Django advanced project #week_3

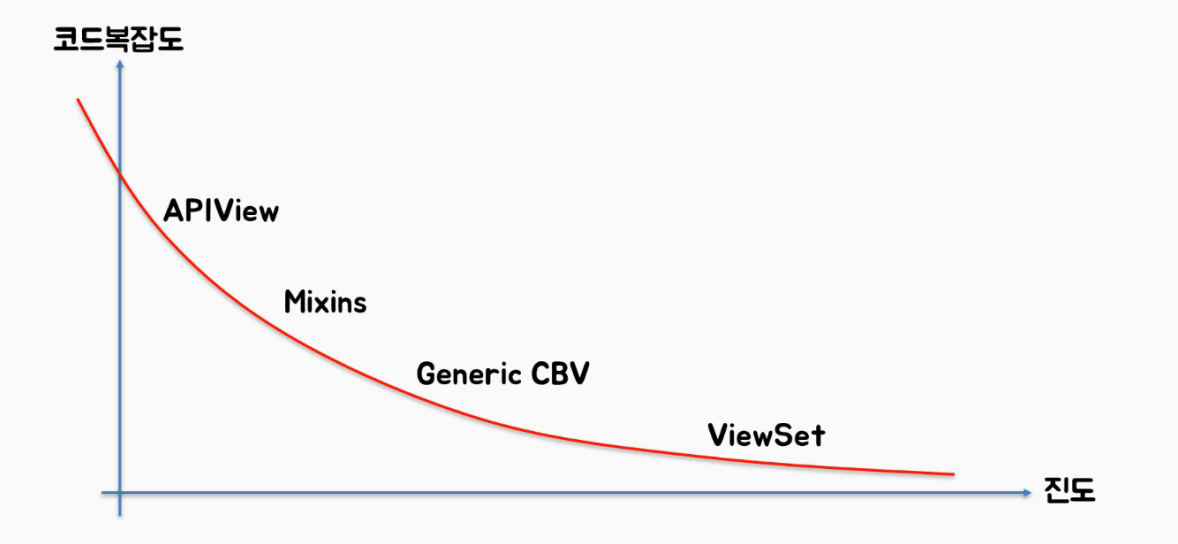
Writer : Taeyeong Kim, DGIST LikeLion

Date: 2019.10.09

아래 자료를 공부하기 전에, https://github.com/lizard-kim/LikeLion_study_Fall_Semester 에 있는 코드를 clone하여 참고하기 바란다. 코드에 대한 자세한 정보는 주석을 통해 알 수 있으니, 이곳에서는 대략적인 설명만 할 것이다.

Views.py 를 더 간단하게!

이전강의에서 ViewSet을 통해 view를 간단하게 구현해보았다. 그밖에도 view를 설계하는 방법은 많은데 종류는 아래 그림과 같다.



> 4가지를 차차 알아갈 것이다.

APIView

views.py

```
# 데이터 처리 대상
from post.models import Post
from post.serializer import PostSerializer
# status에 따라 직접 Response를 처리할 것
from django.http import Http404
from rest_framework.response import Response
from rest_framework import status
# APIView를 상속받은 CBV
from rest_framework.views import APIView
# PostDetail 클래스의 get_object 메소드 대신 이거 써도 된다
from django.shortcuts import get_object_or_404

'''
class balbal(APIView):
```

```
    def 내가_필요로_하는_HTTP_Method: # post get 등의 http method 를 customizing 할 수
있다.
        define...
        pass
'''

class PostList(APIView):
    def get(self, request):
        posts = Post.objects.all()
        serializer = PostSerializer(posts, many=True) # 직렬화 해서 쿼리셋 넘기기
(many=True인자) 다수의 쿼리셋의 경우 many=True 가 필요하다
        return Response(serializer.data) # 직접 Response 리턴해주기 :
serializer.data

    def post(self, request):
        serializer = PostSerializer(data=request.data)
        if serializer.is_valid():   # 직접 유효성 검사
            serializer.save()       # 저장
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    '''
    def delete(self, request):
        pass
    def put(self, request):
        pass
    '''

# PostList 클래스와는 달리 pk값을 받음 (메소드에 pk인자)
class PostDetail(APIView):

    # get_object_or_404를 구현해주는 helper function
    def get_object(self, pk):
        try:
            return Post.objects.get(pk=pk)
        except Post.DoesNotExist:
            raise Http404

    def get(self, request, pk, format=None):
        post = self.get_object(pk)
        # post = get_object_or_404(Post, pk)
        serializer = PostSerializer(post)
        return Response(serializer.data)

    # 위 post 메소드와 비슷비슷한 논리
    def put(self, request, pk, format=None):
        post = self.get_object(pk)
        serializer = PostSerializer(post, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk, format=None):
        post = self.get_object(pk)
        post.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

APIView를 통해서 HTTP method를 직접 구현한 것을 볼 수 있다(심지어 get_object_404도 직접 구현했다). 이때 각 url마다 사용하는 method가 다르니, 분리해서 class를 생성하고, 그에 맞는 함수만 구현해 놓은 것을 볼 수 있다.

간단한 코드라서 이해하기는 어렵지 않을 것이다. 하지만, 뭔가 반복되는 코드가 보이지 않는가? 그렇다. get, post method가 똑같이 쓰이는 것을 볼 수있다. 만약 APIView를 이용한다면, 다른 class를 만들 때에도 다른 class에서 일반적으로 사용하는 method를 구현해주어야 할 것이다. 이를 방지하기 위해 mixins를 활용할 수 있다.

Mixins

mixins_views.py

```
# 데이터 처리 대상 : 모델, Serializer import 시키기
from post.models import Post
from post.serializer import PostSerializer

from rest_framework import generics
from rest_framework import mixins

# mixin 직접 보기 : https://github.com/encode/django-rest-
framework/blob/master/rest_framework/mixins.py
# genericAPIView 직접 보기 : https://github.com/encode/django-rest-
framework/blob/master/rest_framework/generics.py

class PostList(mixins.ListModelMixin, mixins.CreateModelMixin,
               generics.GenericAPIView):
    queryset = Post.objects.all()   # 쿼리셋 등록!
    serializer_class = PostSerializer # Serializer 클래스 등록!

    # get은 list메소드를 내보내는 메소드
    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    # post는 create을 내보내는 메소드
    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

class PostDetail(mixins.RetrieveModelMixin, mixins.UpdateModelMixin,
               mixins.DestroyModelMixin, generics.GenericAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

    # DetailView의 get은 retrieve을 내보내는 메소드
    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    # put은 update을 내보내는 메소드
    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    # delete은 destroy를 내보내는 메소드
    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

코드가 더 간결해졌다. queryset으로 어떤 model을 보낼지 정하고, get, post와 같은 method의 리턴값만 지정해주었다. 실제로 주석에 달려있는 *https://github.com/encode/django-rest-framework/blob/master/rest_framework/mixins.py* 에 접속하여 상속한 class를 살펴볼 수 있다. 자세히 읽어보면, 인자로 queryset과 serializer_class를 받아서 get, post와 같은 method를 구현해 놓은 것을 볼 수 있다.

그러나 아직도 같은 method를 중복해서 사용하는 것을 확인할 수 있는데, 이마저도 사라지게 만드는 것이 바로 generic CBV이다.

generic CBV

generic_views.py

```
from snippets.models import Post
from snippets.serializers import PostSerializer
from rest_framework import generics

# rest_framework/generics.py
# https://github.com/encode/django-rest-framework/blob/master/rest_framework/generics.py

# ListCreateAPIView
# https://github.com/encode/django-rest-framework/blob/0e1c5d313232a131bb4a1a414abf921744ab40e0/rest_framework/generics.py#L232

# RetrieveUpdateDestroyAPIView
# https://github.com/encode/django-rest-framework/blob/0e1c5d313232a131bb4a1a414abf921744ab40e0/rest_framework/generics.py#L274

class PostList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer


class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
```

코드가 더욱 간결해졌다. mixins과 마찬가지로 generics에 구현되어있는 class를 상속하여 queryset과 serializer_class만 넘겨주면 완성이다. 주석에 달려있는 링크로 접속해보면, 우리가 상속한 class가 어떻게 구현되어 있는지 볼 수 있다. 직접 코드를 읽어보면서 어떻게 동작하는지 확인해보기 바란다.

ViewSet

router_views.py

```
from post.models import Post
from post.serializer import PostSerializer

from rest_framework import viewsets

# @action처리
from rest_framework import renderers
```

```
from rest_framework.decorators import action
from django.http import HttpResponse

'''
# ReadOnlyModelViewSet은 말 그대로 ListView, DetailView의 조회만 가능
class PostViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
'''
# ModelViewSet은 ListView와 DetailView에 대한 CRUD가 모두 가능

class PostViewSet(viewsets.ModelViewSet):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

    # @action(method=['post'])
    @action(detail=True, renderer_classes=[renderers.StaticHTMLRenderer])
    # 그냥 얍을 띄우는 custom api
    def highlight(self, request, *args, **kwargs):
        return HttpResponse("얍")
```

코드가 조금 더 간결해졌다. 우리는 ModelViewSet을 통해 ListView와 DetailView에 대한 CRUD가 모두 가능하도록 이렇게 묶을 수 있다. 더 나아가, 새로운 custom api를 쉽게 만들 수 있다(CRUD를 넘어서 다른 logic까지 구현할 수 있다). runserver를 통해 확인해보자.

References

likelion 강의 소스코드 : https://github.com/kangtegong/django-RESTful-API3