MAMS

(Military Asset Management System)

Submittted By : -

Deependra Kumar

9818117453

kushdep017@gmail.com

https://github.com/kushdep

Resume -

https://docs.google.com/presentation/d/1i7HSbiNujLGAhXZ0v2eKKWte8oUlYcMMjxT3qWTJPfE/edit#slide=id.p

Project Overview

1.1 Description

The Military Asset Management System (MAMS) is a centralized web application designed to enable administrators and officers to handle the complete lifecycle of assets, from procurement, assignment, and expenditure of assets. It also efficiently manages, tracks, and records transfer of military assets across multiple bases — within a secure, role-based environment.

1.2 Objectives

1.2.1 Purchase Management

- Enables users to record asset purchases, whether they are new acquisitions or old assets.
- Allows entry of detailed purchase information such as item type, quantity.
- Ensures all purchases are reflected in the purchase history and dashboard and to the base it belongs to.

1.2.2 Asset Management

- Allows available assets to be assigned to individual soldiers
- Tracks the utilization and expenditure of assets.
- Supports role-based tracking so only authorized personnel can assign or set asset expenditure status.

1.2.3 Asset Transfer

- Facilitates inter-base transfer of assets between different Bases.
- The receiving base can update the status of transferred assets upon receipt.
- Ensures transparent movement tracking across all bases with a complete audit history and record of assets.

1.2.4 Dashboard & Reporting

- Provides a centralized dashboard that displays asset summaries, including:
    - Opening balance, closing balance, and net movement of assets
    - Details of purchases, transfer-ins, and transfer-outs
    - Dynamic date and category filters for customized insights
- Enables commanders and logistic officers to quickly analyze operational data and make informed decisions.

1.3 Assumptions

- Each **base** operates as an independent unit with its own asset inventory.
- The data entered by users (such as purchase details, transfer records, and assignments) is assumed to be **accurate and verified** by authorized personnel.
- Once a **purchase, assignment, expenditure, or transfer transaction** is completed, it **cannot be deleted or edited**, ensuring data integrity and auditability.
- Only **three asset categories** exist in the system, predefined by the admin for standardization.
- **Transfers** between bases are a **two-step process**:
    - The sending base can initiate a transfer only if the receiving base has requested it through external communication.
    - Once the transfer is done, the receiving base must **acknowledge receipt** in the system.
- There only one admin present which can access to any base and perform any action

1.4 Limitations

- **No Edit or Delete Option for Transactions**
  Once a purchase, assignment, expenditure, or transfer is finalized, it cannot be modified or removed. Any correction must be made by creating a new transaction, which may increase data volume over time.
- **Manual Coordination for Transfers**
  The inter-base transfer process requires **manual communication** between bases before initiating a transfer, as there is no built-in request/approval workflow.
- **Single Admin Architecture**
  Only one global administrator account exists, which can lead to a **single point of dependency** for system-wide management.
- **No Automated Notifications Between Bases**
  Bases do not receive automatic alerts or in-app messages for incoming transfer requests; acknowledgment must be done manually after communication.
- **Scaling**
  Large inventory may cause some lagging and ui issues in the system

Tech Stack & Architecture

2.1.1 Front-end Technology

Library	Purpose
React	Core UI framework
Redux/Redux Toolkit	State Management
React Router	Routing between pages
React Hot toast	Notifications
Axios	API calls
Bootstrap	Styling and responsiveness
Sentry	Error monitoring and performance tracking

2.1.2 Back-end Technology

Technology / Library	Purpose
Node.js	Server side javascript runtime
Express js	Web framework for routing, middleware and handling HTTPS req/res
Mongoose	Library for mongo db to structure data and maange schemas
Winston	Logging system for requests, errors and transaction logs
Winston-mongodb	To Save logs in to mongodb
dotenv	Enviroment variable management to securely store credentials and configuration
Bcrrypt/jwt	Password hashing and authentication token handling
Joi	Schema validation fot Api requests to ensure incoming data is correct and secure
Agenda	Job scheduling library for node js to perform periodic tasks
CORS	Middleware to handle cross origin requests from front-end clients

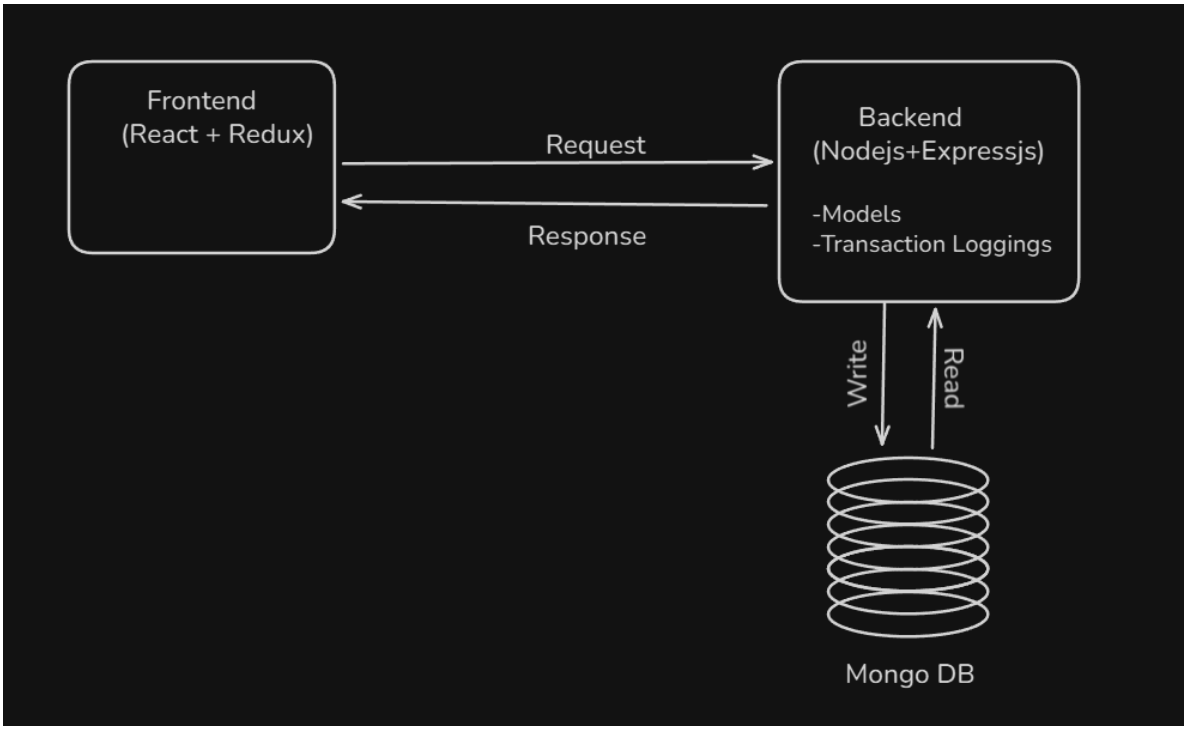
2.2 Database used

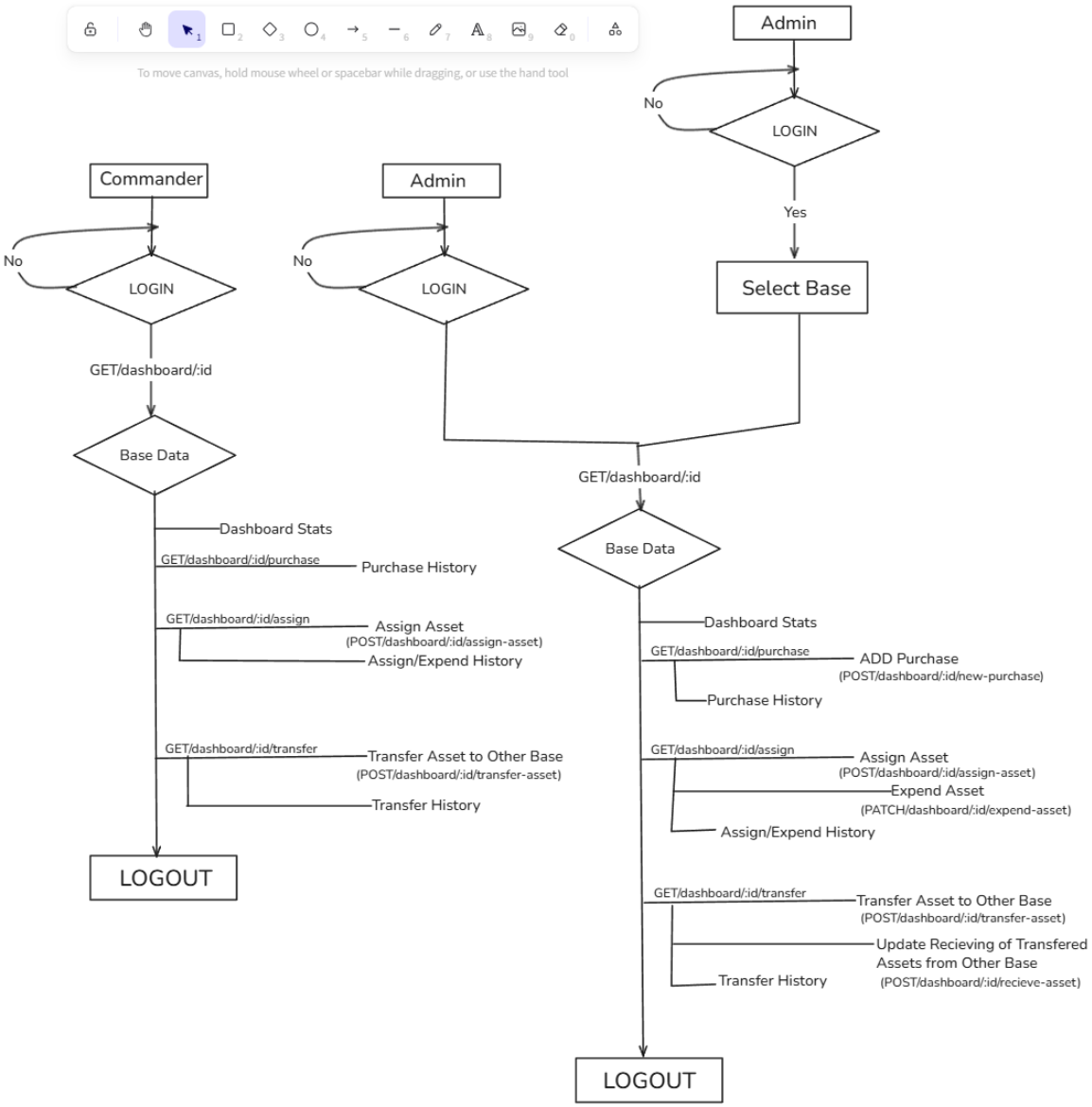
NO SQL Database - MONGO DB

Why MongoDB

1. **Flexible Schema**
   Mongo DB let's store related sub documents in one document or embedded sub documents reducing complex joins which can easily handle changing asset structures and evolving fields without altering the database, which is perfect for a dynamic dashboard.
2. **Mongo DB is a very great Option for Sharding / Horizontal Scaling**
3. **Scalability & Growth**
   Indexes can be applied on any field, including nested fields in JSON documents. Works naturally with **document-oriented queries** and aggregation pipelines. Helps fetch large collections efficiently
4. **Fast Bulk Reads**
   Fetching large datasets for the dashboard is efficient, even if filtering is done on the frontend.
5. **Aggregation Framework**
   Allows complex calculations (totals, averages, balances, trends) in the database if needed in future which reduces computation and improves performance.
6. **Frontend-Friendly Data Format**
   - Stores data as JSON-like documents, mapping naturally to JavaScript objects.
   - Makes filtering, sorting, and displaying data in React straightforward.
7. **Easy Querying & Syntax**
   Supports intuitive queries and aggregation functions, simplifying data fetching and potential future backend computations.

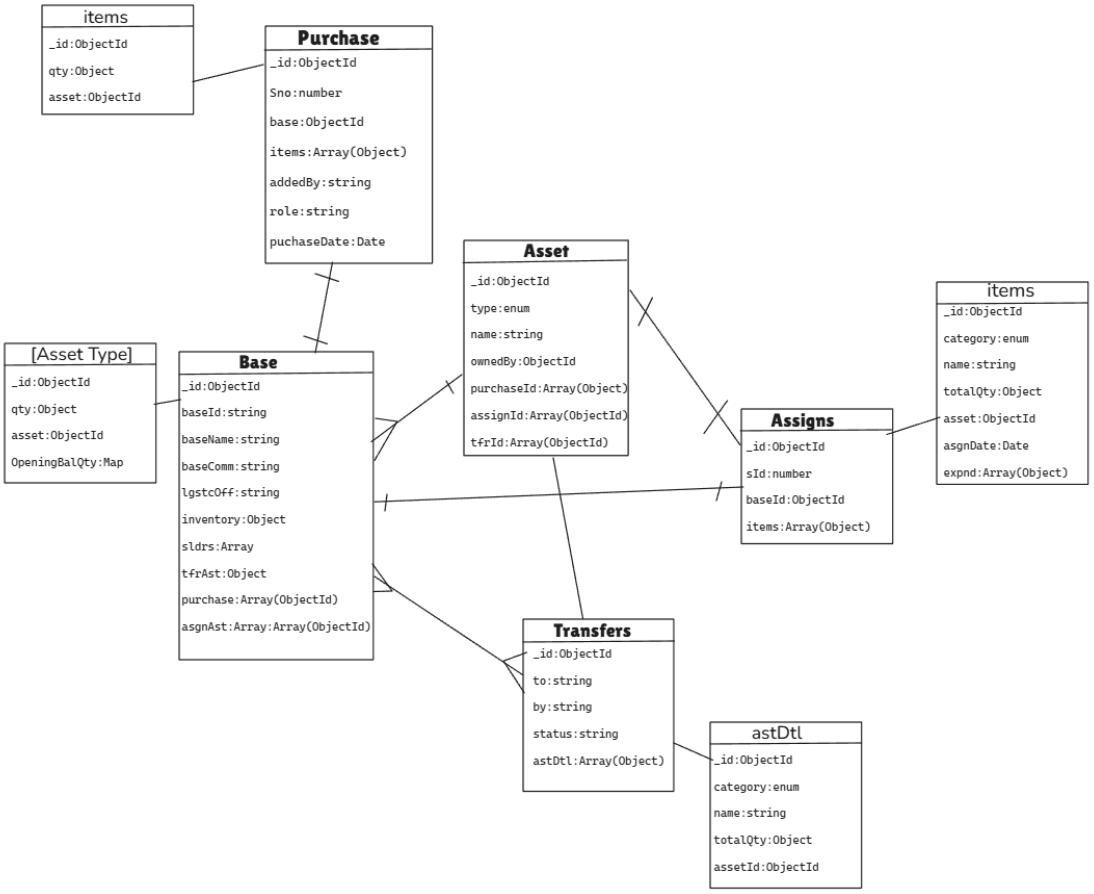
2.3 Architecture




To move canvas, hold mouse wheel or spacebar while dragging, or use the hand tool
Admin
No
LOGIN
Yes
Select Base
Commander
No
LOGIN
GET/dashboard/:id
Base Data
Dashboard Stats
GET/dashboard/:id/purchase
Purchase History
GET/dashboard/:id/assign
Assign Asset
(POST/dashboard/:id/assign-asset)
Assign/Expend History
GET/dashboard/:id/transfer
Transfer Asset to Other Base
(POST/dashboard/:id/transfer-asset)
Transfer History
LOGOUT
Admin
No
LOGIN
GET/dashboard/:id
Base Data
Dashboard Stats
GET/dashboard/:id/purchase
ADD Purchase
(POST/dashboard/:id/new-purchase)
Purchase History
GET/dashboard/:id/assign
Assign Asset
(POST/dashboard/:id/assign-asset)
Expend Asset
(PATCH/dashboard/:id/expend-asset)
Assign/Expend History
GET/dashboard/:id/transfer
Transfer Asset to Other Base
(POST/dashboard/:id/transfer-asset)
Update Recieving of Transfered Assets from Other Base
(POST/dashboard/:id/recieve-asset)
Transfer History
LOGOUT

Data Models/Schemas

3.1 Data Models & Relationships



RBAC Explanation

4.1 Roles & Access Levels

Sno	Actions	ADMIN	Logistic Officer	Commander
i	Access to all Bases	Yes	X	X
ii	Dashboard Stats	Yes	Yes	Yes
iii	Add New Purchase of Asset	Yes	Yes	X
iv	See Purchase History	Yes	Yes	Yes
v	Assign Assets to soldiers	Yes	Yes	Yes
vi	Update Expended Assets	Yes	Yes	X
vii	See Assign & Expend History	Yes	Yes	Yes
viii	Transfer to other Base	Yes	Yes	Yes
ix	Confirm The Transfer Asset	Yes	Yes	X
x	See Transfer History	Yes	Yes	Yes

4.2 Enforcement Methods

1. Token-Based Role Enforcement
   - **Mongo DB** During the **user authentication** process, a JWT (JSON Web Token) is generated that includes a role key ("AD" for Admin, "COMM" for commander, "LGOF" for logistic officer)
   - On every incoming request to protected endpoints, middleware verifies
     - The validity of the token.
     - The user's role inside the token payload.
   - Each API route checks whether the user's role is authorized to access that resource or perform that action.

2. Data-Level Role Validation
   - Each **Base document** stores the **email IDs of both the Commander and Logistic Officer** associated with it.
   - When performing operations like assign, transfer, or update, the backend verifies whether the current user's email matches one of these authorized roles for that base.

Logging & Monitoring

5.1 API Logging

To ensure better observability and debugging, API logging has been implemented using **Winston**, a Node.js logging library

- **Info-level logs** are printed in the **console** to track API requests and general application flow during development and runtime.
- **Error-level logs** are stored in a dedicated **MongoDB collection** for centralized tracking and post-analysis.

This setup allows real-time monitoring during development while maintaining a persistent record of errors in production for audit and troubleshooting.

5.2 Transaction Logging

The system implements **transaction-level logging** using **Winston** to record all key operations performed within the application — such as asset assignments, expenditure, transfers, and updates.

- Every successful or failed **transaction** is logged with details including the **user email**, **action performed**, **affected records**, and **timestamp**.
- **Info-level logs** are generated for valid and completed transactions and are printed in the **console** for real-time tracking.
- **Error-level logs** (in case of failed or invalid transactions) are automatically stored in the **MongoDB collection** via the **Winston MongoDB transport**.

5.3 Monitoring

The application is integrated with **Sentry** for real-time monitoring, error tracking, and performance analysis of frontend

- Sentry captures **JavaScript errors**, **API call failures**, and **UI performance issues** such as slow renders or navigation delays.
- **I**t provides detailed insights including stack traces, user sessions, and browser information,helping identify issues directly in production.

Setup Instructions

6.1 Roles Front-end

Prerequisites

- **Node.js (v16 or higher)** installed
- **npm** package manager
- Backend server running (to connect API endpoints)

Step 1:- Clone the Repository

git clone https://github.com/kushdep/Military-Asset-Management-System.git

cd client

Step 2:- Install Dependencies

npm install

Step 3:- Configure Environment Variables

Create a .env file in the root of the frontend directory and add your backend server URL:

VITE_SERVER_URL=http://localhost:8181

Step 4:- Run the Frontend

npm start

6.2 Back-end

Prerequisites

- **Node.js (v16 or higher)** installed
- **npm** package manager

Step 1:- Clone the Repository

git clone https://github.com/kushdep/Military-Asset-Management-System.git

cd client

Step 2:- Install Dependencies

npm install

Step 3:- Configure Environment Variables

Create a .env file in the root of the frontend directory and add your backend server URL:

PORT =8181

JWT_SECRET=2JN23HS9AH30Q34D3DA

MONGO_URI=……

Step 4:- Run the Backend

npm run dev

6.3 Database

Step 1:- Download and open **MongoDB Compass**.

Step 2:- Connect using your MongoDB URI.

Step 3:- Choose your database (e.g., militaryDB).

Step 4:- Click on **"Create Collection"** → (bases)

Step 5:- Click **"Import Data"** → select MAMS.bases.json file.

Step 6:- Repeat for each mock data file (assets,transfers,purchases,assigns,users)