Express Layered Architecture Project Structure

Clean architecture with distinct layers: presentation, application, domain, and infrastructure.

#express #nodejs #typescript #clean-architecture #ddd #layers

PNG PDF Copy Prompt

Project Directory

```
myproject/
  src/
    index.ts  Bootstrap and D…
    presentation/  HTTP layer
      routes/
        index.ts
        user.routes.ts
      controllers/
        user.controller.ts
      middleware/
        auth.middleware.ts
        error.middleware.ts
      dto/  Request/Respons…
        create-user.dto.ts
        user-response.dto.ts
    application/  Use cases
      use-cases/
        create-user.use-case.ts
        get-user.use-case.ts
        list-users.use-case.ts
      interfaces/  Port definitions
        user-repository.interface.ts
        email-service.interface.ts
    domain/  Business rules
      entities/
        user.entity.ts
      value-objects/
        email.vo.ts
        user-id.vo.ts
      errors/
        domain-error.ts
    infrastructure/  External concer…
      database/
        prisma.client.ts
        user.repository.ts  Implements inte…
      services/
        email.service.ts
      config/
        index.ts
    prisma/
      schema.prisma
      migrations/
    tests/
      unit/
      integration/
  package.json
  tsconfig.json
  .env.example
  .gitignore
```

Why This Structure?

Layered architecture keeps business logic independent from HTTP and database details. Domain entities know nothing about Express or Prisma. Use cases orchestrate operations. Infrastructure adapters can be swapped without touching core logic.

Key Directories

`presentation/` - Express routes, controllers, DTOs—HTTP concerns only

`application/` - Use cases orchestrate domain logic, define port interfaces

`domain/` - Entities, value objects, business rules—no dependencies

`infrastructure/` - Database, external APIs, concrete implementations

Getting Started

1. `npm install`
2. `npm install tsyringe reflect-metadata` (for DI)
3. `cp .env.example .env`
4. `npx prisma migrate dev`
5. `npm run dev`

Request Flow

Route → Controller → Use Case → Domain Entity → Repository Interface → Infrastructure Adapter

When To Use This

- Complex business domains with many rules
- Need to swap databases or external services
- Long-lived projects requiring maintainability
- Teams with domain experts defining rules
- Projects requiring extensive unit testing

Trade-offs

`More indirection` - Simple CRUD requires multiple layers

`Steeper learning curve` - Team needs to understand layer boundaries

`More boilerplate` - Interfaces, DTOs, and mappers add code