tell me how to know that when should i use (useState(), useEffect(), useCallback(), useRef() hooks etc.) , tell me how can i configure that when should i use these things .

To understand when to use React hooks like **useState()**, **useEffect()**, **useCallback()**, and **useRef()**, let's break down their purpose and usage scenarios:

1. useState()

Purpose: Used to manage state in a functional component. It allows you to store values that may change over time and re-render the component when that state changes.

When to use:

> If you need to store and update values (like form inputs, counters, or toggling a UI element).

> For example, tracking whether a modal is open or not.

```
const [isOpen, setIsOpen] = useState(false);
```

2. useEffect()

Purpose: Used to perform side effects in function components. Side effects include data fetching, manual DOM manipulation, or logging, and it runs after the render.

> When to use:

If you want to fetch data from an API after the component renders.

If you want to synchronize with external systems (e.g., setting up a subscription or event listeners).

If you want to update the document title based on state or props.

Example for data fetching:

```
useEffect(() => {
  fetchData();  // Side effect
}, []);
```

3. useCallback()

Purpose: Used to memoize functions to prevent them from being recreated on every render. This can be useful when passing functions to child components that rely on reference equality to avoid unnecessary renders.

> When to use:

If you are passing down a callback (e.g., in props) to a child component and want to prevent that function from being re-created on every re-render.

Helps with performance optimization in components that have costly re-renders or when using functions in useEffect dependencies.

```
const handleClick = useCallback(() => {
  console.log("Clicked!");
}, [dependencies]);
```

4. useRef()

Purpose: Provides a way to access and modify DOM elements directly or store values that persist across renders without causing a re-render.

> When to use:

If you need to access or modify a DOM element, like focusing an input field.

If you want to store a value that doesn't trigger a re-render when updated (e.g., keeping track of the previous state, timers, or IDs).

Example to focus an input field:

```
const inputRef = useRef(null);
useEffect(() => {
  inputRef.current.focus();
}, []);
```

How to know when to use each hook:

> **useState**: Use when you need to store and update state in a component.

> **useEffect**: Use for side effects, like data fetching, API calls, DOM updates, or subscriptions.

> **useCallback**: Use to optimize performance, especially when passing functions as props to child components to avoid unnecessary re-renders.

> **useRef:** Use when you need to reference a DOM element or store values across renders without triggering a re-render.

By identifying the scenario in which you need to store state, perform side effects, or optimize performance, you can decide which hook to use.

is these are the only uses of mentioned hooks ,

The mentioned uses of the hooks (useState, useEffect, useCallback, useRef) cover the primary and most common scenarios, but these hooks can be applied in other nuanced ways depending on the complexity of your React app. Here's a deeper look into additional or less common use cases for each hook:

1. useState()

Additional uses:

> Lazy initialization: You can delay the initial state setup using a function when the initial state requires expensive calculations or operations.

```
const [count, setCount] = useState(() => computeExpensiveValue());
```

Object or array state: You can manage complex states like objects or arrays using useState.

```
const [user, setUser] = useState({ name: "", age: 0 });
```

2. useEffect()

Additional uses:

> **Cleanup effects:** useEffect can return a cleanup function to clean up things like subscriptions or event listeners when a component unmounts or re-renders.

```
useEffect(() => {
  const handleScroll = () => console.log("scrolling");
  window.addEventListener("scroll", handleScroll);
```

```
  return () => {
    window.removeEventListener("scroll", handleScroll); // Cleanup
  };
}, []);
```

> **Conditional effects:** You can run effects only when certain dependencies change. This allows more fine-tuned control over when side effects run.

```
useEffect(() => {
  if (someCondition) {
    doSomething();
  }
}, [someCondition]);  // Only runs when 'someCondition' changes
```

3. useCallback()

Additional uses:

> **Memoizing event handlers:** Beyond preventing re-renders in child components, useCallback can be used to ensure expensive callback functions (like event handlers) are only re-created when necessary.

> **Preventing unnecessary calculations:** It helps with scenarios where callbacks are passed into useEffect, preventing them from being re-run unless necessary.

```
const expensiveCalculation = useCallback(() => computeHeavyTask(), [dependency]);
```

4. useRef()

Additional uses:

> **Persisting mutable values:** Beyond accessing DOM elements, useRef is commonly used to store mutable values that don't cause re-renders. For example, keeping track of the previous render's state or other values that persist through renders.

```
const previousValue = useRef();
useEffect(() => {
  previousValue.current = currentValue;
}, [currentValue]);
```

> **Storing intervals or timers:**You can store references to timers (like setTimeout, setInterval) without triggering re-renders when updating them.

```
const intervalRef = useRef();
useEffect(() => {
```

```
    intervalRef.current = setInterval(() => {
      // Some periodic task
    }, 1000);

    return () => clearInterval(intervalRef.current); // Cleanup
  }, []);
```

Other Related Hooks and Their Uses:

1. useMemo(): Similar to useCallback, but for memoizing values (instead of functions). It prevents expensive calculations from being recalculated on every render unless dependencies change.

```
const memoizedValue = useMemo(() => computeExpensiveValue(), [dependencies]);
```

2. useReducer(): A more advanced alternative to useState when state logic becomes more complex. It's especially useful for managing state with multiple sub-values or when you need state transitions.

```
const [state, dispatch] = useReducer(reducerFunction, initialState);
```

Key Considerations:

> **useState:** Use when you need to manage component state and trigger re-renders.

> **useEffect:** Use for side effects like data fetching, DOM manipulation, or subscriptions.

> **useCallback:** Use to memoize functions for performance optimization.

> **useRef:** Use to persist values across renders or directly interact with the DOM.

Each of these hooks can be creatively adapted to more complex use cases as your React app grows. Their flexibility allows them to be applied beyond the simple scenarios mentioned initially.