# 🏗️ Scalable Architecture for Petrol Pump Management System

## 🔹 1 Frontend (UI Layer)

**React + TypeScript**

Use **Nextjs** (for server-side rendering & faster load times)

UI libraries: **Material UI / Tailwind CSS** for clean dashboards

Role-based dashboards (Admin, Manager)

✅ Why: Scales well, fast rendering, modern appearance

## 🔹 2 Backend (API + Business Logic Layer)

**Nodejs + NestJS (with TypeScript)** → structured, scalable backend framework

REST or GraphQL APIs for communication with frontend

**Microservices-ready**: You can later break it into smaller services (Payments, Credit Management, Notifications)

**Caching with Redis** → for fast dashboard load & reminders

✅ Why:

TypeScript end-to-end (frontend + backend)

NestJS gives modular structure, making scaling easy

Redis improves speed for repeated queries

## 🔹 3 Database (Data Layer)

**PostgreSQL** → for structured financial data (sales, expenses, stock)

**MongoDB** (optional hybrid) → for unstructured data (uploaded documents, logs)

**Prisma ORM** → for easy database access from TypeScript

✅ Why:

PostgreSQL = strong ACID compliance → reliable for finance

Prisma = clean, type-safe DB access → fewer bugs

## 🔹 4 Authentication & Security

**JWT tokens** → secure login for Admin & Manager

**Role-based access control (RBAC)** → Admin can view all, Manager limited

**HTTPS + Encrypted Storage** → for sensitive data like payments

## 🔹 5 Notifications & Reminders

**Background Jobs with BullMQ (Redis-based)** → schedule credit reminders, expiry alerts

**Email/SMS Gateway** → send notifications automatically

✅ Why: Keeps API fast by handling heavy tasks asynchronously

## 🔹 6 Deployment & Scalability

**Docker** containers → makes deployment consistent

**Kubernetes (K8s)** → for scaling automatically when load increases

**NGINX / API Gateway** → for load balancing & routing

**Cloud Hosting** (AWS/GCP/Azure):

EC2 / Cloud Run → Backend

S3 / CloudFront → Frontend

RDS → PostgreSQL

Elasticache → Redis

✅ Why: Cloud-native = auto-scaling, fault-tolerant, cost-efficient

## 🔹 7 Performance Boosters

**Redis caching** for frequently accessed data (sales summary, dashboard totals)

**CDN (CloudFront)** for frontend assets → faster page loads

**Async Processing** → heavy tasks (report generation, reminders) run in background, not blocking users

# 🟢 Final Tech Stack Summary

**Frontend:** React + TypeScript + Nextjs + Material UI/Tailwind

**Backend:** Nodejs + NestJS (TS) + Redis (caching, jobs)

**Database:** PostgreSQL (+ MongoDB optional) with Prisma ORM

**Auth:** JWT + RBAC

**Deployment:** Docker + Kubernetes + Cloud (AWS/GCP/Azure)

**Performance:** Redis caching + CDN + async jobs

✅ With this, you’ll have:

**Fast response times** (Redis + Nextjs SSR)

**Scalability** (Kubernetes + microservices-ready backend)

**Good appearance** (React + Material UI dashboards)

**Future-proof** (can handle 1 pump → 1000 pumps with same design)

You are an expert backend engineer.

I need a secure production-grade authentication system using \*\*Node.js + Express + TypeScript\*\* with JWT.

Requirements:

1. Implement \*\*auth routes\*\*:

- `POST /auth/register` → create new user

- Hash password with \*\*bcrypt\*\* (world standard salt rounds: 10–12)

- `POST /auth/login` → validate user, return access + refresh tokens

- `POST /auth/refresh` → issue new access token using refresh token

- `POST /auth/logout` → invalidate refresh token (remove from DB/whitelist)

2. Tokens:

- Access token (15m expiry) signed with `JWT\_SECRET`

- Refresh token (7d expiry) stored in DB and sent as \*\*httpOnly, secure cookie\*\*

3. Middleware:

- `authMiddleware.ts` → verify JWT access token for protected routes

- Reject unauthenticated users with 401

4. Database:

- Users table/collection with: `id, email, passwordHash, role, createdAt, updatedAt, lastLogin`

- Store refresh tokens securely (whitelist, not blacklist)

5. Passwords:

- Never store plain text

- Use bcrypt hash + salt

6. Security best practices:

- Helmet middleware for HTTP headers

- CORS restricted to frontend domain

- Rate limiting on login endpoint

7. Add a \*\*protected route\*\* `/dashboard` that only works with valid access token

8. On refresh, issue new tokens without requiring login again.

Deliver clean, modular, enterprise-ready code with controllers, services, and middleware.