파이보 뷰/서비스 레이어 분리 리팩토링

- 대상 코드 : https://github.com/pahkey/djangobook/tree/3-09
- 완성 코드 : https://github.com/SCKIMOSU/view_refactoring.git

1. 전통적인 views

- `views.py` 파일은 전통적인 방식으로 로직을 포함하고 있으며, 서비스 레이어 분리 없이 뷰에서 직접 DB 연산과 검증을 수행하고 있음.
- 이를 다음과 같이 리팩토링함:

🔧 리팩토링 계획

1. **서비스 로직 분리**
   - `services/question_service.py` 및 `services/answer_service.py`로 분리
2. **예외 처리**
   - 커스텀 예외 클래스 생성 (`exceptions.py`)
3. **단위 테스트 작성**
   - `tests/test_question_service.py`, `tests/test_answer_service.py`
4. **뷰 파일 리팩토링**
   - 뷰는 서비스 호출만 하도록 단순화

✅ 예시 리팩토링 (1단계: `answer_create`)

📁 `pybo/services/answer_service.py`

```
from django.utils import timezone
from django.shortcuts import get_object_or_404
from pybo.models import Answer, Question

class AnswerService:
    @staticmethod
```

```
    def create_answer(user, question_id, form):
        if not form.is_valid():
            raise ValueError("Invalid form data")

        question = get_object_or_404(Question, pk=question_id)
        answer = form.save(commit=False)
        answer.author = user
        answer.create_date = timezone.now()
        answer.question = question
        answer.save()
        return answer
```

📁 pybo/views.py (변경된 뷰)

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import redirect, render
from pybo.forms import AnswerForm
from pybo.services.answer_service import AnswerService

@login_required(login_url='common:login')
def answer_create(request, question_id):
    if request.method == "POST":
        form = AnswerForm(request.POST)
        try:
            AnswerService.create_answer(request.user, question_id, form)
            return redirect('pybo:detail', question_id=question_id)
        except Exception as e:
            return render(request, 'pybo/question_detail.html', {
                'form': form,
                'error': str(e)
            })
```

📁 pybo/tests/test_answer_service.py

```
import pytest
from django.contrib.auth.models import User
from pybo.models import Question
```

```
from pybo.forms import AnswerForm
from pybo.services.answer_service import AnswerService

@pytest.mark.django_db
def test_create_answer_valid():
    user = User.objects.create_user(username='tester', password='pass')
    question = Question.objects.create(subject="test", content="content", a
uthor=user)

    form_data = {'content': 'test answer'}
    form = AnswerForm(data=form_data)

    answer = AnswerService.create_answer(user, question.id, form)

    assert answer.content == 'test answer'
    assert answer.author == user
    assert answer.question == question
```

이제 위와 같은 방식으로 `QuestionService`, `QuestionForm`, `index`, `detail` 함수도 분리할 수 있음.

2. 리팩토링 views

전체 리팩토링 작업에 대한 **단계별 상세 설명과 코드 요약**입니다.

✅ 전체 구조

```
pybo/
├── views.py                    ← 서비스만 호출 (로직 없음)
├── services/
│   ├── question_service.py     ← 질문 관련 로직 (CRUD)
│   └── answer_service.py       ← 답변 관련 로직 (CRUD)
├── tests/
│   ├── test_question_service.py  ← 질문 서비스 테스트
│   └── test_answer_service.py    ← 답변 서비스 테스트
```

🛠️ 단계별 상세 설명

1. 서비스 레이어 생성

`question_service.py`

```
class QuestionService:
    def create_question(user, form)
    def modify_question(question, form)
    def delete_question(question)
```

- **역할**: 질문을 생성/수정/삭제하는 비즈니스 로직을 뷰에서 분리
- **장점**: 로직의 재사용성 증가, 테스트 용이

`answer_service.py`

```
class AnswerService:
    def create_answer(user, question_id, form)
    def modify_answer(answer, form)
    def delete_answer(answer)
```

- **역할**: 답변 처리 로직 집중 관리
- **추가**: `modify_date` 자동 갱신

---

2. 뷰 (`views.py`) 단순화

- 모든 CRUD 작업은 **서비스에서 처리**
- 뷰는 사용자 요청 처리, 폼 생성/전달, 권한 검사만 수행

예시:

```
@login_required
def question_create(request):
    form = QuestionForm(request.POST or None)
    if request.method == 'POST' and form.is_valid():
        QuestionService.create_question(request.user, form)
        return redirect('pybo:index')
    return render(request, 'pybo/question_form.html', {'form': form})
```

---

3. 예외 처리

- `raise PermissionDenied`를 통해 권한 검사
- 폼 검증 실패 시 `ValueError` 예외 발생 후 messages 처리

---

4. 단위 테스트 (예시)

`test_question_service.py`

```
def test_create_question():
    ...
    form = QuestionForm(data={...})
    q = QuestionService.create_question(user, form)
    assert q.subject == '...'
```

- `pytest`로 테스트 가능
- DB 테스트 위해 `@pytest.mark.django_db` 필요

🧪 실행 방법

```
pytest pybo/tests/
```

---

3. 리팩토링된 전체 프로젝트 구조

- 아래는 리팩토링 완료된 전체 Django 프로젝트 구조
- 각 구성요소는 기능별로 잘 정리되어 있습니다:

📦 전체 프로젝트 구조 (`djangobook-3-09`)

```
djangobook-3-09/
├── manage.py                  # Django 실행 진입점
├── db.sqlite3                 # SQLite DB 파일
├── .gitignore

├── config/                    # Django 설정 모듈
```

```
│   ├── settings.py
│   ├── urls.py
│   ├── wsgi.py
│   └── asgi.py

├── common/                    # 회원가입/로그인 기능 담당 앱
│   ├── views.py
│   ├── forms.py
│   ├── urls.py
│   └── models.py

├── pybo/                      # QnA 주요 앱
│   ├── views.py               # 뷰: 서비스 호출만 수행
│   ├── urls.py
│   ├── models.py
│   ├── forms.py
│   ├── admin.py
│   ├── templatetags/
│   │   └── pybo_filter.py
│   ├── migrations/
│   │   ├── 0001_initial.py 등
│   ├── services/              # 💡 서비스 레이어
│   │   ├── question_service.py    # 질문 등록/수정/삭제/조회
│   │   └── answer_service.py      # 답변 등록/수정/삭제
│   ├── tests/                 # ✅ 단위 테스트
│   │   ├── test_question_service.py
│   │   └── test_answer_service.py
│   └── exceptions/            # 예외 정의 (확장 가능)
│       └── __init__.py

├── static/                    # 정적 리소스 (CSS/JS)
│   ├── bootstrap.min.css
│   ├── bootstrap.min.js
│   ├── jquery-3.4.1.min.js
│   └── style.css

└── templates/
    ├── base.html              # 레이아웃 템플릿
```

```
    ├── navbar.html
    ├── form_errors.html
    ├── common/
    │   ├── login.html
    │   └── signup.html
    └── pybo/
        ├── question_list.html
        ├── question_detail.html
        ├── question_form.html
        └── answer_form.html
```

🔍 핵심 변경 사항 요약

영역	리팩토링 전	리팩토링 후
`views.py`	비즈니스 로직 포함	서비스 호출만 수행
`services/`	없음	질문/답변 CRUD 로직 분리
`tests/`	단일 tests.py	기능별 테스트 파일 분리
`exceptions/`	없음	커스텀 예외 분리 구조 마련
`templates/`	변경 없음	(뷰 리팩토링에 따라 연동만 필요)

---

4. 리팩토링된 서비스 레이어 코드 테스트

- Django 프로젝트에서 서비스 레이어로 분리된 코드를 **테스트하는 방법**.
- 이 설명은 `pytest` 기반 단위 테스트를 기준으로 합니다.

✅ 1. 테스트 환경 설정

▶ `pytest` 및 `pytest-django` 설치

터미널에서 다음 명령어 실행:

```
pip install pytest pytest-django
```

✅ 2. `pytest.ini` 설정 파일 생성

프로젝트 루트(`manage.py`가 있는 위치)에 아래 내용의 파일을 만드세요:

📄 `pytest.ini`

```
[pytest]
DJANGO_SETTINGS_MODULE = config.settings
python_files = tests.py test_*.py *_tests.py
```

> config.settings는 settings.py의 위치를 의미합니다 (예: config/settings.py).

✅ 3. 테스트 실행

루트 디렉토리(예: `djangobook-3-09/`)에서 아래 명령어를 입력하면 전체 테스트 실행됩니다:

```
pytest
```

또는 `pybo/tests` 디렉토리만 테스트하려면:

```
pytest pybo/tests/
```

✅ 4. 테스트 코드 예시

📄 `pybo/tests/test_question_service.py`

```
import pytest
from django.contrib.auth.models import User
from pybo.models import Question
from pybo.forms import QuestionForm
from pybo.services.question_service import QuestionService

@pytest.mark.django_db
def test_create_question():
    user = User.objects.create_user(username='tester', password='1234')
    form = QuestionForm(data={'subject': '제목', 'content': '내용'})
    question = QuestionService.create_question(user, form)

    assert question.subject == '제목'
```

```
    assert question.author == user
    assert Question.objects.count() == 1
```

✅ 5. PyCharm 또는 VS Code에서 테스트 실행

- **PyCharm**: 테스트 함수 우클릭 → `Run pytest for...`
- **VS Code**: `Python Test Explorer` 플러그인 사용 가능

---

5. 테스트시 오류 발생 예1

- **테스트 코드에서 `create_date` 필드를 지정**
- 오류 원인은 **테스트 코드에서 `create_date` 필드를 지정하지 않아 DB에서 NOT NULL 제약 조건 위반이 발생**한 것입니다.

❗ 문제 요약

`Question` 모델에는 `create_date` 필드가 `null=False`로 설정되어 있습니다:

```
create_date = models.DateTimeField(null=False)  # 기본값이 없고 null도 안됨
```

그런데 테스트 코드에서 `Question.objects.create(...)`를 할 때 `create_date`를 지정하지 않았기 때문에 `IntegrityError`가 발생한 것입니다.

✅ 해결 방법

테스트에서 `create_date`를 수동으로 설정하거나, 서비스 레이어를 통해 생성해야 합니다.

🔧 해결방안 A: `create_date` 직접 지정 (테스트 수정)

```
from datetime import datetime
from django.utils.timezone import now

@pytest.mark.django_db
def test_get_paginated_questions():
    user = User.objects.create_user(username='tester', password='pass')
    for i in range(15):
        Question.objects.create(
```

```
            subject=f"q{i}",
            content="내용",
            author=user,
            create_date=now()  # ← 이 줄 추가!
        )
```

동일하게:

```
@pytest.mark.django_db
def test_get_question_by_id():
    user = User.objects.create_user(username='tester', password='pass')
    question = Question.objects.create(
        subject="test",
        content="내용",
        author=user,
        create_date=now()  # ← 추가
    )
```

🔧 해결방안 B: `QuestionService.create_question()` 사용 권장

보다 깔끔한 방법은 서비스 레이어의 생성 메서드를 사용하는 것입니다.

```
form = QuestionForm(data={'subject': '제목', 'content': '내용'})
question = QuestionService.create_question(user, form)
```

이 방법은 `create_date`를 서비스 내부에서 자동 처리하므로 추천드립니다.

`test_create_answer_valid` 테스트가 실패한 이유

현재 `test_create_answer_valid` 테스트가 실패한 이유는 이전과 동일하게 **`Question`을 생성할 때 `create_date` 필드를 지정하지 않았기 때문**입니다.

✅ 해결 방법 (Answer 테스트 수정)

테스트 내에서 `Question` 객체 생성 시 `create_date`를 수동으로 명시해 주세요.

📄 수정된 `test_answer_service.py`

```
import pytest
from django.contrib.auth.models import User
from pybo.models import Question
from pybo.forms import AnswerForm
from pybo.services.answer_service import AnswerService
from django.utils.timezone import now


@pytest.mark.django_db
def test_create_answer_valid():
    user = User.objects.create_user(username='tester', password='pass')
    question = Question.objects.create(
        subject="test",
        content="content",
        author=user,
        create_date=now()  # 🛠️ create_date 명시
    )

    form_data = {'content': 'test answer'}
    form = AnswerForm(data=form_data)

    answer = AnswerService.create_answer(user, question.id, form)

    assert answer.content == 'test answer'
    assert answer.author == user
    assert answer.question == question
```

🔁 그 다음 단계

1. 위 코드로 수정 후 저장
2. 다시 테스트 실행:

```
pytest pybo/tests/
```

> ✅ 이 수정만 하면 모든 테스트가 통과될 것으로 예상됩니다.

---

6. 테스트시 오류 발생 예2

- 암호 필드(`password`)를 직접 비교해서 발생

❌ 오류 설명

```
assert found.author.password == "pass"
```

이런 식의 비교는 **절대 실패합니다. 이유는 다음과 같습니다:**

- Django는 사용자 비밀번호를 **저장할 때 자동으로 해싱**합니다.
- `"pass"`는 평문이지만,
- `found.author.password`는 해시 문자열입니다 (예: `pbkdf2_sha256$260000$...`).

✅ 해결 방법

비밀번호가 맞는지 확인하려면 `check_password()` 메서드를 사용해야 합니다:

```
assert found.author.check_password("pass")
```

🔧 최종 수정된 테스트 함수

```
import pytest
from django.contrib.auth.models import User
from pybo.models import Question
from pybo.services.question_service import QuestionService
from django.utils.timezone import now

@pytest.mark.django_db
def test_get_question_by_id():
    user = User.objects.create_user(username='tester', password='pass')
    question = Question.objects.create(
        subject="test",
        content="내용",
        author=user,
```

```
        create_date=now()
    )
    found = QuestionService.get_question_by_id(question.id)
    assert found.subject == "test"
    assert found.author.username == "tester"
    assert found.author.check_password("pass")  # ✅ 안전한 방식
```

🔁 수정 후 다시 실행

```
pytest pybo/tests/
```

7. 서비스 레이어를 위한 커스텀 예외 클래스(Custom Exceptions) 생성

Django 프로젝트에서 **커스텀 예외 클래스(Custom Exceptions)**를 만들면 **서비스 계층에서 오류를 구분해 핸들링하기 쉽고**, 뷰나 테스트 코드에서도 예외를 명확히 다룰 수 있음.

✅ 1. 커스텀 예외 생성

📁 pybo/exceptions/pybo_exceptions.py

```
class PyboBaseException(Exception):
    """기본 pybo 예외 클래스 (모든 커스텀 예외의 부모)"""
    pass


class InvalidFormException(PyboBaseException):
    """폼 검증 실패 예외"""
    def __init__(self, message="폼 데이터가 유효하지 않습니다."):
        super().__init__(message)


class PermissionDeniedException(PyboBaseException):
    """권한이 없는 작업"""
```

```
    def __init__(self, message="해당 작업을 수행할 권한이 없습니다."):
        super().__init__(message)


class NotFoundException(PyboBaseException):
    """요청한 리소스를 찾을 수 없음"""
    def __init__(self, message="요청한 항목을 찾을 수 없습니다."):
        super().__init__(message)
```

✅ 2. 서비스 계층에서 예외 사용 예시

📄 `question_service.py` 수정 예시

```
from pybo.exceptions.pybo_exceptions import InvalidFormException

@staticmethod
def create_question(user, form):
    if not form.is_valid():
        raise InvalidFormException()
    ...
```

✅ 3. 뷰에서 처리 예시

```
from pybo.exceptions.pybo_exceptions import InvalidFormException

@login_required
def question_create(request):
    if request.method == 'POST':
        form = QuestionForm(request.POST)
        try:
            QuestionService.create_question(request.user, form)
            return redirect('pybo:index')
        except InvalidFormException as e:
            messages.error(request, str(e))
    else:
```

```
        form = QuestionForm()
    return render(request, 'pybo/question_form.html', {'form': form})
```

✅ 4. 장점

- `ValueError`, `Exception`처럼 **일반적인 예외 대신 의미가 명확한 예외**를 사용
- 서비스 로직 → 뷰/컨트롤러 → 사용자 응답 흐름이 **명확해짐**
- 테스트 코드에서도 특정 예외만 `pytest.raises()`로 확인 가능

✅ 5. 테스트 예시

```
import pytest
from pybo.exceptions.pybo_exceptions import InvalidFormException

def test_create_question_with_invalid_form():
    ...
    form = QuestionForm(data={})  # 빈 데이터
    with pytest.raises(InvalidFormException):
        QuestionService.create_question(user, form)
```
