

## 前言

2020年12月3日 21:31

建议直接看代码，根据model-Serializers-view-url的思想，去了解djangoREST的整个流程。我这个pdf比较分散，更像一个中文的tips，不建议直接看pdf来学习。

个人意见：

- 1.了解model-Serializers-view-url的流程
- 2.了解model的相关字段设置
- 3.了解django对数据库的操作，比如搜索、更新、对外键所在表的搜索等等
- 4.熟悉如何运行django项目
- 5.熟悉如何安装mysql数据库、如何在django中建立数据库、如何配置pycharm

# pycharm配置

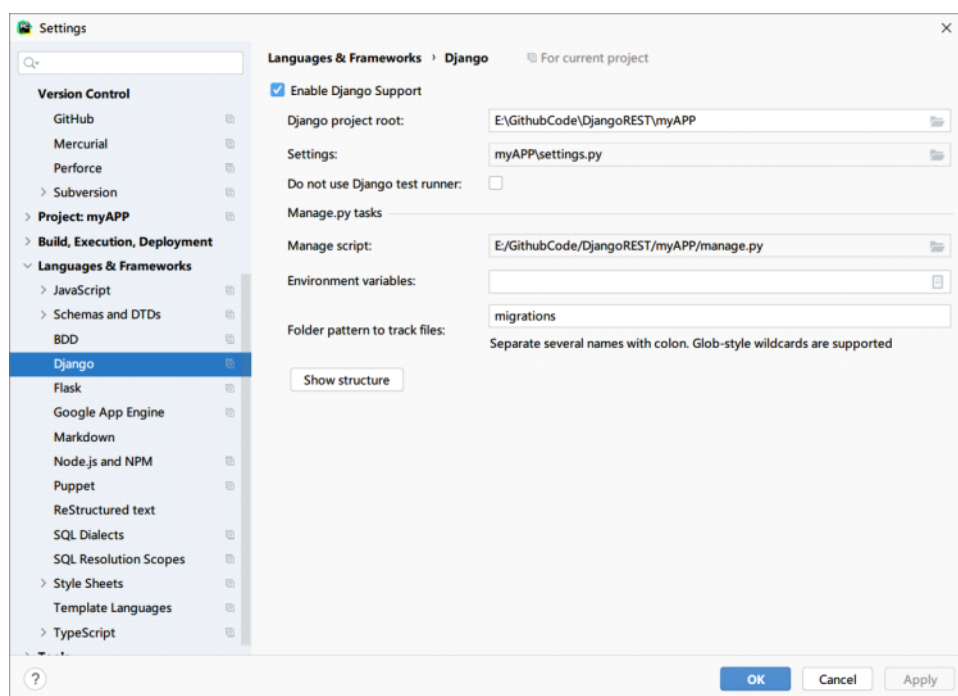
2020年10月30日 21:01

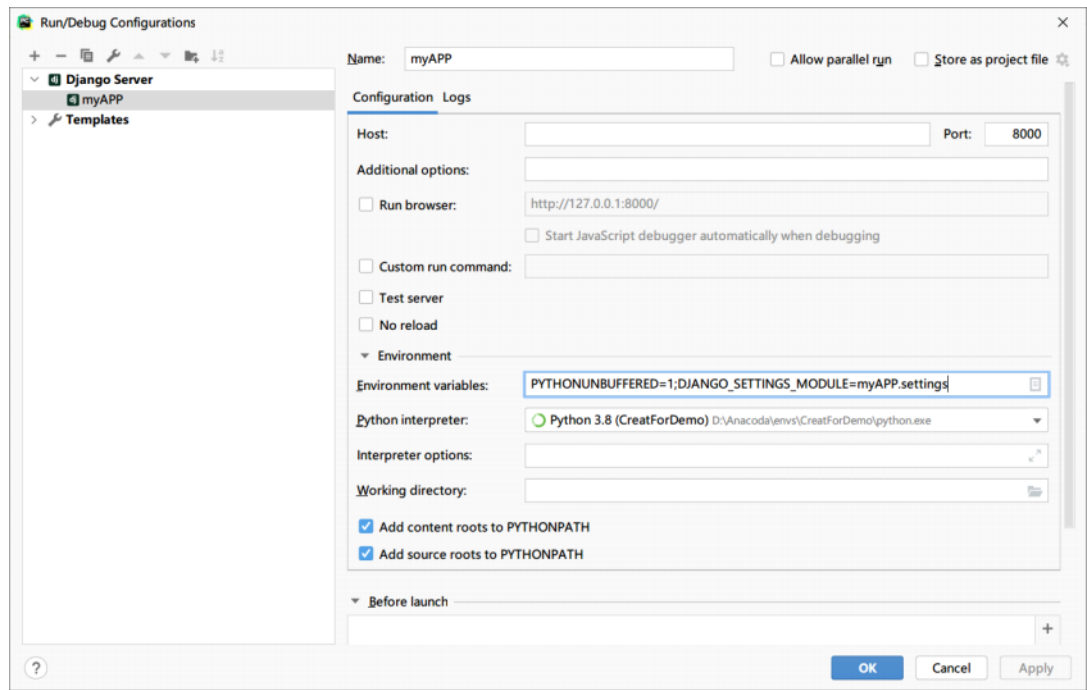
个人建议：

安装anaconda

导入环境

配置pycharm如下

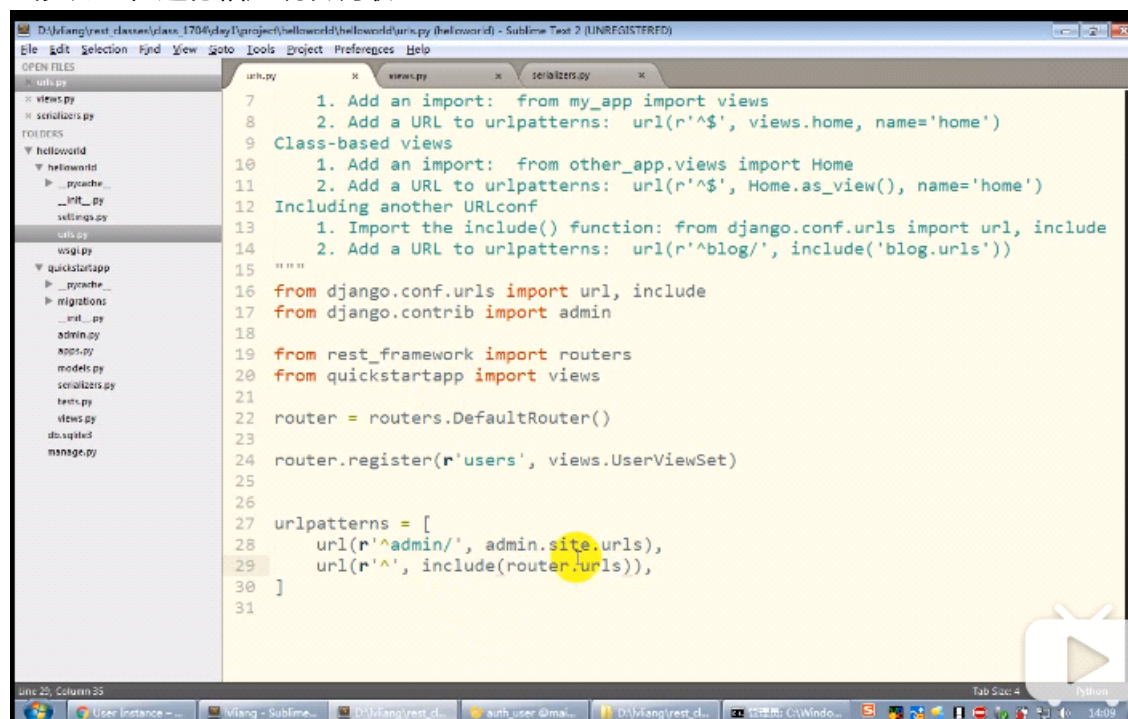




# 基础

2020年10月16日 17:39

- 1.配置文件中：加入框架
- 2.修改url，进行相应跳转调取

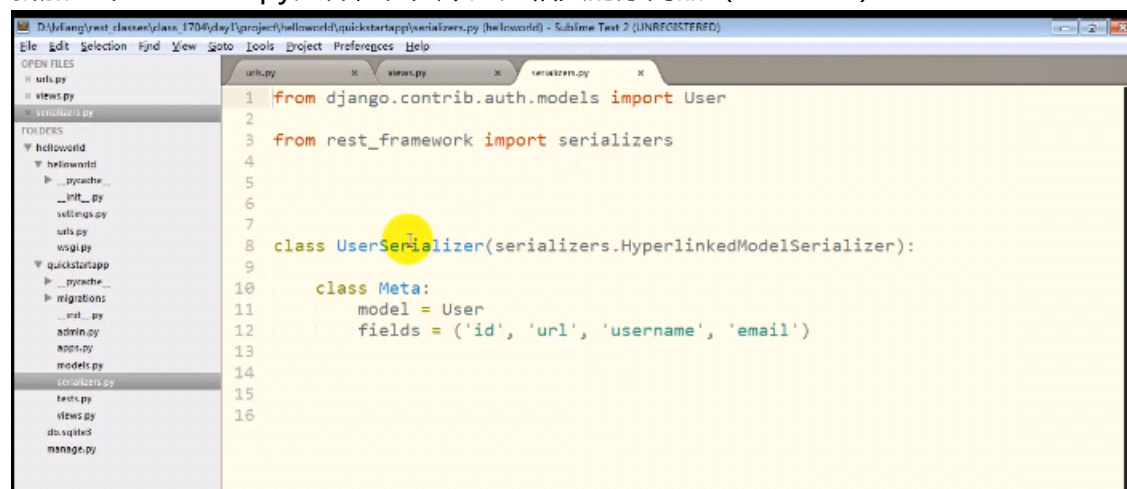


- 1.定义一个默认router
- 2.对router注册为UserViewSet，前面的'users'其实就是设定默认的根目录名称  
这句话就实现了列表的创建、详情、更新、删除操作
- 3.url(r'^',include(router.urls)) ## 可以自动生成对应的地址

## 3.对应的项目文件中

view中加入ViewSet，其实就是方法调用

在当前的操作中，就是获取全部的列表，并把序列化器类型定义为相关的序列化器类  
新加一个serializer.py文件，在其中定义相关的序列化器（serializer）



Meta，元数据，用来在详情页进行显示，注意，元数据里引用的fields，必须是model中定义过的内容，比如没有定义地址，就没办法加入地址显示

也有例外：groups，多对多关系，表示所属的群组，这个是django自动生

成的表

开发顺序:

注: django框架中, 自带账户管理模块, 先注册一个超级管理员, 其他后面再说

1.model中建立模型 (group是django自带的model, 不用建立了)

2.加一个group的序列器, Hyperlinkd是超链接, 这个在管理的时候就能看出来, 是点击链接进行调转的

序列器中还包含了要展现每个数据的哪几项, 比如group中就可以加user\_set

序列器还会自动验证, 不会出现重名情况

```
14
15
16 class GroupSerializer(serializers.HyperlinkedModelSerializer):
17
18     class Meta:
19         model = Group
20         fields = ('id', 'url', 'name',)
21
22
23
```

3.到view里加入viewSet, 这个的功能应该是展现什么, 现在是展现全部列表。应该是根据querylist

```
class GroupViewSet(viewsets.ModelViewSet):
    """
    查看、编辑用户群组的API接口。
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
```

4.然后在url中, 注册, 表示可以被调用, 要是不注册, 上面那些相当于白写了

```
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)
```

5.如果分页显示, 在settings.py里加一行代码即可

```
123
124
125 REST_FRAMEWORK = {
126     'PAGE_SIZE': 3,
127 }
128
129
130
```

6.正序排列或者倒序排列, 其实是根据序列器中的queryset的

也可以实现前端指定排序实现, 这个后面再讲

```
9     queryset = User.objects.all().order_by('-date_joined')
10     serializer_class = UserSerializer
1
2
```

7.登陆功能。名字, include指定, 没啥好说的

```

6 urlpatterns = [
7     url(r'^admin/', admin.site.urls),
8     url(r'^$', include(router.urls)),
9
10    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'
11                                )),
12
13 ]
14 ]
15 ]

```

## 8.利用登陆进行权限控制

默认许可类:

目前只有一项，对rest框架要求，必须是管理员权限。这样就可以实现了必须【登陆】才可以【读和改】

登录了也不一定能看，权限没定义呢，在自带的模块里设置

```

124
125 REST_FRAMEWORK = {
126     'PAGE_SIZE': 3,
127     'DEFAULT_PERMISSION_CLASSES': [
128         'rest_framework.permissions.IsAdminUser',
129     ],
130 }
131
132
133

```

# 五级代码

2020年10月17日 14:44

在view.py中，**第一级别的代码**是这样的

这个函数实现了**列表和创建**功能，其实就是从数据库中获取到全部，然后  
再返回给前端  
分为get和post方法，然后获得  
设置序列化器，实现转换格式。（python和json转换）  
返回序列化器的data

```
15
16 @csrf_exempt
17 def snippet_list(request):
18     """
19     列表、创建 LC
20     """
21     if request.method == 'GET':
22         snippets = Snippet.objects.all()
23         serializer = SnippetSerializer(snippets, many=True)
24         return JsonResponse(serializer.data, safe=False)
25
26     elif request.method == 'POST':
27         data = JSONParser().parse(request)
28         serializer = SnippetSerializer(data=data)
29         if serializer.is_valid():
30             serializer.save()
31             return JsonResponse(serializer.data, status=201)
32         return JsonResponse(serializer.errors, status=400)
33
34
35 @csrf_exempt
36 def snippet_detail(request, pk):
37     """
38     Retrieve, update or delete a code snippet.
39     """
40     try:
```

进行数据转换，此处因为是利用了.all()函数，因此many=True必写

序列化器的另一个作用：验证是否合法

这个是获取、更新、删除

```
34
35 @csrf_exempt
36 def snippet_detail(request, pk):
37     """
38     获取、更新、删除 RUD
39     """
40     try:
41         snippet = Snippet.objects.get(pk=pk)
42     except Snippet.DoesNotExist:
43         return HttpResponse(status=404)
44
45     if request.method == 'GET':
46         serializer = SnippetSerializer(snippet)
47         return JsonResponse(serializer.data)
48
49     elif request.method == 'PUT':
50         data = JSONParser().parse(request)
51         serializer = SnippetSerializer(snippet, data=data)
52         if serializer.is_valid():
53             serializer.save()
54             return JsonResponse(serializer.data)
55         return JsonResponse(serializer.errors, status=400)
56
57     elif request.method == 'DELETE':
58         snippet.delete()
59         return HttpResponse(status=204)
```

装饰器，这里只是用来解决跨域问题

put方法，用于更新

必须要先验证

删除

204表示删除成功



调用这两个函数的方法，是和原来很像的

```
1 from django.conf.urls import url
2 from snippetsapp import views
3
4 urlpatterns = [
5     url(r'^snippets/$', views.snippet_list),
6     url(r'^snippets/(?P<pk>[0-9]+)/$', views.snippet_detail),
7 ]
8
9
10
11
```

## 第二级别的代码，还是函数，因此只理解

Url.py的代码是通用的，view中如下

```
1 from rest_framework import status
2 from rest_framework.decorators import api_view
3 from rest_framework.response import Response
4 from snippetsapp.models import Snippet
5 from snippetsapp.serializers import SnippetSerializer
6
7 装饰器，表示允许get和post方法
8
9 @api_view(['GET', 'POST'])
10 def snippet_list(request, format=None):
11     """
12     LC
13     """
14     if request.method == 'GET':
15         snippets = Snippet.objects.all()
16         serializer = SnippetSerializer(snippets, many=True)
17         return Response(serializer.data)
18
19     elif request.method == 'POST':
20         serializer = SnippetSerializer(data=request.data)
21         if serializer.is_valid():
22             serializer.save()
23             return Response(serializer.data, status=status.HTTP_201_CREATED)
24         return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
25
26 返回变了，第一级是JsonResponse
27 现在用自己的Response，就可以返回成界面，可视化了
28
29 POST中，也不需要先解析json再使用了
30 可以理解为request帮助解析了
31
32 用自己的Response，而且返回status变了，更加清晰
```

url.py其实也可以改成如下

```
1
2
3 from django.conf.urls import url
4 from rest_framework.urlpatterns import format_suffix_patterns
5 from snippetsapp import views
6
7 urlpatterns = [
8     url(r'^snippets/$', views.snippet_list),
9     url(r'^snippets/(?P<pk>[0-9]+)/$', views.snippet_detail),
10 ]
11
12 urlpatterns = format_suffix_patterns(urlpatterns)
13
14
15
```

这样子，直接改为.json就可以返回为json格式的值，为前端服务  
但是直接改后缀不太好



因此，用到了format参数，format=json的时候，就返回json格式了

第二级比第一级强的地方：

用Response替代了原来的JsonResponse，可以返回可视化界面了

POST请求不需要先转格式再使用

使用format，可以根据参数返回api或者json（url中也修改了）

装饰器中，可以限制get，post方法

## 第三级别的代码，实践中使用

Url.py中，三和二对比，其实就是url中不一样

```
4 from django.conf.urls import url
5 from rest_framework.urlpatterns import format_suffix_patterns
6 from snippetsapp import views
7
8 urlpatterns = [
9     url(r'^snippets/$', views.SnippetList.as_view(), name='snippet-list'),
10    url(r'^snippets/(?P<pk>[0-9]+)/$', views.SnippetDetail.as_view(), name='
        snippet-detail'),
11 ]
12
13 urlpatterns = format_suffix_patterns(urlpatterns)
14
15
16
17
18 # from django.conf.urls import url
19 # from rest_framework.urlpatterns import format_suffix_patterns
20 # from snippetsapp import views
21
22 # urlpatterns = [
23 #     url(r'^snippets/$', views.snippet_list),
24 #     url(r'^snippets/(?P<pk>[0-9]+)/$', views.snippet_detail),
25 # ]
26
27 # urlpatterns = format_suffix_patterns(urlpatterns)
28
29
30
```

view中

```
11
12 from snippetsapp.models import Snippet
13 from snippetsapp.serializers import SnippetSerializer
14 from django.http import Http404
15 from rest_framework.views import APIView
16 from rest_framework.response import Response
17 from rest_framework import status
18
19
20 class SnippetList(APIView): 继承APIView，最基本的类
21
22     LC
23
24     def get(self, request, format=None):
25         snippets = Snippet.objects.all()
26         serializer = SnippetSerializer(snippets, many=True)
27         return Response(serializer.data)
28
29     def post(self, request, format=None):
30         serializer = SnippetSerializer(data=request.data)
31         if serializer.is_valid():
32             serializer.save()
33             return Response(serializer.data, status=status.HTTP_201_CREATED)
34         return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
35
36
```

定义方法来区分，而不是if

```

class SnippetDetail(APIView):
    """
    RUD
    """
    def get_object(self, pk):
        try:
            return Snippet.objects.get(pk=pk)
        except Snippet.DoesNotExist:
            raise Http404

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

    def put(self, request, pk, format=None):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk, format=None):
        snippet = self.get_object(pk)

```

## 第3.5级别的代码，主要是使用mixins

```

13
14 from snippetsapp.models import Snippet
15 from snippetsapp.serializers import SnippetSerializer
16 from rest_framework import mixins
17 from rest_framework import generics
18
19 class SnippetList(mixins.ListModelMixin,
20                  mixins.CreateModelMixin,
21                  generics.GenericAPIView):
22     queryset = Snippet.objects.all()
23     serializer_class = SnippetSerializer
24
25     def get(self, request, *args, **kwargs):
26         return self.list(request, *args, **kwargs)
27
28     def post(self, request, *args, **kwargs):
29         return self.create(request, *args, **kwargs)
30

```

继承了多个类：

mixins中实现很多具体的功能，比如列表、创建等等

第三级中是APIView，3.5级的更好用一点

参数 关键词参数

全选，也就是说操作的数据是什么  
定义使用哪个序列化器

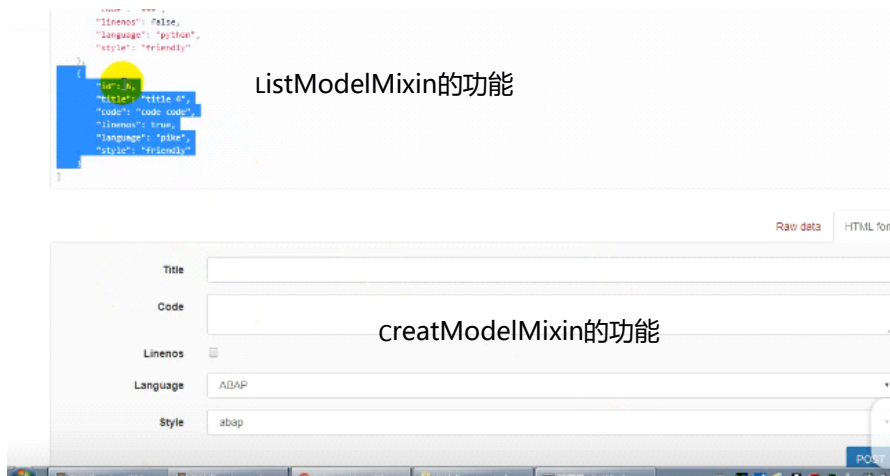
```

1
2 class SnippetDetail(mixins.RetrieveModelMixin,
3                    mixins.UpdateModelMixin,
4                    mixins.DestroyModelMixin,
5                    generics.GenericAPIView):
6     queryset = Snippet.objects.all()
7     serializer_class = SnippetSerializer
8
9     def get(self, request, *args, **kwargs):
10         return self.retrieve(request, *args, **kwargs)
11
12     def put(self, request, *args, **kwargs):
13         return self.update(request, *args, **kwargs)
14
15     def delete(self, request, *args, **kwargs):
16         return self.destroy(request, *args, **kwargs)
17

```

和第三级的区别：

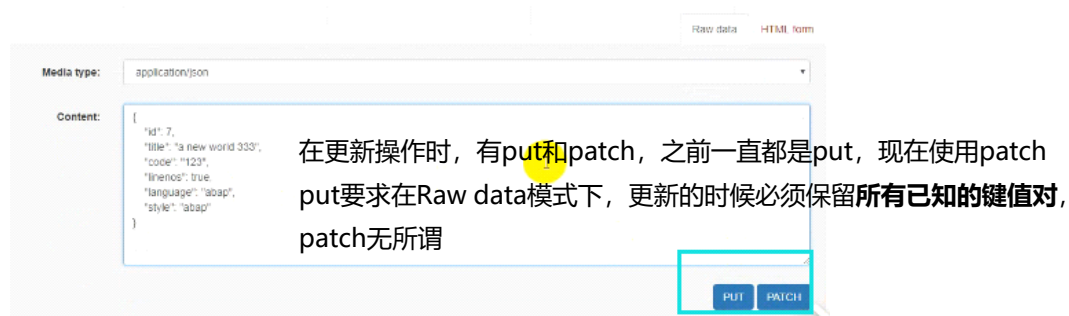
不需要输入完整的json格式进行数据的增加，而是每个条目进行填写



## 第4级别的代码，最常用

不再使用mixins，更加集成

代码已经很少了



其他的一些功能：

在model中，对模型进行了修改，功能是高亮显示相关

```

models.CASCADE, null=True, blank=True)
highlighted = models.TextField(null=True, blank=True)

class Meta:
    ordering = ('created',)

def save(self, *args, **kwargs):
    """
    高亮显示相关
    """
    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)

```

def save 是自带的函数，在保存的时候执行。因此可以在里面进行修改

自动生成一个首页，其实没啥用（第五级会自动生成，但是不推荐用第五级）

```

73
74
75 from rest_framework.decorators import api_view
76 from rest_framework.response import Response
77 from rest_framework.reverse import reverse
78
79
80 @api_view(['GET'])
81 def api_root(request, format=None):
82     return Response({
83         'users': reverse('user-list', request=request, format=format),
84         'snippets': reverse('snippet-list', request=request, format=format)
85     })
86
87

```

## 第五级

view中，均为ViewSet，然后里面定义方法

在url.py中，先用字典的方式新建一个变量，声明每个ViewSet中，get对应什么方法，post对应什么方法

在url里只用写定义的变量即可即可

真正的第五级：

Url.py中写的很少

但是不方便定制，基本不用

```
4
5 from django.conf.urls import url, include
6 from snippetsapp import views
7 from rest_framework.routers import DefaultRouter
8
9 router = DefaultRouter()
10 router.register(r'snippets', views.SnippetViewSet)
11 router.register(r'users', views.UserViewSet)
12
13
14
15 urlpatterns = [
16     url(r'^$', include(router.urls)),
17 ]
18
19
20
21
```

I



# model字段

看P12后面和P13前面建模

2020年10月31日 10:20

## 主键

`id = models.AutoField(primary_key=True)`, 你非要自己设置主键, 那么请务必将字段设置 `primary_key=True`。

Django在一个模型中只允许有一个自增字段, 并且该字段必须为主键!

```
2
3 @python_2_unicode_compatible
4 class Category(models.Model):
5     """
6     商品类别: 笔记本、平板电脑、一体机、台式机、服务器
7     """
8     name = models.CharField(max_length=200)
9     created = models.DateTimeField(auto_now_add=True)
10    updated = models.DateTimeField(auto_now=True)
11
12    def __str__(self):
13        return self.name
14
```

创建时自动保存  
修改时自动保存  
不定义的话, 会返回一个object, 不清晰

```
@python_2_unicode_compatible
class UserProfile(models.Model):
    """
    用户档案
    """
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name='profile_of',)
    mobile_phone = models.CharField(blank=True, null=True, max_length=200)
    nickname = models.CharField(blank=True, null=True, max_length=200)
    description = models.TextField(blank=True, null=True)
    icon = models.ImageField(blank=True, null=True, max_length=200, upload_to='user/uploads/%Y/%m/%d/')
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    delivery_address = models.ForeignKey(DeliveryAddress, related_name='user_delivery_address', on_delete=models.CASCADE, blank=True, null=True, )

```

强制一对一

图片



```
@python_2_unicode_compatible
```

```
class Order(models.Model):
```

```
    """
```

```
    Order
```

```
    """
```

```
    STATUS_CHOICES = (
```

```
        ('0', 'new'),
```

```
        ('1', 'not paid'),
```

```
        ('2', 'paid'),
```

```
        ('3', 'transport'),
```

```
        ('4', 'closed'),
```

```
    )
```

```
    status = models.CharField(choices=STATUS_CHOICES, default='0', max_length=2)
```

```
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE,  
                             related_name='order_of',)
```

```
    remark = models.TextField(blank=True, null=True)
```

```
    product = models.ForeignKey(Product, related_name='order_product', on_delete=models.  
                                CASCADE)
```

```
    price = models.DecimalField(max_digits=12, decimal_places=2)
```

```
    quantity = models.PositiveIntegerField(default=1)
```

```
    address = models.ForeignKey(DeliveryAddress, related_name='order_address',  
                                on_delete=models.CASCADE)
```

```
    created = models.DateTimeField(auto_now_add=True)
```

```
    updated = models.DateTimeField(auto_now=True)
```

```
    def __str__(self):
```

```
        return 'order of %d' % (self.user.id)
```

I

建模的时候，还有状态字可以选择，choice=XXX

数字是实际存储的，后面是描述

# admin.py

2020年10月31日 11:55

用于在admin后台中进行管理数据库, P13

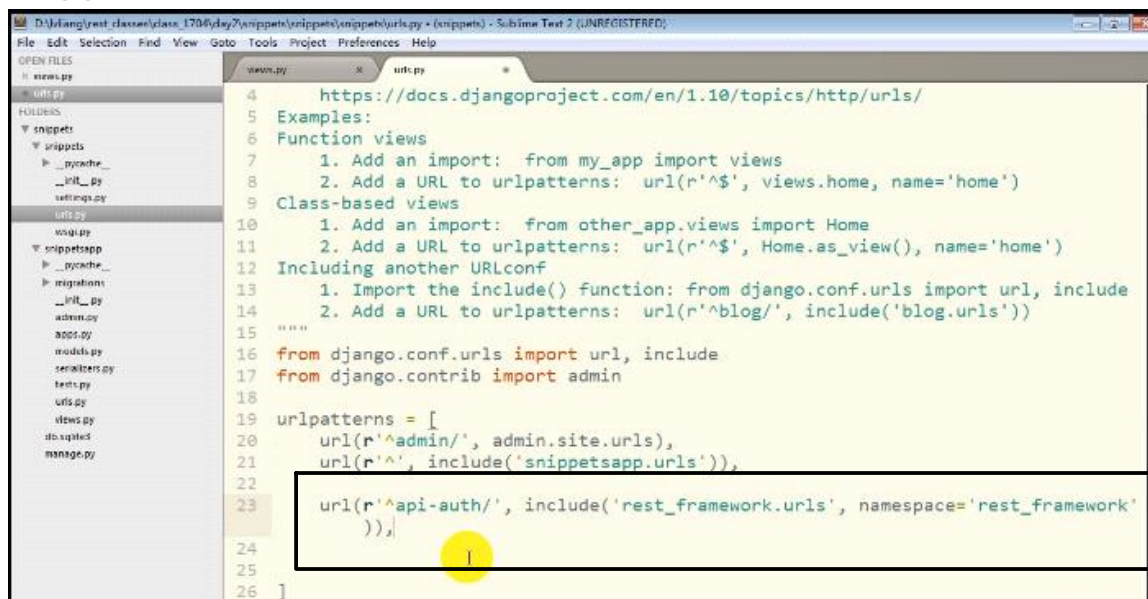
```
class UserProfileAdmin(admin.ModelAdmin):  
    list_display = ['id', 'mobile_phone', 'nickname', 'user',]  
  
admin.site.register(UserProfile, UserProfileAdmin)  
  
class CategoryAdmin(admin.ModelAdmin):  
    list_display = ['id', 'name',]  
  
admin.site.register(Category, CategoryAdmin)  
  
class ManufacturerAdmin(admin.ModelAdmin):  
    list_display = ['id', 'name',]  
  
admin.site.register(Manufacturer, ManufacturerAdmin)
```

# 权限管理

2020年10月22日 16:07

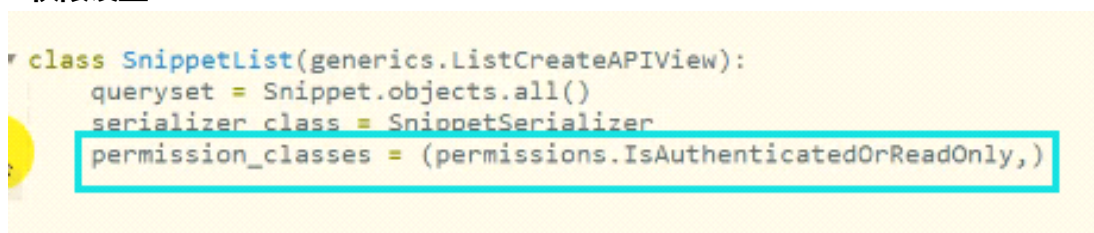
## 权限管理，登陆设置

### 1. 登陆设置：在根目录下加一行



```
4 https://docs.djangoproject.com/en/1.10/topics/http/urls/
5 Examples:
6 Function views
7 1. Add an import: from my_app import views
8 2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')
9 Class-based views
10 1. Add an import: from other_app.views import Home
11 2. Add a URL to urlpatterns: url(r'^$', Home.as_view(), name='home')
12 Including another URLconf
13 1. Import the include() function: from django.conf.urls import url, include
14 2. Add a URL to urlpatterns: url(r'^blog/', include('blog.urls'))
15 """
16 from django.conf.urls import url, include
17 from django.contrib import admin
18
19 urlpatterns = [
20     url(r'^admin/', admin.site.urls),
21     url(r'^', include('snippetsapp.urls')),
22
23     url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'
24                               )),
25
26 ]
```

### 2. 权限设置



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

# User创建

2020年10月20日 17:01

P17: 用户登陆 (从4.51开始), 后半部分是用户信息更新等等  
P18 6.40 用户注册

举例: 用户创建

view中, 单独create的view, 非常简单

```
1  
2 class UserCreateView(generics.CreateAPIView):  
3     serializer_class = UserSerializer  
4
```

序列化器比较麻烦

进行额外的设置, create进行了重载?

为什么呢, 就是

1. 额外的参数: 密码, 只能写入, 不能显示, 也就是说不能返回
2. 数据库里要对密码加密, 因此create中有设置
3. 新建用户后, 还要新建用户档案

```
26  
27 class UserSerializer(serializers.ModelSerializer):  
28  
29     class Meta:  
30         model = User  
31         fields = ('id', 'username', 'password', 'last_name', 'first_name', 'email',)  
32         extra_kwargs = {'password': {'write_only': True}}  
33  
34     def create(self, validated_data):  
35         user = User(**validated_data)  
36         user.set_password(validated_data['password']) 键值对匹配之类的, 然后加密  
37         user.save()  
38         user_profile = UserProfile(user=user)  
39         user_profile.save()  
40         return user  
41
```

序列化器中, 用create, 创建, 功能就是: 新建对象到保存之间, 要做什么操作。  
简单来说就是自己重写了在存储数据的时候要干啥

view中, 用perform\_create

这里还有个有意思的地方, 新建用户信息后, 会新建用户档案, 而使用的都是user这个对象, 这个到底是怎么实现的?

[跳转](#)

多个序列化器可以对应一个model, 使用它的不同字段。

比如:

- 一个序列化器可以展示详情
- 另一个序列化器用来显示list
- 再一个用来创建
- 当然也可以都放到一个序列化器中

## view的函数

2020年10月23日 9:26

这里要注意看view中的一些实用方法

### 1. Get\_queryset()

可以实现获取列表，之前一直是all，这次是有筛选了，可以只返回该用户的東西之类的  
这样子，就可以避免漏洞，获取到别人的数据

### Perform\_create()

创建功能，有一个参数为序列化器实例（当前对象的），可以反映当前用户，因此调用save，就可以保存了，  
这里可以控制创建之后给哪个用户，总不能A创建的，手动指定存到B那里去了

```
77 class DeliveryAddressCView(generics.ListCreateAPIView):
78     """
79     收货地址
80     """
81     serializer_class = DeliveryAddressSerializer
82     permission_classes = (permissions.IsAuthenticated,)
83
84     def get_queryset(self):
85         user = self.request.user
86
87         queryset = DeliveryAddress.objects.filter(user=user)
88
89         return queryset
```

```
91 def perform_create(self, serializer):
92     user = self.request.user
93     s = serializer.save(user=user)
94     profile = user.profile
95     profile.delivery_address = s
96     profile.save()
97
98     # 在model中，可以设置反向关系，
99     # 用relatedname作为标识
```

这几行是为了，新建地址后，保存到用户的档案中，那么首先就要找到该用户的档案，利用了反向关系。然后赋值，保存  
这里还要看一下，地址其实是个外键

```
class DeliveryAddressRUDView(generics.RetrieveUpdateDestroyAPIView):
    """
    收货地址RUD
    """
    serializer_class = DeliveryAddressSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def get_object(self):
        user = self.request.user

        try:
            obj = DeliveryAddress.objects.get(id=self.kwargs['pk'], user=user)
        except Exception as e:
            raise NotFound('not found')

        return obj
```

```
python_2_unicode_compatible
class UserProfile(models.Model):
    """
    用户档案
    """
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE,
    related_name='profile')
    mobile_phone = models.CharField(blank=True, null=True, max_length=200)
    nickname = models.CharField(blank=True, null=True, max_length=200)
    description = models.TextField(blank=True, null=True)
    icon = models.ImageField(blank=True, null=True, max_length=200, upload_to='
    user/uploads/%Y/%m/%d/')
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    delivery_address = models.ForeignKey(DeliveryAddress, related_name='
    user_delivery_address', on_delete=models.CASCADE, blank=True, null=True,)
```

这里就是刚才说的可以传入user自动创建的地方，  
就是在反向关系实现的OneToOne一对一关系！

url(r'^delivery-address-rud/(?P<pk>[0-9]+)/\$', views.DeliveryAddressRUDView.as\_view(), name='delivery\_address\_rud')，

此处pk，其实是url中获取的，应该也可以进行其他方面的操作，比如传入参数，但是这样就更不好了，直接通过修改参数就能获取到别人的地址

```
@python_2_unicode_compatible
class UserProfile(models.Model):
    """
    用户档案
    """
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE,
    related_name='profile')
    mobile_phone = models.CharField(blank=True, null=True, max_length=200)
    nickname = models.CharField(blank=True, null=True, max_length=200)
    description = models.TextField(blank=True, null=True)
    icon = models.ImageField(blank=True, null=True, max_length=200, upload_to='
    user/uploads/%Y/%m/%d/')
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    delivery_address = models.ForeignKey(DeliveryAddress, related_name='
    user_delivery_address', on_delete=models.CASCADE, blank=True, null=True,)
```

一对一关系

一般来说，列表和创建放在一起，既可以展示也可以新建，而且这个时候可以通过设置get\_query，实现只展示当前用户所含有的列表

详情修改删除可以放在一起

详情页面要注意，如果要仅显示当前用户的某些东西，那么其实是get就可以不用filter，因为反正每次只看到一个对象

```
class DeliveryAddressRUDView(generics.RetrieveUpdateDestroyAPIView):
    """
    收货地址RUD
    """
    serializer_class = DeliveryAddressSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def get_object(self):
        user = self.request.user

        obj = DeliveryAddress.objects.get(user=user)

        return obj
```

又不用上面这种方式了，是因为

- 1.上面这个其实是错的，因为过滤之后，会返回一堆地址。
  - 2.和前端配合不够好，因为前端调用的时候，利用地址，传来了pk
  - 3.底下这个方法，第一使用了user，第二使用了pk，就可以确定只有一条了
- 这个其实是，先利用列表，得到一堆该用户的，然后点击，这个时候获取pk也就是id所以：

```
class DeliveryAddressRUDView(generics.RetrieveUpdateDestroyAPIView):
    """
    收货地址RUD
    """
    serializer_class = DeliveryAddressSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def get_object(self):
        user = self.request.user

        # obj = DeliveryAddress.objects.get(user=user)

        try:
            obj = DeliveryAddress.objects.get(id=self.kwargs['pk'], user=user)
        except Exception as e:
            raise NotFound('not found')

        return obj
```

这两个view的序列化器可以是一样的，也可以针对详情的这个view建一个更详细的

# token验证

2020年10月20日 16:26

Settings.py中，添加autotoken，rest自带的

```
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40     'rest_framework',
41     'rest_framework.authtoken',
42     'corsheaders',
43     'computerapp.apps.ComputerappConfig',
44 ]
45
```

```
38
39
40 REST_FRAMEWORK = {
41     'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
42     'PAGE_SIZE': 6,
43     'DEFAULT_AUTHENTICATION_CLASSES': (
44         'rest_framework.authentication.BasicAuthentication',
45         'rest_framework.authentication.SessionAuthentication',
46         'rest_framework.authentication.TokenAuthentication',
47     ),
48 }
49
50
51
```

根目录下，添加对应的路由

```
23
24
25 urlpatterns = [
26     url(r'^admin/', admin.site.urls),
27     url(r'^computer/', include('computerapp.urls')),
28     url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework')),
29     url(r'^api-token-auth/', views.obtain_auth_token),
30 ]
31
32
33
34 ]
35
36
```

以防万一的东西

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'rest_framework.authtoken',
    'django_filters',
    'corsheaders',
    'computerapp.apps.ComputerappConfig',
]

MIDDLEWARE = [

```

前端要存起来，今后进行需要权限的操作，都要发送。这就不是我的问题了

```
$.ajax({
  url: "http://127.0.0.1:8001/api-token-auth/",
  type: "POST",

```



```
$.ajax({
  url: "http://127.0.0.1:8001/api-token-auth/",
  type: "POST",
  data: data,
  success: function(res){
    console.log(res.token);
    localStorage.setItem("token", res.token);
    // window.location.href = "usercenter.html";
  },
  error: function (jqXHR, textStatus, errorThrown)
  {
    console.log(jqXHR, textStatus, errorThrown);
    alert("用户名或密码错误! ");
  }
});
```

# 图片上传

2020年10月18日 13:50

如果需要**上传图片**，可以自定义初始打开的文件夹是哪一层

```
MEDIA_URL = '/media/'  
MEDIA_ROOT = os.path.join(os.path.dirname(BASE_DIR), 'eshop/media')
```

# 测试时不显图

2020年10月22日 16:20

根目录url

```
▼ if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

根目录setting

```
1
2 MEDIA_URL = '/media/'
3 MEDIA_ROOT = os.path.join(os.path.dirname(BASE_DIR), 'eshop/media')
4
5
```

## 分类/搜索/排序

2020年10月20日 16:58

```
class ProductListView(generics.ListAPIView):
    """
    产品列表
    """
    queryset = Product.objects.all()
    serializer_class = ProductListSerializer
    permission_classes = (permissions.AllowAny,)
    filter_backends = (OrderingFilter, SearchFilter)
    ordering_fields = ('category', 'manufacturer', 'created', 'sold',)
    search_fields = ('description', 'model',)
    ordering = ('id',)
    pagination_class = LimitOffsetPagination
```

权限管理

筛选机制: **搜索和排序**

搜索范围: 限定了description

排序范围: 限定了那么多

默认排序: id

要是无限定排序范围, 则可以所有的字段进行排序展示, 这样子反而不好  
限定之后, 只能对限定的字段排序

分类

```
class ProductListByCategoryView(generics.ListAPIView):
    """
    产品按类别列表
    """
    serializer_class = ProductListSerializer
    permission_classes = (permissions.AllowAny,)
    filter_backends = (SearchFilter, OrderingFilter,)
    search_fields = ('description',)
    ordering_fields = ('category', 'manufacturer', 'created', 'sold', 'stock', 'price',)
    ordering = ('id',)

    def get_queryset(self):
        category = self.request.query_params.get('category', None)

        if category is not None:
            queryset = Product.objects.filter(category=category)
        else:
            queryset = Product.objects.all()

        return queryset
```

那么问题来了, 用的什么方法?

噢, 应该是get获取, 然后传入参数

和上面不一样的地方:

1. 第一行, 因为要分类, 因此必不可能用objects.all()函数

2. 利用get\_query()方法, 实现获取query

获取到category参数, 保存, 如果没有此参数, 就设为None

然后判断是否为空, 进行objects的filter罢了

很简单

这是两种方法, 毕竟上面只能  
排序和搜索, 你这个要分类,  
不这么弄怎么办?

可以实现:

分类

搜索

排序

前端只要传入对应的参数即可

比如, 设置按sold排序, 参数ordering=sold即可, 倒序加“-”

127.0.0.1:8001/computer/product\_list/?ordering=sold

搜索, 参数search=xxx即可

127.0.0.1:8001/computer/product\_list/?search=尔

上面已经实现了分类, 现在更加细化, 分类里面再分类, 比如: 笔记本里的联想

```
def get_queryset(self):
    category = self.request.query_params.get('category', None)
    manufacturer = self.request.query_params.get('manufacturer', None)

    if category is not None:
        queryset = Product.objects.filter(category=category, manufacturer=manufacturer)
    else:
        queryset = Product.objects.all()

    return queryset
```

这里, 要是类别都没有, 那选择生产商没有意义, 因此直接all了  
你自己也可以设置其他的逻辑

# APIView/更新

2020年10月22日 15:54

比如：用户信息

```
1 class UserDetailView(APIView):
2     """
3     用户基本信息
4     """
5     permission_classes = (permissions.IsAuthenticated,)
6
7     def get(self, request, format=None):
8         user = self.request.user
9         serializer = UserInfoSerializer(user)
10        return Response(serializer.data)
```

这里其实仅仅设置了用户信息的获取  
要求每个用户只能看到自己的信息

第三级代码有个好处，就是可以自己设置需要什么方法，自己定义get和post之类的

Self.request.user是什么啊？自己添加的参数吗，还是自动会获取到

第四级的代码

直接在网址上带上参数，就可以看到其他账户的信息，不合适

第三级代码中

get方法里强制进行了用户限定

添加信息时，信息在序列器中没有，只在model中有

这个好像之前讲过？

更新用户信息，使用get\_object函数，直接从request.user中获得用户，也是为了防止修改掉其他用户

```
1 class UserProfileRUIView(generics.RetrieveUpdateAPIView):
2     """
3     用户其他信息
4     """
5     serializer_class = UserProfileSerializer
6     permission_classes = (permissions.IsAuthenticated,)
7
8     def get_object(self):
9         user = self.request.user
10
11        obj = UserProfile.objects.get(user=user)
12
13        return obj
```

get\_object是一个自带函数，现在把它改  
了一下，在获取对象的时候会调用

这个更先进一点，详情和更新都有了，首先能拿到对应的用户信息，然后写入到对应的用户，不会出现写错。

# 可显示, Read\_only

2020年10月22日 16:09

想要保存是谁修改了内容:

1.model中加入相关字段

```
style = models.CharField(max_length=512, default='', blank=True, max_length=512)
owner = models.ForeignKey('auth.User', related_name='snippets', on_delete=models.CASCADE)
highlighted = models.TextField(null=True, blank=True)
```

2.序列化器fields中引入字段

3.其实已经实现了, 但是, 此时用户是可选的, 很不好, 解决:

```
8
9 class SnippetList(generics.ListCreateAPIView):
10     queryset = Snippet.objects.all()
11     serializer_class = SnippetSerializer
12     # permission_classes = (permissions.IsAuthenticated,)
13
14     def perform_create(self, serializer):
15         serializer.save(owner=self.request.user)
16
```

创建的时候执行? 啥都没讲

加入一个函数, request中自动包含了很多数据, 其中就有一个是user  
但是这样子操作, 其实还是能选择, 只不过是选了没用罢了

改为: 不能选择

也就是说, 可以显示, 但是post不可修改

注意: 这里也要补上owner

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

**Read\_only就可以实现了**



# 分页

2020年10月22日 16:21

## 全局

在settings.py里加一行代码即可

```
L23
L24
L25 REST_FRAMEWORK = {
L26     'PAGE_SIZE': 3,
L27 }
L28
L29
L30
```

某一部分另外的分页方法

```
class ProductListView(generics.ListAPIView):
    """
    产品列表
    """
    queryset = Product.objects.all()
    serializer_class = ProductListSerializer
    permission_classes = (permissions.AllowAny,)
    filter_backends = (OrderingFilter, SearchFilter)
    ordering_fields = ('category', 'manufacturer', 'created', 'sold',)
    search_fields = ('description', 'model',)
    ordering = ('id',)
    pagination_class = LimitOffsetPagination
```

这样子，就对该view返回的数据进行了设置

参数:

Offset, 第一页默认没有, **offset=x**, 这一页就从全部数据的第x+1个开始  
反正一般也不会用到offset这个参数, 只有能够选择跳转到某一页的时候才用

① 127.0.0.1:8001/computer/product\_list/?offset=6

limit, 设置每一页的数量

① 127.0.0.1:8001/computer/product\_list/?limit=9

几种方法:

1.默认设置每页的页数, 不用管了。因此每次翻页的时候进行计算, offset=XXX即可获得下一页或者上一页

2.数据返回的时候, 就会给到下一页、上一页的链接, 比如

```
{
    "count": 4,
    "next": "http://127.0.0.1:8000/userList/?limit=3&offset=3",
    "previous": null,
    "results": [
        {
            "id": 1,
```

直接调用也行

3.自己设置, 利用limit和offset来计算, 每次都传回去两个参数, 这个既可以自定义每一页返回多少数据, 也可以自定义从哪一条数据开始

## 列表/详情区分

2020年10月22日 15:21

### ProductList

列表的view

区分列表和详情，因为需要展示的信息不一样的

List规定了通过get来获取

```
class ProductListView(generics.ListAPIView):  
    """  
    产品列表  
    """  
    queryset = Product.objects.all()  
    serializer_class = ProductListSerializer  
    permission_classes = (permissions.AllowAny,)
```

产品列表的序列化器

```
class ProductListSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Product  
        fields = ('id', 'model', 'image', 'price', 'sold', 'category', 'manufacturer')
```

产品列表的url

```
from django.conf.urls import url  
from rest_framework.urlpatterns import format_suffix_patterns  
from computerapp import views  
  
urlpatterns = [  
    url(r'^product_list/$', views.ProductListView.as_view(), name='product_list'),  
]  
  
urlpatterns = format_suffix_patterns(urlpatterns)
```

### ProductDetail

详情的view

序列化器不一样，继承的不一样。

retrieve用于只读，以表示单个实例

```
class ProductRetrieveView(generics.RetrieveAPIView):  
    queryset = Product.objects.all()  
    serializer_class = ProductRetrieveSerializer  
    permission_classes = (permissions.AllowAny,)
```

从序列化器看，更详细

```
class ProductRetrieveSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Product  
        fields = ('id', 'model', 'image', 'price', 'sold', 'category', 'manufacturer', 'description')
```

但是view中，使用的APIView不一样了。改成了RetrieveAPIView

用于只读端点，以表示单个模型实例

url也不一样了，这里才是重点

```
url(r'^product_retrieve/(?P<pk>[0-9]+)/$', views.ProductRetrieveView.as_view(), name='product_retrieve'),
```

# 详情传入参数

2020年10月23日 9:49

```
urlpatterns = [
    url(r'^product_list/$', views.ProductListView.as_view(), name='product_list'),
    url(r'^product_list_by_category/$', views.ProductListByCategoryView.as_view(), name=
    url(r'^product_list_by_category_manufacturer/$', views.ProductListByCategoryManufact
    url(r'^product_retrieve/(?P<pk>[0-9]+)/$', views.ProductRetrieveView.as_view(), name=
    url(r'^user_info/$', views.UserInfoView.as_view(), name='user_info'),
    url(r'^user_profile_ru/(?P<pk>[0-9]+)/$', views.UserProfileRUIView.as_view(), name='u
]
```

能理解用retrieve来实现详情，URL中需要带参数，但是前端，需要传输什么信息，才能实现呢？

```
$(document).on("pageshow", "#useraddress_details", function(){
    var url = window.location.href;
    var id = url.split("?")[1].split("=")[1];

    $.ajax({
        url: "http://127.0.0.1:8001/computer/delivery_address_rud/" + id + "/",
        headers: myheader(),
        success: function(res){
            console.log(res);

            $("#contact_person").val(res.contact_person);
            $("#contact_mobile_phone").val(res.contact_mobile_phone);
            $("#delivery_address").val(res.delivery_address);

        },
        error: function (jqXHR, textStatus, errorThrown)
        {
            console.log(jqXHR, textStatus, errorThrown);
        }
    });
});
```

购物车的列表、创建

2020年10月23日 13:17

P20最后

列表

```
class CartListView(generics.ListAPIView):
    """
    Cart List
    """
    serializer_class = OrderListSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def get_queryset(self):
        user = self.request.user
        queryset = Order.objects.filter(user=user, status='0')
        return queryset
```

没什么说的，也就是listAPI  
获取列表的时候注意筛选，本用户且满足状态为0，即购物车

这是其他订单的，不是购物车的，仅仅是状态码有区别

```
queryset = Order.objects.filter(user=user, status__in=['1', '2', '3', '4'])
```

创建

```
class OrderCreateView(generics.CreateAPIView):
    """
    Order Create
    """
    queryset = Order.objects.all()
    serializer_class = OrderCreateSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def perform_create(self, serializer):
        user = self.request.user
        product = serializer.validated_data.get('product')

        serializer.save(user = user, price = product.price, address = self.request.user.
            profile_of.delivery_address)
        # logging.debug( this is debug )
        # logging.info( 'this is critical %s', 'added' )
        logging.info( 'user %d cart changed, product %d related. Time is %s.', user.id, product.
            id, str(datetime.datetime.now()) )
```

这里只有创建功能，所以queryset其实没啥用，直接all也无所谓

这一句是从已经验证过的数据里，取到product的id（前端传过来的，仅仅是id罢了，前端跳转的时候会传这个参数，这就是我要的!!!）然后序列化器会自动转为对应的对象

此处没有提到的，都会直接用前端传来的东西存

这里用到外键的关联作用，直接找到了用户档案

每个带有创建功能的View，基本都要注意perform\_create这个函数，即创建时执行的操作

之所以要重写，基本都是为了避免漏洞（外人可以直接操控对别的账户提交创建

前端只需要传个user，product的id，连价格都不需要，避免漏洞。  
我自己查询之后保存，不让前端给我传送

按理说前端需要传过来一个product的id。但是我看url里没说啥啊

那这个意思是不是说，我只是从前端拿到相应的数据，创建的功能我要重写，自己保存，自己处理？

```
url(r'^order_create/$', views.OrderCreateView.as_view(), name='order_create'),
```

## view中的RUD

2020年10月23日 13:59

更新的时候，也要注意重写。这是为了**防止漏洞，避免用户直接能把购物车里的东西加到订单里**  
这个view的作用，是删除，修改购物车订单，那么只有变为**删除/下单**两种可能，因此**更新操作中，强制status=1是合理的**

```
class OrderRUDView(generics.RetrieveUpdateDestroyAPIView):
    """
    Order rud
    """
    serializer_class = OrderRUDSerializer
    permission_classes = (permissions.IsAuthenticated,)

    def get_object(self):
        user = self.request.user
        obj = Order.objects.get(user=user, id=self.kwargs['pk'])
        return obj

    def perform_update(self, serializer):
        user = self.request.user
        serializer.save(user = user, status = '1')
```

收货地址，在P19  
订单列表，在P20 5.32

这么说的话，既然更新要重写，为了**防止漏洞，删除是不是也要重写？**

**不需要**

**因为本质是详情，进入详情后才能删除或者更新，而进入详情时，已经对用户进行了验证，因此用户只能对自己的东西进行操作**

# 序列化器嵌套

2020年11月2日 16:03

```
class UserInfoSerializer(serializers.ModelSerializer):
    profile_of = UserProfileSerializer()

    class Meta:
        model = User
        fields = ('id', 'username', 'email', 'first_name', 'last_name', 'date_joined', 'profile_of',)
```

这里可以嵌套的，应该是外键的缘故吧？

1. Model中，Profile有一个一对一段user，设置了related\_name

```
@python_2_unicode_compatible
class UserProfile(models.Model):
    """
    用户档案
    """
    user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE,
                               related_name='profile',)
    mobile_phone = models.CharField(blank=True, null=True, max_length=200)
    nickname = models.CharField(blank=True, null=True, max_length=200)
    description = models.TextField(blank=True, null=True)
    icon = models.ImageField(blank=True, null=True, max_length=200, upload_to='user/uploads/%Y/%m/%d/')
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    delivery_address = models.ForeignKey(DeliveryAddress, related_name='user_delivery_address', on_delete=models.CASCADE, blank=True, null=True,)
```

这个很有问题。可能是框架升级了？反正不能用了。回去还得再看、

主要就是要实现：序列化器中显示外键的相关字段

2. 序列化器中，有一个序列化器A，model对应的是userprofile，B对应的user。

在B中，想新加一个字段，就加的是上面的user的东西

```
class UserProfileSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserProfile
        fields = ('id', 'user', 'mobile_phone', 'nickname', 'description', 'icon', 'created', 'updated',)
        read_only_fields = ('user',)

class UserInfoSerializer(serializers.ModelSerializer):
    profile_of = UserProfileSerializer()

    class Meta:
        model = User
        fields = ('id', 'username', 'email', 'first_name', 'last_name', 'date_joined', 'profile_of',)
```