Go Clean Architecture Project Structure

Hexagonal architecture with domain-driven design. Testable, scalable, and framework-agnostic.

#go #golang #clean-architecture #hexagonal #ddd #backend

PNG PDF Copy Prompt

Project Directory

```
myproject/
    go.mod
    go.sum
    .env
    .env.example
    .gitignore
    Makefile  Build, test, li…
    Dockerfile
    docker-compose.yml
  cmd/  Application ent…
    api/
      main.go  Wire up depende…
    worker/  Background work…
      main.go
  internal/
    domain/  Core business l…
      user/
        entity.go  User entity and…
        repository.go  Repository inte…
        service.go  Domain service
        errors.go  Domain errors
      order/
        entity.go
        repository.go
        service.go
    application/  Use cases / orc…
      user/
        create_user.go  One file per us…
        get_user.go
        list_users.go
    infrastructure/  External implem…
      persistence/
        postgres/  PostgreSQL repos
          user_repo.go
          migrations/
      http/  HTTP adapter
        router.go
        handlers/
        middleware/
        dto/  Request/respons…
      messaging/  Queue adapters
        rabbitmq/
      config/
        config.go
  pkg/  Shared utilities
    logger/
    validator/
  test/  Integration tes…
    integration/
    e2e/
```

Why This Structure?

Clean Architecture inverts dependencies so your domain logic has zero external dependencies. The `domain/` layer defines interfaces, `infrastructure/` implements them. This makes the core business logic testable without databases or HTTP and allows swapping frameworks without touching domain code.

Key Directories

`internal/domain/` - Pure business logic, no external imports

`internal/application/` - Use cases orchestrate domain operations

`internal/infrastructure/` - Implements domain interfaces (DB, HTTP, queues)

`cmd/` - Wires everything together with dependency injection

Dependency Inversion

```
// internal/domain/user/repository.go
type UserRepository interface {
    FindByID(ctx context.Context, id string) (*User, error
    Save(ctx context.Context, user *User) error
}

// internal/infrastructure/persistence/postgres/user_repo.
type userRepo struct { db *sql.DB }

func (r *userRepo) FindByID(ctx context.Context, id string
    // Implements domain interface
}
```

Getting Started

1. `go mod init myproject`
2. `docker-compose up -d postgres`
3. Create domain entities and interfaces first
4. Implement infrastructure adapters
5. `make run`

When To Use This

- Complex domain with business rules
- Long-lived projects (2+ years)
- Multiple delivery mechanisms (HTTP, gRPC, CLI)
- Need high test coverage on business logic
- Team of 5+ developers

Trade-offs

Significant boilerplate - Many files and layers for simple operations

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

Learning curve - Team must understand dependency inversion

Initial velocity - Slower start, faster long-term

Best Practices

- Domain layer should have zero imports from infrastructure
- Use interfaces at layer boundaries
- One file per use case in application layer
- Keep HTTP DTOs separate from domain entities
- Write unit tests for domain, integration tests for infra