Backend Handbook

🎯 Purpose

A practical, opinionated handbook for building **production-grade backend** services. It standardizes architecture, code quality, security, observability, and delivery so any engineer can ship reliable APIs and services with confidence.

🧭 Core Principles

- **Clarity > Cleverness**: explicit, simple code.
- **Fail fast in dev; fail gracefully in prod**.
- **Separation of concerns**: API, business logic, data layer.
- **Secure by default**: validate, sanitize, principle of least privilege.
- **Idempotent APIs**: safe to retry.
- **Automate quality**: lint, tests, CI/CD.
- **Observability built-in**: logs, metrics, traces.

🏗️ Project Setup & Tooling

- **Runtime**: Node LTS (locked with .nvmrc).
- **Lang**: TypeScript strict: true.
- **Framework**: Express (with middleware) or Fastify (preferred for perf).
- **Validation**: Zod (request/response schema validation).
- **Database**: MongoDB (Mongoose) or PostgreSQL (Prisma).
- **ORM/ODM**: Prisma or Mongoose; enable strict schemas.
- **Auth**: JWT (short expiry) + Refresh tokens; bcrypt/argon2 for hashing.

- **Testing**: Jest/Vitest (unit), Supertest (integration), Postman/newman (API regression).
- **Quality**: ESLint, Prettier, Husky + lint-staged.
- **Monitoring**: Winston/Pino for logs, Prometheus + Grafana for metrics, Sentry for errors.
- **Env management**: dotenv + Zod to validate envs.
- **Deployment**: Dockerfile, CI/CD (GitHub Actions).

🗂️ Folder Structure (MVC + Services)

```
src/
  app.ts          # bootstrapping, express/fastify instance
  config/         # env config, constants
  middleware/     # auth, rate-limit, error handling
  modules/
    user/
      user.model.ts
      user.service.ts
      user.controller.ts
      user.routes.ts
      user.test.ts
    payment/
      payment.model.ts
      payment.service.ts
      payment.controller.ts
      payment.routes.ts
  shared/
    db.ts
    logger.ts
    utils/
```

Rules:

- Controllers: HTTP layer only (parse input, call service, return response).
- Services: business logic.

- Models: DB schema + methods.
- Shared: cross-cutting utils only.

🔒 Security Standards

- Validate/sanitize all input (Zod schemas).
- Hash + salt passwords (argon2id preferred).
- JWT best practices: short access token, long refresh; revoke refresh on logout.
- Helmet middleware (security headers).
- CORS whitelist.
- Rate limiting + IP blacklisting.
- SQL/NoSQL injection protection via ORM.
- Secrets in env, never in repo.
- Principle of least privilege in DB + cloud creds.

🔗 API Design Guidelines

- REST first; GraphQL only if justified.
- Consistent structure: { data, error } wrapper.
- Use nouns for resources, verbs for actions:
  - POST /users, GET /users/:id, PATCH /users/:id, DELETE /users/:id
- Pagination: ?limit=&offset= or cursor.
- Filtering/sorting via query params.
- Idempotency keys for payment/critical operations.
- Version APIs: /v1/...

🧯 Error Handling

- Centralized error middleware.

- Standard error format:

```
{
  "error": {
    "message": "Invalid input",
    "code": "VALIDATION_ERROR",
    "details": {...}
  }
}
```

- Never leak stack traces in prod.
- Map DB/3rd-party errors to internal error codes.

🧩 Database & Migrations

- Use migrations for schema changes (Prisma migrate, mongoose-migrate).
- Keep seed scripts for local/dev.
- Indexes for query performance; avoid N+1.
- Soft deletes via deletedAt unless truly destructive.
- Always store dates in UTC.

🧠 Business Logic Layer

- Pure functions where possible.
- Services contain all logic; controllers just delegate.
- Idempotent service functions.
- Wrap external calls (payment, email) in adapters for swap/testing.

🔍 Logging & Observability

- Winston/Pino with levels (info, warn, error, debug).
- Request logging: method, path, status, latency, userId.

- Correlation IDs per request.
- Send errors to Sentry with context.
- Metrics: Prometheus middleware (req count, latency, errors).
- Healthcheck endpoint /healthz.

🧪 Testing Strategy

- Unit: pure functions, services.
- Integration: DB + API with Supertest.
- Contract tests: validate API against schema.
- Regression: Postman/newman collection in CI.
- Mock external APIs with MSW/nock.
- Aim: fast local tests, slower e2e in CI nightly.

⚡ Performance Guidelines

- Async/await everywhere; never block event loop.
- DB queries: use indexes, lean queries, projection.
- Cache hot queries in Redis.
- Use message queues (BullMQ, RabbitMQ, Kafka) for async work.
- Batch external API calls.
- Compress responses (gzip/br). Use CDN for assets.

🔁 Release & Ops Hygiene

- Conventional commits + auto changelog.
- Semantic versioning.
- Rollback strategy (blue/green, canary deploys).
- Backups tested.
- Alerts: error rate, latency p95, CPU/memory.

📋 Backend PR Checklist

- Request/response validated with Zod
- Controller thin, service owns logic
- Errors mapped to standard format
- Logs meaningful (no sensitive data)
- Tests updated/added
- DB queries optimized
- Security middleware applied
- Env vars documented

📎 Appendices

- **Recommended libs**: express/fastify, zod, prisma/mongoose, jsonwebtoken, argon2, winston/pino, joi (alt), supertest, jest/vitest, msw/nock, bullmq, redis.
- **VSCode setup**: eslint, prettier, REST Client extension.
- **API docs**: Swagger (OpenAPI) auto-gen.

✅ How to Use This Handbook

1. Scaffold repo with above stack.
2. Keep in /docs/backend-handbook.md.
3. Enforce via CI (lint, type, test).
4. Review with PR checklist before merge.