Military Asset Management System (MAMS)

Project Documentation

1. Project Overview

1.1 Description

The Military Asset Management System (MAMS) is a full-stack web application designed to provide commanders and logistics personnel with a centralized platform to manage the movement, assignment, and expenditure of critical assets across multiple military bases. The system offers a secure, role-based solution to ensure data integrity, streamline logistics, enhance transparency, and enforce accountability.

1.2 Core Capabilities

- **Asset Lifecycle Tracking:** Tracks assets from purchase to expenditure, including inter-base transfers and personnel assignments.
- **Centralized Dashboard:** Provides key performance indicators (KPIs) such as opening/closing balances and net asset movements, with robust filtering capabilities.
- **Role-Based Access Control (RBAC):** Ensures that users can only view data and perform actions appropriate to their designated role and assigned base.
- **Transactional History:** Maintains an immutable log of all asset movements for auditing and accountability.

1.3 Assumptions

- Users are pre-registered in the system by an administrator. There is no public-facing user registration feature.
- The system operates on the assumption that asset movements are recorded after the fact and does not provide real-time GPS tracking.
- The initial asset list and stock levels are seeded into the database to establish a baseline.

1.4 Limitations

- The current version does not include modules for asset maintenance, depreciation, or decommissioning.
- The API logging is currently configured for console output during development and would require integration with a dedicated logging service for production use.

- The frontend UI/UX is functional but designed for simplicity and can be further enhanced.

2. Tech Stack & Architecture

The application follows a modern client-server architecture.

- **Frontend (Client):**
    - **Framework: React (with Vite)** - Chosen for its component-based architecture, which promotes reusable and maintainable code, and its vast ecosystem. Vite provides a superior, fast development experience.
    - **Styling: Tailwind CSS** - A utility-first CSS framework that allows for rapid development of clean, responsive, and minimalistic user interfaces directly within the component markup.
- **Backend (Server):**
    - **Framework: Node.js & Express.js** - Chosen for its high performance in I/O-bound operations (like database queries), its massive library support via npm, and the ability to use JavaScript/TypeScript across the entire stack.
- **Database (DB):**
    - **Type: PostgreSQL (Relational/SQL)**
    - **Justification:** A relational database was chosen because the application's data is highly structured and relational.
        - **Data Integrity:** Foreign key constraints between tables (e.g., assets to bases, users to roles) and ACID compliance are critical for an auditable system of record like this, preventing data corruption.
        - **Complex Queries:** The dashboard requires complex aggregations (SUM, CASE) and joins across multiple tables to calculate metrics. SQL is purpose-built for such queries, making them efficient and straightforward to write.
        - **Schema Consistency:** The data structure for assets, users, and bases is well-defined, making it a perfect fit for a structured SQL schema.

3. Data Models / Schema

The PostgreSQL database is the core of the system, with the following key tables:

- **roles**: Stores the defined user roles (Admin, Base Commander, Logistics Officer).
- **bases**: Contains information about each military base.
- **equipment_types**: Defines categories for assets (Vehicle, Weapon, etc.).
- **users**: Stores user credentials, their hashed password, and foreign keys linking to their role and assigned base.
- **assets**: The master table for every unique asset, containing its name, serial number, type, and current location (current_base_id).
- **asset_movements**: **The central ledger of the application.** This table logs every single transaction. A movement_type enum field (purchase, transfer_in, transfer_out, assignment, expenditure) is used to classify each record. All dashboard metrics are calculated by querying this table, ensuring a single source of truth.

Relationships:

- A user has one role and is assigned to one base (except Admins).
- An asset has one equipment_type and is located at one current_base_id.
- Every record in asset_movements is linked to one asset. It can also be linked to a from_base, to_base, or assigned_to_user_id depending on the transaction type.

4. RBAC Explanation

Role-Based Access Control (RBAC) is enforced on the **backend** to ensure security.

- **Roles:**
    - Admin: Full access to all data and operations across all bases.
    - Base Commander: Access is limited to their assigned base. Can assign/expend assets.
    - Logistics Officer: Access is limited to their assigned base. Can purchase/transfer assets.
- **Enforcement Method:**

1. When a user logs in, the backend validates their credentials and generates a **JSON Web Token (JWT)**.
2. This JWT contains a payload with the user's userId, role, and baseId.

3. The frontend stores this token and includes it in the Authorization header of every subsequent API request.

4. The backend uses custom **Express middleware** (auth.middleware.js) on protected routes. This middleware inspects the JWT on incoming requests to verify the user's role against a list of allowed roles for that specific endpoint.

5. If the user's role is not permitted, the API returns a 403 Forbidden error, preventing the action.

5. API Logging

API transaction logging is a non-functional requirement designed for auditing and debugging.

- **How it is handled:** The project is designed to include a simple middleware on the backend. This middleware would intercept all transactional API requests (e.g., POST, PUT, DELETE on routes like /api/assignments or /api/transfers).
- **Information Logged:** For each transaction, the middleware would log key information to the console, such as:
    - Timestamp
    - Request Method (e.g., POST)
    - Request URL (e.g., /api/transfers)
    - Authenticated User ID (extracted from the JWT)
    - Request Body (the data being submitted)
- **Production Use:** For a live production environment, this console logging would be replaced by a more robust logging service (like Winston, Pino, or a cloud-based log aggregator) to store logs persistently and allow for searching and analysis.

6. Setup Instructions

6.1 Local Development

1. **Prerequisites:** Node.js, npm, and Git must be installed.
2. **Clone Repository:** git clone <repository_url>
3. **Setup Backend:**
    - Navigate to the /server directory.
    - Run npm install.

    - Create a .env file and provide the DATABASE_URL (using the **External URL** from Render), a JWT_SECRET, and PORT=3001.
    - Run npm run dev to start the server.
4. **Setup Frontend:**
    - Navigate to the /client directory.
    - Run npm install.
    - Run npm run dev. The application will open at http://localhost:5173.
5. **Seed Database:** With the backend running, visit http://localhost:3001/api/debug/seed-database in your browser one time to populate the database.

6.2 Production Deployment

- **Backend (Render):** Deploy the /server directory as a Node.js Web Service. Set the DATABASE_URL (using the **Internal URL**) and JWT_SECRET as environment variables.
- **Frontend (Vercel):** Deploy the /client directory. Set the VITE_API_URL environment variable to the public URL of your deployed Render backend (e.g., https://mams-server.onrender.com/api).

7. API Endpoints (Key Examples)

- POST /api/auth/login
    - **Description:** Authenticates a user.
    - **Payload:** { "username": "...", "password": "..." }
    - **Response:** { "token": "...", "user": { ... } }
- GET /api/dashboard/metrics
    - **Description:** Fetches calculated metrics for the dashboard.
    - **Auth:** Admin, Base Commander, Logistics Officer
    - **Query Params:** startDate, endDate, baseId, equipmentTypeId
    - **Response:** { "openingBalance": ..., "closingBalance": ..., ... }
- POST /api/assignments/assign
    - **Description:** Creates a new asset assignment record.

    - **Auth:** Admin, Base Commander
    - **Payload:** { "asset_id": ..., "assigned_to_user_id": ..., "notes": "..." }
    - **Response:** The newly created movement record.
- POST /api/transfers
    - **Description:** Creates transfer_in and transfer_out records and updates the asset's location.
    - **Auth:** Admin, Logistics Officer
    - **Payload:** { "asset_id": ..., "to_base_id": ..., "quantity": ..., "notes": "..." }
    - **Response:** { "message": "Transfer completed successfully" }
- GET /api/debug/seed-database
    - **Description: (Development Only)** Wipes and re-seeds the entire database to a clean, initial state.
    - **Auth:** None
    - **Response:** A success or failure message.