🡪

DRF 自带 ModelSerializer类。 继承该类将提供一些序列器方法。

例如：

is\_vailid(self,..) 验证提供给序列器的数据是否可以有效地更新或者创建数据库中表的条目。

save (self, …) 用于更新或者新建条目。

create (self, …) 用于创建数据库中表的条目。

update (self, …) 用于更新数据库中表的条目。

🡪

Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types.

序列化将Django中的对象，modal变为普通的数据类型，例如字符串等，进而可以通过网络进行传输。

反序列化（[deserialization](https://www.baidu.com/link?url=pph6ZfeRkc-etu8uwaHtAgL34vzxQlZ-8zN8rJ601cOd9s8OwtH3q3Y2Kq9rQIG5asd-9I2GtyqPh0MLP4pfbm-N4aKrQQaGilnVSyrCgENCKzMRgLGPpSIwhc_F59EE&wd=&eqid=e5d0896b00024a800000000660538fb1)）将Django中普通的数据类型变为Django中的对象。

序列化：

serializer = SnippetSerializer(snippet) –> snippet为Django中snippet model的实例，SnippetSerializer为序列化类

content = JSONRenderer().render(serializer.data) –-> serializer.data为序列器序列化的结果，是字符串

反序列化：

import io

stream = io.BytesIO(content)

data = JSONParser().parse(stream) 🡪 data为字典

serializer = SnippetSerializer(data=data) –> serializer此时为Django对象

🡪

使用regr(Serializer)可以显示出序列器的表示。

🡪

SnippetSerializer(serializers.ModelSerializer) 🡪 可以自动提取models中的域，并且实现默认的create和update方法

SnippetSerializer(serializers.Serializer) 🡪 不自动提取models中的域，不默认实现create和update方法

🡪

我们在views.py中去调用serializer.save() 的时候，如果没有重写save()方法，则会根据是否在建立serializer实例的时候存入一个已有的instance来决定调用的是serializer中的udpate()方法， 还是serializer的create（）方法。

🡪

我们可以在调用serializers.save()的时候为实际调用的serializers中的update以及create函数传入额外的数据。

方法是在调用save()方法的时候传入额外的keyword参数，例如

serializer.save(owner=request.user)

这样在serializer的update以及create方法中，validate.data将会有这些额外的参数。

🡪

重写Serializer的save()方法，如果不重写save()方法，那么调用Serializer的save方法将调用Serializer的update或者create方法。如果重写后，则可以根据我们自身的逻辑去执行操作，而不局限于创建或者更新数据库中的对象。

🡪

当我们用序列器进行save（）的时候，我们必须先调用is\_valid()方法去验证上传的数据是否有效，是否符合我们models.py中的要求。

--如果我们调用is\_valid(raise\_exception=true) 参数执行，那么如果上传的数据不合法，还将默认的自动返回400 Bad Request状态码。

--执行完is valid () 方法以后，如果有效则数据会存在于validated\_data当中。如果无效，则错误信息会以字典的形式存放在errors中。

--Field-level validation

我们还可以在序列器中以validate\_<field\_name>的格式，对某一域进行特定的审查。例如可以达到我们不仅要求上传的title是charfield并且，还要求必须含有django关键字。该方法在验证通过时，返回域的值，在不通过时候，返回serializers.ValidationError

--object-level validation

Object-level validation和Field-level validation的区别在于，Object-level validation在验证过程中会关系到多个上传数据的域，而Field-level validation只检查单一域。

🡪

构建序列化实例的时候，需要传给序列器的参数

snippets = Snippet.objects.all()

serializer = SnippetSerializer(snippets, many=True) 🡪 当实例化序列器，进行序列化，为序列器充入数据的时候,传入models实例

serializer = SnippetSerializer(data=request.data) 🡪当实例化序列器，作为反序列化的时候，传入request.data数据

serializer = SnippetSerializer() 🡪 只为实例化序列器时候，不放入任何数据的时候。

🡪

当我们调用Serializers的is\_valid()方法的时候，Serializers将验证我们在Serializers中指定的fields,按照这个fields当初在modesl.py中定义的类型，条件进行验证。

🡪

我们构建完Serializer以后，可以在./manage.py 中进行测试，来确定是否存在语法问题等。

例如：

test = PatchTrackingSerializer() #构建一个Serializer实例

test.is\_valid()

# 此时结果为false, 在错误原因中test.errors。

{'build': [ErrorDetail(string=u'This field is required.', code=u'required')]}

我们需要传入build这个域。由于验证没有通过, test.validated\_data是空的，为{}.

test = PatchTrackingSerializer(data={"comment":"abc","build":build})

此时验证还是false.错误信息为{'build': [ErrorDetail(string=u'Incorrect type. Expected pk value, received rgr\_build\_tbl.', code=u'incorrect\_type')]}由于这个Serializer我们采用ModelSerializer, 所以在build这个外键上需要传入的是一个pk.

test = PatchTrackingSerializer(data={"comment":"abc","build":6290})

此时验证通过，因为在models.py中需要必填的项，我们都填了，并且满足数据域类型的要求，所以通过，可以在. validated\_data中找到数据。

🡪

基于class的Views和Serializer协同工作例子:

def post (self, request, format=None):  
 serializer = PatchTrackingSerializer(data=request.data)

#创建一个PatchTrackingSerializer的序列器实例，这里data不是validated\_data. Validated\_data要序列器调用is\_valid内部的调用to\_internal\_value()以后才会存在。  
 if serializer.is\_valid(): # 如果is\_valid返回true,则是合法数据，从而validated\_data生成  
 serializer.save() # 调用序列器的save()将实际调用序列器中的update或者create方法  
 return Response(serializer.data, status=status.HTTP\_201\_CREATED)  
 return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

Serializer:

def create (self, validated\_data): # 要求create的时候使用的是validated\_data

return Comment.objects.create(\*\*validated\_data)

def update (self, instance, validated\_data): # 要求update的时候使用的是validated\_data

instance.email = validated\_data.get('email', instance.email)

instance.content = validated\_data.get('content', instance.content)

instance.created = validated\_data.get('created', instance.created)

instance.save() 🡪 我们这里重写了update方法，update的逻辑是更新models.py中数据库的数据。我们也可以根据业务逻辑，去执行其他逻辑。

return instance

🡪

序列器 (Serializer)中的数据域:

test = PatchTrackingSerializer(data={"comment":"aaa","build":10146})—>此时test这个序列化实例只有initial\_data。因为还没有经过is\_valid()方法。

test.is\_valid()🡪如果为true,则序列器将拥有额外的两个数据域名。总体将有.initial\_data保留初始的传入serializaer的数据, .data将数据显示成dict格式，.validated\_data是将初始数据验证过的结果数据。

🡪 to\_internal\_value（）方法是调用serializer的is valid () 方法的时候默认调用的方法，它接受当时传入Serializers的data进行验证以后，返回validated\_data或者报错。返回的内容就是以后validated\_data的内容。Serializer的create, update () 方法使用validated data数据去更新或者创建数据库中的实例。

一个重写to\_internal\_value例子,如果不重写，则DRF默认使用models.py中域类型进行验证：

def to\_internal\_value(self, data):

score = data.get('score')

player\_name = data.get('player\_name')

# Perform the data validation.

if not score:

raise ValidationError({ 🡪认证失败，抛出ValidationError,以字典的形式表现错误信息

'score': 'This field is required.'

})

if not player\_name:

raise ValidationError({

'player\_name': 'This field is required.'

})

if len(player\_name) > 10:

raise ValidationError({

'player\_name': 'May not be more than 10 characters.'

})

# Return the validated values. This will be available as the `. validated\_data` property.

return { 🡪 return所返回的内容将会是以后validated\_data的内容

'score': int(score),

'player\_name': player\_name

}

🡪

我们在进行POST REST API请求的时候，进而使用validated\_data去调用save()方法。但是有时候，我们不需要在请求的json体中，包括所有的必须数据，我们便需要在server这边添加一些数据。

例如在views中重写get或者post方法：

try:  
 upload\_data = dict(request.data.items())  
 # Modify Data  
 build\_obj = rgr\_build\_tbl.objects.get(slug=upload\_data['build'])  
 upload\_data['build'] = build\_obj.pk  
 if not upload\_data.get("QATeam\_email", 0):  
 upload\_data["QATeam\_email"] = "hui\_h.jin@nokia.com,xin.deng@nokia.com"  
 if not upload\_data.get("scrumTeam\_email", 0):  
 upload\_data["scrumTeam\_email"] = "hui\_h.jin@nokia.com,xin.deng@nokia.com"  
 if not upload\_data.get("comment", 0):  
 upload\_data["comment"] = "Hui try to create record from rest api"  
except:  
 return Response({"Message": "Error happens when process request.data"}, status=status.HTTP\_400\_BAD\_REQUEST)  
serializer = PatchTrackingSerializer(data=upload\_data) 🡪 在构建Serializer的时候，传入data并不一定是用户直接上传的request.data。我们可以构建一个字典，再根据request.data进行填充以后，传入Serializer,进行再is\_valid以后再调用save()方法。

🡪

调用完Serializer的is\_valid()方法以后，且为true,才会有serializers.data, serializers.validated\_data, save()方法。

🡪

使用DRF更新数据的时候：

如果使用PUT更新数据，那么我们要在views中实现def put方法， 否则报错put方法不被支持。PUT方法的实现，可以自己手动实现，也可以选择一些DRF已经提供实现了的genericView。由于采用的是PUT方法更新，所以应该提交完整的数据对象，也就是所有数据域进行更新。

如果使用patch更新数据，那么我们要在views中实现def patch方法， 否则报错patch方法不被支持。patch方法的实现，可以自己手动实现，也可以选择一些DRF已经提供实现了的genericView。由于采用的是patch方法更新，所以只需要提交需要更改的数据域。并且特别的对于DRF来说，在实例化Serializers的时候，加入关键参数partial=True

**Serializer的域以及Serializer的关系域**

🡪

Serializers中的Field. DRF的Serializer提供了很多的Fields，用于不同的用途。其中有最原始的Field，对其中方法都要重写，也有DRF帮我们写好的Field，例如SlugField，CharField等等，DRF根据他们的用途，默认设置了许多参数。

***如果我们采用ModelSerializer或者其他的DRF提供的序列器，然后fied=\_\_all\_\_， DRF会默认的将model中所有域提取所有成对应的Serializer Field，但是如果我们有自定义的行为，例如我们想外键域不以默认的id显示，则我们要自行重新定义这个field。***

--首先DRF基本上所有的DRF Fields都支持以下几个参数设置。

--read\_only

设置read\_only=True以后，这个域将只会用在序列化当中。不会用于序列器在执行create或者update方法的validated\_data当中。也就是说反序列化的时候，不会使用这个Field的内容。

--write\_only

设置为True以后，这个域会用于反序列化过程中,但是不会用于序列化的字符串,json表示中。

--required

设置为true以后，在反序列化的时候，必须要包含这个域。

--default

设置默认值。当序列化以及反序列化的时候，如果这个域没有提供值，则使用默认值。但是在partial update的时候，不会包含在validated data中，在patch的时候，只有上传的值会包含在validated\_data中。Default的值可以设置为通过function或者callable取得。设置了default值以后，暗示了这个域名是不required的，所以不能够将一个域设为即是required又是default的，这会抛出错误。

--allow\_null

默认情况下，如果这个域没有设置allow\_null为True,则不允许None为此域的值。如果设为True,则允许。并且如果设为True,但是没有同时为这个域设置default,则暗示在序列化是值为null,在反序列化时不包含此值。

--source

指定为这个域指定其数据的方式。可以是一个只允许self作为参数的方法，也可以采用以.方式访问的对象类型。

另外如果指定source=”\*”这样的方式，又其特殊的意义。代表将传入整个对象，作为其域的值。这对于在序列化是，nested representation的时候特别有用。

--validators

为这个域指定将应用的验证器，可以多个。需要注意的是，验证器抛出的异常为DRF框架下的serializers.ValidationError

--error\_messages

一个字典，作为一个mapping的匹配，将映射错误码到错误信息

--label

一小段字符串，作为在HTML的情况下该域的名字

--help\_text

一段字符串，用于形容该域

--initial

为该域执行初始的数据，可以是一个callable的对象

--style

用于指定该域在HTML以什么样式实现

🡪

序列器中用于表示关系（relations）的域，这里所说的域***是专门针对于表示关系的***。普通类型的数据，例如字符串，email等有其他的专门针对于普通数据的普通数据域。

DRF中有专门用于表示数据库中的域，这些域有能力表示在Django中的外键关系，多对多关系，一对一关系，以及自定义的GenericForeignKey.

当我们使用DRF中的现成序列器时候会自动提取models中定义的域，所以如果我们想确定DRF自动提取出了什么类型的域来表示关系时，可以实例化一个序列器，然后print(regr(instance))来查看。

DRF在表示关系时，queryset的作用是验证提供的参数是否在queryset范围当中，并不是其中的某一个对象。

Serializer中用于表示关系的域，SlugRelatedField，PrimaryKeyRelatedField等。这些关系域的表示不仅作用在序列化也会作用在反序列化上。意思是当指定以后，序列化时将显示具体指定的关系中的那个表的域的值，而在反序列化的时候需要的也是具体指定的关系中的那个表的域的值。

例如：

AccountSerializer():

id = IntegerField(label='ID', read\_only=True)

name = CharField(allow\_blank=True, max\_length=100, required=False)

owner = PrimaryKeyRelatedField(queryset=User.objects.all()) 🡪 序列化器自动提取将外键提取为PrimaryKeyRelatedField，并且owner的范围是User.objects.all()所得的queryset中的对象

例子：

假如数据库中如此定义，一个唱片以及歌曲的数据库

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 Meta:

unique\_together = ('album', 'order')

ordering = ['order']

def \_\_unicode\_\_(self):

return '%d: %s' % (self.order, self.title)

***--如果使用StringRelatedField关系域***

StringRelatedField使用models.py中的\_\_unicode\_\_去表示关系的目标。

class AlbumSerializer(serializers.ModelSerializer):

tracks = serializers.StringRelatedField(many=True) # 添加tracks这个关系域

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

序列化的结果为：

{

'album\_name': 'Things We Lost In The Fire',

'artist': 'Low',

'tracks': [

'1: Sunflower',

'2: Whitetail',

'3: Dinosaur Act',

...

]

}

此域将只能是read only,并且由于是一个album对多个track,所以加上了many=True

***--如果使用PrimaryKeyRelatedField***

class AlbumSerializer(serializers.ModelSerializer):

tracks = serializers.PrimaryKeyRelatedField(many=True, read\_only=True) –>Album在models.py中没有这个域，这里额外定义一个域，并且read\_only=True,只会在序列化时显示，在反序列化时不会添加到Album的记录中。

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

序列化的结果为：

{

'album\_name': 'Undun',

'artist': 'The Roots',

'tracks': [

89,

90,

91,

...

]

}

默认情况下，域是read-write。

***--如果使用HyperlinkedRelatedField***

HyperlinkedRelatedField用超链接表示目标。

class AlbumSerializer(serializers.ModelSerializer):

tracks = serializers.HyperlinkedRelatedField(

many=True,

read\_only=True,

view\_name='track-detail'

)

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

序列化的结果为：

{

'album\_name': 'Graceland',

'artist': 'Paul Simon',

'tracks': [

'http://www.example.com/api/tracks/45/', 🡪以超链接的形式表示

'http://www.example.com/api/tracks/46/',

'http://www.example.com/api/tracks/47/',

...

]

}

***--*--****如果使用SlugRelatedField**

SlugRelatedFile用于表示关系目标中的某个域

class AlbumSerializer(serializers.ModelSerializer):

tracks = serializers.SlugRelatedField(

many=True,

read\_only=True,

slug\_field='title' 🡪表示的是Track表中的title域

)

class Meta:

model = Album

fields = ('album\_name', 'artist', 'tracks')

序列化的结果为：

{

'album\_name': 'Dear John',

'artist': 'Loney Dear',

'tracks': [

'Airport Surroundings',

'Everything Turns to You',

'I Was Only Going Out',

...

]

}

默认为read-write,***通常对应表中的域为unique=True的域***，并且必须带有参数slug\_field指定为表中的哪个域）。并且必须显示指定queryset或者指定read\_only=True