**Military Asset Management System**

**Name: K.Abhishek Reddy**

**Email:kr2416@srmist.edu.in**

**Introduction**

In modern military operations, the ability to effectively manage and track critical assets—such as weapons, vehicles, ammunition, and specialized equipment—is essential for mission readiness and operational success. Traditional asset management systems often suffer from fragmented processes, manual record-keeping, and a lack of real-time visibility, which can lead to inefficiencies, misallocations, or even security risks.

To address these challenges, we have developed a robust Military Asset Management System, a full-stack web application designed to streamline the logistics and control of military assets across multiple bases. The system is built with a focus on security, accountability, and operational transparency. It empowers key personnel—such as administrators, base commanders, and logistics officers—to perform role-specific actions including asset purchases, transfers, assignments, and expenditure tracking.

The application features a powerful dashboard to visualize inventory levels and movement trends, a clean and responsive user interface for accessibility across devices, and a secure backend powered by role-based access control (RBAC). By combining modern web technologies with a carefully designed relational database, this system ensures accurate, auditable, and scalable asset tracking aligned with the stringent needs of military logistics.

Whether used for peacetime inventory management or active deployment scenarios, this solution aims to reduce logistical errors, enhance decision-making, and maintain strict accountability across all levels of command.

**1. Project Overview**

The Military Asset Management System is a comprehensive, full-stack web application tailored for defence logistics. It focuses on tracking, managing, and accounting for military-grade assets—vehicles, weapons, ammunition, and other equipment—across multiple military bases. This system introduces transparency and control into military supply chains by digitizing asset lifecycles, from procurement to deployment and consumption. With a dashboard-driven interface, real-time inventory tracking, and strict role-based access control (RBAC), it offers a secure environment for operational logistics.

The system assumes that user accounts are pre-created by administrators and that each user is associated with a specific base, ensuring contextual data isolation. It is also assumed that asset movements are tracked in real-time and that every interaction with the system is authenticated. This makes it ideal for operations requiring accuracy, accountability, and auditability. These assumptions allow for a streamlined implementation, though they may need re-evaluation for scaling to more complex, multi-tenant setups.

Despite its robustness, the system does have a few limitations. It lacks advanced features such as bulk transfer operations for large-scale deployments, file uploads for documentation like receipts or equipment manuals, and sophisticated reporting capabilities. Additionally, the architecture currently supports only a single-tenant model, meaning multiple military organizations cannot use the system in isolation without customization. These constraints, while acceptable for a prototype or MVP, should be addressed for enterprise-level deployments.

**2. Tech Stack & Architecture**

The system uses a modern, full-stack architecture built for performance, scalability, and developer experience. The backend is built with Node.js and Express, written in TypeScript. TypeScript ensures type safety, reducing runtime errors and improving maintainability. Express, being minimalist, allows for fine-grained control over the API design. JWT (JSON Web Token) is employed for stateless authentication, enabling seamless session handling without server-side session storage.

On the frontend, the application uses React combined with TypeScript, giving developers the benefit of reusable components and static typing. TanStack Query (React Query) is used for data fetching, caching, and background synchronization, ensuring that the UI reflects the latest data with minimal manual configuration. Tailwind CSS and shadcn/ui provide a modern, responsive design system. The routing is handled by Wouter, a lightweight and efficient alternative to heavier libraries like React Router.

The database layer leverages PostgreSQL, a powerful and reliable relational database known for its ACID compliance—crucial for accurate asset tracking and transactional integrity. It is hosted using Neon, a scalable, cloud-based PostgreSQL provider. Data operations are handled using Drizzle ORM, which tightly integrates with TypeScript, ensuring type-safe SQL operations and reducing runtime bugs. This architectural combination ensures high performance, scalability, and secure, robust development workflows.

**3. Data Models / Schema**

The system’s data model is well-normalized, with clear entities and relationships that reflect the domain of military logistics. The core entities include:

* Users: Stores authentication details and user roles. Each user is tied to a base (except Admins).
* Bases: Represents military installations or operational units.
* Assets: Catalog of trackable items—categorized by type and description.
* Stocks: Represents the real-time inventory at each base. It captures opening and closing balances, as well as assigned and expended quantities.
* Purchases: Logs each purchase entry tied to a base and asset type.
* Transfers: Records inter-base asset transfers, including source, destination, quantity, and status.
* Assignments: Tracks assignment of assets to personnel, with support for expenditure logging and reasons.
* Logs: An audit trail for all significant user interactions, preserving before and after states for traceability.

The relationships are structured such that users belong to bases (one-to-many), stocks are linked to bases and assets (many-to-one), and all transaction records point back to the user who created them. This design supports scalable querying, allows clear auditability, and ensures that data integrity is preserved across operations. By tracking both assigned and expended statuses, the schema supports a complete lifecycle view of every asset.

**4. RBAC Explanation**

The system implements Role-Based Access Control (RBAC) to restrict functionality and data visibility based on user roles. The three roles are Admin, Commander, and Logistics Officer, each with increasing specificity and decreasing levels of access:

* Admin: Has full access to all resources, across all bases. Admins can manage users, assets, stock levels, audit logs, and system-wide configurations. This role is primarily for system administrators or high-level operational officers.
* Commander: Has limited access, scoped to their specific base. Commanders can initiate transfers and make assignments to personnel. However, they cannot create purchases or manage users. Their dashboard displays metrics specific to their base.
* Logistics Officer: Also base-scoped but with access tailored to procurement and stock replenishment. They can record purchases and view current stock but cannot assign assets or perform cross-base operations.

Enforcement is handled via middleware functions in the Express backend. JWT tokens are issued on login, embedding the user’s role and base ID. Middleware like requireRole or checkBasePermission then verifies user access before executing route handlers. On the frontend, UI components are conditionally rendered based on user role, ensuring that unauthorized buttons or pages are not visible or accessible.

**5. API Logging**

Transaction logging is a central feature of the system’s auditing and accountability strategy. Every major operation—whether it's a login, purchase, asset assignment, or transfer—is automatically logged. These logs include the user ID, type of action, target resource, and both the old and new values of the affected records. This ensures that all changes to the system are traceable and reversible if needed.

Logs are stored in a dedicated logs table, with a timestamp for every entry. The system captures data in JSON format, making it flexible for different types of changes. For example, a purchase log might store the previous and new quantities of an asset in a structured JSON object. This allows auditors or analysts to query changes over time and detect anomalies or errors.

Logging is implemented directly in route handlers, with calls to the logging service after every data-altering operation. This means logs are generated in real-time and stored consistently. In addition, administrators can query the logs via dedicated endpoints with filters such as action type or date range, helping streamline reviews or investigations.

**6. Setup Instructions**

Setting up the system is straightforward, though it requires basic familiarity with Node.js, PostgreSQL, and React. Before starting, ensure that Node.js 18+ and a PostgreSQL database (preferably hosted on Neon) are available.

Environment Setup:

* Create a .env file and define:
  + DATABASE\_URL (PostgreSQL connection string)
  + JWT\_SECRET (used for signing authentication tokens)
* Install backend dependencies using:

bash

CopyEdit

npm install

Database Initialization:

* Run database migration scripts:

bash

CopyEdit

npm run db:migrate

npm run setup:db

This sets up all required tables and seeds initial data such as default users and assets.

Running the App:

* Start the backend server:

bash

CopyEdit

npm run dev

* By default, the server runs on port 5000 and is ready to accept API calls.

Default Credentials:

* Admin: admin@military.gov / admin123
* Commander: commander@fortbragg.mil / commander123
* Logistics: logistics@pendleton.mil / logistics123

For the frontend:

* Navigate to the React project directory.
* Run:

bash

CopyEdit

npm install

npm start

* Open the application in a browser and log in using the above credentials.

**7. API Endpoints**

The API is RESTful, predictable, and grouped by resource type. Authentication and core functionalities are exposed through clean, structured endpoints.

Authentication:

* POST /api/auth/login: Authenticate user and return a JWT.
* GET /api/auth/me: Get the currently logged-in user's data.
* POST /api/auth/register: Register a new user (Admin only).

Core Resources:

* GET /api/bases: List all military bases.
* GET /api/assets: List all assets (weapons, vehicles, etc.).
* GET /api/stocks?baseId=X: View current stock for a specific base.

Transactions:

* POST /api/purchases: Record a new asset purchase.
* POST /api/transfers: Initiate a transfer between bases.
* PATCH /api/transfers/:id/status: Update transfer status (e.g., delivered, rejected).
* POST /api/assignments: Assign assets to personnel.

Analytics:

* GET /api/dashboard/metrics?baseId=X: Fetch base-specific metrics for dashboard.
* GET /api/dashboard/activity?limit=10: Get recent system activity (last N actions).

Audit:

* GET /api/logs?actionType=X&startDate=Y: Retrieve audit logs with filters for action type and date.

These endpoints follow RESTful principles and support easy integration with frontend components or external systems. All write operations are secured using JWT and role checks, ensuring system integrity and role isolation.