Web Based ERP Management System

Project Overview

This Web Based ERP Management System is a complete business solution designed to manage daily operations easily and efficiently. It helps business owners track sales, stock, customers, expenses, and reports without any technical knowledge. The system is user-friendly, fast, secure, and suitable for long-term business growth.

Core Features (Easy to Understand)

Admin Dashboards

- Separate dashboards for Admin and staff
- Simple charts showing sales, profit, and stock status
- Clear daily, monthly, and yearly business overview

Stock Management

- Easy stock entry and update
- Automatic stock increase and decrease
- Low stock alerts

Stock Tracking

- Complete record of purchases, sales, and returns
- Clear stock movement history

Customer Management

- Customer details in one place
- Customer balance and credit record

Customer Reports

- Customer purchase history
- Party and tour reports
- Outstanding balance reports

Sales Management

- Cash sales
- Credit sales
- Delivery challan
- Sale return and exchange

Expenses Management

- Daily expense entry
- Expense category reports

Reports

- Sales reports (daily, monthly, yearly)
- Profit and loss reports
- Custom date-wise reports

User Management

- Separate user accounts
- Role-based access (Admin, Manager, Staff)

System Logs & Security

- User activity record
- Secure login system
- Data safety and backup

Language Support

- English & Urdu billing and reports

Theme Options

- Light and dark themes

Hosting & Infrastructure Cost

- VPS Hosting (NVMe, 4 Fast Servers): 6,000 PKR / Month
- Domain (Lifetime): 3,000 – 4,000 PKR (One-Time)
- Cloud Backup (100 GB): 2,000 PKR / Month

Development & Deployment Cost

- Total Development & Deployment: 135,000 PKR (One-Time)

Payment Models

Option 1: One-Time Payment

- Full ERP System Development
- Deployment on VPS
- Initial setup & configuration
- Cost: 145,000 PKR

Option 2: Monthly Maintenance Plan

- System maintenance
- Bug fixes
- Minor feature enhancements
- Server monitoring
- Backup monitoring

(Maintenance charges can be finalized separately)

Option 3: Yearly One-Time Payment Plan

- Complete ERP system usage for 1 full year
- Hosting on fast VPS server included
- Cloud backup included
- System monitoring and support included
- Bug fixing and smooth system operation
- No monthly tension

Yearly One-Time Payment: 180,000 PKR / Year

Yearly Cost Plan (Estimated)

Fixed Yearly Costs

- VPS Hosting: 6,000 × 12 = 72,000 PKR / Year
- Cloud Backup: 2,000 × 12 = 24,000 PKR / Year
- Domain: 3,000 – 4,000 PKR (One-Time)

Total Yearly Infrastructure Cost

- Approximately: 96,000 – 100,000 PKR / Year

Read this. This is a project of a production level, but I would need a project proposal for my university web application project. I think I would build this project.
What is required in our uni project:
backend(Express) and database(MonogdB)

First, we would need to create a proposal for our uni web application project. Can you write a simple humanized language proposal, on the first page, which will be about description, details, and then all table names that will be required in this project, and all API's
It's simple

Web Based ERP Management System

Project Proposal (Web Application)

Project Description

This project is a **Web Based ERP Management System** developed as a university-level web application using **Express.js (Backend)** and **MongoDB (Database)**. The system is designed to help

small and medium businesses manage their daily operations such as sales, stock, customers, expenses, and reports in a simple and organized way.

The main goal of this project is to demonstrate practical knowledge of backend development, RESTful APIs, database design, authentication, and role-based access control. The system focuses on usability, security, and real-world business workflows while remaining simple enough for academic evaluation.

This ERP system provides separate access for Admin and Staff users, allowing controlled operations based on roles. It maintains complete records of stock movement, sales transactions, customer balances, and expenses, and generates useful reports for decision-making.

Project Objectives

- To design and develop a real-world ERP system using Express.js and MongoDB
- To implement secure authentication and role-based authorization
- To manage business data such as stock, sales, customers, and expenses
- To generate meaningful reports from stored data
- To follow proper backend architecture and API-based communication

---

Technology Stack

- **Backend:** Node.js with Express.js
- **Database:** MongoDB (Mongoose ODM)
- **Authentication:** JWT-based authentication
- **API Type:** RESTful APIs
- **Tools:** Postman, VS Code

Database Design (MongoDB Collections)

Below are the main collections (tables) required for this project:

1. users

Stores system users and their roles.

- _id
- name
- email
- password
- role (Admin, Manager, Staff)
- status
- createdAt

2. products

Stores stock items.

- _id
- productName
- category

- purchasePrice
- salePrice
- quantity
- lowStockAlert
- createdAt

3. stock_logs

Tracks stock movement history.

- _id
- productId
- type (purchase, sale, return)
- quantity
- referenceId
- date

4. customers

Stores customer information.

- _id
- name
- phone
- address
- balance
- createdAt

5. sales

Stores sales transactions.

- _id
- customerId
- saleType (cash, credit)
- totalAmount
- paidAmount
- remainingAmount
- date

6. sale_items

Stores products included in a sale.

- _id
- saleId
- productId
- quantity
- price

7. expenses

Stores daily business expenses.

- _id

- title
- category
- amount
- date
- note

8. expense_categories

Stores expense categories.

- _id
- categoryName

9. returns

Handles sale returns and exchanges.

- _id
- saleId
- productId
- quantity
- refundAmount
- date

10. system_logs

Tracks user activity for security.

- _id
- userId
- action
- timestamp

API Design (Express.js REST APIs)

Authentication APIs

- POST /api/auth/register
- POST /api/auth/login
- GET /api/auth/profile

User Management APIs

- GET /api/users
- POST /api/users
- PUT /api/users/:id
- DELETE /api/users/:id

Product & Stock APIs

- GET /api/products
- POST /api/products
- PUT /api/products/:id

- DELETE /api/products/:id
- GET /api/stock/logs

Customer APIs

- GET /api/customers
- POST /api/customers
- PUT /api/customers/:id
- GET /api/customers/:id/history

Sales APIs

- POST /api/sales
- GET /api/sales
- GET /api/sales/:id

Sale Return APIs

- POST /api/returns
- GET /api/returns

Expense APIs

- POST /api/expenses
- GET /api/expenses
- GET /api/expenses/report

Reports APIs

- GET /api/reports/sales
- GET /api/reports/profit-loss
- GET /api/reports/stock

System Logs APIs

- GET /api/logs

Conclusion

This Web Based ERP Management System is a production-inspired academic project that demonstrates backend development skills using Express.js and MongoDB. It follows real business workflows, uses structured APIs, and applies proper database design. The project is suitable for a university web application course and can be extended in the future with frontend frameworks and

I've created a **clean, university-appropriate project proposal** in the canvas with:

- Simple, humanized language (not commercial or client-facing)
- Clear **project description & objectives**
- Explicit use of **Express.js (backend)** and **MongoDB (database)** as required
- Proper **MongoDB collection design** (tables)
- Well-structured **REST API list** suitable for a web application course
- A concise, professional **"first-page style" proposal** that examiners expect

Why this fits a university project well

- It looks **real-world**, but not over-engineered

- Covers all core backend concepts: CRUD, auth, roles, reports
- Easy to defend in viva: *"ERP is a real business problem"*
- Scalable if your teacher asks: *"How can this be extended?"*

Next logical steps (when you're ready)

1. Convert this into **SRS sections** (Actors, Use Cases)
2. Draw **ERD / Collection relationship diagram**
3. Design **API flow diagrams**
4. Start with **Auth + Users module** in Express

If you want, I can:

- Simplify it further to **1-page only**
- Rewrite it exactly in **UCP / SE department format**
- Create **ERD + API flow** diagrams
- Help you **implement folder structure** for Express + MongoDB

Just tell me what your teacher usually checks first.

here is my plan and idea what we will do, I would need your help to manage some things, for ideas and for information.

First we design a complete structure and design, flow of how the app will work, so we can decide how many APIs we will require and why we requird.
The backend with the required REST APIs, you will provide me with the list of all apis that we will implement.
I'm also thinking we will design a frontend where we can navigate all apis with related data or other things. For example, a logging api will have an endpoint URL (copy button) and json data copy button or other data if you think we need to add.

you will guide me as a professional on how to manage a project and to develop that kind of projects, where to start, how to structure, and move forward step by step.
how to structure the files and folder, best practices, and etc.

You will also provide me with some information or data about the api, why we are creating it, and the scenario in the context of our erp system.

We will move step by step.

Web Based Erp Management System – University Project Proposal Copy Edit Download

This is a solid plan, and it mirrors how a **production-grade backend system** is actually built. I'll act as your **technical lead / mentor**, not just a helper. We will move **step by step**, with reasoning behind every decision.

Below is **Step 0: Project Foundation & Roadmap**.
We will not write random APIs first — we will **design the system thinking-first**.

Step 0 — How We Will Build This ERP (Professional Flow)

Our Development Philosophy

We will follow this order (this is how real teams work):

1. **Business Flow Understanding**
2. **System Architecture & Modules**

3. **API Design (WHY before HOW)**
4. **Database Modeling**
5. **Backend Implementation (Express + MongoDB)**
6. **API Explorer Frontend (for testing & learning)**
7. **Security, Logs, Best Practices**
8. **Documentation & Final Polish**

You will *always* know:

- Why an API exists
- What problem it solves
- Which role uses it
- Which data it touches

Step 1 — Understand the ERP Business Flow (Very Important)

Before APIs, we define **HOW THE SYSTEM IS USED**.

Actors (Users of the System)

1. **Admin**
   - Full control
   - Manages users, products, reports
2. **Staff**
   - Creates sales
   - Manages customers
   - Views limited reports

Core Business Flow (Simple)

1. Admin creates users
2. Admin adds products (stock)
3. Staff creates sales (cash / credit)
4. Stock auto-decreases
5. Customer balance updates
6. Expenses are recorded
7. Reports are generated
8. All actions are logged

👉 Every API we build will support **one of these flows**.

Step 2 — Break ERP into Backend Modules

We **never** build “random APIs”.

We build **modules**.

ERP Backend Modules

1. **Auth Module**
2. **User Module**
3. **Product & Stock Module**
4. **Customer Module**
5. **Sales Module**
6. **Expense Module**
7. **Reports Module**
8. **System Logs Module**

Each module:

- Has its own routes
- Has its own controller

- Has its own model
- Has a clear responsibility

This keeps the project **clean, testable, and scalable**.

Step 3 — API Design Strategy (Very Important)

Rule We Will Follow

> One API = One clear responsibility

We do **NOT**:

- Mix logic in routes
- Create “god APIs”
- Write unclear endpoints

API Structure Convention

```arduino
/api/module-name/action
```

Copy code

Examples:

```bash

/api/auth/login
/api/products/create
/api/sales/create
/api/reports/sales
```

Copy code

Step 4 — Initial API List (High-Level)

⚠️ This is NOT final — we will refine each module.

1. Authentication APIs (WHY: Security & User Identity)

- POST `/api/auth/login`
- GET `/api/auth/me`
- POST `/api/auth/logout`

Scenario:

> Without auth, we don’t know *who* is performing sales or updates.

---

2. User APIs (WHY: Role-based access)

- POST `/api/users/create`
- GET `/api/users`
- PUT `/api/users/:id`
- DELETE `/api/users/:id`

Scenario:

> Admin creates staff accounts with limited permissions.

---

3. Product & Stock APIs (WHY: Inventory control)

- POST `/api/products`

- GET `/api/products`
- PUT `/api/products/:id`
- POST `/api/products/adjust-stock`
- GET `/api/stock/logs`

Scenario:

> When a sale happens, stock decreases automatically.

---

4. Customer APIs (WHY: Credit & history tracking)

- POST `/api/customers`
- GET `/api/customers`
- GET `/api/customers/:id`
- GET `/api/customers/:id/ledger`

Scenario:

> ERP must know who owes money and purchase history.

---

5. Sales APIs (WHY: Core business operation)

- POST `/api/sales`
- GET `/api/sales`
- GET `/api/sales/:id`

Scenario:

> Sales affect stock, customer balance, and reports.

---

6. Expense APIs (WHY: Profit calculation)

- POST `/api/expenses`
- GET `/api/expenses`
- GET `/api/expenses/report`

Scenario:

> Profit = Sales – Expenses

---

7. Reports APIs (WHY: Decision making)

- GET `/api/reports/sales`
- GET `/api/reports/stock`
- GET `/api/reports/profit-loss`

Scenario:

> Owner wants monthly/yearly insights.

---

8. System Logs APIs (WHY: Audit & security)

- GET `/api/logs`

Scenario:

> Track who did what, and when.

Step 5 — API Explorer Frontend (Your Idea Is Excellent)

Your idea is **very professional**.

API Explorer Frontend (For Learning & Demo)

Features:

- API list (grouped by module)
- Endpoint URL with **Copy button**
- Sample JSON request body
- Sample JSON response
- Auth token input (JWT)
- Status code display

This:

- Helps you test APIs
- Impresses teachers
- Replaces Postman for demo

👉 We'll build this **after backend basics**.

Step 6 — Folder Structure (Industry Standard)

This is what we'll use:

```pgsql
src/
 ├── app.js
 ├── server.js
 ├── config/
 │    ├── db.js
 │    ├── jwt.js
 │
 ├── modules/
 │    ├── auth/
 │    │    ├── auth.routes.js
 │    │    ├── auth.controller.js
 │    │    ├── auth.service.js
 │    │
 │    ├── users/
 │    ├── products/
 │    ├── customers/
 │    ├── sales/
 │    ├── expenses/
 │    ├── reports/
 │    ├── logs/
 │
 ├── middlewares/
 │    ├── auth.middleware.js
 │    ├── role.middleware.js
 │
 ├── utils/
 │    ├── errorHandler.js
 │
 └── models/
```

We will **never** put logic inside routes.

Step 7 — How We Will Proceed (Clear Next Steps)

NEXT STEP (Step 1.1)

Copy code

👉 **We will fully design the Authentication Module**

- JWT flow
- User roles
- Auth middleware
- APIs + scenarios

Once Auth is solid:

- Everything else becomes easy.

Your Task (Small)

Just confirm:

1. Do you want **JWT-based auth**? (recommended)
2. Semester timeline (weeks remaining)?

Reply with:

> "Yes, let's start with Auth module."

Then we move to **Step 1.1: Authentication System Design**.

JWT-based auth
will use Controllers
passportjs, passport-local, and passport-jwt

Good. This is a **production-correct choice** and aligns well with Express best practices and university expectations.

We will now **lock the authentication architecture** and then design it cleanly.

Step 1.1 — Authentication System Design (JWT + Passport)

We are designing **identity, security, and access control** for the ERP.
Everything else (sales, stock, reports) depends on this.

---

1. Why Passport + JWT (In Context of ERP)

Why Passport?

- Standardized authentication middleware
- Clean separation of auth logic
- Industry-accepted (teachers like this)

Why `passport-local`?

- Handles **email + password login**
- Used only at **login time**

Why `passport-jwt`?

- Used for **every protected request**
- Stateless (perfect for APIs)

- Scales well

Why JWT for ERP?

- ERP is API-based
- Frontend, mobile, or API explorer can reuse token
- No session storage required

---

2. Authentication Flow (Very Important)

Login Flow (Step-by-step)

1. User sends **email + password**
2. `passport-local` verifies credentials
3. Server generates **JWT**
4. JWT is returned to client
5. Client sends JWT in headers for every request

makefile

```
Authorization: Bearer <token>
```

Copy code

Protected Route Flow

1. Request hits API
2. `passport-jwt` extracts token
3. Token is verified
4. User is attached to `req.user`
5. Controller executes logic

If token is invalid → **401 Unauthorized**

---

3. Auth APIs (Final & Clean)

Auth Endpoints

1. Login

bash

```
POST /api/auth/login
```

Copy code

Why:

Allows staff/admin to access ERP.

Request Body

json

```
{
  "email": "admin@example.com",
  "password": "123456"
}
```

Copy code

Response

json

```
{
  "token": "jwt_token_here",
```

```
  "user": {
    "id": "123",
    "name": "Admin",
    "role": "ADMIN"
  }
}
```

Copy code

2. Get Logged-in User

vbnet

```
GET /api/auth/me
```

Copy code

Why:

Frontend needs to know:

- Who is logged in
- What role they have

Header

makefile

```
Authorization: Bearer <token>
```

Copy code

Response

json

```
{
  "id": "123",
  "name": "Admin",
  "email": "admin@example.com",
  "role": "ADMIN"
}
```

Copy code

3. Logout (Optional but Clean)

bash

```
POST /api/auth/logout
```

Copy code

Why:

Mainly frontend-level (JWT is stateless).

We can implement token blacklist later (optional).

---

4. User Model (MongoDB)

This model supports Passport and ERP roles.

js

```
User
- _id
- name
- email (unique)
- password (hashed)
- role (ADMIN | STAFF)
```

```
- isActive
- createdAt
```

Copy code

Why these fields?

- `role` → role-based access
- `isActive` → disable users without deleting
- `password` → hashed with bcrypt

---

5. Passport Strategy Design

Local Strategy (Login Only)

Purpose:

Validate email + password.

Pseudo logic:

- Find user by email
- Compare password with bcrypt
- If valid → return user

Used only in:

bash

```
POST /api/auth/login
```

Copy code

JWT Strategy (Every Protected Route)

Purpose:

Protect APIs and identify user.

Token payload example:

json

```
{
  "id": "123",
  "role": "ADMIN"
}
```

Copy code

Used in:

php

```
passport.authenticate("jwt", { session: false })
```

Copy code

---

6. Auth Middleware (Clean & Reusable)

`auth.middleware.js`

- Verifies JWT
- Attaches user to `req.user`

`role.middleware.js`

- Checks user role

Example:

```js
authorizeRoles("ADMIN")
```

Copy code

Why this matters in ERP:

- Staff cannot delete users
- Staff cannot access reports
- Admin has full control

---

7. Folder Structure (Auth Module)

```bash
src/modules/auth/
 ├── auth.routes.js
 ├── auth.controller.js
 ├── auth.service.js
 ├── passport.local.js
 ├── passport.jwt.js
```

Copy code

Responsibilities

- **routes** → endpoint definitions
- **controller** → request/response
- **service** → business logic
- **passport files** → auth strategies

This structure scores **very well academically**.

---

8. How Auth Connects to ERP Modules

Example:

- Sales API → needs logged-in staff
- Reports API → admin only
- Logs API → admin only

So every module will use:

```pgsql
auth.middleware
role.middleware
```

Copy code

---

9. What We Will Do Next (Step 1.2)

Next Step (Important)

👉 **Implement Auth Module Code**

- User model
- Passport strategies
- Login controller
- JWT generation
- Protected route example

After that:

➡️ **User Management Module**

➡️ **Product & Stock Module**

Confirm Before We Code

Reply with:

1. **JWT expiry** preference (e.g., 1d, 7d)
2. Password hashing → bcrypt (default)

Then we move to **Step 1.2: Writing the Auth Module Code**.