django-rest-framework

一、序列化

序列化可以把查询集和模型对象转换为json、xml或其他类型，也提供反序列化功能。，也就是把转换后的类型转换为对象或查询集。

REST框架中的序列化程序与Django `Form`和`ModelForm`类的工作方式非常相似。我们提供了一个`Serializer`类，它为您提供了一种强大的通用方法来控制响应的输出，以及一个`ModelSerializer`类，它提供了一个有用的快捷方式来创建处理模型实例和查询集的序列化程序。

1.1声明序列化器

```
from rest_framework import serializers

class xxxSerializer(serializers.Serializer):
    email = serializers.EmailField()
    content = serializers.CharField(max_length=200)
    created = serializers.DateTimeField()
```

1.2 常用field类

- 核心参数

参数名	缺省值	说明
read_only	False	表明该字段仅用于序列化输出，默认False
required	True	如果在反序列化期间未提供字段，则会引发错误。如果在反序列化期间不需要此字段，则设置为false。
default		缺省值，部分更新时不支持
allow_null	False	表明该字段是否允许传入None，默认False
validators	[]	验证器，一个函数列表，验证通不过引起serializers.ValidationError
error_messages	{}	一个错误信息字典

- 常用字段

字段名	说明
BooleanField	对应于django.db.models.fields.BooleanField
CharField	CharField(max_length=None, min_length=None, allow_blank=False,trim_whitespace=True) max_length - 验证输入包含的字符数不超过此数量。 min_length - 验证输入包含不少于此数量的字符。allow_blank- 如果设置为True则则应将空字符串视为有效值。如果设置为False则则空字符串被视为无效并将引发验证错误。默认为False。trim_whitespace- 如果设置为Truethen则会修剪前导和尾随空格。默认为True
EmailField	EmailField(max_length=None, min_length=None, allow_blank=False)
IntegerField	IntegerField(max_value=None, min_value=None) max_value 验证提供的数字是否不大于此值。min_value 验证提供的数字是否不低于此值。
FloatField	FloatField(max_value=None, min_value=None)
DateTimeField	DateTimeField(format=api_settings.DATETIME_FORMAT, input_formats=None, default_timezone=None) format格式字符串可以是显式指定格式的Python strftime格式 input_formats - 表示可用于解析日期的输入格式的字符串列表

1.3 创建Serializer对象

定义好Serializer类后，就可以创建Serializer对象了。Serializer的构造方法为：

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

说明：

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 = UserSerializer(User, 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。

- 实例

```
#models.py
class User(models.Model):
    username = models.CharField(max_length=30)
    password_hash =
models.CharField(max_length=20,db_column='password')
    age = models.IntegerField(default=0)

    class Meta:
        db_table = 'user'


#serializers.py
def validate_password(password):
    if re.match(r'\d+$',password):
        raise serializers.ValidationError("密码不能是纯数字")

class UserSerializer(serializers.Serializer):
    # id = serializers.IntegerField()
    username = serializers.CharField(max_length=30)
    password_hash = serializers.CharField(min_length=3,validators=
[validate_password])
    age = serializers.IntegerField()

    def create(self, validated_data):
```

```
        return User.objects.create(**validated_data)
    def update(self, instance, validated_data):
        instance.username =
validated_data.get('username',instance.username)
        instance.password_hash =
validated_data.get('password_hash',instance.password_hash)
        instance.age = validated_data.get('age',instance.age)
        instance.save()
        return instance
```

```
#view.py
class ExampleView(APIView):
    def get(self,request):
        # 1.查询数据
        user = User.objects.get(pk=1)
        # 2.构造序列化器
        serializer = UserSerializer(instance=user)
        # 获取序列化数据，通过data属性可以获取序列化后的数据
        print(serializer.data)
        return Response(serializer.data)
```

如果要被序列化的是包含多条数据的查询集QuerySet，可以通过添加**many=True**参数补充说明

```
data = User.objects.all()
serializer = UserSerializer(instance=data,many=True)
print(serializer.data)
```

1.4 ModelSerializer

ModelSerializer类能够让你自动创建一个具有模型中相应字段的Serializer类。这个ModelSerializer类和常规的Serializer类一样，不同的是：

- 它根据模型自动生成一组字段。
- 它自动生成序列化器的验证器，比如unique_together验证器。
- 它默认简单实现了.create()方法和.update()方法。

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        #你还可以将fields属性设置成'__all__'来表明使用模型中的所有字段。
        fields = ('id', 'account_name', 'users', 'created')
```

1.5 反向序列化

- 验证

  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或保存成模型类对象。

  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。

  验证失败，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。如果是非字段错误，可以通过修改REST framework配置中的**NON_FIELD_ERRORS_KEY**来控制错误字典中的键名。

  验证成功，可以通过序列化器对象的**validated_data**属性获取数据。

  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。

  ```
  class BookInfoSerializer(serializers.Serializer):
      """图书数据序列化器"""
      id = serializers.IntegerField(label='ID', read_only=True)
      btitle = serializers.CharField(label='名称', max_length=20)
      bpub_date = serializers.DateField(label='发布日期',
  required=False)
      bread = serializers.IntegerField(label='阅读量',
  required=False)
      bcomment = serializers.IntegerField(label='评论量',
  required=False)
      image = serializers.ImageField(label='图片', required=False)
  ```

  通过构造序列化器对象，并将要反序列化的数据传递给data构造参数，进而进行验证

```
from booktest.serializers import BookInfoSerializer
data = {'bpub_date': 123}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()  # 返回False
serializer.errors
# {'btitle': [ErrorDetail(string='This field is required.',
code='required')], 'bpub_date': [ErrorDetail(string='Date has
wrong format. Use one of these formats instead: YYYY[-MM[-DD]].',
code='invalid')]}
serializer.validated_data  # {}

data = {'btitle': 'python'}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()  # True
serializer.errors  # {}
serializer.validated_data  #  OrderedDict([('btitle', 'python')])
```

is_valid()方法还可以在验证失败时抛出异常serializers.ValidationError，可以通过传递**raise_exception=True**参数开启，REST framework接收到此异常，会向前端返回HTTP 400 Bad Request响应。

```
# Return a 400 response if the data was invalid.
serializer.is_valid(raise_exception=True)
```

如果觉得这些还不够，需要再补充定义验证行为，可以使用以下三种方法：

1）`validate_<field_name>`

对`<field_name>`字段进行验证，如

```
class BookInfoSerializer(serializers.Serializer):
    """图书数据序列化器"""
    ...

    def validate_btitle(self, value):
        if 'django' not in value.lower():
            raise serializers.ValidationError("图书不是关于Django
的")
        return value
```

2）validate

在序列化器中需要同时对多个字段进行比较验证时，可以定义validate方法来验证，如

```
class BookInfoSerializer(serializers.Serializer):
    """图书数据序列化器"""
    ...

    def validate(self, attrs):
        bread = attrs['bread']
        bcomment = attrs['bcomment']
        if bread < bcomment:
            raise serializers.ValidationError('阅读量小于评论量')
        return attrs
```

3）validators

在字段中添加validators选项参数，也可以补充验证行为，如

```
def about_django(value):
    if 'django' not in value.lower():
        raise serializers.ValidationError("图书不是关于Django的")

class BookInfoSerializer(serializers.Serializer):
    """图书数据序列化器"""
    id = serializers.IntegerField(label='ID', read_only=True)
    btitle = serializers.CharField(label='名称', max_length=20,
validators=[about_django])
    bpub_date = serializers.DateField(label='发布日期',
required=False)
    bread = serializers.IntegerField(label='阅读量',
required=False)
    bcomment = serializers.IntegerField(label='评论量',
required=False)
    image = serializers.ImageField(label='图片', required=False)
```

2. 保存

如果在验证成功后，想要基于validated_data完成数据对象的创建，可以通过实现create()和update()两个方法来实现。

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

```
    """图书数据序列化器"""
    ...

    def create(self, validated_data):
        """新建"""
        return BookInfo.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """更新，instance为要更新的对象实例"""
        instance.btitle = validated_data.get('btitle',
instance.btitle)
        instance.bpub_date = validated_data.get('bpub_date',
instance.bpub_date)
        instance.bread = validated_data.get('bread',
instance.bread)
        instance.bcomment = validated_data.get('bcomment',
instance.bcomment)
        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'
```

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

```
serializer.save(owner=request.user)
```