NestJS Microservices Project Structure

Microservices architecture with multiple services, shared libraries, and message-based communication using NestJS transporters.

#nestjs #typescript #microservices #distributed #redis #tcp

PNG PDF Copy Prompt

Project Directory

```
myproject/
  apps/ Individual serv…
    gateway/ HTTP entry point
      src/
        main.ts Bootstrap HTTP …
        app.module.ts
        app.controller.ts Routes to servi…
        app.service.ts
      tsconfig.app.json
    users/ User microservi…
      src/
        main.ts Bootstrap TCP/R…
        users.module.ts
        users.controller.ts @MessagePattern…
        users.service.ts
      tsconfig.app.json
    orders/ Order microserv…
      src/
        main.ts
        orders.module.ts
        orders.controller.ts
        orders.service.ts
      tsconfig.app.json
  libs/ Shared code
    common/ Shared utilities
      src/
        index.ts Public exports
        dto/
          pagination.dto.ts
        interfaces/
          service-response.interface.ts
      tsconfig.lib.json
    contracts/ Message patterns
      src/
        index.ts
        users.patterns.ts User service pa…
        orders.patterns.ts
      tsconfig.lib.json
  nest-cli.json Monorepo config
  tsconfig.json Base TS config
  package.json
  .env.example
  .gitignore
  README.md
```

Why This Structure?

NestJS monorepo mode organizes microservices in `apps/` with shared code in `libs/`. The gateway handles HTTP and forwards to services via TCP or Redis. Message patterns in `contracts/` ensure type-safe service communication.

Key Directories

apps/gateway/ - HTTP API gateway that proxies to microservices

apps/*/ - Individual microservices with their own bootstrap

libs/common/ - Shared DTOs, interfaces, and utilities

libs/contracts/ - Message pattern constants for type-safe IPC

Getting Started

1. `npm install`
2. `cp .env.example .env` (set REDIS_URL)
3. `npm run start:dev gateway`
4. `npm run start:dev users`
5. `npm run start:dev orders`

Transport Options

TCP - Default, good for local development

Redis - Production-ready pub/sub transport

RabbitMQ - For complex routing and queues

Kafka - High-throughput event streaming

gRPC - Binary protocol, schema-first

When To Use This

- Services need independent scaling
- Teams own separate domains
- Different services need different databases
- Deploying services independently
- Complex business domains with clear boundaries

Trade-offs

Complexity - Network calls replace function calls—handle failures

Data consistency - No transactions across services—use sagas

Debugging - Distributed tracing needed—add correlation IDs