FastAPI Pytest 설정 가이드

목차

---

1. 프로젝트 구조

1.1 권장 구조 패턴

패턴 A: 모듈별 분리 (소규모 프로젝트)

```
project/
├── pyproject.toml
├── app/
│   ├── __init__.py
│   ├── main.py
│   ├── database/
│   │   ├── __init__.py
│   │   └── session.py
│   ├── users/
│   │   ├── __init__.py
│   │   ├── models.py
│   │   ├── router.py
│   │   └── service.py
│   └── orders/
│       ├── __init__.py
│       ├── models.py
│       └── router.py
└── tests/                          # 프로젝트 루트에 tests 분리
    ├── conftest.py                 # 전체 공통 설정
    ├── test_main.py
    ├── users/
    │   ├── conftest.py             # users 테스트 전용 설정
    │   ├── test_models.py
    │   ├── test_router.py
    │   └── test_service.py
    └── orders/
        ├── conftest.py             # orders 테스트 전용 설정
        └── test_router.py
```

장점:

- conftest.py 모듈 경로가 명확: `conftest`, `users.conftest`, `orders.conftest`
- 테스트 코드와 소스 코드 분리
- CI/CD에서 테스트만 격리하기 쉬움

pyproject.toml 설정:

```
[tool.pytest.ini_options]
pythonpath = "."
testpaths = ["tests"]
```

패턴 B: 모듈 내부 tests (중대규모 프로젝트)

```
project/
├── pyproject.toml
└── app/
    ├── __init__.py
    ├── main.py
    ├── conftest.py              # app 전체 공통 설정
    ├── database/
    │   ├── __init__.py
    │   └── session.py
    ├── users/
    │   ├── __init__.py
    │   ├── models.py
    │   ├── router.py
    │   ├── service.py
    │   └── tests/
    │       ├── conftest.py      # users 테스트 전용
    │       ├── test_models.py
    │       └── test_service.py
    └── orders/
        ├── __init__.py
        ├── models.py
        ├── router.py
        └── tests/
            ├── conftest.py      # orders 테스트 전용
            └── test_router.py
```

장점:

- 모듈과 테스트가 함께 있어 유지보수 용이
- 큰 프로젝트에서 모듈별 독립성 유지

주의사항:

- 각 `tests/conftest.py`의 모듈 경로가 유일해야 함
  - `app.users.tests.conftest` ✓

- `app.orders.tests.conftest` ✓

pyproject.toml 설정:

```
[tool.pytest.ini_options]
pythonpath = "."
testpaths = ["app"]
```

패턴 C: 계층적 구조 (대규모 프로젝트)

```
project/
├── pyproject.toml
└── tests/
    ├── conftest.py              # 레벨 1: 전체 공통
    ├── fixtures/                # 공통 fixture 모듈
    │   ├── __init__.py
    │   ├── database.py
    │   └── clients.py
    ├── unit/
    │   ├── conftest.py          # 레벨 2: unit 테스트 공통
    │   ├── test_models.py
    │   └── test_services.py
    ├── integration/
    │   ├── conftest.py          # 레벨 2: integration 공통
    │   ├── test_api.py
    │   └── test_database.py
    └── e2e/
        ├── conftest.py          # 레벨 2: e2e 테스트 공통
        └── test_workflows.py
```

장점:

- 테스트 종류별 설정 분리
- 각 레벨별 fixture 재사용 최적화

pyproject.toml 설정:

```
[tool.pytest.ini_options]
pythonpath = "."
testpaths = ["tests"]
```

1.2 conftest.py 모듈 경로 규칙

올바른 구조 (모듈명 유일)

```
✓ tests/conftest.py                    → 모듈: conftest
✓ tests/unit/conftest.py               → 모듈: unit.conftest
✓ tests/integration/conftest.py        → 모듈: integration.conftest
✓ app/conftest.py                      → 모듈: app.conftest
✓ app/users/tests/conftest.py          → 모듈: app.users.tests.conftest
```

잘못된 구조 (모듈명 충돌)

```
✗ app/users/tests/conftest.py          → 모듈: tests.conftest
✗ app/orders/tests/conftest.py         → 모듈: tests.conftest (충돌!)

✗ tests/users/conftest.py              → 모듈: users.conftest
✗ app/users/conftest.py                → 모듈: users.conftest (충돌!)
```

핵심 원칙:

> pytest는 각 conftest.py를 Python 모듈로 import합니다. 동일한 모듈 경로를 가진 conftest.py가 여러 개 있으면 `ImportPathMismatchError` 발생.

---

2. pyproject.toml 설정

2.1 기본 설정 (복사해서 사용 가능)

```toml
[project]
name = "your-project"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = [
    "fastapi[standard]>=0.121.0",
    "sqlalchemy[asyncio]>=2.0.0",
    "pydantic-settings>=2.0.0",
]

[dependency-groups]
dev = [
    "pytest>=8.0.0",
    "pytest-asyncio>=0.23.0",
    "pytest-cov>=4.1.0",           # 커버리지 측정
    "httpx>=0.27.0",               # FastAPI TestClient
    "faker>=24.0.0",               # 테스트 데이터 생성
]

[tool.pytest.ini_options]
# 필수 설정
pythonpath = "."                   # 프로젝트 루트를 Python path에 추가
testpaths = ["tests"]              # 테스트 검색 경로
```

```
asyncio_mode = "auto"              # 비동기 테스트 자동 감지
asyncio_default_fixture_loop_scope = "function"  # fixture 스코프

# 상세 출력 설정
addopts = [
    "-v",                          # verbose: 각 테스트 이름 출력
    "-s",                          # stdout/stderr 출력 표시
    "--tb=short",                  # traceback을 짧게 표시
    "--strict-markers",            # 등록되지 않은 마커 사용 시 에러
    "--cov=app",                   # app 디렉토리 커버리지 측정
    "--cov-report=term-missing",   # 커버되지 않은 줄 표시
    "--cov-report=html",           # HTML 리포트 생성
]

# 테스트 파일 패턴
python_files = ["test_*.py", "*_test.py"]
python_classes = ["Test*"]
python_functions = ["test_*"]

# 마커 등록
markers = [
    "unit: Unit tests",
    "integration: Integration tests",
    "e2e: End-to-end tests",
    "slow: Slow running tests",
    "db: Tests requiring database",
]

# 경고 필터
filterwarnings = [
    "error",                       # 모든 경고를 에러로 처리
    "ignore::DeprecationWarning",  # DeprecationWarning 무시
    "ignore::PendingDeprecationWarning",
]
```

2.2 주요 설정 옵션 상세 설명

A. `pythonpath` (중요!)

목적: pytest가 모듈을 import할 때 사용할 Python 경로 설정

사용법:

```
pythonpath = "."           # 프로젝트 루트 추가 (가장 일반적)
pythonpath = [".", "src"]  # 여러 경로 추가
```

예시:

```
# 프로젝트 구조
project/
├── pyproject.toml
├── app/
│   └── database/
│       └── session.py
└── tests/
    └── test_db.py

# pythonpath = "." 설정 시
# tests/test_db.py에서 다음 import 가능:
from app.database.session import engine  # ✓ 작동
```

선택 가능한 값:

- `"."`: 프로젝트 루트 (가장 일반적)
- `"src"`: src 레이아웃 사용 시
- `[".", "lib"]`: 여러 경로

B. `testpaths` (중요!)

목적: pytest가 테스트를 검색할 디렉토리 지정

사용법:

```
testpaths = ["tests"]                    # 단일 경로
testpaths = ["tests", "integration"]     # 여러 경로
testpaths = ["app"]                      # app 내부의 tests/ 검색
```

동작 방식:

```
# testpaths = ["tests"] 설정 시
pytest                    # tests/ 디렉토리만 스캔
pytest app/               # tests/ 무시하고 app/ 스캔 (오버라이드)

# testpaths 미설정 시
pytest                    # 현재 디렉토리 전체 스캔 (느림)
```

선택 가능한 값:

값	설명	사용 시기
`["tests"]`	루트의 tests 디렉토리만	패턴 A (테스트 분리)
`["app"]`	app 내부 tests/ 검색	패턴 B (모듈 내부 테스트)

값	설명	사용 시기
`["tests", "integration"]`	여러 테스트 디렉토리	테스트 종류별 분리
미설정	전체 프로젝트 스캔	권장하지 않음 (느림)

C. `asyncio_mode`

목적: 비동기 테스트 처리 방식 설정

선택 가능한 값:

```
asyncio_mode = "auto"      # @pytest.mark.asyncio 자동 감지 (권장)
asyncio_mode = "strict"    # 명시적 마킹 필수
asyncio_mode = "legacy"    # 구버전 호환 모드
```

비교:

```
# auto 모드 (권장)
async def test_api():      # 자동으로 비동기 테스트 인식
    result = await some_async_function()
    assert result is not None

# strict 모드
@pytest.mark.asyncio       # 명시적 마킹 필수
async def test_api():
    result = await some_async_function()
    assert result is not None
```

D. `addopts` **(실행 옵션)**

목적: pytest 실행 시 기본 옵션 지정

자주 사용하는 옵션:

```
addopts = [
    # 출력 관련
    "-v",                  # verbose: 각 테스트 이름 표시
    "-vv",                 # 더 상세한 출력 (assert 값 표시)
    "-s",                  # stdout/stderr 출력 (print 문 표시)
    "-q",                  # quiet: 간결한 출력

    # Traceback 관련
    "--tb=short",          # 짧은 traceback (권장)
    "--tb=long",           # 긴 traceback (디버깅용)
    "--tb=no",             # traceback 숨김
```

```
    "--tb=line",              # 한 줄로 표시

    # 실행 제어
    "-x",                     # 첫 실패 시 중단
    "--maxfail=3",            # 3번 실패 시 중단
    "--lf",                   # 마지막 실패한 테스트만 재실행
    "--ff",                   # 실패한 테스트 먼저 실행

    # 성능
    "-n auto",                # pytest-xdist: 병렬 실행 (CPU 코어 수만큼)
    "--durations=10",         # 가장 느린 10개 테스트 표시

    # 커버리지
    "--cov=app",              # app 디렉토리 커버리지
    "--cov-report=term",      # 터미널 출력
    "--cov-report=html",      # HTML 리포트
    "--cov-fail-under=80",    # 80% 미만 시 실패

    # 경고
    "--strict-markers",       # 미등록 마커 사용 시 에러
    "--disable-warnings",     # 모든 경고 숨김
    "-W error",               # 경고를 에러로 처리
]
```

실무 추천 조합:

개발 중:

```
addopts = ["-v", "-s", "--tb=short", "--lf"]
```

CI/CD:

```
addopts = [
    "-v",
    "--tb=short",
    "--cov=app",
    "--cov-report=xml",
    "--cov-fail-under=80",
    "--maxfail=1"
]
```

E. `markers` **(마커 등록)**

목적: 테스트 그룹화 및 선택적 실행

설정:

```
markers = [
    "unit: Unit tests - 단위 테스트",
    "integration: Integration tests - 통합 테스트",
    "e2e: End-to-end tests - E2E 테스트",
    "slow: Slow running tests - 느린 테스트",
    "db: Tests requiring database - DB 필요",
    "external: Tests calling external APIs - 외부 API 호출",
]
```

사용:

```
# tests/test_service.py
@pytest.mark.unit
def test_calculate():
    assert calculate(2, 3) == 5

@pytest.mark.integration
@pytest.mark.db
async def test_create_user(db_session):
    user = await create_user(db_session, "test@example.com")
    assert user.id is not None

@pytest.mark.slow
@pytest.mark.external
async def test_send_email():
    result = await send_email("test@example.com")
    assert result is True
```

실행:

```
pytest -m unit                    # unit 테스트만
pytest -m "integration and db"    # DB 사용하는 integration 테스트만
pytest -m "not slow"              # slow 제외
pytest -m "unit or integration"   # unit 또는 integration
```

2.3 환경별 설정 예시

로컬 개발 환경

```
[tool.pytest.ini_options]
pythonpath = "."
testpaths = ["tests"]
asyncio_mode = "auto"
addopts = [
    "-v",
```

```
    "-s",
    "--tb=short",
    "--lf",                          # 마지막 실패 테스트 재실행
    "--durations=5",                 # 느린 테스트 5개 표시
]
markers = ["unit", "integration", "slow", "db"]
```

CI/CD 환경

```
[tool.pytest.ini_options]
pythonpath = "."
testpaths = ["tests"]
asyncio_mode = "auto"
addopts = [
    "-v",
    "--tb=short",
    "--cov=app",
    "--cov-report=xml",              # CI에서 읽을 수 있는 XML
    "--cov-report=term-missing",
    "--cov-fail-under=80",           # 80% 미만 실패
    "--maxfail=1",                   # 빠른 피드백
    "-n auto",                       # 병렬 실행
]
```

---

3. conftest.py 사용법

3.1 conftest.py 역할

conftest.py는 **테스트 설정 및 공유 fixture**를 정의하는 특수 파일입니다.

주요 기능:

1. Fixture 정의 (DB 세션, 테스트 클라이언트 등)
2. 테스트 환경 설정 (환경변수, mock 등)
3. pytest hook 커스터마이징
4. 공통 유틸리티 함수

스코프:

- conftest.py는 **해당 디렉토리와 하위 디렉토리**의 모든 테스트에 적용
- 상위 디렉토리의 conftest.py도 자동 로드 (계층적)

3.2 계층적 conftest.py 구조

```
tests/
├── conftest.py                  # 레벨 1: 전체 공통
```

```
│   └── 제공: db_engine, settings
│
├── unit/
│   ├── conftest.py              # 레벨 2: unit 전용
│   │   └── 제공: mock_db
│   └── test_models.py
│       └── 사용 가능: db_engine, settings, mock_db
│
└── integration/
    ├── conftest.py              # 레벨 2: integration 전용
    │   └── 제공: db_session, test_client
    └── test_api.py
        └── 사용 가능: db_engine, settings, db_session, test_client
```

로딩 순서:

```
1. tests/conftest.py 로드
2. tests/integration/conftest.py 로드
3. tests/integration/test_api.py 실행
   → 모든 상위 conftest.py의 fixture 사용 가능
```

3.3 실무 conftest.py 예시

tests/conftest.py (전체 공통)

```python
"""전체 테스트 공통 설정"""
import asyncio
import pytest
from typing import AsyncGenerator
from sqlalchemy.ext.asyncio import AsyncSession, create_async_engine,
async_sessionmaker

from app.core.config import settings


# ============================================================
# 1. 이벤트 루프 설정 (비동기 테스트용)
# ============================================================
@pytest.fixture(scope="session")
def event_loop():
    """세션 스코프 이벤트 루프 생성"""
    loop = asyncio.get_event_loop_policy().new_event_loop()
    yield loop
    loop.close()


# ============================================================
# 2. 테스트 환경 설정
```

```python
# ============================================================
@pytest.fixture(scope="session", autouse=True)
def setup_test_env():
    """테스트 환경 변수 설정 (모든 테스트 실행 전 자동 실행)"""
    import os
    os.environ["ENVIRONMENT"] = "test"
    os.environ["DATABASE_URL"] =
"mysql+aiomysql://test:test@localhost:3306/test_db"
    yield
    # Teardown (필요시)


# ============================================================
# 3. 데이터베이스 설정
# ============================================================
@pytest.fixture(scope="session")
async def db_engine():
    """세션 스코프 DB 엔진 (모든 테스트에서 공유)"""
    engine = create_async_engine(
        settings.DATABASE_URL,
        echo=False,
        pool_pre_ping=True,
    )

    # 테스트 시작 전: 테이블 생성
    async with engine.begin() as conn:
        from app.database.models import Base
        await conn.run_sync(Base.metadata.create_all)

    yield engine

    # 테스트 종료 후: 테이블 삭제 및 엔진 정리
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.drop_all)
    await engine.dispose()


@pytest.fixture
async def db_session(db_engine) -> AsyncGenerator[AsyncSession, None]:
    """함수 스코프 DB 세션 (각 테스트마다 새로운 세션)"""
    async_session = async_sessionmaker(
        db_engine,
        class_=AsyncSession,
        expire_on_commit=False,
    )

    async with async_session() as session:
        # 트랜잭션 시작
        async with session.begin():
            yield session
            # 테스트 종료 후 자동 롤백 (다음 테스트에 영향 없음)
            await session.rollback()
```

```python
# ============================================================
# 4. 테스트 데이터 Factory
# ============================================================
@pytest.fixture
def user_factory():
    """테스트 사용자 생성 팩토리"""
    from app.users.models import User

    def _create_user(**kwargs):
        defaults = {
            "email": "test@example.com",
            "username": "testuser",
            "is_active": True,
        }
        defaults.update(kwargs)
        return User(**defaults)

    return _create_user


# ============================================================
# 5. pytest hook 커스터마이징
# ============================================================
def pytest_configure(config):
    """pytest 초기화 시 실행"""
    print("\n🚀 Starting test suite...")


def pytest_collection_modifyitems(items):
    """테스트 수집 후 수정"""
    # DB 마커가 있는 테스트를 뒤로 이동 (빠른 테스트 먼저)
    items.sort(key=lambda item: "db" in [mark.name for mark in
item.iter_markers()])
```

---

tests/integration/conftest.py (통합 테스트 전용)

```python
"""통합 테스트 전용 설정"""
import pytest
from httpx import AsyncClient, ASGITransport
from typing import AsyncGenerator

from app.main import app


# ============================================================
# 1. FastAPI 테스트 클라이언트
# ============================================================
@pytest.fixture
async def client(db_session) -> AsyncGenerator[AsyncClient, None]:
    """FastAPI 테스트 클라이언트"""
```

```
    from app.database.session import get_session

    # DB 세션 의존성 오버라이드
    async def override_get_session():
        yield db_session

    app.dependency_overrides[get_session] = override_get_session

    # 비동기 클라이언트 생성
    async with AsyncClient(
        transport=ASGITransport(app=app),
        base_url="http://test"
    ) as ac:
        yield ac

    # Teardown
    app.dependency_overrides.clear()


# ============================================================
# 2. 인증된 클라이언트
# ============================================================
@pytest.fixture
async def authenticated_client(client, db_session, user_factory) -> AsyncClient:
    """인증된 테스트 클라이언트"""
    from app.auth.service import create_access_token

    # 테스트 사용자 생성
    user = user_factory(email="auth@example.com")
    db_session.add(user)
    await db_session.commit()

    # JWT 토큰 생성
    token = create_access_token(user.id)

    # Authorization 헤더 추가
    client.headers["Authorization"] = f"Bearer {token}"

    return client


# ============================================================
# 3. 테스트 데이터 시드
# ============================================================
@pytest.fixture
async def seed_users(db_session, user_factory):
    """다수의 테스트 사용자 생성"""
    users = [
        user_factory(email=f"user{i}@example.com", username=f"user{i}")
        for i in range(10)
    ]
    db_session.add_all(users)
    await db_session.commit()
    return users
```

tests/unit/conftest.py (단위 테스트 전용)

```python
"""단위 테스트 전용 설정"""
import pytest
from unittest.mock import AsyncMock, MagicMock


# ============================================================
# 1. Mock 객체
# ============================================================
@pytest.fixture
def mock_db_session():
    """Mock DB 세션"""
    session = AsyncMock()
    session.commit = AsyncMock()
    session.rollback = AsyncMock()
    session.close = AsyncMock()
    return session


@pytest.fixture
def mock_redis():
    """Mock Redis 클라이언트"""
    redis = AsyncMock()
    redis.get = AsyncMock(return_value=None)
    redis.set = AsyncMock(return_value=True)
    redis.delete = AsyncMock(return_value=1)
    return redis


# ============================================================
# 2. 환경 격리
# ============================================================
@pytest.fixture(autouse=True)
def reset_singletons():
    """싱글톤 객체 초기화 (테스트 격리)"""
    # 캐시 초기화
    from app.core.cache import cache
    cache.clear()

    yield

    # Teardown
    cache.clear()
```

3.4 conftest.py Best Practices

1. Fixture 스코프 선택

```
# scope="session": 전체 테스트 세션 동안 1번만 생성 (DB 엔진 등)
@pytest.fixture(scope="session")
async def db_engine():
    engine = create_async_engine(...)
    yield engine
    await engine.dispose()

# scope="module": 모듈(파일)당 1번 생성
@pytest.fixture(scope="module")
def settings():
    return Settings()

# scope="function" (기본값): 각 테스트마다 생성 (DB 세션 등)
@pytest.fixture
async def db_session():
    async with session_maker() as session:
        yield session
```

2. autouse 플래그

```
# autouse=True: 명시하지 않아도 모든 테스트에 자동 적용
@pytest.fixture(autouse=True)
def setup_logging():
    """모든 테스트 실행 전 로깅 설정"""
    import logging
    logging.basicConfig(level=logging.DEBUG)
```

3. Fixture 체이닝

```
@pytest.fixture
async def db_engine():
    """레벨 1: 엔진"""
    ...

@pytest.fixture
async def db_session(db_engine):
    """레벨 2: 세션 (엔진 의존)"""
    ...

@pytest.fixture
async def test_user(db_session):
    """레벨 3: 사용자 (세션 의존)"""
    user = User(email="test@example.com")
    db_session.add(user)
    await db_session.commit()
```

```
    return user

# 테스트에서는 최상위 fixture만 사용
async def test_get_user(test_user):
    # db_engine, db_session은 자동으로 생성됨
    assert test_user.email == "test@example.com"
```

---

4. FastAPI 테스트 패턴

4.1 API 엔드포인트 테스트

```
# tests/integration/test_users_api.py
import pytest
from httpx import AsyncClient


@pytest.mark.integration
async def test_create_user(client: AsyncClient):
    """사용자 생성 API 테스트"""
    response = await client.post(
        "/api/v1/users",
        json={
            "email": "newuser@example.com",
            "username": "newuser",
            "password": "secret123"
        }
    )

    assert response.status_code == 201
    data = response.json()
    assert data["email"] == "newuser@example.com"
    assert "id" in data
    assert "password" not in data  # 비밀번호는 응답에 포함되지 않아야 함


@pytest.mark.integration
async def test_get_user_unauthorized(client: AsyncClient):
    """인증 없이 사용자 조회 시 401"""
    response = await client.get("/api/v1/users/me")
    assert response.status_code == 401


@pytest.mark.integration
async def test_get_user_authenticated(authenticated_client: AsyncClient):
    """인증된 사용자 조회"""
    response = await authenticated_client.get("/api/v1/users/me")

    assert response.status_code == 200
    data = response.json()
    assert data["email"] == "auth@example.com"
```

4.2 데이터베이스 테스트

```python
# tests/integration/test_user_repository.py
import pytest
from sqlalchemy.ext.asyncio import AsyncSession

from app.users.repository import UserRepository
from app.users.models import User


@pytest.mark.integration
@pytest.mark.db
async def test_create_user(db_session: AsyncSession):
    """사용자 생성 테스트"""
    repo = UserRepository(db_session)

    user = await repo.create(
        email="test@example.com",
        username="testuser"
    )

    assert user.id is not None
    assert user.email == "test@example.com"
    assert user.created_at is not None


@pytest.mark.integration
@pytest.mark.db
async def test_get_user_by_email(db_session: AsyncSession, user_factory):
    """이메일로 사용자 조회"""
    # Given: 사용자 생성
    user = user_factory(email="find@example.com")
    db_session.add(user)
    await db_session.commit()

    # When: 이메일로 조회
    repo = UserRepository(db_session)
    found = await repo.get_by_email("find@example.com")

    # Then: 올바른 사용자 반환
    assert found is not None
    assert found.email == "find@example.com"


@pytest.mark.integration
@pytest.mark.db
async def test_get_user_not_found(db_session: AsyncSession):
    """존재하지 않는 사용자 조회"""
    repo = UserRepository(db_session)
    user = await repo.get_by_email("notfound@example.com")
```

```
    assert user is None
```

4.3 서비스 레이어 테스트

```python
# tests/unit/test_user_service.py
import pytest
from unittest.mock import AsyncMock

from app.users.service import UserService
from app.users.exceptions import UserAlreadyExistsError


@pytest.mark.unit
async def test_create_user_success(mock_db_session, user_factory):
    """사용자 생성 성공"""
    # Mock repository
    mock_repo = AsyncMock()
    mock_repo.get_by_email.return_value = None  # 중복 없음
    mock_repo.create.return_value = user_factory(id=1)

    # Service 실행
    service = UserService(mock_repo)
    user = await service.create_user("new@example.com", "newuser")

    # 검증
    assert user.id == 1
    mock_repo.create.assert_called_once()


@pytest.mark.unit
async def test_create_user_duplicate(mock_db_session, user_factory):
    """중복 이메일로 사용자 생성 시 에러"""
    # Mock repository: 이미 존재하는 사용자
    mock_repo = AsyncMock()
    mock_repo.get_by_email.return_value = user_factory()

    # Service 실행 및 예외 검증
    service = UserService(mock_repo)
    with pytest.raises(UserAlreadyExistsError):
        await service.create_user("existing@example.com", "user")
```

---

5. 실무 Best Practices

5.1 테스트 실행 명령어 모음

```
# 기본 실행
pytest                                    # 모든 테스트

# 마커별 실행
pytest -m unit                            # unit 테스트만
pytest -m "integration and not slow"      # 빠른 integration 테스트만
pytest -m "db"                            # DB 테스트만

# 특정 파일/디렉토리
pytest tests/unit/                        # unit 테스트 디렉토리
pytest tests/test_users.py                # 특정 파일
pytest tests/test_users.py::test_create   # 특정 테스트 함수

# 출력 제어
pytest -v                                 # verbose
pytest -vv                                # 더 상세한 출력
pytest -s                                 # print 출력 표시
pytest -q                                 # quiet

# 실패 제어
pytest -x                                 # 첫 실패 시 중단
pytest --maxfail=3                        # 3번 실패 시 중단
pytest --lf                               # 마지막 실패 테스트만
pytest --ff                               # 실패한 테스트 먼저

# 커버리지
pytest --cov=app                          # 커버리지 측정
pytest --cov=app --cov-report=html        # HTML 리포트
pytest --cov-fail-under=80                # 80% 미만 실패

# 병렬 실행 (pytest-xdist 필요)
pytest -n auto                            # CPU 코어 수만큼 병렬
pytest -n 4                               # 4개 워커로 병렬

# 성능 분석
pytest --durations=10                     # 느린 10개 테스트 표시
pytest --profile                          # 프로파일링

# 디버깅
pytest --pdb                              # 실패 시 pdb 디버거 시작
pytest --trace                            # 시작부터 pdb
```

5.2 CI/CD 통합

GitHub Actions 예시

```
# .github/workflows/test.yml
name: Tests
```

```yaml
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:8.0
        env:
          MYSQL_ROOT_PASSWORD: root
          MYSQL_DATABASE: test_db
        ports:
          - 3306:3306
        options: >-
          --health-cmd="mysqladmin ping"
          --health-interval=10s
          --health-timeout=5s
          --health-retries=5

    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          pip install uv
          uv sync --dev

      - name: Run tests
        env:
          DATABASE_URL: mysql+aiomysql://root:root@localhost:3306/test_db
        run: |
          uv run pytest \
            --cov=app \
            --cov-report=xml \
            --cov-report=term-missing \
            --cov-fail-under=80

      - name: Upload coverage
        uses: codecov/codecov-action@v3
        with:
          file: ./coverage.xml
```

요약

핵심 체크리스트

프로젝트 구조

- ☐ conftest.py 모듈 경로가 유일한가?
- ☐ testpaths가 올바르게 설정되었는가?
- ☐ pythonpath가 프로젝트 루트를 포함하는가?

pyproject.toml

- ☐ pytest-asyncio 설치 및 asyncio_mode 설정
- ☐ 마커 등록 (unit, integration, slow 등)
- ☐ 적절한 addopts 설정 (로컬/CI 분리)

conftest.py

- ☐ 계층적 구조로 fixture 정리
- ☐ 적절한 fixture 스코프 사용 (session/module/function)
- ☐ DB 트랜잭션 격리 구현

테스트 작성

- ☐ AAA 패턴 (Arrange-Act-Assert) 준수
- ☐ 의미 있는 테스트 이름
- ☐ 적절한 마커 사용
- ☐ 테스트 격리 보장 (서로 영향 없음)

이제 실무 수준의 pytest 설정과 테스트 작성이 가능합니다!