Question 1: What are React hooks? How do useState() and useEffect() hooks work in functional components?

React Hooks are functions that let developers use state and other React features (like lifecycle methods) in functional components, avoiding the need for class components.

- **useState**: Allows functional components to have state variables. It returns an array with the state value and a setter function to update the state.

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

- **useEffect**: Performs side effects (e.g., fetching data, subscriptions). It runs after render and can clean up when the component unmounts.

```
useEffect(() => {
  console.log('Component mounted or updated');
  return () => {
    console.log('Component unmounted or before re-render');
  };
}, [dependencies]);
```

Question 2: What problems did hooks solve in React development? Why are hooks considered an important addition to React?

Problems Hooks Solved:

1. **Complex State Logic**: Managing state logic in class components was cumbersome, especially for components with complex state and side effects.
2. **Code Reusability**: Sharing stateful logic between components was difficult, often requiring higher-order components (HOCs) or render props, which led to complex and less readable code.
3. **Verbose Syntax**: Class components required verbose syntax, including constructor, this bindings, and lifecycle methods.
4. **Side Effects Management**: Handling side effects (e.g., subscriptions) in class components was difficult as lifecycle methods like componentDidMount and componentWillUnmount were split across the component lifecycle.

Importance of Hooks:

1. **Functional Components**: Hooks allow functional components to manage state and side effects, making them as powerful as class components.
2. **Simpler Syntax**: Hooks reduce boilerplate and make components easier to understand and write.
3. **Reusable Logic**: Custom hooks enable reusable stateful logic, improving code modularity and readability.

4. **Better Composition**: Hooks make it easier to compose functionalities by combining multiple hooks within a component.

Question 3: What is useReducer? How do we use it in a React app?

useReducer:

- useReducer is a React Hook used for managing complex state logic. It is an alternative to useState, particularly useful when the state depends on previous states or involves multiple sub-values.

How it Works:

- It accepts two arguments:
    1. **Reducer Function**: A function (state, action) => newState that determines the state update logic.
    2. **Initial State**: The initial value of the state.
- Returns:
    - The current state.
    - A dispatch function to trigger state updates.

Usage Example:

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

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:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

return (
```

```
  <div>
    <p>Count: {state.count}</p>
    <button onClick={() => dispatch({ type: 'increment' })}>+</button>
    <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
   </div>
  );
}
```

Question 4: What is the purpose of useCallback and useMemo Hooks?

useCallback:

- Used to memoize functions.
- Prevents re-creation of functions on every render, which helps avoid unnecessary renders in child components when the function is passed as a prop.

Usage Example:

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

useMemo:

- Used to memoize values.
- Prevents recalculating values on every render unless dependencies change.

Question 5: What's the Difference between useCallback and useMemo?

Feature	useCallback	useMemo
Purpose	Memoizes a function	Memoizes a value
Return	Returns a memoized function reference	Returns a memoized value
Use Case	Avoiding re-creation of functions	Avoiding recalculation of expensive values

---

Question 6: What is useRef? How does it work in a React app?

useRef:

- useRef creates a mutable object that persists across renders.
- It can hold a reference to a DOM element or store any mutable value that doesn't trigger a re-render when updated.

How It Works:

1. Create a ref using useRef.
2. Attach it to a DOM element using the ref attribute to directly manipulate the DOM.

LAB EXERCISE:

Task 1: • Create a functional component with a counter using the useState() hook. Include buttons to increment and decrement the counter

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


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


 return (
  <div>
   <h1>Counter: {count}</h1>
   <button onClick={() => setCount(count + 1)}>Increment</button>
   <button onClick={() => setCount(count - 1)}>Decrement</button>
  </div>
 );
}
export default Counter;
```

Task 2: Fetching Data with useEffect

```
import React, { useEffect, useState } from 'react';
function FetchData() {
 const [data, setData] = useState([]);
 const [loading, setLoading] = useState(true);


 useEffect(() => {
  fetch('https://jsonplaceholder.typicode.com/posts')
   .then(response => response.json())
   .then(data => {
    setData(data);
```

```
      setLoading(false);
    })
    .catch(error => console.error('Error fetching data:', error));
 }, []);
 if (loading) {
  return <p>Loading...</p>;
 }
 return (
  <div>
   <h1>Posts</h1>
   <ul>
    {data.map(post => (
     <li key={post.id}>{post.title}</li>
    ))}
   </ul>
  </div>
 );
}
export default FetchData;
```

Task 3: React App with useSelector and useDispatch

Store and Reducer Setup

javascript

Copy code

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';
const store = configureStore({
 reducer: {
  counter: counterReducer,
 },
```

```
});
export default store;
```

Slice Setup

```
// counterSlice.js
import { createSlice } from '@reduxjs/toolkit';
export const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: state => {
      state.value += 1;
    },
    decrement: state => {
      state.value -= 1;
    },
  },
});

export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

React Component

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './counterSlice';

function CounterApp() {
  const count = useSelector(state => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
```

```
      <h1>Counter: {count}</h1>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>
  );
}

export default CounterApp;
```

Entry Point

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux';
import store from './store';
import CounterApp from './CounterApp';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <Provider store={store}>
      <CounterApp />
    </Provider>
  </React.StrictMode>
);
```

Task 4: Avoid Re-renders with useRef

```
import React, { useRef, useState } from 'react';
function AvoidReRenders() {
  const [count, setCount] = useState(0);
  const renders = useRef(0);
 renders.current += 1;
return (
```

```
    <div>
      <h1>Count: {count}</h1>
      <h2>Renders: {renders.current}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
export default AvoidReRenders;
```

Question 1: What is the Context API in React? How is it used to manage global state across multiple components?

What is Context API?

- The Context API in React is a tool for managing global state across multiple components without the need to pass props through the component tree manually.
- It provides a way to share data (e.g., themes, user authentication, language settings) between components efficiently.

How it Works:

1. **Context Provider**: Supplies the global state to components in the tree.
2. **Context Consumer**: Accesses and uses the global state provided by the Context.

Why Use Context API?

- Avoids "prop drilling" (passing props through multiple levels of components).
- Simplifies state management for small to medium-sized apps without requiring external libraries like Redux.

Question 2: Explain how createContext() and useContext() are used in React for sharing state.

createContext():

- Used to create a Context object.
- Provides a Provider component to supply the context value and a Consumer component to consume the value.

useContext():

- A React Hook used to access the value of a context directly without the need for a Consumer component.
- Simplifies context consumption by avoiding the nested function structure.