Table of Contents

Django-REST-framework教程中文版

django-rest-framework,是一套基于Django的REST框架，目前仅计划翻译教程部分，至于API部分时间充足的话也会进行翻译。本文仅是个人阅读文档的产物，不准确的地方还请大家指正。

另外各位看官请注意，本文基于**Django1.9**以及**restframework-v3.3.3**版本,很可能您阅读本文时候官网文档已经产生变化了。

更多翻译文章、技术文章请移步本人博客。

2016-06-24

快速入门

在这里我们创建一个简单的API，让管理员查看、编辑用户和组信息。

项目设置

新建名为 `tutorial` 的django项目并在其中建立一个名为 `quickstart` 的APP：

```
# 新建目录
mkdir tutorial
cd tutorial

# 新建虚拟环境
virtualenv env
source env/bin/activate  # Windows使用 `env\Scripts\activate`

# 在虚拟环境中安装依赖
pip install django
pip install djangorestframework

# 新建项目
django-admin.py startproject tutorial .  # 注意后面的 '.'
cd tutorial
django-admin.py startapp quickstart
cd ..
```

使用下面的命令创建表：

```
python manage.py migrate
```

然后创建一个用户名为 `admin` 密码 `password123` 的管理员：

```
python manage.py createsuperuser
```

以上设置完成后，进入APP的目录来编写代码...

序列化

首先我们创建一个文件 `tutorial/quickstart/serializers.py` 来编写序列化相关的代码：

```
from django.contrib.auth.models import User, Group
from rest_framework import serializers


class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ('url', 'username', 'email', 'groups')


class GroupSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Group
        fields = ('url', 'name')
```

注意在上面的代码中我们使用了 `HyperlinkedModelSerializer` 来建立超链接关系，你也可以使用主键或其他关系，但hyperlinking是一个好的RESTful设计。

Views

现在让我们来编写视图文件 `tutorial/quickstart/views.py` ：

```
from django.contrib.auth.models import User, Group
from rest_framework import viewsets
from tutorial.quickstart.serializers import UserSerializer, Grou
pSerializer


class UserViewSet(viewsets.ModelViewSet):
    """
    查看、编辑用户的界面
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserSerializer


class GroupViewSet(viewsets.ModelViewSet):
    """
    查看、编辑组的界面
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
```

我们把许多常见的操作都封装在了类 `ViewSets` 中，这样就不用编写重复代码了。当然你可以按照自己的需求编写view，但使用 `ViewSets` 可以保持view代码的简洁以及逻辑的清晰。

URLs

接下来编写 `tutorial/urls.py` ：

```
from django.conf.urls import url, include
from rest_framework import routers
from tutorial.quickstart import views

router = routers.DefaultRouter()
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)

# 使用URL路由来管理我们的API
# 另外添加登录相关的URL
urlpatterns = [
    url(r'^', include(router.urls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace=
'rest_framework'))
]
```

因为我们使用了 `ViewSets` ，所以我们可以通过使用Router类来自动生成URL配置信息。

重申一次，如果需要更自主的配置URL，可以使用常规的类视图以及显式编写URL配置。

最后我们添加了默认的登录、登出视图在浏览API时候使用，这是一个可选项，但如果你想在浏览API时使用认证功能这是非常有用的。

Settings

我们还需要进行一些全局设置。我们想启用分页功能以及只有管理员能访问，编辑 `tutorial/settings.py` ：

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

REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': ('rest_framework.permissions.IsAdminUser',),
    'PAGE_SIZE': 10
}
```

至此，我们完成了全部工作。

测试

现在我们来测试我们的API，在终端中输入：

```
python ./manage.py runserver
```

现在我们可以使用命令行工具访问API了，比如 `curl`：

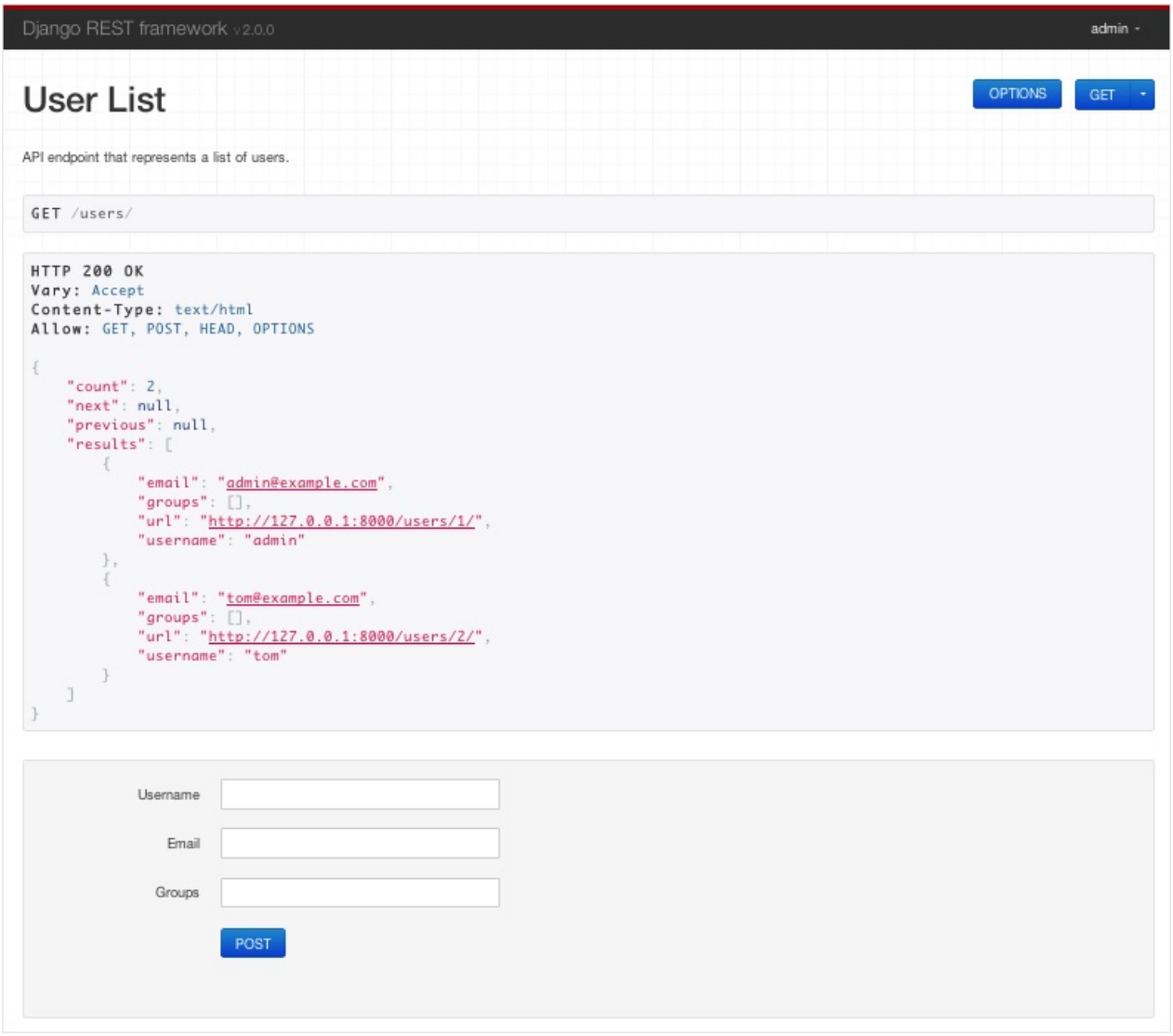
```
curl -H 'Accept: application/json; indent=4' -u admin:password12
3 http://127.0.0.1:8000/users/
{
    "count": 2,
    "next": null,
    "previous": null,
    "results": [
        {
            "email": "admin@example.com",
            "groups": [],
            "url": "http://127.0.0.1:8000/users/1/",
            "username": "admin"
        },
        {
            "email": "tom@example.com",
            "groups": [                ],
            "url": "http://127.0.0.1:8000/users/2/",
            "username": "tom"
        }
    ]
}
```

或者 `httpie`

```
http -a admin:password123 http://127.0.0.1:8000/users/

HTTP/1.1 200 OK
...
{
    "count": 2,
    "next": null,
    "previous": null,
    "results": [
        {
            "email": "admin@example.com",
            "groups": [],
            "url": "http://localhost:8000/users/1/",
            "username": "paul"
        },
        {
            "email": "tom@example.com",
            "groups": [                ],
            "url": "http://127.0.0.1:8000/users/2/",
            "username": "tom"
        }
    ]
}
```

或者直接打开浏览器：



如果使用浏览器，请确保已经使用右上角的登录功能登录。

好极了，就是这么简单！

教程1：序列化

在这个教程中将创建一个支持代码高亮展示的API。我们会介绍组成REST framework的各个组件，并且让你明白这些组件是如何相互协作的。 这篇教程会比较深入，所以开始之前你应该准备好零食和啤酒。如果你仅仅想大概了解，请看快速入门。

提示：这篇教程的源码可以在tomchristie/rest-framework-tutorial找到，也提供了在线版供大家测试，请点击这里。

设置一个新虚拟环境

首先我们使用 `virtualenv` 创建一个虚拟环境，这样可以很好的和其他项目隔离所需的依赖库：

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

然后在虚拟环境中安装需要的依赖：

```
pip install django
pip install djangorestframework
pip install pygments # 这个用于语法高亮
```

注意使用 `deactive` 来退出虚拟环境，更多信息请看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
```

接下来创建一个APP：

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

然后编辑 `tutorial/settings.py` ，把我们的 `snippets` 和 `rest_framework` 添加到 `INSTALLED_APPS` ：

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

创建Model

首先我们创建一个 `Snippet` 模型来存储代码片段。注意：写注释是一个好编程习惯，不过为了专注于代码本身，我们在下面省略了注释。编辑 `snippets/models.py` ：

```
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 LEX
ERS])
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, defaul
t='python', max_length=100)
    style = models.CharField(choices=STYLE_CHOICES, default='fri
endly', max_length=100)

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

接下来在数据库中建表：

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

创建Serializer类

第一步我们需要为API提供一个序列化以及反序列化的方法，用来把snippet对象转化成json数据格式，创建serializers和创建Django表单类似。我们在 `snippets` 目录创建一个 `serializers.py` ：

```
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHO
ICES


class SnippetSerializer(serializers.Serializer):
    pk = serializers.IntegerField(read_only=True)
    title = serializers.CharField(required=False, allow_blank=Tr
ue, max_length=100)
    code = serializers.CharField(style={'base_template': 'textar
ea.html'})
    linenos = serializers.BooleanField(required=False)
    language = serializers.ChoiceField(choices=LANGUAGE_CHOICES,
 default='python')
    style = serializers.ChoiceField(choices=STYLE_CHOICES, defau
lt='friendly')

    def create(self, validated_data):
        """
        如果数据合法就创建并返回一个snippet实例
        """
        return Snippet.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """
        如果数据合法就更新并返回一个存在的snippet实例
        """
        instance.title = validated_data.get('title', instance.ti
tle)
        instance.code = validated_data.get('code', instance.code
)
        instance.linenos = validated_data.get('linenos', instanc
e.linenos)
        instance.language = validated_data.get('language', insta
nce.language)
        instance.style = validated_data.get('style', instance.st
yle)
        instance.save()
        return instance
```

在第一部分我们定义了序列化/反序列化的字段，`creat()` 和 `update()` 方法则定义了当我们调用 `serializer.save()` 时如何来创建或修改一个实例。

serializer类和Django的Form类很像，并且包含了相似的属性标识，比如 `required` 、 `max_length` 和 `default` 。

这些标识也能控制序列化后的字段如何展示，比如渲染成HTML。上面的 `{'base_template': 'textarea.html'}` 和在Django的Form中设定 `widget=widgets.Textarea` 是等效的。 这一点在控制API在浏览器中如何展示是非常有用的，我们后面的教程中会看到。

我们也可以使用 `ModelSerializer` 类来节省时间，后续也会介绍，这里我们先显式的定义serializer。

使用serializer

继续之前我们先来熟悉一下我们的serializer类，打开Django shell:

```
python manage.py shell
```

引入相关的代码后创建2个snippet实例：

```
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()
```

现在我们有了可以操作的实例了，让我们来序列化一个实例：

```
serializer = SnippetSerializer(snippet)
serializer.data
# {'pk': 2, 'title': u'', 'code': u'print "hello, world"\n', 'li
nenos': False, 'language': u'python', 'style': u'friendly'}
```

这里我们把snippet转换成了Python基本数据类型，接下来我们把其转换成json数据：

```
content = JSONRenderer().render(serializer.data)
content
# '{"pk": 2, "title": "", "code": "print \\"hello, world\\"\\n",
 "linenos": false, "language": "python", "style": "friendly"}'
```

反序列化也类似，我们先把一个stream转换成python基本数据类型：

```
from django.utils.six import BytesIO

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

然后将其转换为实例：

```
serializer = SnippetSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# OrderedDict([('title', ''), ('code', 'print "hello, world"\n')
, ('linenos', False), ('language', 'python'), ('style', 'friendl
y')])
serializer.save()
# <Snippet: Snippet object>
```

可以看出这和Django的Form多么相似，当我们使用serializer编写view时这一特效会更明显。

我们也可以序列化实例的集合，仅需要设置serializer的参数 `many=True` 即可：

```
serializer = SnippetSerializer(Snippet.objects.all(), many=True)
serializer.data
# [OrderedDict([('pk', 1), ('title', u''), ('code', u'foo = "bar
"\n'), ('linenos', False), ('language', 'python'), ('style', 'fr
iendly')]), OrderedDict([('pk', 2), ('title', u''), ('code', u'p
rint "hello, world"\n'), ('linenos', False), ('language', 'pytho
n'), ('style', 'friendly')]), OrderedDict([('pk', 3), ('title',
u''), ('code', u'print "hello, world"'), ('linenos', False), ('l
anguage', 'python'), ('style', 'friendly')])]
```

使用ModelSerializers

我们的 `SnippetSerializer` 类和 `Snippet` 模型有太多重复的代码，如果让代码更简洁就更好了。

Django提供了 `Form` 类和 `ModelForm` 类，同样的，REST framework提供了 `Serializer` 类和 `ModelSerializer` 。

让我们使用 `ModelSerializer` 来重构代码，编辑 `snippets/serializers.py` ：

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

我们可以通过打印输出来检查serializer包含哪些字段，打开Django shell并输入一下代码：

```
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, require
d=False)
#    code = CharField(style={'base_template': 'textarea.html'})
#    linenos = BooleanField(required=False)
#    language = ChoiceField(choices=[('Clipper', 'FoxPro'), ('Cu
cumber', 'Gherkin'), ('RobotFramework', 'RobotFramework'), ('aba
p', 'ABAP'), ('ada', 'Ada')...
#    style = ChoiceField(choices=[('autumn', 'autumn'), ('borlan
d', 'borland'), ('bw', 'bw'), ('colorful', 'colorful')...
```

请记住 `ModelSerializer` 并没有使用任何黑科技，它仅仅是一个创建serializer类的简单方法：

- 自动检测字段
- 简单的定义了 `create()` 和 `update()` 方法。

使用Serializer编写常规的Django视图

现在我们来使用Serializer类来编写API视图，这里我们不使用任何REST framewrok的其他特性，仅使用Django的常规方法编写视图。

首先我们创建一个 `HttpResponse` 的子类来返回json类型数据。

编辑 `snippets/views.py` ：

```
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):
    """
    用于返回JSON数据.
    """
    def __init__(self, data, **kwargs):
        content = JSONRenderer().render(data)
        kwargs['content_type'] = 'application/json'
        super(JSONResponse, self).__init__(content, **kwargs)
```

我们API的根目录是一个list view, 用于展示所有存在的snippet, 或建立新的snippet:

```
@csrf_exempt
def snippet_list(request):
    """
    展示所以snippets,或创建新的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，这里我们为了能简单的在客户端进行POST操作而使用了 `csrf_exempt` ，正常情况下你不应该这么做，REST framework提供了更安全的做法。

我们也需要一个页面用来展示、修改或删除某个具体的snippet：

```
@csrf_exempt
def snippet_detail(request, pk):
    """
    修改或删除一个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)
```

接下来修改 `snippets/urls.py` ：

```
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),
]
```

这里我们有许多细节没有处理，比如畸形的JSON数据、不支持的HTTP请求方法，这里我们都暂时返回500错误。

测试API

现在我们可以启动我们的服务器了。

首先使用 `quit()` 退出shell,然后启动服务：

```
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.
```

打开另一个终端，我们可以使用curl或httpie来进行测试。httpie是一个用python编写的易用的http客户端，首先来安装httpie:

```
pip install httpie
```

最终，我们得到了一个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"
  }
]
```

或者通过ID来获取某个特定的snippets：

```
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"
}
```

同样的，你可以使用浏览器访问那些URL。

教程2：Requests和Responses

从本章开始，我们将开始探索框架的核心，这里先介绍一些重要的概念。

Request对象

REST framework使用一个叫 `Requests` 的对象扩展了原生的 `HttpRequest` ，并提供了更灵活的请求处理。 `Requests` 对象的核心属性就是 `request.data` ，和 `requests.POST` 类似，但更强大：

```
request.POST  # 只处理form数据.只接受'POST'方法.
request.data  # 处理任意数据.接受'POST','PUT'和'PATCH'方法.
```

Response对象

REST framework也提供了一个类型为 `TemplateResponse` 的 `Response` 对象，它返回类型由数据决定。(原文如下，这句话我怎么翻译都觉得别扭，意思就是客户端要神码类型它返回神码类型)

> REST framework also introduces a Response object, which is a type of TemplateResponse that takes unrendered content and uses content negotiation to determine the correct content type to return to the client.

```
return Response(data)  # 返回类型由发出请求的客户端决定
```

状态码

使用数字HTTP状态码并不总是易于阅读的，并且当你得到一个错误的状态码时不容易引起注意。REST framework为每个状态码都提供了更明显的标志，比如 `status` 模块中的 `HTTP_400_BAD_REQUEST` ，使用它们替代数字是一个好注意。

装饰API视图

REST framework提供了2种装饰器来编写视图：

1. 基于函数视图的 `@api_view`
2. 基于类视图的 `APIView`

这些装饰器提供了少许功能，比如确保在视图中接收 `Request` 实例，添加context到 `Resonse` 对象来决定返回类型。

此外还提供了适当的错误处理，比如 `405 Method Not Allowed` 、有畸形数据传入时引发的 `ParseError` 异常等。

协同工作

现在我们使用这些组件来改写views.

我们不再需要 `JSONResponse` 类了，删除它后修改代码如下：

```
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):
    """
    展示或创建snippets.
    """
    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_40
0_BAD_REQUEST)
```

改造后的代码更简洁了，并且更像Forms。也使用了命名后的状态码让Response含义更加明显。

接下来修改单一的snippets视图：

```
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
    """
    修改或删除一个snippet.
    """
    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.dat
a)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_40
0_BAD_REQUEST)

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

一切都是如此熟悉，这和原生的Django Form很像。

注意我们并没有显示的声明request和response中的内容类型，`request.data`可以处理请求传入的json类型数据，也可以处理其他类型数据。同理，Response对象也可以为我们返回正确的数据类型。

为URL添加可选的数据格式后缀

我们的responses支持多种返回格式，利用这点我们可以通过在URL中添加格式后缀的方法来获取单一数据类型，这意味着我们的URL可以处理类似`http://example.com/api/items/4/.json`这样的格式。

首先在views添加 `format` 参数：

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

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

然后修改 `urls.py` ，添加 `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)
```

我们不需要添加任何额外信息，它给我们一个简单清晰的方法去获取指定格式的数据。

测试

和上一章一样，我们使用命令行来进行测试。尽管添加了相应的错误处理，但一切还是那么简单：

```
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"
  }
]
```

我们可以通过控制 `Accept` 来控制返回的数据类型：

```
http http://127.0.0.1:8000/snippets/ Accept:application/json  # Request JSON
http http://127.0.0.1:8000/snippets/ Accept:text/html         # Request HTML
```

或者通过添加格式后缀：

```
http http://127.0.0.1:8000/snippets.json  # JSON suffix
http http://127.0.0.1:8000/snippets.api   # Browsable API suffix
```

同样的，我们可以通过 `Content-Type` 控制发送请求的数据类型：

```
# 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"
}
```

或打开浏览器访问 `http://127.0.0.1:8000/snippets/`

易读性

由于API根据客户端发出的请求来决定返回的数据类型，当我们使用浏览器访问时默认要求返回HTML数据格式，这使我们的API在浏览器访问时结果格式十分易读。

使用浏览器访问时返回易于阅读结果是一个巨大的进步，这让我们更容易的开发、使用API。这使其他人员更加方便的使用、检查API。

教程3：类视图

我们也可以使用类视图来编写API的view,而不是函数视图。正如我们了解的，类视图是一种非常给力的模式让我们重用代码，保持DRY原则。

使用类视图重写API

我们来使用类视图来重写 `views.py`：

```
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_40
0_BAD_REQUEST)
```

目前看起来还不错，和之前的写法一样，但更清晰的分离了HTTP的请求方法。我们也需要修改单一的实例视图：

```
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.dat
a)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_40
0_BAD_REQUEST)

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

好极了，这也和函数视图非常相似。

因为使用了类视图，我们也需要改写 `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_vi
ew()),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

使用Mixins

使用类视图的一大优点就是可以方便的重用代码，目前我们使用了很多相似的代码来进行增删改查操作，这些常见的操作已经被封装在了REST framework的mixins类中。让我们使用mixins再次修改 `views.py` ：

```
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)
```

我们花点时间来看看这里发生了什么：我们使用 `GenericAPIView` 、 `ListModelMixin` 、 `CreateModelMixin` 改写了代码，基类提供了核心功能，而mixin类则提供了 `.list()` 和 `.ctreae()` 的行为，这里我们显式绑定GET方法和POST方法对应的功能，一切都很简单：

```
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)
```

类似的，我们使用 `GenericAPIView` 提供核心功能而mixin提供 `.retrieve()` 、 `.update()` 和 `.destroy()` 行为。

使用通用类视图

使用mixin类重写views让我们的代码比之前简洁了很多，但我们可以更进一步。REST framework提供了通用类视图来让代码更加精简：

```
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
```

WOW,真的酷毙了！我们节省的大量的时间，代码也更加强壮、简洁、符合Django风格。

第四章-认证和权限

现在我们的API没有任何限制谁可以编辑删除代码片段，我们需要一些更好的功能来满足下列要求：

- 代码片段需要和创建者进行关联
- 只有认证用户可以创建新的代码片段
- 只有创建者才能修改或删除代码片段
- 未认证用户只有只读权限

向模型中新增字段

`Snippet` 模型需要作出一些改变。首先添加2个字段，一个字段用于表明谁创建了这个代码片段，另一个用来存储高亮的HTML代码。

编辑 `models.py` 中关于 `Snippet` 的模型，新增下列代码：

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

也要确保当实例被保存时，使用 `pygments` 库来正确处理highlighted字段。

所以需要引入一些额外的包：

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

接下来，添加 `.save()` 到模型中：

```
def save(self, *args, **kwargs):
    """
    使用pygments来创建高亮的HTML代码。
    """
    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)
```

完成上面的步骤后，我们需要更新数据库表。正常情况下我们需要创建一个数据库迁移任务来实现这个目标，但在教程中，我们删除整个数据库重新建表：

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

你也许还需要创建几个用户来测试API，最方便的办法还是使用 `python manage.py createsuperuser` 命令。

为用户模型添加接口

现在我们有一些用户了，我们最好在API中添加些用户相关的接口。修改 `serializers.py` 并添加下列代码：

```
from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):
    snippets = serializers.PrimaryKeyRelatedField(many=True, que
ryset=Snippet.objects.all())

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

因为snippets和用户是一种反向关联，默认情况下不会包含在 `ModelSerializer` 类中，所以我们需要手动添加。

我们也需要对 `views.py` 进行修改，另外用户页面是只读的，所以使用 `ListAPIView` 以及 `RetrieveAPIView` 这2个通用类视图：

```
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
`
```

确保引入了 `UserSerializer` 类：

```
from snippets.serializers import UserSerializer
```

最后还需要对其进行URL的配置，修改 `urls.py` ，添加下列代码：

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

关联user和snippet

现在如果我们创建一个代码片段，是没法和用户进行关联的。因为用户信息是通过request获取而不是以serialized数据传递的。

为了解决这个问题。我们需要重写snippet视图中 `.perform_create()` 方法，这个方法准许我们修改实例如何被保存、处理任何由request或requested URL传递进来的隐含数据。

修改 `SnippetList` ，新增下列代码：

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

现在 `create()` 方法将获得一个新的字段'owner',值就是request中的用户信息。

更新serializer

现在代码片段和创建者之间建立了联系，现在需要修改 `SnippetSerializer` 来反映这一变化。修改 `serializers.py` 添加下列代码：

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

注意：确保你也添加了'owner'到内部类 `Meta` 中的fields字段里。

这个字段做了些很有趣的事情。 `source` 决定了显示user的哪个参数值，并且可以使用user的任何属性。

这里我们还使用了 `ReadOnlyField` 类型，不同于其他字段类型，比如 `CharField` ， `BooleanField` 等。这种类型是只读的，用于进行序列化时候的展示，并且反序列化时不会被修改。这里我们也可以使用 `CharField(read_only=True)` 来替代它。

添加权限认证

现在我们将代码片段和用户进行了关联，但我们需要确保只有认证用户才能创建、修改、删除代码片段。

REST framework包含了多种认证类，这里我们使用 `IsAuthenticatedOrReadOnly` ，这个类确保了只有认证用户才有读写权限，未认证用户则只有只读权限。

修改 `views.py` ,添加：

```
from rest_framework import permissions
```

然后修改 `SnippetList` 和 `SnippetDetail` 类，添加：

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

添加登录功能

如果你现在打开浏览器，你会发现你无法创建新的代码片段了，所以我们需要启用用户登录功能。编辑 `tutorial/urls.py` 新增代码:

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

在最后，添加：

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

你可以将'api-auth'替换成任何你喜欢的字符串，唯一的限制就是namespace必须为'rest_framework'。在Django1.9+的版本中，REST framework将自动设置，所以无需理会。

现在再次刷新页面将在右上角看到'Login'按钮，登录后就可以创建新代码片段了。

当你创建了几个代码片段后，访问 `/users/` 页面，就会看到每个用户对应的代码片段的主键列表了。

对象级别的权限

现在所有人都可以浏览代码片段了，接下来实现只有创建者才能删除或修改自己的代码片段功能。想实现这点，我们需要自定义权限认证方法。

新建文件 `permissions.py` ：

```
from rest_framework import permissions


class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    自定义权限，只有创建者才能编辑
    """

    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
```

接下来修改 `SnippetDetail` 视图：

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

并确保引入了所需的类：

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

再次打开浏览器，如果你是这个代码片段的创建者的话，你会看到'DELETE'和'PUT'操作按钮。

为API调用添加认证信息

因为我们添加了权限认证，所以我们想编辑任何代码片段时都需要提供认证信息。我们并没有设置任何认证相关的类，目前默认的认证方法是 `SessionAuthentication` 和 `BasicAuthentication` 。

当我们使用浏览器访问时，我们可以登录，浏览器会自动处理session信息用于认证。当我们使用编程方式调用API时，我们需要显式的为每个请求提供认证信息。

如果我们尝试创建一个代码片段而未提供认证消息的话，我们会得到一个错误返回：

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

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

而提供用户名和密码就可以创建了：

```
http -a tom:password123 POST http://127.0.0.1:8000/snippets/ cod
e="print 789"

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

第五章-Relationships和Hyperlinked

目前我们使用主键来表示模型之间的关系。在本章，我们将使用超链接关系来提高API的内聚性以及可读性。

为API创建一个根URL

现在我们'snippets'和'users'创建了相应的URL，但我们的API没有一个统一的入口。我们使用早些介绍的普通函数视图和 `@api_view` 装饰器来创建一个，编辑 `snippets/views.py` 添加下面代码：

```
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=fo
rmat),
        'snippets': reverse('snippet-list', request=request, for
mat=format)
    })
```

这里有两件事需要注意：首先我们使用的是REST框架提供的 `reverse` 函数来返回完全限定的URL;第二，URL通过稍后定义在 `snippets/urls.py` 中的名字来被识别。

为语法高亮功能创建URL

很明显的一件事就是我们还没为语法高亮功能创建URL。

与其它的API不同，这个接口我们想使用HTML来表示而不是JSON。REST框架提供了2种呈现HTML的方法，一种是使用模板渲染，另一种则是使用已经构建好的HTML代码。这里我们使用第二种方式。

另一个需要考虑的就是不存在通用类视图来供我们创建语法高亮视图使用，所以这里不返回一个对象实例，而是返回对象实例的属性。

这里我们使用最基础的类视图的 `get()` 方法而非通用类视图，
在 `snippets/views.py` 中添加如下代码：

```
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)
```

就像往常一样，我们还需要为新的视图来创建URL配置。修
改 `snippets/urls.py` ，添加：

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

使用超链接

在Web API中处理实体之间的关系是一件非常头疼的事情。下面有几种不同的方法来表示关系：

- 使用主键
- 使用超链接
- 在相关实体间使用唯一的slug字段表示
- 在相关实体间使用默认的字符串表示

- 将相关的子实体嵌套到上级关系中
- 其他自定义方法

REST framework支持上述所有方法，并且可以应用于正向关系、反向关系或类似通用外键这类自定义管理项中。

在这里我们在实体间使用超链接来进行关联，为了达到这个目的我们需要修改serializers，使用 `HyperlinkedModelSerializer` 来替代原先的 `ModelSerializer` ：

- HyperlinkedModelSerializer默认不包含主键
- HyperlinkedModelSerializer自动包含URL字段HyperlinkedIdentityField
- 使用HyperlinkedRelatedField来替代PrimaryKeyRelatedField表示关系

我们可以很容易的改写代码，编辑 `snippets/serializers.py` ：

```
 class SnippetSerializer(serializers.HyperlinkedModelSerializer):

    owner = serializers.ReadOnlyField(source='owner.username')
    highlight = serializers.HyperlinkedIdentityField(view_name='
snippet-highlight', format='html')

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


class UserSerializer(serializers.HyperlinkedModelSerializer):
    snippets = serializers.HyperlinkedRelatedField(many=True, vi
ew_name='snippet-detail', read_only=True)

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

这里新增了一个 `highlight` 字段，这个字段和 `url` 字段类型相同，区别就是它指向 `snippet-highlight` 而非 `snippet-detail` 。

由于我们有 `.json` 格式的后缀，所以我们也要指明 `highlight` 字段使用 `.html` 来返回相应的格式。

为URL命名

如果我们创建了一个基于超链接的API，我们需要确保每个URL都被命名了。让我们看看那些需要被命名的URL：

- 根URL包含'user-list'和'snippet-list'
- snippet serializer包含指向'snippet-highlight'的字段
- user serializer包含指向'snippet-detail'的字段
- snippet serializers和user serializers 包含'url'字段，这个字段默认指向'{model_name}-detail',这里分别是'snippet-detail'和'user-detail'

最终，我们的 `snippets/urls.py` 如下：

```
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')
])
```

分页

列表视图可能为用户返回很多代码片段，所以我们需要对结果进行分页，并且可以遍历每个单独的页面。

我们可以使用分页来修改默认的列表样式，修改 `tutorial/settings.py` 添加：

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

注意，所有关于REST框架的设定都在一个叫做'REST_FRAMEWORK'的字典中，这帮助我们将设定信息和其他的库分离开来。

如果需要的话我们也可以自定义分页样式，但这里我们先使用默认选项。

测试

打开浏览器，就会发现你可以使用超链接来简单的浏览API了。你也会在snippet中看到'highlight'链接,这将返回高亮的HTML格式代码。

教程6：ViewSets和Routers

REST framework提供了一种叫做 `ViewSets` 的抽象行为，它可以使开发人员聚焦于API的状态和实现，基于常见的约定而自动进行URL配置。

`ViewSet` 和 `View` 很像，除了它提供的是 `read` 以及 `update` 操作而不是HTTP的 `get` 或者 `post` 。

`ViewSet` 仅在被调用的时候才会和对应的方法进行绑定，当它被实例化时——通常是在使用 `Route` 类管理URL配置的时候。

使用ViewSet重构代码

首先我们使用但一个 `UserViewSet` 来取代 `UserList` 和 `UserDetail` ，我们先移除那两个类，并添加：

```
from rest_framework import viewsets

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

现在我们使用 `ReadOnlyModelViewSet` 自动提供了“只读”方法，并且依然想使用常规视图是那样设置了 `queryset` 和 `seriallizer_class` 属性，但我们不需要写2个类了。

下面我们修改 `SnippetList` 、 `SnippetDetail` 和 `SnippetHighlight` 类，同样删除它们并修改成一个类：

```
from rest_framework.decorators import detail_route

class SnippetViewSet(viewsets.ModelViewSet):
    """
    This viewset automatically provides `list`, `create`, `retri
eve`,
    `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` 来获取完整的读和写操作。注意我们使用 `@detail_route` 装饰器来创建自定义动作，这个装饰器可以用于任何不符合标准 `create/update/delete` 的动作。

使用 `@detail_route` 装饰的用户自定义动作默认相应GET请求，如果需要响应POST操作需要指定 `methods` 参数。

默认情况下，自定义动作对应的URL取决于它们的函数名，也可以通过给装饰器传递 `url_path` 参数来进行修改。

显式绑定URL和ViewSets

仅仅在我们定义URL配置时HTTP方法才会和我们定义的行为进行绑定。为了理解细节，我们先显式的在 `urls.py` 中进行操作：

```
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'
})
```

注意我们为每个 `ViewSet` 都创建了多个View，并且为每个View的行为和HTTP方法进行了绑定。

绑定后，我们可以像平常那样定义URL：

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

使用Routers

因为我们使用了 `ViewSet` 而不是 `View` ，实际上我们不需要自己定义URL。使用 `Router` 类可以自动的进行上述操作，我们需要做的仅仅是正确的注册View到Router中：

```
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'))
]
```

使用route生成了相同的URL路径，我们包含了2个参数——URL前缀以及Viewset本身。

`DefaultRouter` 类自动创建了根URL，所以也可以在 `views` 中移除 `api_root` 了。

权衡Views和Viewsets

viewsets是一种很用的抽象，它帮助我们确保URL符合惯例，减少代码编写量，使你专注于API交互和设计而不是URL配置上。

但这并不意味这总是一种好的选择，就好象函数视图和类视图之间的权衡一样，使用viewsets相比于显示构建vews，有些隐晦。

总结

通过少量的代码，我们构建出一个完备的Web API,完美支持浏览器访问、用户认证、权限管理，并且支持多种返回格式。

我们经历了每步设计的过程，并且知道了如果我们需要自定义功能，可以方便的使用Django原生的views.

你可以在github上找到最终的代码，以及模拟程序。

展望

整个教程到这就结束了，如果你想得到更多信息，这里有几点建议：

- 在Github提问并且提交pull requests。
- 加入REST framework discussion组并为社区做贡献。
- 在Twitter上跟随作者并say hi。