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。

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：

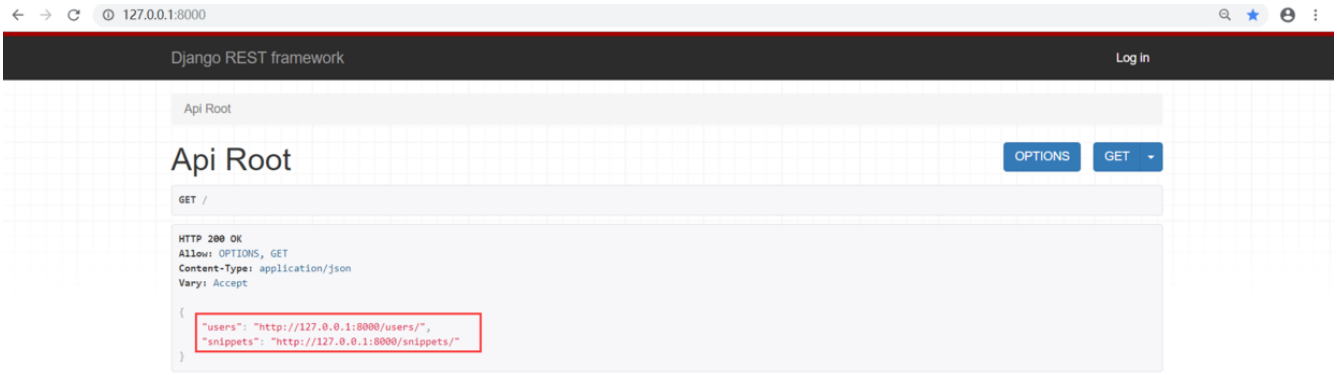
```
path('snippets/<int:pk>/highlight/', views.SnippetHighlight.as_view()),
```

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。

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')
```

> 译者注：这个教程的脑洞很大，跳跃性很强，在进行过程中，有时候会出错，是因为还没有完成后面的工作，这里只是过程稿。

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'` ，我们还需要在 `highlight` 字段上指出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
}
```

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。

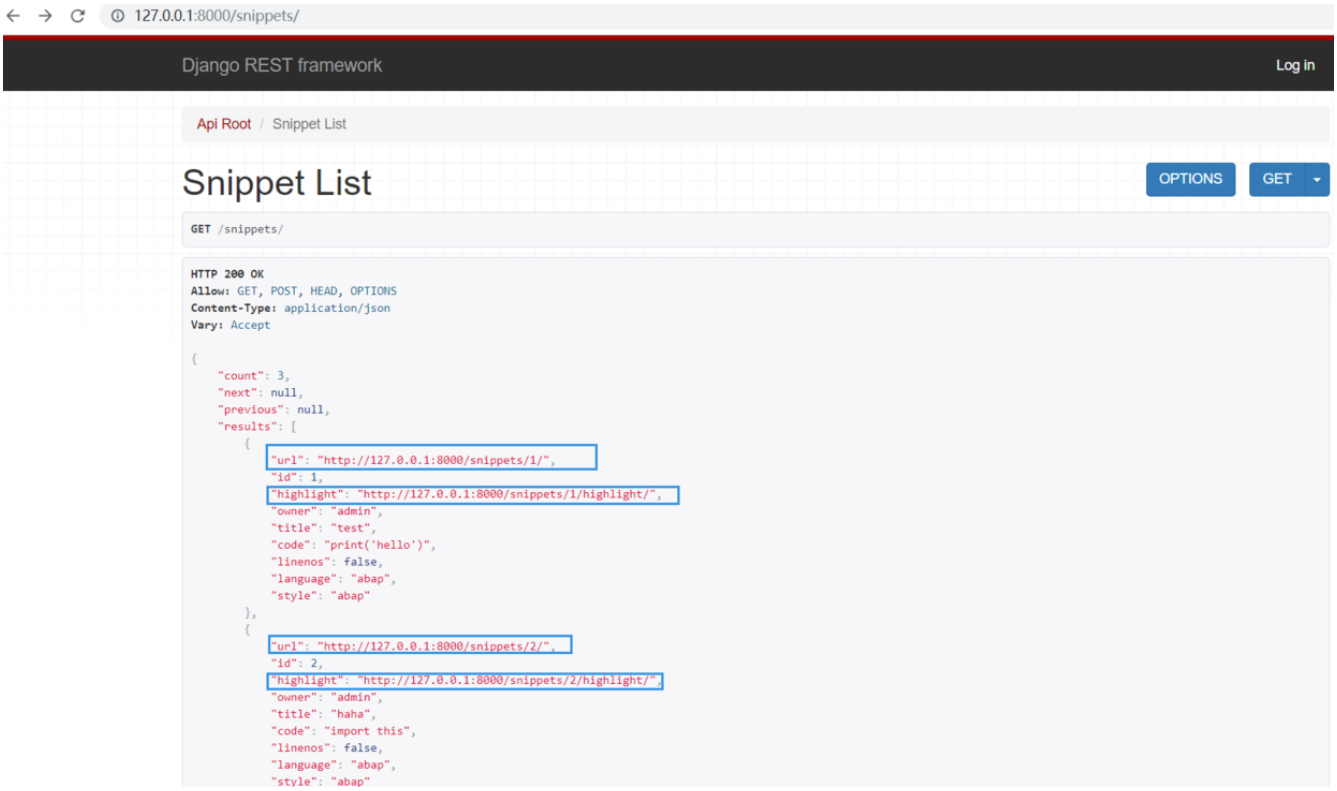
如果需要的话，我们也可以自定义分页风格，但在这个教程中，我们将一直使用默认设置。

六、浏览API

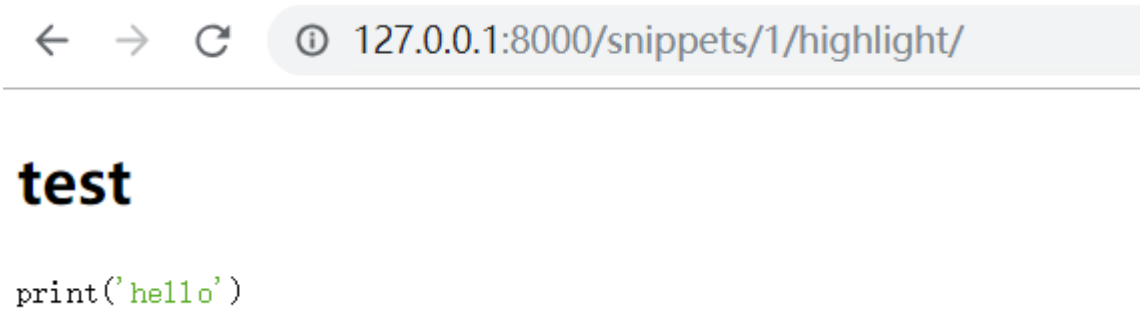
好了，可以打开浏览器并浏览我们的API了，你可以简单的通过页面上的超链接来了解API。

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。