**Day-21 Evening Assessment**

**React Hooks & Context**

31. -Only call hooks at the top level.

-Only call hooks from React functions.

32. -useState returns a state variable + setter.  
-Class components use this.state and this.setState.  
-useState replaces this.setState with simpler syntax.

33. setCount(prev => prev + 1);

34. -Data fetching  
-Subscribing to events

-Updating document title  
-Cleanup

35. useEffect(() => {  
 const timer = setInterval(...);  
 return () => clearInterval(timer);  
}, []);

36. The effect runs after every render, causing possible performance issues or infinite loops.

37. Prop drilling: pass data manually through many layers.  
useContext: access shared state directly from context, avoiding unnecessary prop passing.

38. const ThemeContext = React.createContext();  
  
function App() {  
 return (  
<ThemeContext.Provider value="dark">  
<Child />  
</ThemeContext.Provider>  
 );  
}  
  
function Child() {  
 const theme = React.useContext(ThemeContext);  
 return <div>{theme}</div>;  
}

39. <Context.Provider value={useMemo(() => ({user}), [user])}>  
Split contexts so unrelated consumers don’t re-render.  
40. function useInput(initial) {  
 const [value, setValue] = useState(initial);  
 const onChange = e => setValue(e.target.value);  
 return { value, onChange };  
}

41. useEffect: runs after paint (async, non-blocking).  
useLayoutEffect: runs before paint (blocking, good for DOM measurements).

42. function useFetch(url) {  
 const [data, setData] = useState(null);  
 useEffect(() => {  
   fetch(url).then(res => res.json()).then(setData);  
 }, [url]);  
 return data;  
}

43. Multiple useEffect: easier to separate concerns (e.g., one for fetching, one for event listeners).  
Single useEffect: harder to maintain if logic is mixed.

44. Because hook order must remain consistent across renders. Conditional usage breaks React’s hook tracking.

45. By creating a custom hook and using it in different components.  
  
**API Integration with Fetch & Axios**

46. Fetch: built-in, returns promises, needs manual error handling.  
Axios: external library, automatic JSON parsing, supports interceptors, request cancellation, better error handling.

47. useEffect(() => {  
 axios.get("/api/data")  
   .then(res => setData(res.data))  
   .catch(console.error);  
}, []);

48. axios.get("/api/data")  
 .catch(err => {  
   if (err.response) console.log(err.response.data);  
   else console.log("Network error");  
 });

49. axios.post("/api/users", { name: "amitha", age: 23 });

50. Fetch: does not set Content-Type automatically.  
Axios: automatically sets Content-Type: application/json for JSON requests.

51. axios.put("/api/users/1", { name: "Amitha" });

52. axios.delete("/api/users/1");

53. const controller = new AbortController();  
axios.get("/api/data", { signal: controller.signal });  
controller.abort();

54. A function that modifies requests or responses globally .

55. const [loading, setLoading] = useState(false);  
  
useEffect(() => {  
 setLoading(true);  
 axios.get("/api/data")  
   .then(res => setData(res.data))  
   .finally(() => setLoading(false));  
}, []);

**Pure Components**

56. A component that only re-renders if props/state change shallowly.

57. They skip unnecessary renders by doing a shallow comparison of props/state.

58. React.memo is the functional equivalent → prevents re-renders unless props change.

59. Any shallowly unequal props (new object/array reference, primitive value change).

60. -Shallow comparison only → nested objects may still cause re-renders.  
-Can add complexity if overused.  
-Doesn’t help if component always depends on context or parent state changes.