




8주차 DRF (1)



서강대 김동윤



목차

1. Django REST Framework?
2. Serializer
3. View

DRF ?

REST란?

Representational State Transfer의 약자로
'웹 상의 자료를 HTTP 위에서 별도의 전송계층 없이 전송하기 위한 아주 간단한 인터페이스'
필딩의 REST 원리를 따르는 시스템을 RESTful이란 용어로 지칭

저희는 앞으로 RESTful한 API를 만들어보려고 하는데요,
이를 위한 주가 되는 규칙에는 크게 두 가지가 있습니다!

1. URI : 정보의 자원을 표현 (리소스명은 동사보다는 명사를 사용)
2. 자원에 대한 행위는 HTTP Method(GET, POST, PUT, DELETE 등)로 표현

HTTP Method ?

METHOD	역할
POST	POST를 통해 해당 URI를 요청하면 리소스를 생성해요!
GET	GET를 통해 해당 리소스를 조회합니다. 리소스를 조회하고 해당 도큐먼트에 대한 자세한 정보를 가져와요!
PUT	PUT를 통해 해당 리소스를 수정해요!
DELETE	DELETE를 통해 리소스를 삭제해요!

GET /members/delete/1 [X]

DELETE /members/1 [O]

회원정보를 가져오는 URI

GET /members/show/1 [X]

GET /members/1 [O]

회원을 추가할 때

GET /members/insert/2 [X]

POST /members/2 [O]

장고에서 이러한 웹 API를 RESTful하게 만들 수 있도록
도와주는 아이가
바로 Django REST Framework,
줄여서 DRF입니다!

그렇다면 어떻게 DRF를 사용할까요?

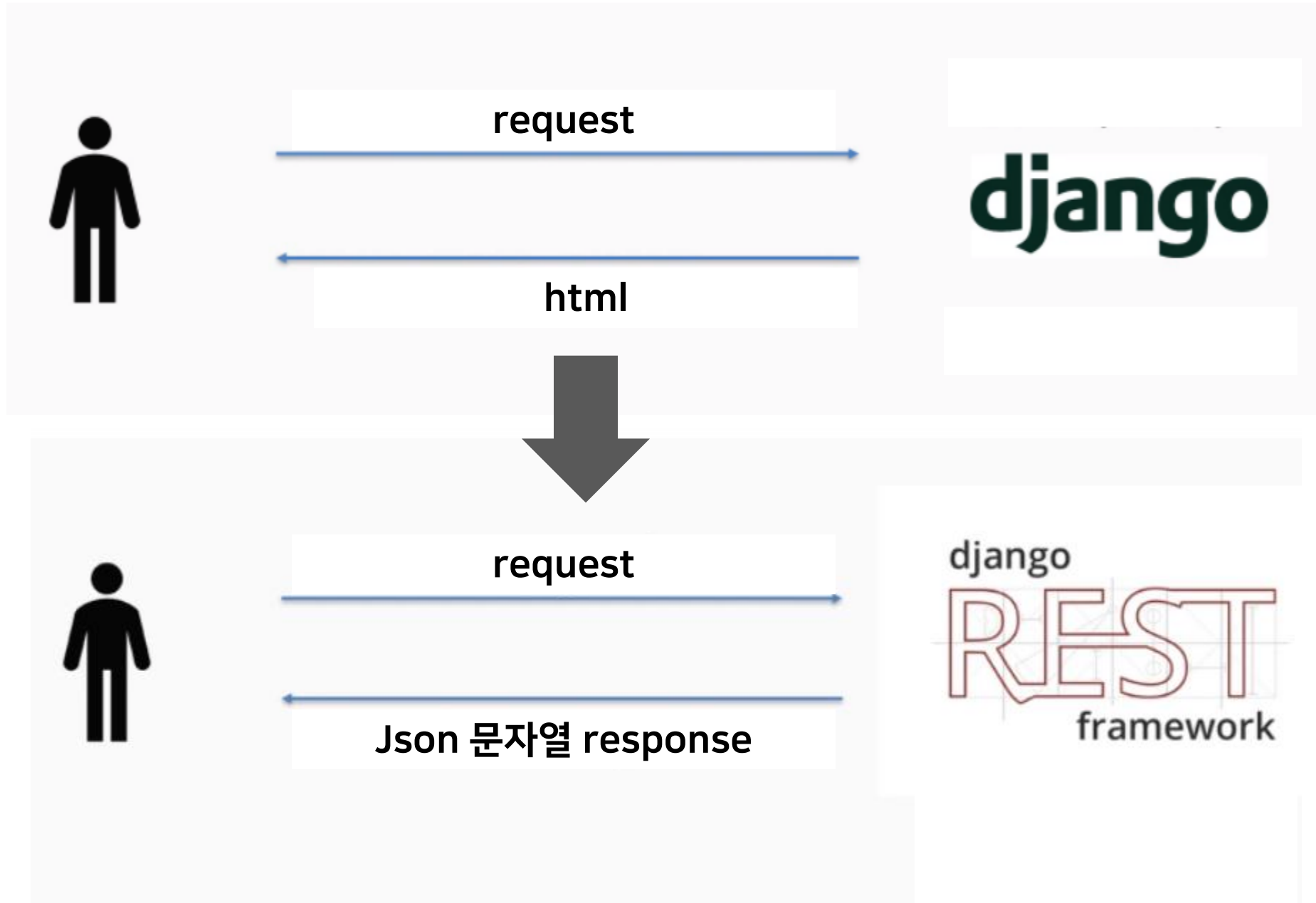
1.명령어를 통해 설치해주세요

```
pip install djangorestframework
```

2.settings.py의 INSTALLED_APPS에
rest_framework와 내가 만든 앱을 추가해주세요

```
INSTALLED_APPS = [ ... 'rest_framework ' , 'app 이름', ]
```

이 두 가지만 먼저 해주시면
바로 DRF를 이용하실 수 있습니다!



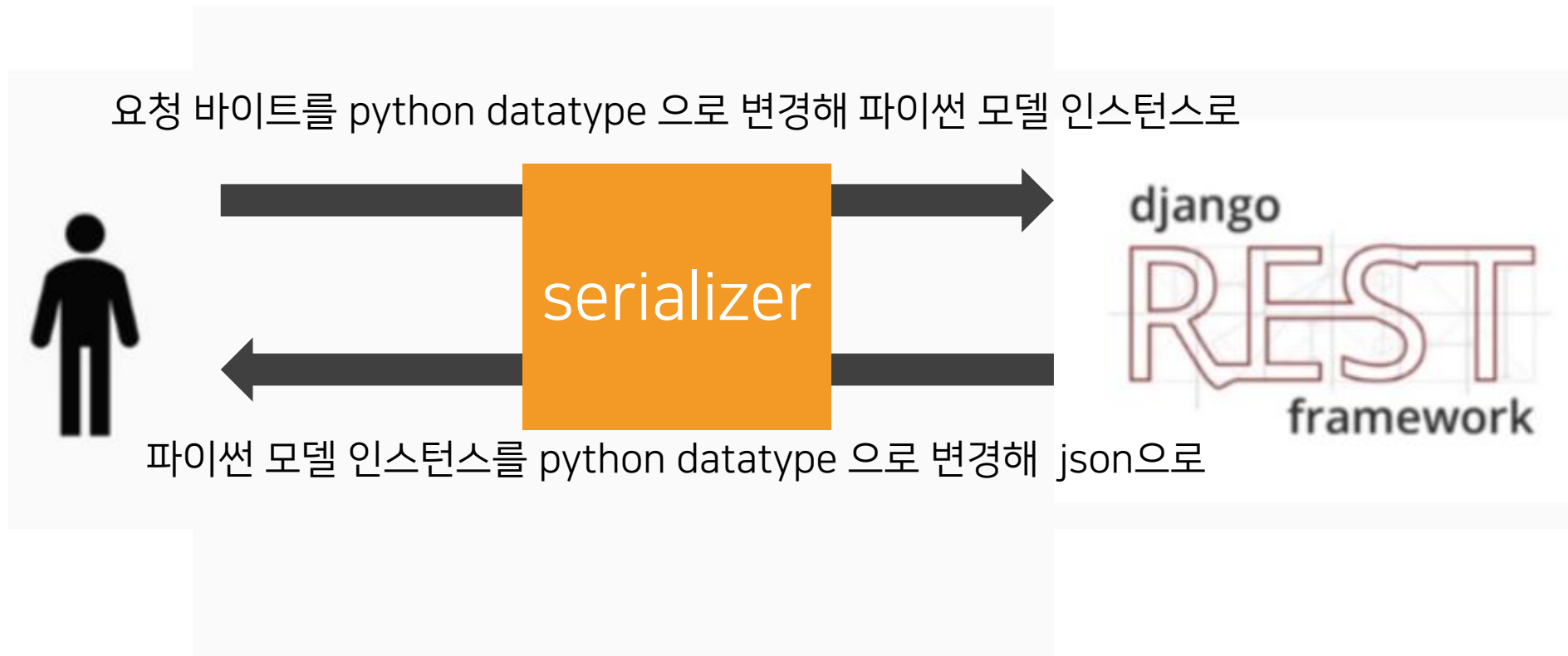
Serializer

Django REST Framework?

Client 개발자 ↔ Server 개발자
API 서버는 데이터를 제공하는 서버로 이해할 수 있는데요.
마치 식당[=Service]에서 음식[=Data]을
서빙해주는 서버[=Server]와 같습니다.

단, Client 개발자(프론트엔드)와 Server 개발자(백엔드)가 주고받는 데이터는
문자열로 표현되어야 하므로
객체는 직렬화 Serialization와 비직렬화 Deserialization
과정을 거쳐야만 합니다.

Django 패러다임 아래서 이러한 API 서버를 보다 빠르게 제작할 수 있습니다.



왜 굳이 직렬화로 데이터 주고받아?

데이터를 클라이언트와 서버가 송수신 한다는 것은
그 안에 유의미한 정보를 담고 있을 때 유효한 결과를 가지게 됩니다.

서버가 데이터를 저장하고 있는 방식은 크게 두가지로 나뉘 수 있는데, 다음과 같습니다.

1. 값 형식 데이터

Int, float, char, string 등

2. 레퍼런스 형식 데이터 (객체)

메모리 번지를 저장해두고, 해당 번지에 비로소 값이 존재 (포인터 개념)

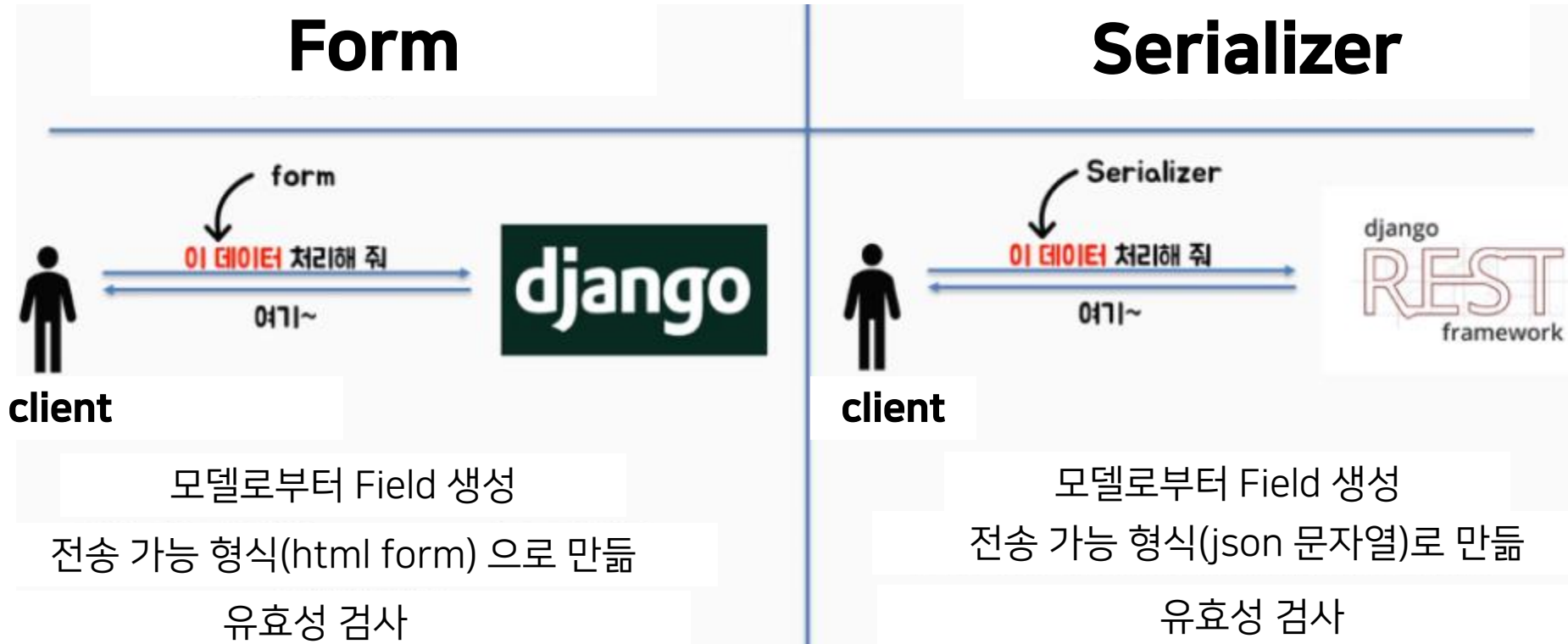
왜 굳이 직렬화로 데이터 주고받아?

여기서 만약 JSON의 value 값이 레퍼런스 형식의 데이터(ex. 또다른 객체) 이고,
이를 그대로 클라이언트가 넘겨 받게 된다면,
서버에서 해당 객체가 저장되었던 엄한 메모리 주소를 넘겨 받게 되고,
결국 이는 클라이언트가 식별할 수 없는, 무의미한 정보겠조?

따라서 직렬화가 필요해지는데, 이는 결국 주소값을 건네주지 않고
주소값의 실체를 다 끌어와서 값 형식 데이터로 모두 변조하는 작업을 수행하는 것입니다.

JSON 같은 경우는 직렬화 이후 텍스트로 된 데이터 모양을 띄게 됩니다.
따라서 클라이언트는 해당 정보를 받아 parse를 통해
서버에서 보낸 유의미한 정보를 받아볼 수 있게 됩니다.

Serializer?



django rest framework는 클라이언트의 요청에 대해 JSON 문자열을 돌려줌으로써 소통

이를 쉽게 가능하도록 해주는 것이
Serializer

Serializer?

Django	Django Rest Framework
Form/ModelForm	Serializer/ModelSerializer
Model로부터 Field 읽어옴	
유효성 검사	
HTML Form 생성	JSON 문자열 생성

django rest framework는 클라이언트의 요청에 대해 JSON
문자열을 돌려줌으로써 소통

이를 쉽게 가능하도록 해주는 것이
Serializer

Serializer?

1) queryset 및 모델 인스턴스와 같은 복잡한 데이터를
json, xml 또는 기타 content 유형으로 쉽게 렌더링 할 수 있는
네이티브 python datatype 으로 변환!

2) Serializer는 deserialization 또한 제공,
so 들어오는 데이터를 검증 후 다시 복잡한 데이터로 쉽게 변환 가능

DRF에서 제공하는 Serializer 클래스를 통해
모델 인스턴스와 queryset을 다루는 serializer를 쉽게 생성 가능 !

?queryset : 데이터베이스에서 전달 받은 객체의 목록 (Django ORM에서 발생한 자료형)

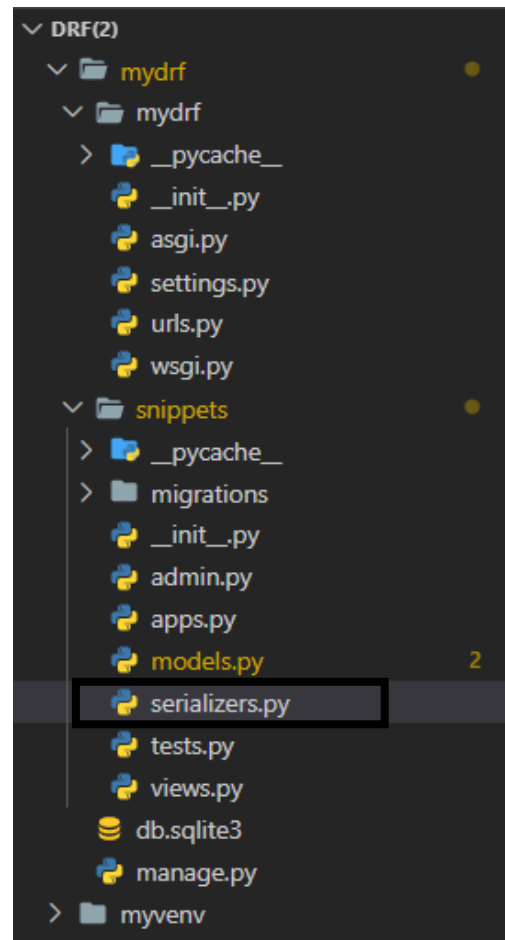

```
from django.db import models
from django.db import models
from pygments.lexers import get_all_lexers
from pygments.styles import get_all_styles
LEXERS = [item for item in get_all_lexers() if item[1]]
LANGUAGE_CHOICES = sorted([(item[1][0], item[0]) for item in LEXERS])
#pygment에서 제공하는 언어 종류들 (python, java 등)
STYLE_CHOICES = sorted([(item, item) for item in get_all_styles()])
#pygment에서 제공하는 코드 보여주기 옵션들
class Snippet(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=100, blank=True, default='')
    code = models.TextField()
    linenos = models.BooleanField(default=False) #행번호 보여줄 지 유무
    language = models.CharField(choices=LANGUAGE_CHOICES, default='python', max_length=100)
    style = models.CharField(choices=STYLE_CHOICES, default='friendly', max_length=100)

    class Meta:
        ordering = ['created']
#Meta클래스는 권한, 데이터베이스 이름, 단 복수 이름, 추상화, 순서 지정 등과 같은 모델에 대한 다양
한 사항을 정의하는 데 사용
```

우선 Snippet이라는 모델 클래스를 제작해줍니다.

```
python manage.py makemigrations  
python manage.py migrate
```

모델을 만들었으니 반영해줍시다!
그리고 이제 직접 serializer를 제작해볼거예요.
그리고 Snippets 아래에 폴더로 serializers.py 라는 파일을 하나 만들게요



```
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES

class SnippetSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    title = serializers.CharField(required=False, allow_blank=True, max_length=100)
    code = serializers.CharField(style={'base_template': 'textarea.html'})
    # Serializer 클래스에서 사용하는 {'base_template': 'textarea.html'} 플래그는
    # 장고 폼 클래스에서 widget=widgets.Textarea를 사용하는 것과 동일
    linenos = serializers.BooleanField(required=False)
    language = serializers.ChoiceField(choices=LANGUAGE_CHOICES, default='python')
    style = serializers.ChoiceField(choices=STYLE_CHOICES, default='friendly')
```

Serializer 클래스의 처음은
직렬화 또는 반 직렬화될 필드
(id, title,...)를 정의하는 것입니다!

우리가 처음 구경하는 serializer 클래스에 create()
메서드와 update() 메서드를 정의하였습니다.

이 메서드들은 serializer.save() 가 호출되었을 때,
인스턴스가 생성 또는 수정되는 방법을
정의하는 것입니다!

Save()가 수행되는 경우는 create, update겠죠?
이때 넘어온 매개변수의 개수로 create, update 메
소드 중 적절한 메소드를 수행시켜 serialize 하게
됩니다.

사실 이 복잡한 serializer은 바로 다음에 배울
ModelSerializer 로 아주 간단히 쓸 수 있으니
가볍게 이렇구나~ 정도만 봐주시면 됩니다!

```
def create(self, validated_data):
    # 새로운 인스턴스를 validated 데이터 (생성 데이터) 로 생성해서 돌려주기
    return Snippet.objects.create(**validated_data)

def update(self, instance, validated_data):
    # 현재 존재하는 객체 인스턴스의 데이터를 방금 들어온 후 시리얼라이즈되고 유효한
    validated(수정 데이터)로 대체!

    # instance = 수정대상이 되는 아이
    # validated_data.get(validated_data의 키, 인스턴스의 값(
    - validated_data에 키값이 존재하면 validated_data에서 키값에 해당하는 value 가져오기
    # 키 값이 존재하지 않으면 기존 수정대상의 아이의 속성값으로 채우기!

    instance.title = validated_data.get('title', instance.title)
    instance.code = validated_data.get('code', instance.code)
    instance.linenos = validated_data.get('linenos', instance.linenos)
    instance.language = validated_data.get('language', instance.language)
    instance.style = validated_data.get('style', instance.style)
    instance.save()
    return instance
```

그러면 Snippet이라는 모델 인스턴스를 생성한 후 json으로 render하는 과정을 한번 보도록 해봅시다!

Shell을 열어서 여기서 인스턴스를 만들고, db에 저장할 거예요

```
python manage.py shell
```

```
(myvenv)
DONGYUN@DESKTOP-HN4KK92 MINGW64 ~/Desktop/멋사/세션/dr-f(2)/mydrf (main|SPARSE)
$ python manage.py shell
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (
2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> █
```

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser

snippet1 = Snippet(code='foo = "bar"\n')
snippet1.save()

snippet2 = Snippet(code='print("hello, world")\n')
snippet2.save()
```

Snippet 모델 (붕어빵 틀)의
인스턴스(붕어빵) 를 만들어 준 후,
그 인스턴스를 저장(save)
해주는 과정입니다!

파이썬 shell에서
방금 만든
모델의 인스턴스 2개를
생성해봅시다

```
>>> from snippets.models import Snippet
>>> from snippets.serializers import SnippetSerializer
>>> from rest_framework.renderers import JSONRenderer
>>> from rest_framework.parsers import JSONParser
>>> snippet1 = Snippet(code='foo = "bar"\n')
>>> snippet1.save()
>>> snippet2 = Snippet(code='print("hello, world")\n')
>>> snippet2.save()
>>>
```

```
>>> snippet1
<Snippet: Snippet object (7)>
>>> snippet2
<Snippet: Snippet object (8)>
```

이 친구들은 object의 형태를 띄고 있네요!
근데 클라이언트에게 이렇게 객체로 주게 되면 알아
볼 수 없으니, 직렬화 과정이 필요하겠죠?

직렬화 과정

(객체를 문자열 또는 바이트로 변환하여 데이터를 전송하는 과정)
클라이언트가 요청한 데이터를 찾아 돌려주는 과정에서 serializer가 수행하는 일입니다.

Model object instance => Python native datatypes => json

```
serializer1 = SnippetSerializer(snippet1)
serializer2 = SnippetSerializer(snippet2)
serializer1.data
serializer2.data
```

```
content1 = JSONRenderer().render(serializer1.data)
content1
content2 = JSONRenderer().render(serializer2.data)
content2
```

Model object instance => Python native datatypes

Python native datatypes => json

```
>>> snippet1
<Snippet: Snippet object (7)>
>>> snippet2
<Snippet: Snippet object (8)>
```

Model object instance

```
>>> serializer1 = SnippetSerializer(snippet1)
>>> serializer2 = SnippetSerializer(snippet2)
>>>
>>> serializer1.data
{'id': 3, 'title': '', 'code': 'foo = "bar"\n', 'linenos': False, 'language': 'python', 'style': 'friendly'}
>>> serializer2.data
{'id': 4, 'title': '', 'code': 'print("hello, world")\n', 'linenos': False, 'language': 'python', 'style': 'friendly'}
>>> content1 = JSONRenderer().render(serializer1.data)
```

Python native datatypes

```
>>> content1 = JSONRenderer().render(serializer1.data)
>>> content1
b'{"id":3,"title":"","code":"foo = \\\"bar\\\"\\n","linenos":false,"language":"python","style":"friendly"}'
>>> content2 = JSONRenderer().render(serializer2.data)
>>> content2
b'{"id":4,"title":"","code":"print(\\\"hello, world\\\")\\n","linenos":false,"language":"python","style":"friendly"}'
```

json

DRF (1)

역직렬화 과정



수신한 바이트를 다시 객체로 변환(메모리 상의 복원)
클라이언트가 요청한 자바스크립트 객체를 json 문자열로 변환하는 과정입니다.

입력받은 stream(json문자열) => Python native datatypes => Model object instance

```
import io

stream1 = io.BytesIO(content1)
data1 = JSONParser().parse(stream1)
stream2 = io.BytesIO(content2)
data2 = JSONParser().parse(stream2)
```

입력받은 stream(json문자열) => Python native datatypes

```
>>> import io
>>> stream1 = io.BytesIO(content1)
>>> data1 = JSONParser().parse(stream1)
>>> stream2 = io.BytesIO(content2)
>>> data2 = JSONParser().parse(stream2)
>>>
>>> serializer1 = SnippetSerializer(data=data1)
>>> serializer1.is_valid()
True
>>> serializer2 = SnippetSerializer(data=data2)
>>> serializer2.is_valid()
True
```

```
>>> data1
{'id': 7, 'title': '', 'code': 'foo = "bar"\n', 'linenos': False, 'language': 'python', 'style': 'friendly'}
>>> data2
{'id': 8, 'title': '', 'code': 'print("hello, world")\n', 'linenos': False, 'language': 'python', 'style': 'friendly'}
>>> type(data1)
```

```
serializer1 = SnippetSerializer(data=data1)
serializer1.is_valid()
```

#모델 오브젝트로 바꿨을 때, 이 값이 유효해야만 역직렬화가 정상적으로 된 것입니다.

그래서 is_valid()를 통해 유효한 데이터가 들어온 지 여부를 확인해주게 됩니다. True 면 유효한 값이 들어온 것이고, False면 유효한 값이 아니게 됩니다.

```
serializer2 = SnippetSerializer(data=data2)
serializer2.is_valid()
```

Python native datatypes => Model object instance

역직렬화 과정

수신한 바이트를 다시 객체로 변환(메모리 상의 복원)
클라이언트가 요청한 자바스크립트 객체를 json 문자열로 변환하는 과정입니다.

입력받은 stream => Python native datatypes => Model object instance

```
serializer1.validated_data
serializer2.validated_data
# 클라이언트의 요청을 모델인스턴스로
# 변환한 후에 is_valid() 가 True라면, 그 값에서
# validated_data 를 사전의 형태로 데려올 수 있어용
```

```
serializer1.save()
serializer2.save()

#모델 인스턴스로 변환된 아이를 저장
```

```
>>> serializer1.validated_data
OrderedDict([('title', ''), ('code', 'foo = "bar"'), ('linenos', False), ('language', 'python'), ('style', 'friendly')])
>>> serializer2.validated_data
OrderedDict([('title', ''), ('code', 'print("hello, world")'), ('linenos', False), ('language', 'python'), ('style', 'friendly')])
>>> serializer1.save()
<Snippet: Snippet object (5)>
>>> serializer2.save()
<Snippet: Snippet object (6)>
>>>
```


추가로 여러 모델 인스턴스를 serializing하려면 다음과 같이 해야 합니다.

`Snippet.objects.all()` : 인스턴스를 모두 불러오는 것

```
Serializer_all_objects = SnippetSerializer(Snippet.objects.all(), many=True)
Serializer_all_objects.data
```

```
>>> Serializer_all_objects.data
[OrderedDict([('id', 1), ('title', ''), ('code', 'foo = "bar"\n'), ('linenos', False), ('language', 'python'), ('style', 'friendly')]), OrderedDict([('id', 2), ('title', ''), ('code', 'print("hello, world")\n'), ('linenos', False), ('language', 'python'), ('style', 'friendly')]), ...]
```

지금까지 serializer를 이용하여 작업하는 과정을 살펴보았는데요,
저희는 간편하게 사용 가능한 **ModelSerialize** 클래스를 사용할 예정입니다!

ModelSerializer 클래스는 사용할 모델을 선언해주고
해당 모델의 fields 중 serializer에 사용할 field를
선택하여 대입해주면 됩니다.

또한 create() 와 update() 함수가 미리 구현되어 있어
따로 만들어줄 필요가 없습니다!

```
from rest_framework import serializers
from snippets.models import Snippet # 사용할 모델 import

class SnippetSerializer(serializers.ModelSerializer):
    class Meta:
        model = Snippet # 사용할 모델
        fields = ['id', 'title', 'code', 'linenos', 'language', 'style'] # 사용할 모델의 필드
```

ModelSerializer

Django에서 Form 클래스와 Model Form 클래스를 제공하듯,
하나의 모델에 대해서 Serializer 클래스를 자동으로 만들어주는 클래스입니다.


우리는 내부 클래스인 Meta 클래스에서
어떤 모델에 대해서 Serializer를 만드는 것인지를 알려주고,
필드들에 대한 정보만 주면 serializer 생성 완료입니다!

```
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES

class SnippetSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    title = serializers.CharField(required=False, allow_blank=True, max_length=100)
    code = serializers.CharField(style={'base_template': 'textarea.html'})
    linenos = serializers.BooleanField(required=False)
    language = serializers.ChoiceField(choices=LANGUAGE_CHOICES, default='python')
    style = serializers.ChoiceField(choices=STYLE_CHOICES, default='friendly')

    def create(self, validated_data):
        """
        Create and return a new `Snippet` instance, given the validated data.
        """
        return Snippet.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """
        Update and return an existing `Snippet` instance, given the validated data.
        """
        instance.title = validated_data.get('title', instance.title)
        instance.code = validated_data.get('code', instance.code)
        instance.linenos = validated_data.get('linenos', instance.linenos)
        instance.language = validated_data.get('language', instance.language)
        instance.style = validated_data.get('style', instance.style)
        instance.save()
        return instance
```



```
from rest_framework import serializers
from snippets.models import Snippet # 사용할 모델 import

class SnippetSerializer(serializers.ModelSerializer):
    class Meta:
        model = Snippet # 사용할 모델
        fields = ['id', 'title', 'code', 'linenos', 'language', 'style'] # 사용할 모델의 필드
```

Shell에서 모델 Serializer 가 어떻게 생겼는지 볼까요?

```
from snippets.serializers import SnippetSerializer
serializer = SnippetSerializer()
print(repr(serializer))
```

```
print(Hello, World ), ( linenos , False), ( language , python ), ( style , friendly )]]]
>>> from snippets.serializers import SnippetSerializer
>>> serializer = SnippetSerializer()
>>> print(repr(serializer))
SnippetSerializer():
  id = IntegerField(read_only=True)
  title = CharField(allow_blank=True, max_length=100, required=False)
  code = CharField(style={'base_template': 'textarea.html'})
  linenos = BooleanField(required=False)
  language = ChoiceField(choices=[('abap', 'ABAP'), ('abnf', 'ABNF'), ('actionscript', 'ActionScript'), ('actionscript3', 'ActionScript 3'), ('ada', 'Ada'), ('adl', 'ADL'), ('agda', 'Agda'), ('aheui', 'Aheui'), ('alloy', 'Alloy'), ('ambienttalk', 'AmbientTalk'), ('amdgpu', 'AMDGPU'), ('ampl', 'Ampl'), ('ansys', 'ANSYS parametric design language'), ('antlr', 'ANTLR'), ('antlr-actionscript', 'ANTLR With ActionScript Target'), ('antlr-cpp', 'ANTLR With CPP Target'), ('antlr-csharp', 'ANTLR With C# Target'), ('antlr-java', 'ANTLR With Java Target'), ('antlr-objc', 'ANTLR With ObjectiveC Target'), ('antlr-perl', 'ANTLR With Perl Target'), ('antlr-python', 'ANTLR With Python Target'), ('antlr-ruby', 'ANTLR With Ruby Target'), ('apacheconf', 'ApacheConf'), ('apl', 'APL'), ('applescript', 'AppleScript'), ('arduino', 'Arduino'), ('arrow', 'Arrow'), ('asc', 'ASCII armored'), ('aspectj', 'AspectJ'), ('aspx-cs', 'aspx-cs'), ('aspx-vb', 'aspx-vb'), ('asymptote', 'Asymptote'), ('augeas', 'Augeas'), ('autohotkey', 'autohotkey')])
```

Nested Serializer

두 모델이 relationship을 가지고 있을 때, 사용할 수 있는 것이 nested serializer입니다.

ModelSerializer 를 사용하게 되면
하나의 serializer에 하나의 모델만 사용할 수 있는데,
여기서 nested serializer를 통해 다른 모델과의 관계를 표현할 수 있습니다!

```
# models.py

class Album(models.Model):
    album_name = models.CharField(max_length=100)
    artist = models.CharField(max_length=100)

class Track(models.Model):
    album = models.ForeignKey(Album,
related_name='tracks', on_delete=models.CASCADE)
    order = models.IntegerField()
    title = models.CharField(max_length=100)
    duration = models.IntegerField()
```

Every letter I sent you.

Yerin Baek



```
class Album(models.Model):
    album_name = models.CharField(max_length=100)
    artist = models.CharField(max_length=100)
```

```
class Track(models.Model):
    album = models.ForeignKey(Album, related_name='tracks',
on_delete=models.CASCADE)
    order = models.IntegerField()
    title = models.CharField(max_length=100)
    duration = models.IntegerField()
```

```
class Track(models.Model):
    album = models.ForeignKey(Album, related_name='tracks',
on_delete=models.CASCADE)
    order = models.IntegerField()
    title = models.CharField(max_length=100)
    duration = models.IntegerField()
```

```
class Track(models.Model):
    album = models.ForeignKey(Album, related_name='tracks',
on_delete=models.CASCADE)
    order = models.IntegerField()
    title = models.CharField(max_length=100)
    duration = models.IntegerField()
```

앨범을 가져올 때, 해당 앨범을 가지는 모든 트랙들을 함께 앨범정보를 serialize
해서 가지고 오고 싶다면!?
그러나 앨범 모델 필드에는 트랙이라는 필드가 없잖아!? 😐

Nested Serializer

```
# serializers.py

class TrackSerializer(serializers.ModelSerializer):
    class Meta:
        model = Track
        fields = ['order', 'title', 'duration']

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackSerializer(many=True, read_only=True)

    class Meta:
        model = Album
        fields = ['album_name', 'artist', 'tracks']
```


Python shell에서 모델 객체를 생성하고 serializing한 결과는 다음과 같습니다.
앨범을 시리얼라이저해서 부르면,
앨범을 참조하는 트랙들이 트랙시리얼라이즈 된 후 함께 출력 됩니다

```
>>> album = Album.objects.create(album_name="The Grey Album", artist='Danger Mouse')
>>> Track.objects.create(album=album, order=1, title='Public Service Announcement', duration=245)
<Track: Track object>
>>> Track.objects.create(album=album, order=2, title='What More Can I Say', duration=264)
<Track: Track object>
>>> Track.objects.create(album=album, order=3, title='Encore', duration=159)
<Track: Track object>
>>> serializer = AlbumSerializer(album)
>>> serializer.data
{
  'album_name': 'The Grey Album',
  'artist': 'Danger Mouse',
  'tracks': [
    {'order': 1, 'title': 'Public Service Announcement', 'duration': 245},
    {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
    {'order': 3, 'title': 'Encore', 'duration': 159},
    ...
  ],
}
```

Serializer Method Field

만약 시리얼라이저로 객체를 직렬화한 JSON에 **모델에 없는 필드를 추가하고 싶거나**
모델에 있는 값을 변형해서 새로운 필드의 값으로 넣고 싶을 때 사용하는 기능입니다.
(예를 들어 모델에는 full_name 필드만 있는데 여기서 이름만 뽑아내서 JSON에 first_name을 추가)

SerializerMethodField로 (모델에 없는) 새로운 필드를 추가가 가능
→ JSON에 필드가 추가됨

기존 트랙에는 없던 앨범 아티스트라는 필드를 추가해주기 위한 메소드를 추가해줘야 합니다
이때 메소드 이름은 get_추가 필드이름 으로 지어주고 리턴값으로 적절한 데이터를 돌려주게 합니다.
여기선 트랙이 참조하는 앨범의 아티스트를 반환하게 해주며 앨범 아티스트를 필드로 삼게 하네요

```
class TrackSerializer(serializers.ModelSerializer):
    album_artist = serializers.SerializerMethodField()

    class Meta:
        model = Track
        fields = '__all__'

    def get_album_artist(self, obj):
        # 함수의 리턴값을 필드로(default=get_필드이름)
        # obj: track 객체
        return obj.album.artist
        # 나의 필드 중 album_artist라는 필드만을 돌려주는 메소드
```

```
>>> serializer = TrackSerializer(track)
>>> serializer.data
{
    'album_artist': 'Danger Mouse',
    'order': 3,
    'title': 'Encore'
    ...
}
```

Serializer Method Field

만약 시리얼라이저로 객체를 직렬화한 JSON에 **모델에 없는 필드를 추가하고 싶거나**
모델에 있는 값을 변형해서 새로운 필드의 값으로 넣고 싶으면
(예를 들어 모델에는 full_name 필드만 있는데 여기서 이름만 뽑아내서 JSON에 first_name을 추가)

객체의 속성 값을 변형해 새로운 필드의 값에 넣을 수 있다.

여기서는 기존에 있던 정보를 변형해줘서 반환하게 해주네요~?
기존 필드에는 유저가 가입한 날짜만 존재했었는데
현재날짜에서 유저가 가입한 날짜를 뺀 일수를 새로운 필드로 넣어줄 수가 있네요~

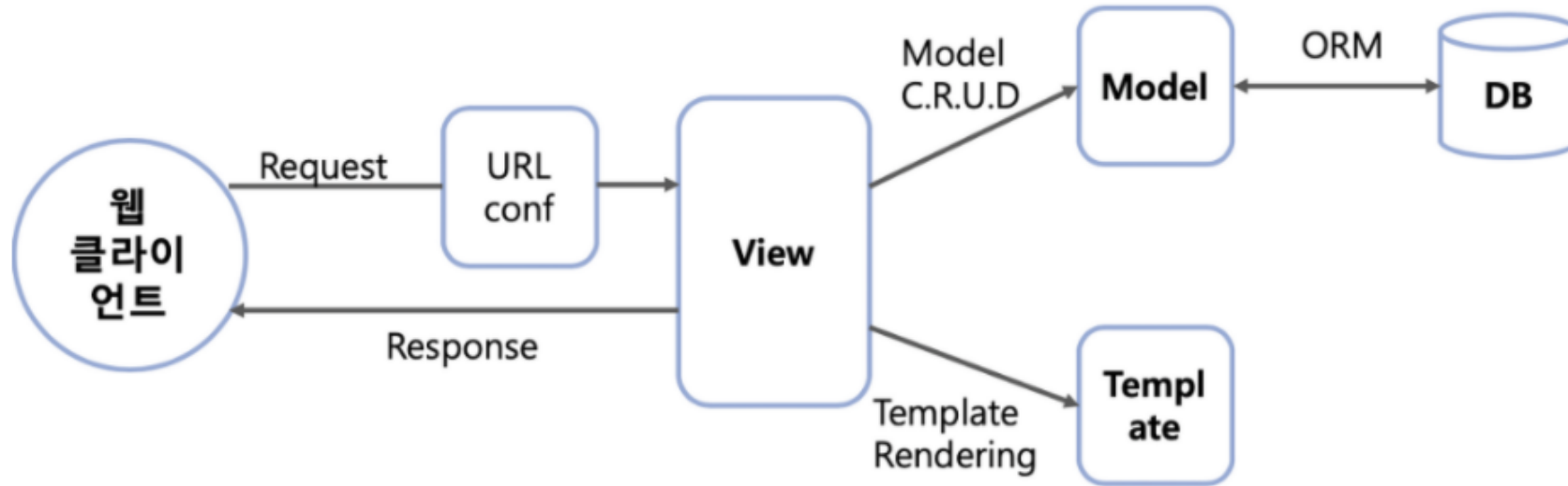
```
from django.contrib.auth.models import User
from django.utils.timezone import now
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    days_since_joined =
    serializers.SerializerMethodField()

    class Meta:
        model = User

    def get_days_since_joined(self, obj):
        return (now() - obj.date_joined).days
```

View



저희가 만들 API 서버의 경우, 위의 그림처럼 View가 Template을 렌더링하는 것이 아니라 JSON 파일을 반환해줘야 합니다.

DRF에서 지원하는 view의 형태는 **단계적**입니다.

DRF에는 APIView를 패턴화한 Generics,
그리고 generics를 구조화한 viewsets가 있습니다.

DRF 화) django view → rest_framework APIView

1.패턴화) APIView → Generic Views

2.구조화) Generic Views → Viewsets

APIView -> Generic views -> Viewsets
으로 갈수록 코드 간략!

```
urlpatterns =  
    path('blog/', views.BlogList.as_view()),  
    # 블로그 글을 post, 모두 get 해올 때 쓸 url (pk값 필요없음)  
  
    path('blog/<int:pk>/', views.BlogDetail.as_view()),  
    # 특정 블로그 글 인스턴스를 가져오고, 삭제하고, 수정할 때 쓸 url  
    # pk값을 줘야지 블로그 글 중 내가 원하는 아이로 특정 가능  
  
]
```

API View

```
# 데이터 처리
from .models import Blog
from .serializers import BlogSerializer

# APIView를 사용하기 위해 import
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from django.http import Http404

# Blog의 목록을 보여주는 역할
class BlogList(APIView):
    # Blog list를 보여줄 때
    def get(self, request):
        blogs = Blog.objects.all()
        # 여러 개의 객체를 serialization하기 위해 many=True로 설정
        serializer = BlogSerializer(blogs, many=True)
        return Response(serializer.data)

    # 새로운 Blog 글을 작성할 때
    def post(self, request):
        # request.data는 사용자의 입력 데이터
        serializer = BlogSerializer(data=request.data)
        if serializer.is_valid(): #유효성 검사
            serializer.save() # 저장
            return Response(serializer.data,
                status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
            status=status.HTTP_400_BAD_REQUEST)
```

```
# Blog의 detail을 보여주는 역할
class BlogDetail(APIView):
    # Blog 객체 가져오기
    def get_object(self, pk):
        try:
            return Blog.objects.get(pk=pk)
        except Blog.DoesNotExist:
            raise Http404

    # Blog의 detail 보기
    def get(self, request, pk, format=None):
        blog = self.get_object(pk)
        serializer = BlogSerializer(blog)
        return Response(serializer.data)

    # Blog 수정하기
    def put(self, request, pk, format=None):
        blog = self.get_object(pk)
        serializer = BlogSerializer(blog, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors,
            status=status.HTTP_400_BAD_REQUEST)

    # Blog 삭제하기
    def delete(self, request, pk, format=None):
        blog = self.get_object(pk)
        blog.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Mixins

```
from .models import Blog
from .serializers import BlogSerializer
from rest_framework import generics
from rest_framework import mixins

# Blog의 목록을 보여주는 역할
class BlogList(mixins.ListModelMixin,
               mixins.CreateModelMixin,
               generics.GenericAPIView):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer

    # Blog list를 보여줄 때
    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    # 새로운 Blog 글을 작성할 때
    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```

```
# Blog의 detail을 보여주는 역할
class BlogDetail(mixins.RetrieveModelMixin,
                 mixins.UpdateModelMixin,
                 mixins.DestroyModelMixin,
                 generics.GenericAPIView):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer

    # Blog의 detail 보기
    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    # Blog 수정하기
    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    # Blog 삭제하기
    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

Mixins 부터 queryset 이라는 멤버와 serializer_class 라는 멤버(또는 get_serializer_class() 라는 함수)들이 필요해요
이를 통해서 GenericAPIView는 기본적인 REST 기능을 수행할 수 있게 됩니다.

(다음 시간에 view 실습으로 다시 익힐 것이니 오늘은 가볍게만 넘어가주셔도 됩니다)

Queryset : 어떠한 모델을 보여줄것인가? 를 정의

데이터베이스에서 전달 받은 객체의 목록 (Django ORM에서 발생한 자료형)

Blog 데이터들을 보여줄 테니, 전체 Blog 데이터를 반환

Serializer : REST 로 데이터를 주고 받을 때, 모델을 어떻게 주고 받을 것인가를 정의하기 위한 클래스

Generic CBV


DRF (1)

```
from .models import Blog
from .serializers import BlogSerializer
from rest_framework import generics
from rest_framework import mixins

# Blog의 목록을 보여주는 역할
class BlogList(mixins.ListModelMixin,
               mixins.CreateModelMixin,
               generics.GenericAPIView):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer

    # Blog list를 보여줄 때
    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    # 새로운 Blog 글을 작성할 때
    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```



```
from .models import Blog
from .serializers import BlogSerializer
from rest_framework import generics


# Blog의 목록을 보여주는 역할
class BlogList(generics.ListCreateAPIView):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer
```

```
# Blog의 detail을 보여주는 역할
class BlogDetail(mixins.RetrieveModelMixin,
                 mixins.UpdateModelMixin,
                 mixins.DestroyModelMixin,
                 generics.GenericAPIView):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer

    # Blog의 detail 보기
    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    # Blog 수정하기
    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    # Blog 삭제하기
    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```



```
# Blog의 detail을 보여주는 역할
class BlogDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer
```

ViewSet(FBV)

APIView보다 좀 더 발전한 형태인 generics의
ListCreateAPIView, RetrieveUpdateDestroyAPIView 등과 같은 형태의 뷰는
하나의 url에 대해 http 메소드(get, post, put, patch, delete 등)를 다르게 사용해
여러 개의 뷰를 처리하도록 해준답니다.

ViewSet은 여기서 더 나아가 여러 개의 url 패턴에 대해
여러 개의 http 메소드를 사용해 여러 개의 뷰를 처리하도록 해주고 있어요.

ViewSet 은 말그대로 View들의 Set이에요.

즉, 요청과 응답에 사용되는 여러 View들이 모여있다는 것입니다!

가장 기본적으로 사용하는 ViewSet이 바로 ModelViewSet입니다.

ModelViewSet은 기본적으로
Retrieve, List, Create, Destroy, Update의 뷰를 제공합니다

(세부적으로 커스텀하기 위해서는
ViewSet의 내장된 기능을 파헤쳐보는 것이 필요합니다.)

```
from .models import Blog
from .serializers import BlogSerializer
from rest_framework import viewsets

# Blog의 목록, detail 보여주기, 수정하기, 삭제하기 모두 가능
class BlogViewSet(viewsets.ModelViewSet):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer
```

```
C: > Users > DONGYUN > AppData > Local > Programs > Python > Python39 > Lib > site-packages > rest_fr
234
235 class ModelViewSet(mixins.CreateModelMixin,
236                     mixins.RetrieveModelMixin,
237                     mixins.UpdateModelMixin,
238                     mixins.DestroyModelMixin,
239                     mixins.ListModelMixin,
240                     GenericViewSet):
241     """
242     A viewset that provides default `create()`, `retrieve()`, `update()`,
243     `partial_update()`, `destroy()` and `list()` actions.
244     """
245     pass
246
```


Router 사용해 url 매핑하기

ViewSet의 경우에는 하나의 class에 Blog 목록과 detail을 보여주는 기능이 모두 존재합니다.

List, Create 뷰는 `model_name/`와 같은 형태의 url을 사용해 해당 모델의 여러 인스턴스를 보여주거나, 새로운 인스턴스를 만들어주는 역할을 합니다.

Retrieve, Update, Destroy 뷰는 `model_name/<int:pk>/`와 같은 형태의 url을 사용해 해당 pk에 해당하는 모델 인스턴스를 보여주거나, 업데이트 하거나, 제거해주는 역할을 합니다.

Router 사용해 url 매핑하기

하지만 이를 하나의 path 함수로는 표현할 수 없습니다.
왜냐하면 모든 Blog 목록을 보여주는 url의 경우에는 pk 값이 필요가 없지만,
detail을 보여주기 위해서는 pk 값이 반드시 필요하기 때문입니다.

따라서 ViewSet을 사용하기 위해서는
서로 다른 path 함수를 하나로 묶어주는 과정이 필요합니다!

이렇게 자동으로 url과 메소드들이 매핑되기 위해서는
urls.py에 Router를 아래와 같이 추가해주셔야 합니다

```
from django.urls import path, include
from .views import BlogViewSet #뷰에서 정의한 클래스명
from rest_framework.routers import DefaultRouter
# 라우트 위해 import

router = DefaultRouter()
#rest framework의 DefaultRouter라는 객체를 생성
#mapping 하고자 하는 view를 등록

router.register('blog', BlogViewSet)
# 첫 번째 인자는 url의 prefix
# 두 번째 인자는 ViewSet

urlpatterns = [
    path('', include(router.urls))
]
```

```
urlpatterns =  
    path('blog/', views.BlogList.as_view()),  
    # 블로그 글을 post, 모두 get 해올 때 쓸 url (pk값 필요없음)  
  
    path('blog/<int:pk>', views.BlogDetail.as_view()),  
    # 특정 블로그 글 인스턴스를 가져오고, 삭제하고, 수정할 때 쓸 url  
    # pk값을 줘야지 블로그 글 중 내가 원하는 아이로 특정 가능  
  
]
```



```
router.register('blog', BlogViewSet)  
# 첫 번째 인자는 url의 prefix  
# 두 번째 인자는 ViewSet  
  
urlpatterns =  
    path('', include(router.urls))  
  
]
```



수고하셨습니다



서강대 김동윤

