🚀 KGL_L APPLICATION

Complete Code Documentation

Agricultural Supply Chain Management System
Version: 1.0.0
Date: February 2026

📋 Project Overview

The KGL_L application is a comprehensive web-based platform for managing agricultural supply chain operations. It includes user authentication, role-based access control, and dashboard management for different user roles including Admins, Managers, Procurement Officers, and Sales Agents.

📑 Table of Contents

1. System Architecture
2. Project Dependencies
3. Backend - Server Configuration (server.js)
4. Authentication Routes (routes/auth.js)
5. User Database Model (models/User.js)
6. Frontend - HTML Pages
7. API Endpoints Reference
8. Authentication Flow
9. User Roles & Permissions

10. Setup & Installation

1️⃣ System Architecture

Architecture Overview

The KGL_L application follows a **Client-Server Architecture**:

✓ **Frontend (Client):** HTML5, CSS3, JavaScript (Vanilla JS)

✓ **Backend (Server):** Node.js with Express.js

✓ **Database:** MongoDB (NoSQL)

✓ **Authentication:** JWT (JSON Web Tokens)

✓ **File Upload:** Multer for handling image uploads

System Flow Diagram

```
Client (Browser)
    ↓
Frontend Pages (Login/Register/Dashboards)
    ↓
HTTP Requests (Fetch API)
    ↓
Express Server (Port 3000)
    ↓
Authentication Routes
    ↓
MongoDB Database
```

2️⃣ Project Dependencies

NPM Packages Used

Package	Version	Purpose
express	^5.2.1	Web framework for creating API endpoints and serving pages
mongoose	^9.2.0	MongoDB object modeling for database operations
bcryptjs	^3.0.3	Password hashing for secure user authentication
jsonwebtoken	^9.0.3	Generate and verify JWT tokens for user sessions
multer	^2.0.2	Handle file uploads (profile photos)
cors	^2.8.6	Enable Cross-Origin Resource Sharing
dotenv	^17.2.4	Load environment variables from .env file
body-parser	^2.2.2	Parse incoming request bodies

3️⃣ Backend - Server Configuration (server.js)

📄 File: server.js

Purpose

The server.js file initializes the Express application, sets up middleware, connects to MongoDB, defines database schemas, and creates API endpoints for managing procurements, sales, and reports.

Key Components

1. Module Imports

```
const express = require('express'); const mongoose =
require('mongoose'); require('dotenv').config(); const path =
require('path');
```

What it does:

- `express` - Web framework for handling HTTP requests
- `mongoose` - Library for connecting to and querying MongoDB
- `dotenv` - Loads environment variables from .env file (like database URL)
- `path` - Built-in Node.js module for handling file paths

2. CORS Configuration

```
let cors; try { cors = require('cors'); } catch (e) { console.warn('cors
not found; using fallback'); cors = null; } if (cors) { app.use(cors());
} else { app.use((req, res, next) => { res.setHeader('Access-Control-
Allow-Origin', '*'); res.setHeader('Access-Control-Allow-Methods',
'GET,POST,PUT,DELETE'); res.setHeader('Access-Control-Allow-Headers',
'Content-Type, Authorization'); if (req.method === 'OPTIONS') return
res.sendStatus(200); next(); }); }
```

What it does:

- Attempts to load the CORS package
- If available, enables CORS for all routes
- If not available, creates a fallback CORS middleware manually
- Allows requests from any origin (frontend on different port)
- Sets allowed HTTP methods and headers

3. Middleware Setup

```
app.use(express.json({ limit: '10mb' })); app.use(express.urlencoded({
limit: '10mb', extended: true }));
app.use(express.static(path.join(__dirname, 'public')));
```

What it does:

- `express.json()` - Parses incoming JSON request bodies (max 10MB)
- `express.urlencoded()` - Parses form data
- `express.static()` - Serves static files from public directory (CSS, JS, images)

4. Debug Logging Middleware

```
app.use((req, res, next) => { console.log(`${new Date().toISOString()} -
${req.method} ${req.path}`); if (req.path.endsWith('.html') || req.path
=== '/login' || req.path === '/register') { res.set('Cache-Control',
'no-store, no-cache, must-revalidate'); } next(); });
```

What it does:

- Logs every incoming request with timestamp, HTTP method, and path
- Disables caching for HTML pages to ensure fresh content
- Prevents browser from caching login/register pages

5. MongoDB Connection

```
const MONGODB_URI = process.env.MONGODB_URI ||
'mongodb://localhost:27017/kgl'; mongoose.connect(MONGODB_URI) .then(()
=> console.log('Connected to MongoDB')) .catch((err) =>
console.error('MongoDB connection error:', err));
```

What it does:

- Reads MongoDB connection URL from environment variables
- Falls back to local MongoDB on port 27017 if not specified
- Connects to database named 'kgl'
- Logs success or error message

6. Route Definition

```
const authRoutes = require('./routes/auth'); app.use('/api/auth',
authRoutes); app.get('/', (req, res) => res.redirect('/login'));
app.get('/login', (req, res) => res.sendFile(path.join(__dirname,
'login', 'login.html'))); app.get('/register', (req, res) =>
res.sendFile(path.join(__dirname, 'login', 'register.html')));
```

What it does:

- Imports authentication routes from separate file
- Mounts auth routes under /api/auth prefix
- Root path (/) redirects to /login
- Serves login and register HTML pages

7. Database Schemas

```
const produceSchema = new mongoose.Schema({ name: String, type: String,
tonnage: Number, cost: Number, dealerName: String, branch: String,
contact: String, salePrice: Number, createdAt: { type: Date, default:
Date.now } }); const Produce = mongoose.model('Produce', produceSchema);
```

Produce Schema - Represents agricultural products:

- `name` - Product name (e.g., "Maize", "Wheat")
- `type` - Product category
- `tonnage` - Quantity in metric tons
- `cost` - Purchase cost per unit
- `dealerName` - Name of supplier/dealer
- `branch` - Branch location
- `contact` - Dealer contact information
- `salePrice` - Selling price per unit

- `createdAt` - Timestamp of entry creation

8. API Endpoints - Procurement

```
app.post('/api/procurement', (req, res) => { const produce = new
Produce(req.body); produce.save((err, produce) => { if (err)
res.status(400).json({ error: err.message }); else
res.status(201).json(produce); }); }); app.get('/api/procurement', (req,
res) => { Produce.find().sort({ createdAt: -1 }).exec((err, produces) =>
{ if (err) res.status(400).json({ error: err.message }); else
res.json(produces); }); });
```

What it does:

- **POST /api/procurement** - Creates a new procurement record
- **GET /api/procurement** - Retrieves all procurements sorted by newest first
- Uses Mongoose to save/query data from MongoDB
- Returns appropriate HTTP status codes (201 for creation, 400 for errors)

9. Server Startup

```
const PORT = process.env.PORT || 3000; app.listen(PORT, () => {
console.log(`Server running on http://localhost:${PORT}`); });
```

What it does:

- Reads port from environment or uses default 3000
- Starts HTTP server and listens for incoming requests
- Logs server startup message

4️⃣ Authentication Routes (routes/auth.js)

📄 File: routes/auth.js

Purpose

This file handles all user authentication operations: registration, login, file uploads, password hashing, and JWT token generation.

1. Module Setup

```
const express = require('express'); const router = express.Router();
const bcrypt = require('bcryptjs'); const jwt = require('jsonwebtoken');
const User = require('../models/User'); const JWT_SECRET =
process.env.JWT_SECRET || 'your-secret-key-change-this';
```

What it does:

- `bcryptjs` - Securely hashes passwords before storing
- `jsonwebtoken` - Creates JWT tokens for session management
- `JWT_SECRET` - Secret key for signing tokens (should be in .env file)

2. Multer File Upload Configuration

```
let multer; let uploadMiddleware = null; try { multer =
require('multer'); const storage = multer.diskStorage({ destination:
function (req, file, cb) { cb(null, 'public/uploads'); }, filename:
function (req, file, cb) { const uniqueSuffix = Date.now() + '-' +
Math.round(Math.random() * 1E9); const sanitized =
file.originalname.replace(/[^a-zA-Z0-9.\-\_]/g, '_'); cb(null,
uniqueSuffix + '-' + sanitized); } }); const upload = multer({ storage:
storage, limits: { fileSize: 5 * 1024 * 1024 } }); uploadMiddleware =
upload.single('photo'); } catch (e) { console.warn('Multer not
installed; file uploads disabled'); }
```

What it does:

- Configures Multer for handling file uploads
- Sets upload destination to 'public/uploads' folder
- Creates unique filenames using timestamp + random number
- Sanitizes original filenames (removes special characters)
- Limits file size to 5MB
- Handles gracefully if Multer is not installed

3. Register Endpoint (With File Upload)

```
router.post('/register', uploadMiddleware, async (req, res) => { try {
const { name, email, password, confirmPassword, role } = req.body; //
Validation if (!name || !email || !password || !confirmPassword ||
!role) { return res.status(400).json({ error: 'All fields are required'
}); } if (password !== confirmPassword) { return res.status(400).json({
error: 'Passwords do not match' }); } // Check if user exists const
userExists = await User.findOne({ email }); if (userExists) { return
res.status(400).json({ error: 'Email already registered' }); } // Hash
password const hashedPassword = await bcrypt.hash(password, 10); //
Create user object const userData = { name, email, password:
hashedPassword, role }; // Add photo if uploaded if (req.file) {
userData.photo = '/uploads/' + req.file.filename; } // Save to database
const user = new User(userData); await user.save(); // Generate JWT
token const token = jwt.sign( { userId: user._id, email: user.email,
role: user.role }, JWT_SECRET, { expiresIn: '7d' } ); // Return success
response res.status(201).json({ message: 'User registered successfully',
token, role: user.role, userId: user._id, name: user.name, photo:
user.photo }); } catch (error) { console.error('Register error:',
error); res.status(500).json({ error: error.message }); } });
```

Registration Process Step by Step:

1. **Extract Data:** Gets name, email, password, role from request
2. **Validate:** Checks all required fields are provided
3. **Verify Passwords:** Ensures password and confirmPassword match
4. **Check Uniqueness:** Queries database to ensure email isn't already registered
5. **Hash Password:** Uses bcrypt to securely hash password (10 salt rounds)
6. **Handle File:** If user uploaded photo, stores path in userData
7. **Save User:** Creates and saves new User document to MongoDB
8. **Generate Token:** Creates JWT token valid for 7 days
9. **Return Response:** Sends token, user info, and photo URL to frontend

4. Login Endpoint

```
router.post('/login', async (req, res) => { try { const { email,
password } = req.body; // Validate if (!email || !password) { return
res.status(400).json({ error: 'Email and password required' }); } //
Find user const user = await User.findOne({ email }); if (!user) {
return res.status(401).json({ error: 'Invalid email or password' }); }
// Verify password const passwordMatch = await bcrypt.compare(password,
user.password); if (!passwordMatch) { return res.status(401).json({
error: 'Invalid email or password' }); } // Generate token const token =
jwt.sign( { userId: user._id, email: user.email, role: user.role },
JWT_SECRET, { expiresIn: '7d' } ); // Return response res.json({
message: 'Login successful', token, role: user.role, userId: user._id,
name: user.name, photo: user.photo }); } catch (error) {
console.error('Login error:', error); res.status(500).json({ error:
error.message }); } });
```

Login Process Step by Step:

1. **Extract Credentials:** Gets email and password from request
2. **Find User:** Queries database for user with this email
3. **Verify Password:** Uses bcrypt.compare() to check hashed password
4. **Generate Token:** Creates new JWT token
5. **Return Data:** Sends token and user info (name, role, photo)

5. User Profile Endpoint

```
router.get('/profile/:userId', async (req, res) => { try { const {
userId } = req.params; const user = await User.findById(userId); if
(!user) { return res.status(404).json({ error: 'User not found' }); }
res.json({ userId: user._id, name: user.name, email: user.email, role:
user.role, photo: user.photo }); } catch (error) {
console.error('Profile error:', error); res.status(500).json({ error:
error.message }); } });
```

What it does:

- Retrieves user profile information by user ID
- Returns user details: name, email, role, photo URL
- Used by dashboards to fetch user data

5️⃣ User Database Model (models/User.js)

📄 File: models/User.js

Purpose

Defines the MongoDB schema for User documents and enforces data validation rules.

```
const mongoose = require('mongoose'); const userSchema = new
mongoose.Schema({ name: { type: String, required: true }, email: { type:
String, required: true, unique: true }, password: { type: String,
required: true }, role: { type: String, enum: ['admin', 'manager',
'procurement', 'agent'], required: true }, photo: { type: String,
default: null } }, { timestamps: true }); module.exports =
mongoose.model('User', userSchema);
```

Schema Fields Explained:

- **name** - User's full name (required)
- **email** - User's email address (required, unique - no duplicates)
- **password** - Hashed password (required, never stored as plaintext)
- **role** - User's role (must be one of 4 options: admin, manager, procurement, agent)
- **photo** - URL to profile photo (optional, defaults to null)
- **timestamps: true** - Automatically adds createdAt and updatedAt fields

Security Note: The schema enforces that each email is unique, preventing duplicate accounts. Passwords are hashed by bcryptjs before storage.

6️⃣ Frontend - HTML Dashboard Pages

Dashboard Overview

Each user role has a dedicated dashboard with role-specific features and styling.

👤 **Admin Dashboard**

👨‍💼 **Manager Dashboard**

📦 **Procurement Dashboard**

🚀 **Sales Agent Dashboard**

Common Dashboard Features

- ✓ Responsive layout with sidebar navigation
- ✓ User profile display with photo
- ✓ Role-based menu items
- ✓ Statistics/metrics cards
- ✓ Data tables and forms
- ✓ Logout functionality

Frontend Authentication Logic

```
document.addEventListener('DOMContentLoaded', () => { try { const token = localStorage.getItem('token'); if (!token) { window.location.href = 'http://localhost:3000/login'; return; } const userName = localStorage.getItem('userName') || 'User'; const userPhoto = localStorage.getItem('userPhoto'); document.getElementById('userName').textContent = userName; if (userPhoto) { const profileImg = document.getElementById('profileImg'); if (profileImg) { profileImg.src = userPhoto; profileImg.style.display = 'block'; } } } catch (error) { console.error('Dashboard init error:', error); } });
```

What it does:

- Waits for page to fully load
- Checks if user is logged in (token in localStorage)
- Redirects to login if not authenticated
- Loads user name and displays it
- Loads profile photo if available
- Displays photo in circular image element

Login Page - Form Submission

```
form.addEventListener('submit', async (e) => { e.preventDefault(); const email = document.getElementById('email').value; const password = document.getElementById('password').value; try { const response = await fetch('http://localhost:3000/api/auth/login', { method: 'POST', headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({ email, password }) }); const data = await response.json(); if (!response.ok) { throw new Error(data.error || 'Login failed'); } // Store auth data localStorage.setItem('token', data.token); localStorage.setItem('role', data.role); localStorage.setItem('userId', data.userId); localStorage.setItem('userName', data.name); if (data.photo) { localStorage.setItem('userPhoto', data.photo); } // Redirect based on role const roleRoutes = { 'manager': 'http://localhost:3000/manager-dashboard', 'admin': 'http://localhost:3000/admin-dashboard', 'procurement': 'http://localhost:3000/procurement-dashboard', 'agent': 'http://localhost:3000/agent-dashboard' }; window.location.href = roleRoutes[data.role] || 'http://localhost:3000/manager-dashboard'; } catch (error) { errorMessage.textContent = error.message; } });
```

Login Flow:

1. User clicks login button
2. Form data is sent to backend via fetch API
3. Backend verifies credentials
4. Backend returns token and user data
5. Frontend stores all data in localStorage
6. Frontend redirects to appropriate dashboard based on role

Logout Functionality

```
function logout() { localStorage.removeItem('token'); localStorage.removeItem('role'); localStorage.removeItem('userId'); localStorage.removeItem('userName');
localStorage.removeItem('userPhoto'); window.location.href =
'http://localhost:3000/login'; }
```

What it does:

- Clears all session data from localStorage
- Removes token, preventing further authenticated requests
- Redirects to login page

7️⃣ Complete API Endpoints Reference

Authentication Endpoints

POST `/api/auth/register`

Purpose: Register a new user account

Request Body:

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "securePassword123",
  "confirmPassword": "securePassword123",
  "role": "admin",
  "photo": "[File - optional]"
}
```

Response (Success):

```
{
  "message": "User registered successfully",
  "token": "eyJhbGciOiJIUzI1NiIs...",
  "role": "admin",
  "userId": "65a4b3c2d1e8f9g0h1i2j3k4",
  "name": "John Doe",
  "photo": "/uploads/1234-profile.jpg"
}
```

Status Codes:

- 201 - User created successfully
- 400 - Missing fields or validation error
- 500 - Server error

POST `/api/auth/login`

Purpose: Authenticate user and get session token

Request Body:

```
{
  "email": "john@example.com",
  "password": "securePassword123"
}
```

Response (Success):

```
{
  "message": "Login successful",
  "token": "eyJhbGciOiJIUzI1NiIs...",
  "role": "admin",
  "userId": "65a4b3c2d1e8f9g0h1i2j3k4",
  "name": "John Doe",
  "photo": "/uploads/1234-profile.jpg"
}
```

Status Codes:

- 200 - Login successful
- 401 - Invalid credentials
- 400 - Missing email or password

GET `/api/auth/profile/:userId`

Purpose: Retrieve user profile information

URL Parameter:

`userId - MongoDB user ID`

Response (Success):

```
{
  "userId": "65a4b3c2d1e8f9g0h1i2j3k4",
  "name": "John Doe",
  "email": "john@example.com",
  "role": "admin",
  "photo": "/uploads/1234-profile.jpg"
}
```

Status Codes:

- 200 - Profile retrieved
- 404 - User not found

Procurement Endpoints

POST `/api/procurement`

Purpose: Create new procurement record

Request Body:

```
{
  "name": "Maize",
  "type": "Grain",
  "tonnage": 500,
  "cost": 150000,
  "dealerName": "John Supplies",
  "branch": "Main",
  "contact": "0712345678",
  "salePrice": 200000
}
```

Status Codes: 201 Created, 400 Bad Request

GET `/api/procurement`

Purpose: Retrieve all procurement records (newest first)

Response: Array of procurement documents

Status Codes: 200 OK, 400 Error

Sales Endpoints

POST `/api/sales`

Purpose: Record a new sale

Request Body:

```
{
  "produceName": "Maize",
  "tonnage": 100,
  "amountPaid": 20000000,
  "buyerName": "ABC Trading",
  "salesAgentName": "John Doe"
}
```

Status Codes: 201 Created, 400 Bad Request

GET `/api/sales`

Purpose: Retrieve all sales records

Response: Array of sale documents

Status Codes: 200 OK, 400 Error

Reports Endpoints

POST `/api/reports`

Purpose: Create a new report

Request Body:

```
{
  "reportType": "Monthly Sales",
  "branch": "Main"
}
```

Status Codes: 201 Created, 400 Bad Request

GET `/api/reports`

Purpose: Retrieve all reports

Response: Array of report documents

Status Codes: 200 OK, 400 Error

8️⃣ User Roles & Permissions

Role Distribution

Role	Dashboard URL	Primary Functions
Admin 👤	/admin-dashboard	System management, user management, activity logs
Manager 👔	/manager-dashboard	Procurement oversight, stock management, sales tracking, reports
Procurement Officer 📦	/procurement-dashboard	Purchase orders, supplier management, inventory tracking
Sales Agent 🚀	/agent-dashboard	Record sales, view sales history, track targets

Role-Based Features

👤 Admin Role

- ✓ View total users count
- ✓ Monitor active sessions
- ✓ Check system status
- ✓ Access system settings
- ✓ View activity logs
- ✓ Manage user accounts

🧑‍💼 *Manager Role*

> ✓ View all procurement items
>
> ✓ Record new procurement
>
> ✓ Track stock levels
>
> ✓ Record sales transactions
>
> ✓ View sales analytics
>
> ✓ Generate reports

📦 *Procurement Officer Role*

> ✓ Create purchase orders
>
> ✓ Manage supplier information
>
> ✓ Track delivery status
>
> ✓ View procurement history

🚀 *Sales Agent Role*

> ✓ Record new sales
>
> ✓ View sales history
>
> ✓ Track monthly targets
>
> ✓ View customer information

9️⃣ Setup & Installation Guide

Prerequisites

✓ Node.js (v14 or higher)

✓ MongoDB (local or cloud - MongoDB Atlas)

✓ npm (comes with Node.js)

✓ A code editor (VSCode recommended)

Step-by-Step Installation

1. Clone/Download Project

```
cd ~/Desktop/KGL_L
```

2. Install Dependencies

```
npm install
```

This command will install all packages listed in package.json:

- express - Web framework
- mongoose - MongoDB driver
- bcryptjs - Password hashing
- jsonwebtoken - JWT creation
- multer - File uploads
- cors - Cross-origin requests
- dotenv - Environment variables

3. Create .env File

```
Create file: .env Add content: MONGODB_URI=mongodb://localhost:27017/kgl
JWT_SECRET=your-secret-key-here PORT=3000
```

4. Start MongoDB

```
Windows: mongod Mac/Linux: brew services start mongodb-community
```

5. Create Upload Directory

```
Create folder: public/uploads
```

6. Start Application

```
npm start or node server.js
```

7. Access Application

```
Open browser: http://localhost:3000 Login page:
http://localhost:3000/login Register page:
http://localhost:3000/register
```

Troubleshooting

Issue: "Cannot find module 'express'"
Solution: Run `npm install` to install dependencies

Issue: "MongoDB connection error"
Solution: Make sure MongoDB is running. For Windows, check Services panel. For Mac/Linux, run `brew services start mongodb-community`

Issue: "Port 3000 already in use"
Solution: Change PORT in .env file or kill process using port 3000

Issue: "CORS error" when accessing from different port
Solution: Ensure redirect URLs use full address: http://localhost:3000/dashboard

🔟 Complete Data Flow Diagrams

User Registration Flow

```
FORM SUBMISSION ↓ Client validates (passwords match, password length) ↓
Send POST request to /api/auth/register ↓ Server receives FormData (with
file if photo uploaded) ↓ Validate all required fields ↓ Hash password
using bcryptjs ↓ Create User document in MongoDB ↓ Upload photo to
public/uploads folder ↓ Generate JWT token ↓ Store in localStorage:
token, role, userId, userName, userPhoto ↓ Redirect to appropriate
dashboard ↓ Dashboard loads user data from localStorage
```

User Login Flow

```
LOGIN FORM SUBMISSION ↓ Send POST request to /api/auth/login ↓ Server
finds user by email in MongoDB ↓ Compare submitted password with stored
hashed password ↓ If match, generate JWT token ↓ Return token, role,
userId, userName, photo URL ↓ Client stores all data in localStorage ↓
Redirect based on role: - admin → /admin-dashboard - manager → /manager-
dashboard - procurement → /procurement-dashboard - agent → /agent-
dashboard ↓ Dashboard DOMContentLoaded event: - Checks for token in
localStorage - If no token, redirect to login - If token exists, load
user data - Display username and photo
```

Dashboard Access Control

```
USER VISITS DASHBOARD ↓ DOMContentLoaded event fires ↓ Check
localStorage for token ↓ Token exists? ├─ NO → Redirect to login └─ YES
→ Continue ↓ Load userName from localStorage ↓ Load userPhoto from
localStorage ↓ Display username in navbar ↓ Display profile photo in
circular container ↓ Initialize dashboard features
```

Logout Flow

```
USER CLICKS LOGOUT BUTTON ↓ Clear localStorage: - token - role - userId
- userName - userPhoto ↓ Redirect to login page ↓ All session data is
```

```
erased ↓ User must login again to access dashboard
```

🔐 Security Measures

Password Security:

- Passwords are hashed using bcryptjs with 10 salt rounds
- Never stored as plaintext in database
- Hashed password compared during login

Authentication:

- JWT tokens issued after successful login
- Tokens valid for 7 days
- Token stored in browser localStorage
- Token required to access protected dashboards

Data Validation:

- Email uniqueness enforced at database level
- Required fields validated before database operations
- Password confirmation validation on client side

⚠️ Security Warnings for Production:

- Use HTTPS instead of HTTP
- Store JWT_SECRET in secure environment variable
- Don't expose database credentials in code
- Implement rate limiting on auth endpoints
- Use secure session storage instead of localStorage
- Implement CSRF protection
- Add input sanitization
- Enable HTTPS-only cookies

📝 Summary & Key Takeaways

Architecture Overview

The KGL_L application implements a modern three-tier architecture:

1. **Presentation Layer:** HTML, CSS, JavaScript dashboards
2. **Business Logic Layer:** Express.js server with authentication
3. **Data Layer:** MongoDB database with user and transaction data

Key Features

- ✓ User registration with profile photo upload
- ✓ Secure login with JWT tokens
- ✓ Role-based access control (4 roles)
- ✓ MongoDB database for persistence
- ✓ RESTful API endpoints
- ✓ Responsive dashboards
- ✓ File upload management

Authentication Flow (Summary)

Users register with name, email, password, and role. Passwords are securely hashed. After successful login, a JWT token is issued and stored in localStorage. Dashboards check for this token on load—if missing, users are redirected to login. Logout clears all session data from localStorage.

Database Structure

MongoDB stores User documents with fields for authentication (email, hashed password), profile information (name, role, photo URL), and Procurement/Sales/Reports collections for business data.

KGL_L Application Documentation v1.0
Created: February 2026 | Last Updated: February 14, 2026
This documentation covers all aspects of the KGL_L codebase including backend, frontend, and API structure.

For questions or updates, please review the code comments and server console logs during development.