**# day15 drf- next part**

Contents of the previous section: Overview of front-end and back-end separation, pure projects, request objects, authentication, permissions, current limiting, versions, parsers, serializers, etc.

In this section :

- **\*\* Serializer \*\*** , **\*\* serialize \*\* the** database QuerySet or data object obtained by ORM into JSON format + **\*\* request data format verification \*\*** . (most important )

- Pagination: paginate the data obtained from ORM and return it to the user in batches .

- View, drf provides APIView + other view classes for us to inherit .

- Routing, in conjunction with views, quickly generates routing + view relationships related to addition, deletion, modification, and query .

- Conditional filtering and writing API search .

**## 1. Blog system (case )**

Develop a blog system, including: blog list, details, login, registration, **\*\* comments, likes, and blog publishing \*\*** .

**### 1.1 Table structure**

```Python

from django.db import models

class UserInfo(models.Model):

username = models.CharField(verbose\_name= " username " , max\_length= 32 , db\_index= True )

password = models.CharField(verbose\_name= " password " , max\_length= 64 )

token = models.CharField(verbose\_name= "TOKEN" , max\_length= 64 , null= True , blank= True ,db\_index= True )

class Blog(models.Model):

category\_choices = (( 1 , " Cloud Computing " ), ( 2 , "Python Full Stack " ), ( 3 , "Go Development " ))

category = models.IntegerField(verbose\_name= " Category " , choices=category\_choices)

image = models.CharField(verbose\_name= " Cover " , max\_length= 255 )

title = models.CharField(verbose\_name= " Title " , max\_length= 32 )

summary = models.CharField(verbose\_name= " Introduction " , max\_length= 256 )

text = models.TextField(verbose\_name= " Blog Post " )

ctime = models.DateTimeField(verbose\_name= " creation time " , auto\_now\_add= True )

creator = models.ForeignKey(verbose\_name= " Creator " , to= "UserInfo" , on\_delete=models.CASCADE)

comment\_count = models.PositiveIntegerField(verbose\_name= " Number of comments " , default= 0 )

favor\_count = models.PositiveIntegerField(verbose\_name= " Number of Likes " , default= 0 )

class Favor(models.Model):

    """ praise """

blog = models.ForeignKey(verbose\_name= " Blog " , to= "Blog" , on\_delete=models.CASCADE)

user = models.ForeignKey(verbose\_name= " User " , to= "UserInfo" , on\_delete=models.CASCADE)

create\_datetime = models.DateTimeField(verbose\_name= " creation time " , auto\_now\_add= True )

    class Meta:

constraints = [

models.UniqueConstraint(fields=[ 'blog' , 'user' ], name= 'uni\_favor\_blog\_user' )

]

class Comment(models.Model):

    """ Comment form """

blog = models.ForeignKey(verbose\_name= " Blog " , to= "Blog" , on\_delete=models.CASCADE)

user = models.ForeignKey(verbose\_name= " User " , to= "UserInfo" , on\_delete=models.CASCADE)

content = models.CharField(verbose\_name= " content " , max\_length= 150 )

create\_datetime = models.DateTimeField(verbose\_name= " creation time " , auto\_now\_add= True )

```

**### 1.2 Features**

- Blog List

```

  Routing + View ( in reverse chronological order) + Serialization

```

- Blog Details

```

  Routing ( PK ) + new view or original view + serialization

```

- Comments list

```

  Pass the blog ID in the URL through the GET method and get related comments based on the blog ID .

  Tip: Requests can be returned individually or in a blog-detailed request .

```

- register

```

  Enter user information + repeat password to register

```

- Log in

```

  Login is successful, generate TOKEN + expiration date, and return .

```

- Create a comment (Login required )

```

  Pass in the blog ID + comment information in the request body via GET on the URL Send to backend API

-Authentication component, request.user

-Save configuration parameters

```

- Like (Login required )

```

- I can't like it again.

- Combine transactions to implement adding likes + updating the number of likes .

```

- Create a new blog post (login required )

**## 2. Pagination**

the API for viewing data lists , if Data volume It is relatively large, so it is definitely impossible to display all the data to the user, and it needs to be displayed through paging .

In drf, we are provided with some paging related classes :

```

BasePagination , the base class

PageNumberPagination (BasePagination) supports paging in the format of /accounts/?page=4&page\_size=100

LimitOffsetPagination (BasePagination) supports paging in the format of offset=100&limit=10

```

**### 2.1 PageNumberPagination**

![ image-20210826165642846 ]( assets/image-20210826165642846.png )

![ image-20210826165918075 ]( assets/image-20210826165918075.png )

**### 2.2 LimitOffsetPagination**

![ image-20210826170617347 ]( assets/image-20210826170617347.png )

Note: limitOffset is usually combined with the previous product ID to query products with IDs smaller than this ID . Scroll down for analysis .

application :

- Blog list, using LimitOffsetPagination

- Comment list, using PageNumberPagination

**## 3. View**

**### 3.1 APIView**

- View , Django

- APIView , drf , when a request comes in, added the following functions: exemption from csrf , request encapsulation, version, authentication, permission, and current limiting .

```Python

class GenericAPIView(APIView):

    pass #10 Features

class GenericViewSet(xxxx.View - 2 functions , GenericAPIView):

    pass # 5 Functionality

class UserView(GenericViewSet):

    def get(self, request):

        pass

```

`APIView` is a drf " Top floor " The view class mainly implements the use of drf- based components, such as version, authentication, permission, current limiting, etc.

```Python

# urls.py

from django.urls import path, re\_path, include

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view()),

path( 'api/users/<int:pk>/' , views.UserDetailView.as\_view()),

]

```

```Python

# views.py

from rest\_framework.views import APIView

from rest\_framework.response import Response

class UserView(APIView):

    #Authentication , permissions, current limiting, etc.

    def get(self, request):

        #Business logic: View list

        return Response({ "code" : 0 , 'data' : "..." })

    def post(self, request):

        #Business Logic: Create a new

        return Response({ 'code' : 0 , 'data' : "..." })

class UserDetailView(APIView):

    #Authentication , permissions, current limiting, etc.

    def get(self, request, pk):

        #Business logic: View the details of a certain data

        return Response({ "code" : 0 , 'data' : "..." })

    def put(self, request,pk):

        #Business logic: All modified

        return Response({ 'code' : 0 , 'data' : "..." })

    def patch(self, request,pk):

        #Business logic: local modification

        return Response({ 'code' : 0 , 'data' : "..." })

    def delete(self, request,pk):

        #Business logic: delete

        return Response({ 'code' : 0 , 'data' : "..." })

```

**### 3.2 GenericAPIView**

`GenericAPIView` Inherits APIView and adds some functions based on APIView , such as: `get\_queryset` , `get\_object` , etc.

In actual development, it is generally not directly inherited, it is more of a The role of a ` middleman` is to provide common functionality to subclasses .

```Python

# urls.py

from django.urls import path, re\_path, include

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view()),

path( 'api/users/<int:pk>/' , views.UserDetailView.as\_view()),

]

```

```Python

# views.py

from rest\_framework.generics import GenericAPIView

from rest\_framework.response import Response

class UserView(GenericAPIView):

queryset = models.UserInfo.objects.filter(status= True )

serializer\_class = Serializer class

    def get(self, request):

queryset = self .get\_queryset()

ser = self .get\_serializer(intance=queryset,many= True )

print(ser.data)

        return Response({ "code" : 0 , 'data' : "..." })

```

Note: The greatest significance is to extract database queries and serialization classes into class variables, and then provide public get/post/put/delete methods later, so that developers only need to define class variables and automatically implement addition, deletion, modification and query .

**### 3.3 GenericViewSet**

![ image-20210824092131703 ]( assets/image-20210824092131703.png )

There is no code defined in the `GenericViewSet` class. It just inherits `ViewSetMixin` and `GenericAPIView` , that is to say, its function is to inherit the functions of the two inherited classes together .

- `GenericAPIView` extracts the definitions of database queries and serialization classes into class variables for later processing .

- `ViewSetMixin` maps get/post/put/delete and other methods to list , create , retrieve , update , partial\_update , and destroy methods, so that the view no longer needs two classes .

```Python

# urls.py

from django.urls import path, re\_path, include

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view({ "get" : "list" , "post" : "create" })),

path( 'api/users/<int:pk>/' , views.UserView.as\_view({ "get" : "retrieve" , "put" : "update" , "patch" : "partial\_update" , "delete" : "destory" })),

]

```

```Python

# views.py

from rest\_framework.viewsets import GenericViewSet

from rest\_framework.response import Response

class UserView(GenericViewSet):

    #Authentication , permissions, current limiting, etc.

queryset = models.UserInfo.objects.filter(status= True )

serializer\_class = Serializer class

    def list(self, request):

        #Business logic: View list

queryset = self .get\_queryset()

ser = self .get\_serializer(intance=queryset,many= True )

print(ser.data)

        return Response({ "code" : 0 , 'data' : "..." })

    def create(self, request):

        #Business Logic: Create a new

        return Response({ 'code' : 0 , 'data' : "..." })

    def retrieve(self, request,pk):

        #Business logic: View the details of a certain data

        return Response({ "code" : 0 , 'data' : "..." })

    def update(self, request,pk):

        #Business logic: All modified

        return Response({ 'code' : 0 , 'data' : "..." })

    def partial\_update(self, request,pk):

        #Business logic: local modification

        return Response({ 'code' : 0 , 'data' : "..." })

    def destroy(self, request,pk):

        #Business logic: delete

        return Response({ 'code' : 0 , 'data' : "..." })

```

Note: In development, it is generally rare to inherit it directly, because it also belongs to The ` middleman` class, in the original `GenericAPIView` It just adds another mapping on the basis .

**### 3.4 Five categories**

In drf, we have provided 5 Add, delete, modify (including partial modification), check list, check single data ( need to combine `GenericViewSet` use) .

```Python

# urls.py

from django.urls import path, re\_path, include

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view({ "get" : "list" , "post" : "create" })),

path( 'api/users/<int:pk>/' , views.UserView.as\_view({ "get" : "retrieve" , "put" : "update" , "patch" : "partial\_update" , "delete" : "destroy" })),

]

```

```Python

# views.py

from rest\_framework.viewsets import GenericViewSet

from rest\_framework.mixins import (

ListModelMixin, CreateModelMixin, RetrieveModelMixin, UpdateModelMixin,

DestroyModelMixin, ListModelMixin

)

class UserView(CreateModelMixin,RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin,ListModelMixin,GenericViewSet):

    #Authentication , permissions, current limiting, etc.

queryset = models.UserInfo.objects.filter(status= True )

serializer\_class = Serializer class

```

It has been written for us in these 5 classes `list` , `create` , `retrieve` , `update` , `partial\_update` , `destory` method, we only need to write Class variables : queryset , serializer\_class .

**\*\* Example 1 : \*\***

![ image-20210824230441249 ]( assets/image-20210824230441249.png )

```Python

# urls.py

from django.urls import path

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view({ "get" : "list" })),

path( 'api/users/<int:pk>/' , views.UserView.as\_view({ "get" : "retrieve" })),

]

```

```Python

# views.py

from rest\_framework import serializers

from rest\_framework.viewsets import GenericViewSet

from rest\_framework import mixins

from app01 import models

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

    class Meta:

model = models.UserInfo

fields = [ "username" , "age" , "email" , "level\_text" , "extra" ]

    def get\_extra(self, obj):

        return 666

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

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

```

**\*\* Example 2 : \*\***

![ image-20210824231043061 ]( assets/image-20210824231043061.png )

```Python

# urls.py

from django.urls import path

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view({ "get" : "list" , "post" : "create" })),

path( 'api/users/<int:pk>/' , views.UserView.as\_view({ "get" : "retrieve" })),

]

```

```Python

# views.py

from rest\_framework import serializers

from rest\_framework.viewsets import GenericViewSet

from rest\_framework import mixins

from app01 import models

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

    class Meta:

model = models.UserInfo

fields = [ "username" , "age" , "email" , "level\_text" , "extra" ]

    def get\_extra(self, obj):

        return 666

class UserView(mixins.ListModelMixin, mixins.RetrieveModelMixin, mixins.CreateModelMixin, GenericViewSet):

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

    def perform\_create(self, serializer):

        """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

```

**\*\* Example 3 : \*\***

```Python

# urls.py

from django.urls import path

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view(

{ "get" : "list" , "post" : "create" }

)),

path( 'api/users/<int:pk>/' , views.UserView.as\_view(

{ "get" : "retrieve" , "put" : "update" , "patch" : "partial\_update" , "delete" : "destroy" }

)),

]

```

```Python

# views.py

from rest\_framework import serializers

from rest\_framework.viewsets import GenericViewSet

from rest\_framework import mixins

from app01 import models

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

    class Meta:

model = models.UserInfo

fields = [ "username" , "age" , "email" , "level\_text" , "extra" ]

    def get\_extra(self, obj):

        return 666

class UserView(mixins.ListModelMixin,

mixins.RetrieveModelMixin,

mixins.CreateModelMixin,

mixins.UpdateModelMixin,

mixins.DestroyModelMixin,

GenericViewSet):

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

    def perform\_create(self, serializer):

        """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

    def perform\_update(self, serializer):

serializer.save()

    def perform\_destroy(self, instance):

instance.delete()

```

**\*\* Example 4 : \*\***

```Python

# urls.py

from django.urls import path

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view(

{ "get" : "list" , "post" : "create" }

)),

path( 'api/users/<int:pk>/' , views.UserView.as\_view(

{ "get" : "retrieve" , "put" : "update" , "patch" : "partial\_update" , "delete" : "destroy" }

)),

]

```

```Python

# views.py

from rest\_framework import serializers

from rest\_framework.viewsets import ModelViewSet

from app01 import models

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

    class Meta:

model = models.UserInfo

fields = [ "username" , "age" , "email" , "level\_text" , "extra" ]

    def get\_extra(self, obj):

        return 666

class UserView(ModelViewSet):

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

    def perform\_create(self, serializer):

        """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

```

Use during development Five categories​ or `ModelViewSet` It is quite common, and if some of their internal functions are not enough, they can be expanded by re-using some methods .

Question: There are so many views provided in drf , which one will be used more often in the future ?

- The interface has nothing to do with database operations and directly inherits APIView

- The database needs to be operated behind the interface, generally: `ModelViewSet` or `CreateModelMixin , ListModelMixin...`

```

- Utilize hooks to customize functionality .

-Rewrite a certain write method to achieve more complete functions .

```

- Customize according to your company's habits : `ModelViewSet` or `CreateModelMixin , ListModelMixin...`

**### 3.5 Additional**

```Python

from rest\_framework.viewsets import ModelViewSet

from rest\_framework.decorators import action

class XXXModelSerializer(serializers.ModelSerializer):

    class Meta:

model = models.UserInfo

fields = "\_\_all\_\_"

class XXXView(ModelViewSet):

queryset = models.UserInfo.objects.all()

serializer\_class = XXXModelSerializer

    # @action(detail=False, methods=['get'], url\_path="yyy/(?P<xx>\d+)/xxx")

    # def get\_password(self, request, xx, pk=None):

    # print(xx)

    # return Response("...")

    # @action(detail=True, methods=['get'], url\_path="yyy/(?P<xx>\d+)/xxx")

    # def set\_password(self, request, xx, pk=None):

    # print(xx)

    # return Response("...")

```

**### Supplement: Permissions**

When defining a permission class previously, two methods can be defined in the class: `has\_permission` and `has\_object\_permission`

- `has\_permission` , which is executed before the request enters the view .

- `has\_object\_permission` , when called in a view `self.get\_object` is called when deleting, updating, or viewing an object. It is generally used to check whether you have permission to operate on an object .

```Python

classPermissionA (BasePermission):

message = { "code" : 1003 , 'data' : " No access " }

    def has\_permission(self, request, view):

exists = request.user.roles.filter(title= " Employee " ).exists()

        if exists:

            return True

        return False

    def has\_object\_permission(self, request, view, obj):

        return True

```

So, when we write a view class, if we directly obtain the indirectly inherited GenericAPIView , and call it internally `get\_object` method, so that it is passed in the permission `has\_object\_permission` You can then process permissions .

**## 4. Routing**

drf before , we generally configure routing in two ways :

- View inherits APIView

```Python

  from django.urls import path

  from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view()),

]

```

- View inheritance `ViewSetMixin` ( GenericViewSet , ModelViewSet )

```Python

  from django.urls import path, re\_path, include

  from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view({ "get" : "list" , "post" : "create" })),

path( 'api/users/<int:pk>/' , views.UserView.as\_view({ "get" : "retrieve" , "put" : "update" , "patch" : "partial\_update" , "delete" : "destory" })),

]

```

  For this form of routing, drf provides a simpler way :

```Python

  from rest\_framework import routers

  from app01 import views

router = routers.SimpleRouter()

router.register( r 'api/users' , views.UserView)

urlpatterns = [

      #Other URLs​

      # path('xxxx/', xxxx.as\_view()),

]

urlpatterns += router.urls

```

  You can also use include to add a prefix to the URL :

```Python

  from django.urls import path, include

  from rest\_framework import routers

  from app01 import views

router = routers.SimpleRouter()

router.register( r 'users' , views.UserView)

urlpatterns = [

path( 'api/' , include((router.urls, 'app\_name' ), namespace= 'instance\_name' )),

      #Other URLs​

      # path('forgot-password/', ForgotPasswordFormView.as\_view()),

]

```

**## 5. Conditional search**

If an API needs to pass some conditions for search, you can just pass the parameters via GET after the URL , for example :

```

/api/users?age=19&category=12

```

in drf that can support conditional search .

**### 5.1 Custom Filter**

![ image-20210825200814769 ]( assets/image-20210825200814769.png )

```Python

# urls.py

from django.urls import path

from app01 import views

urlpatterns = [

path( 'api/users/' , views.UserView.as\_view(

{ "get" : "list" , "post" : "create" }

)),

path( 'api/users/<int:pk>/' , views.UserView.as\_view(

{ "get" : "retrieve" , "put" : "update" , "patch" : "partial\_update" , "delete" : "destroy" }

)),

]

```

```Python

# views.py

from rest\_framework import serializers

from rest\_framework.viewsets import ModelViewSet

from rest\_framework.filters import BaseFilterBackend

from app01 import models

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

    class Meta:

model = models.UserInfo

fields = [ "username" , "age" , "email" , "level\_text" , "extra" ]

    def get\_extra(self, obj):

        return 666

class Filter1(BaseFilterBackend):

    def filter\_queryset(self, request, queryset, view):

age = request.query\_params.get( 'age' )

        if not age:

            return queryset

        return queryset.filter(age=age)

class Filter2(BaseFilterBackend):

    def filter\_queryset(self, request, queryset, view):

user\_id = request.query\_params.get( 'id' )

        if not user\_id:

            return queryset

        return queryset.filter(id\_\_gt=user\_id)

class UserView(ModelViewSet):

filter\_backends = [Filter1, Filter2]

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

    def perform\_create(self, serializer):

        """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

```

**### 5.2 Third-party Filter**

There is a commonly used third-party filter in drf development : DjangoFilterBackend .

```

pip install django-filter

```

Register app :

```Python

INSTALLED\_APPS = [

...

    'django\_filters' ,

...

]

```

View configuration and application (example 1 ) :

```Python

# views.py

from rest\_framework import serializers

from rest\_framework.viewsets import ModelViewSet

from django\_filters.rest\_framework import DjangoFilterBackend

from app01 import models

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

    class Meta:

model = models.UserInfo

fields = [ "username" , "age" , "email" , "level\_text" , "extra" ]

    def get\_extra(self, obj):

        return 666

class UserView(ModelViewSet):

filter\_backends = [DjangoFilterBackend, ]

filterset\_fields = [ "id" , "age" , "email" ]

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

    def perform\_create(self, serializer):

        """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

```

View configuration and application (example 2 ) :

```Python

from rest\_framework import serializers

from rest\_framework.viewsets import ModelViewSet

from django\_filters.rest\_framework import DjangoFilterBackend

from django\_filters import FilterSet, filters

from app01 import models

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

depart\_title = serializers.CharField(

source = "depart.title" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

    class Meta:

model = models.UserInfo

fields = [ "id" , "username" , "age" , "email" , "level\_text" , "extra" , "depart\_title" ]

    def get\_extra(self, obj):

        return 666

class MyFilterSet(FilterSet):

depart = filters.CharFilter(field\_name= "depart\_\_title" , lookup\_expr= "exact" )

min\_id = filters.NumberFilter(field\_name= 'id' , lookup\_expr= 'gte' )

    class Meta:

model = models.UserInfo

fields = [ "min\_id" , "depart" ]

class UserView(ModelViewSet):

filter\_backends = [DjangoFilterBackend, ]

filterset\_class = MyFilterSet

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

    def perform\_create(self, serializer):

        """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

```

View configuration and application (Example 3 ) :

```Python

from rest\_framework import serializers

from rest\_framework.viewsets import ModelViewSet

from django\_filters.rest\_framework import DjangoFilterBackend, OrderingFilter

from django\_filters import FilterSet, filters

from app01 import models

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

depart\_title = serializers.CharField(

source = "depart.title" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

    class Meta:

model = models.UserInfo

fields = [ "id" , "username" , "age" , "email" , "level\_text" , "extra" , "depart\_title" ]

    def get\_extra(self, obj):

        return 666

class MyFilterSet(FilterSet):

    # /api/users/?min\_id=2 -> id>=2

min\_id = filters.NumberFilter(field\_name= 'id' , lookup\_expr= 'gte' )

    # /api/users/?name=wupeiqi -> not ( username=wupeiqi )

name = filters.CharFilter(field\_name= "username" , lookup\_expr= "exact" , exclude= True )

    # /api/users/?depart=xx -> depart\_\_title like %xx%

depart = filters.CharFilter(field\_name= "depart\_\_title" , lookup\_expr= "contains" )

    # /api/users/?token=true -> "token" IS NULL

    # /api/users/?token=false -> "token" IS NOT NULL

token = filters.BooleanFilter(field\_name= "token" , lookup\_expr= "isnull" )

    # /api/users/?email=xx -> email like xx%

email = filters.CharFilter(field\_name= "email" , lookup\_expr= "startswith" )

    # /api/users/?level=2&level=1 -> "level" = 1 OR "level" = 2 (The data must exist, otherwise an error will be reported --> there is an internal verification mechanism )

    # level = filters.AllValuesMultipleFilter(field\_name="level", lookup\_expr="exact")

level = filters.MultipleChoiceFilter(field\_name= "level" , lookup\_expr= "exact" , choices=models.UserInfo.level\_choices)

    # /api/users/?age=18,20 -> age in [18,20]

age = filters.BaseInFilter(field\_name= 'age' , lookup\_expr= "in" )

    # /api/users/?range\_id\_max=10&range\_id\_min=1 -> id BETWEEN 1 AND 10

range\_id = filters.NumericRangeFilter(field\_name= 'id' , lookup\_expr= 'range' )

    # /api/users/?ordering=id -> order by id asc

    # /api/users/?ordering=-id -> order by id desc

    # /api/users/?ordering=age -> order by age asc

    # /api/users/?ordering=-age -> order by age desc

ordering = filters.OrderingFilter(fields=[ "id" , "age" ])

    # /api/users/?size=1 -> limit 1 (custom search )

size = filters.CharFilter(method= 'filter\_size' , distinct= False , required= False )

    class Meta:

model = models.UserInfo

fields = [ "id" , "min\_id" , "name" , "depart" , "email" , "level" , "age" , 'range\_id' , "size" , "ordering" ]

    def filter\_size(self, queryset, name, value):

int\_value = int(value)

        return queryset[ 0 :int\_value]

class UserView(ModelViewSet):

filter\_backends = [DjangoFilterBackend, ]

filterset\_class = MyFilterSet

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

    def perform\_create(self, serializer):

        """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

```

There are many common choices for `lookup\_expr` :

```Python

'exact' : \_( '' ),

'iexact' : \_( '' ),

'contains' : \_( 'contains' ),

'icontains' : \_( 'contains' ),

'startswith' : \_( 'starts with' ),

'istartswith' : \_( 'starts with' ),

'endswith' : \_( 'ends with' ),

'iendswith' : \_( 'ends with' ),

'gt' : \_( 'is greater than' ),

'gte' : \_( 'is greater than or equal to' ),

'lt' : \_( 'is less than' ),

'lte' : \_( 'is less than or equal to' ),

'in' : \_( 'is in' ),

'range' : \_( 'is in range' ),

'isnull' : \_( '' ),

'regex' : \_( 'matches regex' ),

'iregex' : \_( 'matches regex' ),

```

Global configuration and application :

```Python

# settings.py global configuration

REST\_FRAMEWORK = {

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

}

```

**### 5.3 Built-in Filter**

There are two filters built into the drf source code :

- OrderingFilter , which supports sorting .

```Python

  from rest\_framework import serializers

  from rest\_framework.viewsets import ModelViewSet

  from app01 import models

  from rest\_framework.filters import OrderingFilter

  class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

depart\_title = serializers.CharField(

source = "depart.title" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

      class Meta:

model = models.UserInfo

fields = [ "id" , "username" , "age" , "email" , "level\_text" , "extra" , "depart\_title" ]

      def get\_extra(self, obj):

          return 666

  class UserView(ModelViewSet):

filter\_backends = [OrderingFilter, ]

      # ?order=id

      # ?order=-id

      # ?order=age

ordering\_fields = [ "id" , "age" ]

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

      def perform\_create(self, serializer):

          """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

```

- SearchFilter , supports fuzzy search .

```Python

  from rest\_framework import serializers

  from rest\_framework.viewsets import ModelViewSet

  from app01 import models

  from rest\_framework.filters import SearchFilter

  class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(

source = "get\_level\_display" ,

read\_only= True

)

depart\_title = serializers.CharField(

source = "depart.title" ,

read\_only= True

)

extra = serializers.SerializerMethodField(read\_only= True )

      class Meta:

model = models.UserInfo

fields = [ "id" , "username" , "age" , "email" , "level\_text" , "extra" , "depart\_title" ]

      def get\_extra(self, obj):

          return 666

  class UserView(ModelViewSet):

      # ?search=武沛%齐

filter\_backends = [SearchFilter, ]

search\_fields = [ "id" , "username" , "age" ]

queryset = models.UserInfo.objects.all()

serializer\_class = UserModelSerializer

      def perform\_create(self, serializer):

          """ Serialization: After the requested data is successfully verified, save it. """

serializer.save(depart\_id= 1 , password= "123" )

```

```Python

  "app01\_userinfo" . "id" LIKE %武 Peiqi % ESCAPE '\'

OR

  "app01\_userinfo" . "username" LIKE %武 Peiqi % ESCAPE '\'

OR

  "app01\_userinfo" . "age" LIKE %武 Peiqi % ESCAPE '\'

```