Military Asset Management System - Technical Documentation

Version: 1.0
Date: June 19, 2025
Author: Development Team
Document Type: Technical Specification & Implementation Guide

---

Table of Contents

---

1. Project Overview

1.1 Description

The Military Asset Management System is a comprehensive web application designed to manage military assets across multiple bases with role-based access control. The system enables commanders and logistics personnel to track asset movements, manage assignments, record expenditures, and maintain complete audit trails.

1.2 Core Objectives

- **Asset Lifecycle Management**: Track assets from procurement to disposal
- **Multi-Base Operations**: Support operations across multiple military installations
- **Role-Based Security**: Ensure appropriate data access based on user roles
- **Audit Compliance**: Maintain complete transaction logs for accountability
- **Real-Time Reporting**: Provide dashboard metrics for operational insights

1.3 Key Features

- **Dashboard**: Real-time metrics with filtering capabilities
- **Asset Management**: CRUD operations for military assets

- **Purchase Tracking**: Record and manage asset procurement
- **Transfer System**: Inter-base asset movement with validation
- **Assignment Management**: Personnel asset allocation tracking
- **Expenditure Tracking**: Asset consumption and disposal records
- **Audit Trail**: Complete API logging for compliance

1.4 Assumptions

- **Database Access**: PostgreSQL database with appropriate user permissions
- **Network Environment**: Local or internal network deployment
- **User Training**: Personnel familiar with web-based applications
- **Data Integrity**: Trusted users with proper authorization
- **Backup Strategy**: Regular database backups maintained externally

1.5 Limitations

- **Concurrent Users**: Optimized for up to 100 concurrent users
- **File Uploads**: No support for asset images or documents
- **Offline Mode**: Requires network connectivity for all operations
- **Mobile Optimization**: Responsive design but not native mobile app
- **Reporting**: Basic dashboard metrics, no advanced report generation
- **Integration**: Standalone system without external API integrations

---

2. Tech Stack & Architecture

2.1 Architecture Overview

The system follows a three-tier architecture pattern:

- **Presentation Layer**: Next.js frontend (Port 3000)
- **Application Layer**: Express.js backend (Port 4000)

- **Data Layer**: PostgreSQL database (Port 5432)

2.2 Backend Technology Stack

Express.js with TypeScript

- **Choice Rationale**: Mature, lightweight framework with extensive middleware ecosystem
- **Benefits**: Fast development, strong community support, excellent for RESTful APIs
- **TypeScript Integration**: Enhanced code quality, compile-time error checking, better IDE support

Prisma ORM

- **Choice Rationale**: Type-safe database access with excellent PostgreSQL integration
- **Benefits**: Automatic query generation, database migrations, strong typing
- **Features**: Query optimization, connection pooling, transaction support

Authentication & Security

- **JWT (jsonwebtoken)**: Stateless authentication with secure token generation
- **bcryptjs**: Password hashing with salt for enhanced security
- **CORS**: Cross-origin resource sharing configuration
- **Middleware-based RBAC**: Role-based access control at API level

2.3 Frontend Technology Stack

Next.js 15 with TypeScript

- **Choice Rationale**: Modern React framework with server-side rendering capabilities
- **Benefits**: Excellent developer experience, automatic code splitting, production optimizations
- **App Router**: New Next.js 13+ app directory structure for better organization

TailwindCSS

- **Choice Rationale**: Utility-first CSS framework for rapid UI development
- **Benefits**: Responsive design utilities, consistent design system, small bundle size
- **Mobile-First**: Responsive design approach for multi-device compatibility

React Query (@tanstack/react-query)

- **Choice Rationale**: Powerful server state management with caching capabilities
- **Benefits**: Background refetching, optimistic updates, error handling
- **Performance**: Reduced API calls through intelligent caching

2.4 Database Technology Stack

PostgreSQL

- **Choice Rationale**: Enterprise-grade relational database with ACID compliance
- **Benefits**: Excellent for audit trails, complex queries, data integrity
- **Features**: Foreign key constraints, transactions, indexing for performance
- **Scalability**: Supports complex relationships and large datasets

---

3. Data Models / Schema

3.1 Entity Relationship Overview

The database schema consists of core entities with well-defined relationships:

- **Users**: Connected to Roles (Many-to-One) and Bases (Many-to-One)
- **Bases**: Connected to Assets, Purchases, Transfers, Assignments, Expenditures (One-to-Many)
- **Assets**: Connected to all transaction types (One-to-Many)

3.2 Core Tables

Users Table

```
Users {
  id: Int (Primary Key, Auto-increment)
  email: String (Unique)
  name: String
  password: String (Hashed)
  roleId: Int (Foreign Key -> Roles.id)
  baseId: Int (Foreign Key -> Bases.id, Nullable)
  createdAt: DateTime
  updatedAt: DateTime
}
```

Roles Table

```
Roles {
  id: Int (Primary Key, Auto-increment)
  name: String (Unique)
  createdAt: DateTime
  updatedAt: DateTime
}
```

Bases Table

```
Bases {
  id: Int (Primary Key, Auto-increment)
  name: String (Unique)
  location: String
  createdAt: DateTime
  updatedAt: DateTime
}
```

Assets Table

```
Assets {
  id: Int (Primary Key, Auto-increment)
  type: String
  serial: String (Unique)
  description: String (Nullable)
  baseId: Int (Foreign Key -> Bases.id)
  createdAt: DateTime
  updatedAt: DateTime
}
```

3.3 Transaction Tables

Purchases Table

```
Purchases {
  id: Int (Primary Key, Auto-increment)
  assetId: Int (Foreign Key -> Assets.id)
  baseId: Int (Foreign Key -> Bases.id)
  quantity: Int
  date: DateTime (Default: now())
  createdById: Int (Foreign Key -> Users.id)
  createdAt: DateTime
  updatedAt: DateTime
}
```

Transfers Table

```
Transfers {
  id: Int (Primary Key, Auto-increment)
  assetId: Int (Foreign Key -> Assets.id)
  fromBaseId: Int (Foreign Key -> Bases.id)
  toBaseId: Int (Foreign Key -> Bases.id)
  quantity: Int
  date: DateTime (Default: now())
  createdById: Int (Foreign Key -> Users.id)
  createdAt: DateTime
  updatedAt: DateTime
}
```

Assignments Table

```
Assignments {
  id: Int (Primary Key, Auto-increment)
  assetId: Int (Foreign Key -> Assets.id)
  baseId: Int (Foreign Key -> Bases.id)
  personnelId: Int (Foreign Key -> Users.id)
```

```
  assignedById: Int (Foreign Key -> Users.id)
  quantity: Int
  date: DateTime (Default: now())
  createdAt: DateTime
  updatedAt: DateTime
}
```

Expenditures Table

```
Expenditures {
  id: Int (Primary Key, Auto-increment)
  assetId: Int (Foreign Key -> Assets.id)
  baseId: Int (Foreign Key -> Bases.id)
  personnelId: Int (Foreign Key -> Users.id)
  expendedById: Int (Foreign Key -> Users.id)
  quantity: Int
  date: DateTime (Default: now())
  createdAt: DateTime
  updatedAt: DateTime
}
```

3.4 Audit Table

ApiLogs Table

```
ApiLogs {
  id: Int (Primary Key, Auto-increment)
  userId: Int (Foreign Key -> Users.id)
  action: String
  entity: String
  entityId: Int (Nullable)
  timestamp: DateTime (Default: now())
  details: String (Nullable)
  createdAt: DateTime
  updatedAt: DateTime
}
```

---

4. RBAC Explanation

4.1 Role Definitions

Admin Role

- **Access Level**: Full system access
- **Permissions**:
  - View all bases and assets

  - Create, update, delete any record
  - Access all dashboard metrics
  - Manage users and roles
  - Complete audit log access

Base Commander Role

- **Access Level**: Single base access
- **Permissions**:
  - View only assigned base data
  - Manage assets within their base
  - Approve transfers and assignments
  - View base-specific dashboard metrics
  - Limited audit log access (base-only)

Logistics Officer Role

- **Access Level**: Multi-base operational access
- **Permissions**:
  - Create and manage purchases
  - Initiate transfers between bases
  - View operational dashboard metrics
  - Access transfer and purchase logs
  - Read-only access to assignments

4.2 Enforcement Methods

Database Level Filtering The system implements role-based filtering at the database query level:

```
// Base Commander middleware
const effectiveBaseId = userRole === 'Base Commander'
  ? userBaseId
  : (baseId ? parseInt(baseId) : undefined);
```

API Middleware

```
// RBAC Middleware Implementation
export function authorize(roles: string[]) {
  return (req: AuthRequest, res: Response, next: NextFunction) => {
    if (!req.user || !roles.includes(req.user.role)) {
      return res.status(403).json({ message: 'Forbidden: insufficient role' });
    }
    next();
  };
}
```

Frontend Route Protection

```
// Component-level protection
<ProtectedRoute allowedRoles={['Admin', 'Logistics Officer']}>
  <PurchasesPage />
</ProtectedRoute>
```

4.3 Security Implementation

JWT Token Structure

```
{
  "userId": 1,
  "email": "admin@military.local",
  "role": "Admin",
  "baseId": 1,
  "iat": 1672531200,
  "exp": 1672617600
}
```

Password Security

- **Hashing**: bcryptjs with salt rounds (10)
- **Storage**: Never store plain text passwords
- **Validation**: Strong password requirements on frontend

---

5. API Logging

5.1 Logging Strategy

All critical operations are logged to the `ApiLogs` table for complete audit trail compliance.

5.2 Logged Operations

- **Asset Creation/Updates**: New assets and modifications
- **Purchases**: All procurement transactions
- **Transfers**: Inter-base asset movements
- **Assignments**: Personnel asset allocations
- **Expenditures**: Asset consumption records

5.3 Log Implementation

```
// Example: Purchase Logging
await prisma.apiLog.create({
  data: {
    userId: req.user.id,
```

```
    action: 'CREATE',
    entity: 'Purchase',
    entityId: purchase.id,
    details: JSON.stringify({
      assetId: purchase.assetId,
      baseId: purchase.baseId,
      quantity: purchase.quantity
    })
  }
});
```

5.4 Log Data Structure

```
{
  "id": 1,
  "userId": 1,
  "action": "CREATE",
  "entity": "Transfer",
  "entityId": 15,
  "timestamp": "2025-06-19T08:30:00Z",
  "details": "{\"fromBaseId\":1,\"toBaseId\":2,\"assetId\":5,\"quantity\":3}"
}
```

---

6. Setup Instructions

6.1 Prerequisites

- **Node.js**: Version 18.0.0 or higher
- **PostgreSQL**: Version 13.0 or higher
- **npm**: Version 8.0.0 or higher
- **Git**: For version control

6.2 Database Setup

```
# Install PostgreSQL (macOS)
brew install postgresql
brew services start postgresql

# Create database
createdb -U postgres military_asset_db

# Import database dump
psql -U postgres -d military_asset_db < database_dump.sql
```

6.3 Backend Setup

```
# Navigate to backend directory
cd backend

# Install dependencies
npm install

# Configure environment variables
cp .env.example .env
# Edit .env file:
# DATABASE_URL="postgresql://postgres:password@localhost:5432/military_asset_db"
# JWT_SECRET="your-secure-secret-key"

# Generate Prisma client
npx prisma generate

# Run database migrations (if needed)
npx prisma db push

# Seed database with sample data
npm run seed

# Start development server
npm run dev
```

6.4 Frontend Setup

```
# Navigate to frontend directory
cd frontend

# Install dependencies
npm install

# Start development server
npm run dev
```

6.5 Verification

Backend: http://localhost:4000/health should return {“status”:“ok”} **Frontend**: http://localhost:3000 should show login page **Login**: admin@military.local / admin123

---

7. API Endpoints

7.1 Authentication Endpoints

POST /api/auth/login

```
POST /api/auth/login
Content-Type: application/json

{
  "email": "admin@military.local",
  "password": "admin123"
}

Response:
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "id": 1,
    "email": "admin@military.local",
    "name": "Admin User",
    "role": "Admin",
    "baseId": 1
  }
}
```

7.2 Dashboard Endpoints

GET /api/dashboard/metrics

```
GET /api/dashboard/metrics?startDate=2025-06-01&endDate=2025-06-30&baseId=1
Authorization: Bearer <token>

Response:
{
  "openingBalance": 45,
  "closingBalance": 52,
  "netMovement": {
    "total": 7,
    "purchases": 10,
    "transfersIn": 3,
    "transfersOut": 6
  },
  "assigned": 8,
  "expended": 2
}
```

7.3 Asset Management Endpoints

GET /api/assets

```
GET /api/assets
Authorization: Bearer <token>

Response:
[
  {
    "id": 1,
    "type": "Vehicle",
    "serial": "ASSET-1",
    "description": "Test Asset 1",
    "baseId": 1,
    "base": {
      "id": 1,
      "name": "HQ",
      "location": "Central Command"
    }
  }
]
```

POST /api/assets

```
POST /api/assets
Authorization: Bearer <token>
Content-Type: application/json

{
  "type": "Weapon",
  "serial": "WPN-001",
  "description": "M4 Rifle",
  "baseId": 1
}

Response:
{
  "id": 11,
  "type": "Weapon",
  "serial": "WPN-001",
  "description": "M4 Rifle",
  "baseId": 1
}
```

7.4 Transaction Endpoints

POST /api/purchases

```
POST /api/purchases
Authorization: Bearer <token>
Content-Type: application/json

{
  "assetId": 1,
  "baseId": 1,
  "quantity": 5
}

Response:
{
  "id": 6,
  "assetId": 1,
  "baseId": 1,
  "quantity": 5,
  "date": "2025-06-19T08:30:00Z",
  "createdById": 1
}
```

POST /api/transfers

```
POST /api/transfers
Authorization: Bearer <token>
Content-Type: application/json

{
  "assetId": 1,
  "fromBaseId": 1,
  "toBaseId": 2,
  "quantity": 3
}

Response:
{
  "id": 2,
  "assetId": 1,
  "fromBaseId": 1,
  "toBaseId": 2,
  "quantity": 3,
  "date": "2025-06-19T08:30:00Z",
  "createdById": 1
}
```

7.5 Error Responses

```
// Authentication Error
401 Unauthorized
{
  "message": "Invalid or expired token"
}

// Authorization Error
403 Forbidden
{
  "message": "Forbidden: insufficient role"
}

// Validation Error
400 Bad Request
{
  "message": "Validation failed",
  "errors": [
    {
      "field": "quantity",
      "message": "Must be a positive integer"
    }
  ]
}
```

Conclusion

The Military Asset Management System provides a comprehensive solution for managing military assets with enterprise-grade security, complete audit trails, and role-based access control. The system is built using modern technologies and follows best practices for scalability, maintainability, and security.

For support or questions, refer to the comprehensive documentation provided with the system.

Document Version: 1.0
Last Updated: June 19, 2025
Contact: Development Team