We use `useEffect` in React for several key reasons related to managing side effects in functional components. Side effects are operations that interact with the outside world or require cleanup, such as data fetching, subscriptions, timers, and manipulating the DOM. Here’s a detailed breakdown of why and when we use `useEffect`:

Reasons to Use `useEffect`

1. **Data Fetching**:
   - **Purpose**: Fetch data from an API or other external sources when a component mounts or when specific dependencies change.
   - **Example**: Fetching user data when a component mounts.
   ```javascript
   useEffect(() => {
     fetch('https://api.example.com/user')
       .then(response => response.json())
       .then(data => setUser(data));
   }, []); // Empty array ensures this runs only once on mount
   ```

2. **Event Listeners and Subscriptions**:
   - **Purpose**: Add event listeners or subscribe to external events and clean them up when the component unmounts to prevent memory leaks.
   - **Example**: Adding a window resize event listener.
   ```javascript
   useEffect(() => {
     const handleResize = () => console.log('Window resized');
     window.addEventListener('resize', handleResize);

     return () => {
       window.removeEventListener('resize', handleResize); // Cleanup on unmount
     };
   }, []); // Empty array ensures this runs only once on mount
   ```

3. **Updating the DOM**:
   - **Purpose**: Perform DOM manipulations that are outside the scope of React's rendering cycle.
   - **Example**: Updating the document title based on state changes.
   ```javascript
   useEffect(() => {
     document.title = `Count: ${count}`;
   }, [count]); // Runs every time count changes
   ```

4. **Timers and Intervals**:
   - **Purpose**: Set up timers or intervals and clear them when the component unmounts to avoid memory leaks.
   - **Example**: Creating a timer that updates every second.
   ```javascript
   useEffect(() => {
     const interval = setInterval(() => {
       setSeconds(prev => prev + 1);
     }, 1000);

     return () => {
       clearInterval(interval); // Cleanup interval on unmount
     };
   }, []); // Empty array ensures this runs only once on mount
   ```

5. **Effect Dependencies**:
   - **Purpose**: Perform actions when specific props or state variables change.
   - **Example**: Fetching data when the URL prop changes.

```javascript
   useEffect(() => {
     fetch(url)
       .then(response => response.json())
       .then(data => setData(data));
   }, [url]); // Runs every time url changes
   ```

Benefits of Using `useEffect`

1. **Unified Lifecycle Management**:
   - Combines the functionality of `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` from class components into a single, consistent API.

2. **Declarative Code**:
   - Makes it clear what side effects are happening and when they should run, improving the readability and maintainability of the code.

3. **Automatic Cleanup**:
   - Provides a way to clean up resources (like event listeners and intervals) when the component unmounts, preventing memory leaks and other issues.

4. **Synchronization with React's Render Cycle**:
   - Ensures that side effects are synchronized with the component's lifecycle, avoiding issues with stale data and ensuring that effects run at the correct times.

Example Scenarios for Using `useEffect`

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

function UserProfile({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`https://api.example.com/users/${userId}`)
      .then(response => response.json())
      .then(data => setUser(data));
  }, [userId]); // Runs every time userId changes

  if (!user) {
    return <div>Loading...</div>;
  }

  return <div>{user.name}</div>;
}
```

Event Listeners
```javascript
import React, { useEffect } from 'react';

function ResizeLogger() {
  useEffect(() => {
    const handleResize = () => {
      console.log('Window resized');
    };

    window.addEventListener('resize', handleResize);

    return () => {
      window.removeEventListener('resize', handleResize);
```

```javascript
    };
  }, []); // Empty array ensures this runs only once on mount

  return <div>Resize the window and check the console</div>;
}
```

Updating the Document Title
```javascript
import React, { useEffect, useState } from 'react';

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

  useEffect(() => {
    document.title = `Count: ${count}`;
  }, [count]); // Runs every time count changes

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

By using `useEffect`, you can handle side effects in a clean, declarative manner, ensuring that your components remain predictable and easy to understand.