目录

[各部门介绍 2](#_Toc34254802)

[DjangoRestFrameWork必知概念 3](#_Toc34254803)

[DRF框架初步使用 5](#_Toc34254804)

[API文档的使用 8](#_Toc34254805)

[PostMan API 测试工具的使用 9](#_Toc34254806)

[DjangoRestFrameWork自定义序列化类 10](#_Toc34254807)

[DJangoRestFrameWork之中的请求、响应 12](#_Toc34254808)

[DJangoRestFrameWork基于函数的视图 13](#_Toc34254809)

[DjangoRestFrameWork基于类的视图 15](#_Toc34254810)

[DjangoRestFrameWork基于混合类的视图 17](#_Toc34254811)

[DjangoRestFrameWork基于视图集的视图 19](#_Toc34254812)

[DjangoRestFrameWork 用户模型类 20](#_Toc34254813)

[DjangoRestFrameWork 用户注册路由 21](#_Toc34254814)

[DjangoRestFrameWork JWT认证 23](#_Toc34254815)

[DjangoRestFrameWork 自定义认证类 24](#_Toc34254816)

[DjangoRestFrameWork 权限 25](#_Toc34254817)

[DjangoRestFrameWork 频次组件 27](#_Toc34254818)

[DjangoRestFrameWork 分页组件 29](#_Toc34254819)

[DjangoRestFrameWork 过滤组件 31](#_Toc34254820)

[DjangoRestFrameWork 跨域 32](#_Toc34254821)

# 各部门介绍

公司项目部门工作内容：

产品：

1. 研究市场相关(竞品)产品，决定项目方向，项目内容，初步效果

成果: 提供产品原型

出身：工作经验比较丰富，可能来源于技术

参考：https://jobs.zhaopin.com/CC428974935J00395506506.htm

设计：

1. 网页设计/平面设计/UI设计 PS、 AI、 CDR、 AE

成果：页面上的布局样式、效果图、素材

可能懂前端开发

参考：https://jobs.zhaopin.com/CC205116018J00201474514.htm

前端：

1. 传统的HTML+CSS+JS+前端框架(jquery+bootstrap)
2. 工程化的前端开发(Vue工程、Vue/cli)

成果：将设计提供的素材制作成静态的html页面元素，并且添加交互功能

可能懂设计，可能懂后端

参考：https://jobs.zhaopin.com/CC196911813J00196464615.htm

后端：

1. 将前端开发出来的HTML+CSS+JS实现的静态页面拿到自己的项目中

并且对项目文件进行修改，以实现数据的动态表现(结合数据库)

在Django中需要将MVT中的T(html)进行修改(由谁来改？)

工作量非常大，项目推进困难

1. 为前端开发人员编写的Vue工程提供项目数据接口(网址地址)

前端人员请求该地址可以获取数据，并且自己将数据展示在Vue项目中

在Django中完全不需要MVT设计中的T

有个更好的Django+DRF框架

主流方向：开发周期短，协同效率

前后端不分离需要懂前端，前后端分离可以不懂前端

参考：https://jobs.zhaopin.com/CC332511830J00193862909.htm

每个岗位工作会有交叉，具体分工要依据公司规模、项目规模决定

# DjangoRestFrameWork必知概念

1. 一个建立在Django基础上的框架，前期知识，熟练使用django
2. Rest：一种面向资源的架构，互联网一切皆资源，一切操作皆是全部面向资源的操作

请求书籍列表 请求书籍详情 添加书籍 删除书籍 更新书籍 书籍模型就是资源

1. Restful：称为Restful风格，一种建立在Rest架构设计上的应用
2. RestfulAPI：面向资源应用的API接口(面向资源的形式设计API)

RestFulAPI的构成：https://api.zzy.com/v1/books

协议：https

域名：一个用于部署服务器的电脑(专用) 通过域名和IP地址绑定 最好包含 api. zzy.com

版本：携带API版本 /v1

路径：虚拟地址 /home 推荐使用名词 一般为模型名复数

1. 动词可以代表针对于资源的操作类型

GET：代表获取资源列表 获取图书列表

POST：代表添加新的资源 添加一本书

PUT：代表修改资源 PUT需要提供书籍的所有字段

PATCH：代表修改资源 PATCH 只需要提供修改的字段

DELETE： 代表删除资源 删除一本书

OPTIONS：关于这个资源可以有哪些操作

HEAD：可以获取元数据

使用名词+动词混合设计路由混乱

https:// zzy.com/getbooks/ 获取书籍列表

https:// zzy.com/addbook/ 添加书籍

https:// zzy.com/getbook/bookid/ 获取书籍详情

https:// zzy.com/updatebook/bookid/ 更新书籍信息

https:// zzy.com/deletebook/bookid/ 删除书籍

使用RestFulAPI设计API

<https://api.zzy.com/v1/books/>

GET获取列表、POST添加书籍

<https://api.zzy.com/v1/books/booid/>

GET 获取书详情、PUT、PATCH修改书信息、DELETE删除书信息

1. 状态码

每一次请求的结果中都包含是否成功的标识

成功：

GET 200

POST、PATCH、PUT 201

DELETE 204

失败：

401 代表没有认证

403 已经认证但是权限不够

404 请求地址不对

错误：

500 服务器错误

# DRF框架初步使用

在DRF中使用ModelSerializer + ModelViewSet + DefaultRouter可以快速实现RestFulAPI

1. 模型设计

分类与商品之间为一对多关系

from django.db import models

# Create your models here.

class Category(models.Model):

name = models.CharField(max\_length=20,verbose\_name="分类名")

def \_\_str\_\_(self):

return self.name

class Good(models.Model):

name = models.CharField(max\_length=20,verbose\_name="商品名字")

desc = models.CharField(max\_length=100,null=True,blank=True,verbose\_name="商品描述")

# 在序列化关联模型时一定要声明related\_name

# 一找多 related\_name 没有定义 c1.good\_set.all() related\_name定义了 c1.goods.all()

category = models.ForeignKey(Category,on\_delete=models.CASCADE,verbose\_name="分类",related\_name='goods')

def \_\_str\_\_(self):

return self.name

1. 序列化类设计

from rest\_framework import serializers  
from .models import \*  
  
class CustField(serializers.RelatedField):  
 """  
 自定义序列化类 重写展示方法  
 """  
 def to\_representation(self, value):  
 print(value,type(value))  
 return str(value.id)+"--"+value.name  
  
class GoodSerializer(serializers.ModelSerializer):  
 # 在序列化时指定字段 在多方 使用source = 模型名.字段名 read\_only= True标识不能更改  
 category = serializers.CharField(source='category.name',read\_only=True)  
 class Meta:  
 model = Good  
 fields = ('name','desc','category')  
  
  
class CategorySerizlizer(serializers.ModelSerializer):  
 # goods 一定要和 related\_name 的值一致  
  
 # StringRelatedField() 可以显示关联模型中的 \_\_str\_\_返回值 many=True 代表多个对象 read\_only=True 代表只读  
 # goods = serializers.StringRelatedField(many=True)  
  
 # PrimaryKeyRelatedField 显示主键值  
 # goods = serializers.PrimaryKeyRelatedField(many=True,read\_only=True)  
  
 # SlugRelatedField 显示自定义字段值  
 # goods = serializers.SlugRelatedField(slug\_field='name',many=True,read\_only=True)  
  
 # 显示资源RestFulAPI  
 # goods = serializers.HyperlinkedRelatedField(read\_only=True,many=True,view\_name='good-detail')  
  
 # 显示模型所有字段  
 # goods = GoodSerializer(many=True,read\_only=True)  
  
 # 使用自定义序列化类  
 goods = CustField(many=True,read\_only=True)  
  
 class Meta:  
 model = Category  
 # \_\_all\_\_ 代表模型中的所有字段  
 # fields = "\_\_all\_\_"  
  
 # fields 指明序列化哪些字段  
 fields = ('name','goods')

1. 视图集合设计

from rest\_framework import viewsets  
from .models import \*  
from .serializers import \*  
  
class CategoryViewSets(viewsets.ModelViewSet):  
 """  
 分类视图  
 继承ModelViewSet 之后拥有GET POST PUT PATCH DELETE等HTTP动词操作  
 queryset 指明 需要操作的模型列表  
 serializer\_class 指明序列化类  
 """  
 queryset = Category.objects.all()  
 serializer\_class = CategorySerizlizer  
  
class GoodViewSets(viewsets.ModelViewSet):  
 queryset = Good.objects.all()  
 serializer\_class = GoodSerializer

1. 资源路由设计

from django.contrib import admin  
from django.urls import path,include  
from shop.views import \*  
# 引入DRF自带的路由类  
from rest\_framework import routers  
router = routers.DefaultRouter()  
  
# 可以通过router默认路由注册资源  
router.register('categorys',CategoryViewSets)  
router.register('goods',GoodViewSets)  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 # 配置RestFulAPI  
 path('api/v1/',include(router.urls)),  
 # 为了在DRF路由调试界面能够使用用户相关功能需要引入以下路由  
 path('',include('rest\_framework.urls'))  
]

# API文档的使用

在开发中需要前端后端开发人员实现约定数据接口的地址、类型、参数、返回值等

口头协定肯定不行

需要有API文档

DRF 自带了一套自动生成API文档的工具(有点专业)

使用流程：

1. 安装coreapi

pip install coreapi

1. 添加API文档路由

# 引入API文档路由

from rest\_framework.documentation import include\_docs\_urls

# 在项目路由下方添加API地址

path('api/v1/docs/',include\_docs\_urls(title="RestFulAPI",description="RestFulAPI v1")),

1. 此时需要指定主题

会出现错误：'AutoSchema' object has no attribute 'get\_link'

可以直接修改Django配置文件添加一下内容

# 此处可以对DjangoRestFrameWork重新配置

REST\_FRAMEWORK = {

# Schema

'DEFAULT\_SCHEMA\_CLASS': 'rest\_framework.schemas.AutoSchema',

}

# PostMan API 测试工具的使用

创建集合

创建文件夹

创建请求

保存请求

导出文档

# DjangoRestFrameWork自定义序列化类

在DjangoRestFrameWork快速搭建RestFulAPI一文中咱们使用了 ModelSerializer 模型序列化类快速实现了商品与分类模型的序列化，此文咱们介绍下自定义模型序列化类的使用

通过继承Serializer 类之后可以重写create与update方法以实现创建于更改模型

重写 validate\_属性名 可以用于校验属性输入是否合法

重写 validate可以对输入的属性进行加工处理

1. 序列化分类模型

class CategorySerizlizer(serializers.Serializer):

"""

序列化类 决定了模型序列化细节

"""

name = serializers.CharField(max\_length=10,min\_length=3,error\_messages={

"max\_length":"最多10个字",

"min\_length":"最少3个字"

})

def create(self, validated\_data):

"""

通过重写create方法 来定义模型创建方式

:param validated\_data:

:return:

"""

instance = Category.objects.create(\*\*validated\_data)

return instance

def update(self, instance, validated\_data):

"""

通过重写update，来定义模型的更新方法

:param instance: 更改之前的实例

:param validated\_data: 更改参数

:return: 返回的新实例

"""

instance.name = validated\_data.get("name",instance.name)

instance.save()

return instance

1. 序列化商品模型

class GoodSerializer(serializers.Serializer):

name = serializers.CharField(max\_length=20,min\_length=2,error\_messages={

"max\_length":"最多20个字",

"min\_length":"最少2个字"

})

category = CategorySerizlizer(label="分类")

def validate\_category(self, category):

"""

处理category

:param category: 处理的原始值

:return: 返回新值

"""

try:

Category.objects.get(name = category["name"])

except:

raise serializers.ValidationError("输入的分类名不存在")

return category

def validate(self, attrs):

try:

c = Category.objects.get(name=attrs["category"]["name"])

except:

c = Category.objects.create(name = attrs["category"]["name"])

attrs["category"] = c

return attrs

def create(self, validated\_data):

instance = Good.objects.create(\*\*validated\_data)

return instance

def update(self, instance, validated\_data):

instance.name = validated\_data.get("name",instance.name)

instance.category = validated\_data.get("category",instance.category)

instance.save()

return instance

# DJangoRestFrameWork之中的请求、响应

Django框架

请求类型为： WSGIRequest

提取请求中的参数使用： request.GET request.POST request.FILES

响应类型：HttpResponse JsonResponse HttpResponseRedirect

DRF框架

请求类型为：rest\_framework.request.Request 其实是针对于WSGIRequest的封装

原来的获取参数仍然可以使用，并且提供了更加优化统一的获取参数方式

提取请求中的参数使用：GET 方法使用request.query\_params

POST PUT PATCH 都是用request.data

响应类型：Response

DRF提供装饰器api\_view可以对基于函数的视图进行装饰

实现将WSGIRequest转换为基于rest\_framework.request.Request的实现

代码参考

from django.http import HttpResponse

# 通过api\_view装饰器可以将基于函数的视图转换成APIView基于类的视图

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

@api\_view(['GET','POST'])

def categoryList(request):

print(request, type(request) )

if request.method == "GET":

print("获取到GET请求参数",request.query\_params)

return HttpResponse("获取列表成功")

elif request.method == "POST":

print("获取到POST请求参数", request.data)

return HttpResponse("创建成果")

@api\_view(['GET','PUT','PATCH','DELETE'])

def categoryDetail(request,cid):

if request.method == "GET":

print("获取到GET请求参数", request.query\_params)

return HttpResponse("获取单个成功")

elif request.method == "PUT" or request.method == "PATCH":

print("获取到PUT/PATCH请求参数", request.data)

return HttpResponse("修改成功")

elif request.method == "DELETE":

return HttpResponse("删除成功")

# DJangoRestFrameWork基于函数的视图

DRF提供装饰器api\_view可以对基于函数的视图进行装饰

实现将视图函数转换为基于APIView的实现

@api\_view(['GET','POST']) 参数列表中应该列举该视图函数允许的http方法

# 通过api\_view装饰器可以将基于函数的视图转换成APIView基于类的视图

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from rest\_framework import status

from django.shortcuts import get\_object\_or\_404

@api\_view(['GET','POST'])

def categoryList(request):

if request.method == "GET":

# instance 为需要序列化的对象 来源于数据库

seria = CategorySerizlizer(instance=Category.objects.all(),many=True)

return Response(seria.data,status=status.HTTP\_200\_OK)

elif request.method == "POST":

# data 为序列化对象 来源于请求中提取的数据

seria = CategorySerizlizer(data=request.data)

# 从请求中提取的数据序列化之前需要进行校验

if seria.is\_valid():

seria.save()

return Response(seria.data,status=status.HTTP\_201\_CREATED)

else:

return Response(seria.errors,status= status.HTTP\_400\_BAD\_REQUEST)

@api\_view(['GET','PUT','PATCH','DELETE'])

def categoryDetail(request,cid):

model = get\_object\_or\_404(Category,pk=cid)

if request.method == "GET":

seria = CategorySerizlizer(instance=model)

return Response(seria.data,status=status.HTTP\_200\_OK)

elif request.method == "PUT" or request.method == "PATCH":

# 更新就是从请求中提取参数 替换掉数据库中取出的数据

seria = CategorySerizlizer(instance=model,data=request.data)

# 验证是否合法

if seria.is\_valid():

seria.save()

return Response(seria.data,status=status.HTTP\_200\_OK)

else:

return Response(seria.errors,status=status.HTTP\_400\_BAD\_REQUEST)

elif request.method == "DELETE":

model.delete()

return Response(status=status.HTTP\_204\_NO\_CONTENT)

# DjangoRestFrameWork基于类的视图

DRF提供APIView来完成对Django中View的封装扩展

在类中仍然需要实现对应的http方法

class CategoryListView(APIView):

"""

1继承Django自带的View类需要重写对应的http方法

2继承DRF自带的APIView类即可完成请求响应的封装 APIView继承封装了Django的View

"""

def get(self,request):

# instance从数据库取

seria = CategorySerizlizer(instance=Category.objects.all(),many=True)

return Response(seria.data,status=status.HTTP\_200\_OK)

def post(self,request):

# data从请求中取

seria = CategorySerizlizer(data=request.data)

# if seria.is\_valid():

# seria.save()

# return Response(seria.data,status=status.HTTP\_201\_CREATED)

# else:

# return Response(seria.errors,status=status.HTTP\_400\_BAD\_REQUEST)

seria.is\_valid(raise\_exception=True)

seria.save()

return Response(seria.data,status=status.HTTP\_201\_CREATED)

class CategoryDetailView(APIView):

def get(self,request,cid):

seria = CategorySerizlizer(instance=get\_object\_or\_404(Category,pk=cid))

return Response(seria.data,status=status.HTTP\_200\_OK)

def put(self,request,cid):

seria = CategorySerizlizer(instance=get\_object\_or\_404(Category,pk=cid),data=request.data)

if seria.is\_valid():

seria.save()

return Response(seria.data,status=status.HTTP\_200\_OK)

else:

return Response(seria.errors,status=status.HTTP\_200\_OK)

def patch(self,request,cid):

seria = CategorySerizlizer(instance=get\_object\_or\_404(Category, pk=cid), data=request.data)

if seria.is\_valid():

seria.save()

return Response(seria.data, status=status.HTTP\_200\_OK)

else:

return Response(seria.errors, status=status.HTTP\_200\_OK)

def delete(self,request,cid):

get\_object\_or\_404(Category,pk=cid).delete()

return Response(status=status.HTTP\_204\_NO\_CONTENT)

# DjangoRestFrameWork基于混合类的视图

DRF实现了基于GenericAPIView 与 mixins混合类共同实现视图类的功能

GenericAPIView中负责 定义 get\_queryset、get\_object、serializer\_class等属性的实现，为类视图提供数据与序列化属性

mixins中实现了CreateModelMixin、ListModelMixin、RetrieveModelMixin、UpdateModelMixin、DestroyModelMixin混合类，混合类的作用分别是实现http请求中post、get、patch、put、delete中具体实现、供视图类中直接调用

1. 使用GenericAPIView与mixins混合类配合定义视图

class CategoryListView(generics.GenericAPIView,mixins.ListModelMixin,mixins.CreateModelMixin):

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

def get(self,request):

return self.list(request)

def post(self,request):

return self.create(request)

class CategoryDetailView2(generics.GenericAPIView,

mixins.RetrieveModelMixin,

mixins.UpdateModelMixin,

mixins.DestroyModelMixin):

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

def get(self,request,pk):

return self.retrieve(request,pk)

def put(self,request,pk):

return self.update(request,pk)

def patch(self,request,pk):

return self.update(request,pk)

def delete(self,request,pk):

return self.delete(request,pk)

1. 使用generics中封装好的混合视图类

为了减少http方法的调用在generic.py还定义了一些辅助类，在辅助类中直接实现了http方法的调用

class CategoryListView(generics.ListCreateAPIView):

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

class CategoryDetailView(generics.RetrieveUpdateDestroyAPIView):

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

# DjangoRestFrameWork基于视图集的视图

DRF在viewsets.py中使用ViewSetMixin重新封装了as\_view方法

并通过GenericViewSet完成了ViewSetMixin 与GenericAPIView的集成

1. 视图集中使用路由
2. 重写过as\_view之后我们在调用该方法的时候可以在参数中将http方法与混合类中函数绑定

as\_view({'get':'list','post':'create'})

as\_view({'get':'retrieve','put':'update','patch':'update','delete':'destroy'}

1. 而且可以直接使用自带的rest\_framework. Routers 自动生成对应的系列路由

**from** rest\_framework **import** routers  
router = routers.DefaultRouter()

router.register('categorys',CategoryViewSets)

1. 视图集定义
2. 可以直接使用拥有get、post、put、patch、delete方法的ModelViewSet

class CategoryViewSets(viewsets.ModelViewSet):

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

1. 也可以自定义拥有指定http方法的视图集

定义只拥有get的视图集

class CategoryViewSets(viewsets.GenericViewSet,mixins.ListModelMixin,mixins.RetrieveModelMixin):

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

# DjangoRestFrameWork 用户模型类

在django中可以继承自带的AbstractUser类从而方便的使用django自带的用户系统

1. 新建模型类 User

from django.contrib.auth.models import AbstractUser

class User(AbstractUser):

telephone = models.CharField(max\_length=11,verbose\_name="手机号")

1. 配置上述User类为DJango用户类

AUTH\_USER\_MODEL = 应用名.User'

1. 生成迁移文件 执行迁移

Python manage.py makemigrations

Python manage.py migrate

注意：这一步骤最好在没有创建超级管理员之前，否则可能导致数据不一致

# DjangoRestFrameWork 用户注册路由

用户路由主要有

登录：后续使用JWT认证实现

注册：序列化类 只需要使用用户名 密码 重复密码 验证码之类

显示、更新个人信息：序列化类几乎需要使用到用户的所有字段

所以关于用户可以设计两个序列化类 UserRegistSerializer以及UserSerializer

1. 注册用户资源生成用户资源路由

from rest\_framework import routers

router = routers.DefaultRouter()

router.register('users',UserViewSets)

path('api/v1/',include(router.urls)),

1. 编写用户视图集合类UserViewSets
2. 可以使用扩展资源http方法的装饰器 action

**from rest\_framework.decorators import api\_view,action**

**class UserViewSets1(viewsets.GenericViewSet,mixins.RetrieveModelMixin,mixins.UpdateModelMixin,mixins.DestroyModelMixin):**

**"""**

**声明用户资源类 用户操作： 获取个人信息 更新个人信息 删除账户**

**扩展出action路由 用户操作： 创建账户**

**"""**

**queryset = User.objects.all()**

**serializer\_class = UserSerializer**

**# 使用action扩展资源的http方法**

**@action(methods=["POST"],detail=False)**

**def regist(self,request):**

**seria = UserRegistSerializer(data=request.data)**

**seria.is\_valid(raise\_exception=True)**

**seria.save()**

**return Response(seria.data,status=status.HTTP\_201\_CREATED)**

1. 也可以使用get\_serializers 针对不同的http方法返回不同的序列化类

c**lass UserViewSets(viewsets.GenericViewSet,mixins.CreateModelMixin, mixins.RetrieveModelMixin,mixins.UpdateModelMixin,mixins.DestroyModelMixin):**

**"""**

**声明用户资源类 用户操作： 获取个人信息 更新个人信息 删除账户**

**使用action属性判定http方法 确定序列化类**

**"""**

**queryset = User.objects.all()**

**# serializer\_class = UserSerializer**

**def get\_serializer\_class(self):**

**print("action代表http方法",self.action)**

**if self.action == "create":**

**return UserRegistSerializer**

**return UserSerializer**

1. 用户序列化类与用户注册序列化类的实现
2. 用户注册序列化类

**class UserRegistSerializer(serializers.Serializer):**

**username = serializers.CharField(max\_length=10,min\_length=3,error\_messages={**

**"required":"用户名必填"**

**})**

**password = serializers.CharField(max\_length=10,min\_length=3,write\_only=True)**

**password2 = serializers.CharField(max\_length=10, min\_length=3,write\_only=True)**

**def validate\_password2(self, data):**

**if data != self.initial\_data["password"]:**

**raise serializers.ValidationError("密码不一致")**

**else:**

**return data**

**def create(self, validated\_data):**

**print("提交数据",validated\_data)**

**return User.objects.create\_user(username=validated\_data.get("username"),email=validated\_data.get("email"),password=validated\_data.get("password"))**

1. 用户序列化类

**class UserSerializer(serializers.ModelSerializer):**

**class Meta:**

**model = User**

**# fields = "\_\_all\_\_"**

**exclude = ["user\_permissions","groups"]**

**def validate(self, attrs):**

**print("原生创建")**

**from django.contrib.auth import hashers**

**if attrs.get("password"):**

**attrs["password"] = hashers.make\_password(attrs["password"])**

**print("修改之后的字段")**

**return attrs**

# DjangoRestFrameWork JWT认证

JWT为Json Web Token 是目前最流行的身份验证解决方案

JWT包含头部、有效荷载、签名三部分内容

头部表明签名使用算法以及类型

荷载在JWT需要传递的数据

签名就是对上面两部分通过加密算法进行哈希，确保数据不会被更改

基于Session的认证一般都是保存在内容或者数据库中，随着用户的增多，内存消耗太大。

并且用户只能请求session信息所在的服务器上，不利于应用的扩展。

基于token的鉴权机制类似于http协议也是无状态的，不需要在服务端去保留用户的认证信息或者会话信息。服务器只需要在内存中执行响应的运算，即可得到令牌对应的用户信息。

1. 安装djangorestframework-simplejwt

pip install djangorestframework-simplejwt

1. 配置JWT认证

REST\_FRAMEWORK = {

# 全局认证 优先级高于试图类中的配置

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework\_simplejwt.authentication.JWTAuthentication',

'rest\_framework.authentication.SessionAuthentication',

'rest\_framework.authentication.BasicAuthentication'

],

}

1. 设置获取Token路由

from rest\_framework\_simplejwt.views import token\_obtain\_pair,token\_refresh

# 先通过用户名密码 得到Token VUE将refresh以及access保存 通过access请求服务器 通过refresh获取新的access

url(r'^obtaintoken/$',token\_obtain\_pair,name='login'),

url(r'^refreshtoken/$',token\_refresh,name='refresh'),

此时可以使用POST请求 附带参数 username、password发起请求

返回示例

{

"refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV4cCI6MTU4MzMyNzA2NSwianRpIjoiOTJkZDk5YmMwNGJmNDA1MWIzZjM3MzRhMzE5Mjk0OTciLCJ1c2VyX2lkIjozfQ.HjOpI4xwkQZ1LP1BEGYQBZwBiCOVeYLNLxGipzeYb6I",

"access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTgzMjQwOTY1LCJqdGkiOiI4MmY3ZjlkNTEyZTU0N2EzYWViMWQzY2U3OWE3NTNmOCIsInVzZXJfaWQiOjN9.MYJE0RYfuS\_fDRq1nJPOOQ5vHvAqXj73EU8nz9Pkw-4"

}

Refresh 用于token的刷新、Access 用于token的认证

1. PostMan测试

配置需要使用认证的路由

Authorization 中type选择 Bear Token 然后 Token使用上述的access即可

# DjangoRestFrameWork 自定义认证类

在进行登录时默认需要使用username以及password字段，如果想要使用telephone、email字段配合密码登录则需要自定义认证类了

实现原理

Django框架中通过django.conf.global\_settings.py配置了项目默认认证类

AUTHENTICATION\_BACKENDS = ['django.contrib.auth.backends.ModelBackend']

其中ModelBackend 类就是项目默认认证类

在类中对authenticate方法进行了重写 方法中需要针对用户名、密码与数据库进行匹配

如果匹配成功返回当前用户 即认证成功 否则认证失败

1. 在应用根目录新建 auth.py

编写自定义认证类

from django.contrib.auth.backends import BaseBackend

from .models import User

from django.db.models import Q

class MyLoginBackend(BaseBackend):

def authenticate(self, request, \*\*kwargs):

username = kwargs["username"]

user = User.objects.filter(Q(email=username) | Q(telephone=username) | Q(username=username) ).first()

if user:

if user.check\_password(kwargs["password"]):

return user

return None

在自定义的MyLoginBackend类中对authenticate方法进行重写，提取kwargs中的参数，参数的值即为登录时提交的内容，可以那用户名与用户名、邮箱、手机号等字段进行匹配，有一个匹配成功即开始进行密码的匹配

Q查询是django中自带的查询包装条件，适合多个条件进行或且非逻辑判断

1. 添加自定义认证类的配置

在settings.py中添加配置

# 自定义认证类

AUTHENTICATION\_BACKENDS=("shop.auth.MyLoginBackend",)

# DjangoRestFrameWork 权限

DRF中的权限可以分为视图级别权限以及对象级别权限

在DRF中提供了 permission.py 文件对基础权限类进行实现

其中BasePermission 权限定义了has\_permission以及has\_object\_permission两种级别的权限认证

1. 全局配置

在DRF的默认配置中使用了AllowAny权限类来给与所有视图默认的权限

REST\_FRAMEWORK = {

# 全局配置 优先级低于视图类中的配置

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.AllowAny',

],

}

1. 类视图配置

可以在视图类中通过permission\_classes属性 或者 get\_permissions 方法进行权限配置

class CategoryViewSets(viewsets.ModelViewSet):

"""

分类视图

继承ModelViewSet 之后拥有GET POST PUT PATCH DELETE等HTTP动词操作

queryset 指明 需要操作的模型列表

serializer\_class 指明序列化类

"""

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

# 授权的前提是认证 登录过（认证过）之后可以进行权限判断

# 所有的http方法都需要以下权限

# 用户非管理员则不具备操作权限 优先级别高于全局配置

# permission\_classes = [permissions.IsAdminUser]

# 通过action属性判定http操作行为

# 超级管理员可以修改分类 用户是否登录都可以查看分类

def get\_permissions(self):

if self.action == "create" or self.action == "update" or self.action == "partial\_update" or self.action == "destroy":

return [permissions.IsAdminUser()]

else:

return []

1. 自定义权限类

"""  
可以自定义权限  
"""  
from rest\_framework import permissions  
  
class CategorysPermission(permissions.BasePermission):  
 def has\_permission(self, request, view):  
 return True  
class OrdersPermission(permissions.BasePermission):  
 def has\_object\_permission(self, request, view, obj):  
 # 当对象的拥有者为当前用户时 赋予操作权限  
 return obj.user == request.user

# DjangoRestFrameWork 频次组件

为了降低服务器的开销，减少不必要的成本，我们需要限制非正常手段、非人为操作类的请求。DRF中给我们封装好了频次限制组件。原理是当服务器收到请求时会将发起请求的主机IP以及请求时间进行记录。当请求频率超过限定时不再给与响应。

1. 在Rest\_framework.throttling.py 中给我们定义了BaseThrottle 类用以实现频次控制，在自定义的控制类中可以对其进行重写。
2. 在settings.py中为我们提供了DEFAULT\_THROTTLE\_CLASSES与DEFAULT\_THROTTLE\_RATES用来实现对频次控制类的全局指定。
3. 在rest\_framework.views.APIView中throttle\_classes = api\_settings.DEFAULT\_THROTTLE\_CLASSES 声明了视图类默认使用的频次限制类为配置文件中指定的频次限制类，在视图中我们可以对throttle\_classes 属性进行重写，来覆盖全局指定。

具体执行逻辑 当调用视图类的as\_view()方法时会通过 dispatch执行视图类下对应http方法名的方法。执行http方法前会在initial(request, \*args, \*\*kwargs)函数中对频次、认证、授权之类的进行验证，如果验证通过则执行对应的http方法，否则抛出异常

1. 配置全局频次限定

REST\_FRAMEWORK = {

'DEFAULT\_THROTTLE\_CLASSES': [

#配置匿名用户访问频次类

'rest\_framework.throttling.AnonRateThrottle',

#配置认证用户访问频次类

'rest\_framework.throttling.UserRateThrottle'

],

'DEFAULT\_THROTTLE\_RATES': {

#配置匿名用户访问频次

'anon': '1/day',

#配置认证用户访问频次

'user': '2/day'

},

}

二、自定义频次限制类

from rest\_framework import throttling

class MyAnon(throttling.AnonRateThrottle):

THROTTLE\_RATES = {

'anon':'100/day'

}

class MyUser(throttling.UserRateThrottle):

THROTTLE\_RATES = {

'user':'200/day'

}

三、在视图类中指定频次类

class CategoryViewSets(viewsets.ModelViewSet):

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

# 部分视图限制频次

throttle\_classes = [MyAnon,MyUser]

# DjangoRestFrameWork 分页组件

在使用DRF的分页之前呢，先来了解下Django自带的分页功能

from django.core.paginator import Paginator,Page

# 一个Page中有 object\_list代表当前页的所有对象

# has\_next 是不是有下一页

# has\_previous 是否有上一页

# next\_page\_number 下一页的编号

# previous\_page\_number 上一页的编号

# self.number 当前页的编号

# self.paginator 当前页的分页器

# 一个Paginator中的object\_list 代表所有未分页对象

# self.per\_page 每一页有几个对象

# get\_page(self, number): 从分页器中取第几页

# page\_range(self): 返回分页列表

Django原生通过Paginator与Page对象完成了分页相关实现

DRF对Django的分页进行了封装

在rest\_framework.pagination.py 总使用BasePagination类对响应的分类方法进行了实现

其中最主要的是paginate\_queryset与get\_paginated\_response方法，通过类PageNumberPagination、LimitOffsetPagination、CursorPagination实现了基本的分页形式。

当然我们也可以对必要的分页实现进行重写

1. 配置全局分页

在settings.py中配置全局分页器

REST\_FRAMEWORK = {

# 全局配置分页

# 'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.LimitOffsetPagination',

# 'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

# 'PAGE\_SIZE': 2,

}

在rest\_framework.generics.py 中的GenericAPIView类中使用pagination\_class = api\_settings.DEFAULT\_PAGINATION\_CLASS声明了默认使用全局分页配置

二、自定义分页

from rest\_framework import pagination

class MyPagination(pagination.PageNumberPagination):

page\_size = 3

page\_query\_param = 'p'

page\_size\_query\_param = 'num'

三、视图中配置自定义分页

class CategoryViewSets(viewsets.ModelViewSet):

queryset = Category.objects.all()

serializer\_class = CategorySerizlizer

#配置视图类使用的分页

# pagination\_class = MyPagination

# DjangoRestFrameWork 过滤组件

1. 安装django-filter

pip install django-filter

1. 注册应用

INSTALLED\_APPS = [  
 'django\_filters'  
]

1. 配置全局过滤

REST\_FRAMEWORK = {

'DEFAULT\_FILTER\_BACKENDS': ['django\_filters.rest\_framework.DjangoFilterBackend']

}

1. 配置视图过滤

filter\_backends = [DjangoFilterBackend]

# 为每一个需要过滤的视图都需要添加filterset\_fields 属性  
filterset\_fields = ['name']

1. 使用搜索

from rest\_framework import filters

filter\_backends = [filters.SearchFilter]

# 为每一个需要搜索的视图都需要添加search\_fields 属性

search\_fields = ["name"]

1. 使用排序

from rest\_framework import filters

# 为每一个需要排序的视图都需要添加ordering\_fields 属性

filter\_backends = [filters.OrderingFilter]

ordering\_fields = ["id"]

# DjangoRestFrameWork 跨域

浏览器的同源策略即同一个域名同一个端口下可以访问数据

默认不允许跨域名，跨端口访问数据

1. 安装模块

pip install django-cors-headers

1. 配置INSTALLED\_APPS和MIDDLEWARE

INSTALLED\_APPS = (

...

'corsheaders',

...

)

MIDDLEWARE = [

...

#注意CorsMiddleware需放在CsrfViewMiddleware 之前

'corsheaders.middleware.CorsMiddleware',

'django.middleware.common.CommonMiddleware',

...

]

1. 设置CORS\_ORIGIN\_ALLOW\_ALL

#setting.py

CORS\_ORIGIN\_ALLOW\_ALL = True