**Detailed Specifications for the Warehouse Management System (WMS)**

**1. Core Features**

**User Authentication and Authorization**

* **Login/Signup:**
  + Admins can create accounts for warehouse managers.
  + Use JWT for secure token-based authentication.
  + Password hashing with libraries like bcrypt.
* **Role-Based Access Control (RBAC):**
  + **Admin:**
    - Full access to all inventory, orders, and users.
    - Can add/edit/remove users and define roles.
  + **Warehouse Manager:**
    - Limited access to inventory and orders only.
    - Cannot manage users or system settings.

**Inventory Management**

* **CRUD Operations:**
  + Add, update, and delete items with fields like:
    - Item Name
    - SKU (Stock Keeping Unit) or unique identifier
    - Quantity in stock
    - Category (e.g., electronics, apparel, etc.)
    - Location (linked to the location management feature)
    - Supplier information (if needed)
  + Validate data on both frontend and backend.
* **Stock Alerts:**
  + Notify admin/managers if stock falls below a threshold.
  + Optionally display alerts in the dashboard or via real-time notifications.

**Order Management**

* **Purchase Orders (PO):**
  + Create orders for incoming inventory, with fields like:
    - PO number, item details, expected quantity, and ETA.
* **Sales Orders (SO):**
  + Track outgoing inventory, with fields like:
    - Order number, customer details, items, quantity, and shipping status.
* **Order Status:**
  + Pending, Fulfilled, or Shipped.
  + Allow filtering orders by status.

**Location Management**

* **Bin Locations:**
  + Organize inventory into locations such as aisles, racks, or bins.
  + Store location details (e.g., Aisle 3, Bin B-12).
* **Search by Location:**
  + Ability to filter or search for items stored in specific bins.

**Reporting**

* **Basic Reports:**
  + Total inventory items.
  + Low-stock items.
  + Order statistics (e.g., number of pending/fulfilled orders).
* **Graphs/Charts:**
  + Use libraries like Chart.js to visualize inventory trends.

**2. Frontend (React)**

**Dynamic UI**

* **Responsive Design:**
  + Ensure mobile and desktop compatibility using Material-UI (MUI) or TailwindCSS.
* **Real-Time Updates:**
  + Update inventory or order statuses dynamically using Socket.IO or periodic polling.

**State Management**

* **Redux/Context API:**
  + Centralize application state for smoother management across components.
  + Examples:
    - Store user authentication status.
    - Manage inventory and order data globally.

**Frontend Features**

1. **Login/Signup Pages:**
   * User-friendly forms for authentication.
2. **Dashboard:**
   * Overview of key metrics like stock count, pending orders, and low-stock alerts.
   * Highlight actionable insights (e.g., items needing replenishment).
3. **Inventory Page:**
   * Interactive data table for inventory, with:
     + Sorting, filtering, and searching options.
     + Action buttons (edit/delete).
   * Modal for adding or editing items.
4. **Orders Page:**
   * Similar table-based interface for orders.
   * Buttons for changing order status (e.g., Pending → Fulfilled).
5. **Reports Page:**
   * Dynamic charts for inventory and order trends.

**3. Backend (Node.js & Express.js)**

**REST APIs**

1. **Authentication:**
   * Endpoints:
     + POST /api/auth/register
     + POST /api/auth/login
     + GET /api/auth/profile (to fetch current user info).
2. **Inventory Management:**
   * Endpoints:
     + GET /api/inventory (list all items).
     + POST /api/inventory (add a new item).
     + PUT /api/inventory/:id (update an item).
     + DELETE /api/inventory/:id (delete an item).
3. **Order Management:**
   * Endpoints:
     + GET /api/orders (list all orders).
     + POST /api/orders (create a new order).
     + PUT /api/orders/:id (update order status).
4. **Reporting:**
   * Endpoints:
     + GET /api/reports/low-stock
     + GET /api/reports/orders

**Real-Time Notifications**

* Use Socket.IO for:
  + Stock alerts (e.g., low-stock notification).
  + Live updates on order fulfillment.

**4. Database Design (MongoDB)**

**Users Collection**

{

"\_id": "UUID",

"name": "John Doe",

"email": "john@example.com",

"password": "hashed\_password",

"role": "admin" // or "manager"

}

**Inventory Collection**

{

"\_id": "UUID",

"name": "Widget A",

"sku": "W123",

"quantity": 100,

"category": "Electronics",

"location": "Aisle 3, Bin B-12"

}

**Orders Collection**

{

"\_id": "UUID",

"type": "Purchase" // or "Sales",

"items": [

{ "sku": "W123", "quantity": 50 }

],

"status": "Pending",

"assignedTo": "manager\_id"

}

**Locations Collection**

{

"\_id": "UUID",

"bin": "Bin B-12",

"aisle": "Aisle 3",

"description": "Near exit"

}

**5. Deployment**

* **Frontend:** Deploy on Vercel.
* **Backend:** Deploy on Render/Heroku.
* **Database:** Use MongoDB Atlas for cloud hosting.

Would you like guidance on implementing any specific feature or tackling challenges like deployment or state management?

**Detailed Explanation of Order Management: Purchase Orders (PO)**

**Order Management** is a critical feature in a Warehouse Management System (WMS), responsible for tracking and managing the flow of inventory, both incoming (Purchase Orders) and outgoing (Sales Orders).

**What is a Purchase Order (PO)?**

A **Purchase Order** is a formal request to procure goods or inventory from a supplier. It typically includes details like the type, quantity, and agreed price of the items. In your system, POs will represent incoming inventory.

**Key Features for Purchase Orders**

**1. Purchase Order Creation**

* **Data Fields for PO:**
  + **PO Number:** A unique identifier for the purchase order (e.g., PO-20240101-001).
  + **Date:** The date when the PO was created.
  + **Supplier Details:**
    - Supplier name.
    - Contact information.
  + **Items List:**
    - Item SKU or name.
    - Quantity ordered.
    - Expected delivery date.
  + **Total Cost (Optional):**
    - Automatically calculated as the sum of the cost of all items.

**2. Purchase Order Status Tracking**

* **Statuses:**
  + **Pending:** Order has been created but not yet delivered.
  + **In Progress:** Supplier has confirmed the order and started processing.
  + **Completed:** All items in the PO have been delivered to the warehouse.
  + **Cancelled:** PO was cancelled.

**3. Receiving Purchase Orders**

* Once items arrive at the warehouse, the system allows users to:
  + **Update the status** to "Completed."
  + Confirm the **received quantity** for each item (to handle discrepancies).
  + Log discrepancies between ordered and received quantities.

**4. PO Filtering and Searching**

* Users can filter POs by:
  + Date range.
  + Status (Pending, Completed, etc.).
  + Supplier.
* Search functionality to find POs by:
  + PO number.
  + Item name or SKU.

**5. PO History and Audit Logs**

* Track changes to POs, including:
  + Status updates.
  + Quantity adjustments.
  + Date and time of modifications.
  + User who made the changes.

**Frontend Implementation for POs**

**1. PO List Page**

* A dynamic table displaying:
  + PO number.
  + Supplier name.
  + Date created.
  + Status.
* Actions:
  + View details.
  + Edit or update PO status.
  + Delete PO (optional, depending on permissions).

**2. PO Details Page**

* View detailed information about a specific PO, including:
  + Item list with quantities and costs.
  + Supplier information.
  + Status history.

**3. Create/Edit PO Form**

* A form for adding or editing POs:
  + Dropdown or autocomplete for selecting items and suppliers.
  + Input fields for quantities and delivery date.
  + Submit button to save the PO.

**4. Alerts**

* Real-time notifications for pending POs that are overdue or need attention.

**Backend Implementation for POs**

**Database Schema for Purchase Orders**

{

"\_id": "UUID",

"poNumber": "PO-20240101-001",

"supplier": {

"name": "Supplier A",

"contact": "supplier@example.com"

},

"items": [

{

"sku": "ITEM123",

"name": "Widget A",

"quantityOrdered": 100,

"quantityReceived": 90

},

{

"sku": "ITEM456",

"name": "Widget B",

"quantityOrdered": 50,

"quantityReceived": 50

}

],

"status": "Completed",

"dateCreated": "2024-12-28",

"expectedDelivery": "2024-12-30",

"totalCost": 1500

}

**API Endpoints for POs**

1. **Create a PO:**
   * **POST** /api/purchase-orders
   * Payload:
   * {
   * "poNumber": "PO-20240101-001",
   * "supplier": {
   * "name": "Supplier A",
   * "contact": "supplier@example.com"
   * },
   * "items": [
   * { "sku": "ITEM123", "quantityOrdered": 100 },
   * { "sku": "ITEM456", "quantityOrdered": 50 }
   * ],
   * "expectedDelivery": "2024-12-30"
   * }
2. **Get All POs:**
   * **GET** /api/purchase-orders
   * Optional query parameters for filtering:
     + ?status=pending
     + ?supplier=Supplier A
3. **Get PO Details:**
   * **GET** /api/purchase-orders/:id
4. **Update PO Status:**
   * **PUT** /api/purchase-orders/:id
   * Payload:
   * {
   * "status": "Completed",
   * "items": [
   * { "sku": "ITEM123", "quantityReceived": 90 },
   * { "sku": "ITEM456", "quantityReceived": 50 }
   * ]
   * }
5. **Delete PO:**
   * **DELETE** /api/purchase-orders/:id

**Real-World Use Case Example**

1. **Scenario:**
   * A warehouse manager creates a PO to order 200 widgets from Supplier A.
   * The PO is given the ID PO-20240101-001 and is set to "Pending."
   * Two days later, Supplier A delivers the widgets, but only 190 are received.
   * The manager updates the PO status to "Completed" and logs the received quantity.
   * An audit log entry is created to reflect the discrepancy.
2. **Benefits:**
   * Accurate tracking of ordered and received items.
   * Improved accountability through status updates and audit logs.

Would you like help with coding the API endpoints, database schema, or frontend integration for purchase orders?