`useEffect` is a React Hook that lets you perform side effects in functional components. Side effects are operations that occur outside the normal flow of your application, such as data fetching, subscriptions, or manually changing the DOM. `useEffect` combines the lifecycle methods `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` from class components into a single API.

Basic Syntax

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

function MyComponent() {
  useEffect(() => {
    // Side effect code here

    return () => {
      // Cleanup code here (optional)
    };
  }, [/* dependencies */]);

  return (
    <div>
      {/* Component JSX */}
    </div>
  );
}
```

Detailed Explanation

1. **Effect Callback**:
   The first argument to `useEffect` is a callback function that contains the side effect code.
   ```javascript
   useEffect(() => {
     // Code to run on mount and updates
   });
   ```

2. **Cleanup Function**:
   The callback can return a function that cleans up after the effect. This cleanup function runs before the component unmounts or before the effect runs again.
   ```javascript
   useEffect(() => {
     // Code to run on mount and updates

     return () => {
       // Cleanup code
     };
   });
   ```

3. **Dependencies Array**:
   The second argument is an array of dependencies that control when the effect runs. The effect runs after the initial render and whenever any value in the dependencies array changes.
   ```javascript
   useEffect(() => {
     // Code to run on mount and updates

     return () => {
       // Cleanup code
   ```

```
    };
  }, [dependency1, dependency2]);
  ```

4. **Empty Dependencies Array**:
   If you pass an empty array, the effect runs only once after the initial render (like `componentDidMount`).
   ```javascript
   useEffect(() => {
     // Code to run only on mount

     return () => {
       // Cleanup code to run on unmount
     };
   }, []);
   ```

Example Use Cases

1. **Fetching Data**:
   ```javascript
   import React, { useEffect, useState } from 'react';

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

     useEffect(() => {
       fetch('https://api.example.com/data')
         .then(response => response.json())
         .then(data => setData(data));
     }, []); // Empty array ensures this runs only once

     return (
       <div>
         {data ? <pre>{JSON.stringify(data, null, 2)}</pre> : <p>Loading...</p>}
       </div>
     );
   }
   ```

2. **Subscribing to an Event**:
   ```javascript
   import React, { useEffect } from 'react';

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

       window.addEventListener('resize', handleResize);

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

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

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

```
  function DocumentTitleUpdater() {
    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>
    );
  }
  ```

Benefits of `useEffect`

- **Unified API**: Combines component lifecycle methods (`componentDidMount`, `componentDidUpdate`, `componentWillUnmount`) into a single API.
- **Declarative**: Makes it clear when and why side effects are run.
- **Cleanup**: Allows for easy cleanup of resources when the component unmounts or before the effect runs again.

`useEffect` is a powerful tool for managing side effects in functional components, making it easier to write and maintain React applications.