RESTful设计

HTTP常用动词:

GET（SELECT）：从服务器取出资源（一项或多项）。

POST（CREATE）：在服务器新建一个资源。

PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）。

DELETE（DELETE）：从服务器删除资源。

不常用:

PATCH（UPDATE）：在服务器更新(更新)资源（客户端提供改变的属性）。

HEAD：获取资源的元数据。

OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的。

过滤信息

?limit=10：指定返回记录的数量

?offset=10：指定返回记录的开始位置。

?page=2&per\_page=100：指定第几页，以及每页的记录数。

?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序。

?animal\_type\_id=1：指定筛选条件

状态码

200 OK - [GET]：服务器成功返回用户请求的数据

201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。

202 Accepted - [\*]：表示一个请求已经进入后台排队（异步任务）

204 NO CONTENT - [DELETE]：用户删除数据成功。

400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作

401 Unauthorized - [\*]：表示用户没有权限（令牌、用户名、密码错误）。

403 Forbidden - [\*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。

404 NOT FOUND - [\*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。

406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。

410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。

422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。

500 INTERNAL SERVER ERROR - [\*]：服务器发生错误，用户将无法判断发出的请求是否成功。

序列化 dict🡪json json.dumps() 上传

反序列化 json🡪dict json.loads() 下载

DRF框架

添加rest\_framework

INSTALLED\_APPS = [

...

'rest\_framework',

]

新建serializers.py

from rest\_framework.viewsets import ModelViewSet

from .serializers import BookInfoSerializer

from .models import BookInfo

class BookInfoViewSet(ModelViewSet):

queryset = BookInfo.objects.all()

serializer\_class = BookInfoSerializer

编写视图

from rest\_framework.viewsets import ModelViewSet

from .serializers import BookInfoSerializer

from .models import BookInfo

class BookInfoViewSet(ModelViewSet):

queryset = BookInfo.objects.all()

serializer\_class = BookInfoSerializer

定义路由

from . import views

from rest\_framework.routers import DefaultRouter

urlpatterns = [...]

router = DefaultRouter() # 可以处理视图的路由器

router.register(r'books', views.BookInfoViewSet) # 向路由器中注册视图集

urlpatterns += router.urls # 将路由器中的所以路由信息追到到django的路由列表中

定义serializer

class BookInfoSerializer(serializers.Serializer):

"""图书数据序列化器"""

id = serializers.IntegerField(label='ID', read\_only=True)

btitle = serializers.CharField(label='名称', max\_length=20)

bpub\_date = serializers.DateField(label='发布日期', required=False)

bread = serializers.IntegerField(label='阅读量', required=False)

bcomment = serializers.IntegerField(label='评论量', required=False)

image = serializers.ImageField(label='图片', required=False)

创建serializer对象

BookInfoSerializer(instance=None, data=empty, context={'request': request})

instance:序列化时传入模型类对象

data:传入字典数据

context:添加额外的数据

序列化使用

book = BookInfo.objects.get(id=2) # 创建模型类对象

s = BookInfoSerializer(book) # 构造序列化器对象

s.data # 获取序列化数据

# 被序列化的是多值查询集,增加many=True来说明

books = BookInfo.objects.all()

s = BookInfoSerializer(books, many=True)

多对一的关联对象序列化

class HeroInfoSerializer(serializers.Serializer):

"""英雄数据序列化器"""

GENDER\_CHOICES = ((0, 'male'), (1, 'female'))

...

hgender = serializers.ChoiceField(choices=GENDER\_CHOICES, label='性别', required=False)

hcomment = serializers.CharField(label='描述信息', max\_length=200, required=False, allow\_null=True)

对于关联字段hbook创建

1.PrimaryKeyRelatedField此字段将被序列化为关联对象的主键

hbook = serializers.PrimaryKeyRelatedField(label='图书', read\_only=True)

或

hbook = serializers.PrimaryKeyRelatedField(label='图书', queryset=BookInfo.objects.all())

2. StringRelatedField此字段将被序列化为关联对象的字符串表示方式（即\_\_str\_\_方法的返回值）

hbook = serializers.StringRelatedField(label='图书')

3. 使用关联对象的序列化器

hbook = BookInfoSerializer()

4. HyperlinkedRelatedField此字段将被序列化为获取关联对象数据的接口链接(eg: 'hbook': 'http://127.0.0.1:8000/books/2/')

hbook = serializers.HyperlinkedRelatedField(label='图书', read\_only=True, view\_name='books-detail')

5. SlugRelatedField此字段将被序列化为关联对象的指定字段数据

hbook = serializers.SlugRelatedField(label='图书', read\_only=True, slug\_field='bpub\_date')

6.重写to\_representation方法

序列化器的每个字段实际都是由该字段类型的to\_representation方法决定格式的

eg:自定义一个新的字段

class BookRelateField(serializers.RelatedField):

"""自定义用于处理图书的字段"""

def to\_representation(self, value):

return 'Book: %d %s' % (value.id, value.btitle)

...

hbook = BookRelateField(read\_only=True)

...

一对多的关联序列 many参数

class BookInfoSerializer(serializers.Serializer):

...

heroinfo\_set = serializers.PrimaryKeyRelatedField(read\_only=True, many=True)

反序列化使用

验证

data = {...} # 获取到的字典数据

s = BookInfoSerializer(data=data)

s.is\_vaild() # 合法返回True,否则False

s.errors # 返回报错信息,合法为空

s. validated\_data # 验证成功获得数据

s.is\_valid(raise\_exception=True) # raise\_exception开启后,is\_vaild()接收到serializers.ValidationError异常后,会向前端返回HTTP 400 Bad Request响应

3种自定义验证方法

1. validate\_<field\_name>

对<field\_name>字段进行验证

class BookInfoSerializer(serializers.Serializer):

...

def validate\_btitle(self, value):

if 'django' not in value.lower():

raise serializers.ValidationError("图书不是关于Django的")

return value

1. validate

在序列化器中需要同时对多个字段进行比较验证时，可以定义validate方法来验证

class BookInfoSerializer(serializers.Serializer):

...

def validate(self, attrs):

bread = attrs['bread']

bcomment = attrs['bcomment']

if bread < bcomment:

raise serializers.ValidationError('阅读量小于评论量')

return attrs

1. validators

在字段中添加validators选项参数，补充验证行为(类似第一种字段验证,方便复用)

def about\_django(value):

if 'django' not in value.lower():

raise serializers.ValidationError("图书不是关于Django的")

class BookInfoSerializer(serializers.Serializer):

"""图书数据序列化器"""

id = serializers.IntegerField(label='ID', read\_only=True)

btitle = serializers.CharField(label='名称', max\_length=20, validators=[about\_django])

...

REST framework提供的validators

单字段唯一

from rest\_framework.validators import UniqueValidator

slug = SlugField( max\_length=100,

validators=[UniqueValidator(queryset=BlogPost.objects.all())]

)

联合唯一

from rest\_framework.validators import UniqueTogetherValidator

class ExampleSerializer(serializers.Serializer):

# ...

class Meta:

validators = [

UniqueTogetherValidator(

queryset=ToDoItem.objects.all(),

fields=('list', 'position')

)

]

保存save()

执行save返回对象同时也保存到数据库

删除红色部分就返回数据的对象,未保存到数据库

class BookInfoSerializer(serializers.Serializer):

...

def create(self, validated\_data):

"""新建"""

return BookInfo.objects.create(\*\*validated\_data)

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

"""更新，instance为要更新的对象实例"""

instance.btitle = validated\_data.get('btitle', instance.btitle)

...

instance.save()

return instance

eg:

data = {...}

s = BookInfoSerializer(data=data)

s.is\_vaild()

book = s.save()

补充:

在对序列化器进行save()保存时，可以额外传递数据，这些数据可以在create()和update()中的validated\_data参数获取到

serializer.save(owner=request.user)

默认序列化器必须传递所有required的字段，否则会抛出验证异常。但可以使用partial参数来允许部分字段更新

serializer = CommentSerializer(comment, data={'content': u'foo bar'}, partial=True)

模型类序列化器ModelSerializer

class BookInfoSerializer(serializers.ModelSerializer):

"""图书数据序列化器"""

class Meta:

model = BookInfo

fields = '\_\_all\_\_'

1. \_\_all\_\_表明所有字段, fields能用来指定字段

fields = ('id', 'btitle', 'bpub\_date')

2. exclude排除指定字段

exclude = ('image',)

3. depth来简单的生成嵌套表示，depth是整数，表明嵌套的层级数量

depth = 1

1. 显示指明字段

class HeroInfoSerializer(serializers.ModelSerializer):

hbook = BookInfoSerializer()

class Meta:

model = HeroInfo

fields = ('id', 'hname', 'hgender', 'hcomment', 'hbook')

5.指明只读字段read\_only\_fields

class BookInfoSerializer(serializers.ModelSerializer):

"""图书数据序列化器"""

class Meta:

model = BookInfo

fields = ('id', 'btitle', 'bpub\_date'， 'bread', 'bcomment')

read\_only\_fields = ('id', 'bread', 'bcomment')

添加额外参数extra\_kwargs

class BookInfoSerializer(serializers.ModelSerializer):

"""图书数据序列化器"""

class Meta:

model = BookInfo

fields = ('id', 'btitle', 'bpub\_date'， 'bread', 'bcomment')

extra\_kwargs = {

'bread': {'min\_value': 0, 'required': True}},

'bcomment': {'max\_value': 0, 'required': True}},

}

Request请求

Request对象的数据是自动根据前端发送数据的格式进行解析之后的结果。

1. .data

request.data 返回解析之后的请求体数据, 类似于Django中标准的request.POST和 request.FILES属性

1. . query\_params

request.query\_params与Django标准的request.GET相同

Response响应

setting中配置默认相应格式

REST\_FRAMEWORK = {

'DEFAULT\_RENDERER\_CLASSES': ( # 默认响应渲染类

'rest\_framework.renderers.JSONRenderer', # json渲染器

'rest\_framework.renderers.BrowsableAPIRenderer', # 浏览API渲染器

)

}

构造方法

Response(data, status=None, template\_name=None, headers=None, content\_type=None)

1. .data

传给response对象的序列化后，但尚未render处理的数据

1. .status\_code

状态码的数字

1. .content

经过render处理后的响应数据

视图

APIView 🡪 GenericAPIView 🡪 ListAPIView

ListModelMixin

eg:扩展类Mixin和GenericAPIView

from rest\_framework.mixins import ListModelMixin

class BookListView(ListModelMixin, GenericAPIView):

queryset = BookInfo.objects.all()

serializer\_class = BookInfoSerializer

def get(self, request):

return self.list(request)

⬄

class BookListView(ListAPIView):

queryset = BookInfo.objects.all()

serializer\_class = BookInfoSerializer

五个扩展类

ListModelMixin get列表

CreateModelMixin post

RetrieveModelMixin get指定

UpdateModelMixin put

DestroyModelMixin delete

对应的几个子类视图

CreateAPIView

ListAPIView

RetireveAPIView

DestoryAPIView

UpdateAPIView

RetrieveUpdateAPIView

RetrieveUpdateDestoryAPIView

视图集(适合对数据库操作)

ViewSet视图集类不再实现get()、post()等方法，而是实现动作action如list()、create()等,在get请求上添加action操作。

list() 提供一组数据

retrieve() 提供单个数据

create() 创建数据

update() 保存数据

destory() 删除数据

常用视图集父类

1） ViewSet

继承自APIView，作用也与APIView基本类似，提供了身份认证、权限校验、流量管理等。

2）GenericViewSet

继承自GenericAPIView，作用也与GenericAPIVIew类似，提供了get\_object、get\_queryset等方法便于列表视图与详情信息视图的开发。

3）ModelViewSet

继承自GenericAPIVIew，同时包括了ListModelMixin、RetrieveModelMixin、CreateModelMixin、UpdateModelMixin、DestoryModelMixin。

4）ReadOnlyModelViewSet

继承自GenericAPIVIew，同时包括了ListModelMixin、RetrieveModelMixin。

eg:

from rest\_framework import mixins

from rest\_framework.viewsets import GenericViewSet

from rest\_framework.decorators import action

class BookInfoViewSet(mixins.ListModelMixin, mixins.RetrieveModelMixin, GenericViewSet):

queryset = BookInfo.objects.all()

serializer\_class = BookInfoSerializer

# detail为False 表示不需要处理具体的BookInfo对象

@action(methods=['get'], detail=False)

def latest(self, request):

book = BookInfo.objects.latest('id')

serializer = self.get\_serializer(book)

return Response(serializer.data)

# detail为True，表示要处理具体与pk主键对应的BookInfo对象

@action(methods=['put'], detail=True)

def read(self, request, pk):

book = self.get\_object()

book.bread = request.data.get('read')

book.save()

serializer = self.get\_serializer(book)

return Response(serializer.data)

url中:

urlpatterns = [

url(r'^books/$', views.BookInfoViewSet.as\_view({'get': 'list'})),

url(r'^books/latest/$', views.BookInfoViewSet.as\_view({'get': 'latest'})),

url(r'^books/(?P<pk>\d+)/$', views.BookInfoViewSet.as\_view({'get': 'retrieve'})),

url(r'^books/(?P<pk>\d+)/read/$', views.BookInfoViewSet.as\_view({'put': 'read'})),

]

路由Routers

创建router对象

from rest\_framework import routers

router = routers.SimpleRouter()

router.register(r'books', BookInfoViewSet, base\_name='book')

添加路由

urlpatterns = [ 或 urlpatterns = [

... ...

] url(r'^', include(router.urls))

urlpatterns += router.urls ]

路由形成URL方式

SimpleRouter和DefaultRouter

DefaultRouter会多附带一个默认的API根视图，返回一个包含所有列表视图的超链接响应数据

其他功能

[认证](ENoteBook/django/DRF框架/index.html)

[权限](ENoteBook/django/DRF框架/index.html)

[限流](ENoteBook/django/DRF框架/index.html)

[过滤](ENoteBook/django/DRF框架/index.html)

[分页](ENoteBook/django/DRF框架/index.html)

[版本](ENoteBook/django/DRF框架/index.html)

[异常处理](ENoteBook/django/DRF框架/index.html)

[自动生成接口文档](ENoteBook/django/DRF框架/index.html)