FastAPI Layered Architecture Project Structure

Clean Architecture with domain, application, and infrastructure layers. Dependency inversion for testability.

#fastapi #python #api #clean-architecture #layered #ddd

PNG PDF Copy Prompt

Project Directory

```
myproject/
    main.py  Composition root
  src/  Source code
    __init__.py
    domain/  Business logic,…
      __init__.py
      entities/
        __init__.py
        user.py  Domain entity
        order.py
      value_objects/
        __init__.py
        email.py
        money.py
      repositories/  Abstract interf…
        __init__.py
        user_repository.py  ABC
        order_repository.py
      services/  Domain services
        __init__.py
        pricing_service.py
    application/  Use cases, orch…
      __init__.py
      use_cases/
        __init__.py
        create_user.py
        place_order.py
        get_user.py
      dto/  Data transfer o…
        __init__.py
        user_dto.py
        order_dto.py
    infrastructure/  External concer…
      __init__.py
      persistence/  Database implem…
        __init__.py
        database.py
        models.py  SQLAlchemy
        user_repository_impl.py  Concrete impl
        order_repository_impl.py
      external/  Third-party ser…
        __init__.py
        email_service.py
        payment_gateway.py
    api/  HTTP layer
      __init__.py
      dependencies.py  DI wiring
      v1/
        __init__.py
        router.py
        users.py
        orders.py
      schemas/  Pydantic reques…
        __init__.py
        user.py
        order.py
  tests/
    __init__.py
    conftest.py
    unit/  Domain/applicat…
      __init__.py
      test_entities.py
      test_use_cases.py
    integration/
      __init__.py
      test_repositories.py
      test_api.py
  requirements.txt
  requirements-dev.txt
  .env.example
  .gitignore
  pyproject.toml
```

Why This Structure?

This structure enforces the Dependency Rule: inner layers know nothing about outer layers. Domain contains pure business logic with zero imports from infrastructure. Use cases orchestrate domain objects. Infrastructure implements interfaces defined in domain.

Key Directories

src/domain/ - Entities, value objects, repository interfaces—no deps

src/application/ - Use cases that orchestrate domain logic

src/infrastructure/ - Database, external APIs—implements domain interfaces

src/api/ - HTTP routes, Pydantic schemas, DI wiring

Getting Started

1. `python -m venv venv && source venv/bin/activate`
2. `pip install -r requirements.txt`
3. `cp .env.example .env`
4. `uvicorn main:app --reload`

Dependency Rule

Domain - No imports from other layers

Application - Imports only from domain

Infrastructure - Imports from domain and application

API - Imports from all layers, wires DI

Repository Pattern

```python
# src/domain/repositories/user_repository.py
from abc import ABC, abstractmethod

class UserRepository(ABC):
    @abstractmethod
    async def get_by_id(self, id: int) -> User | None: ...

# src/infrastructure/persistence/user_repository_impl.py
class SQLAlchemyUserRepository(UserRepository):
    async def get_by_id(self, id: int) -> User | None:
        # actual DB query here
```

When To Use This

- Complex domains with significant business logic
- Projects requiring high testability
- Teams familiar with DDD concepts
- Long-lived applications expecting evolution
- When you need to swap infrastructure easily

Trade-offs

More boilerplate - Interfaces, DTOs, mapping between layers

Learning curve - Team needs to understand layer boundaries

Overkill for CRUD - Simple apps don't benefit from this complexity

Testing Strategy

tests/unit/ - Test domain and use cases with mock repos

tests/integration/ - Test repos against real DB, API e2e

Domain isolation - Unit tests need zero infrastructure