Capstone Project: NexSales Enterprise Dashboard

1. Project Scenario

You have been hired as the Lead Frontend Engineer for **NexSales**, a high-frequency trading and inventory management startup. The current MVP (Minimum Viable Product) is sluggish, difficult to maintain, and crashes frequently.

Your Mission: Build the internal "Admin Dashboard" from scratch. This dashboard must handle high-velocity data updates, manage complex global state, and provide a resilient user experience for data analysts who view thousands of rows of data simultaneously.

2. Functional Requirements

The application must include the following core features:

1. **Authentication System:** A login screen that restricts access to the dashboard.
2. **Live Inventory Feed:** A data table displaying 5,000+ product rows with sorting and filtering.
3. **Product Editor:** A complex form to edit product details with validation.
4. **Analytics Widget:** A heavy visualization component (charts/graphs).
5. **Notification System:** Global pop-up toasts for success/error messages.

3. Technical Requirements

You must implement the following technical constraints to demonstrate mastery of Module 5.

A. State Management (Redux Toolkit)

- **Store Architecture:** Use configureStore to set up the global store.

- **Slices:**
  - authSlice: Manage user tokens and role-based permissions.
  - inventorySlice: Manage the list of products. Use **createEntityAdapter** (normalization) to manage the IDs and Entities of products efficiently.
  - uiSlice: Manage global UI state (Sidebar open/close, Dark Mode toggle).
- **Async Logic:** Use createAsyncThunk to simulate fetching data from an API. Handle pending, fulfilled, and rejected states.

B. Performance Engineering

- **The Laggy Table Challenge:** The inventory list will simulate 5,000 items.
  - Implement **Virtualization** (optional challenge) or strict pagination.
  - Use **React.memo** on the InventoryRow component to prevent re-rendering all 5,000 rows when a single row changes.
  - Use **useMemo** to cache the "Filtered Products" list so filtering doesn't run on every keystroke or unrelated state change.
  - Use **useCallback** for the "Delete" button handler passed to every row.
- **Code Splitting:** The "Analytics" page is heavy. Use **React.lazy** and **Suspense** to load this route only when the user clicks the "Analytics" tab.

C. Advanced Design Patterns

- **Compound Components:** Build a reusable <DataTable> component that consumes the inventory data.
  - *Usage Example:*

    ```
    JavaScript
    <DataTable data={products}>
      <DataTable.Column header="Name" field="name" />
      <DataTable.Column header="Price" field="price" />
    </DataTable>
    ```

- **Portals:** Create a Modal component using createPortal for the "Delete Confirmation" dialog. Ensure it renders outside the root DOM hierarchy but maintains event bubbling.
- **Error Boundaries:** Wrap the "Analytics" widget in a Class-based **Error Boundary** (or react-error-boundary). Add a "Simulate Crash" button inside the analytics widget to prove the boundary catches the error and displays a Fallback UI without crashing the Navigation bar.

D. Testing (RTL & Jest)

- **Integration Test:** Write a test for the **Login Flow**.
  1. Render the App.
  2. Simulate typing into Email/Password.
  3. Simulate clicking "Login".
  4. **Mock** the network request using Jest or MSW.
  5. **Assert** that the user is redirected to the Dashboard (verify the "Welcome" text appears).

---

4. Implementation Milestones

Phase 1: The Foundation (State & Routing)

1. Initialize the app with create-react-app or Vite.
2. Install @reduxjs/toolkit, react-redux, react-router-dom.
3. Set up the Redux Store and the authSlice.
4. Create the Routes: /login, /dashboard (protected), /dashboard/analytics (lazy loaded).

Phase 2: The Data Engine (Complex State)

1. Create inventorySlice using createEntityAdapter.
2. Create a mock API function that returns an array of 5,000 dummy products after a 2-second delay.
3. Dispatch the fetchInventory thunk when the Dashboard mounts.

Phase 3: The Optimization Pass (Performance)

1. Render the list naively (mapping over all items). Notice the lag.
2. Implement useMemo for filtering logic.
3. Extract the row into a separate component and wrap with React.memo.
4. Pass action handlers using useCallback.
5. *Verification:* Use React DevTools Profiler to show "Did not render" for unaffected rows during an update.

Phase 4: Architecture Polish (Patterns)

1. Refactor the Table into the **Compound Component** pattern.
2. Implement the **Portal** for the "Edit Product" modal.
3. Add the **Error Boundary** around the heavy Analytics component.

Phase 5: Quality Assurance (Testing)

1. Set up Jest/RTL.
2. Write the "Happy Path" integration test for the User Login.
3. Write a unit test for the inventorySlice reducer to ensure adding a product updates the state correctly.

---

5. Bonus Challenges

- **Custom Hook:** Extract the fetch logic into a useFetchInventory hook that handles the dispatch and selector logic internally.
- **HOC:** Create a withRole('admin') HOC that hides the "Delete" button for non-admin users.
- **Selector Optimization:** Use createSelector from Redux Toolkit to create a memoized selector that returns the *total value* of the inventory (Price * Quantity) without recalculating on every render.