

django-rest-framework-tutorial-cn

Cole Smith

Published
with GitBook



Table of Contents

Introduction	0
Tutorial	1
Part 1 序列化	1.1
Part 2 请求与响应	1.2
Part 3 基于视图的类	1.3
Part 4 授权与权限	1.4
Part 5 关系与超链接	1.5
Part 6 视图集与路由	1.6

Django Rest Framework 中文版教程

测试版本

- django
 - v1.9.0
- django-rest-framework
 - v3.3.2

翻译相关

- 翻译者
 - [Eason Smith](#)

Tutorial 1: 序列化 Serialization

介绍

本教程将会通过编写简单代码来实现Web API。这个过程中, 将会介绍组成Rest框架的各种组件, 并让你深刻理解各组件是如何一起工作的。本教程相当花时间, 所以在开始本教程之前, 你应该去准备一些酒和饼干。如果你只是想快速预览, 你应该直接访问[快速开始](#)文档。

笔记: 本教程的代码在[Github](#)。完整的实现过程也是[在线](#)的, 就像用于测试的沙盒版本。

创建一个新环境

在做其他事之前, 我们会用[virtualenv](#)创建一个新的虚拟环境。这将确保我们的包配置与我们正在工作的其他项目完全隔离。

```
virtualenv env          # 创建虚拟环境, 命名: env
source env/bin/activate # 进入虚拟环境env
```

既然我们已经在虚拟环境中, 那么我们就可以安装我们依赖的包了。

```
pip install django
pip install djangorestframework
pip install pygments # 代码高亮插件
```

笔记: 任何时候只要输入 `deactivate` 就可以退出虚拟环境。更多信息, 请查看[virtualenv](#)文档。

开始

首先, 我们来创建一个新项目。

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

输完以上命令，我们就可以创建一个应用，我们将会用他来创建简单的Web API。

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

我们会添加一个新的 `snippets` 应用和 `rest_framework` 应用到 `INSTALLED_APPS`。让我们编辑 `tutorial/settings.py` 文件：

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

Ok, 我们准备下一步。

创建一个 Model

为了实现本教程的目的，我们将创建一个简单的 `Snippet` 模型，这个模型用来保存 `snippets` 代码。开始编辑 `snippets/models.py` 文件。注意：优秀的编程实践都会注释。尽管你可以在我们教程代码的仓库版本中发现注释，但是我们这里省略注释，关注代码本身。

```
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])
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='python')
    style = models.CharField(choices=STYLE_CHOICES, default='friendly')

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

我们也需要为我们的snippet模型创建一个初始迁移(initial migration), 然后第一次同步数据库。

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

创建一个序列化类 (Serializer class)

着手我们的Web API, 首先要做的就是就是, 提供一种讲我们的snippet实例[序列化/反序列化](#)成表述, 例如 `json`。我们可以通过声明序列来完成, 这些序列与Django的表单(forms)工作相似。在 `snippets` 目录创建一个新文件 `serializers.py`, 添加下列代码。

```

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

class SnippetSerializer(serializers.Serializer):
    pk = serializers.IntegerField(read_only=True)
    title = serializers.CharField(required=False, allow_blank=True)
    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='emacs')

    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

```

序列化类(serializer class)的第一部分定义了一些需要被[序列化/反序列化](#)字段。 `create()` 和 `update()` 方法定义了调用 `serializer.save()` 时成熟的实例是如何被创建和修改的。序列化类(serializer class)与Django的 表单类(Form class) 非常相似, 包括对各种字段有相似的确认标志(flag), 例如 `required`, `max_length` 和 `default`。在某些情况下, 这些字段标志也能控制序列应该怎么表现, 例如在将序列渲染成HTML时。 `{'base_template': 'textarea.html'}` 标志相当于对Django 表单(Form) 类使

用 `widget=widgets.Textarea` 。这对控制API的显示尤其有用，以后的教程将会看到。事实上，以后我们可以通过使用 `ModelSerializer` 类来节约我们的时间，但是现在为了让我们序列化定义更清晰，我们用 `Serializer` 类。

用序列化(Serializers)工作

在我们深入之前，我们需要熟练使用新的序列化列(serializer class)。然我们开始使用Django命令行吧。

```
python manage.py shell
```

Okay，让我们写一些snippets代码来使序列化工作。

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

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

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

现在我们已经有了有一些snippet实例。让我们看看如何将其中一个实例序列化。

注: Model -> Serializer

```
serializer = SnippetSerializer(snippet)
serializer.data
# {'pk': 2, 'title': u'', 'code': u'print "hello, world"\n', 'liner
```

现在，我们已经将模型实例(model instance)转化成Python原生数据类型。为了完成实例化过程，我们要将数据渲染成 `json` 。

注: Serializer -> JSON


```
content = JSONRenderer().render(serializer.data)
content
# '{"pk": 2, "title": "", "code": "print \\"hello, world\\"\\n", "i
```

反序列化也一样。首先，我们需要将流(stream)解析成Python原生数据类型...

注: stream -> json

```
from django.utils.six import BytesIO

stream = BytesIO(content)
data = JSONParser().parse(stream)
```

...然后我们要将Python原生数据类型恢复成正常的对象实例。

注: json -> serializer

```
serializer = SnippetSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# OrderedDict([('title', ''), ('code', 'print "hello, world"\n'), ('pk', 2)])
serializer.save()
# <Snippet: Snippet object>
```

可以看到，API和表单(forms)是多么相似啊。当我们用我们的序列写视图的时候，相似性会相当明显。除了将模型实例(model instance)序列化外，我们也能序列化查询集(querysets)，只需要添加一个序列化参数 `many=True`。

```
serializer = SnippetSerializer(Snippet.objects.all(), many=True)
serializer.data
# [OrderedDict([('pk', 1), ('title', u''), ('code', u'foo = "bar"\n')])]
```

使用模型序列化ModelSerializers

我们的 `SnippetSerializer` 类复制了包含 `Snippet` 模型在内的很多信息。如果我们能简化我们的代码，那就更好了。以Django提供 `表单(Form)` 类和 `模型表单(ModelForm)` 类相同的方式，REST 框架包括了 `实例化(Serializer)` 类和 `模型实例化(ModelSerializer)` 类。我们来看看用 `ModelSerializer` 类创建的序列。再次打开 `snippets/serializers.py` 文件，用下面的代码重写 `SnippetSerializer` 类。

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

序列一个非常棒的属性就是，你能够通过打印序列实例的结构(representation)查看它的所有字段。输入 `python manage.py shell` 打开命令行，然后尝试以下代码：

```
from snippets.serializers import SnippetSerializer
serializer = SnippetSerializer()
print(repr(serializer))
# SnippetSerializer():
#   id = IntegerField(label='ID', read_only=True)
#   title = CharField(allow_blank=True, max_length=100, required=True)
#   code = CharField(style={'base_template': 'textarea.html'})
#   linenos = BooleanField(required=False)
#   language = ChoiceField(choices=[('Clipper', 'FoxPro'), ('Cucur', 'Cucur')])
#   style = ChoiceField(choices=[('autumn', 'autumn'), ('borland', 'borland')])
```

记住，`ModelSerializer` 类并没有做什么有魔力的事情，它们仅仅是一个创建序列话类的快捷方式。

- 一个自动决定的字段集合。
- 简单的默认 `create()` 和 `update()` 方法的实现。

用我们的序列化来写常规的Django视图

让我们看看，使用我们新的序列化类，我们怎么写一些API视图。此刻，我们不会使用REST框架的其他特性，仅仅像写常规Django视图一样。通过创建 `HttpResponse` 的一个子类来开始，其中，我们可以用这个子类来渲染任何我们返回的 `json` 数据。编辑 `snippets/views.py` 文件，添加以下代码。

```
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)
```

我们的根API将是一个支持列出所有存在的snippets的视图，或者创建一个新的snippet对象。

```
@csrf_exempt
def snippet_list(request):
    """
    List all code snippets, or create a new snippet.
    """
    if request.method == 'GET':
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, 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=400)
```

注意，因为我们希望可以从没有 `CSRF token` 的客户端POST数据到这个视图，我们需要标记这个视图为 `csrf_exempt`。通常，你并不想这么做，并且事实上REST框架视图实用用更实用的行为而不是那样，但是目前来说，这足以到达我们的目的。我们也需要一个与单个`snippet`对象相应的视图，并且我们使用这个视图来恢复、更新或者删除这个`snippet`对象。

```
@csrf_exempt
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a code snippet.
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        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)
```

最终，我们需要用线将这些视图连起来。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),
]
```

我们也需要在根url配置文件 `tutorial/urls.py` 中添加我们的snippet应用URL。

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

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

有一些当时我们没有正确处理的边缘事件是没有价值的。如果我们发送不正确的 json 数据，或者如果我们制造了一个视图没有写处理的方法(method)，那么我们会得到500“服务器错误”的响应。当然，现在也会出现这个问题。

测试我们Web API的第一次努力

现在我们开始创建一个测试服务器来服务我们的snippets应用。 退出命令行.....

```
quit()
```

...然后启动Django开发服务器。

```
python manage.py runserver

Validating models...

0 errors found
Django version 1.8.3, using settings 'tutorial.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

我们可以在另一个终端测试服务器。 我们可以用curl和httpie来测试我们的API。Httpie是一个面向用户的非常友好的http客户端，它是用Python写的。让我们来安装它。你可以通过pip来安装httpie：

```
pip install httpie
```

最后，我们来获取一个包含所有snippets的列表：

```
http http://127.0.0.1:8000/snippets/

HTTP/1.1 200 OK
...
[
  {
    "id": 1,
    "title": "",
    "code": "foo = \"bar\"\\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  },
  {
    "id": 2,
    "title": "",
    "code": "print \"hello, world\"\\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  }
]
```

或者我们可以通过id来获取指定的snippet：

```
http http://127.0.0.1:8000/snippets/2/

HTTP/1.1 200 OK
...
{
  "id": 2,
  "title": "",
  "code": "print \"hello, world\"\\n",
  "linenos": false,
  "language": "python",
  "style": "friendly"
}
```

相似地，你可以通过在浏览器中访问这些链接来获得相同的 `json` 数据。

我们现在在哪

到目前为止，我们做的都很好，我们已经获得一个序列化API，这和Django的表单API非常相似，并且我们写好了一些常用的Django视图。现在，我们的API视图除了服务于 `json` 外，不会做任何其他特别的东西，并且有一些错误我们仍然需要清理，但是它是一个可用的Web API。我们将会在本教程的第二部分改善这里东西。

Tutorial 2: 请求与响应

从这开始，我们将接触REST框架的核心。让我们来介绍一系列必要的搭建模块。

请求对象

REST框架介绍了一个 `请求(Request)` 对象，它扩展了常规的 `HttpRequest`，并且，提供更灵活的请求解析。`请求(Request)` 对象的核心功能是 `request.data` 属性，这个属性与 `request.POST` 相似，但是它对Web APIs更加有用。

```
request.POST # 只处理表单数据。只对'POST'方法起作用。
request.data # 可以处理任意数据。对'POST', 'PUT'和'PATCH'方法起作用。
```

响应对象

REST 框架也介绍了 `Response` 对象，它是一类用未渲染内容和内容协商来决定正确内容类型并把它返回给客户端的 `模板响应(TemplateResponse)`。

```
return Response(data) # 根据客户端的请求来渲染成指定的内容类型。
```

状态码

总是在你的视图中用数字的HTTP状态码会更加容易理解，并且如果你用其他错误代码表示错误，就不太容易注意到了。REST框架为每个状态码(status code)提供更明确的标识符，例如在 `状态(status)` 模型中的 `HTTP_400_BAD_REQUEST`。用这些标识符代替纯数字的HTTP状态码是很好的注意。

装饰API视图

REST框架提供两个装饰器，你可以用它们来写API视图。

- 1 `@api_view` 装饰器用在基于视图的方法上。
- 2 `APIView` 类用在基于视图的类上。这些装饰器提供一些功能，例如去报在你的视图中接收 `Request` 对象，例如在你的 `Response` 对象中添加上下文，这样我们就能实现内容通信。这里装饰器也提供了一些行为，例如在合适的时候返回 `405 Method Not Allowed` 响应，例如处理任何在访问错误输入的 `request.data` 时出现的 `解析错误(ParseError)` 异常。

结合在一起

好了，让我们开始用这些新的组件写一些视图。我们不再需要在我们的 `视图` (`views.py`) 中使用 `JSONResponse` 类，所有现在把它删掉。一旦我们这样做了，我们就能很快重建我们的视图。

```
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer

@api_view(['GET', 'POST'])
def snippet_list(request):
    """
    List all snippets, or create a new snippet.
    """
    if request.method == 'GET':
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = SnippetSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

我们的实例视图是我们之前例子的改良版。简明了很多，并且目前的代码和我们使用Forms API很相似。我们也用有意义的状态码标识符。在 `views.py` 模块中，有一个独立的snippet视图。

```
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a snippet instance.
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

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

    elif request.method == 'PUT':
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_F)

    elif request.method == 'DELETE':
        snippet.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

这对我们来说应该非常熟悉，因为它与常规的Django视图没有什么区别。注意，我们不再明确打印我们的对指定内容类型的请求或响应。`request.data` 能够处理 `json` 请求，但是它也能处理其他格式。相似地，虽然我们可以在响应对象中带数据，但允许REST框架渲染响应成正确的内容类型。

在我们的链接(URLs)后添加可选格式后缀

为了利用我们的响应内容不再是单一格式的事实，我们应该为我们的API尾部添加格式后缀。用格式后缀给我们明确参考指定格式的URL，这意味着我们的API能够处理像 `http://example.com/api/items/4/.json` 一样的链接。在视图函数中添加一个 `format` 参数，像这样：

```
def snippet_list(request, format=None):
```

和

```
def snippet_list(request, pk, format=None):
```

现在可一很快更新 `urls.py` 文件，在已经存在的URL中添加一个 格式后缀模式 (`format_suffix_patterns`)。

```
from django.conf.urls import url
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

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

urlpatterns = format_suffix_patterns(urlpatterns)
```

我们不必添加额外的URL模式，但是它给我们简单、清楚的方式渲染除特定的格式。

看看吧


和[教程第一部分](#)一样，我们要开始从命令行测试API。虽然我们能在发送无效的请求时更妥当处理错误，但是现在一切都做的够好了。我们能想之前一样获取所有的 `snippets` 列表。

```
http http://127.0.0.1:8000/snippets/

HTTP/1.1 200 OK
...
[
  {
    "id": 1,
    "title": "",
    "code": "foo = \"bar\"\\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  },
  {
    "id": 2,
    "title": "",
    "code": "print \"hello, world\"\\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  }
]
```

我们能控制我们返回的响应格式，或者使用 `Accept` 响应头。

```
http http://127.0.0.1:8000/snippets/ Accept:application/json # Rec
http http://127.0.0.1:8000/snippets/ Accept:text/html # Rec
```



或者在URL后添加格式后缀：

```
http http://127.0.0.1:8000/snippets.json # JSON 后缀
http http://127.0.0.1:8000/snippets.api # 浏览用的 API 后缀
```

同样地，我们可以控制我们发送的请求格式，用 `Content-Type` 请求头。

```
# POST using form data
http --form POST http://127.0.0.1:8000/snippets/ code="print 123"

{
  "id": 3,
  "title": "",
  "code": "print 123",
  "linenos": false,
  "language": "python",
  "style": "friendly"
}

# POST using JSON
http --json POST http://127.0.0.1:8000/snippets/ code="print 456"

{
  "id": 4,
  "title": "",
  "code": "print 456",
  "linenos": false,
  "language": "python",
  "style": "friendly"
}
```

你也可以从浏览器打开API，通过访问<http://127.0.0.1:8000/snippets/>。

Browsable

因为API是基于客户端请求来选择响应内容的类型，所以默认情况下，在Web浏览器访问资源时，API返回HTML格式的资源。这使API返回完全可以网页浏览的HTML。有可以网页浏览API是很好的，这使开发和使用你的API更简单，这也为其他想要查看和使用你的API的开发者大大降低了门槛。关于可浏览API的特性和如何自定义可浏览API，请见[可浏览API](#)话题。

接下来要干什么？

在教程的第三部分，我们基于视图用类，并且看看普通的视图我们如何减少代码。

Tutorial 3: 基于视图的类

除了可以用基于视图的函数写我们的API，我们也可以用基于视图的类。正如我们所见，这是一个非常有利的模式，允许我们重用同样的功能，并帮助我们使代码紧凑。

用基于视图的类重写我们的API

我们将会想重写一个基于视图的类一样重写根视图。这包括重构 `views.py` 文件。

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from django.http import Http404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status

class SnippetList(APIView):
    """
    List all snippets, or create a new snippet.
    """
    def get(self, request, format=None):
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = SnippetSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```


到目前为止，一切都很好。这和之前的情况很相似，但是我们已经很好地通过不同的HTTP方法区分。现在我們也需要在 `views.py` 中更新实例视图。

```
class SnippetDetail(APIView):
    """
    Retrieve, update or delete a snippet instance.
    """
    def get_object(self, pk):
        try:
            return Snippet.objects.get(pk=pk)
        except Snippet.DoesNotExist:
            raise Http404

    def get(self, request, pk, format=None):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet)
        return Response(serializer.data)

    def put(self, request, pk, format=None):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_F)

    def delete(self, request, pk, format=None):
        snippet = self.get_object(pk)
        snippet.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

那看起来不错。再次强调，这和基于视图的函数非常相似。我们也需要用基于视图的类重构我们的 `urls.py`。

```
from django.conf.urls import url
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

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

urlpatterns = format_suffix_patterns(urlpatterns)
```

好了，我们做完了。如果你启用开发服务器，那么一切都和之前一样。

使用混合(mixins)

使用基于视图的类最大的一个好处是，它允许我们快速创建可复用的行为。我们一直使用的 `create/retrieve/update/delete` 操作将和我们创建的任何后端模型 API 视图非常相似。这些普遍的行为是通过 REST 框架的混合类(mixin classes)实现的。让我们看看如何通过混合类(mixin classes)组建视图。下面是我们的 `views.py` 模型。

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import mixins
from rest_framework import generics

class SnippetList(mixins.ListModelMixin,
                  mixins.CreateModelMixin,
                  generics.GenericAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```

我们会花一会儿准确测试这里发生了什么。我们使用 `GenericAPIView` 加上 `ListModelMixin` 和 `CreateModelMixin` 创建我们的视图。基类提供核心功能，混合类提供 `.list()` 和 `.create()` 动作。然后我们合适的动作绑定明确的 `get` 和 `post` 方法。到目前为止，东西已经足够简单。

```
class SnippetDetail(mixins.RetrieveModelMixin,
                    mixins.UpdateModelMixin,
                    mixins.DestroyModelMixin,
                    generics.GenericAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

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

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

太像了。我们用 `GenericAPIView` 类提供核心功能，添加混合(mixin)，来提供 `.retrieve()`，`.update()` 和 `.destroy()` 动作。

使用基于视图的一般类(generic class)

尽管我们已经使用混合类(mixin classes)以比之前更少的代码重写了视图，但是我们可以进一步深入。REST框架提供一个已经混入一般视图的集合，我们能用来整理我们的 `views.py` 模块。

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics

class SnippetList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer

class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
```

哇，如此简洁。我们的代码看起来是如此简洁、地道的Django。接下来我们要学习[本教程的第四部分](#)，在第四部分我们会为我们的API处理授权(authentication)和权限(permissions)。

Tutorial 4: 授权(Authentication)与权限(Permissions)

当前，我们的API没有限制谁能编辑或删除snippets代码。我们想要一些更高级的行为以确保：

- snippets数据总是与创建者联系在一起。
- 只有授权用户才能创建snippets。
- 只有snippet的创建者才能更新或者删除它。
- 没有授权的请求应该只有只读权限。

在我们的模型中添加信息

我们打算对我们的 `Snippet` 模型类做些改变。首先，让我们添加几个字段。其中一个字段将显示出哪个用户创建里snippet数据。另一个字段将用于HTML代码高亮。

```
owner = models.ForeignKey('auth.User', related_name='snippets')
highlighted = models.TextField()
```

我们也需要确保模型什么保存了，为此我们用 `pygments` 代码高亮库来形成高亮字段。我们需要一些额外的引用：

```
from pygments.lexers import get_lexer_by_name
from pygments.formatters.html import HtmlFormatter
from pygments import highlight
```

然后给我们的模型类添加 `.save()` 方法：

```
def save(self, *args, **kwargs):
    """
    Use the `pygments` library to create a highlighted HTML
    representation of the code snippet.
    """
    lexer = get_lexer_by_name(self.language)
    linenos = self.linenos and 'table' or False
    options = self.title and {'title': self.title} or {}
    formatter = HtmlFormatter(style=self.style, linenos=linenos,
                              full=True, **options)
    self.highlighted = highlight(self.code, lexer, formatter)
    super(Snippet, self).save(*args, **kwargs)
```

然后，我们需要更细我们的数据库表。为此，正常情况下，我们会创建数据库迁移(database migration)，但是就本教程来说，我们只需要删除原来的数据库，然后重新创建即可。

```
rm -f tmp.db db.sqlite3
rm -r snippets/migrations
python manage.py makemigrations snippets
python manage.py migrate
```

你可能也想要创建不同的用户来测试API。最快的方式就是用 `createsuperuser` 命令。

```
python manage.py createsuperuser
```

为我们的用户模型添加端点

既然我们已经创建了多个用户，那么我们最好将用户添加到我们的API。很容易创建一个新的序列。在 `serializers.py` 中添加；

```

from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):
    snippets = serializers.PrimaryKeyRelatedField(many=True, querys

    class Meta:
        model = User
        fields = ('id', 'username', 'snippets')

```

因为 'snippets' 在用户模型中是一个相反的关系，默认情况下在使用 `ModelSerializer` 类时我们不会包括，所以我们需要手动为用户序列添加这个字段。我们需要添加在 `views.py` 中添加一些视图。我们想要为用户添加只读视图，所以我们会使用基于视图的一般类 `ListAPIView` 和 `RetrieveAPIView`。

```

from django.contrib.auth.models import User

class UserList(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer

class UserDetail(generics.RetrieveAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer

```

确保文件中引入了 `UserSerializer` 类。

```

from snippets.serializers import UserSerializer

```

最后，我们需要通过修改URL配置，将这些视图添加进API。添加以下 `urls.py` 中。

```

url(r'^users/$', views.UserList.as_view()),
url(r'^users/(?P<pk>[0-9]+)/$', views.UserDetail.as_view()),

```

将用户和Snippets连接起来

现在，如果我们创建snippet数据，我们没办法将用户和snippet实例联系起来。虽然用户不是序列表示的部分，但是它是请求的一个属性。我们通过重写snippet视图的 `.perform_create()` 方法来做到，这个方法允许我们修改如何保存实例，修改任何请求对象或者请求连接里的信息。在 `SnippetList` 视图类中添加以下方法：

```
def perform_create(self, serializer):
    serializer.save(owner=self.request.user)
```

现在，我们序列的 `create()` 方法将会另外传入一个来自有效的请求数据的 `'owner'` 字段。

更新我们的序列

既然已经将snippets和创建它们的用户联系在一起了，那么我们需要更新对应的 `SnippetSerializer`。在 `serializers.py` 的序列定义(serializer definition)中添加以下字段：

```
owner = serializers.ReadOnlyField(source='owner.username')
```

注意；去报你也添加 `'owner'`，到内部类 `Meta` 的字段列表里。这个字段很有趣。`source` 参数控制哪个属性被用于构成一个字段，并且能够指出序列实例的任何属性。它也能想上面一样使用点标记(`.`)，这中情况下他会横贯给定的属性，就是我们使用Django模板语言一样。我们添加的字段是隐式 `ReadOnly` 类，与其他类相反，如 `CharField`，`BooleanField`，隐式 `ReadOnlyField` 总是只读的，用于序列化表示，但在数据非序列化时不能用于更新实例。这里我们也可以使用 `CharField(read_only=True)`。

为视图添加需要的权限

snippets数据已经和用户联系在一起，我们想确保只有授权的用户可以创建、更新和删除snippet数据。REST框架包括许多权限类(permission classes)，我们可以使用这些权限类来现在视图的访问权限。这种情况下，其中我们需

要 `IsAuthenticatedOrReadOnly`，这个类确保授权请求有读写权限，而没有授权的用户只有只读权限。首先，在视图模块中引入以下代码

```
from rest_framework import permissions
```

然后，在 `SnippetList` 和 `SnippetDetail` 视图类中添加以下属性。

```
permission_classes = (permissions.IsAuthenticatedOrReadOnly, )
```

在浏览器API中添加登录

如果你现在用浏览器打开API，你会发现你已经不能创建新的snippets数据。为此，我们需要以用户身份登录。为了使用浏览器打开API，我们需要添加一个登录视图，编辑URL配置(URLconf)文件 `urls.py` 文件。在 `urls.py` 顶部添加下面import。

```
from django.conf.urls import include
```

并且，在 `urls.py` 底部为API添加一个包括登录和退出视图的url样式。

```
urlpatterns += [  
    url(r'^api-auth/', include('rest_framework.urls',  
                               namespace='rest_framework')),  
]
```

url样式的 `r'^api-auth/'` 部分实际上可以是任何你想要的URL。唯一的限制就是include的链接必须使用 `'rest_framework'` 名字空间。在Django 1.9+，REST框架会设置名字空间，所以你必须写。现在如果你刷新浏览器页面，你会看到右上角的'Login'链接。如果你用之前创建的用户登录，你就可以再次写snippets数据了。一旦你创建snippets数据，浏览'/users/'，然后你会发现每个用户的'snippets'字段，显示的内容包括与每个用户相关的snippets主键。

对象等级权限

虽然我们真的想任何人都和一看见snippets数据，但也要确保只有创建snippet的用户可以修改或删除他的snippet。为此，我们需要创建自定义权限。在snippets app中，创建一个新文件 `permissions.py`。

```
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    Custom permission to only allow owners of an object to edit it.
    """

    def has_object_permission(self, request, view, obj):
        # Read permissions are allowed to any request,
        # so we'll always allow GET, HEAD or OPTIONS requests.
        if request.method in permissions.SAFE_METHODS:
            return True

        # Write permissions are only allowed to the owner of the snippet
        return obj.owner == request.user
```

然后编辑 `SnippetDetail` 视图类中的 `permission_classes` 属性，添加自定义权限。

```
permission_classes = (permissions.IsAuthenticatedOrReadOnly,
                      IsOwnerOrReadOnly,)
```

确保引入了 `IsOwnerOrReadOnly` 类。

```
from snippets.permissions import IsOwnerOrReadOnly
```

现在，如果你再次打开浏览器，你会发现只有你登入，你才能删除(DELETE)或更新(PUT)属于你的snippet数据。

授权API

因为我们的API有一系列权限，所以如果我们想编辑任何snippets，我们需要授权我们的请求。我们现在还没有任何 授权类(authentications classes)，所以默认情况下只有 SessionAuthentication 和 BasicAuthentication。当我们通过Web浏览器与API交互时，我们可以登录，然后浏览器会话(session)将会提供必须的请求授权。如果我们通过程序与API交互，我们需要为每个请求提供明确的授权证明。如果我们在没有授权的情况下创建一个snippet，那么我们会得到下面的错误：

```
http POST http://127.0.0.1:8000/snippets/ code="print 123"

{
  "detail": "Authentication credentials were not provided."
}
```

为了请求成功，我们需要包含用户名和密码。

```
http -a tom:password123 POST http://127.0.0.1:8000/snippets/ code='

{
  "id": 5,
  "owner": "tom",
  "title": "foo",
  "code": "print 789",
  "linenos": false,
  "language": "python",
  "style": "friendly"
}
```

总结

现在我们已经我们的Web API上，为我们的系统用户和snippet的创建者，添加了很多权限和端点。在[第五部分](#)，我们将会看怎么我们可以通过为的高亮 snippets 创建HTML端点来将所有东西联系在一起，然后在系统内用超链接将我们的API联系起来。

Tutorial 5: 关系(Relationships)与超链接API(Hyperlinked APIs)

现在，用主键代表我们API之间的关系。在这部分教程，我们会用超链接改善API之间的关系。

为我们的API根创建一个端点

现在，我们已经为'snippets'和'users'设置了端点，但是我们没有为我们的API设置单独的入口点。因此，我们会一个基于方法的常规视图和 `@api_view` 装饰器来创建一个入口点。在你的 `snippets/views.py` 中添加：

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework.reverse import reverse

@api_view(('GET',))
def api_root(request, format=None):
    return Response({
        'users': reverse('user-list', request=request, format=format),
        'snippets': reverse('snippet-list', request=request, format=format)
    })
```

我们会注意到两件事：第一，我们用了REST框架的 `reverse` 方法为了返回高质量的URL；第二，URL格式是方便的名字标识符，我们会在之后会在 `snippets/urls.py` 中声明。

创建一个高亮的snippets端点

另一件明显的事就是，我们的API缺乏代码高亮端点。和我们所有的API端点不一样，我们不想用JSON，而只是想用HTML显示。REST框架提供两种HTML渲染样式，一种是用模板渲染处理HTML，另一种是用预渲染HTML。第二种是我们想要

用的方式。在创建代码时，我们需要考虑的是，高亮视图在我们使用的普通视图中是不存在的。我们不会返回一个对象实例，而是对象实例的一个属性。我们会是使用基类代表实例，并创建我们自己的 `.get()` 方法，而不是用普通的视图。在你的 `snippets/views.py` 添加：

```
from rest_framework import renderers
from rest_framework.response import Response

class SnippetHighlight(generics.GenericAPIView):
    queryset = Snippet.objects.all()
    renderer_classes = (renderers.StaticHTMLRenderer,)

    def get(self, request, *args, **kwargs):
        snippet = self.get_object()
        return Response(snippet.highlighted)
```

通常，我们需要添加新的视图到我们的URL配置。然后，在 `snippet/urls.py` 中添加一个链接：

```
url(r'^$', views.api_root),
```

然后，为高亮snippet添加一个url样式：

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

为我们的API添加超链接

处理好实体之间的关系是Web API设计中极具挑战性的方面之一。代表一种关系可以有多种方式：

- 使用主键。
- 在实体之间使用超链接。
- 在相关的实体上使用独一无二的slug。
- 使用相关的实体的默认字符串。
- 在父表述使用嵌套的实体。

- 一些自定义的表述。

REST框架支持以上所有方式，都能适应正向或者反向关系，或者就行使用一般的外键一样使用自定义的管理方式。

这种情况下，我们想要在实体之间使用超链接方式。为了达到目的，我们需要修改我们的序列(serializers)，以拓展 `HyperlinkedModelSerializer`，不是使用已经存在的 `ModelSerializer`。

以下是 `HyperlinkedModelSerializer` 不同于 `ModelSerializer` 的地方：

- `HyperlinkedModelSerializer` 默认不包括 `pk` 字段。
- 它只包括一个 `url` 字段，使用 `HyperlinkedIdentityField`。
- 关系使用 `HyperlinkedRelatedField`，而不是 `PrimaryKeyRelatedField`。我们能使用超链接快速重写现存的序列。在 `snippets/serializers.py` 中添加：

```
class SnippetSerializer(serializers.HyperlinkedModelSerializer):
    owner = serializers.ReadOnlyField(source='owner.username')
    highlight = serializers.HyperlinkedIdentityField(view_name='snippets-detail', url_kw='highlight')

    class Meta:
        model = Snippet
        fields = ('url', 'highlight', 'owner', 'title', 'code', 'linenos', 'language', 'style')

class UserSerializer(serializers.HyperlinkedModelSerializer):
    snippets = serializers.HyperlinkedRelatedField(many=True, view_name='snippets-list', url_kw='user')

    class Meta:
        model = User
        fields = ('url', 'username', 'snippets')
```

注意，我们已经添加了一个新字段 `highlight`。这个字段类型是和 `url` 一样的，只是它指向 `snippet-highlight` url模式，而不是 `snippet-detail` url模式。因为我们已经包含了格式后缀的URL，如 `.json`，所以我們也需要在 `highlight` 字段指明，任何格式后缀超链接应该用 `.html` 后缀。

确保我们的URL模式是有名字的

如果我们想要超链接的API，那么我们要保证我们给URL起了名字。让我们看看我们需要命名哪个链接。

- 我们API根指向 `user-list` 和 `snippet-list` 。
- 我们的snippet序列包括一个指向 `snippet-highlight` 的字段。
- 我们的用户序列包括一个指向 `snippet-detail` 的字段。
- 我们的snippet和用户序列包括 `url` 字段，这个字段默认指向 `'{model_name}-detail'`，这种情况下，它是 `snippet-detail` 和 `user-detail`。在将那些名字加入我们的URL配置(URLconf)后，我们的 `snippets/urls.py` 应该是下面的样子：

```
from django.conf.urls import url, include
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

# API endpoints
urlpatterns = format_suffix_patterns([
    url(r'^$', views.api_root),
    url(r'^snippets/$',
        views.SnippetList.as_view(),
        name='snippet-list'),
    url(r'^snippets/(?P<pk>[0-9]+)/$',
        views.SnippetDetail.as_view(),
        name='snippet-detail'),
    url(r'^snippets/(?P<pk>[0-9]+)/highlight/$',
        views.SnippetHighlight.as_view(),
        name='snippet-highlight'),
    url(r'^users/$',
        views.UserList.as_view(),
        name='user-list'),
    url(r'^users/(?P<pk>[0-9]+)/$',
        views.UserDetail.as_view(),
        name='user-detail')
])

# Login and logout views for the browsable API
urlpatterns += [
    url(r'^api-auth/', include('rest_framework.urls',
                               namespace='rest_framework')),
]
```

添加分页

用户和snippet的列表视图会返回很多实例，所以我们想要给这些结果分页，分页后允许API客户端访问每个单页。我们可以用分页改变默认的列表风格，只要稍微修改 `tutorial/settings.py` 文件。添加下面设置：


```
REST_FRAMEWORK = {  
    'PAGE_SIZE': 10  
}
```

注意：REST框架的分页设置(settings)是一个单独的字典，叫'REST_FRAMEWORK'，它可以帮我们区分项目中的其他配置。如果我们需要，我们可以自定义分页样式，但是现在我们只是用默认的。

浏览API

如果我们打开浏览器访问API，那么你会发现你可以通过下面的链接使用API。你也可以看见snippet实例的 高亮(highlight) 链接，这些链接会返回高亮HTML代码。在本教程的第六部分，我们会用 ViewSets 和 Routers 来减少我们API的代码量。

Tutorial 6: 视图集(ViewSets)和路由(Routers)

REST框架包括对 `ViewSet` 的简短描述，这可以让开发者把精力集中在构建状态和交互的API模型，而且它可以基于一般规范自动构建URL。 `ViewSet` 类几乎和 `View` 类一样，除了他们提供像 `read` 或 `update` 的操作，而不是像 `get` 和 `put` 的方法。目前，一个 `ViewSet` 类只绑定一个方法的集合，当它初始化一个视图的集合时，一般使用为你处理复杂的URL定义的 `Router` 类。

使用视图集(ViewSets)重构

让我们来用视图集重写当前视图。首先，我们要把我们的 `UserList` 和 `UserDetail` 视图重写成单个 `UserViewSet`。我们可以用 `UserViewSet` 代替 `UserList` 和 `UserDetail`。

```
from rest_framework import viewsets

class UserViewSet(viewsets.ReadOnlyModelViewSet):
    """
    This viewset automatically provides `list` and `detail` actions
    """
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

这里我们使用 `ReadOnlyModelViewSet` 类自动提供默认的'只读'操作。当我们使用常规视图的时候，我们仍然需要设置准确设置 `queryset` 和 `serializer_class` 属性，但是我们不再需要为两个分开的类提供同样的信息。接下来，我们将用 `SnippetHighlight` 视图类来代替 `SnippetList` 和 `SnippetDetail`。我们可以用一个类代替之前的三个类。

```
from rest_framework.decorators import detail_route

class SnippetViewSet(viewsets.ModelViewSet):
    """
    This viewset automatically provides `list`, `create`, `retrieve`,
    `update` and `destroy` actions.

    Additionally we also provide an extra `highlight` action.
    """
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,
                          IsOwnerOrReadOnly,)

    @detail_route(renderer_classes=[renderers.StaticHTMLRenderer])
    def highlight(self, request, *args, **kwargs):
        snippet = self.get_object()
        return Response(snippet.highlighted)

    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

这次我们使用 `ModelViewSet` 类是为了获得完整的默认读写操作的集合。注意：我们也用了 `@detail_route` 装饰器来创建自定义动作，命名为 `highlight`。这个装饰器用于添加任何自定义的端点，这些端点不符合标准的 `create/update/delete` 方式。使用 `@detail_route` 装饰器的自定义动作会响应 GET 请求。如果我们让动作响应 POST 请求，我们可以使用 `methods` 参数。自定义动作的URL在默认情况下是依赖于方法本身。如果你想改变url本来创建的方式，你可以将`url_path`包含在装饰器关键参数中。

明确绑定视图集到URL

我们定义`URLConf`的时候，处理方法只绑定了动作。为了看看发生了什么，我们必须从我们的视图集(ViewSets)创建一个视图集合。在 `urls.py` 文件中，我们将 `ViewSet` 类绑定到具体视图的集合。

```
from snippets.views import SnippetViewSet, UserViewSet, api_root
from rest_framework import renderers

snippet_list = SnippetViewSet.as_view({
    'get': 'list',
    'post': 'create'
})
snippet_detail = SnippetViewSet.as_view({
    'get': 'retrieve',
    'put': 'update',
    'patch': 'partial_update',
    'delete': 'destroy'
})
snippet_highlight = SnippetViewSet.as_view({
    'get': 'highlight'
}, renderer_classes=[renderers.StaticHTMLRenderer])
user_list = UserViewSet.as_view({
    'get': 'list'
})
user_detail = UserViewSet.as_view({
    'get': 'retrieve'
})
```

注意我们如何通过绑定http方法到每个视图需要的动作来从 `ViewSet` 类创建多视图。既然我们已经绑定了我们的资源和具体视图，我们就可以和以前一样将我们的视图注册到URL配置中。

```
urlpatterns = format_suffix_patterns([
    url(r'^$', api_root),
    url(r'^snippets/$', snippet_list, name='snippet-list'),
    url(r'^snippets/(?P<pk>[0-9]+)/$', snippet_detail, name='snippet-detail'),
    url(r'^snippets/(?P<pk>[0-9]+)/highlight/$', snippet_highlight, name='snippet-highlight'),
    url(r'^users/$', user_list, name='user-list'),
    url(r'^users/(?P<pk>[0-9]+)/$', user_detail, name='user-detail')
])
```

使用路由

因为我们使用 `ViewSet` 类而不是 `View` 类，所以实际上我们不需要自己设计URL配置。按惯例，使用 `Router` 类就可以自动将资源与视图(`views`)、链接(`urls`)联系起来。我们需要做的只是用一个路由注册合适的视图集合。现在，我们把剩下的做完。我们重写了 `urls.py` 文件。

```
from django.conf.urls import url, include
from snippets import views
from rest_framework.routers import DefaultRouter

# Create a router and register our viewsets with it.
router = DefaultRouter()
router.register(r'snippets', views.SnippetViewSet)
router.register(r'users', views.UserViewSet)

# The API URLs are now determined automatically by the router.
# Additionally, we include the login URLs for the browsable API.
urlpatterns = [
    url(r'^$', include(router.urls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

用路由注册视图和提供一个`urlpatterns`是相似的，包括两个参数--视图的URL前缀和视图本身。我们使用的 默认路由(`DefaultRouter`) 类会自动为我们创建API根视图，所以我们可以从我们的 `views` 模块删除 `api_root` 方法。

views和viewsets的比较

使用视图集(`viewsets`)真的很有用。它保证URL规范存在你的API中，让你写最少的代码，允许你把注意力集中在你的API提供的交互和表现上而不需要特定的URL配置。这并不意味着这样做总是正确的。在使用基于类的视图代替基于函数的视图时，我们总会发现 `views` 与 `viewsets` 有相似的地方。使用视图集(`viewsets`)没有比你自己的视图更清晰。

回顾

难以置信，用这么少的代码，我们已经完成了一个Web API，它是完全可浏览的，拥有完整的授权(authentication)、每个对象权限(per-object permissions)和多重渲染格式(multiple renderer formats)。我们已经经历了设计过程的每一步，看到了如果我们只是使用常规的Django视图自定义任何东西。你可以回顾Github上的[教程代码](#)，或者在[沙箱](#)测试例子。

继续向前

我们已经到了教程的结尾。如果你想了解更多关于REST框架项目，你可以从看看下面：

- 可以在[Github](#)通过评论、提交问题、拉取请求来为项目做贡献。
- 加入[REST框架讨论组](#)，帮助搭建社区。
- 在Twitter上关注并问候[作者](#)。

现在，做一些很棒的东西吧。