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 `pygments` 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` 文件中添加下面的代码，UserSerializer类就是我们为那个user字段写的序列化类：

```
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：

```python
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：

```python
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

```

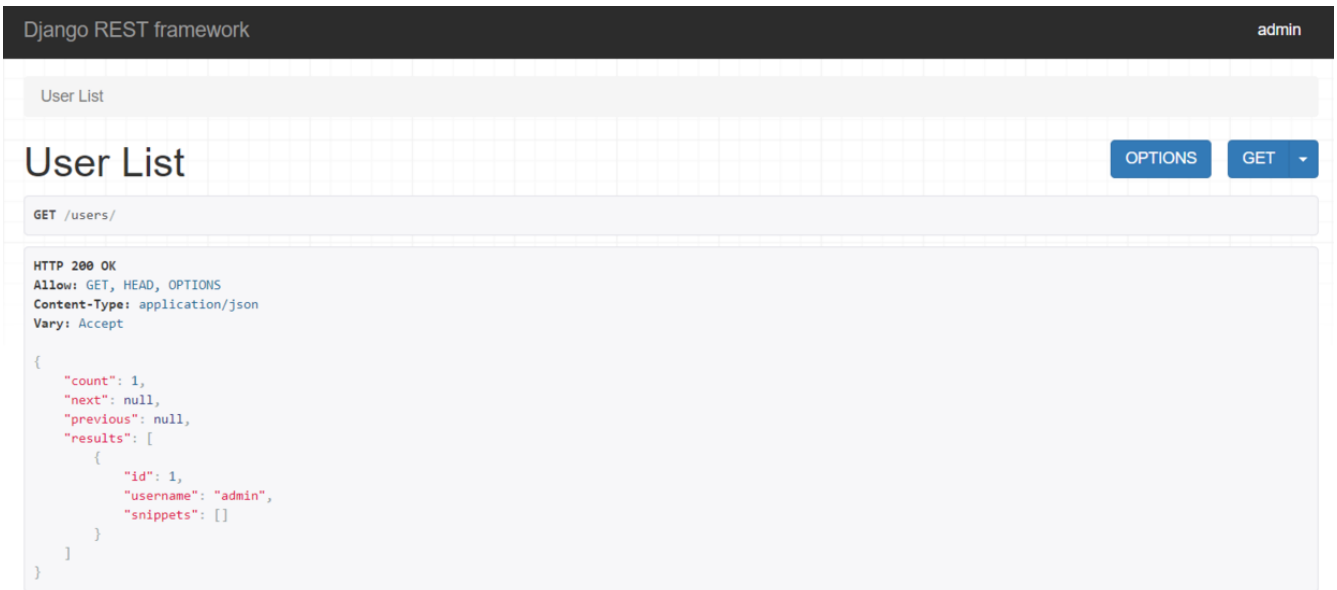
最后，我们还需要在URL conf中添加路由。将以下内容添加到 `snippets.urls.py` 文件的urlpatterns中。

```
    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！

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：

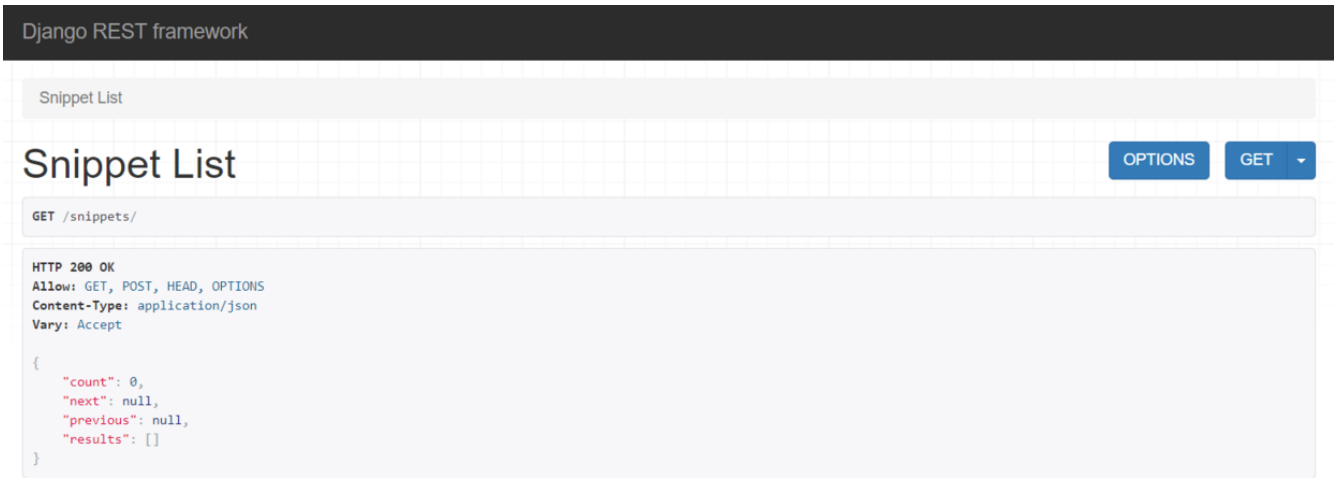
```
1   owner = serializers.ReadOnlyField(source='owner.username')
```

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,)
```

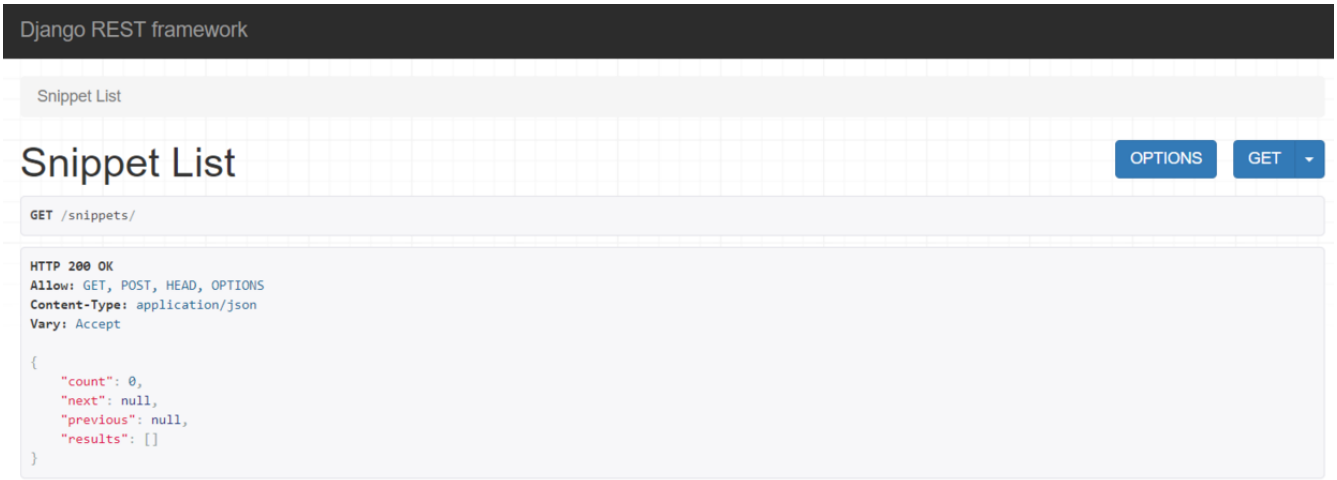
目前为止，views.py的完整内容如下：

```
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：

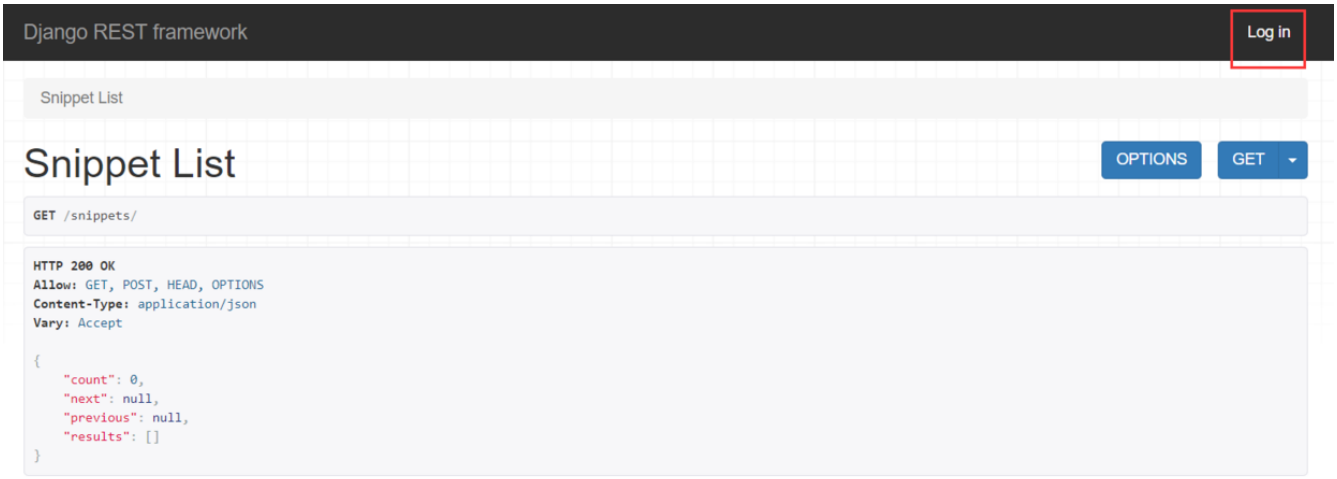
```
1  from django.conf.urls import include
```

在文件末尾添加下面的代码：

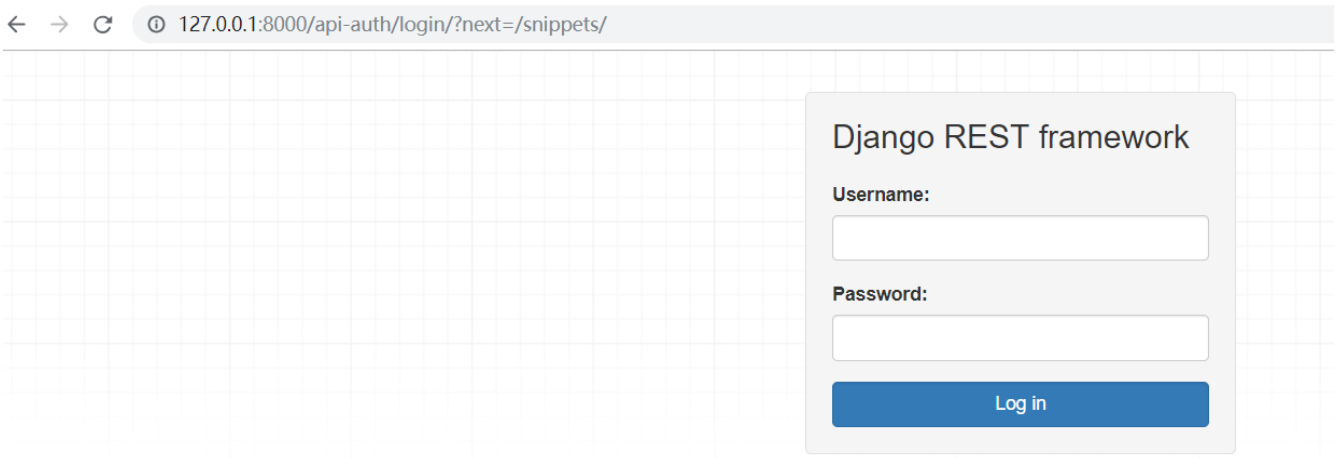
```
1  urlpatterns += [
2      path('api-auth/', include('rest_framework.urls')),
3  ]
```

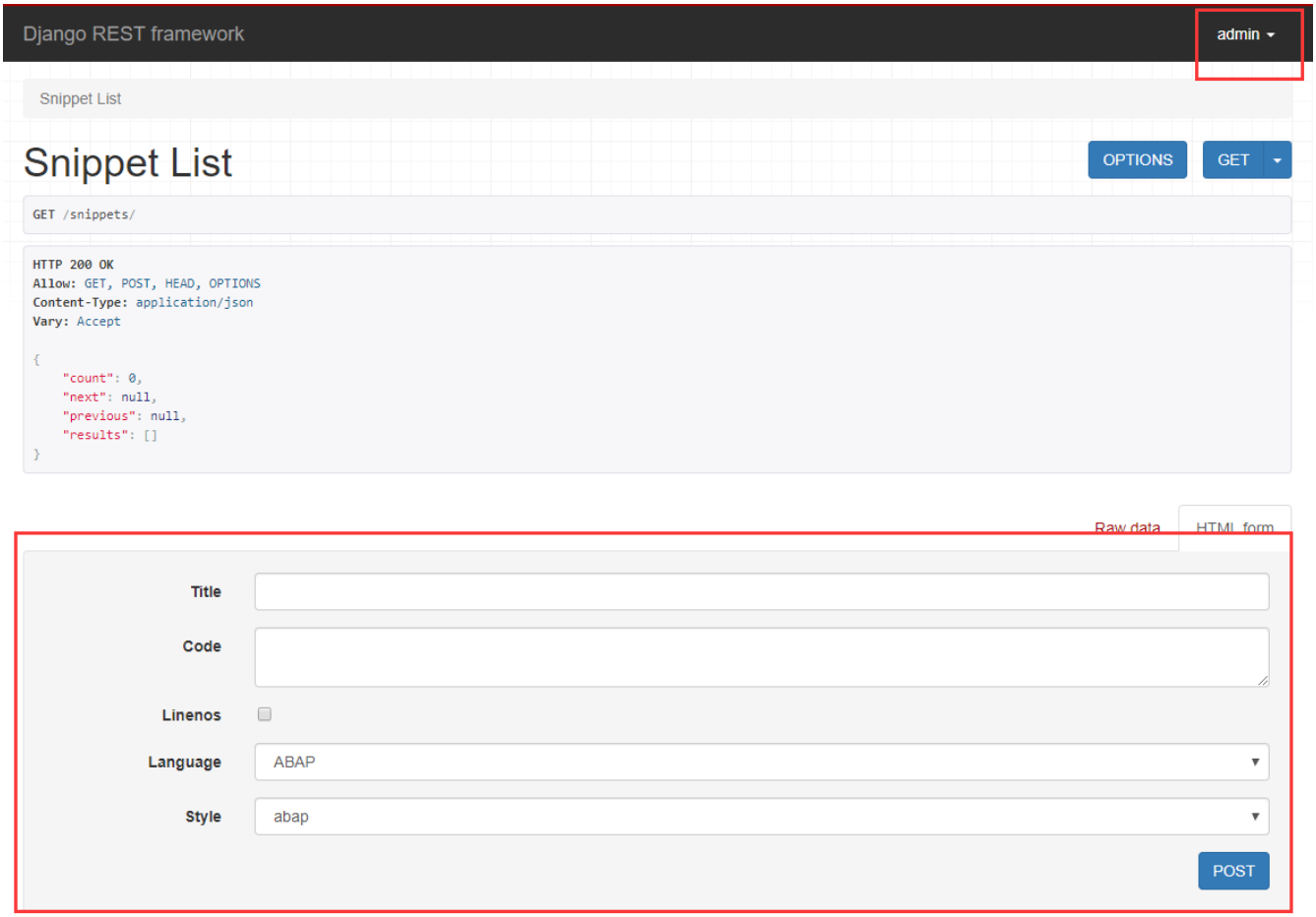
模式的 `api-auth/'` 部分实际上可以是任何你想使用的字符串。

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。

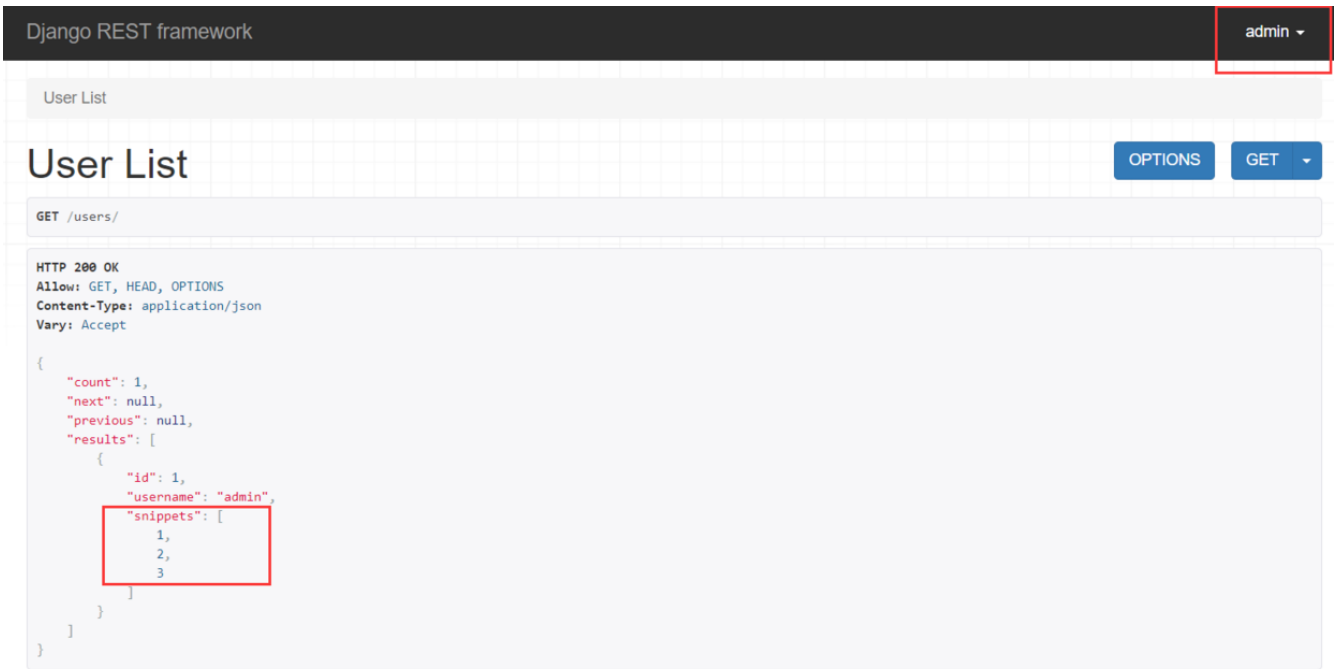

Django REST framework
Log in
Snippet List
Snippet List
OPTIONS
GET
GET /snippets/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
"count": 0,
"next": null,
"previous": null,
"results": []
}

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。

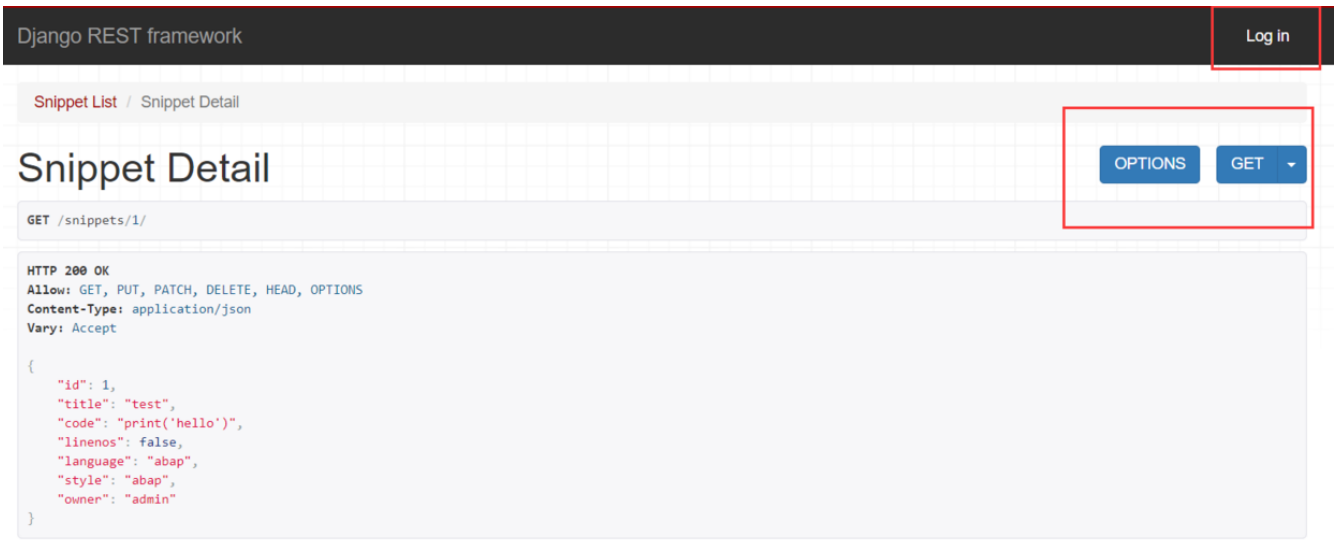
在snippets这个app中，创建一个新文件 `permissions.py` ，并写入下面的代码：

```
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：

```
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"
}
```