Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)

What are React hooks? How do useState() and useEffect() hooks work in functional components?

Hooks let you use features like **state** and **lifecycle methods** in **functional components**, without needing classes.

useState()

- Adds **state** to a function component.
- Returns a value and a function to update it.

```
const [count, setCount] = useState(0);
```

useEffect()

- Handles **side effects** (e.g., fetching data, timers).
- Runs after render. Can also run on updates or unmount.

```
useEffect(() => {
  // effect code

  return () => {
    // cleanup code
  };
}, []);
```

useState() manages state, and useEffect() handles side effects—both are key hooks in React functional components.

What problems did hooks solve in React development? Why are hooks considered an important addition to React?

Before hooks, managing state and side effects in React required class components, which were often:

- Hard to reuse logic across components
- Messy with large lifecycle methods
- Confusing this binding

Why Are Hooks Important?
Hooks allow:

- State and side effects in functional components
- Cleaner, reusable logic with custom hooks
- Simpler and more readable code

Hooks made it easier to write cleaner, reusable, and more functional React code—solving many limitations of class components.

What is useReducer ? How we use in react app?

useReducer is a React hook used to manage complex state logic in functional components. It's an alternative to useState, especially when state updates depend on previous state or involve multiple sub-values.

```
const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
```

```
    return { count: state.count - 1 };
  default:
    return state;
 }
}

const [state, dispatch] = useReducer(reducer, initialState);
Then use it in JSX:
<button onClick={() => dispatch({ type: "increment" })}>+</button>
<p>{state.count}</p>
```

useReducer is useful for managing more complex state logic in React apps. It works like Redux but is built into React.

What is the purpose of useCallback & useMemo Hooks?

The useCallback and useMemo hooks in React are used to optimize performance by memoizing values or functions, preventing unnecessary recalculations or re-creations during re-renders.

1. useCallback:
   - Purpose: useCallback is used to memoize a function, ensuring that the function reference remains the same between renders unless its dependencies change.
   - Use Case: It's useful when passing functions as props to child components. Without useCallback, a new function would be created on every render, causing unnecessary re-renders of child components. useCallback helps maintain the same function reference across renders.

Example:

```
const [count, setCount] = useState(0);

const increment = useCallback(() => {
  setCount(count + 1);
```

```
}, [count]);  // Only recreate the function when `count` changes

return <button onClick={increment}>Increment</button>;
```

Here, increment is only recreated if count changes, preventing unnecessary re-creations on each render.

2. useMemo:

- Purpose: useMemo is used to memoize the result of a computation or a derived value, recalculating it only when its dependencies change.
- Use Case: It's ideal for expensive calculations or operations that don't need to be re-executed unless specific dependencies change.

Example:

```
const [count, setCount] = useState(0);

const expensiveCalculation = useMemo(() => {
  return count * 1000;  // Only recompute when `count` changes
}, [count]);

return <div>{expensiveCalculation}</div>;
```

Here, expensiveCalculation is only recalculated when count changes, preventing it from being recomputed on every render.

Summary:

- useCallback: Memoizes functions to prevent unnecessary re-creations.
- useMemo: Memoizes the result of a calculation to avoid unnecessary recalculations.

Both hooks improve performance by reducing unnecessary operations during re-renders, but they should be used only when necessary. Overuse of these hooks can lead to performance

overhead, so they are best suited for cases with expensive computations or functions passed to child components.

What's the Difference between the useCallback & useMemo Hooks?

The **useCallback** and **useMemo** hooks in React are used to optimize performance by memoizing values and functions, but they serve different purposes.

1. useCallback:

- **Purpose**: useCallback is used to **memoize functions**, ensuring that the function is not recreated on every render unless its dependencies change.
- **When to Use**: It is primarily useful when passing functions as props to child components, preventing unnecessary re-creations of the function during re-renders.

Example:

```
const increment = useCallback(() => {
  setCount(count + 1);
}, [count]);  // The function is recreated only when `count` changes
```

Here, increment is only recreated when count changes, optimizing performance when passing the function down to child components.

2. useMemo:

- **Purpose**: useMemo is used to **memoize the result** of a computation, preventing expensive recalculations unless its dependencies change.
- **When to Use**: It is useful for preventing unnecessary recalculation of complex values or operations during re-renders.

Example:

```
const expensiveCalculation = useMemo(() => {
  return count * 1000;  // Only recalculated when `count` changes
}, [count]);
```

In this example, expensiveCalculation is only recalculated when count changes. If count remains the same, React uses the cached value.

Key Differences:

- **useCallback** memoizes **functions**, ensuring that the function reference remains stable across renders.
- **useMemo** memoizes the **result of a calculation** or **value**, avoiding unnecessary recalculation unless the dependencies change.

Summary:

- **useCallback**: Memoizes a function and prevents its recreation on every render, ideal for optimizing performance when functions are passed to child components.
- **useMemo**: Memoizes the result of an expensive computation or derived value, preventing unnecessary recalculation.

Both hooks help optimize performance, especially in large or complex React applications, but should be used wisely to avoid unnecessary overhead.

What is useRef ? How to work in react app?

useRef is a React Hook that allows you to persist values across renders without causing re-renders and to directly interact with DOM elements. It can be particularly useful for handling mutable values or interacting with elements outside the typical React data flow.

1. Accessing DOM Elements:

One of the most common use cases of useRef is for accessing and manipulating **DOM elements** directly, such as focusing on an input field or measuring the size of a DOM element.

Example:

```
import React, { useRef } from 'react';


function App() {
  const inputRef = useRef(null);


  const handleFocus = () => {
    inputRef.current.focus();  // Focus the input element
  };


  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleFocus}>Focus on Input</button>
    </div>
  );
}


export default App;
```

In this example, inputRef stores a reference to the input element, and inputRef.current is used to access and focus the input.

2. Persisting Values Across Renders:

Another important feature of useRef is its ability to **persist values** between renders without causing re-renders. This makes it ideal for scenarios where you need to store mutable data (such as previous values or timers) that doesn't need to trigger a re-render when updated.

Example:

```
import React, { useState, useEffect, useRef } from 'react';


function Timer() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef();


  useEffect(() => {
    prevCountRef.current = count;  // Store the previous count
  }, [count]);


  return (
    <div>
      <h1>Current Count: {count}</h1>
      <h2>Previous Count: {prevCountRef.current}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

export default Timer;

Here, prevCountRef holds the previous count value across renders, but updating it doesn’t trigger a re-render, unlike useState.

Key Benefits:

- **No Re-renders**: Updating a useRef value doesn't cause the component to re-render, unlike useState.
- **Access to DOM**: useRef is often used for DOM manipulation, like focusing an element, measuring sizes, or handling animations.
- **Persistent Values**: It can store values across renders, like tracking previous states or storing timers.

useRef is a powerful hook for accessing DOM elements and persisting values without causing re-renders. It's useful for scenarios like focusing inputs, tracking mutable values, or managing timers, making it an essential tool for performance optimization and handling side effects in React.

Lists and Keys

How do you render a list of items in React? Why is it important to use keys when rendering lists?

To render a list in React, you use **Array.map()** to loop through the items and return JSX for each one:

```
const fruits = ["Apple", "Banana", "Orange"];


return (
  <ul>
    {fruits.map((fruit, index) => (
      <li key={index}>{fruit}</li>
    ))}
  </ul>
);
```

Why Use Keys?

Keys help React **identify which items changed, added, or removed**. This makes updates more efficient and prevents bugs.

- Keys should be **unique** and **stable** (like an ID).
- Avoid using array indexes if the list can change order.

```
<li key={fruit.id}>{fruit.name}</li>
```

Use .map() to render lists, and always include a unique key to help React optimize rendering and maintain UI stability.

What are keys in React, and what happens if you do not provide a unique key?

Keys are special props used when rendering lists in React. They help React **identify each item** and track changes like additions, deletions, or reordering.

{items.map(item => <li key={item.id}>{item.name})}

What If You Don’t Use Unique Keys?

- React may **re-render incorrectly** or **mix up items**.
- Performance may suffer.
- UI bugs can occur (e.g., incorrect input values or animations).

Always use **unique and stable keys** (like IDs). Without them, React can’t reliably track list items, leading to bugs and inefficient updates.

Context API

What is the Context API in React? How is it used to manage global state across multiple components?

The **Context API** in React is a way to manage and share **global state** across multiple components without having to pass props manually through each component. It allows components at different nesting levels to access the same state, avoiding **prop drilling**.

How it Works:

1. **createContext**: Creates a context object to store and provide global data.
2. **Provider**: A component that makes the context's value available to all child components.
3. **useContext**: A hook used to consume the context and access its value in child components.

Example:

1. **Create Context**:
2. const MyContext = React.createContext();
3. **Provider**:
4. function App() {
5. const [state, setState] = useState("Hello World");
6.
7. return (
8. <MyContext.Provider value={{ state, setState }}>

```
      <ChildComponent />
        </MyContext.Provider>
      );
    }
```

Consume Context:

```
    function ChildComponent() {
      const { state, setState } = useContext(MyContext);

      return (
        <div>
          <p>{state}</p>
          <button onClick={() => setState("New Value")}>Change State</button>
        </div>
      );
    }
```

Key Points:

- **Global State**: Ideal for managing state that needs to be accessed by multiple components.
- **No Prop Drilling**: Prevents the need to pass props through many layers of components.
- **Dynamic Updates**: Any update in the context triggers a re-render in the consuming components.

Use Cases:

- **User authentication** state, **themes**, **language preferences**, or any global data that needs to be shared across the app.

The **Context API** simplifies state management for global data, making it accessible to any component without manually passing props.

Explain how createContext() and useContext() are used in React for sharing state.

In React, **createContext()** and **useContext()** are used together to share state across components without the need for prop drilling.

1. createContext():

- **Purpose**: It creates a **context object** that will hold the shared state. This context object has a Provider and Consumer for sharing and accessing the state.
- **Usage**:
- const MyContext = React.createContext();

2. useContext():

- **Purpose**: It allows components to **consume** the shared state from the context.
- **Usage**:
- const value = useContext(MyContext);

Example:

1. **Create Context**:
2. const MyContext = React.createContext();
3. **Provide State via Provider**:

```
function App() {
  const [state, setState] = useState("Hello World");

  return (
    <MyContext.Provider value={{ state, setState }}>
      <ChildComponent />
    </MyContext.Provider>
  );
}
```

Consume State with useContext():

```
function ChildComponent() {
  const { state, setState } = useContext(MyContext);

  return (
    <div>
      <p>{state}</p>
      <button onClick={() => setState("New Value")}>Change State</button>
    </div>
  );
}
```

Summary:

- **createContext()** creates the context.
- **Provider** shares the state.

- **useContext()** allows components to access the shared state.

This combination simplifies state management across deeply nested components without prop drilling.