REST framework 序列化组件之Serializer

原创

i0208 于 2020-02-15 17:49:52 发布 161 收藏
分类专栏： REST framework

Json序列化与Serializer序列化的区别

Json:

序列化: 将对象序列化成字符串用于传输
反序列化: 将字符串反序列化成对象用于使用

Serializer:

序列化: 将Model类对象序列化成字符串用于传输
反序列化: 将字符串反序列化成Model对象用于使用

Serializer

序列化组件的工作方式与django froms组件非常相似,为每一个model类通过一套序列化工具类

Serializer的构造方法为：

```
Serializer(instance=None, data=empty, **kwarg)
```

1. 用于序列化时，将模型类对象传入instance参数
2. 用于反序列化时，将要被反序列化的数据传入data参数
3. 除了instance和data参数外，在构造Serializer对象时，还可通过context参数额外添加数据，如

```
serializer = AccountSerializer(account, context={'request': request})
```

通过context参数附加的数据，可以通过Serializer对象的context属性获取。

注意:

1. 使用序列化器的时候一定要注意，序列化器声明了以后，不会自动执行，需要我们在视图中进行调用才可以。
2. 序列化器无法直接接收数据，需要我们在视图中创建序列化器对象时把使用的数据传递过来。
3. 序列化器的字段声明类似于我们前面使用过的表单系统。
4. 开发restful api时，序列化器会帮我们把模型数据转换成字典.
5. drf提供的视图会帮我们把字典转换成json,或者把客户端发送过来的数据转换字典.

使用

序列化器的使用分两个阶段：

在客户端提交数据时，使用序列化器可以完成对数据的反序列化。

在服务器响应数据时，使用序列化器可以完成对数据的序列化。

- models.py

```
class User(models.Model):
    SEX_CHOICES = [
        [0, '男'],
        [1, '女'],
    ]
    name = models.CharField(max_length=64)
    pwd = models.CharField(max_length=32)
    phone = models.CharField(max_length=11, null=True, default=None)
    sex = models.IntegerField(choices=SEX_CHOICES, default=0)
    icon = models.ImageField(upload_to='icon', default='icon/default.jpg')

    class Meta:
        # 自定义创建的表名
        db_table = 'user'
        # admin界面中显示的表面与表名复数形式
        verbose_name = '用户'
        verbose_name_plural = verbose_name

    def __str__(self):
        return self.name

# 补充: 获取choices的值:
sex = obj.get_sex_display()  #  得到 '男' 或 '女'
```

- settings.py

```
# 注册rest_framework
INSTALLED_APPS = [
    # ...
    'rest_framework',
]

# 配置数据库
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'db',
        'USER': 'root',
        'PASSWORD': 'root'
    }
}

# media资源
MEDIA_URL = '/media/'  # 后期高级序列化类与视图类，会使用该配置
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')  # media资源路径

# 国际化配置
LANGUAGE_CODE = 'zh-hans'
TIME_ZONE = 'Asia/Shanghai'
USE_I18N = True
USE_L10N = True
USE_TZ = False
```

- 主路由

```
from django.conf.urls import url, include
from django.contrib import admin
from django.views.static import serve
from django.conf import settings
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    # 路由分发
    url(r'^api/', include('api.urls')),
    # 暴露静态文件
    url('^media/(?P<path>.*)', serve, {'document_root':settings.MEDIA_ROOT})
]
```

- 子路由

```
urlpatterns = [
    url(r'^users/$', views.User.as_view()),
    url(r'^users/(?P<pk>.*)/$', views.User.as_view()),
]
```

Serializer 序列化的使用

序列化提供给前台的字段个数由后台决定，可以少提供

但是要提供与数据库对应的字段，名字一定要与数据库字段相同

可自定义序列化字段，字段名可以随意，但要定义 def get_字段名(self, obj) 函数，来完成一些需要处理在返回的数据

- 应用下创建serializers.py

```
from rest_framework import serializers, exceptions
from django.conf import settings

from . import models

class UserSerializer(serializers.Serializer):
    # 定义需要序列化的字段 名字一定要与数据库字段相同
    name = serializers.CharField()
    phone = serializers.CharField()

    # 自定义序列化字段
    # 属性名随意，值由固定的命名规范方法提供:
    # get_属性名(self, 参与序列化的model对象)
    # 返回值就是自定义序列化属性的值
    gender = serializers.SerializerMethodField()
    def get_gender(self, obj):  # obj是对应表中的一条记录(即:对象)
        # choice类型的解释型值 get_字段_display() 来访问
        return obj.get_sex_display()
    # get_gender函数的返回值就是gender字段的值

    icon = serializers.SerializerMethodField()
    def get_icon(self, obj):
        # settings.MEDIA_URL: 自己配置的 /media/，给后面高级序列化与视图类准备的
        # obj.icon不能直接作为数据返回，因为内容虽然是字符串，但是类型是ImageFieldFile类型
        return '%s%s%s' % (r'http://127.0.0.1:8000', settings.MEDIA_URL, str(obj.icon))
```

- views.py

1）从数据库中将要序列化给前台的model对象，或是多个model对象查询出来

```
user_obj = models.User.objects.get(pk=pk)
# 或者
user_obj_list = models.User.objects.all()  # 得到的是查询集QuerySet
```

2）将对象交给序列化处理，产生序列化对象，如果序列化的是多个数据(查询集QuerySet)，要设置many=True

```
user_ser = serializers.UserSerializer(user_obj) 或者
user_ser = serializers.UserSerializer(user_obj_list, many=True)
# 注意: many会遍历user_obj_list,触发是一个具体对象,否则就需要制定many=True
```

3）序列化 对象.data 就是可以返回给前台的序列化数据

```
return Response({
        'status': 0,
        'msg': 0,
        'results': user_ser.data
    })
```

```
class User(APIView):
    def get(self, request, *args, **kwargs):
        pk = kwargs.get('pk')
        if pk:
            try:
                # 用户对象不能直接作为数据返回给前台
                user_obj = models.User.objects.get(pk=pk)
                # 序列化一下用户对象
                user_ser = serializers.UserSerializer(user_obj)
                # print(user_ser)
                # print(user_ser.data)  # {'name': 'xiongda', 'phone': '110'} 序列化后的字段
                return Response({
                    'status': 0,
                    'msg': 0,
                    'results': user_ser.data
                })
            except:
                return Response({
                    'status': 2,
                    'msg': '用户不存在',
                })
        else:
            # 用户对象列表(queryset)不能直接作为数据返回给前台
            user_obj_list = models.User.objects.all()
            # 序列化一下用户对象
            user_ser_data = serializers.UserSerializer(user_obj_list, many=True).data
            return Response({
                'status': 0,
                'msg': 0,
                'results': user_ser_data
            })
```

Serializer 反序列化的使用

使用序列化器进行反序列化时，需要对数据进行验证后，才能获取验证成功的数据

在定义序列化器时，指明每个字段的序列化类型和选项参数，本身就是一种验证行为。

1）设置必填与选填序列化字段，设置校验规则
2）为需要额外校验的字段提供局部钩子函数，如果该字段不入库，且不参与全局钩子校验，可以将值取出校验
3）为有联合关系的字段们提供全局钩子函数，如果某些字段不入库，可以将值取出校验
4）重写create方法，完成校验通过的数据入库工作，得到新增的对象

- serializers.py

```
# 1) 哪些字段必须反序列化
# 2) 字段都有哪些安全校验
# 3) 哪些字段需要额外提供校验
# 4) 哪些字段间存在联合校验
# 注：反序列化字段都是用来入库的，不会出现自定义方法属性，会出现可以设置校验规则的自定义属性(如:re_pwd)

class UserDeserializer(serializers.Serializer):
    name = serializers.CharField(
        max_length=64,
        min_length=3,
        error_messages={
            'max_length': '太长',
            'min_length': '太短'
        }
    )
    pwd = serializers.CharField()
    phone = serializers.CharField(required=False)  # required = False 该字段可以不传值
    sex = serializers.IntegerField(required=False)

    # 自定义有校验规则的反序列化字段
    re_pwd = serializers.CharField(required=True)

    # 其中：
    # name，pwd，re_pwd为必填字段
    # phone，sex为选填字段
    # 五个字段都必须提供完成的校验规则


    # 局部钩子：validate_要校验的字段名(self, 当前要校验字段的值)
    # 校验规则：校验通过返回原值，校验失败，抛出异常
    def validate_name(self, value):
        print(value)  # name字段的值
        if 'g' in value.lower():  # 名字中不能出现g
            raise exceptions.ValidationError('名字非法，是个鸡贼！')
        return value

    # 全局钩子：validate(self, 系统与局部钩子校验通过的所有数据)
    # 校验规则：校验通过返回原值，校验失败，抛出异常
    def validate(self, attrs):
        print(attrs)  # OrderedDict([('name', 'haha'), ('pwd', '123'), ('phone', '120'), ('re_pwd', '123')]
        pwd = attrs.get('pwd')
        re_pwd = attrs.pop('re_pwd')
        if pwd != re_pwd:
            raise exceptions.ValidationError({'pwd&re_pwd': '两次密码不一致'})
        return attrs

    # 要完成新增，需要自己重写 create 方法(类似接口类的规则)
    def create(self, validated_data):
        print(validated_data)  # {'name': 'haha', 'pwd': '123', 'phone': '120'}
        # 尽量在所有校验规则完毕之后，数据可以直接入库
        return models.User.objects.create(**validated_data)
```

- views.py

1) 前端传来的数据字段必须赋值给系列化器的data参数

```
book_ser = serializers.UserDeserializer(data=request_data)
```

2) 在获取反序列化的数据前，必须调用**is_valid()**方法进行验证，验证成功返回True，否则返回False。

```
book_ser.is_valid()  # 结果为 通过(True) | 不通过(Fales)
```

若检验不通过,可以通过序列化器对象的**errors**属性获取错误信息，返回字典，包含了字段和字段的错误

```
book_ser.errors
```

若校验通过,可以通过序列化器对象的**validated_data**属性获取数据。

```
book_ser.calidated_data
```

3) 得到新增的对象，再正常返回

```
book_ser.save()
```

```
class User(APIView):
    # 只考虑单增
    def post(self, request, *args, **kwargs):
        # 请求数据
        request_data = request.data
        # 数据是否合法（增加对象需要一个字典数据）
        if not isinstance(request_data, dict) or request_data == {}:
            return Response({
                'status': 1,
                'msg': '数据有误',
            })
        # 数据类型合法，但数据内容不一定合法，需要校验数据，校验(参与反序列化)的数据需要赋值给data
        book_ser = serializers.UserDeserializer(data=request_data)

        # 序列化对象调用is_valid()完成校验，校验失败的失败信息都会被存储在 序列化对象.errors
        if book_ser.is_valid():
            # 校验通过，完成新增
            book_obj = book_ser.save()  # 自动调用系列化器里的create方法,并拿到新增的对象
            return Response({
                'status': 0,
                'msg': 'ok',
                'results': serializers.UserSerializer(book_obj).data  # 拿到序列化数据给前台
            })
        else:
            # 校验失败
            return Response({
                'status': 1,
                'msg': book_ser.errors,  # 将错误字段的信息返回给前台
            })
```

反序列化-保存数据

前面的验证数据成功后,我们可以使用序列化器来完成数据反序列化的过程.这个过程可以把数据转成模型类对象.

可以通过实现create()和update()两个方法来实现。

```
class UserDeserializer(serializers.Serializer):
    ...

    def create(self, validated_data):  # 新增
        return models.User.objects.create(**validated_data)

    def update(self, instance, validated_data):  # 更新
        # instance为要更新的对象实例
        instance.name = validated_data.get('name', instance.name)
        instance.pwd= validated_data.get('pwd', instance.pwd)
        instance.save()
        return instance
```

实现了上述两个方法后，在反序列化数据的时候，就可以通过save()方法返回一个数据对象实例了

```
book = serializer.save()
```

如果创建序列化器对象的时候，没有传递instance实例，则调用save()方法的时候，create()被调用，相反，如果传递了instance实例，则调用save()方法的时候，update()被调用。

```
from db.serializers import BookInfoSerializer
data = {'btitle': '封神演义'}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()  # True
serializer.save()  # <BookInfo: 封神演义>

from db.models import BookInfo
book = BookInfo.objects.get(id=2)
data = {'btitle': '倚天剑'}
serializer = BookInfoSerializer(book, data=data)
serializer.is_valid()  # True
serializer.save()  # <BookInfo: 倚天剑>
book.btitle  # '倚天剑'
```

附加说明

1） 在对序列化器进行save()保存时，可以额外传递数据，这些数据可以在create()和update()中的validated_data参数获取到

```
# request.user 是django中记录当前登录用户的模型对象
serializer.save(xionger=request.user)
```

2）默认序列化器必须传递所有required的字段，否则会抛出验证异常。但是我们可以使用partial参数来允许部分字段更新

```
# Update `comment` with partial data
serializer = CommentSerializer(comment, data={'content': u'foo bar'}, partial=True)
```

补充

常用字段类型：

字段	字段构造方式
BooleanField	BooleanField()
NullBooleanField	NullBooleanField()
CharField	CharField(max_length=None, min_length=None, allow_blank=False, trim_whitespace=True)
EmailField	EmailField(max_length=None, min_length=None, allow_blank=False)
RegexField	RegexField(regex, max_length=None, min_length=None, allow_blank=False)
SlugField	SlugField(max_length=50, min_length=None, allow_blank=False) 正则字段，验证正则模式 [a-zA-Z0-9*-]+
URLField	URLField(max_length=200, min_length=None, allow_blank=False)
UUIDField	UUIDField(format='hex_verbose') format: 1) 'hex_verbose' 如"5ce0e9a5-5ffa-654b-cee0-1238041fb31a" 2） 'hex' 如 "5ce0e9a55ffa654bcee01238041fb31a" 3）'int' - 如: "123456789012312313134124512351145145114" 4）'urn' 如: "urn:uuid:5ce0e9a5-5ffa-654b-cee0-1238041fb31a"
IPAddressField	IPAddressField(protocol='both', unpack_ipv4=False, **options)
IntegerField	IntegerField(max_value=None, min_value=None)
FloatField	FloatField(max_value=None, min_value=None)
DecimalField	DecimalField(max_digits, decimal_places, coerce_to_string=None, max_value=None, min_value=None) max_digits: 最多位数 decimal_palces: 小数点位置
DateTimeField	DateTimeField(format=api_settings.DATETIME_FORMAT, input_formats=None)
DateField	DateField(format=api_settings.DATE_FORMAT, input_formats=None)
TimeField	TimeField(format=api_settings.TIME_FORMAT, input_formats=None)
DurationField	DurationField()
ChoiceField	ChoiceField(choices) choices与Django的用法相同
MultipleChoiceField	MultipleChoiceField(choices)
FileField	FileField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)
ImageField	ImageField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)
ListField	ListField(child=, min_length=None, max_length=None)
DictField	DictField(child=)

选项参数：

参数名称	作用
max_length	最大长度
min_lenght	最小长度
allow_blank	是否允许为空

参数名称	作用
trim_whitespace	是否截断空白字符
max_value	最小值
min_value	最大值

通用参数：

参数名称	说明
read_only	表明该字段仅用于序列化输出，默认False
write_only	表明该字段仅用于反序列化输入，默认False
required	表明该字段在反序列化时必须输入，默认True
default	反序列化时使用的默认值
allow_null	表明该字段是否允许传入None，默认False
validators	该字段使用的验证器
error_messages	包含错误编号与错误信息的字典
label	用于HTML展示API页面时，显示的字段名称
help_text	用于HTML展示API页面时，显示的字段帮助提示信息