DAY-21 EVENING ASSESSMNET

**React Hooks & Context**

31. The two main rules of hooks are: (1) Only call hooks at the top level of a component, (2) Only call hooks from React function components or custom hooks.

32. useState returns a state variable and setter function, while class components use this.state and this.setState. useState replaces local state logic without classes.

33. Pass a callback to the setter: setCount(prev => prev + 1). This ensures the update is based on the latest state.

34. Common use cases for useEffect: fetching data, setting up subscriptions, updating the DOM, logging, timers/intervals.

35. Clean up by returning a function inside useEffect,

e.g. useEffect(()🡪 { const id = setInterval(...); return () 🡪 clearInterval(id); }, []);.

36. Without a dependency array, useEffect runs after every render, potentially causing infinite loops.

37. useContext avoids prop drilling by letting you directly access context values instead of passing props down through multiple components.

38. Create context: const MyContext = React.createContext();. Provide with <MyContext.Provider value={...}>. Consume with useContext(MyContext).

39. To avoid re-renders, memoize context values using useMemo or split contexts so only relevant consumers update.

40. Example custom hook for form input:  
  
function useInput(initial) {  
 const [value, setValue] = useState(initial);  
 const onChange = e => setValue(e.target.value);  
 return { value, onChange };  
}

41.useEffect runs after the render is committed, while useLayoutEffect runs synchronously before the browser paints, useful for DOM measurements.

42.A custom hook for API fetching: use useState, useEffect to fetch data and return { data, loading, error }.

43.Multiple useEffect hooks let you separate concerns (cleaner and easier to manage), while one big useEffectmixes unrelated logic.

44.Hooks can’t be used inside conditionals because React relies on call order to track hook state. Conditional usage would break that.

45.You share logic between multiple components by extracting it into a custom hook and reusing it.

**API Integration with Fetch & Axios**   
46. Fetch is built-in, returns promises, and requires manual error handling. Axios is an external library with simpler syntax, automatic JSON parsing, interceptors, and better error handling.

47. Example GET with Axios:  
  
useEffect(() 🡪 { axios.get("/api/data").then(res => setData(res.data)); }, []);

48.Handle errors with .catch(err => console.error(err)) or try/catch inside async functions.

49.POST with JSON body:  
axios.post("/api/data", { name: "John" }, { headers: { "Content-Type": "application/json" } });

50.Fetch does not set Content-Type automatically; Axios sets application/json for objects by default.

51.PUT request:  
axios.put("/api/data/1", { name: "Updated" });

52.DELETE request:  
axios.delete("/api/data/1");

53.Cancel request with CancelToken or AbortController:  
const controller = new AbortController();  
axios.get("/api", { signal: controller.signal });  
controller.abort();

54.An interceptor lets you run logic before a request/response (e.g., attach auth tokens, log errors).

55.Handle loading states with a loading state variable set to true before request, false after success or error.

**Pure Components**

56. A Pure Component is a class component that implements shouldComponentUpdate with a shallow prop and state comparison.

57. They improve performance by preventing unnecessary re-renders when props/state haven’t changed.

58. React.memo is the functional component equivalent of PureComponent, memoizing output unless props change.

59. A Pure Component re-renders only if props or state values change in a shallow comparison (primitive values or different object references).

60. Limitations: shallow comparison misses deep changes, may skip updates if mutable objects are mutated, adds overhead for small components.