# Day6,7 DRF

| 자료 | DB |
| --- | --- |
| 구분 | DB |



## 1. 기초 설정

가상환경을 켠 후에 패키지 설치까지 하자.

```
# [예시]
$ python -m venv ~/venv
$ source ~/venv/Scripts/activate
$ pip install -r ./requirements.txt
```

```
# 기존에 사용하던 환경이 있으면 사용가능하다
# 단, djangorestframework 는 설치되어 있어야 한다.
# 혹은 requirements.txt 파일을 만들고 복붙을 해도 좋다.

asgiref==3.7.2
asttokens==2.4.0
autopep8==2.0.2
backcall==0.2.0
beautifulsoup4==4.9.3
certifi==2022.12.7
cffi==1.15.1
charset-normalizer==3.1.0
colorama==0.4.6
cryptography==40.0.1
decorator==5.1.1
defusedxml==0.7.1
dj-rest-auth==3.0.0
Django==4.2.6
django-allauth==0.54.0
django-appconf==1.0.5
django-bootstrap-v5==1.0.5
django-cors-headers==3.14.0
django-extensions==3.2.3
django-imagekit==4.1.0
djangorestframework==3.14.0
exceptiongroup==1.1.3
executing==1.2.0
idna==3.4
ipython==8.15.0
jedi==0.19.0
matplotlib-inline==0.1.6
oauthlib==3.2.2
parso==0.8.3
pickleshare==0.7.5
pilkit==2.0
Pillow==9.5.0
prompt-toolkit==3.0.39
pure-eval==0.2.2
pycodestyle==2.10.0
pycparser==2.21
Pygments==2.16.1
PyJWT==2.6.0
python3-openid==3.2.0
pytz==2021.1
requests==2.28.2
requests-oauthlib==1.3.1
six==1.16.0
soupsieve==2.2.1
```

```
sqlparse==0.4.1
stack-data==0.6.2
tomli==2.0.1
traitlets==5.10.0
typing_extensions==4.6.3
tzdata==2023.3
urllib3==1.26.15
wcwidth==0.2.6
```

프로젝트와 필요한 앱을 생성한다.

```
$ django-admin startproject mypjt .
$ python manage.py startapp movies
```

일단, `settings.py` 부터 건드린다.

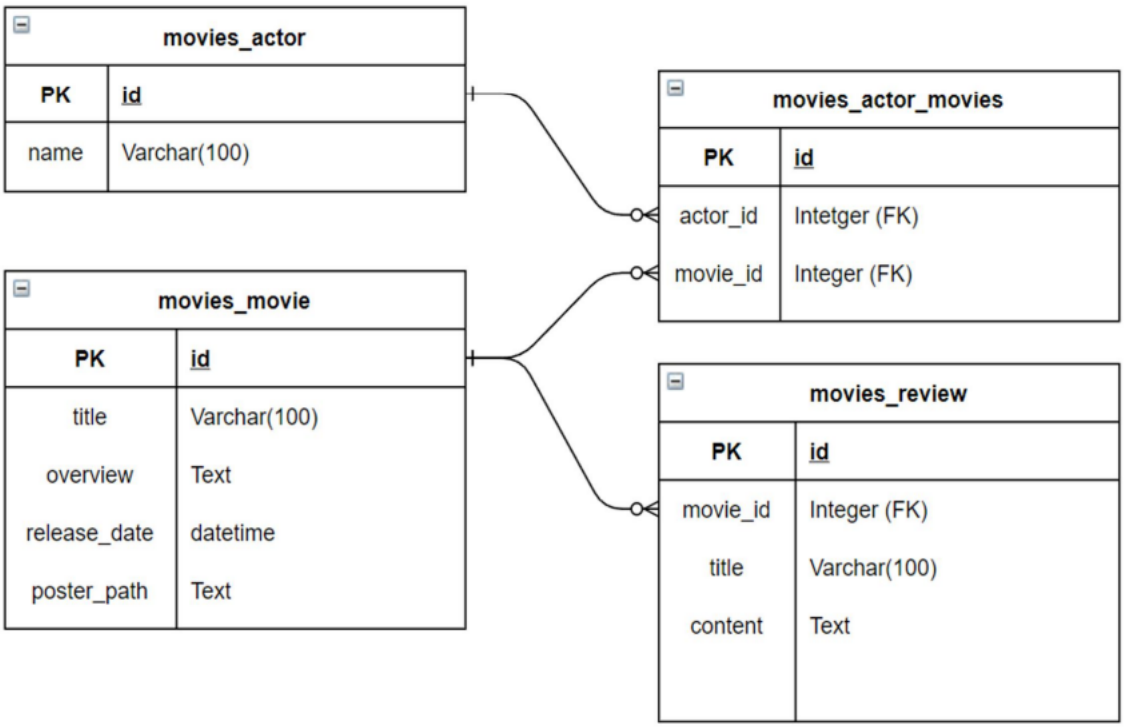
```
# settings.py

INSTALLED_APPS = [
    'movies',
    'rest_framework',
]

LANGUAGE_CODE = 'ko-kr'

TIME_ZONE = 'Asia/Seoul'
```

다음, 전역 `urls.py` 를 설정하자.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('movies.urls')),
]
```

이제 `movies` 앱을 설정하자.

우리가 구현할 ERD 는 다음과 같은 형태이다.




위의 ERD를 기반으로 구현한 `movies/models.py` 는 다음과 같다.

```
from django.db import models


class Actor(models.Model):
    name = models.CharField(max_length=100)


class Movie(models.Model):
    actors = models.ManyToManyField(Actor, related_name='movies')
    title = models.CharField(max_length=100)
    overview = models.TextField()
    release_date = models.DateTimeField()
    poster_path = models.TextField()


class Review(models.Model):
    movie = models.ForeignKey(Movie, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
    content = models.TextField()
```

여기서 눈여겨 볼 부분은 ERD 에서 `movies_actor_movies` 는 브릿지테이블(중개테이블) 이고
실제로 movies 와 actors 의 (M:N)다대다 관계이다. 따라서, `ManyToManyField` 를 사용해
(M:N)다대다 관계로 이어줄 것이라는 것이다.
그리고 `._set` 대신 “역참조”시 사용할 이름으로, `related_name='movies'` 라고 적어 주겠다.

`movies/urls.py` 는 다음과 같다.

```
from django.urls import path
from . import views
```

```
urlpatterns = []
```

`movies/admin.py` 는 다음과 같다.

```
from django.contrib import admin
from .models import Actor, Movie, Review

admin.site.register(Actor)
admin.site.register(Movie)
admin.site.register(Review)
```

이제, 마이그레이션 생성 및 마이그레이트, 관리자 생성하자.

```
$ python manage.py makemigrations
$ python manage.py migrate
$ python manage.py createsuperuser
```

movies.json

actors.json

reviews.json

이후, 주어진 fixture 파일을 로드하자.

```
# json 파일은 movies/fixtures/movies 에 위치하겠다.

$ python manage.py loaddata movies/actors.json movies/movies.json movies/reviews.json
```

그리고, SQLite 를 열어 `movies_actor` 테이블을 확인해보면 다음과 같다.

SQL ▼ 1 / 1 1 - 10 of 10

| id | name |
|---|---|
| 1 | Case imagine simple shake ahead try. |
| 2 | Quite despite how entire second. Tough will actually. |
| 3 | Order I run common man actually tax determine. Coach process letter visit expert house example. |
| 4 | Hundred president tough such. Water because can personal. Produce million when information fall. |
| 5 | Bad last father organization edge. |
| 6 | Evening worry together their hold article not decade. |
| 7 | Give yet notice simple. Positive site size movie. Light without drive during just rate kid. |
| 8 | Card movie feel run authority. Leave throughout decade whom enter. Short often large no. |
| 9 | Save coach result our little professor like use. Son conference game claim administration. |
| 10 | Southern hard often require. Couple find card. Meet instead film property build page bar high. |

다른 테이블도 확인해보자. 특히, `movies_movie_actors` 테이블이 다음과 같이 자동으로 생성되었다.

SQL ▼

| id | movie_id | actor_id |
|---|---|---|
| 1 | 1 | 6 |
| 2 | 2 | 9 |
| 3 | 2 | 6 |
| 4 | 3 | 1 |
| 5 | 3 | 9 |
| 6 | 4 | 8 |
| 7 | 4 | 7 |
| 8 | 5 | 8 |
| 9 | 5 | 1 |
| 10 | 5 | 10 |

## 2. `urls.py`

다음 조건에 맞춰 작성한다.

| | |
|---|---|
| /actors/ | 전체 배우 목록 제공 |
| /actors/<int:actor_pk>/ | 단일 배우 정보 제공 (출연 영화 제목 포함) |

| | |
|---|---|
| /movies/ | 전체 영화 목록 제공 |
| /movies/<int:movie_pk>/ | 단일 영화 정보 제공 (출연 배우 이름과 리뷰 목록 포함) |
| /reviews/ | 전체 리뷰 목록 제공 (영화 제목 포함) |
| /movies/<int:movie_pk>/reviews/ | 리뷰 생성 |
| /reviews/<int:review_pk>/ | 단일 리뷰 조회 & 수정 & 삭제 (영화 제목 포함) |

```
from django.urls import path
from . import views


urlpatterns = [
    path('actors/', views.actor_list),
    path('actors/<int:actor_pk>/', views.actor_detail),
    path('movies/', views.movie_list),
    path('movies/<int:movie_pk>/', views.movie_detail),
    path('reviews/', views.review_list),
    path('movies/<int:movie_pk>/reviews/', views.create_review),
    path('reviews/<int:review_pk>/', views.review_detail),
]
```

`views.py` 를 다음과 같이 최소한만 작성해두자.

```
from rest_framework.decorators import api_view
from .models import Actor, Movie, Review


@api_view(['GET'])
def actor_list(request):
    pass


@api_view(['GET'])
def actor_detail(request, actor_pk):
    pass


@api_view(['GET'])
def movie_list(request):
    pass


@api_view(['GET'])
def movie_detail(request, movie_pk):
    pass


@api_view(['GET'])
def review_list(request):
    pass


@api_view(['POST'])
def create_review(request, movie_pk):
    pass


@api_view(['GET', 'PUT', 'DELETE'])
def review_detail(request, review_pk):
    pass
```

`urls.py` 에 미리 적어놓은 최소한의 함수만 작성했다.

# 3. serializers

actor, movie, review 각 세 개의 serializer 를 정의해 보자.

- serializer : 쿼리문으로 구성된 데이터를 JSON 포맷으로 쉽게 변환하게 해주는 객체

먼저, `/movies/serializers/actor.py` 는 다음과 같다.

```
# /movies/serializers/actor.py

from rest_framework import serializers
from ..models import Actor, Movie

class ActorListSerializer(serializers.ModelSerializer):
    class Meta:
        model = Actor
        fields = '__all__'


class ActorSerializer(serializers.ModelSerializer):
    class MovieSerializer(serializers.ModelSerializer):
        class Meta:
            model = Movie
            fields = ('title',)

    movies = MovieSerializer(many=True, read_only=True)

    class Meta:
        model = Actor
        fields = '__all__'
```

위 코드를 살펴보면 두 개의 serializer 가 존재할 것이다.

`~ListSerializer` 라는 이름은 “모든 데이터” 를 다 가져올 때 사용했다.

그리고 `~Serializer` 는 “하나의 단일 데이터” 를 가져올 때 사용했다.

배우로 치면, 모든 배우 데이터 목록을 가져오는지, 단일 배우 데이터를 가져오는지 의 차이다.

`ActorListSerializer` 의 모델은 `Actor` 이며, 모든 필드를 다 가져와 사용한다.

`ActorSerializer` 는 `MovieSerializer` 를 가지고 있다.

`MovieSerializer` 의 모델은 `Movie` 이며, 제목인 `title` 만 가져다가 사용한다.

여기에서 `MovieSerializer` 는 모든 데이터를 다 가져 오는 것이 아니라, 각각의 단일 영화 데이터를 하나씩 가져 올 것이다.

코드에서 movies = MovieSerializer(many=True, read_only=True) 이 부분을 보자.

여기서 필드가 될 `movies` 는 방금전에 만든 `MovieSerializer` 의 결과이며,

결과는 단일 데이터가 아닐 수 있기 때문에 `many=True` 를 사용했고,

수정이 불가능 하기에 `read_only=True` 를 적어주었다.

즉, 여러 개의 데이터를 조회한 결과가 `movies` 이다.

즉, `movies` 필드는 한 명의 배우가 어떤 영화들에 출연을 했는지를 보여준다.

class Meta:
    model = Actor
    fields = '__all__'

`ActorSerializer` 의 모델은 `Actor` 로 가지며, 모든 필드를 다 가져와 사용한다.

다음, `/movies/serializers/movie.py` 는 다음과 같다.

```
# /movies/serializers/movie.py

from rest_framework import serializers
from ..models import Movie, Actor, Review
```

```
class MovieListSerializer(serializers.ModelSerializer):
    class Meta:
        model = Movie
        fields = ('title', 'overview',)


class MovieSerializer(serializers.ModelSerializer):

    class ActorSerializer(serializers.ModelSerializer):
        class Meta:
            model = Actor
            fields = ('name',)

    class ReviewListSerializer(serializers.ModelSerializer):
        class Meta:
            model = Review
            fields = ('title', 'content')

    actors = ActorSerializer(many=True, read_only=True)
    review_set = ReviewListSerializer(many=True, read_only=True)

    class Meta:
        model = Movie
        fields = '__all__'
```

`MovieListSerializer` 는 `title`, `overview` 필드만 가져와 사용한다.

`MovieSerializer` 는 `ActorSerializer` 와 `ReviewListSerializer` 를 정의해두었다가,
`actors` 와 `review_set` 필드를 만드는 데 사용한다. 즉, 하나의 영화에 어떤 배우들이 있고,
어떤 리뷰들이 있는지를 나타내주는데, 배우는 배우 이름(`name`), 리뷰는 리뷰 제목(`title`)과 내용(`content`)이다.

여기서, 배우는 List 가 없고, 리뷰는 List 가 있는 클래스 이름을 사용했다.
하나의 영화는 여러 명의 “단일” 배우들을 가진다. 각각의 배우 데이터에 접근해서 가져와야
하는 것이다. 따라서 List 가 들어가지 않는 것이 맞다.

그러나 하나의 영화에 들어갈 “리뷰들” 은 고정되어있다. 해당 리뷰가 다른 영화에도 달리는
것이 아니다. 여러 개의 리뷰들은 오로지 하나의 영화에만 들어가기 때문에
List 가 붙는 게 맞다.

`MovieSerializer` 는 `Movie` 모델을 가지며, 모든 필드를 가져다 쓴다.

마지막으로, `/movies/serializers/review.py` 는 다음과 같다.

```
from rest_framework import serializers
from ..models import Review, Movie


class ReviewListSerializer(serializers.ModelSerializer):

    class Meta:
        model = Review
        fields = ('title', 'content',)


class ReviewSerializer(serializers.ModelSerializer):
    class MovieSerializer(serializers.ModelSerializer):
        class Meta:
            model = Movie
            fields = ('title',)

    movie = MovieSerializer(read_only=True)

    class Meta:
```

```
        model = Review
        fields = '__all__'
```

`ReviewListSerializer` 는 `Review` 를 기본 모델로 가지며, 보여줄 필드는 `title`, `content` 이다.

`ReviewSerializer` 는 `MovieSerializer` 를 정의해 사용하는데,

기본 모델은 `Movie` 이고, `title` 필드를 가져온다.

그리고 `MovieSerializer` 의 결과는 `movie` 필드가 되는데,

여기에서는 `many=true` 가 사용되지 않았다. 그 이유는 리뷰는

오로지 하나의 영화에만 해당되기 때문이다. 따라서 여러 영화가 아니라

단 하나의 영화만 가져온다.

`ReviewSerializer` 의 기본 모델은 `Review` 이며, 모든 필드를 다 가져올 것이다.

## 4. `views.py`

차분하게 하나씩 구현해 보자. 우선, 아까전에 만든 `urls.py` 표를 가지고

아래와 같이 `views.py` 를 표로 표현 할 수 있다.

| | |
|---|---|
| actor_list | 전체 배우 목록 제공 |
| actor_detail | 단일 배우 정보 제공 (출연 영화 제목 포함) |
| movie_list | 전체 영화 목록 제공 |
| movie_detail | 단일 영화 정보 제공 (출연 배우 이름과 리뷰 목록 포함) |
| review_list | 전체 리뷰 목록 제공 (영화 제목 포함) |
| create_review | 리뷰 생성 |
| review_detail | 단일 리뷰 조회 & 수정 & 삭제 (영화 제목 포함) |

구현 순서는 다음과 같다.

1. 먼저, 배우, 영화, 리뷰 중 모든 데이터를 가져오는 것들 먼저 구현한다.
    a. - actor_list, movie_list, review_list
2. 다음, 단일 데이터를 가져오는 것들 구현한다.
    a. - actor_detail, movie_detail, review_detail
3. 생성을 구현한다. 이 API 에선 오로지 리뷰에만 존재한다.
    a. - create_review
4. 수정, 삭제를 구현한다. 이 API 에선 오로지 리뷰에만 존재한다.
    a. - review_detail

먼저, `actor_list` 함수이다.

```
# actor_list
from django.shortcuts import get_list_or_404, get_object_or_404
from rest_framework.response import Response
from rest_framework.decorators import api_view
from rest_framework import status
from .models import Actor, Movie, Review
from .serializers.actor import ActorListSerializer, ActorSerializer
from .serializers.movie import MovieListSerializer, MovieSerializer
from .serializers.review import ReviewListSerializer, ReviewSerializer

@api_view(['GET'])
def actor_list(request):
    actors = get_list_or_404(Actor)
    serializer = ActorListSerializer(actors, many=True)
    return Response(serializer.data)
```

`get_list_or_404` 와, `get_object_or_404` , 그리고 만들어둔 serializer 들은 전부 미리 `import` 해두었다.

단계는 다음과 같은데,

1. 먼저 DB 에서 데이터를 가져온다. 가져온 내용은 `actors` 이며, 모든 배우들 목록이 쿼리문으로 리턴되었다. 실패할 경우, 404 에러를 리턴 할 것이다.
2. 우리가 만들어둔 `ActorListSerializer` 는 쿼리문을 JSON 으로 바꾸는 역할을 한다. 첫번째 인자로 `actors` 를 넣고, 단일 데이터가 아니라 여러 데이터를 가져와야 하기에 `many=True` 를 두번째 인자로 넣어 `serializer` 를 리턴받는다.
3. DRF 에서 제공하는 `Response` 객체를 이용해 `serializer` 객체 안에 `data` 만 리턴한다.

결과 확인해보자. Postman 으로 테스트하겠다. `localhost:8000/api/v1/actors/` 로 신호를 보내보자.


```
[
    {
        "id": 1,
        "name": "Case imagine simple shake ahead try."
    },
    {
        "id": 2,
        "name": "Quite despite how entire second. Tough will actually."
    },
    {
        "id": 3,
        "name": "Order I run common man actually tax determine. Coach process letter visit expert house example."
    }
```


배우 목록이 다음과 같은 형태로 잘 들어옴을 알 수 있다.

이 원리대로라면, `movie_list` 함수도 똑같은 형태로 구현 가능하다.

```
# movie_list

@api_view(['GET'])
def movie_list(request):
    movies = get_list_or_404(Movie)
    serializer = MovieListSerializer(movies, many=True)
    return Response(serializer.data)
```

`review_list` 역시 마찬가지다.

```
# review_list

@api_view(['GET'])
def review_list(request):
    reviews = get_list_or_404(Review)
    serializer = ReviewListSerializer(reviews, many=True)
    return Response(serializer.data)
```

각각 Postman 으로 테스트해보자.

다음, 단일 데이터를 가져오는 것들을 구현해야 한다.

먼저, `actor_detail` 이다.

```
# actor_detail

@api_view(['GET'])
def actor_detail(request, actor_pk):
    actor = get_object_or_404(Actor, pk=actor_pk)
```

```
    serializer = ActorSerializer(actor)
    return Response(serializer.data)
```

두번째 인자로 `pk=actor_pk` 를 준 것에 유의하자.

단일 데이터를 찾으려면 "무엇을" 찾아야 되는지 당연히 알아야 한다. 또한 여기에서

`many=True` 를 사용하지 않았다. 단일 데이터를 가져오기 때문이다.

결과 확인해보자. `localhost:8000/api/v1/actors/1` 으로 신호를 보내보겠다.


```
{
    "id": 1,
    "movies": [
        {
            "title": "Administration that great close eight become."
        },
        {
            "title": "Medical summer indicate management sense pay."
        },
        {
            "title": "Cup decade air."
        }
    ],
    "name": "Case imagine simple shake ahead try."
}
```


해당 배우의 이름 (`name`) 뿐만 아니라, 어떤 영화들에 출연했는지 `title` 까지 볼 수 있는데,

우리가 `ActorSerializer` 를 구현할 때 해당 필드를 정의해두었기 때문이다.

또한, `model.py` 작성시, 영화가 역참조를 당할 때 접근할 수 있는 필드 이름을

`related_name='movies'` 로 정의해두었기에, `movie_set` 이라는 이름 대신 `movies` 로 접근 가능하다.

이 원리 대로라면, `movie_detail` , `review_detail` 도 구현 가능하다.

```
# movie_detail

@api_view(['GET'])
def movie_detail(request, movie_pk):
    movie = get_object_or_404(Movie, pk=movie_pk)
    serializer = MovieSerializer(movie)
    return Response(serializer.data)
```

```
# review_detail

@api_view(['GET', 'PUT', 'DELETE'])
def review_detail(request, review_pk):
    review = get_object_or_404(Review, pk=review_pk)
    serializer = ReviewSerializer(review)
    return Response(serializer.data)
```

각각 잘 작동되는지 테스트해보자. 테스트할 때, 어떤 테이블을 역참조 하는지 확인은 필수다.

다음, `create_review` 함수를 구현해보자.

```
from rest_framework import status

@api_view(['POST'])
def create_review(request, movie_pk):
    movie = get_object_or_404(Movie, pk=movie_pk)
    serializer = ReviewSerializer(data=request.data)

    if serializer.is_valid(raise_exception=True):
```

```
        serializer.save(movie=movie)
        return Response(serializer.data, status=status.HTTP_201_CREATED)
```

일단, 어떤 영화에 달아야 할 리뷰일지 알아야 하기 때문에, 해당 영화정보를 가져와야 한다.

그리고, 사용자가 JSON 으로 작성한 정보들을 `ReviewSerializer` 의

`data=request.data` 인자를 사용해 입력하고, 유효성 검사를 해준다.

`raise_exception=True` 로 설정 할 경우, 클라이언트는 어떤 에러가 발생했는지

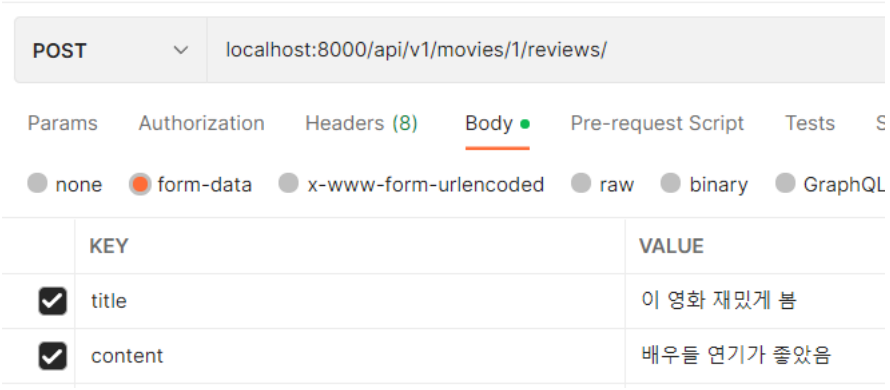
자세하게 볼 수 있다.

유효성검사를 통과하면, `ReviewSerializer` 에 우리가 정의한 `movie` 필드에, 우리가 리뷰를

작성하고자 하는 영화 정보를 넣어주고, `save` 해주며, 추가 성공한 리뷰 내용과 함께

201 상태(자원 생성 성공)를 사용자에게 리턴을 해준다.

Postman 으로 테스트해보자.




다음과 같이 작성한 상태에서 send 버튼을 누르면,

```
{
    "id": 11,
    "movie": {
        "title": "Act why team bag tell over smile themselves."
    },
    "title": "이 영화 재밌게 봄",
    "content": "배우들 연기가 좋았음"
}
```

다음과 같이, 성공한 JSON 이 리턴된다. `GET /movies/1` 요청을 보내보자.

```
{
    "id": 1,
    "actors": [
        {
            "name": "Evening worry together their hold article not decade."
        }
    ],
    "review_set": [
        {
            "title": "Health support surface standard challenge of.",
            "content": "One hit prevent mouth there time. Reach picture reason nature worry drive reveal sometimes.\nDifficult page sp
        },
        {
            "title": "Theory simply around hope throw.",
            "content": "Final create person. Agent language lawyer assume media.\nThan least us why political summer however. Party tra
        },
        {
            "title": "이 영화 재밌게 봄",
            "content": "배우들 연기가 좋았음"
        }
    ],
    "title": "Act why team bag tell over smile themselves.",
```

```
    "overview": "Once feeling according. Follow several Republican best about accept.\nAgency play what report. Know sound shoulder sma
    "release_date": "1978-01-22T21:48:49+09:00",
    "poster_path": "New fish right agreement night. Create name yet smile pay west.\nEvent cause method exist detail new. Fire stand ha
}
```

`review_set` 의 맨 마지막에, 우리가 방금 기록한 리뷰가 확인된다.

다음, 리뷰 삭제다. `review_detail` 함수에서 조회, 수정, 삭제를 모두 담당하기에,
조건문을 사용해 분기를 해 주어야 한다. (나눠줘야 한다)

```
# review_detail

@api_view(['GET', 'PUT', 'DELETE'])
def review_detail(request, review_pk):
    review = get_object_or_404(Review, pk=review_pk)
    if request.method == 'GET':
        serializer = ReviewSerializer(review)
        return Response(serializer.data)
    elif request.method == 'DELETE':
        review.delete()
        data = {
            'delete': f'review {review_pk} is deleted'
        }
        return Response(data, status=status.HTTP_204_NO_CONTENT)
```

조회든, 수정이든, 삭제든 모두 해당 리뷰를 가져와서 활용한다.

`elif request.method == 'DELETE'` 부분을 보면, 해당 리뷰를 지워버리고,
사용자에게 간단한 메세지를 보내며, 삭제 완료를 뜻하는 204 상태를 보낸다.
Postman 으로 테스트해보자. 단, Postman에서 맨 뒤에 슬래시( `/` ) 를 넣지 않으면
무조건 GET 으로 받아들이므로 주의하자.

`DELETE /reviews/2/`

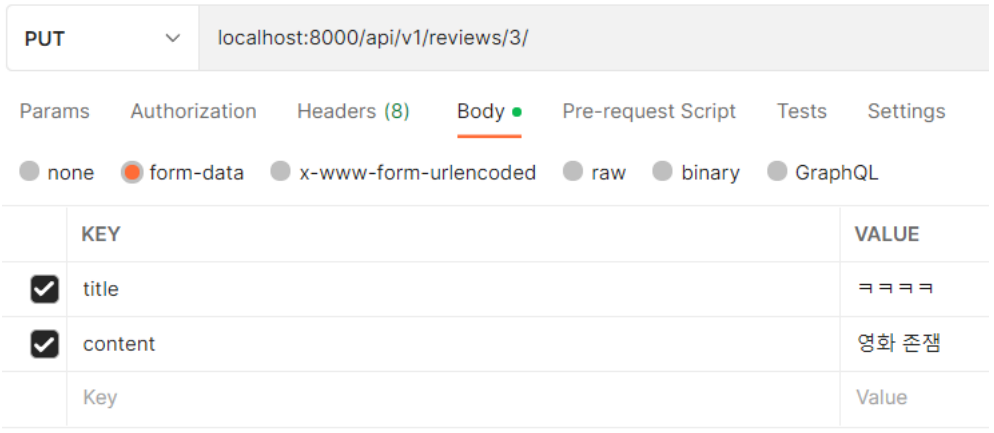
```
{
    "delete": "review 2 is deleted"
}
```

마지막으로, 수정을 구현해보자.

```
@api_view(['GET', 'PUT', 'DELETE'])
def review_detail(request, review_pk):
    review = get_object_or_404(Review, pk=review_pk)

    if request.method == 'GET':
        serializer = ReviewSerializer(review)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = ReviewSerializer(review, data=request.data)
        if serializer.is_valid(raise_exception=True):
            serializer.save()
            return Response(serializer.data)

    elif request.method == 'DELETE':
        review.delete()
        data = {
            'delete': f'review {review_pk} is deleted'
        }
        return Response(data, status=status.HTTP_204_NO_CONTENT)
```

`ReviewSerializer` 의 첫번째 인자로 수정 대상을 넣고,
두번째 인자로 사용자가 작성한 수정 내역을 넣는다.
만약 유효성검사를 통과하면 `save` 한다.

Postman 으로 테스트해보자.




리뷰 3번 수정을 위해 다음과 같이 보냈고,

```
{
    "id": 3,
    "movie": {
        "title": "Military maintain tree close rich four."
    },
    "title": "ㅋㅋㅋㅋ",
    "content": "영화 존잼"
}
```

다음과 같은 리턴값을 받았다. `GET /reviews/3` 을 요청해 바뀌었는지 보자.

<끝>