🡪

DRF框架提供Serializer支持（例如ModelSerializer基类, 可以重写create，update方法）,用于支持序列化反序列化。

DRF框架提供View支持（例如APIVIEW基类，generics.ListCreateAPIView, Mixins基类， 可以重写get, post方法）。用于处理不同的restful请求。

对于一个Restful POST请求，我们先用view接受该请求，然后在view中使用serializer来验证数据更新数据库。

对于一个Restful GET请求，我们先用view接受该请求，然后在view中从数据库取出数据，使用serializer来序列化结果返回结果。

🡪

Django Rest Framework提供特别的Request对象，其中的request数据，使用request.data来获取。Request.data装载来自于GET, POST, PUT请求的所有数据。而普通的Django框架中的Request对象，不存在request.data属性。Django框架中普通的Request对象，只有request.POST对应来自于POST请求的数据，request.GET对应来自于GET请求的数据。

Django Rest Framework提供特别的Response对象，它接受还没有渲染的数据，并决定返回什么数据类型给client.

Django Rest Framework提供特别的Status模块，用于提供更明确的状态码。

Django Rest Framework提供三种特别的Wrapper，这些Wrapper将保证在处理函数中存在Request对象(这个Request对象不是普通的Django下的HTTPRequest对象，而是专门针对DRF的Request对象) ，并且为返回的Response对象添加context用于Response返回内容类型时候的协商（这个Response对象也不是是普通的Django下的HttpResponse对象，而是专门针对DRF的HttpResponse对象）。（我们通过这些Wrapper去Wrap我们的处理函数，从而提供一些默认的功能。这是推荐的做法。从本质来说，我们也可直接写函数在views当中，但是不被任何wrapper, 也可以达到目的，但是此时很多其他的情况都需要考虑）：

第一种wrapper是@api\_view,用于装饰函数,默认只接受GET方法，POST等方法需要显示指明。

第二种wrapper是APIView基类。使用情况为完全自主定义行为

第三种wrapper是generics.\*基类（generics类也是对mixin类的进一步抽象整合），其默认实现部分CRUD操作这些Generic类继承APIView, 并且扩充其方法。我们可以根据需要选择不同的Generic类，也可以在选择的基础上再去override一些方法。

第三种是Viewsets基类，在APIVIEW的基础上，默认的提供`list`, `create`, `retrieve`,

`update` and `destroy`方法。也就是默认实现CRUD方法。在接下来的urls.py中将list, create, retrieve, update, destroy方法绑定到http的get, post, patch, delete请求即可。

且经常和Router类联用。当使用了Router类以后，我们不需要在urls.py中显示的写出什么路径对应什么函数，DRF会根据我们在router传入的viewsets, 直接自动的构建出对应的url地址。这样的好处是所有自动构建的url地址都是由DRF自动生成的，所以具有非常高的统一性。但是不好的地方是，没有我们手动指定的语义明显。

🡪

generic.ListCreateAPIView基类，展示实例或者创建。支持GET和POST请求。

generic.RetrieveDestroyAPIView基类，获取单独实例或者删除他们。支持GET和DELETE请求。

generic.CreateAPIView基类，创建实例。支持POST请求。

--重写类中的get\_queryset方法可以为返回的结果添加筛选条件

--重写类中的post方法可以修改当接受POST请求时的行为

🡪

DRF提供多种Resful服务的认证服务。包括最流行的token认证，先使用账号密码登录然后获取token, 使用该token进行其他restful请求。

🡪

相比于普通的Django Framework, DRF需要url, view和序列器。url用于控制API的终点站在哪，view定义到数据库取数据的逻辑，序列器用于将我们需要的数据变为JSON格式进行传输。

🡪

当我们引入DRF到现有project的时候，我们可以为每个原本的现有app添加serializers.py, 在views.py中添加函数。但是最好的做法是，新建一个app, 这个app专门管理DRF, 在这个新建的app中一样可以读取其他apps的models, 为其指定序列器，views等。最后在这个app中采用router + viewsets的方式进行抽象。

🡪

在urls.py定义中format\_suffix\_patterns用于为url增加后缀，

例如增加以后，将从<http://example.com/api/items/4.json>也会被接受，而不止<http://example.com/api/items/4>

🡪

Class-based与function-based的对于GET, POST方法的不同。

**如果是function-based的函数**，在处理Request的不同请求方法时为：

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

def snippet\_detail(request, pk): # 该pk的值由urls.py中(?P<pk>.\*)获得。并且和普通Django框架一样，第一个参数是request,第二个参数开始是由urls.py命名参数中获取

"""

Retrieve, update or delete a code snippet.

"""

try:

snippet = Snippet.objects.get(pk=pk)

except Snippet.DoesNotExist:

return Response(status=status.HTTP\_404\_NOT\_FOUND)

if request.method == 'GET':

serializer = SnippetSerializer(snippet)

return Response(serializer.data)

elif request.method == 'PUT':

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)

elif request.method == 'DELETE':

snippet.delete()

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

**如果是class-based的函数时**，在处理Request的不同请求方法时为：

class SnippetDetail(APIView):

"""

Retrieve, update or delete a snippet instance.

"""

def get\_object(self, pk): # 该pk的值由下面处理get请求，put请求，delete请求的时候，调用get\_object方法的时候传入

try:

return Snippet.objects.get(pk=pk)

except Snippet.DoesNotExist:

raise Http404

def get(self, request, pk, format=None): # 该pk的值由urls.py中(?P<pk>.\*)获得。并且和普通Django框架一样，第一个参数是request,第二个参数开始是由urls.py命名参数中获取

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)

snippet.delete()

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

🡪

我们在views中所读到的requests.data是不可变的，所以无法直接通过requests.data[‘field\_name’] = “…” 去为requests.data增加或者改变值。

🡪

在对DRF进行PUT方向执行serializers的update方法的时候，默认情况下，不需要提供所有modesl.py中定义的所有数据域，但是必须提供modesl.py中不允许为空的数据域，并且update默认改变提交数据中所包含的数据域，不改变其他数据域。

🡪

RESTful API规范：

GET： 从服务器取出资源

POST: 从服务器新建一个资源

PUT： 在服务器更新资源（客户端提供改变后的完整资源），提交更新的时候是提交完整的对象。

PATCH： 在服务器更新资源（客户端提供改变的属性），提交更新的时候只提交改变的数据域。

DELETE： 在服务器删除资源

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

🡪

DRF中的views或者serializers改变是需要重启服务器进行生效的。

🡪

DRF的配置在settings.py中会专门以REST\_FRAMEWORK的字段，并且以字典的形式进行配置。其他的配置包括DRF的访问权限，分页等。

🡪

Mixins的用法。DRF提供了Mixin类，这个类中提供了许多REST中常用的操作，例如list展示所有存在数据库中的对象，create创建对象，delete删除对象等。

例子：

如果不采用mixins类，只采用APIView作为基类，则要列出所有记录时，通常会这样写：

class PatchTrackingList(APIView):  
 """  
 List all PatchTrackingList, or create a new PatchTracking Record.  
 """  
 def get(self, request, format=None):  
 patch\_tracking\_objs = rgr\_patch\_tracking\_tbl.objects.all()  
 serializer = PatchTrackingSerializer(patch\_tracking\_objs, many=True)  
 return Response(serializer.data)  
  
 def post(self, request, format=None):  
 serializer = PatchTrackingSerializer(data=request.data)  
 if serializer.is\_valid():  
 serializer.save()  
 return Response(serializer.data, status=status.HTTP\_201\_CREATED)  
 return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

这是列出所有PatchTrackingList的view函数，同样的对于其他需要操作的表执行的操作都相同，在get方法中都是先找到所有对象，然后序列化。对于post都是先给序列器加入数据，验证是否合法，然后调用序列器的save()方法。基于这样重复的行为，DRF在mixins中添加许多子类，用于对应这些常用的操作。

Mixins.ListModelMixin 🡪 提供list方法，用于展示所有的对象。

Mixins.CreateModelMixin 🡪 提供create方法，用于创建和保存一个新的对象。

Mixins.RetrieveModelMixin 🡪 用于查询一个在数据库中的对象。

Mixins.UpdateModelMixin 🡪 用于更新对象，更新的对象包括PUT更新，以及只上传部分更新域的PATCH更新。

Mixins.DestroyModelMixin 🡪 用于删除一个对象。

所以上面采用APIView的view函数可以改为：

class PatchTrackingList(mixins.ListModelMixin, 🡪表示可用于列出所有对象  
 mixins.CreateModelMixin, 🡪表示可以用于创建对象  
 generics.GenericAPIView): 🡪加入通用函数  
 """  
 List all PatchTrackingList, or create a new PatchTracking Record.  
 """  
 queryset = rgr\_patch\_tracking\_tbl.objects.all()🡪为queryset提供重写值  
 serializer\_class = PatchTrackingSerializer 🡪采用什么样的序列器  
  
 def get(self, request, \*args, \*\*kwargs):  
 return self.list(request, \*args, \*\*kwargs)  
  
 def post(self, request, \*args, \*\*kwargs):  
 return self.create(request, \*args, \*\*kwargs)

同样的我们可以改写针对每个对象的view方法：

原来：

class PatchTrackingDetail(APIView):  
 """  
 Retrieve, update or delete a rgr\_patch\_tracking\_tbl instance.  
 """  
 def get\_object(self, build\_slug):  
 try:  
 return rgr\_patch\_tracking\_tbl.objects.get(build\_\_slug=build\_slug)  
 except rgr\_patch\_tracking\_tbl.DoesNotExist:  
 raise Http404  
  
 def get(self, request, build\_slug, format=None):  
 patch\_tracking\_obj = self.get\_object(build\_slug)  
 serializer = PatchTrackingSerializer(patch\_tracking\_obj)  
 return Response(serializer.data)  
  
 def put(self, request, pk, format=None):  
 patch\_tracking\_obj = self.get\_object(pk)  
 serializer = PatchTrackingSerializer(patch\_tracking\_obj, data=request.data)  
 if serializer.is\_valid():  
 serializer.save()  
 return Response(serializer.data)  
 return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)  
  
 def patch(self, request, pk, format=None):  
 patch\_tracking\_obj = self.get\_object(pk)  
 serializer = PatchTrackingSerializer(patch\_tracking\_obj, data=request.data, partial=True)  
 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):  
 patch\_tracking\_obj = self.get\_object(pk)  
 patch\_tracking\_obj.delete()  
 return Response(status=status.HTTP\_204\_NO\_CONTENT)

使用Mixins:

class PatchTrackingDetail(mixins.RetrieveModelMixin,🡪处理取得某个特定的object  
 mixins.UpdateModelMixin, ->更新某个特定的object  
 mixins.DestroyModelMixin, 🡪删除某个特定的object  
 generics.GenericAPIView): 🡪提供通用功能  
 """  
 Retrieve, update or delete a rgr\_patch\_tracking\_tbl instance.  
 """  
 queryset = rgr\_patch\_tracking\_tbl.objects.all()🡪为queryset提供重写值  
 serializer\_class = PatchTrackingSerializer 🡪 选择序列器  
  
 def get(self, request, \*args, \*\*kwargs):  
 return self.retrieve(request, \*args, \*\*kwargs)  
  
 def put(self, request, \*args, \*\*kwargs):  
 return self.update(request, \*args, \*\*kwargs)  
  
 def delete(self, request, \*args, \*\*kwargs):  
 return self.destroy(request, \*args, \*\*kwargs)

🡪

使用抽象程度更高的Generic类

我们通过使用Mixins,进行一定程度的抽象，view变为：

class PatchTrackingDetail(mixins.RetrieveModelMixin,🡪处理取得某个特定的object  
 mixins.UpdateModelMixin, ->更新某个特定的object  
 mixins.DestroyModelMixin, 🡪删除某个特定的object  
 generics.GenericAPIView): 🡪提供通用功能,这个是必须的因为RetrieveModelMixin,里面调用的方法retrieve里面使用get\_object方法，如果在我们不重写的情况下，使用的便是GenericAPIView里面的get\_object方法。  
 """  
 Retrieve, update or delete a rgr\_patch\_tracking\_tbl instance.  
 """  
 queryset = rgr\_patch\_tracking\_tbl.objects.all()🡪为queryset提供重写值  
 serializer\_class = PatchTrackingSerializer 🡪 选择序列器  
  
 def get(self, request, \*args, \*\*kwargs):  
 return self.retrieve(request, \*args, \*\*kwargs)  
  
 def put(self, request, \*args, \*\*kwargs):  
 return self.update(request, \*args, \*\*kwargs)  
  
 def delete(self, request, \*args, \*\*kwargs):  
 return self.destroy(request, \*args, \*\*kwargs)

这样的确比用APIView简单，但是对于每一个view都要输入mixins.xxx, mixins,xxx.所以DRF在Generics中提供的某些子类，可以直接让Generic的子类，直接继承这些Mixins类。

改写为：

class SnippetDetail(generics.RetrieveUpdateDestroyAPIView): 🡪这个类包含了mixins.Retrieve,

🡪mixins.Update, mixins.Destory,

🡪以及通用函数

queryset = Snippet.objects.all()

serializer\_class = SnippetSerializer

🡪

Generic View的queryset, serializer\_class属性。

在DRF文档中，可以看到继承Generic View的Class-based View中将指定queryset,以及serializer\_class属性。例如：

class UserList(generics.ListCreateAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

**queryset属性设置**的含义在于此View默认返回什么querySet.例如generics.ListCreateAPIView默认支持GET请求并且返回数据。在不重写任何代码情况下，当收到GET请求时，此View默认调用get\_queryset()方法，而get\_queryset()方法默认返回queryset属性所指定的数据。

在此有两个默认行为， 当处理GET请求时，默认调用get\_queryset()方法。 并且get\_queryset()方法默认返回queryset属性。

从而如果我们想接收到GET请求时候，进行过滤结果再返回。

我们可以重写def get方法，使其不是默认的调用get\_querytset()方法。在重写的get方法中，手动根据url参数获取queryset进行返回。

我们也可以重写get\_queryset()方法。使其不是默认的返回queryset属性。

我们可以设置queryset = User.objects.filter(name\_\_contains=”GGG”).使其在GET请求情况下，返回的用户数据是含有GGG名字的用户。

serializer\_class**属性设置**的含义在于当此View进行序列化反序列化数据的时候，默认采用什么序列器。 例如generics.ListCreateAPIView默认支持GET, POST请求。 在GET, POST请求中需要进行序列化，以及反序列化行为。在不重写GET, POST方法的情况下，默认使用什么序列器来进行序列化反序列化。

🡪

genericView中的RetrieveUpdateDestroyAPIView类，里面包含了对于单一的对象，可以进行查询，更新，删除。但是默认的get\_object行为是根据pk来查询，进而再进行更新等操作的。如果我们想根据其他域来查找特定的object时候，

我们可以在views.py中重写他的get\_object方法，那么其余操作会因为我们重写的这个方法，在寻找特定object的时候，采用不同于pk的查询方式。

🡪

DRF通常对URL需要添加两个额外的信息，一个头一个尾巴。意思是是头部将添加/api/的根目录，作用是展示出所有的可用api地址，称作api\_root. 一个是尾巴称作,suffix,即是满足,自动根据url判别需要的返回类型。

🡪

DRF框架自带测试DRF RESTFUL API的包， APIRequestFactory, APIClient, APITestCase.大致流程为为每个测试用例设置user进行登录，然后指定进行的请求，参数以及期待的HTTPRESPONSE状态码。

🡪

URL地址中的参数。

URL地址中参数包含两个部分：URL-PATH以及Query argument。在进行GET请求时候，经常通过URL-PATH以及Query argument来过滤GET返回的结果。除此以外，GET请求也可以放置参数在请求体BODY中，但是不推荐。

URL-PATH中一般包含的是指明资源实体，或者资源ID,或者某些重要的必须指定的条件

例如：

/api/resources/fruit?category=apple # /api/resources/fruit为URL-PATH其中fruit表明对fruit资源实体进行操作

/api/resources/fruit/1 # /api/resources/fruit/1为URL-PATH，其中fruit表明对fruit资源实体ID为1进行操作

/v1/{tenant\_id}/stacks # /v1/{tenant\_id}/stacks为URL-PATH，其中{tenant\_id}/表明必要的必填的条件

Query argument中一般包括可选条件，可以指明也可以不指定。 包括返回的格式，包括分页的页数等。

例如：

/api/resources/fruit?category=apple # ?category=apple为Query Parameter,指明为apple则返回apple类型水果。如果不知名则返回所有类型水果。

/api/resources/fruit?category=apple&format=csv # ?category=apple&format=csv为Query Parameter,指明为apple则返回apple类型水果，并且要求返回为csv格式。同时也可以不指定format=csv返回其他默认格式例如JSON字符串。

例子：

/api/resources/fruit?category=apple

其中/api/resources/fruit为URL-Path部分，这部分用于表明资源实体。例如/api/resources/fruit用于获取关于fruit的资源。

category=apple为Query argument部分，这部分主要用于限制获取的结果集。例如在指向/api/resources/fruit的实体下，我们只想获取apple类别的fruit结果集，所以在Query argument中指定。

如果需要获取fruit资源以及farm资源时候，因为属于不同的资源实体，所以应该指定不同的URL-PATH.

分别为/api/resources/fruit以及/api/resources/farm.而找寻类别为苹果的fruit,或者处于某个地区的farm,这些属于对某一资源实体，添加搜索条件，限制获取结果集。此时应该使用Query argument.

当返回结果集为空时，返回200 HTTP代码，表达查询合法但结果为空。

例子2：

对某个ID的资源进行查询或者更改。一般来说，ID对于资源是唯一的标识符，返回的不是结果集。所以一般将ID设置在URL PATH中而不是Query argument中。

例如：

/api/resources/fruit/1

当返回结果集为空时，返回404 HTTP代码，表达ID标识符所对应的资源不存在。

🡪DRF获取URL地址中的参数

在DRF中，通过**self.kwargs**来获取URL-PATH部分的参数

例如：

urls.py中定义：

re\_path('^purchases/(?P<username>.+)/$', PurchaseList.as\_view())

则在View中通过self.kwargs获取参数

class PurchaseList(generics.ListAPIView):

serializer\_class = PurchaseSerializer

def get\_queryset(self):

"""

This view should return a list of all the purchases for

the user as determined by the username portion of the URL.

"""

username = self.kwargs['username']

return Purchase.objects.filter(purchaser\_\_username=username)

在DRF中，通过**self.request.query\_params字典**来获取Query argument中的参数

例如：

访问url地址为：<http://example.com/api/purchases?username=denvercoder9>

则在View中通过self.request.query\_params字典获取Query argument中的username参数

class PurchaseList(generics.ListAPIView):

serializer\_class = PurchaseSerializer

def get\_queryset(self):

"""

Optionally restricts the returned purchases to a given user,

by filtering against a `username` query parameter in the URL.

"""

queryset = Purchase.objects.all()

username = self.request.query\_params.get('username')

if username is not None:

queryset = queryset.filter(purchaser\_\_username=username)

return queryset