FastAPI Clean Architecture 분석 (2025-09-17)

Clean Architecture 개요

Clean Architecture는 소프트웨어를 **동심원 구조**로 설계하는 방법입니다.

핵심 원칙

- **의존성 방향**: 바깥쪽 → 안쪽으로만 의존 (안쪽은 바깥쪽을 모름)
- **관심사 분리**: 각 레이어는 명확한 단일 책임
- **독립성**: 외부 기술 변경이 비즈니스 로직에 영향을 주지 않음

4개 레이어

1. **Domain** - 비즈니스 로직, 엔티티
2. **Application** - 유스케이스, 서비스
3. **Infrastructure** - 데이터베이스, 외부 API
4. **Interface** - 컨트롤러, API 엔드포인트

---

1. Domain Layer - 비즈니스의 핵심

`user/domain/user.py` - 엔티티

```
@dataclass
class User:
    id: str
    name: str
    email: str
    password: str
    created_at: datetime
    updated_at: datetime
```

🎯 설계 철학:

- **Pure Business Object**: FastAPI, SQLAlchemy 등 기술적 의존성 0
- **Immutable Design**: `@dataclass`로 구조체처럼 동작
- **Domain Language**: 비즈니스 용어 그대로 사용

🔍 세부 분석:

- `id: str` → ULID 사용 의도 (UUID보다 정렬 가능)
- `password: str` → 이미 암호화된 상태로 저장
- `created_at/updated_at` → 감사(Audit) 목적

`user/domain/repository/user_repo.py` - 포트(Port)

```python
class IUserRepository(metaclass=ABCMeta):
    @abstractmethod
    def save(self, user: User):
        raise NotImplementedError

    @abstractmethod
    def find_by_email(self, email: str) -> User:
        """
        이메일로 유저를 검색한다.
        검색한 유저가 없을 경우 422 에러를 발생시킨다.
        """
        raise NotImplementedError
```

🎯 **설계 철학:**

- **Hexagonal Architecture의 Port**: 외부와의 계약서
- **의존성 역전**: Domain이 Infrastructure를 모름
- **테스트 용이성**: Mock 구현체로 쉽게 테스트 가능

---

2. Application Layer - 유스케이스 오케스트레이션

`user/application/user_service.py`

```python
class UserService:
    def __init__(self):
        self.user_repo: IUserRepository = UserRepository()  # DI
        self.ulid = ULID()
        self.crypto = Crypto()

    def create_user(self, name: str, email: str, password: str):
        # 1. 중복 검사
        try:
            _user = self.user_repo.find_by_email(email)
        except HTTPException as e:
            if e.status_code != 422:  # 422가 아닌 에러는 재발생
                raise e

        if _user:
            raise HTTPException(status_code=422)  # 중복 유저

        # 2. 도메인 객체 생성
        now = datetime.now()
        user: User = User(
            id=self.ulid.generate(),
            name=name,
            email=email,
            password=self.crypto.encrypt(password),
            created_at=now,
            updated_at=now,
```

```
        )

        # 3. 저장
        self.user_repo.save(user)
        return user
```

🎯 설계 철학:

- **Use Case Pattern**: "회원가입"이라는 비즈니스 시나리오 구현
- **Service Layer**: 도메인 객체들을 조합하여 복잡한 비즈니스 로직 처리
- **Transaction Script**: 절차적 스타일로 명확한 흐름

🔍 비즈니스 로직 흐름:

1. **중복 검사**: 422 예외 처리로 "유저 없음"을 정상 플로우로 처리
2. **도메인 규칙**: 이메일 중복 불허
3. **보안**: 패스워드 암호화
4. **ID 생성**: ULID로 고유성 보장

예외 처리 전략:

```
if e.status_code != 422:
    raise e
```

- 422(유저 없음)는 정상, 다른 에러는 시스템 오류로 처리

---

3. Infrastructure Layer - 기술적 구현

`user/infra/db_models/user.py` - ORM 모델

```
class User(Base):
    __tablename__ = "User"

    id: Mapped[str] = mapped_column(String(36), primary_key=True)
    name: Mapped[str] = mapped_column(String(32), nullable=False)
    email: Mapped[str] = mapped_column(String(64), nullable=False, unique=True)
    password: Mapped[str] = mapped_column(String(64), nullable=False)
    created_at: Mapped[datetime] = mapped_column(DateTime, nullable=False)
    updated_at: Mapped[datetime] = mapped_column(DateTime, nullable=False)
```

🎯 설계 철학:

- **Persistence Model**: 순수 데이터베이스 관점의 모델
- **SQLAlchemy 2.0 Style**: 최신 타입 힌트 방식 사용
- **Domain과 분리**: 같은 이름이지만 완전히 다른 목적

🔍 **데이터 제약사항:**

- `email: unique=True` → DB 레벨에서 중복 방지
- `String(36)` → ULID 길이에 맞춤
- `String(64)` → 암호화된 패스워드 길이 고려

`user/infra/repository/user_repo.py` - Adapter 구현

```
class UserRepository(IUserRepository):
    def save(self, user: UserVO):
        new_user = User(  # Domain → DB 모델 변환
            id=user.id,
            email=user.email,
            name=user.name,
            password=user.password,
            created_at=user.created_at,
            updated_at=user.updated_at,
        )

        with SessionLocal() as db:
            try:
                db.add(new_user)
                db.commit()
            finally:
                db.close()

    def find_by_email(self, email: str) -> UserVO:
        with SessionLocal() as db:
            user = db.query(User).filter(User.email == email).first()

        if not user:
            raise HTTPException(status_code=422)

        return UserVO(**row_to_dict(user))  # DB → Domain 변환
```

🎯 **설계 철학:**

- **Hexagonal Architecture의 Adapter**: Port를 실제 기술로 구현
- **Object Mapping**: Domain ↔ Persistence 변환 담당
- **Transaction Management**: DB 세션 관리

🔍 **모델 변환 패턴:**

```
# Domain → Infrastructure
new_user = User(id=user.id, ...)

# Infrastructure → Domain
return UserVO(**row_to_dict(user))
```

4. Interface Layer - 외부 세계와의 접점

`user/interface/controllers/user_controller.py`

```
from fastapi import APIRouter
from pydantic import BaseModel
from user.application.user_service import UserService

router = APIRouter(prefix="/users")

class CreateUserBody(BaseModel):
    name: str
    email: str
    password: str

@router.post("")
def create_user(user: CreateUserBody):
    user_service = UserService()
    created_user = user_service.create_user(
        name=user.name,
        email=user.email,
        password=user.password
    )

    return created_user
```

🎯 설계 철학:

- **API Gateway Pattern**: HTTP 요청을 Application 레이어로 라우팅
- **DTO Pattern**: `CreateUserBody`로 입력 데이터 검증
- **Presentation Layer**: 외부 프로토콜(HTTP)과 내부 비즈니스 로직 분리

🔍 컨트롤러 책임:

1. **요청 파싱**: HTTP → Python 객체
2. **서비스 호출**: Application 레이어 위임
3. **응답 변환**: Python 객체 → HTTP JSON

입력 검증:

```
class CreateUserBody(BaseModel):
    name: str
    email: str
    password: str
```

- **Pydantic**: FastAPI와 통합된 자동 검증
- **Type Safety**: 런타임 타입 체크
- **API 문서**: 자동 OpenAPI 스키마 생성

전체 아키텍처 흐름

```
HTTP Request
    ↓
🌐 Interface Layer (user_controller.py)
    ↓ CreateUserBody → name, email, password
🎯 Application Layer (user_service.py)
    ↓ 비즈니스 로직: 중복 검사, 암호화, ID 생성
💎 Domain Layer (user.py, user_repo.py)
    ↓ Pure Business Objects
🔧 Infrastructure Layer (user_repo.py, db_models)
    ↓ DB 저장
📊 Database
```

핵심 설계 패턴들

1. 의존성 역전 (DIP)

```
UserService → IUserRepository ← UserRepository
```

2. 레이어 분리

- **Interface**: HTTP 관심사
- **Application**: 비즈니스 흐름
- **Domain**: 핵심 규칙
- **Infrastructure**: 기술 구현

3. 모델 분리

- `CreateUserBody` (입력 DTO)
- `User` (Domain Entity)
- `User` (DB Model)

각각 다른 관심사를 가진 완벽한 Clean Architecture 구현!

학습 포인트

1. **테스트 용이성**: DB 없이도 비즈니스 로직 테스트 가능
2. **유연성**: DB를 MySQL에서 PostgreSQL로 바꿔도 핵심 로직은 그대로
3. **독립성**: 프레임워크가 바뀌어도 도메인 로직은 보호
4. **비즈니스 로직 보호**: 외부 기술로부터 핵심 로직 분리

핵심은 **비즈니스 로직을 외부 기술로부터 보호**하는 것! 🎯