|  |  |
| --- | --- |
| Q. No | Questions |
| 01 | **What is Redux,** **and why is it used?** |
|  | Redux is a **predictable state container** for JavaScript applications, primarily used with **React** but also compatible with other frameworks. It helps manage application state in a **centralized store**, making state changes more predictable and easier to debug.  **Why is Redux Used?**   1. **Centralized State Management** – Redux maintains the entire application state in a single store, making data management consistent across components. 2. **Predictability** – Since state changes are handled using **pure functions (reducers)**, debugging and tracking state updates become easier. 3. **Easier Debugging** – With tools like Redux DevTools, developers can **time travel**, inspect state changes, and replay actions. 4. **Better Scalability** – As apps grow, Redux helps manage complex state interactions more efficiently. 5. **Improved Performance** – Prevents unnecessary re-renders by optimizing how state updates propagate through the app. 6. **Global State Sharing** – Makes sharing state between distant components easier, avoiding **prop drilling** (passing props through multiple levels). |
| 02 | **What are the core principles of Redux? Explain in-depth answers.** |
|  | 1. **Single Source of Truth**   **Explanation:**   * The entire state of the application is stored in a **single JavaScript object** inside a **Redux store**. * This makes it easier to track state changes, debug issues, and persist state across sessions.   **Example:**   |  | | --- | | **const initialState = {**  **user: { name: "John", loggedIn: false },**  **cart: { items: [], total: 0 },**  **};**  **const store = createStore(reducer, initialState);** |  * Here, the state object contains **everything** related to the application, such as user details and cart information.   **Advantages:** ✅ Centralized state management → Makes debugging and testing easier. ✅ Persistent state → Helps in saving user preferences and maintaining state after page reloads. ✅ Predictability → State updates always follow the same pattern.  **Potential Drawbacks & Solutions:** ❌ **State becomes too large** → Use **Redux Toolkit** with features like createEntityAdapter. ❌ **Performance issues** → Use **selectors (Reselect)** to optimize state access.   1. **State is Read-Only (Immutable State)**   **Explanation:**   * The only way to change the state is by dispatching an action. * Reducers must be pure functions—they should not directly modify the existing state but return a new state. * This ensures that every state change is trackable and can be replayed (useful for debugging with Redux DevTools).   **Bad Practice (Mutating State - ❌ DO NOT DO THIS!)**   |  | | --- | | function cartReducer(state, action) {  if (action.type === "ADD\_ITEM") {  state.items.push(action.payload); // ❌ Direct mutation  return state;  }  return state;  } |   **Correct Approach (Immutable Update - ✅ DO THIS!)**   |  | | --- | | function cartReducer(state, action) {  if (action.type === "ADD\_ITEM") {  return {  ...state,  items: [...state.items, action.payload], // ✅ Returns a new state object  };  }  return state;  } |   **Advantages:** ✅ Ensures **predictability** → No unexpected mutations. ✅ Enables **time-travel debugging** → Since every state change is logged, developers can revert or replay changes. ✅ Encourages **functional programming** → Helps maintain clean and reusable logic.  **Potential Drawbacks & Solutions:** ❌ **Too many manual updates** → Use **Immer.js** (included in Redux Toolkit) for easier state updates. ❌ **Performance issues due to deep cloning** → Use **structural sharing** with spread operators and createSelector.   1. **Changes are Made with Pure Functions (Reducers)**   **Explanation:**   * The logic to update the state is handled in reducers, which are pure functions (i.e., they always produce the same output given the same input and have no side effects). * Reducers take (previous state, action) → return new state.   **Example of a Reducer:**   |  | | --- | | function counterReducer(state = { count: 0 }, action) {  switch (action.type) {  case "INCREMENT":  return { count: state.count + 1 };  case "DECREMENT":  return { count: state.count - 1 };  default:  return state;  }  } |   **Key Rules for Reducers:** ✅ Always return a new state object. ✅ Do not modify the original state. ✅ Do not perform side effects like API calls inside reducers.  **Advantages:** ✅ Ensures **predictability** → Given the same action and state, the result is always the same. ✅ Makes **debugging easy** → Each state change is explicit and can be logged. ✅ Improves **testability** → Since reducers are pure functions, unit testing becomes straightforward.  **Potential Drawbacks & Solutions:** ❌ **Boilerplate Code** → Use **Redux Toolkit’s createSlice** to reduce boilerplate. ❌ **Handling async operations is tricky** → Use **middlewares like Redux Thunk or Redux Saga**.  **Summary Table:**   |  |  |  | | --- | --- | --- | | **Principle** | **Explanation** | **Benefits** | | |  | | --- | | **Single Source of Truth** |  |  | | --- | |  | | |  | | --- | | All state is stored in a single store. |  |  | | --- | |  | | |  | | --- | | Centralized state, easier debugging, predictable behaviour. |  |  | | --- | |  | | | |  | | --- | | **State is Read-Only** |  |  | | --- | |  | | |  | | --- | | State cannot be modified directly; only actions can change it. |  |  | | --- | |  | | Predictability, immutability, enables time-travel debugging. | | |  | | --- | | **Changes are Made with Pure Functions (Reducers)** |  |  | | --- | |  | | |  | | --- | | Reducers handle state updates without side effects. |  |  | | --- | |  | | Ensures functional purity, testability, and maintainability. | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |