.display}>Count: {count}</p>
      <div style={styles.buttonGroup}>
        <button style={styles.increaseBtn} onClick={() => setCount(count + 1)}>
          Increase
        </button>
        <button style={styles.resetBtn} onClick={() => setCount(0)}>
          Reset
        </button>
      </div>
    </div>
  );
}

const styles = {
  container: {
    textAlign: "center",
```



```javascript

const styles = {
  container: {
    textAlign: "center",
    marginTop: "50px",
    padding: "20px",
    width: "300px",
    backgroundColor: "#e0f2ff", // Changed to soft blue
    border: "2px solid #2196f3", // Blue border for visibility
    borderRadius: "12px",
    boxShadow: "0 4px 12px rgba(0, 0, 0, 0.15)",
    marginLeft: "auto",
    marginRight: "auto",
  },
  title: {
    fontSize: "24px",
    color: "#0d47a1", // Dark blue text
    marginBottom: "10px",
  },
  display: {
    fontSize: "20px",
    margin: "15px 0",
    color: "#1565c0",
    fontWeight: "bold",
  },
  buttonGroup: {
    display: "flex",
    justifyContent: "center",
    gap: "10px",
  },
```



```
const styles = {
  buttonGroup: {
  },
  increaseBtn: {
    backgroundColor: "#4CAF50",
    color: "white",
    border: "none",
    padding: "10px 15px",
    fontSize: "16px",
    borderRadius: "8px",
    cursor: "pointer",
    transition: "background 0.3s ease",
  },
  resetBtn: {
    backgroundColor: "#f44336",
    color: "white",
    border: "none",
    padding: "10px 15px",
    fontSize: "16px",
    borderRadius: "8px",
    cursor: "pointer",
    transition: "background 0.3s ease",
  },
};

export default App;
```



Conclusion:

In this task, I learned the basic usage of the useEffect Hook in React. I now understand how to run code after rendering and control when it runs using dependencies. This Hook is very useful for handling side effects like updating the page title, fetching data, or managing timers in functional components.