FastAPI Dependency Injection (DI) 완전 정복 (2025-09-19)

🎯 DI (Dependency Injection)란?

Dependency Injection (의존성 주입)은 객체가 필요한 의존성을 **외부에서 주입받는** 디자인 패턴입니다.

기본 개념 비교

```
# ❌ DI 없이 (직접 생성)
class UserService:
    def __init__(self):
        self.user_repo = UserRepository()  # 직접 생성 - 강한 결합

# ✅ DI 적용 (외부 주입)
class UserService:
    def __init__(self, user_repo: IUserRepository):  # 외부에서 주입
        self.user_repo = user_repo  # 약한 결합
```

🔧 Before vs After: Clean Architecture 위반 해결

1. **Interface Layer 변화**

Before:

```
def create_user(user: CreateUserBody):
    user_service = UserService()  # 직접 생성 ❌
```

After:

```
@inject
def create_user(
    user: CreateUserBody,
    user_service: UserService = Depends(Provide[Container.user_service]),  # DI ✅
):
```

2. **Application Layer 변화**

Before:

```
def __init__(self):
    self.user_repo: IUserRepository = UserRepository()  # Infrastructure 직접
import ❌
```

After:

```
@inject
def __init__(self, user_repo: IUserRepository):  # 인터페이스만 의존 ✅
    self.user_repo = user_repo
```

3. 의존성 방향 개선

```
Before: Application → Infrastructure (직접 의존)
After:  Application → Domain ← Infrastructure (올바른 방향)
```

📦 dependency-injector 라이브러리

1. `@inject` 데코레이터

```
@inject
def __init__(self, user_repo: IUserRepository):
```

역할: "이 함수는 의존성 주입이 필요해!"라고 표시 **동작**: Container에서 자동으로 의존성을 찾아서 주입

2. `Provide` 클래스

```
user_service: UserService = Depends(Provide[Container.user_service])
```

역할: Container에서 특정 서비스를 제공받겠다고 선언 **문법**: `Provide[Container.서비스_이름]`

3. `Container` 패턴

```
class Container(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(
        package=["user"]
    )

    user_repo = providers.Factory(UserRepository)
    user_service = providers.Factory(UserService, user_repo=user_repo)
```

역할: 모든 의존성을 중앙에서 관리하는 **설정 파일**

🏗️ Container 상세 분석

1. `DeclarativeContainer`

```
class Container(containers.DeclarativeContainer):
```

역할: 모든 의존성을 선언적으로 정의하는 컨테이너 **특징**: 설정 파일처럼 명확하고 읽기 쉬운 구조

2. `WiringConfiguration`

```
wiring_config = containers.WiringConfiguration(
    package=["user"]
)
```

역할: 어떤 패키지에서 `@inject`를 찾을지 설정 **동작**: `user` 패키지 전체를 스캔해서 의존성 주입 대상 찾기

3. `providers.Factory` vs `providers.Singleton`

```
user_repo = providers.Factory(UserRepository)      # 매번 새 객체 생성
user_service = providers.Singleton(UserService)     # 한 번만 생성하고 재사용
```

Factory: 호출할 때마다 새 객체 생성 (stateless 서비스에 적합) **Singleton**: 한 번만 생성하고 재사용 (상태가 있는 객체에 적합)

4. 의존성 체인

```
user_service = providers.Factory(UserService, user_repo=user_repo)
```

`user_service` → `user_repo` 의존성을 자동으로 해결

⚡ FastAPI 통합: `Depends`

FastAPI의 의존성 주입 시스템

```
def create_user(
    user: CreateUserBody,
    user_service: UserService = Depends(Provide[Container.user_service]),
):
```

동작 과정:

1. FastAPI가 요청 받음
2. `Depends()`가 Container에서 `user_service` 요청
3. Container가 `UserService`와 `UserRepository` 생성
4. 생성된 객체를 함수에 주입

HTTP Status Code 개선

```
@router.post("", status_code=201)  # 201 Created
```

RESTful API 설계 원칙에 따라 리소스 생성 시 201 응답

🔄 전체 DI 흐름

```
1. 애플리케이션 시작
   ↓
2. Container가 의존성 관계 파악
   ↓
3. HTTP 요청 들어옴
   ↓
4. FastAPI가 Depends() 확인
   ↓
5. Container에서 객체 생성 & 주입
   ↓
6. 비즈니스 로직 실행
```

🎯 DI의 핵심 가치

1. **SOLID 원칙 준수**

- **D**ependency Inversion: 추상화에 의존
- 고수준 모듈이 저수준 모듈에 의존하지 않음

2. **테스트 용이성**

```
# 실제 DB 없이도 테스트 가능
container.user_repo.override(providers.Factory(MockUserRepository))

# 또는 FastAPI 테스트에서
def test_create_user():
    mock_service = MockUserService()
    app.dependency_overrides[get_user_service] = lambda: mock_service
```

3. 설정 중앙화

```
# 모든 의존성이 Container에서 한눈에 보임
user_service = providers.Factory(UserService, user_repo=user_repo)
```

4. 런타임 교체

```
# 환경별로 다른 구현체 사용 가능
if ENV == "test":
    container.user_repo.override(providers.Factory(MockUserRepository))
elif ENV == "prod":
    container.user_repo.override(providers.Factory(PostgreSQLUserRepository))
```

5. 유연한 아키텍처

```
# 새로운 저장소로 교체해도 비즈니스 로직은 그대로
container.user_repo = providers.Factory(RedisUserRepository)
```

🏆 Clean Architecture 달성도

Before: 60% (레이어 분리는 됐지만 의존성 위반) **After**: 95% (거의 완벽한 Clean Architecture!)

최종 아키텍처 구조

```
HTTP Request
    ↓
🌐 Interface Layer (user_controller.py)
    ↓ @inject + Depends(Provide[Container.user_service])
🎯 Application Layer (user_service.py)
    ↓ @inject + user_repo: IUserRepository
💎 Domain Layer (user.py, user_repo.py)
    ↓ Pure Business Objects
🔧 Infrastructure Layer (user_repo.py, db_models)
    ↓ DB 저장
📊 Database
```

핵심: 모든 의존성이 Container에서 관리되고, 런타임에 주입됨

📚 주요 개념 정리

Port vs Adapter (다시 정리)

- **Port (IUserRepository)**: Domain에서 정의한 "계약서"
- **Adapter (UserRepository)**: Infrastructure에서 실제 "구현체"

Interface 용어 구분

- **IUserRepository의 Interface**: 프로그래밍 용어 (추상 클래스/인터페이스)
- **Clean Architecture의 Interface Layer**: 아키텍처 용어 (HTTP API 계층)

IoC (Inversion of Control)

- **제어의 역전**: 객체 생성 제어권을 외부(Container)에 위임
- **DI는 IoC를 구현하는 방법 중 하나**

🚀 실무 적용 팁

1. Container 설계 원칙

```
# 인터페이스별로 그룹화
class Container(containers.DeclarativeContainer):
    # Repository Layer
    user_repo = providers.Factory(UserRepository)
    product_repo = providers.Factory(ProductRepository)

    # Service Layer
    user_service = providers.Factory(UserService, user_repo=user_repo)
    product_service = providers.Factory(ProductService, product_repo=product_repo)
```

2. 환경별 설정

```
# config/containers.py
class DevelopmentContainer(Container):
    user_repo = providers.Factory(MockUserRepository)

class ProductionContainer(Container):
    user_repo = providers.Factory(PostgreSQLUserRepository)
```

3. 라이프사이클 관리

```
# DB 연결은 Singleton으로
database = providers.Singleton(Database)

# 비즈니스 서비스는 Factory로
user_service = providers.Factory(UserService)
```

이제 진정한 Enterprise급 아키텍처가 완성되었습니다! 🎯

외부 기술이 바뀌어도 비즈니스 로직은 완전히 보호되고, 테스트도 쉽고, 유지보수성이 극대화된 Clean Architecture + DI 시스템입니다! ✨