Day6,7 DRF

자료	DB
구분	DB
과목	

movies.json

actors.json

reviews.json

REST API 란.txt

1. 기초 설정

가상환경을 켠 후에 패키지 설치까지 하자.

```
$ python -m venv venv
$ source ./venv/Scripts/activate
```

requirements.txt 파일을 만들고 복사 그리고 붙여넣기를 하자.

```
asgiref==3.8.1
autopep8==2.3.1
Django==4.2.16
djangorestframework==3.15.2
pycodestyle==2.12.1
pytz==2024.1
sqlparse==0.5.1
tomli==2.0.2
typing_extensions==4.12.2
tzdata==2024.2
```

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')),
]
```

클라이언트가 요청할 때 사용 하게 될 URL 경로를 작성해 보았다.

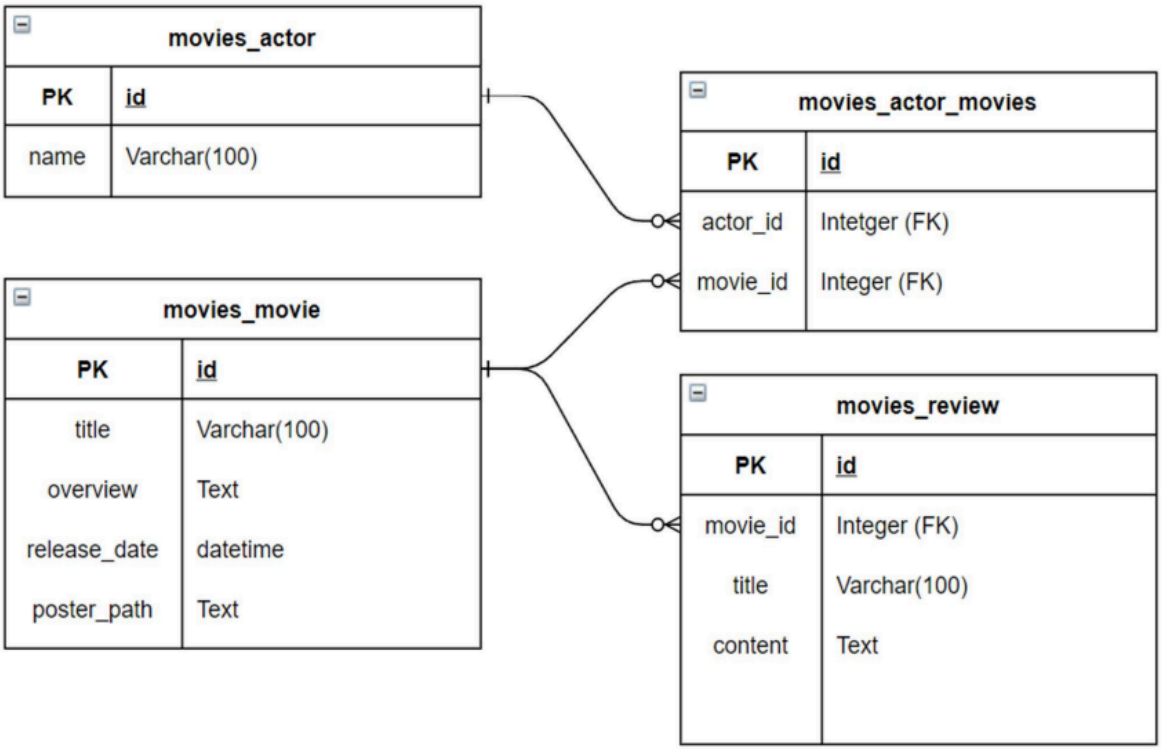
지금까지는 클라이언트에서 요청에 대한 결과를 Template으로 확인 했지만, 오늘부터는

클라이언트의 요청에 대한 결과를 DRF의 웹 인터페이스를 통해서 확인하고자

URL에 api/v1/ 를 추가 해 주었다. (DRF - Django Rest Framwork)

이제 `movies` 앱을 설정하자.

우리가 구현할 ERD 는 다음과 같은 형태이다.



- 영화관련 정보를 저장하고 movie 모델
- 영화출연진 정보를 저장하고 있는 actor 모델
- 영화리뷰를 저장하는 review 모델 이다.

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 신경써서 볼 부분은 ERD 에서 `movies_actor_movies` 는 중개테이블이고 `ManyToManyField` 를 사용해 (M:N)다대다 관계를 만들어 준 것이다.
그리고 "역참조"시 사용할 이름으로, `related_name='movies'` 라고 적어 주었다.

`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
```

그다음 테스트를 위해 준비한 샘플 파일은 아래와 같다.

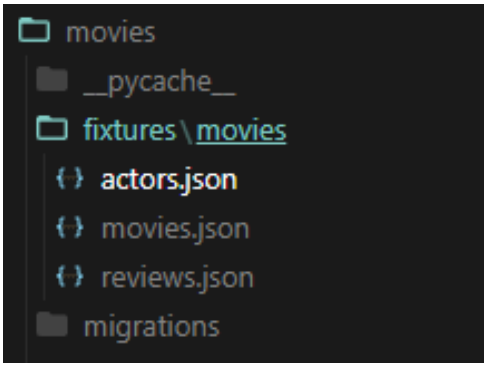
movies.json

actors.json

reviews.json

이후, 주어진 fixture 파일을 로드하자.

JSON 파일들은 movies/fixtures/movies 에 위치 시키자.




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

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

	* id INTEGER	* name varchar(100)
	Filter	Filter
	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.

	* id INTEGER	* movie_id bigint	* actor_id bigint
	Filter	Filter	Filter
	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 영화배우 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)

```
# movies/urls.py

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 필요한 함수만 최소한으로 작성해 놓았다.

3. Serializers

Serializer 란?

Django REST framework에서 사용하는 클래스이다. 이는 모델 인스턴스(객체)를 json 으로

변환해 주거나 json 데이터를 모델 인스턴스로 변환해주는 역할을 한다.

웹은 클라이언트와 서버의 요청과 응답의 연속이다. 이 과정 중 “직렬화”/”역직렬화”는 꼭 필요한 과정이다. 먼저 역직렬화(Deserialization) 가 무엇인지 살펴보고 직렬화 (Serialization) 이 무엇인지 살펴보자.

1. 클라이언트 → 서버로 데이터 요청

1. 예를들어, 사용자가 “회원가입” 폼 작성 후 전송버튼을 눌렀다.
2. 이 때 JSON형식 또는 문자열 형식으로 데이터가 서버로 보내진다.
3. 서버는 이 데이터를 받아서 → “역직렬화”를 해서 → 파이썬 객체로 변환하고 저장한다.

- **여기서 “역직렬화”가 발생한 것이다.**
- 클라이언트에서 JSON 형식으로 서버로 요청 → 서버에서는 JSON형식을 파이썬 객체로 변환한다. 이를 역직렬화 라고 한다.

2. 서버 → 클라이언트로 응답

1. 예를들어, 서버가 DB에서 사용자 정보를 가져와서 응답을 할 차례다.
2. 서버는 파이썬 객체를 → JSON 형태로 변환해서 클라이언트에 전달한다.

- **여기서 직렬화 발생한다.**
- 서버에서 응답할 파이썬 객체를 → JSON 형식으로 변환한다. 이를 직렬화라고 한다.

직렬화(Serializer) 그리고 역직렬화(Deserialization)를 알았으면 이제 DRF가 무엇인지 살펴보자.

4. DRF (Django Rest Framework)

DRF는 Django를 기반으로 백엔드 API를 만들 때

개발자가 쉽고 빠르게 RESTful API를 만들 수 있도록 도와주는 도구를 말한다.

[참고] RESTful API 란?

GET, POST, PUT, DELETE와 같은 HTTP 메서드를 활용하여

URL과 메서드만 보고도 어떤 동작인지 유추할 수 있도록

구조를 깔끔하고 일관성 있게 설계한 API를 말한다.

- restful한 api의 예시 :
  - `GET /users` → 사용자 목록 조회
  - `GET /posts/1` → 1번 게시글 조회
  - `POST /products/123/reviews` → 123번 상품에 리뷰 작성

- restful하지 않은 api의 예시:
  - `GET /getAllUsers` → 사용자 목록 조회
  - `POST /createPost?id=1` → 1번 게시글 조회
  - `POST /addReviewToProduct?product_id=123` → 123번 상품에 리뷰 작성

따라서 RESTful API 란?
URL과 HTTP 메서드를 체계적으로 잘 사용하는 API 스타일로
우리가 지금까지 django를 학습하면서 사용했던 url을 만드는 스타일을 말한다.

DRF(Django REST Framework)는 개발자가 쉽고 빠르게 RESTful API를 만들 수 있도록 도와주는 강력한 도구라고 했다.

DRF가 하는 역할은 무엇이 있을까?

1. `Serializer`**를 이용한 직렬화/역직렬화 처리**

   → 요청과 응답에서 사용되는 데이터를 JSON ↔ Python 객체로 변환해주며, API 통신의 핵심 역할을 한다.

2. **인증(Authentication)과 권한(Authorization) 처리**

   → 로그인한 사용자만 접근 가능하게 하거나, 관리자만 특정 기능을 사용할 수 있도록 다양한 인증/권한 기능을 쉽게 설정할 수 있다.

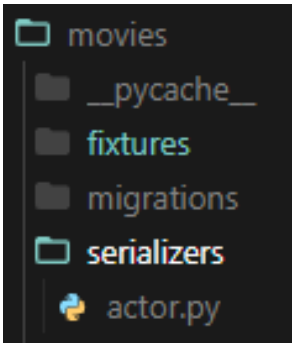
3. **유효성 검사 및 에러 처리**

   → 클라이언트가 보낸 데이터가 올바른지 자동으로 검사하고, 문제 있을 경우 적절한 에러 메시지를 반환해준다.

자, 이제 DRF 그리고 Serializer가 무엇인지 이해가 되었다면

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

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




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

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

# 전체 영화배우 목록
class ActorListSerializer(serializers.ModelSerializer):
    class Meta:
        model = Actor
        fields = '__all__'


# 단일배우 정보제공 (출연한 영화 제목포함)
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 코드는 DRF에서 배우 정보와 출연 영화 정보를 어떻게 보여 줄 것인지 정의를 해 놓은 것이다.

먼저, 전체 위 코드의 전체 구조를 보자.

구조를 보면 두 가지 Serializer 클래스가 있다. (`ActorListSerializer` / `ActorSerializer`)

1. `ActorListSerializer`
    - 이 클래스는 모든 배우의 데이터 목록을 가져오기 위해 정의를 해 주었다.
    - 이 클래스는 `Actor`라는 모델의 정보를 그대로 JSON으로 바꿔주는 역할을 한다. (직렬화)
    - `fields = '__all__'` 이라고 되어 있어서, `Actor` 모델의 모든 필드를 포함해서 응답해줄 것이다.

2. `ActorSerializer`
    - 이 클래스는 특정 단일 배우의 데이터를 가져오고, 그 배우가 출연한 영화 제목을 포함 시킬 것이다.
    - 특히 ActorSerializer 안에 들어있는 MovieSerializer 를 보자.

```
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__'
```

MovieSerializer 코드를 통해서 특정 단일 배우가 출연한 영화들 중, 영화 제목만 보여주겠다고 설정한 것이다.

- `movies = MovieSerializer(many=True, read_only=True)` 이 부분을 보자.
  - 여기서 필드가 될 `movies` 는 방금전에 정의한 `MovieSerializer` 의 결과이며, 결과는 단일 데이터가 아닐 수 있기 때문에 `many=True` 를 작성해 주었고, 수정이 불가능 하기에 `read_only=True` 를 적어주었다.
  - 즉, 여러 개의 데이터를 조회한 결과가 `movies` 가 될 것이다. 다시말해, `movies` 필드는 한 명의 배우가 어떤 영화들에 출연을 했는지 보여 줄 것이다.
- class Meta:
  model = Actor
  fields = '__all__'
  - `ActorSerializer` 의 모델은 `Actor` 로 가지며, 모든 필드를 다 가져와 사용한다고 정의해 주었다.

그리고 Serializer 클래스를 정의할 때 눈여겨 볼 점이 있다.

~ListSerializer는 “모든 데이터” 를 다 가져오기 위해서 사용했으며

그리고 ~Serializer는 “하나의 단일 데이터” 를 가져오기 위해서 사용 되었다.

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

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

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

# 전체 영화 목록
class MovieListSerializer(serializers.ModelSerializer):
    class Meta:
        model = Movie
        fields = ('title', 'overview',)

# 단일 영화 정보 (출연배우 그리고 리뷰목록 포함)
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 정의를 해 두었다가,

`actors` 와 `review_set` 필드를 만드는 데 사용한다.

즉, `actors` 는 하나의 영화에 어떤 배우들이 있고, `review_set` 를 통해서 하나의 영화에 어떤 리뷰들이 있는지 나타내 줄 것이다.

배우는 배우 이름(`name`), 리뷰는 리뷰 제목(`title`)과 내용(`content`)이 될 것이다.

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

review를 위한 serializer를 정의해 보자.

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

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

# 전체 리뷰 목록
class ReviewListSerializer(serializers.ModelSerializer):
```

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

# 단일 리뷰 정보 (해당되는 영화의 제목 포함)
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 정의해서 사용하는데,
  - MovieSerializer의 기본 모델은 `Movie` 이고, `title` 필드를 가져온다.
  - `MovieSerializer` 의 결과는 `movie` 필드가 되는데, 여기에서는 `many=true` 가 사용되지 않았다. 그 이유는 하나의 리뷰는 오직 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 구현 해보자. 우선, 위에서 만든 `urls.py` 표를 참고해서

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.

   1-1 **actor_list, movie_list, review_list**

2. 다음, 단일 데이터를 가져오는 것들 구현한다.

   2-1 **actor_detail, movie_detail, review_detail**

3. 리뷰를 생성하는 것을 구현한다.

   3-1 **create_review**

4. 리뷰를 수정하고 삭제하는 것을 구현한다.

   4-1 **review_detail**

먼저, `actor_list` 함수이다.

`actor_list` 함수는 데이터베이스의 전체 배우 목록을 제공 해주는 함수이다.

```
# views.py/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 미리 만든 Serializer 들을 모두 `import` 해두었다.

`actor_list` 함수로 요청이 들어오면

1. 먼저 DB의 Actor 모델에서 데이터를 가져온다. 가져온 내용은 `actors` 이며, 모든 배우들 목록이

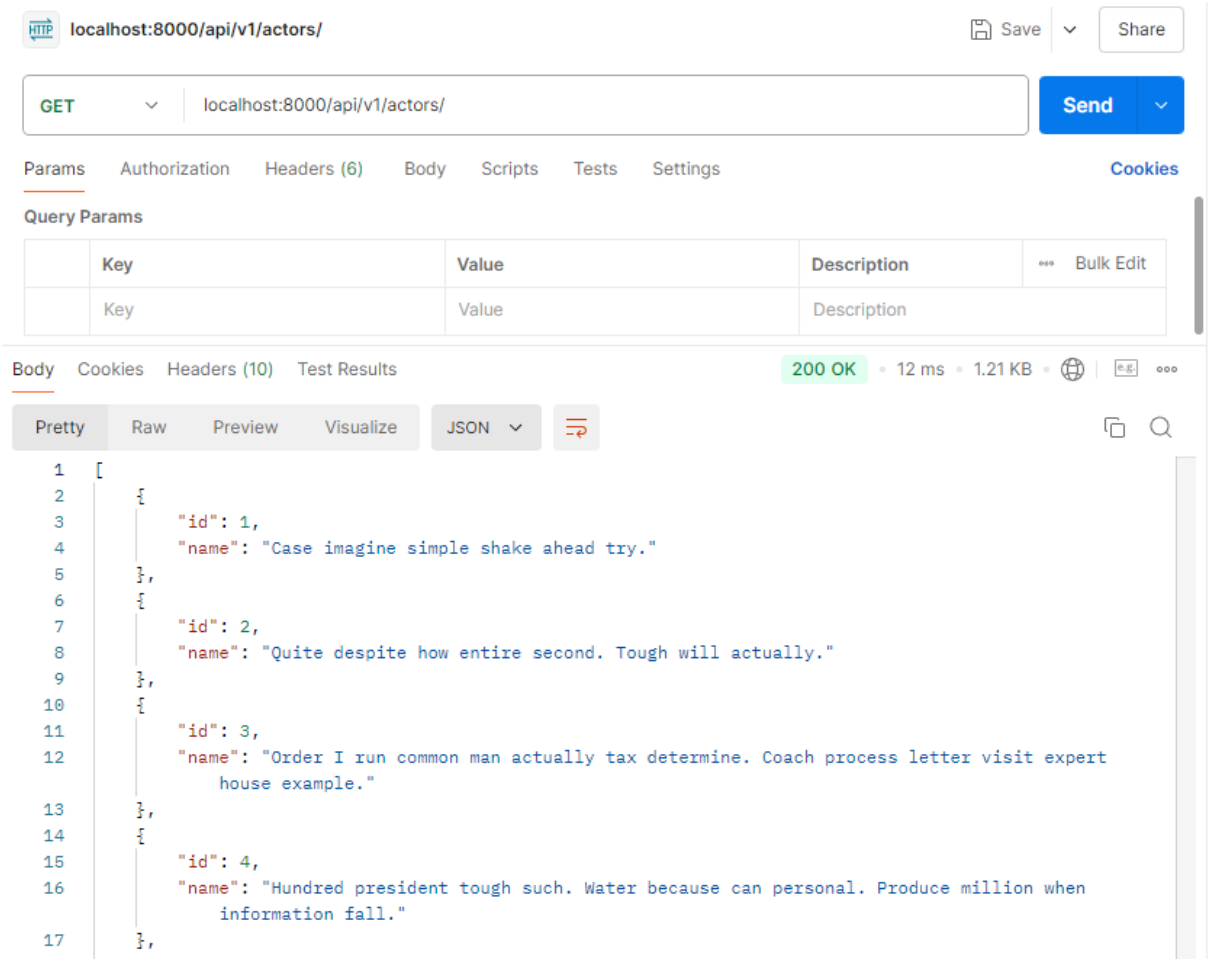
   쿼리문으로 리턴되며 실패할 경우, 404 에러를 리턴 할 것이다.
2. 우리가 만들어둔 `ActorListSerializer` 는 쿼리문을 JSON 형식으로 바꾸는 역할을 할 것이다.

   첫번째 인자로 `actors` 를 넣고, 단일 데이터가 아니라 여러 데이터를 가져와야 하기에 `many=True` 를 두번째 인자로 넣어 `serializer` 를 리턴 받는다.
3. DRF 에서 제공하는 `Response` 객체를 이용해 `serializer` 객체 안에 `data` 만 리턴 한다.

결과 확인해보자.

migrations 그리고 migrate를 하고 서버를 켜자.

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



랜덤한 문자열로 배우이름을 만들어서.. 이게 이름인가? 싶겠지만..ㅎ DB를 확인해 보자.

배우 목록이 위와 같은 형태로 잘 들어옴을 알 수 있다.

이 원리 대로, `movie_list` 함수도 똑같은 형태로 구현 가능하다.

`movie_list` 함수는 데이터베이스의 전체 영화 목록을 제공 해주는 함수이다.

```
# 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`함수는 데이터베이스의 전체 리뷰들을 제공 해주는 함수이다.

```
# 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 각각 테스트해보자.

`localhost:8000/api/v1/movies/`

`localhost:8000/api/v1/reviews/`

Postman으로 응답을 확인 하였다면,

이번에는 단일 데이터를 가져오는 것들을 구현해야 한다.

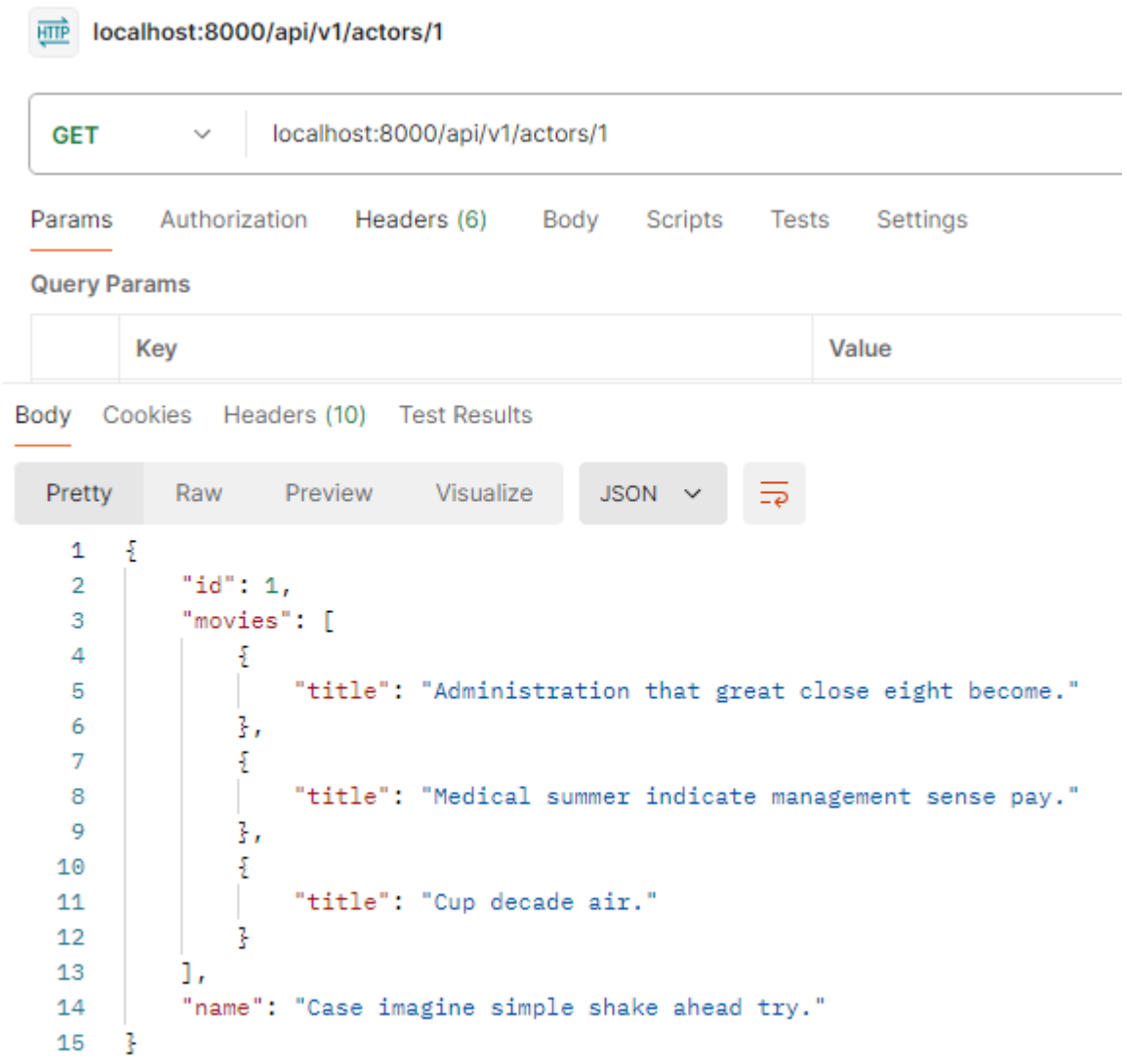
먼저, `actor_detail` 이다. `actor_detail` 함수는 단일 영화배우의 정보를 가져오는 함수이다.

```
# 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. 우리는 단일 데이터를 가져오기 때문이다.

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



해당 배우의 이름 (`name`) 뿐만 아니라, 어떤 영화들에 출연했는지 `title` 까지 볼 수 있는데, 우리가 `ActorSerializer` 를 구현할 때 해당 필드를 정의해두었기 때문이다.

자, 이번에는 `movie_detail` , `review_detail` 도 구현해보자.

```
# movie_detail

@api_view(['GET'])
def movie_detail(request, movie_pk):
    movie = get_object_or_404(Movie, pk=movie_pk)
```

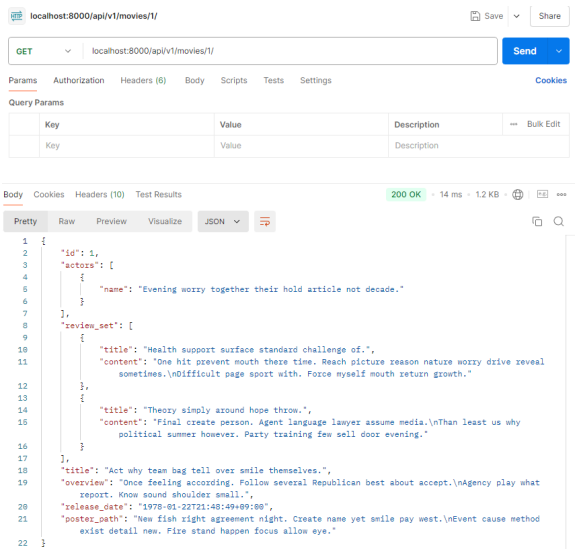
```
    serializer = MovieSerializer(movie)
    return Response(serializer.data)
```

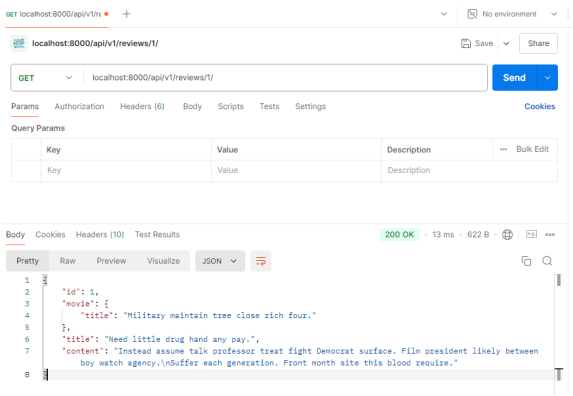
```
# review_detail

@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.

`localhost:8000/api/v1/movies/1/`

`localhost:8000/api/v1/review/1/`





잘 된다.

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)
```

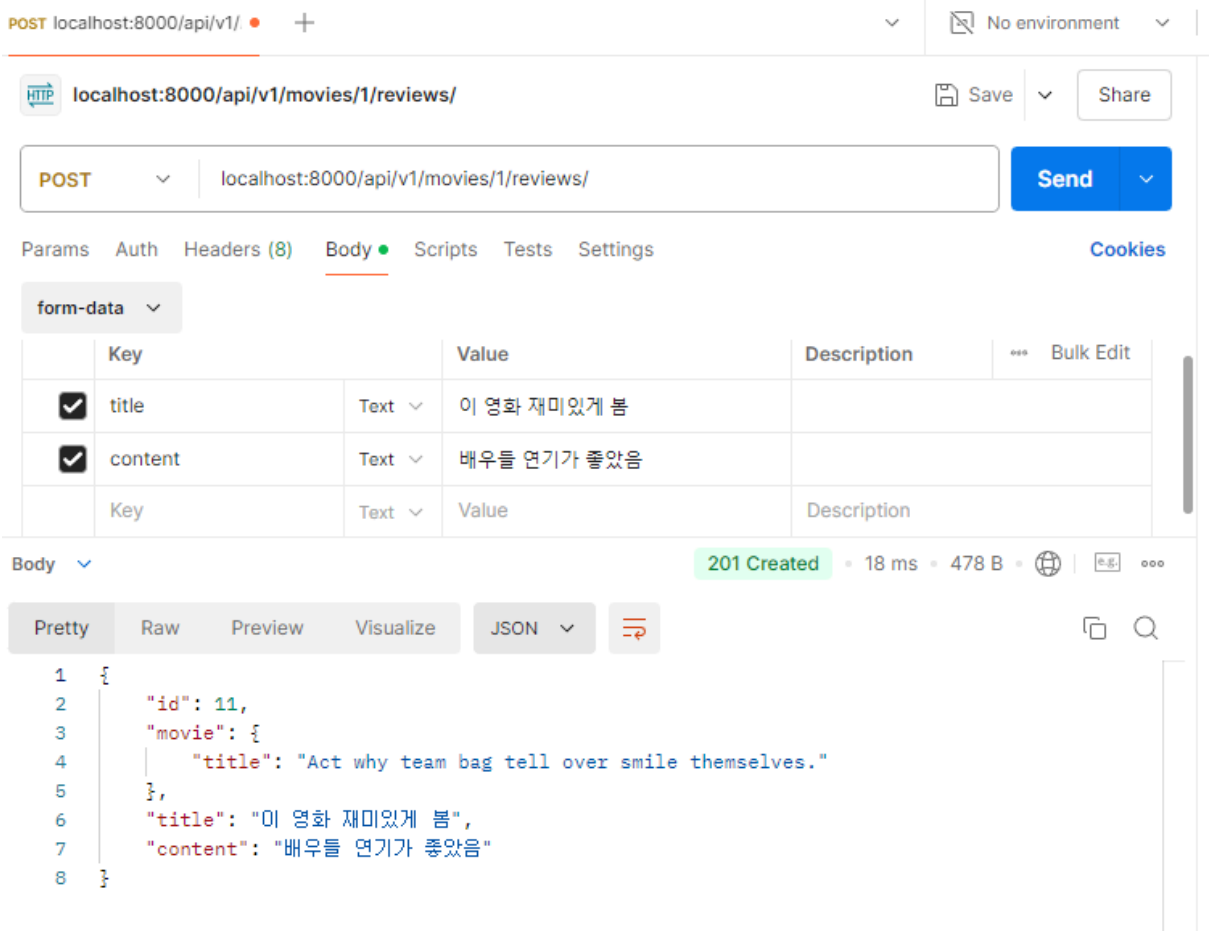
일단, 어떤 영화에 달아야 할 리뷰일지 알아야 하기 때문에, 해당 영화정보를 가져와야 한다.

그리고, 사용자가 JSON 으로 작성한 정보들을 `ReviewSerializer` 의 `data=request.data` 인자를 사용해 입력하고, 유효성 검사를 해준다.

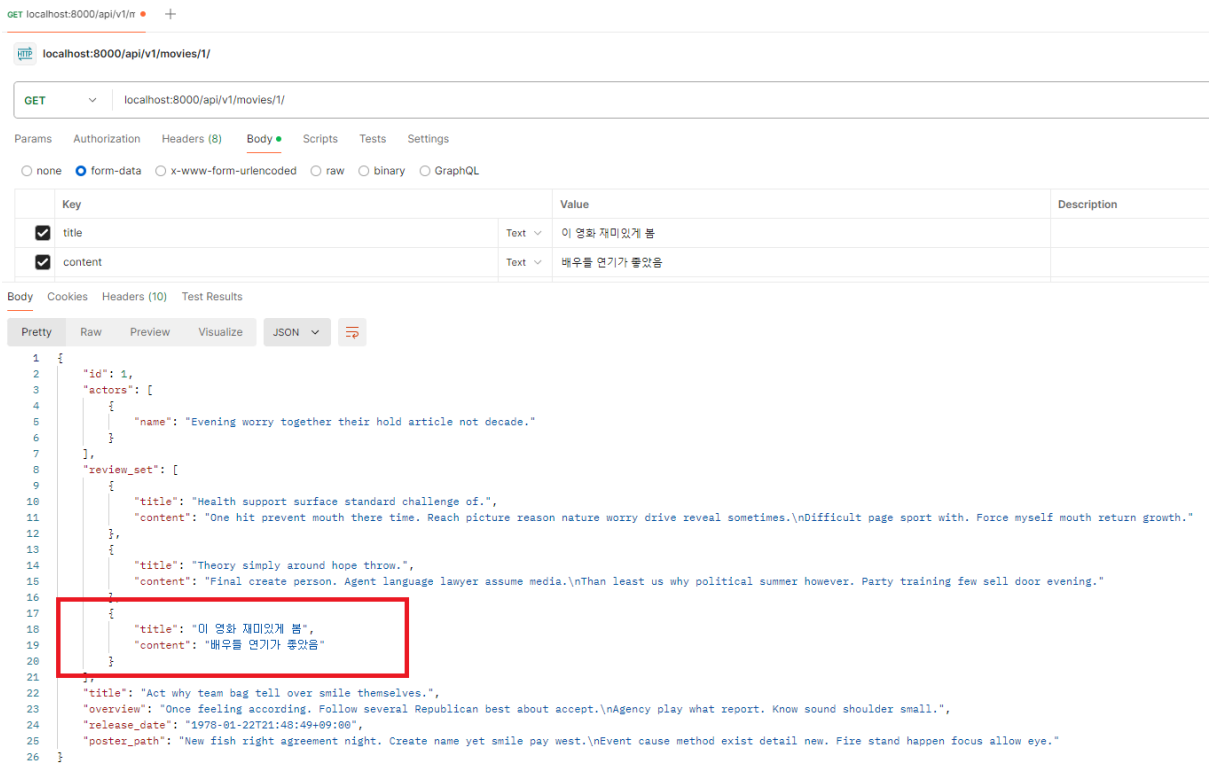
`raise_exception=True` 로 설정 할 경우, 클라이언트는 어떤 에러가 발생했는지 자세하게 볼 수 있다.

유효성 검사를 통과하면, `ReviewSerializer` 에 우리가 정의한 `movie` 필드에, 우리가 리뷰를 작성하고자 하는 영화 정보를 넣어주고, `save` 해주며, 추가 성공한 리뷰 내용과 함께 201 상태 메시지(자원 생성 성공) 를(을) 사용자에게 리턴을 해준다.

Postman 으로 테스트해보자.



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



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

다음, 리뷰 삭제다. `review_detail` 함수에서 조회, 수정, 삭제를 모두 담당하면 되므로 조건문을 사용해 분기를 해줘야 한다. (나눠줘야 한다.) 수정해보자.

```
# views.py/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)
```

단일 리뷰를 조회를 하든 수정 또는 삭제 기능을 구현하려면 모두 해당 리뷰를 가져와서 활용한다.

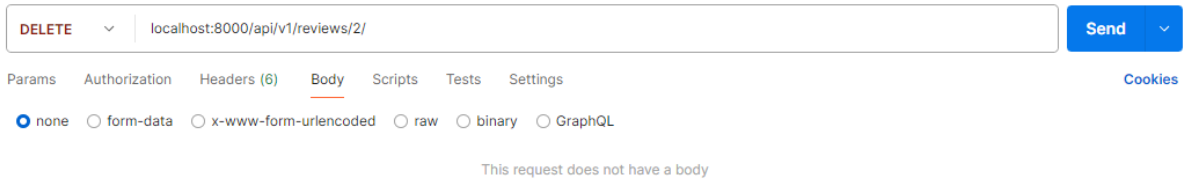
`elif request.method == 'DELETE'` 부분을 보면, 해당 리뷰를 지워버리고,

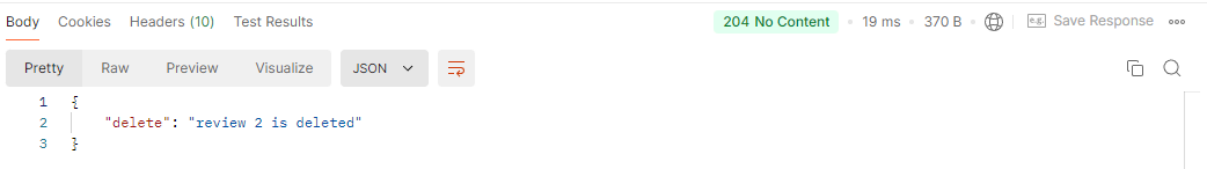
사용자에게 간단한 메세지를 보내며, 삭제 완료를 뜻하는 204 상태 메시지를 보냈다.

Postman 으로 테스트해보자.

단, Postman에서 맨 뒤에 슬래시(`/`) 를 넣지 않으면 무조건 GET 으로 받아들이므로 주의하자.

DELETE method 를 이용해서 `localhost:8000/api/v1/review/1/` 요청으로 확인해보자.





review 2 is deleted 라는 메시지 까지 잘 확인이 된다.

마지막으로, 수정을 구현해보자. Detail 함수에 PUT 파트를 넣어주자.

```
@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, partial=True)
        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 인자**로 **수정할 대상**을 넣고, **두번째 인자**로 사용자가 작성한 **수정 내역**을 넣는다.
- `partial=True` 옵션은 DRF에서 직렬화할 때 부분적인 업데이트를 허용한다는 옵션이다.

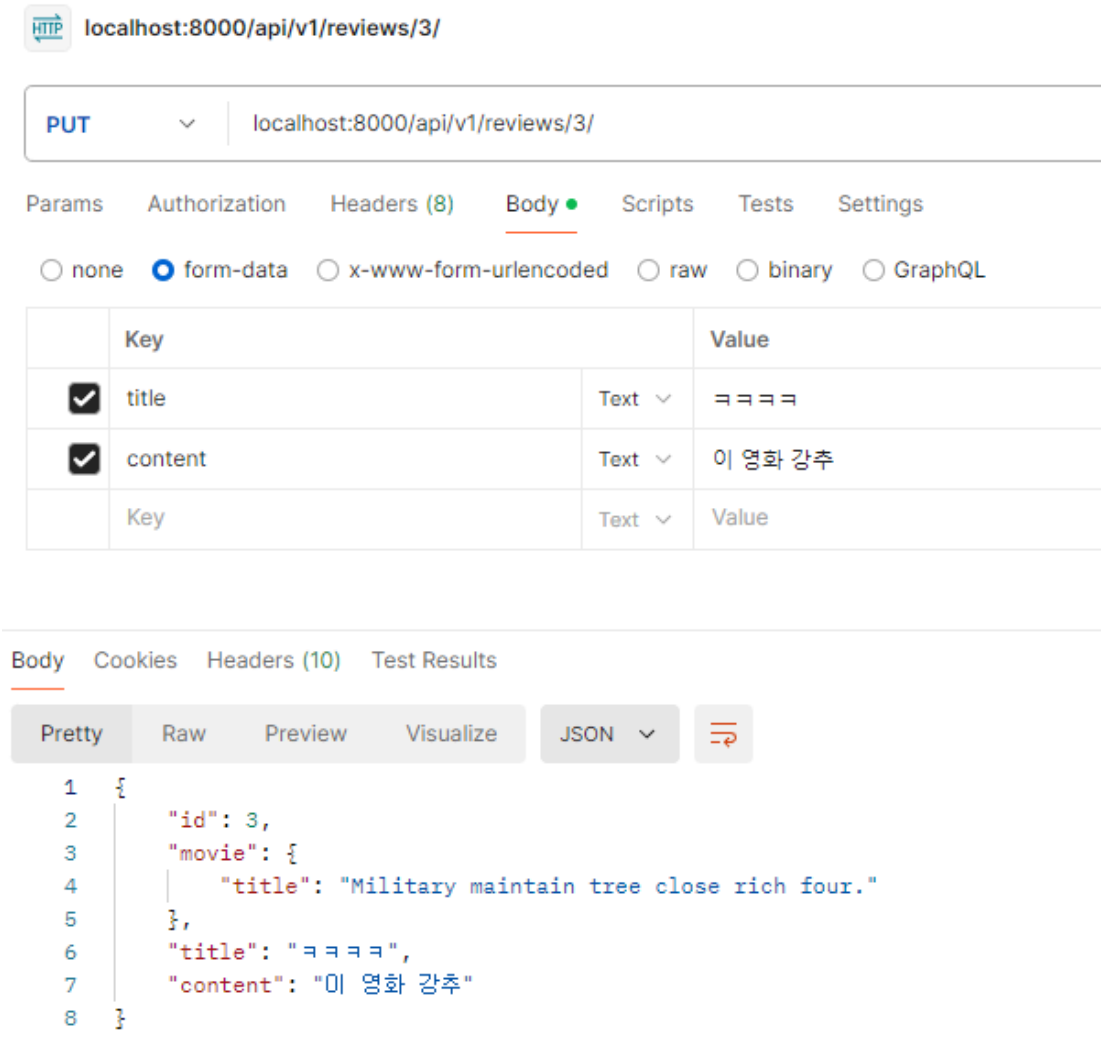
  예를들어, 회원정보 수정시 프로필 사진만 바꾸고 싶다면 프로필 사진 데이터만

  업데이트 하면 되는데 `partial=True`를 사용하지 않으면, 수정하고 싶지 않은 데이터를 포함하여

  모든 필드를 새로 작성을 해야 한다. 그렇지 않으면 유효성 검사에서 실패를 하기 때문이다.

Postman 으로 테스트해보자.

리뷰 3번 수정을 위해 다음과 같이 보냈고,



리턴값도 잘 받았다. `GET /reviews/3` 내용이 실제로 바뀌었는지 DataBase를 확인하며 마무리 하자.

<끝>

DRF_test.zip