In React, the useEffect hook is used to manage side effects in function components. A **side effect** refers to any operation that interacts with the outside world or system outside of the component, such as fetching data from an API, directly manipulating the DOM, or subscribing/unsubscribing from a service.

Here's why and when you use useEffect.

**1. Performing Side Effects**

React components are primarily focused on rendering UI. However, sometimes you need to perform operations that don’t directly relate to rendering but are necessary for the component, like:

* Fetching data from an external source (API calls).
* Subscribing to WebSocket connections or timers.
* Manually updating the DOM (like setting the document title).
* Storing values in local storage.

Without useEffect, managing these tasks within function components would be cumbersome and not intuitive.

**2. Lifecycle Management in Functional Components**

Before hooks, lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount) were only available in class components. The useEffect hook allows functional components to simulate these lifecycle methods:

* **On Mount (componentDidMount)**: To execute an effect when a component is first rendered.
* **On Update (componentDidUpdate)**: To run an effect when certain props or state variables change.
* **On Unmount (componentWillUnmount)**: To clean up resources or subscriptions when the component is removed.

**3. useEffect Syntax**

Here’s the basic syntax for useEffect:

tsx

Copy code

useEffect(() => {

// Code to run on mount or when dependencies change

return () => {

// Cleanup code (runs when component unmounts or dependencies change)

};

}, [dependencies]); // dependencies array

**4. When to Use useEffect**

**a) Fetching Data**

You can use useEffect to fetch data from an API when a component mounts.

tsx

Copy code

useEffect(() => {

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data));

}, []); // Empty array = run once when component mounts

**b) Subscribing to Events**

For example, listening to window resize events or a WebSocket stream:

tsx

Copy code

useEffect(() => {

const handleResize = () => {

console.log('Resized!', window.innerWidth);

};

window.addEventListener('resize', handleResize);

return () => {

window.removeEventListener('resize', handleResize); // Cleanup on unmount

};

}, []); // Empty array = run on mount and cleanup on unmount

**c) Running Code When a State or Prop Changes**

You can use useEffect to run some logic when a specific state or prop changes:

tsx

Copy code

useEffect(() => {

console.log('Count changed:', count);

}, [count]); // Runs whenever 'count' changes

**5. Dependencies Array**

* The second argument to useEffect is an optional dependencies array. If any value in the array changes, the effect will re-run.
  + **Empty array ([])**: Runs the effect only once (on mount).
  + **Array with dependencies ([var1, var2])**: Runs the effect whenever var1 or var2 changes.
  + **No array**: Runs the effect after every render.

**6. Cleaning Up**

When the component unmounts (or before re-running the effect in case dependencies change), you can perform cleanup to prevent memory leaks (e.g., unsubscribing from events, canceling API calls).

tsx

Copy code

useEffect(() => {

const timer = setInterval(() => {

console.log('Interval running');

}, 1000);

return () => clearInterval(timer); // Cleanup when component unmounts

}, []); // Effect runs on mount, cleanup runs on unmount

**Summary of Key Points:**

* **useEffect manages side effects** like data fetching, subscriptions, and manual DOM manipulations.
* It replaces class-based lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount) in function components.
* It has a **dependency array** to control when the effect runs, and it can return a **cleanup function** to handle unmounting.

Overall, useEffect is a powerful tool that allows React function components to perform side effects in a clean and manageable way.

This **Demo Project** that covers the following React hooks: useState, useCallback, useContext, useEffect, useMemo, useReducer, and useRef. The project will be a **simple Todo List** with the following features:

* Adding todos
* Completing todos
* Counting completed and pending tasks
* Persisting the data using localStorage

& **A theme toggle** page for useContext

**Project Overview:**

1. **useState**: For managing the list of todos and input fields.
2. **useCallback**: For memoizing functions that are passed as props to child components.
3. **useContext**: For managing a global theme (light/dark mode).
4. **useEffect**: For syncing todos to localStorage.
5. **useMemo**: For calculating the total completed/pending tasks efficiently.
6. **useReducer**: For managing state updates in a more structured way than useState.
7. **useRef**: For focusing on the input field automatically after adding a todo.

**Explanation of Concepts:**

1. **useState**:
   * Used to handle input field state (inputValue), which tracks the current value of the input for adding new todos.
2. **useCallback**:
   * We use useCallback to memoize the addTodo and toggleTodo functions, ensuring they don't get re-created on every render.
3. **useContext**:
   * The ThemeContext is used to globally manage the theme (light/dark mode), and we can toggle the theme using the toggleTheme function.
4. **useEffect**:
   * useEffect is used to load the todos from localStorage when the component mounts and save the todos back to localStorage whenever the todos change.
5. **useMemo**:
   * useMemo is used to optimize the calculation of completed and pending todos by memoizing the result and recalculating only when the state.todos array changes.
6. **useReducer**:
   * useReducer manages the todo list state (adding todos, toggling their completion status), providing a more structured approach than useState for state management.
7. **useRef**:
   * useRef is used to keep a reference to the input field, allowing us to automatically focus the input field after a todo is added.