Military Asset Management System

Aakash Singh

Contents

Section 1 Purpose

The Military Asset Management System (MAMS) is designed to help military commanders and logistics personnel manage the movement, assignment, and expenditure of critical assets such as vehicles, weapons, and ammunition across multiple bases. The system provides transparency, streamlines logistics, and ensures accountability with role-based access controls.

1.1 Key Objective

- Track Opening Balances, Closing Balances, and Net Movements (Purchases + Transfers In - Transfers Out)
- Record asset assignments and expenditures.
- Facilitate asset transfers between bases with a clear history of movements.
- Ensure role-based access control for security and accountability.

1.2 Assumptions

- All bases are centrally managed in the system.
- Asset IDs are unique across the organization.
- Personnel accessing the system have a valid role assigned.

1.3 Limitations

- The system does not handle predictive analytics for asset needs.
- Offline mode is not supported; real-time internet connectivity is required.

Section 2 Tech Stack & Architecture

2.1 Frontend

The frontend of the Military Asset Management System is developed using **React.js**, chosen for its flexibility, component-based architecture, and efficiency in building responsive user interfaces. The frontend stack includes:

- **React.js** – Provides a modular, maintainable structure for building complex UIs.
- **Bootstrap** – Ensures a responsive and mobile-friendly design with pre-built UI components.
- **React Router** – Enables seamless client-side routing between pages like Dashboard, Purchases, Transfers, and Assignments.
- **Axios** – Handles API calls to the backend in a clean and promise-based manner.
- **react-hot-toast** – Provides interactive, non-intrusive notifications for user actions like purchases, transfers, or errors.

This combination allows the application to be highly interactive, user-friendly, and responsive across devices.

2.2 Backend

The backend of the Military Asset Management System is developed using **Spring Boot**, chosen for its robustness, scalability, and ease of building secure RESTful APIs.
Key components and libraries used:

- **Spring Boot** – Provides a streamlined framework for building enterprise-grade applications with minimal configuration.
- **Spring Security with JWT (JSON Web Token)** – Ensures secure authentication and authorization. JWT allows stateless, role-based access control (RBAC) to protect endpoints and sensitive data.
- **RESTful APIs** – Exposes endpoints for core operations such as purchases, transfers, assignments, and asset tracking.
- **Middleware / Interceptors** – Enforce RBAC at the API level, ensuring that users can only access resources allowed for their role.

This backend stack ensures high security, scalability, and maintainability while allowing seamless integration with the React frontend through RESTful APIs.

2.3 Database

The system uses **MySQL** as the primary relational database to store and manage all structured data related to assets, bases, purchases, transfers, assignments, and users.
Key reasons for choosing MySQL:

- **Relational structure** – Supports structured data with relationships, ensuring data consistency for tracking assets and their movements.

- **ACID compliance** – Guarantees reliable transactions for critical operations like purchases, transfers, and assignments.
- **Scalability and performance** – Can handle large volumes of data efficiently with indexing and optimized queries.

AWS S3 Bucket is used for storing asset-related images and documents, such as:

- Photos of vehicles, weapons, or other assets.
- Purchase invoices or documentation for audit purposes.

Integration with Backend:

- Images and documents are uploaded to S3 via the backend and their URLs are stored in the MySQL database.

Section 3 Data Models / Schema

The Military Asset Management System uses a relational database (MySQL) to store structured data related to assets, categories, items, and users. Below are the core tables and their schema.

3.1 Category Table (tbl_category)

Stores categories for assets/items.

Column Name	Type	Constraints	Description
id	BIGINT	PK, Auto-increment	Internal primary key
categoryId	VARCHAR	Unique	Unique identifier for the category
name	VARCHAR	Unique	Category name
description	VARCHAR	Optional	Description of the category
bgColor	VARCHAR	Optional	Background color for UI display
imageUrl	VARCHAR	Optional	URL for category image stored in S3
createdAt	TIMESTAMP	Auto-generated, non-updatable	Timestamp when the category was created
updatedAt	TIMESTAMP	Auto-generated	Timestamp when the category was last updated

3.2 Item Table (tbl_items)

Stores information about items/assets linked to a category.

Column Name	**Type**	**Constraints**	**Description**
id	**BIGINT**	**PK, Auto-increment**	**Internal primary key**
itemId	**VARCHAR**	**Unique**	**Unique identifier for the item**
name	**VARCHAR**	**Required**	**Item name**
price	**DECIMAL**	**Optional**	**Price of the item**
description	**VARCHAR**	**Optional**	**Description of the item**
imgUrl	**VARCHAR**	**Optional**	**URL for item image stored in S3**
category_id	**BIGINT**	**FK → tbl_category(id)**	**Foreign key linking item to category**
createdAt	**TIMESTAMP**	**Auto-generated, non-updatable**	**Timestamp when the item was created**
updatedAt	**TIMESTAMP**	**Auto-generated**	**Timestamp when the item was last updated**

Relationships:

- **Many Items belong to one Category (Many-to-One).**
- **Deletion of a category is restricted if items exist (OnDeleteAction.RESTRICT).**

3.3 User Table (tbl_user)

Stores system users with authentication and role information.

Column Name	Type	Constraints	Description
id	BIGINT	PK, Auto-increment	Internal primary key
userId	VARCHAR	Unique	Unique identifier for the user
name	VARCHAR	Required	Full name of the user
email	VARCHAR	Unique	User email for login and notifications
password	VARCHAR	Required	Encrypted password
role	VARCHAR	Required	Role of the user (Admin, Commander, Logistics)
createdAt	TIMESTAMP	Auto-generated, non-updatable	Timestamp when the user was created
updatedAt	TIMESTAMP	Auto-generated	Timestamp when the user was last updated

3.4 Relationships Overview

CategoryEntity (1) ——< ItemEntity (Many)

UserEntity is independent and manages role-based access

- **Category → Item: One-to-Many relationship.**

- **User: No direct relationship to Category or Item, but interacts through transactions like purchases, transfers, and assignments.**

Section 4 RBAC Explanation

Role	**Access Level**
Admin	Full access to all data, create/update/delete assets, transfers, purchases.
Base Commander	View and manage Dashboard and purchases for their assigned base.
Logistics Officer	Can record and View and manage Dashboard and purchases for their assigned base.

- RBAC is implemented as a **middleware** in the backend.
- Each API endpoint checks the JWT token for role and assigned base.
- Unauthorized requests return HTTP 403 Forbidden.

Section 5 Setup Instructions

This section guides you through setting up the Military Asset Management System locally or on a server.

Before starting, ensure the following software is installed on your system:

- **Java 17+** (for Spring Boot)
- **Maven** (for building Spring Boot project)
- **Node.js 18+** (for React frontend)
- **MySQL 8+** (or compatible version)
- **AWS Account** (for S3 bucket configuration)
- **Git** (for cloning the repository

Step 1 : Clone the **backend** repository:
Step 2 : git clone https://github.com/AakashSingh-oo7/mams-backend.git

Step 3 : Clone the **frontend** repository:

Step 4 : git clone https://github.com/AakashSingh-oo7/mams-frontEnd.git

Step 5 : Open the backend project in your preferred IDE (e.g., IntelliJ IDEA)

Step 6 : Navigate to src/main/resources/application.properties.

Step 7 : Add your environment-specific keys and credentials:

Step 8 : spring.datasource.url=jdbc:mysql://<DB_HOST>:3306/mams_db

```
spring.datasource.username=<DB_USERNAME>
spring.datasource.password=<DB_PASSWORD>
jwt.secret=<JWT_SECRET>
aws.s3.bucket=<S3_BUCKET_NAME>
aws.access.key=<AWS_ACCESS_KEY>
aws.secret.key=<AWS_SECRET_KEY>
```

Step 9 : Build and run the backend using your IDE (IntelliJ, Eclipse) or via command line:

```
mvn clean install
mvn spring-boot:run
```

Backend APIs will be accessible at: http://localhost:8080/

Step 10 : Navigate to the frontend directory:cd mams-frontEnd Install dependencies:

Step 11 : npm install

Step 12 : Create a .env file in the frontend root folder and add the backend API URL:

Step 13 : REACT_APP_API_BASE_URL=http://localhost:8080/api

Step 14 : Start the frontend: npm start

Step 15 : The frontend will run at:

Step 16 : http://localhost:3000

Section 6 API Endpoints

The backend is implemented using **Spring Boot**, exposing RESTful APIs under the base path /api/v1.0. All endpoints are secured using **JWT authentication**, and access is controlled based on roles: **Admin, Base Commander, Logistics Officer**.

7.1 Authentication

Endpoint	Method	Description	Request Body	Response	Access
/api/v1.0/login	POST	Authenticates a user and returns a JWT token	{ "email": "user@example.com", "password": "password" }	{ "email": "user@example.com", "role": "Admin", "token": "jwt-token" }	Public
/api/v1.0/encode	POST	Encode a plain password (for testing)	{ "password": "plaintext" }	"hashed_password"	Public

Notes:

- JWT token must be included in the Authorization header for all protected endpoints:
- Authorization: Bearer <jwt-token>

7.2 Category Management

Endpoint	Method	Description	Access
/api/v1.0/categories	GET	Fetch all categories	Admin, Base Commander, Logistics
/api/v1.0/admin/categories	POST	Add a new category with optional image upload	Admin
/api/v1.0/admin/categories/{categoryId}	DELETE	Delete a category by ID	Admin

Request Example (Add Category):

- Multipart form data:
  - category → JSON string for category (name, description, bgColor, etc.)
  - file → Image file

7.3 Item / Asset Management

Endpoint	Method	Description	Access
/api/v1.0/admin/items	GET	Fetch all items	Admin
/api/v1.0/admin/items	POST	Add a new item with optional image	Admin
/api/v1.0/admin/items/{itemId}	DELETE	Delete an item by ID	Admin

Request Example (Add Item):

- Multipart form data:
  - item → JSON string for item (name, price, description, categoryId, etc.)
  - file → Image file

7.4 User Management

Endpoint	Method	Description	Access

Endpoint	Method	Description	Access
/api/v1.0/admin/register	POST	Register a new user	Admin
/api/v1.0/admin/users	GET	Fetch all users	Admin
/api/v1.0/admin/users/{id}	DELETE	Delete a user by ID	Admin