06-快速入门4--认证和权限

目前为止，我们的API对谁可以编辑或删除代码段没有任何限制。也就是说没有任何认证和权限相关的设置。通常我们都会做一些权限方面的设定，以确保：

- 每个**代码片段**都关联一个创建者
- 只有通过身份验证的用户可以**创建片段**
- 只有**代码片段**的创建者可以更新或删除它
- 未经身份验证的请求应具有全部的只读的访问权限

一、为模型添加用户字段

我们将对 `Snippet` 模型类进行一些更改。首先，添加几个字段。其中一个字段用于表示创建代码段的用户，另一个字段将用于存储代码的高亮显示的HTML内容。

将以下两个字段添加到 `models.py` 文件中的 `Snippet` 模型中。

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

上面的 `auth.User` 自动指向 `django.contrib.auth.models.User` 模型。

我们还需要确保在保存模型时，使用
pygaments填充要高亮显示的字段。

我们需要导入额外的模块：

```
1   from pygments.lexers import get_lexer_by_name
2   from pygments.formatters.html import HtmlFormatter
3   from pygments import highlight
```

现在我们可以在模型中添加一个 `.save()` 方法，它会覆盖父类的save方法，添加我们自己的一些逻辑：

```
def save(self, *args, **kwargs):
    """
    使用pygements库为代码片段创建高亮的HTML表示
    """
    lexer = get_lexer_by_name(self.language)
    linenos = 'table' if self.linenos else False
    options = {'title': self.title} if self.title else {}
    formatter = HtmlFormatter(style=self.style, linenos=linenos,
                              full=True, **options)
    self.highlighted = highlight(self.code, lexer, formatter)
    super(Snippet, self).save(*args, **kwargs)
```

> 这种做法是Django模型层提供的钩子，有兴趣可以看我的Django教程。save方法里，关于代码高亮的处理工作我们不用关心，在这里不是重点。

完成这些工作后，我们需要更新我们的数据库表。 通常这种情况我们会创建一个数据库迁移（migration）来实现这一点，但由于现在我们只是个教程示例，所以简单粗暴地选择直接删除数据库并重新开始。

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

注：在Pycharm里也是同样删除两个内容。

再使用 `createsuperuser` 命令，创建一个管理员账号，用于测试：

```
python manage.py  createsuperuser
```

到目前为止，我们的models.py文件是这样的：

```
from django.db import models
from pygments.lexers import get_all_lexers
from pygments.styles import get_all_styles
from pygments.lexers import  get_lexer_by_name
from pygments.formatters.html import  HtmlFormatter
from pygments import highlight

# 下面的几行代码是处理代码高亮的，不好理解，但没关系，它不重要。
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())
```

```
12
13
14  class Snippet(models.Model):
15      created = models.DateTimeField(auto_now_add=True)
16      title = models.CharField(max_length=100, blank=True, default='')
17      code = models.TextField()
18      linenos = models.BooleanField(default=False)
19      language = models.CharField(choices=LANGUAGE_CHOICES, default='python',
    max_length=100)
20      style = models.CharField(choices=STYLE_CHOICES, default='friendly',
    max_length=100)
21      owner = models.ForeignKey('auth.User', related_name='snippets',
    on_delete=models.CASCADE)
22      highlighted = models.TextField()
23
24      class Meta:
25          ordering = ('created',)
26
27      def save(self, *args, **kwargs):
28          """
29          Use the `pygments` library to create a highlighted HTML
30          representation of the code snippet.
31          """
32          lexer = get_lexer_by_name(self.language)
33          linenos = 'table' if self.linenos else False
34          options = {'title': self.title} if self.title else {}
35          formatter = HtmlFormatter(style=self.style, linenos=linenos,
36                                    full=True, **options)
37          self.highlighted = highlight(self.code, lexer, formatter)
38          super(Snippet, self).save(*args, **kwargs)
```

二、创建用户的序列化器

上面，我们为Snippet模型添加了两个字段，其中关于代码高亮的字段，我们在save方法里处理了。但是那个user字段呢？它关联到Django.contrib.auth.models.User模型了！

想一想，我们在序列化Snippet的时候，这个User字段怎么办？

当然也要序列化了！那谁来序列化它呢？没人帮你，得你自己写！

在 `serializers.py` 列化类：

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

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

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

依然是风骚的继承ModelSerializer类，然后在Meta中，指定model和fileds属性的值。

因为 `'snippets'` 字段在用户模型中是一个**反向**外键关系。在使用 `ModelSerializer` 类时它默认不会被包含，所以我们需要为它添加一个显式字段。

到目前为止，serializers.py文件中的全部内容如下：

```
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES,  STYLE_CHOICES
from django.contrib.auth.models import User


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


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

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

注意：此时你是无法添加新的代码片段的，会报错，因为它的外键字段owner没有定义序列化方法！

序列化器创建好了，那我们同时也为User模型创建两个API视图吧！当然也可以不创建！

为了将用户展示为只读视图，我们将使用 `ListAPIView` 和 `RetrieveAPIView` 这两个基于类的通用视图。在 `views.py` 中添加下面的代码：

```
from django.contrib.auth.models import User
from snippets.serializers import  UserSerializer

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


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

目前为止，views.py文件的内容如下：

```
from snippets.models import  Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics
from django.contrib.auth.models import User
from snippets.serializers import  UserSerializer


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


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


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


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

```

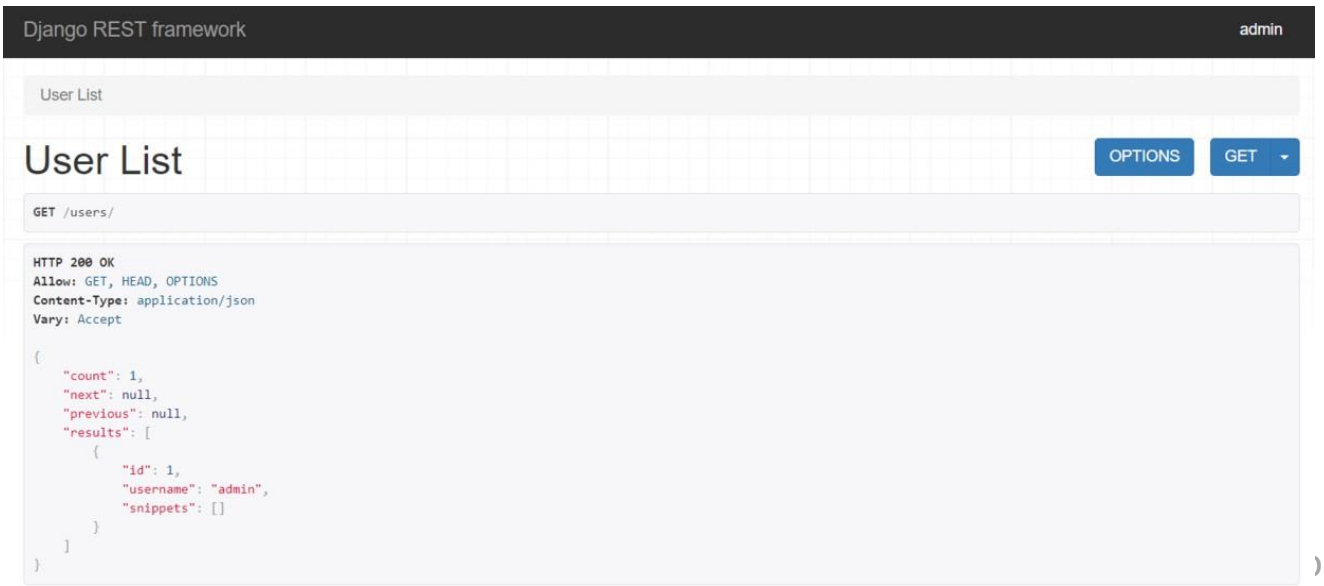
最后，我们还需要在URL conf中添加路由。将以下内容添加到 `snippets.urls.py` 文件的urlpatterns中。

```
    path('users/', views.UserList.as_view()),
    path('users/<int:pk>/', views.UserDetail.as_view()),
```

目前为止，`snippets.urls.py`文件的w内容如下：

```
# from django.urls import path
# from snippets import views
#
# urlpatterns = [
#     path('snippets/', views.snippet_list),
#     path('snippets/<int:pk>/',  views.snippet_detail),
# ]


from django.urls import path
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

urlpatterns = [
    path('snippets/', views.SnippetList.as_view()),
    path('snippets/<int:pk>/', views.SnippetDetail.as_view()),
    path('users/', views.UserList.as_view()),
    path('users/<int:pk>/', views.UserDetail.as_view()),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

重启服务器，访问`http://127.0.0.1:8000/users/`看看我们的用户API：



可以看到，User的页面里只有list和detail，无法post和put用户。也就是只读！

为什么snippet也不能新增？猜一下

三、关联Snippet和用户

当前，我们是不能创建Snippet对象的。

解决这个问题的办法是在我们的代码片段视图中重写父类 `.perform_create()` 方法，这个方法是DRF给我们提供的钩子，让我们可以修改实例创建的方法，添加我们需要的代码逻辑。

在 `SnippetList` 视图类中，添加以下方法：

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

后台的 `create()` 方法现在将具有 `'owner'` 参数，以及request.user这个参数值。这样，完整的Snippet实例就被创建了。

> 要注意，因为官方文档的逻辑混乱，对新手造成很大的困扰。从原则上说，User的序列化和Snippet的序列化完全是两码事，互不影响。但是Snippet的序列化，需要一个user关联的字段，也就是这个owner字段。所以，这个知识点，其实应该挪到上面去。

> 并且要注意，这个perform_create方法是放在视图中的，不是序列化器里的。而前面的highlighted字段呢？在模型的save方法里处理！WTF，混乱的设计逻辑！太不优雅了！DRF是我研究过的最糟糕的库！

四、更新我们的序列化器

现在，让我们更新 `SnippetSerializer` 类来体现这个关联。将以下字段添加到 `SnippetSerializer` 中：

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

注意：确保你还将 `'owner'` 添加到内部 `Meta` 类的字段列表中。

这个字段非常有趣。 `source` 参数控制哪个属性用于填充字段，并且可以指向序列化实例上的任何属性。它也可以采用如上所示点链接的方式，类似Django模板语言的方式遍历给定的属

性。

我们添加的字段是无类型的 `ReadOnlyField`类，区别于其他类型的字段（如`CharField`，`BooleanField`等）。无类型的`ReadOnlyField`始终是只读的，只能用于序列化表示，不能用于在反序列化时更新模型实例。我们也可以在这里使用`CharField(read_only=True)`，效果是一样的。

现在，你重启服务器，再看看界面：

下面还有个post表单：

但是，你以为这样就可以在表单中填写数据，然后创建一个Snippet对象了吗？你太幼稚了！我们还没有登录呢？当前是没有request.user的，会报下面的错误：

```
1   ValueError at /snippets/
2   Cannot assign "<django.contrib.auth.models.AnonymousUser object at
    0x000002CCCCB1FC88>": "Snippet.owner" must be a "User"  instance.
```

意思是不能给Snippet.owner字段赋值一个非User模型的匿名对象！

五、为视图设置权限

搞了半天，才真正到我们的权限部分。

现在，代码片段与用户是相关联的，我们希望只有经过身份验证的用户才能创建，更新和删除代码片段。

REST框架包括许多权限类，我们可以使用这些权限类来限制谁可以访问给定的视图。 本教程中，我们使用 `IsAuthenticatedOrReadOnly` 类，这将使得经过身份验证的请求获得读写权限，未经身份验证的请求只有只读权限。

首先要在视图模块中导入以下内容：

```
from rest_framework import  permissions
```

然后，将以下属性添加到 `SnippetList` 和 `SnippetDetail` 视图类中。

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

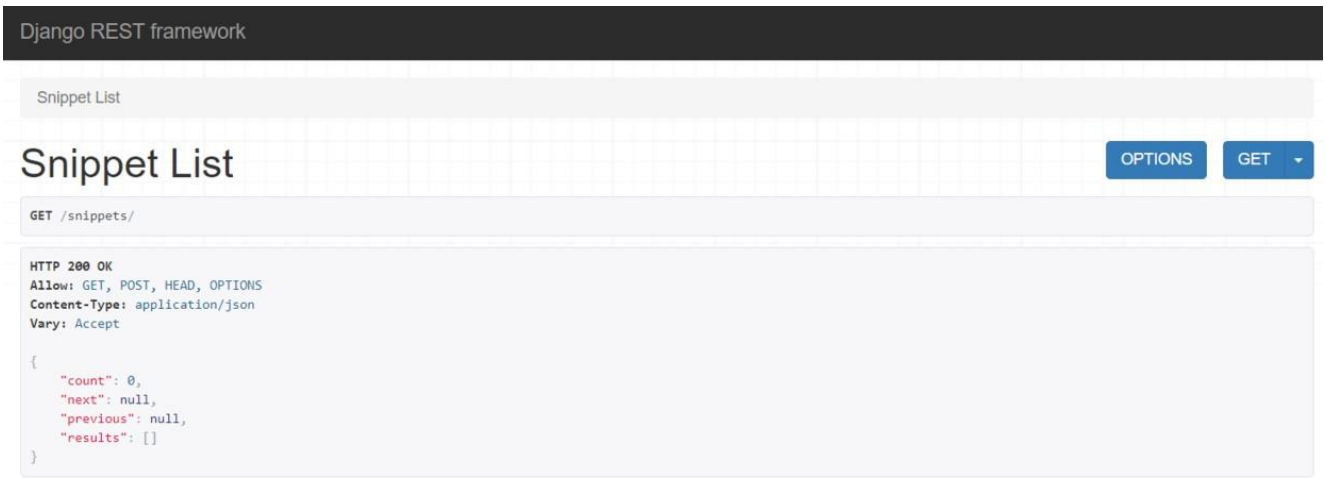
目前为止，views.py的完整内容如下：

```
from snippets.models import  Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics
from django.contrib.auth.models import User
from snippets.serializers import  UserSerializer
from rest_framework import permissions


class SnippetList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)

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


class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)


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

```

```
28
29  class UserDetail(generics.RetrieveAPIView):
30      queryset = User.objects.all()
31      serializer_class = UserSerializer
```

六、给浏览器上的可视化API添加登陆功能

如果此时，你打开浏览器并浏览API，那么你会发现不能创建新的代码片段。因为你还没有登录，只有登陆用户才能创建新的代码片段。



那么去哪里登录呢？没有输入框啊！

别急，只需要在项目根 `urls.py` 文件中的URLconf来添加可浏览的API使用的登录视图的路由即可。

在项目根 `urls.py` 顶部添加以下导入：

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

在文件末尾添加下面的代码：

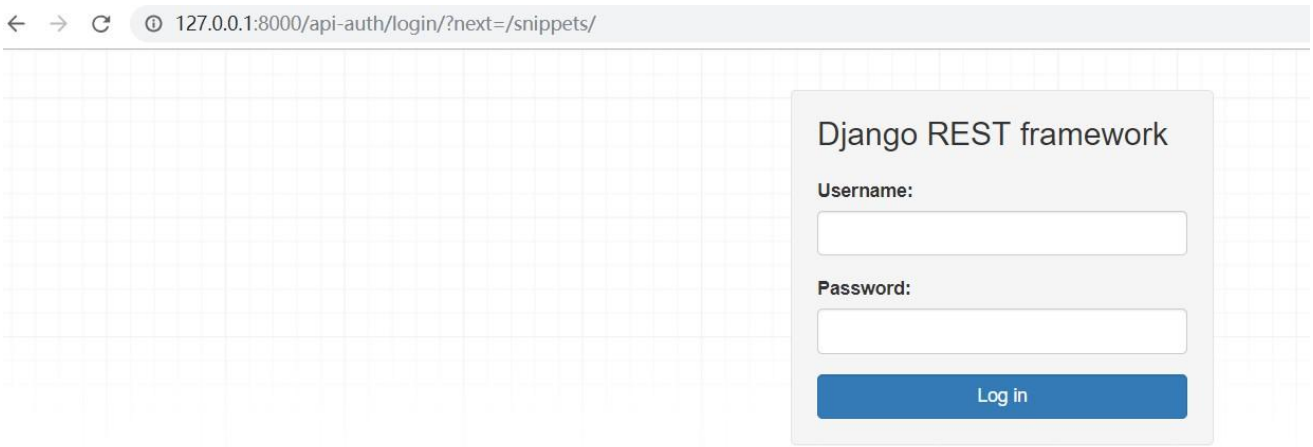
```
1   urlpatterns += [
2   path('api-auth/', include('rest_framework.urls')), 3
    ]
```

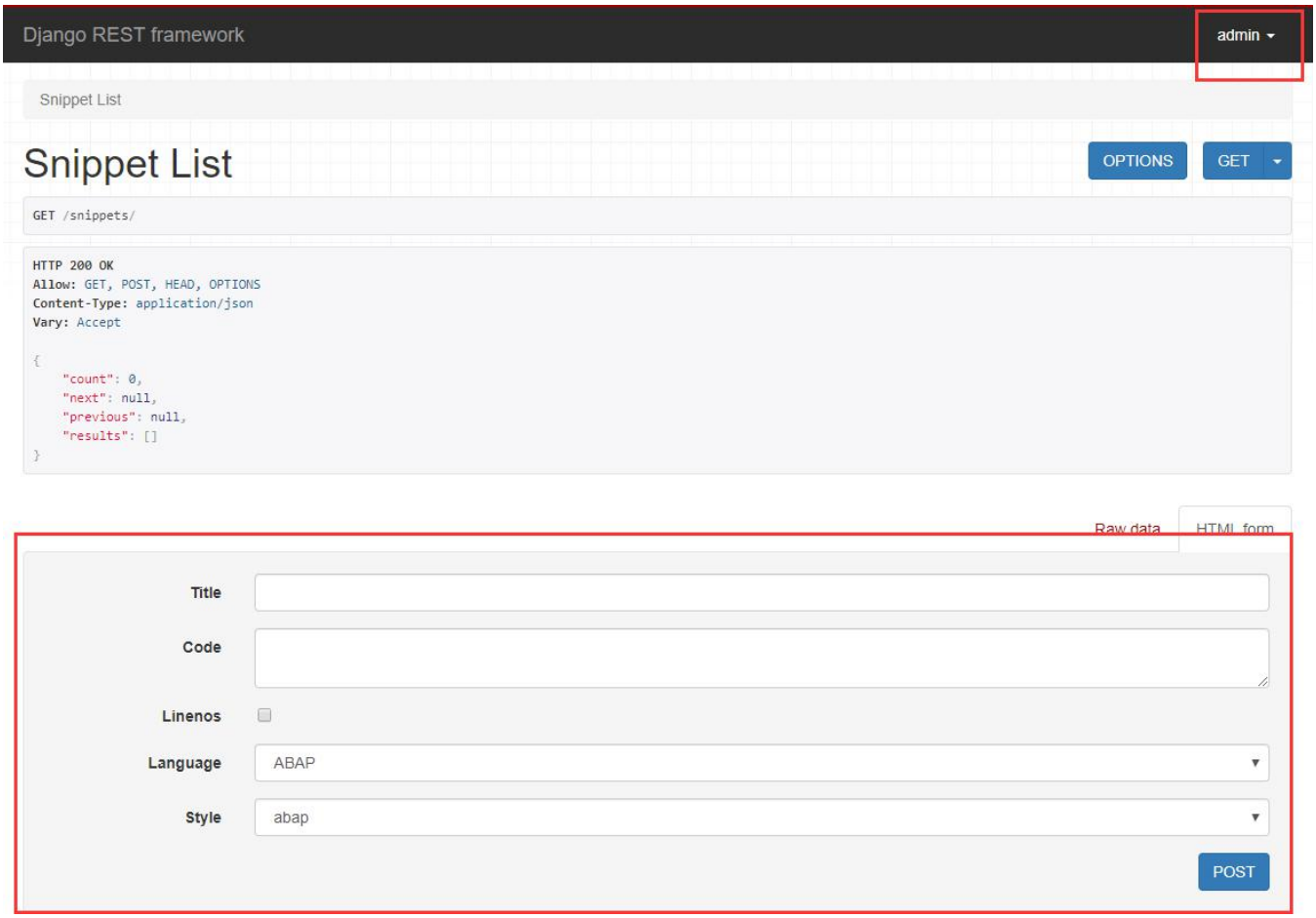
模式的 `api-auth/'` 部分实际上可以是任何你想使用的字符串。

现在，如果你再次重启服务器，打开浏览器并刷新页面，你将在页面右上角看到一个“login”链接。如果你用先前创建的超级用户登录，就可以再次创建代码片段。

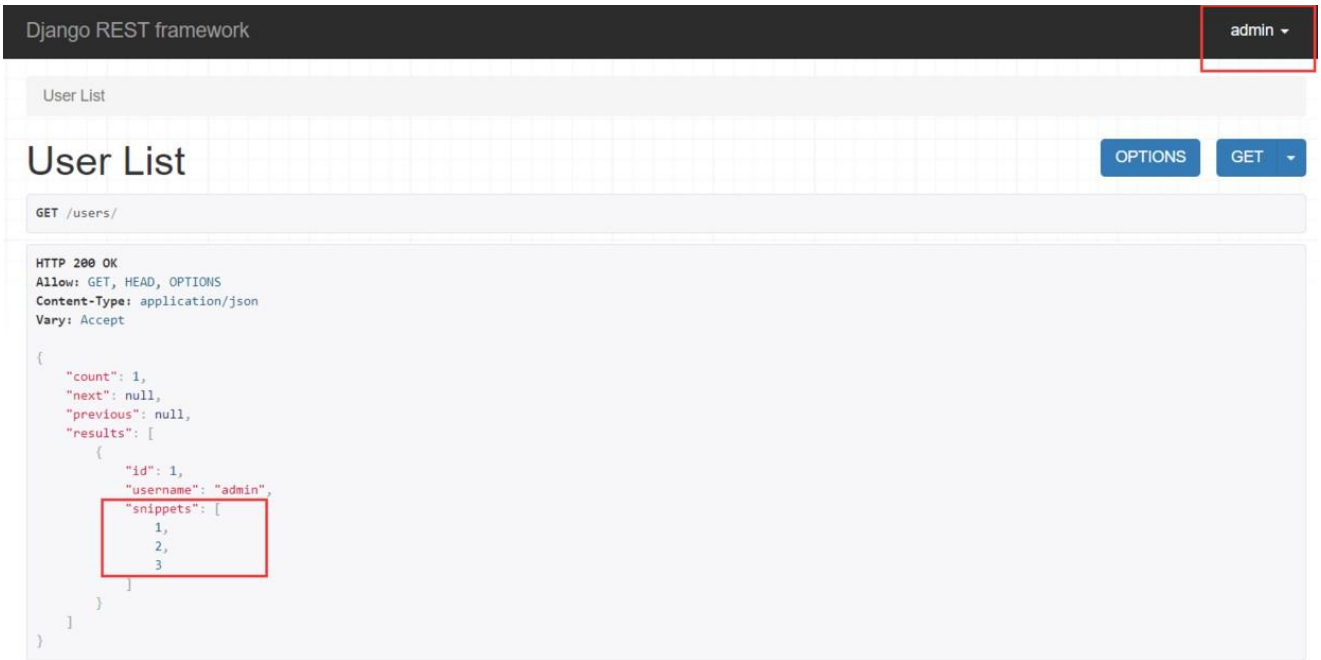

Django REST framework
Log in
Snippet List
Snippet List
OPTIONS
GET
GET /snippets/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
"count": 0,
"next": null,
"previous": null,
"results": []
}

登录一下试试：


127.0.0.1:8000/api-auth/login/?next=/snippets/
Django REST framework
Username:
Password:
Log in



创建一些代码片段后，访问'/users/'这个url路径，你会发现每个用户创建的'snippets'对象都包含在内。



七、设置对象级别的权限

我们希望所有的代码片段都可以被任何人看到，但也要确保只有创建代码片段的用户才能更新或删除它。

为此，我们将需要创建一个自定义权限。

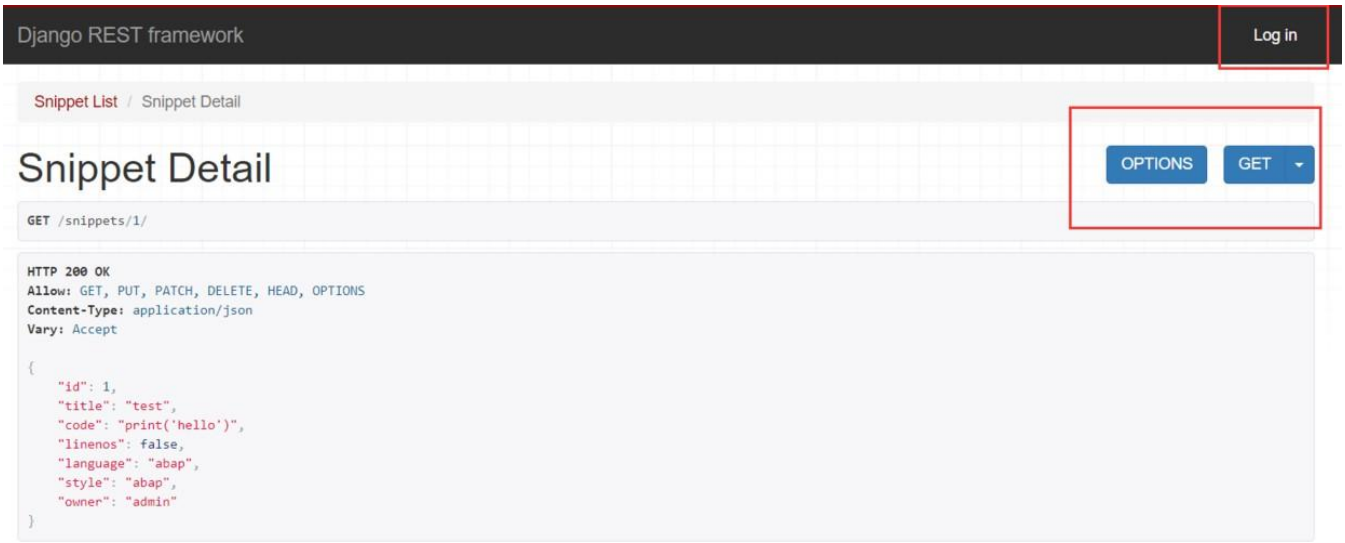
在snippets这个app中，创建一个新文件 `permissions.py` ，并写入下面的代码：

```
from rest_framework import  permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    """
    自定义权限只允许对象的所有者编辑它。
    """

    def has_object_permission(self, request, view, obj):
        # 允许任何请求进行读取
        # 所以我们总是允许GET, HEAD或OPTIONS请求。
        if request.method in permissions.SAFE_METHODS:
            return True

        # 只有该snippet的所有者才允许写权限。
        # 别告诉我你读不懂这句代码和这里的if/else逻辑
        return obj.owner == request.user
```

现在，我们可以在 `SnippetDetail` 视图类中编辑 `permission_classes` 属性将该自定义权限添加到我们的代码片段实例路径：

```
# 先导入我们自定的权限类
from snippets.permissions import  IsOwnerOrReadOnly

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

现在，重启服务器，再次打开浏览器，你会发现如果你以代码片段创建者的身份登录的话，“DELETE”和“PUT”操作才会显示在页面上。否则如下图所示：



到目前为止，veiws.py的全部代码如下：

```
from snippets.models import  Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics
from django.contrib.auth.models import User
from snippets.serializers import UserSerializer
from rest_framework import permissions
from snippets.permissions import  IsOwnerOrReadOnly


class SnippetList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)

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


class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,
                          IsOwnerOrReadOnly,)


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

```
29
30
31  class UserDetail(generics.RetrieveAPIView):
32      queryset = User.objects.all()
33      serializer_class = UserSerializer
```

八、使用API进行身份验证

现在因为我们在API上设置了权限，如果我们要编辑某个代码片段，我们都需要验证请求是否认证了。我们还没有设置任何身份验证类，所以应用的是默认的 `SessionAuthentication` 和 `BasicAuthentication` 这两个认证类。

当我们通过Web浏览器与API进行交互时，我们可以登录，然后浏览器会话将为请求提供所需的身份验证，也就是我们上面的操作过程。

如果我们在代码中与API交互，则需要在每次请求上显式地提供身份验证凭据。

如果我们没有经过身份验证就尝试创建一个代码片段，就会像下面展示的那样收到错误提示：

```
1   命令行中执行: http POST http://127.0.0.1:8000/snippets/ code="print 123"
2
3   HTTP/1.1 403 Forbidden
4   Allow: GET, POST, HEAD, OPTIONS
5   Content-Length: 58
6   Content-Type: application/json
7   Date: Sun, 28 Apr 2019 09:17:51  GMT
8   Server: WSGIServer/0.2 CPython/3.7.3
9   Vary: Accept, Cookie
10  X-Frame-Options: SAMEORIGIN
11
12  {
13      "detail": "Authentication credentials were not provided."
14  }
```

这个时候可以通过加上用户名和密码来提供身份认证：

07-快速入门5--关系和超链接API

目前我们的API中的关系是用主键表示的。下面我们将通过使用超链接来提高我们API的内聚力和可发现性。

一、为我们的API创建一个根路径

现在我们有'snippets'和'users'的这两个url，但是我们的API却没有一个入口点。我们将使用一个常规的基于函数的视图和我们前面介绍的 `@api_view` 装饰器创建一个。在你的 `snippets/views.py` 中添加下面的代码：

```
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=format),
        'snippets': reverse('snippet-list', request=request, format=format)
    })
```

这里应该注意两件事。首先，我们使用REST框架的 `reverse`功能来返回完整的URL；第二，URL模式是通过简单方便易懂的名称来标识的，我们稍后将在 `snippets/urls.py` 中声明。

（反向解析必须修改url中，给path('snippets/',
views.SnippetList.as_view(),name='snippet-list'),
）

二、为高亮显示的代码片段创建路径

我们的API中另一个明显缺少的是高亮代码的路径。

与所有其他API路径不同，我们不想使用JSON，而只是需要HTML表示。REST框架提供了两种HTML渲染器，一种用于处理使用模板渲染的HTML，另一种用于处理预渲染的HTML。教程里，我们选择第二个渲染器。

创建代码高亮视图时需要考虑的另一件事是，我们没有可用的具体通用视图。我们不是返回对象实例，而是返回对象实例的某个属性。

不是使用某个DRF提供的具体通用视图，下面我们将使用基类来表示实例，并创建我们自己的 `.get()` 方法。在你的 `snippets/views.py` 中添加：

```
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)
```

像往常一样，我们需要在URLconf中添加新视图的路由。在 `snippets/urls.py` 中为 `api_root` 视图添加下面的url路由：

```
path('',  views.api_root),
```

以及为高亮代码片段添加一个url模式：

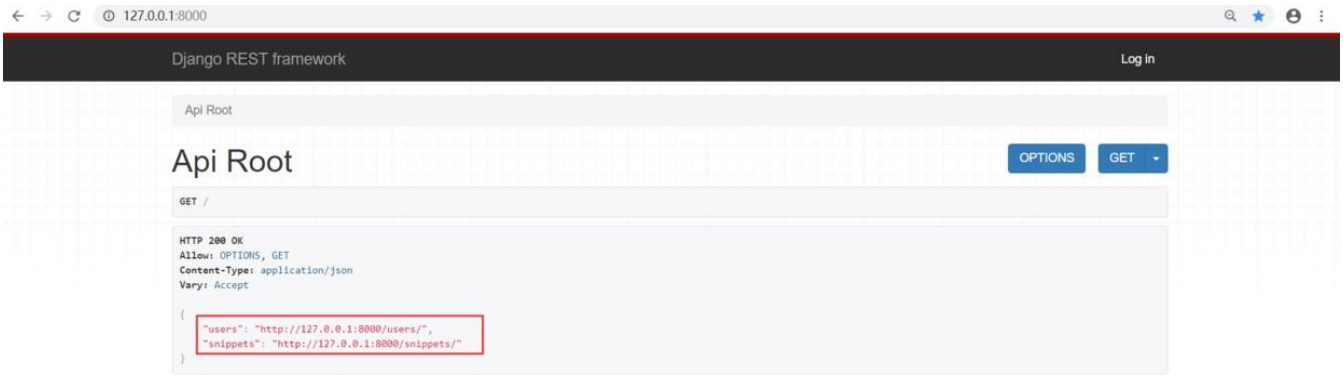
```
path('snippets/<int:pk>/highlight/',   views.SnippetHighlight.as_view()),
```

到目前，我们的views.py内容如下：

```
from snippets.models import  Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics
from django.contrib.auth.models import User
from snippets.serializers import UserSerializer
from rest_framework import permissions
from snippets.permissions import  IsOwnerOrReadOnly


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


@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)
    })

```

```
23
24  class SnippetHighlight(generics.GenericAPIView):
25      queryset = Snippet.objects.all()
26      renderer_classes = (renderers.StaticHTMLRenderer,)
27
28      def get(self, request, *args, **kwargs):
29          snippet = self.get_object()
30          return Response(snippet.highlighted)
31
32
33  class SnippetList(generics.ListCreateAPIView):
34      queryset = Snippet.objects.all()
35      serializer_class = SnippetSerializer
36      permission_classes = (permissions.IsAuthenticatedOrReadOnly,)
37
38      def perform_create(self, serializer):
39          serializer.save(owner=self.request.user)
40
41
42  class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
43      queryset = Snippet.objects.all()
44      serializer_class = SnippetSerializer
45      permission_classes = (permissions.IsAuthenticatedOrReadOnly,
46                            IsOwnerOrReadOnly,)
47
48
49  class  UserList(generics.ListAPIView):
50      queryset =  User.objects.all()
51      serializer_class = UserSerializer
52
53
54  class UserDetail(generics.RetrieveAPIView):
55      queryset = User.objects.all()
56      serializer_class = UserSerializer
```

重启服务器，访问 `127.0.0.1:8000` ，会看到如下的页面，注意其中的红框：



点击链接会跳转到对应的页面。

#反向解析一定要在url中配置name

三、使用链接形式的API

处理好实体之间的关系是Web API设计中比较有挑战性的工作。我们可以选择几种不同的方式来代表一种关联关系，比如：

- 使用主键。
- 在实体之间使用超级链接。
- 在相关实体上使用唯一的标识字段。
- 使用相关实体的默认字符串表示形式。
- 将相关实体嵌套在父表示中。
- 一些其他自定义表示。

REST框架支持所有这些方式，并且可以将它们应用于正向或反向关联，也可以在诸如通用外键之类的自定义管理器上应用。

本教程中，我们采用超链接的方式。这样的话，我们需要修改我们的序列化类来，改为继承 `HyperlinkedModelSerializer` 类而不是现有的 `ModelSerializer` 类。

`HyperlinkedModelSerializer` 类是DRF为我们提供的用于实现超级链接模型序列化器的父类。也是常用选择之一。

`HyperlinkedModelSerializer` 与 `ModelSerializer` 有以下区别：

- 默认情况下不包括 `id` 字段。
- 自带一个 `url` 字段，使用 `HyperlinkedIdentityField` 。
- 关联关系使用 `HyperlinkedRelatedField` 字段类型，而不是 `PrimaryKeyRelatedField` 字段类型。

修改你的 `snippets/serializers.py`，如下所示：

```
1   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', 'id', 'highlight', 'owner',
                  'title', 'code', 'linenos', 'language',  'style')


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

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

下面给出完整的serializers.py的代码：

```
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES
from django.contrib.auth.models import User


# class SnippetSerializer(serializers.ModelSerializer):
#     owner = serializers.ReadOnlyField(source='owner.username')
#
#     class Meta:
#         model = Snippet
#         fields = ('id', 'title', 'code', 'linenos', 'language', 'style', 'owner')
#
#
# class UserSerializer(serializers.ModelSerializer):
#     snippets = serializers.PrimaryKeyRelatedField(many=True, queryset=Snippet.objects.all())
#
#     class Meta:
#         model = User
#         fields = ('id', 'username',  'snippets')


class SnippetSerializer(serializers.HyperlinkedModelSerializer):
```

```
23      owner = serializers.ReadOnlyField(source='owner.username')
24      highlight = serializers.HyperlinkedIdentityField(view_name='snippet-
    highlight', format='html')
25
26      class Meta:
27          model = Snippet
28          fields = ('url', 'id', 'highlight', 'owner',
29                    'title', 'code', 'linenos', 'language',  'style')
30
31
32  class UserSerializer(serializers.HyperlinkedModelSerializer):
33      snippets = serializers.HyperlinkedRelatedField(many=True,
    view_name='snippet-detail', read_only=True)
34
35      class Meta:
36          model = User
37          fields = ('url', 'id', 'username',  'snippets')
```

官方教程的跳跃性这一节有些大

请注意，我们添加了一个新的 `'highlight'` 字段。该字段与 `url` 字段的类型相同，不同之处在于它指向 `'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，例如 `'.json'` ，我们还需要 `highlight` 字段上指出在任何格式后缀的超链接，它应该使用 `'.html'` 后缀。

四、确保我们的URL模式使用了name参数

如果我们要使用超链接的API，那么需要确保为我们的URL模式命名，也就是给路由添加name参数。我们来看看我们需要命名的URL模式。

- API的根路径指向了 `'user-list'` 和 `'snippet-list'` 。
- 代码片段序列化器包含一个指向 `'snippet-highlight'` 的字段。
- 我们的用户序列化器包含一个指向 `'snippet-detail'` 的字段。
- 我们的代码片段和用户序列化程序包括 `'url'` 字段，默认情况下将指向 `'{model_name}-detail'` ，在这个例子中就是 `'snippet-detail'` 和 `'user-detail'` 。

所以，我们要为上面几条，分别在对应的路由条目后面添加name参数，参数的值就是上面的那些对应的字符串。

将所有这些名称添加到我们的URLconf中后，最终我们的 `snippets/urls.py` 文件应该如下所示：

```
from django.urls import path
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

# API endpoints
urlpatterns =
    format_suffix_patterns([ path('',
    views.api_root), path('snippets/',
        views.SnippetList.as_view(),
        name='snippet-list'),
    path('snippets/<int:pk>/',
        views.SnippetDetail.as_view(),
        name='snippet-detail'),
    path('snippets/<int:pk>/highlight/',
        views.SnippetHighlight.as_view(),
        name='snippet-highlight'),
    path('users/',
        views.UserList.as_view(),
        name='user-list'),
    path('users/<int:pk>/',
        views.UserDetail.as_view(),
        name='user-detail')
])
```

最终snippet/urls.py的内容如下：

```
from django.urls import path
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

# urlpatterns = [
#     path('snippets/', views.SnippetList.as_view()),
#     path('snippets/<int:pk>/', views.SnippetDetail.as_view()),
#     path('users/', views.UserList.as_view()),
#     path('users/<int:pk>/', views.UserDetail.as_view()),
#     path('', views.api_root),
#     path('snippets/<int:pk>/highlight/',
views.SnippetHighlight.as_view()),
# ]


```

```
# 略微调整了一下顺序
urlpatterns = [
    path('', views.api_root),
    path('snippets/', views.SnippetList.as_view(), name='snippet-list'),
    path('snippets/<int:pk>/', views.SnippetDetail.as_view(),
name='snippet-detail'),
    path('snippets/<int:pk>/highlight/', views.SnippetHighlight.as_view(),
name='snippet-highlight'),
    path('users/', views.UserList.as_view(), name='user-list'),
    path('users/<int:pk>/', views.UserDetail.as_view(), name='user-detail')
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

五、添加分页功能

用户和代码片段的列表视图可能会返回相当多的实例，因此我们想要对结果进行分页，并允许API客户端依次获取每个单独的页面内容。

稍微修改下我们的 `tutorial/settings.py` 文件，添加以下设置：

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
'PAGE_SIZE': 10 4
}
```

请注意，REST框架中的所有设置都放在一个名为"REST_FRAMEWORK"的字典中，这有助于区分项目中的其他设置。

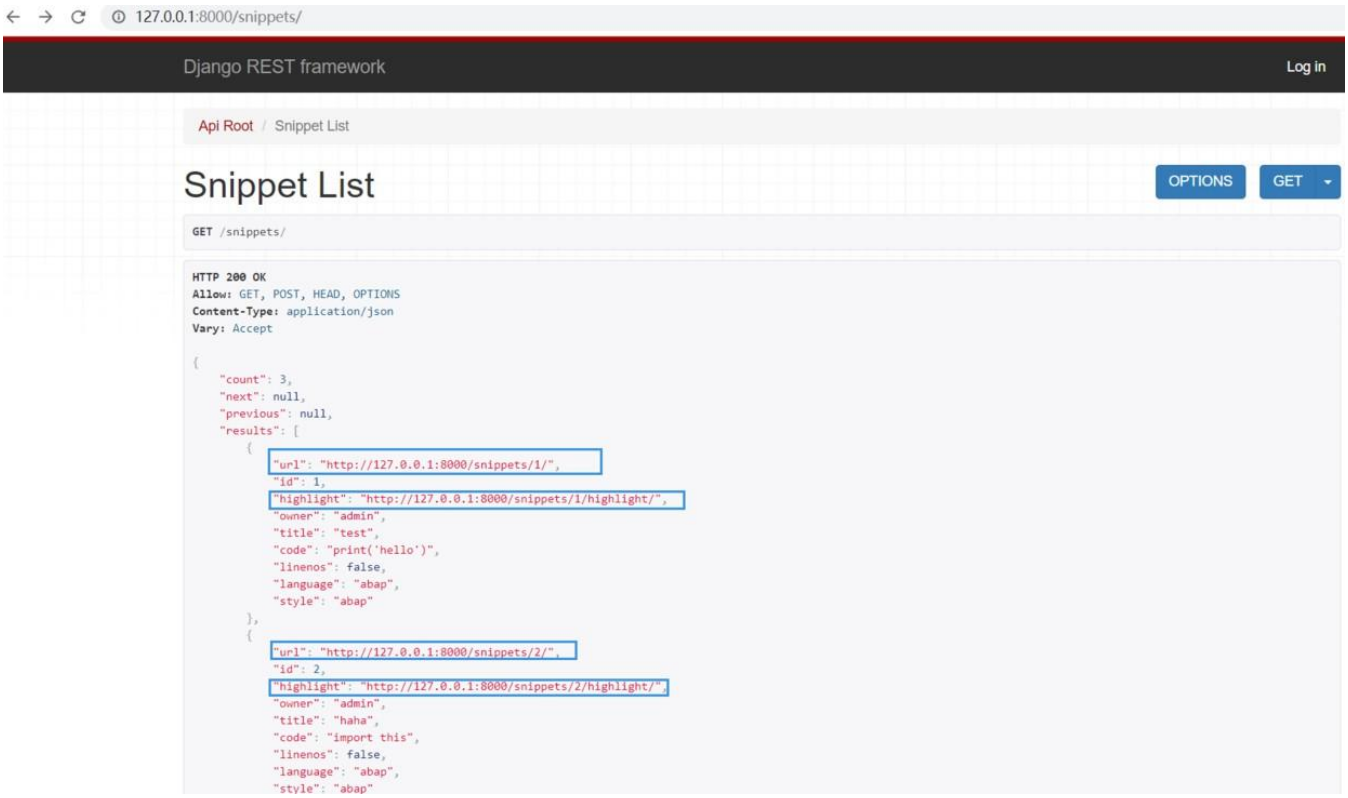
如果需要的话，我们也可以自定义分页风格，但在这个教程中，我们将一直使用默认设置。

六、浏览API

好了，可以打开浏览器并浏览我们的API了，你可以简单的通过页面上的超链接来了解API。

你还可以看到代码片段实例上的'highlight'链接，它能带你跳转到高亮显示的代码HTML表示。

先看snippets_list页面：



通过页面上的url链接可以跳转到对应的页面，比如高亮页面：

总结

关系带给我们的是可跳转；超链接序列化器带给我们的是可点击的对象链接，而不是冰冷的主键数字。也更利于前端发现后端的API。

08-快速入门6--视图集和路由器

除了前面已经让人头疼的视图体系外，REST框架还给我们提供了一个更加抽象的ViewSets视图集。ViewSets提供一套自动的urlconf路由，让开发人员可以将更多的精力用于对API的状态和交互，而不必操心路由路径的编写工作。

`ViewSet` 类与 `View`类几乎相同，不同之处在于它们提供诸如 `read` 或 `update` 之类的操作，而不是 `get` 或 `put` 等方法处理程序。这是DRF在设计的时候，为了防止和原生的Django语

法之间的冲突。

`ViewSet` 通常使用 `Router` 类来处理URL conf。

一、使用ViewSets重构视图

我们准备把目前的视图重构成视图集。

首先让我们将 `UserList` 和 `UserDetail` 视图重构为一个 `UserViewSet` 。我们可以删除这两个视图，并用一个类替换它们：

```
from rest framework import  viewsets

class UserViewSet(viewsets.ReadOnlyModelViewSet):
    """
    这个视图集会自动提供`list`和 `detail`操作
    """
    queryset = User.objects.all()
    serializer_class =  UserSerializer
```

可以看到，一个视图集同时提供了原来两个类视图的功能。

这里，我们使用 `ReadOnlyModelViewSet` 类来自动提供默认的“只读”操作。我们仍然像使用常规视图那样设置 `queryset` 和 `serializer_class` 属性，但我们不再需要向两个不同的类提供相同的信息。

接下来，我们将替换 ，`SnippetList` `SnippetDetail` `SnippetHighlight` 视图类。我们可以删除三个视图，并再次用一个类替换它们。

```
from rest_framework.decorators import  action
from rest_framework.response import  Response

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,)

    @action(detail=True, 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)
```

这次我们使用了 `ModelViewSet` 类来获取完整的默认读写操作。

请注意，我们还使用 `@detail_route` 装饰器创建一个名为 `highlight` 的自定义操作。这个装饰器可用于添加不符合标准 `create` / `update` / `delete` 样式的任何自定义路径。

默认情况下，使用 `@detail_route` 装饰器的自定义操作将响应 `GET` 请求。如果我们想要一个响应 `POST` 请求的动作，我们可以使用 `methods` 参数。

默认情况下，自定义操作的URL取决于方法名称本身。如果要更改URL的构造方式，可以为装饰器设置url_path关键字参数。

> 以上的内容对于新手来说，太难理解了。不看后面的API参考，你根本无法明白它的意思。放在快速入门教程里，真的好吗？

至此，完整的，包括被注释掉的，用于对比的，先前的视图，整个veiws.py的代码如下（调整了import语句的位置）：

```
from snippets.models import  Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics
from django.contrib.auth.models import User
from snippets.serializers import UserSerializer
from rest_framework import permissions
from snippets.permissions import  IsOwnerOrReadOnly


from rest_framework.decorators import  api_view
```

```
11  from rest_framework.response import Response
12  from rest_framework.reverse import reverse
13  from rest_framework import renderers
14
15  from rest_framework import viewsets
16  from rest_framework.decorators import action
17  from rest_framework.response import Response
18
19
20  @api_view(['GET'])
21  def api_root(request, format=None):
22      return Response({
23          'users': reverse('user-list', request=request, format=format),
24          'snippets': reverse('snippet-list', request=request, format=format)
25      })
26
27
28  # class SnippetHighlight(generics.GenericAPIView):
29  #     queryset = Snippet.objects.all()
30  #     renderer_classes = (renderers.StaticHTMLRenderer,)
31  #
32  #     def get(self, request, *args, **kwargs):
33  #         snippet = self.get_object()
34  #         return Response(snippet.highlighted)
35  #
36  #
37  # class SnippetList(generics.ListCreateAPIView):
38  #     queryset = Snippet.objects.all()
39  #     serializer_class = SnippetSerializer
40  #     permission_classes = (permissions.IsAuthenticatedOrReadOnly,)
41  #
42  #     def perform_create(self, serializer):
43  #         serializer.save(owner=self.request.user)
44  #
45  #
46  # class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
47  #     queryset = Snippet.objects.all()
48  #     serializer_class = SnippetSerializer
49  #     permission_classes = (permissions.IsAuthenticatedOrReadOnly,
50  #                           IsOwnerOrReadOnly,)
51
```

```
52
53  class SnippetViewSet(viewsets.ModelViewSet):
54      """
```

```
55          This viewset automatically provides `list`, `create`,  `retrieve`,
56          `update` and `destroy` actions.
57
58          Additionally we also provide an extra `highlight` action.
59          """
60          queryset = Snippet.objects.all()
61          serializer_class = SnippetSerializer
62          permission_classes = (permissions.IsAuthenticatedOrReadOnly,
63                                IsOwnerOrReadOnly,)
64
65          @action(detail=True, renderer_classes=[renderers.StaticHTMLRenderer])
66          def highlight(self, request, *args, **kwargs):
67              snippet = self.get_object()
68              return Response(snippet.highlighted)
69
70          def perform_create(self, serializer):
71              serializer.save(owner=self.request.user)
72
73
74      # class UserList(generics.ListAPIView):
75      #     queryset = User.objects.all()
76      #     serializer_class = UserSerializer
77      #
78      #
79      # class UserDetail(generics.RetrieveAPIView):
80      #     queryset = User.objects.all()
81      #     serializer_class = UserSerializer
82
83
84      class UserViewSet(viewsets.ReadOnlyModelViewSet):
85          """
86          This viewset automatically provides `list` and `detail` actions.
87          """
88          queryset = User.objects.all()
89          serializer_class = UserSerializer
90
```

二、显式地将ViewSets绑定到URL路由上

既然视图类发生了改变，那么我们的路由也必须针对性的调整。 在类绑定到一组具体视图中。 `urls.py` 文件中，我们将 `ViewSet`

```
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 =
```

上面的不理解也没关系，因为我们马上要删掉它。如果看不懂，等你以后再来看也行。

现在我们可以像通常一样在URL conf中注册视图。

```
    urlpatterns = format_suffix_patterns([
        path('', api_root),
        path('snippets/', snippet_list, name='snippet-list'),
        path('snippets/<int:pk>/', snippet_detail, name='snippet-detail'),
        path('snippets/<int:pk>/highlight/', snippet_highlight, name='snippet-
 highlight'),
        path('users/', user_list, name='user-list'),
    path('users/<int:pk>/', user_detail, name='user-detail') 8
    ])
```

三、使用DRF提供的路由器Router

DRF为 `ViewSet` 视图集，设计了专门的路由器类Router。 `Router` 类专门为ViewSet提供全自动的urlconf。我们需要做的就是使用路由器注册相应的视图集，然后让它执行其余操作。

上面的内容全删了，重写 `urls.py` 文件。

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

# 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.
urlpatterns = [
    path('', include(router.urls)),
]
```

使用路由器注册viewsets类似于提供urlpattern。我们包含两个参数 - 视图的URL前缀和视图本身。

`DefaultRouter` 类也会自动为我们创建API根视图，因此可以从views.py中删除 `api_root` 方法。

完整的snippets/urls.py内容如下（带先前版本的对比，被注释了）：

```
# from django.urls import path
# from snippets import views
#
# urlpatterns = [
#     path('snippets/', views.snippet_list),
#     path('snippets/<int:pk>/',  views.snippet_detail),
# ]


# from django.urls import path
# from rest_framework.urlpatterns import format_suffix_patterns
# from snippets import views

# urlpatterns = [
#     path('snippets/', views.SnippetList.as_view()),
#     path('snippets/<int:pk>/',  views.SnippetDetail.as_view()),
#     path('users/', views.UserList.as_view()),
```

```
#     path('users/<int:pk>/', views.UserDetail.as_view()),
#     path('', views.api_root),
#     path('snippets/<int:pk>/highlight/',
views.SnippetHighlight.as_view()),
# ]


# 略微调整了一下顺序
# urlpatterns = [
#     path('', views.api_root),
#     path('snippets/', views.SnippetList.as_view(),  name='snippet-list'),
#     path('snippets/<int:pk>/', views.SnippetDetail.as_view(),
name='snippet-detail'),
#     path('snippets/<int:pk>/highlight/',
views.SnippetHighlight.as_view(),  name='snippet-highlight'),
#     path('users/', views.UserList.as_view(), name='user-list'),
#     path('users/<int:pk>/', views.UserDetail.as_view(), name='user-
detail')
# ]
#
# urlpatterns = format_suffix_patterns(urlpatterns)

from django.urls import path, include
from rest_framework.routers import DefaultRouter
from snippets import views

# 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.
urlpatterns = [
path('', include(router.urls)), 48 ]
```

四、三种视图编写方法之间的权衡

使用视图集可以最大限度地减少代码量，让你能够专注于API提供的交互和表示，而不是URLconf的细节。但这并不是说它就是最佳最好最优的解决方案，事实上，抽象得越多，可定制性就越低，适用的场景就越少。

三种视图的构建方法：

- 基于函数的视图 @api_view
- 基于类的视图 APIView GenericView ListModelView
- 基于视图集的视图 ViewSet

没有绝对的好坏，使用哪种取决于你的需求。

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

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

09-快速入门7---概要和coreapi-客户端(不重要)

概要是一种机器可读的文档，用于描述可用的API，其URLS，以及它们支持的操作。

概要可用于自动生成文档，也可以用于驱动可与API进行交互的动态客户端库。

一、Core API

为了提供概要支持，REST框架引用了Core API规范。

Core API是用于描述API的文档规范。它用于提供API的内部表示形式和交互方式。可同时用于服务器端或客户端。

当在服务器端使用时，Core API支持以各种模式或超媒体格式呈现。

当在客户端使用时，Core API允许动态驱动的客户端库与任何公开支持的模式或超媒体格式的API交互。

二、添加概要

REST框架支持明确定义的概要视图，也可以自动生成概要。由于教程走到这里，我们使用的是视图集和路由器，所以可以简单地使用自动生成概要的方式。

我们需要安装`coreapi` python包才能生成API概要，还需要安装pyyaml库，渲染概要，使之成为通用的基于YAML格式的OpenAPI 。

```
$ pip install coreapi pyyaml
```

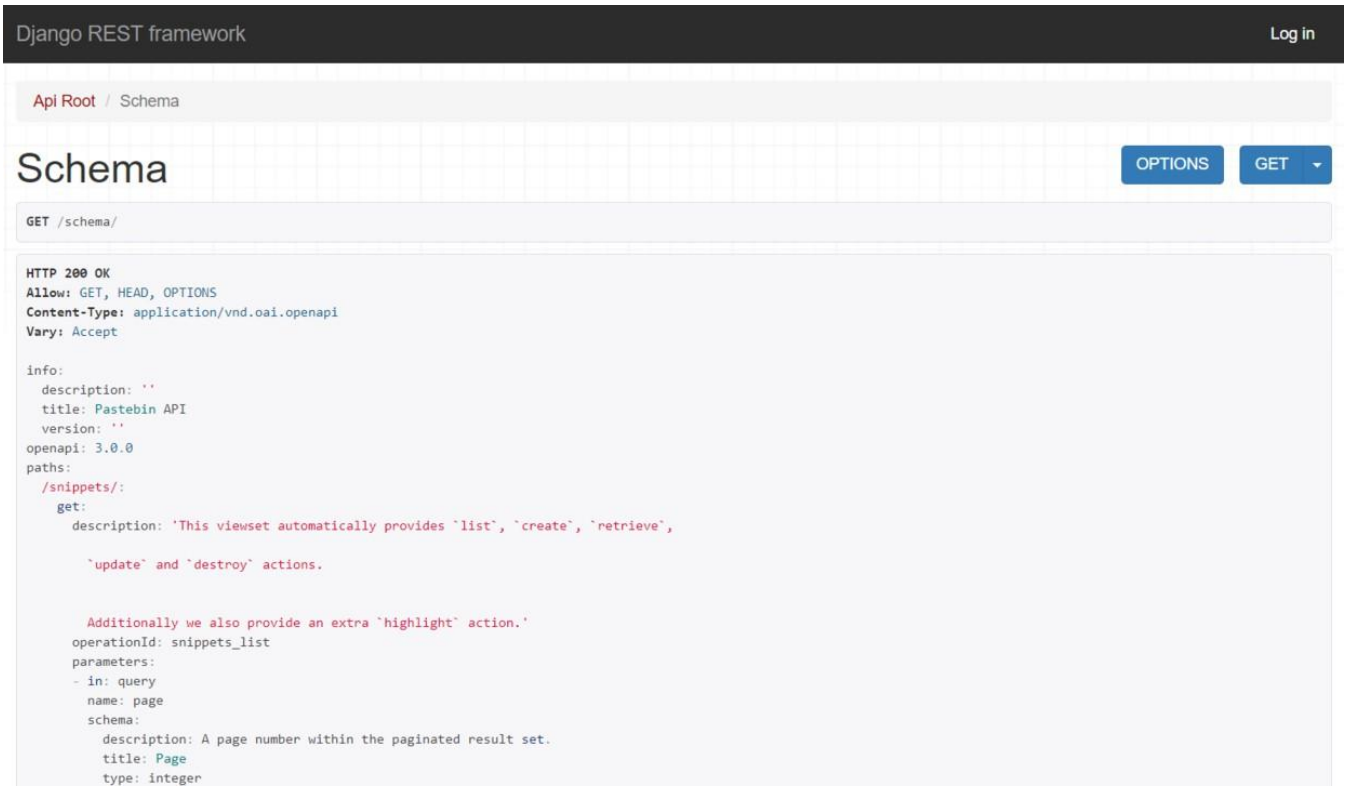
现在我们可以通过在URL配置中包含一个自动生成的概要视图来为API添加概要。

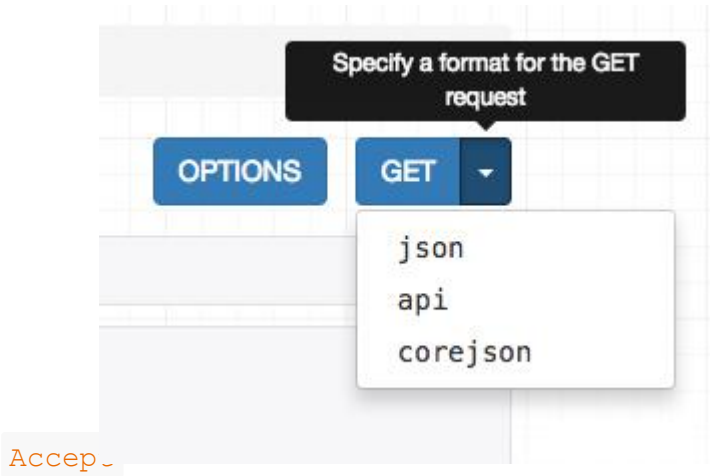
在根路由urls.py下，灵活地插入下面的代码：

```
from rest framework.schemas import  get schema view

schema_view = get_schema_view(title='Pastebin  API')

urlpatterns = [
    path('schema/', schema_view),
    ...
]
```

`corejson`

重启服务器，在浏览器中访问 `http://127.0.0.1:8000/schema/` ，可以看到 成为可用选项之一。





`Accep`

我们也可以通过在 标头中指定所需的内容类型从命令行请求概要。

```
$ http http://127.0.0.1:8000/schema/ Accept:application/coreapi+json



HTTP/1.1 200 OK
Allow: GET, HEAD, OPTIONS
Content-Length: 1919
Content-Type:  application/coreapi+json
```

```
Date: Sun, 28 Apr 2019 15:01:04  GMT
Server: WSGIServer/0.2 CPython/3.7.3
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "_meta": {
        "title": "Pastebin API",
        "url":  "http://127.0.0.1:8000/schema/"
    },
    "_type": "document",
    "snippets": {
    ...
```

默认输出样式是使用Core JSON编码。

还支持其他概要格式，如Open API（以前叫Swagger）。

三、 使用命令行客户端与API进行交互

现在我们的API暴露了一个概要url，我们可以使用一个动态的客户端库与API进行交互。为了演示这个，我们来使用Core API命令行客户端。

先安装需要的 `coreapi-cli` 包：

```
$ pip install coreapi-cli
```

检查一下安装是否成功：

```
$ coreapi


Usage: coreapi [OPTIONS] COMMAND [ARGS]...

  Command line client for interacting with CoreAPI services.

  Visit http://www.coreapi.org for more  information.

Options:
  --version  Display the package version  number.
  --help     Show this message and exit.

Commands:
```

```
  action        Interact with the active document.
  bookmarks     Add, remove and show bookmarks.
  clear         Clear the active document and other state.
  codecs        Manage the installed codecs.
  credentials   Configure request credentials.
  describe      Display description for link at given PATH.
  dump          Dump a document to console.
  get           Fetch a document from the given URL.
  headers       Configure custom request headers.
  history       Navigate the browser history.
  load          Load a document from disk.
  reload        Reload the current document.
  show          Display the current  document.
```

首先，使用命令行客户端加载API概要：

```
$ coreapi get  http://127.0.0.1:8000/schema/



<Pastebin API "http://127.0.0.1:8000/schema/">
    snippets: {
        list([page])
        read(id)
        highlight(id)
    }
    users: {
        list([page])
        read(id)
    }
```

我们还没有认证，所以现在只能看到只读的API，这与我们设置的API权限是一致的。

使用命令行客户端，尝试列出现有的代码片段：

```
$ coreapi action snippets  list

{
    "count": 3,
    "next": null,
    "previous": null,
    "results": [
        {
            "url":  "http://127.0.0.1:8000/snippets/1/",
```

```
            "id": 1,
            "highlight": "http://127.0.0.1:8000/snippets/1/highlight/",
            "owner": "admin",
            "title": "test",
            "code": "print('hello')",
            "linenos": false,
            "language": "abap",
            "style": "abap"
        },
        {
            "url": "http://127.0.0.1:8000/snippets/2/",
            "id": 2,
            "highlight": "http://127.0.0.1:8000/snippets/2/highlight/",
            "owner": "admin",
            "title": "haha",
            "code": "import this",
            "linenos": false,
            "language": "abap",
            "style": "abap"
        },
        {
            "url": "http://127.0.0.1:8000/snippets/3/",
            "id": 3,
            "highlight": "http://127.0.0.1:8000/snippets/3/highlight/",
            "owner": "admin",
            "title": "what is new",
            "code": "print('i dont know')",
            "linenos": false,
            "language": "abap",
            "style": "abap"
        }
    ]
}
```

访问一些API需要提供关键字参数。例如，要获取特定代码片段的高亮HTML表示，我们需要提供一个id：

```
$ coreapi action snippets highlight --param  id=1

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML  4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">

<html>
<head>
```

```
    <title>test</title>
    <meta http-equiv="content-type" content="text/html; charset=None">
    <style type="text/css">
td.linenos { background-color: #f0f0f0; padding-right: 10px; }
span.lineno { background-color: #f0f0f0; padding: 0 5px 0 5px; }
pre { line-height: 125%; }
...
...
body .il { color: #33aaff } /* Literal.Number.Integer.Long  */

    </style>
</head>
<body>
<h2>test</h2>

<div class="highlight"><pre><span></span><span class="nv">print</span><span
class="p">(</span><span class="s1">&#39;hello&#39;</span><span cla
ss="p">)</span>
</pre></div>
</body>
</html>

```

带认证参数进行API访问

如果我们想要创建，编辑和删除代码片段，我们需要进行合法的用户身份验证。在教程中，我们只需使用基本的auth。

请确保使用实际的用户名和密码替换下面的 `<username>` 和 `<password>` 。

```
$ coreapi credentials add 127.0.0.1 <username>:<password> --auth basic



Added credentials
127.0.0.1 "Basic YWRtaW46MTIzNGFzZGY="

```

现在，如果我们再次访问概要API，我们将获得完整可用的操作权限。

```
 $ coreapi reload

```

```
3
4   <Pastebin API "http://127.0.0.1:8000/schema/">
5       snippets: {
6           list([page])
7           create(code, [title], [linenos], [language], [style])
8           read(id)
9           update(id, code, [title], [linenos], [language], [style])
10          partial_update(id, [title], [code], [linenos], [language], [style])
11          delete(id)
12          highlight(id)
13      }
14      users: {
15          list([page])
16          read(id)
17      }
```

我们现在能够与这些API行交互。例如，要创建一个新的代码片段:

```
1   $ coreapi action snippets create --param title="Example" --param
    code="print('hello, world')"
2
3
4   {
5       "url": "http://127.0.0.1:8000/snippets/4/",
6       "id": 4,
7       "highlight": "http://127.0.0.1:8000/snippets/4/highlight/",
8       "owner": "admin",
9       "title": "Example",
10      "code": "print('hello, world')",
11      "linenos": false,
12      "language": "python",
13      "style": "friendly"
14  }
```

或者删除一个代码片段:

```
1   $ coreapi action snippets delete --param id=7
```

似乎和HTTPie差不多。

最后

DRF的官方快速入门教程到这里就结束了。