# 🏗️ 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)

# Production-ready NestJS ERP — Architecture, Modules & Process

> Purpose: Turn your existing `Node.js + NestJS + Redis + Prisma + PostgreSQL + JWT + RBAC` stack into a world‑class, production-ready, modular platform that can run multiple ERP instances from the same codebase with strong isolation and resilience.

---

## 1. Design goals

\* \*\*Isolation & Resilience:\*\* a fault in one module/service must not cascade to others.

\* \*\*Multi‑ERP support:\*\* single codebase, multiple tenant/ERP instances (config-driven).

\* \*\*Maintainability:\*\* clear module boundaries, consistent coding, automated tests.

\* \*\*Scalability:\*\* horizontally scale per module.

\* \*\*Security & Compliance:\*\* secure secrets, token rotation, auditing.

\* \*\*Observability & Operability:\*\* tracing, metrics, logs, health checks.

---

## 2. High-level architecture (recommended)

Two safe patterns depending on team/scale:

### Option A — Modular Monolith with Process-level Isolation

\* Codebase: single repo with feature modules, shared libs.

\* Runtime: split into \*\*multiple processes\*\* (e.g., `api`, `worker`, `ingest`, `scheduler`, `admin`) — each process runs only selected Nest modules.

\* Communication: HTTP (internal) + Redis/BullMQ + events (Redis streams or NATS).

\* Pros: easier development, single deployment pipeline, lower operational overhead.

\* Use-case: medium teams, want simpler ops but still resilience.

### Option B — Microservices (Service per bounded context)

\* Services: `auth-service`, `users-service`, `org-service`, `erp-core-service`, `integration-service`, `reporting-service`, `notification-service`, `jobs-service`.

\* Communication: gRPC / REST + message broker (NATS / RabbitMQ / Kafka) for events.

\* Pros: stronger isolation, independent scaling & deploys.

\* Use-case: large teams, high-scale, independent release cycles.

> Recommendation: Start with \*\*Modular Monolith + Process isolation\*\*. Split to microservices later along clear module boundaries.

---

## 3. Module & Process breakdown

\*\*Modules (within repo)\*\*

\* `auth` — login, JWT, refresh tokens, MFA, OAuth connectors.

\* `users` — profile, password, roles, permissions.

\* `orgs` — tenant/org settings, quotas, feature toggles.

\* `rbac` — permission model, decorators, guards.

\* `erp-core` — finance, inventory, purchase, sales (split submodules inside).

\* `integrations` — connectors to 3rd‑party ERPs, webhooks, adapters.

\* `jobs` — background processors, scheduled tasks.

\* `cache` — cache utilities, invalidation rules.

\* `notifications` — email, SMS, in-app.

\* `analytics` — metrics aggregation, reporting.

\* `admin` — system admin, health, migration tools.

\* `shared` — DTOs, interfaces, logging, exceptions, utilities.

\*\*Processes (to run separately)\*\*

\* `api` — public REST/GraphQL + auth guards.

\* `worker` — BullMQ processors, heavy jobs.

\* `scheduler` — cron tasks, scheduled reports.

\* `ingest` — external integrations and ETL.

\* `admin` — maintenance tooling (migrations, seeding, admin UI).

Each process imports only necessary modules to reduce blast radius.

---

## 4. Multi‑ERP / Multi‑Tenant strategy

Choose 1 of 3 based on isolation & scale:

1. \*\*Shared schema (row-level tenancy)\*\*

\* Single DB, all tables have `org\_id` column.

\* Pros: simpler operations, cheaper.

\* Cons: noisy neighbors; harder backups per tenant.

2. \*\*Schema-per-tenant\*\*

\* Each tenant gets own DB schema (Postgres schema). Easier logical separation.

3. \*\*Database-per-tenant\*\*

\* Each tenant receives separate database instance.

\* Highest isolation and compliance (GDPR, financial separation).

\*\*Recommendation:\*\* Start with \*\*Shared Schema\*\* with strong row-level filters and `org\_id` enforcement; later move sensitive tenants to schema/db-per-tenant if required.

Implement tenancy in these layers:

\* \*\*Middleware\*\*: resolve tenant from token / host / header, attach `req.tenant`.

\* \*\*Prisma\*\*: include `orgId` in all queries; enforce via centralized repository helpers.

\* \*\*Guards\*\*: ensure `@Roles` + tenant ownership checks.

---

## 5. Error isolation & resilience

\* \*\*Process isolation:\*\* run `api`, `worker`, `scheduler` independently (Docker containers / k8s pods).

\* \*\*Circuit Breakers:\*\* use `opossum` or built-in client libraries to fail fast on flaky downstreams.

\* \*\*Bulkheads:\*\* assign CPU/memory limits; isolate queues & DB pools.

\* \*\*Retries:\*\* exponential backoff for idempotent operations.

\* \*\*Dead-letter queues:\*\* for failed jobs (BullMQ DLQ).

\* \*\*Graceful shutdown:\*\* implement `onModuleDestroy()` and SIGTERM handling.

---

## 6. Inter‑module communication

\* Prefer \*\*async events\*\* for decoupling: publish `user.created`, `invoice.paid`, `stock.reserved`.

\* Use \*\*Redis Streams / BullMQ events\*\* for simple cases; use \*\*NATS/Kafka\*\* for high throughput.

\* For sync needs, use internal HTTP/gRPC with timeouts and circuit breakers.

---

## 7. Data layer: Prisma + PostgreSQL

\* Use explicit repository or service wrappers around Prisma to centralize tenant enforcement and logging.

\* Migrations: use `prisma migrate deploy` in CI/CD; keep migration plan & check for long locks.

\* Indexing: add indexes on `org\_id`, `email`, `created\_at`, and any FK used for joins.

\* Soft delete: `deletedAt` and `isDeleted` pattern.

\* Secrets: DB credentials in secret manager (AWS Secrets Manager / Vault / Kubernetes secrets).

---

## 8. Redis & Job Queue

\* Use \*\*BullMQ\*\* backed by Redis for job queueing.

\* Create dedicated Redis instances for critical subsystems (cache vs queues vs sessions) or logical DB numbers.

\* Implement priority queues for urgent jobs.

\* Monitor queue lengths and worker processing time.

---

## 9. Security checklist

\* Passwords: `argon2` or `bcrypt` with strong config.

\* JWT: short-lived access tokens (15m), refresh tokens (7–30d) hashed and stored in DB.

\* Token rotation: issue new refresh token on use and blacklist previous.

\* Rate limiting: per IP and per org (Throttler middleware).

\* Helmet, CORS, CSP headers.

\* Input validation: DTOs + `class-validator` + sanitize inputs.

\* Secrets management and least-privilege DB roles.

\* Penetration testing and dependency vulnerability scanning (Snyk, Dependabot).

---

## 10. Observability & Monitoring

\* \*\*Tracing:\*\* OpenTelemetry (auto-instrument Nest + Prisma). Correlate traces across processes.

\* \*\*Metrics:\*\* Prometheus + Grafana. Expose `/metrics` endpoint.

\* \*\*Logging:\*\* structured JSON logs (Winston or Pino), ship to ELK/Datadog.

\* \*\*Error tracking:\*\* Sentry for exceptions.

\* \*\*Health checks:\*\* readiness & liveness endpoints for k8s.

---

## 11. CI/CD & Deployment

\* \*\*CI:\*\* run tests, lint, Prisma migrate diff check, build Docker images.

\* \*\*CD:\*\* apply migrations with downtime-minimizing strategy; deploy to k8s or managed containers.

\* Use \*\*feature flags\*\* to enable incremental rollouts.

\* Deploy strategy: Canary or Blue-Green for safe releases.

\* Automate DB backups and test restore procedures.

---

## 12. Folder structure (monorepo, modular)

```

/ (repo)

├─ apps/

│ ├─ api/ # main Nest process (selective modules)

│ ├─ worker/ # job worker process

│ └─ admin-cli/ # admin tools, migrations runner

├─ libs/

│ ├─ shared/ # DTOs, interfaces, validators

│ ├─ prisma/ # schema and client wrapper

│ ├─ auth/ # auth helpers, guards, strategies

│ └─ integrations/ # 3rd party adapters

├─ infra/

│ ├─ docker/ # Dockerfiles, compose

│ └─ k8s/ # k8s manifests & helm charts

├─ scripts/

└─ README.md

```

---

## 13. Example: Running selective modules in a process

\* `api` process bootstrap imports `AuthModule`, `UsersModule`, `OrgModule`, `EPRCoreModule` (API controllers only)

\* `worker` bootstrap imports `JobsModule`, `IntegrationsModule` and processors only

\* This keeps memory footprint low and avoids loading unused code in a process.

---

## 14. Testing strategy

\* Unit tests for services & guards.

\* Integration tests for module endpoints (use test db instances with Docker Compose).

\* E2E tests (Playwright or Cypress for frontend + API flows).

\* Contract testing for inter-service APIs (Pact).

---

## 15. Backups, DR & Compliance

\* Regular DB backups (daily full, hourly WAL if needed).

\* Test restores monthly.

\* Maintain data retention policies and purge flows.

\* Encryption at rest and transit.

---

## 16. Operational checklist before production

\* [ ] Secrets in place (no .env committed)

\* [ ] TLS for all endpoints

\* [ ] Monitoring, tracing, and alerting configured

\* [ ] Auto-scaling policies defined

\* [ ] Disaster recovery plan documented

\* [ ] Pen-test and vuln scans completed

\* [ ] SLOs and error budgets defined

---

## 17. Roadmap / Next steps (practical)

1. Implement process-level splits (create `worker` & `api` entrypoints).

2. Add middleware to resolve tenant and enforce `org\_id` on repositories.

3. Centralize auth: hashed refresh tokens + rotation + blacklist.

4. Add BullMQ for jobs + DLQs.

5. Add OpenTelemetry and Prometheus exporter.

6. Dockerize processes; add k8s manifests.

7. Build CI pipeline (test → build → migrate → deploy).

---

## 18. Appendix: Useful patterns & snippets (links omitted)

\* `@Roles()` decorator example

\* Prisma repository wrapper pattern

\* BullMQ worker structure

\* OpenTelemetry bootstrap in Nest

---

If you want, I can now:

\* generate a \*\*concrete repo skeleton\*\* (files + example code) implementing the above,

\* produce \*\*Dockerfiles & k8s manifest templates\*\*, or

\* draft the \*\*CI pipeline (GitHub Actions)\*\* and Prisma migration/runbook.

Tell me which of the three you'd like first and I will scaffold it.

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.