Clean Architecture Project Structure

Uncle Bob's Clean Architecture with TypeScript. Concentric layers enforcing dependency inversion.

#clean-architecture #typescript #architecture #solid #ddd #uncle-bob

PNG PDF Copy Prompt

Project Directory

```
myproject/
    package.json
    tsconfig.json
    .gitignore
    .env.example
    README.md
  src/
    index.ts App entry point
  domain/ Enterprise busi…
    entities/ Core business o…
      User.ts
      Order.ts
      Product.ts
    value-objects/ Immutable domai…
      Email.ts
      Money.ts
    errors/
      DomainError.ts
  application/ Application bus…
    use-cases/ Application-spe…
      user/
        CreateUser.ts
        GetUserById.ts
        UpdateUser.ts
      order/
        CreateOrder.ts
        GetOrdersByUser.ts
    interfaces/ Abstract contra…
      repositories/
        IUserRepository.ts
        IOrderRepository.ts
      services/
        IEmailService.ts
        IPaymentService.ts
    dtos/ Data transfer o…
      UserDTO.ts
      OrderDTO.ts
  infrastructure/ Frameworks and …
    persistence/ Database implem…
      prisma/
        schema.prisma
        migrations/
      repositories/
        PrismaUserRepository.ts
        PrismaOrderRepository.ts
    http/ Web framework
      server.ts Express/Fastify…
      routes/
        userRoutes.ts
        orderRoutes.ts
      controllers/
        UserController.ts
        OrderController.ts
      middleware/
        auth.ts
        errorHandler.ts
    services/ External servic…
      SendGridEmailService.ts
      StripePaymentService.ts
    config/
      env.ts Environment con…
      container.ts DI container
  tests/
    unit/
      domain/
      use-cases/
    integration/
      repositories/
      api/
```

Why This Structure?

Clean Architecture organizes code in concentric circles: entities at the center, use cases around them, then interfaces, and frameworks at the edge. Dependencies only point inward—infrastructure depends on application, never the reverse. This makes your core logic framework-agnostic.

Key Directories

domain/ - Entities and value objects—zero dependencies on anything

application/ - Use cases orchestrating domain logic, defines interfaces

infrastructure/ - Implements interfaces with real frameworks (Express, Prisma)

application/interfaces/ - Contracts that infrastructure must implement

Use Case Example

```
// application/use-cases/user/CreateUser.ts
export class CreateUser {
  constructor(
    private userRepo: IUserRepository,
    private emailService: IEmailService
  ) {}

  async execute(dto: CreateUserDTO): Promise<User> {
    const user = User.create(dto.email, dto.name);
    await this.userRepo.save(user);
    await this.emailService.sendWelcome(user.email);
    return user;
  }
}
```

When To Use This

- Business logic is complex and needs isolation
- You expect frameworks to change over time
- Testing core logic without infrastructure
- Multiple delivery mechanisms (API, CLI, events)
- Domain experts involved in development

The Four Layers

Entities - Enterprise business rules, used across apps

Use Cases - Application-specific business rules

Interface Adapters - Convert data between layers

Frameworks - Web, DB, external services—details

Trade-offs

More abstractions - Interfaces and mappers add code

Learning curve - Team must understand layer boundaries

Overkill for CRUD - Simple apps don't need this isolation

Testing Strategy

Domain tests - Pure unit tests, no mocks needed

Use case tests - Mock repository/service interfaces

Integration tests - Test real infrastructure implementations

Best Practices

- Use cases should have single responsibility
- Entities contain behavior, not just data
- DTOs cross layer boundaries, entities don't
- Use dependency injection for interface implementations
- Keep controllers thin—delegate to use cases immediately

Naming Conventions

Use cases - Verb + noun: `CreateUser`, `GetOrderById`

Interfaces - Prefix with I: `IUserRepository`

Implementations - Tech prefix: `PrismaUserRepository`