# django rest framework

## 教程一： Serializers *序列化*

*自动地将模型的实例对象序列化，生成json数据更方便前端使用*

*就是帮你把response打包成某种格式（如JSON)的东西。 可以根据一个model来定义一个serializer。 如一个model 叫PersonModel， 里面有name和age， PersonSerilizer 之后会是{name: Xiaoming, age: 18}这样的格式。 同理接收到的POST data，可以通过PersonSerializer(POST.data)来重建一个Person.*

*如果阅读Django-REST-Framework的文档，你会发现serializer实际上做了一件事情 serilization : isntance → native datatype → Json，将model实例的转为json格式 response出去。同理，deserializer则是Json → native datatype → isntance。 然而从 REST的设计原则看，可以知道它实际上是为了满足客户端的需求，现在的web后端与客 户端（ios/android）打交道的多，这样的格式化response更便于它们解析。*

### 1. 介绍

本教程覆盖创建一个简单的pastebin 代码高亮（code highlighting）的Web API. 接下来，本教程将会介绍各种组件来组成Rest框架。最终能够综合的理解这些组件是如何协同工作的。

**Note:**此教程的代码可以从[tomchristie/rest-framework-tutorial](tomchristie/rest-framework-tutorial)仓库获取。完整的实现也可以通过一个在线的沙盒版本呈现[点击此处](点击此处)。

### 2. 开始一个新的环境

在我们开始前，我们将创建一个新的虚拟环境，使用virtualenv.这样可以使我们的包配置与其他的项目保持隔离。

```
virtualenv env
source env/bin/activate
```

现在我们在这个虚拟环境里安装需要用的python包

```
pip install django
pip install djangorestframework
pip install pygments  # 用来制作代码高亮
```

**NOTE:**退出虚拟环境使用 `deactivate`

## 3. 开始

开始写代码。在开始之前，创建一个新的项目

```
django-admin.py startproject tutorial
cd tutorial
```

此时我们可以创建app用来创建一个简单的Web APP

```
python manage.py startapp snippets
```

我们需要添加snippets应用和rest_framework应用到INSTALLED_APPS.我们来编辑文件tutorial/settings.py

```
INSTALLED_APPS = (
    ...
    'rest_framework',
    'snippets.apps.SnippetsConfig',
)
```

好了，我们可以继续啦

## 4. 创建Model

为了达到此教程的目的，我们将要创建一个简单的Snippetmodel 用来存储代码片段（code snippets）。下面将编辑Snippet/models.py文件。**Note:**好的编程习惯是包括*书写注释*。

```
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 LEXE
STYLE_CHOICES = sorted((item, item) for item in get_all_styles())

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)
    language = models.CharField(choices=LANGUAGE_CHOICES, default
    style = models.CharField(choices=STYLE_CHOICES, default='frie

    class Meta:
        ordering = ('created',)
```

在创建完model后，我们需要为我们的snippet model进行一次初始化迁移（创建数据库），并第一次同步数据库

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

## 5. 创建一个序列化器类（Serializer class）

在开发我们的Web API的第一件事情是提供一种方法去序列化和反序列化snippet实例成一个表现形式（格式），例如json。我们可以像Django的forms一样通过声明序列化器（serializers）。在snippets目录下创建一个名为serializes.py的文件。

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

class SnippetSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    title = serializers.CharField(required=False, allow_blank=Tru
    code = serializers.CharField(style={'base_template': 'textare
    linenos = serializers.BooleanField(required=False)
    language = serializers.ChoiceField(choices=LANGUAGE_CHOICES,
    style = serializers.ChoiceField(choices=STYLE_CHOICES, defaul

    def create(self, validated_data):
        """
        Create and return a new 'Snippet' instance, given the val
        :param validated_data:
        :return:
        """
        return Snippet.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """
        update and return an existing 'Snippet' instance, given t
        :param instance:
        :param validated_data:
        :return:
        """
        instance.title = validated_data.get('title', instance.tit
        instance.code = validated_data.get('code', instance.code)
        instance.linenos = validated_data.get('linenos', instance
        instance.language = validated_data.get('language', instan
        instance.style = validated_data.get('style', instance.sty
        instance.save()
        return instance
```

序列化器类的第一部分定义了被序列化和反序列化的field。create()和

update()方法定义了一个完整的成形的实例是怎么被创建的或者在serializer.save()后如何被修改的

序列化器类与Django的Form类是很相似的，包括相似的*在各个fields中的有效标志*，比如：required，max_length，default

这些field标志还可以控制序列化器应该怎么在特定的环境下展示（displayed），例如何时渲染成html页面。上面的{'base_template': 'textarea.html'}标记相当于在DjangoForm类中使用widget=widgets.Textarea。这对于控制应该怎么展示可浏览的API是非常有用的，在我们后面的教程中会看到。

我们也可以使用ModelSerializer来代替Serializer。在后面的章节会体现。

## 6. 使用序列化器进行工作

在我们更进一步前，我们可以使用我们新的序列化类来熟悉序列化怎么工作。让我们使用Django shell来测试。

```
python manage.py shell
```

下面，在一些必要的imports后，我们来创建一些code snippets 实例

```
In [1]: from snippets.models import Snippet

In [2]: from snippets.serializers import SnippetSerializer

In [3]: from rest_framework.renderers import JSONRenderer

In [4]: from rest_framework.parsers import JSONParser

In [5]: snippet = Snippet(code='foo = "bar"\n')

In [6]: snippet.save()

In [7]: snippet = Snippet(code='print "hello, world"\n')

In [8]: snippet.save()
```

这是我们得到了一些snippet instances，下面我们来序列化其中一个实例

```
In [9]: serializer = SnippetSerializer(snippet)

In [10]: serializer.data
Out[10]:
ReturnDict([('id', 2),
            ('title', u''),
            ('code', u'print "hello, world"\n'),
            ('linenos', False),
```

```
                ('language', 'python'),
                ('style', 'friendly')])
```

此时我们获取了从model实例翻译过来的python内部数据类型。下面我们完成序列化，将数据渲染成JSON。

```
In [11]: content = JSONRenderer().render(serializer.data)

In [12]: content
Out[12]: '{"id":2,"title":"","code":"print \\"hello, world\\"\\n"
```

反序列化也是类似的。首先我们将一个流解析成python的内建数据类型

```
In [13]: from django.utils.six import BytesIO

In [14]: stream = BytesIO(content)

In [17]: data = JSONParser().parse(stream)

In [18]: data
Out[18]:
{u'code': u'print "hello, world"\n',
 u'id': 2,
 u'language': u'python',
 u'linenos': False,
 u'style': u'friendly',
 u'title': u''}
```

然后我们恢复这些内建数据类型成一个完全填充的对象实例

```
In [19]: serializer = SnippetSerializer(data=data)

In [20]: serializer.is_valid()
Out[20]: True

In [21]: serializer.validated_data
Out[21]:
OrderedDict([(u'title', u''),
             (u'code', u'print "hello, world"'),
             (u'linenos', False),
             (u'language', 'python'),
             (u'style', 'friendly')])

In [22]: serializer.save()
Out[22]: <Snippet: Snippet object>
```

我们发现这个过程与forms很相似。当我们使用serializer开始写views的时候，这个相似性会更加的明显。

我们也可以序列化querysets来代替model实例。这么做的前提是添加标志many=True到serializer参数中。

```
In [23]: serializer = SnippetSerializer(Snippet.objects.all(), ma

In [24]: serializer.data
Out[24]: [OrderedDict([('id', 1), ('title', u''), ('code', u'foo
```

## 7. 使用ModelSerializer

我们在编辑SnippetSerializer类的时候大量的复制了modelSnippet的信息。使用ModelSerializer可以使我们的代码更加简洁。

非常相似的，Django提供了Form和ModelForm类，REST framework提供了Serializer和ModelSerializer类

我们来使用ModelSerializer来重构我们的序列化器。再次打开并编辑snippets/serializers.py，使用以下代码来代替原来的SnippetSerializer类。

```
class SnippetSerializer(serializers.ModelSerializer):
    class Meta:
        model = Snippet
        fields = ('id', 'title', 'code', 'linenos', 'language', '
```

serializers 有一个很友好的属性可以在一个serializer实例中通过打印他的表现形式（representation）帮助检查所有的fields。

打开django shell尝试下列代码

```
In [1]: from snippets.serializers import SnippetSerializer

In [2]: serializer = SnippetSerializer()

In [3]: print(repr(serializer))
SnippetSerializer():
    id = IntegerField(label='ID', read_only=True)
    title = CharField(allow_blank=True, max_length=100, required=
    code = CharField(style={'base_template': 'textarea.html'})
    linenos = BooleanField(required=False)
    language = ChoiceField(choices=[('abap', 'ABAP'), ('abnf', 'A
    style = ChoiceField(choices=[('algol', 'algol'), ('algol_nu',
```

**Note:**ModelSerializer类只是创建serializer类的快捷方式，而并不会做其他特别神奇的事情。功能主要是：

- 一个自动判定fields组

- 简单的实施create()和update()方法。

## 8. 使用Serializer编辑标准的Django view

接下来我们书写一些API的视图（View）来使用我们的Serializer 类。现在我们并不使用任何其他的REST framework的其他功能，我们只编辑普通的django view。

我们通过创建HttpResponse的子类可以用来渲染所有我们要写入到JSON的数据

编辑snippets/views.py文件，添加下列内容。

```
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer

class JSONResponse(HttpResponse):
    """
    An HttpResponse that renders its content into JSON.
    """
    def __init__(self, data, **kwargs):
        content = JSONRenderer().render(data)
        kwargs['content_type'] = 'application/json'
        super(JSONResponse, self).__init__(content, **kwargs)
```

我们的API的根是成为一个支持*监听所有已经存在的snippets* 或者 *创建一个新的snippet*。

```
@csrf_exempt
def snippet_list(request):
    """
    List all code snippets, or create a new snippet
    :param request:
    :return:
    """

    if request.method == 'GET':
        snippts = Snippet.objects.all()
        serializer = SnippetSerializer(snippts, many=True)
        return JSONResponse(serializer.data)

    elif request.method == 'POST':
        data = JSONParser().parse(request)
        serializer = SnippetSerializer(data=data)
        if serializer.is_valid():
            serializer.save()
```

```
            return JSONResponse(serializer.data, status=201)
        return JSONResponse(serializer.errors, status=404)
```

**Note:**在这里因为我们的要POST到这个view的客户端没有CSRF token，所以我们将view标识为`csrf_exempt`。这并不是我们通常想要的，而且REST 框架views实际上使用了比这种方法更加敏感的行为。

在这里，我们仍然需要一个view去对应处理一个独立的snippet，能够去获取，更新，删除这个snippet

```
@csrf_exempt
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a code snippet.
    :param request:
    :param pk:
    :return:
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Exception as e:
        print e
        return HttpResponse(status=404)

    if request.method == 'GET':
        serializer = SnippetSerializer(snippet)
        return JSONResponse(serializer.data)

    elif request.method == 'PUT':
        data = JSONParser().parse(request)
        serializer = SnippetSerializer(snippet, data=data)
        if serializer.is_valid():
            serializer.save()
            return JSONResponse(serializer.data)
        return JSONResponse(serializer.errors, status=400)

    elif request.method == 'DELETE':
        snippet.delete()
        return HttpResponse(status=204)
```

之后我们需要接通这些views，创建`snippets/urls.py`文件：

```
from django.conf.urls import url
from snippets import views

urlpatterns = [
    url(r'^snippets/$', views.snippet_list),
    url(r'^snippets/(?P<pk>[0-9]+)/$', views.snippet_detail),
]
```

最后我们需要接通*根urlconf*, 在tutorial/urls.py中，包含我们的snippet app的URLs。

```
from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^', include('snippets.urls')),
]
```

值得注意的是，有一些边界问题我们并没有处理。如果我们发送了一个难看的JSON，或者如果一个requeset是被一个view无法处理的方法创建的，那么我们会被一个500的错误结束。

## 9. 验证

在终端启动Web server

```
python manage.py runserver 0.0.0.0:8000
```

我们可以使用curl或者httpie来测试我们的API。Httpie是由python写的一个友好的http客户端工具。使用前安装

```
pip install httpie
```

我们获取所有snippets的列表

```
# http http://127.0.0.1:8000/snippets/
HTTP/1.0 200 OK
Content-Type: application/json
Date: Wed, 28 Dec 2016 09:44:22 GMT
Server: WSGIServer/0.1 Python/2.7.5
X-Frame-Options: SAMEORIGIN

[
    {
        "code": "foo = \"bar\"\n",
        "id": 1,
        "language": "python",
        "linenos": false,
        "style": "friendly",
        "title": ""
    },
    {
        "code": "print \"hello, world\"\n",
```

```
            "id": 2,
            "language": "python",
            "linenos": false,
            "style": "friendly",
            "title": ""
        },
        {
            "code": "print \"hello, world\"",
            "id": 3,
            "language": "python",
            "linenos": false,
            "style": "friendly",
            "title": ""
        }
    ]
```

通过指定id获取某个snippets

```
    # http http://127.0.0.1:8000/snippets/1/
    HTTP/1.0 200 OK
    Content-Type: application/json
    Date: Wed, 28 Dec 2016 09:52:28 GMT
    Server: WSGIServer/0.1 Python/2.7.5
    X-Frame-Options: SAMEORIGIN

    {
        "code": "foo = \"bar\"\n",
        "id": 1,
        "language": "python",
        "linenos": false,
        "style": "friendly",
        "title": ""
    }
```

### 10. 现在

我们现在学习到了serialization API感觉和Django的Form API很相似，还学习到了使用普通的django view

我们的API views 目前并没有做什么特别的事情，除了服务JSON的响应（response），我们仍然有一些错误处理的边界问题需要处理。不过我们仍然完成了一个功能性的Web API。

---

# 教程二：请求和响应（request and response）

*从现在开始我们将开始真正的覆盖REST 框架的核心。让我们介绍一些基本的构建块*

## 1. Request 对象

REST 框架引入了一个Request对象，它扩展了普通的HttpRequest对象，并且提供了更负载的请求解析。Request对象的核心功能是request.data属性，与request.POST属性相似，但对于Web API更加有用。

```
request.POST # 只处理form的数据。只对'POST'方法工作

request.data # 处理任意的数据。对'POST','PUT','PATCH'起作用。
```

## 2. Response 对象

REST框架还引入了Response对象，这是一种TemplateResponse，能够处理（take）未渲染的内容（content）并且使用内容协商（negotiation）机制去判定正确的内容类型（content type）最终返回到客户端。

```
return Response(data)    # 渲染成特定的由客户端请求决定的内容类型。
```

## 3. 状态码（Status Codes ）

在views中使用数值型的HTTP状态码往往不能够很明显的来阅读代码。而且在获取了一个error code错误，并不容易引起注意。REST 框架为每一个状态码都提供了更清楚明确的标示，例如在status模块中的HTTP_400_BAD_REQUEST。使用这些标示比数值型的标示更加明智。