**# day14 drf- middle**

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

In this section :

- **\*\* Version \*\*** , carry the version number in the request to facilitate subsequent API updates and iterations .

```

http://www.5xclass/api/v1/info

http://www.5xclass/api/v2/info

```

- **\*\* Parser \*\*** reads data in different formats, parses them and assigns them to objects such as request.data .

```

user=wupeiqi&age=123

{"user":"wupeiqi","age":123}

```

- **\*\* 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 .

**## 1. Version**

In the RESTful specification, the backend API needs to reflect the version .

**### 1.1 GET parameter passing**

![ image-20210819154455680 ]( assets/image-20210819154455680.png )

```Python

# settings.py

REST\_FRAMEWORK = {

    "VERSION\_PARAM" : "v" ,

    "DEFAULT\_VERSION" : "v1" ,

    "ALLOWED\_VERSIONS" : [ "v1" , "v2" , "v3" ],

    "DEFAULT\_VERSIONING\_CLASS" : "rest\_framework.versioning.QueryParameterVersioning"

}

```

Source code execution process :

![ image-20210820105543193 ]( assets/image-20210820105543193.png )

**### 1.2 URL path passing ( \* )**

![ image-20210819154955480 ]( assets/image-20210819154955480.png )

**### 1.3 Request header transmission**

![ image-20210819155617845 ]( assets/image-20210819155617845.png )

**### 1.4 Reverse URL generation**

`reverse` method is also defined in each version processing class , which is used to reverse generate URLs and carry relevant version information, for example :

![ image-20210820105543193 ]( assets/image-20210820105543193-3386187.png )

![ image-20210820112152615 ]( assets/image-20210820112152615.png )

**## 2. Parser**

use `request.data` Get the data in the request body .

this `reqeust.data` How does the data come from? In fact, the parser inside DRF processes it according to the data format + request header passed in by the requester .

**### 2.1 JSONParser ( \* )**

![ image-20210827081058194 ]( assets/image-20210827081058194.png )

**### 2.2 FormParser**

![ image-20210827081244795 ]( assets/image-20210827081244795.png )

**### 2.3 MultiPartParser ( \* )**

![ image-20210827083047327 ]( assets/image-20210827083047327.png )

```html

<!DOCTYPE html >

<html lang = "en" >

<head>

    <meta charset = "UTF-8" >

    <title> Title </title>

</head>

<body>

<form action = "http://127.0.0.1:8000/test/" method = "post" enctype = "multipart/form-data" >

    <input type = "text" name = "user" />

    <input type = "file" name = "img" >

    <input type = "submit" value = " Submit " >

</form>

</body>

</html>

```

**### 2.4 FileUploadParser ( \* )**

![ image-20210827084403453 ]( assets/image-20210827084403453.png )

You can set multiple parsers, the default parser is :

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework.parsers import MultiPartParser, JSONParser, FormParser

class UserView(APIView):

    def post(self, request):

print(request.content\_type)

print(request.data)

        return Response( "..." )

```

**## 3. Serializer**

**### 3.1 Serialized Data**

**#### 3.1.1 Serializer**

```Python

from django.db import models

class Role(models.Model):

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

order = models.IntegerField(verbose\_name= " order " )

```

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from api import models

class InfoSerializer(serializers.Serializer):

id = serializers.IntegerField()

title = serializers.CharField()

order = serializers.IntegerField()

class InfoView(APIView):

    def get(self, request):

        # 1. Get multiple data from the database

        # queryset = models.Role.objects.all()

        # ser = InfoSerializer(instance=queryset, many=True)

        # 2. Get a single piece of data from the database

instance = models.Role.objects.all().first()

ser = InfoSerializer(instance=instance, many= False )

print(type(ser.data), ser.data)

        return Response(ser.data)

```

**#### 3.1.2 ModelSerializer**

```Python

from django.db import models

class Role(models.Model):

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

order = models.IntegerField(verbose\_name= " order " )

```

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from api import models

class InfoSerializer(serializers.ModelSerializer):

    class Meta:

model = models.Role

        # fields = "\_\_all\_\_"

        # fields = ['id', 'title', 'order']

exclude = [ "id" ]

class InfoView(APIView):

    def get(self, request):

        # 1. Get multiple records from the database

        # queryset = models.Role.objects.all()

        # ser = InfoSerializer(instance=queryset, many=True)

        # 2. Get a single piece of data from the database

instance = models.Role.objects.all().first()

ser = InfoSerializer(instance=instance, many= False )

print(type(ser.data), ser.data)

        return Response(ser.data)

```

Obviously, if you want to serialize the fields in the data table, using ModelModelSerializer is more concise than Serializer .

**#### 3.1.3 Fields and Parameters**

In both `ModelModelSerializer` and `Serializer`, you can customize fields and pass in some related parameters .

```Python

from django.db import models

class Role(models.Model):

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

order = models.IntegerField(verbose\_name= " order " )

class UserInfo(models.Model):

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

gender = models.SmallIntegerField(verbose\_name= " gender " , choices=(( 1 , " male " ), ( 2 , " female " )))

role = models.ForeignKey(verbose\_name= " role " , to= "Role" , on\_delete=models.CASCADE)

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

```

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from api import models

class InfoSerializer(serializers.ModelSerializer):

gender = serializers.CharField(source= "get\_gender\_display" )

role = serializers.CharField(source= "role.title" )

ctime = serializers.DateTimeField(format= "%Y-%m-%d" )

other\_name = serializers.CharField(source= "name" )

mine = serializers.SerializerMethodField()

    class Meta:

model = models.UserInfo

fields = [ 'id' , 'name' , 'gender' , "role" , 'ctime' , "other\_name" , "mine" ]

    def get\_mine(self, obj):

        return "xx-{}" .format(obj.name)

class InfoView(APIView):

    def get(self, request):

queryset = models.UserInfo.objects.all()

ser = InfoSerializer(instance=queryset, many= True )

print(type(ser.data), ser.data)

        return Response(ser.data)

```

![ image-20220917161121853 ]( assets/image-20220917161121853.png )

**#### 3.1.4 Serialization class nesting**

Mainly corresponding to `ForeignKey` in ORM class and `ManyToManyField` fields are serialized .

- Serialize the associated table data based on the `SerializerMethodField` custom method

- Based on nested serialization class implementation

![ image-20220917161811985 ]( assets/image-20220917161811985.png )

```Python

from django.db import models

class Role(models.Model):

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

order = models.IntegerField(verbose\_name= " order " )

class Tag(models.Model):

caption = models.CharField(verbose\_name= " name " , max\_length= 32 )

class UserInfo(models.Model):

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

gender = models.SmallIntegerField(verbose\_name= " gender " , choices=(( 1 , " male " ), ( 2 , " female " )))

role = models.ForeignKey(verbose\_name= " role " , to= "Role" , on\_delete=models.CASCADE)

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

tags = models.ManyToManyField(verbose\_name= " Tag " , to= "Tag" )

```

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from api import models

class RoleSerializer(serializers.ModelSerializer):

    class Meta:

model = models.Role

        # fields = "\_\_all\_\_"

fields = [ "id" , 'title' ]

class TagSerializer(serializers.ModelSerializer):

    class Meta:

model = models.Tag

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

class InfoSerializer(serializers.ModelSerializer):

role = RoleSerializer()

tags = TagSerializer(many= True )

    class Meta:

model = models.UserInfo

fields = [ 'id' , 'name' , "role" , "tags" ]

class InfoView(APIView):

    def get(self, request):

queryset = models.UserInfo.objects.all()

ser = InfoSerializer(instance=queryset, many= True )

print(type(ser.data), ser.data)

        return Response(ser.data)

```

**#### 3.1.5 Serialization class inheritance**

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from api import models

class MySerializer(serializers.Serializer):

more = serializers.SerializerMethodField()

    def get\_more(self, obj):

        return "123"

class InfoSerializer(serializers.ModelSerializer, MySerializer):

    class Meta:

model = models.UserInfo

fields = [ "id" , "name" , 'more' ]

class InfoView(APIView):

    def get(self, request):

instance = models.UserInfo.objects.all().first()

ser = InfoSerializer(instance=instance, many= False )

print(type(ser.data), ser.data)

        return Response(ser.data)

```

**#### 3.1.6 Bottom-level implementation principle (extension )**

Disclaimer: Mastering the above knowledge points will enable you to complete common tasks at work. The following knowledge points are just for expansion and can be skipped .

**##### 1. Metaclass**

Objects are instantiated from classes .

```Python

class Foo(object):

    pass

#Step 1 : Call Foo 's \_\_new\_\_ method to create an empty object .

#Step 2 : Call Foo 's \_\_init\_\_ method to initialize the object .

obj = Foo()

```

Who created the class? It was created by type (default) .

```Python

class Foo(object):

v1 = 123

    def func(self):

        return 666

```

```Python

Foo = type( "Foo" ,(object,),{ "v1" : 123 , "func" : lambda self: 666 })

```

When defining a class, add metaclass to specify the creator of the current class .

```Python

# Create Foo type from type

class Foo(object):

    pass

```

```Python

# Create Foo type from ` stuff`​

class Foo(object, metaclass= stuff ):

    pass

```

Specify the metaclass to create the class .

```Python

class MyType(type):

    def \_\_new\_\_(cls, \*args, \*\*kwargs):

new\_cls = super().\_\_new\_\_( cls , \*args, \*\*kwargs)

print( " Create class: " , new\_cls)

        return new\_cls

class Foo(metaclass=MyType):

    pass

```

```Python

class MyType(type):

    def \_\_init\_\_(self, \*args, \*\*kwargs):

print( " Step 2 : Initialize class members: " , args, \*\*kwargs)

super().\_\_init\_\_(\*args, \*\*kwargs)

    def \_\_new\_\_(cls, \*args, \*\*kwargs):

new\_cls = super().\_\_new\_\_( cls , \*args, \*\*kwargs)

print( " Step 1 : Create class: " , new\_cls)

        return new\_cls

class Foo(metaclass=MyType):

v1 = 123

    def func(self):

        pass

```

```Python

class MyType(type):

    def \_\_init\_\_(cls, \*args, \*\*kwargs):

print( " Step 2 : Initialize class members: " , args, \*\*kwargs)

super().\_\_init\_\_(\*args, \*\*kwargs)

    def \_\_new\_\_(cls, \*args, \*\*kwargs):

new\_cls = super().\_\_new\_\_( cls , \*args, \*\*kwargs)

print( " Step 1 : Create class: " , new\_cls)

        return new\_cls

    def \_\_call\_\_(cls, \*args, \*\*kwargs):

print( " Step 3 : Create objects & initialize objects " , cls )

        # 1. Call the \_\_new\_\_ method of your own class to create an object

new\_object = cls .\_\_new\_\_( cls , \*args, \*\*kwargs)

        # 2. Call your own class \_\_init\_\_ to initialize

        cls .\_\_init\_\_(new\_object, \*args, \*\*kwargs)

        return new\_object

class Foo(metaclass=MyType):

v1 = 123

    def func(self):

        pass

obj = Foo()

```

**##### 2. Instantiate field objects**

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from api import models

class InfoSerializer(serializers.Serializer):

id = serializers.IntegerField()

title = serializers.CharField()

order = serializers.IntegerField

```

For the above code, before the class `InfoSerializer` is created, its internal `id , title , order` fields will be instantiated first .

The inheritance relationship of these fields such as `IntegerField` and `CharField` is as follows :

```Python

class Field:

\_creation\_counter = 0

class IntegerField(Field):

    pass

class CharField(Field):

    pass

class DateTimeField(Field):

    pass

```

When fields such as `IntegerField` and `CharField` are instantiated, a counter is maintained internally to indicate the order of instantiation .

```Python

class Field:

\_creation\_counter = 0

    def \_\_init\_\_(self, \*, read\_only= False ...):

        self .\_creation\_counter = Field.\_creation\_counter

Field.\_creation\_counter += 1

class IntegerField(Field):

    def \_\_init\_\_(self, \*\*kwargs):

...

super().\_\_init\_\_(\*\*kwargs)

class CharField(Field):

    def \_\_init\_\_(self, \*\*kwargs):

...

super().\_\_init\_\_(\*\*kwargs)

```

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from api import models

class InfoSerializer(serializers.Serializer):

id = serializers.IntegerField() #object , internal\_creation\_counter =0

title = serializers.CharField() #object , internal\_creation\_counter =1

order = serializers.IntegerField #object , internal\_creation\_counter =2

```

Note: This counter will be used for sorting later to implement the order of execution of the fields .

**##### 3. Creation of serialization class**

```Python

class SerializerMetaclass(type):

    def \_\_new\_\_(cls, name, bases, attrs):

attrs[ '\_declared\_fields' ] = cls .\_get\_declared\_fields(bases, attrs)

        return super().\_\_new\_\_( cls , name, bases, attrs)

```

```Python

class Serializer(BaseSerializer, metaclass=SerializerMetaclass):

...

class ModelSerializer(Serializer):

...

class RoleSerializer(serializers.ModelSerializer):

gender = serializers.CharField(source= "get\_gender\_display" )

    class Meta:

model = models.Role

fields = [ "id" , 'title' , "gender" ]

```

metaclass in the parent class , the subclass will also create classes based on this metaclass .

**##### 4. \_declared\_fields**

Before creating the class, the metaclass's \_\_new\_\_ method adds a \_declared\_fields (class variable) to the class members .

```Python

class SerializerMetaclass(type):

@classmethod

    def \_get\_declared\_fields(cls, bases, attrs):

        # 1. Loop through all members (class variables, methods) defined in the class and filter out field objects of classes that inherit from Fields .

        #Note : The fields will also be removed from the current class members

fields = [

(field\_name, attrs.pop(field\_name))

            for field\_name, obj in list(attrs.items())

            if isinstance(obj, Field)

]

        # 2. Sort by the \_creation\_counter of the field

fields.sort(key= lambda x: x[ 1 ].\_creation\_counter)

        # Ensures a base class field doesn't override cls attrs, and maintains

        # field precedence when inheriting multiple parents. eg if there is a

        # class C(A, B), and A and B both define 'field', use 'field' from A.

known = set(attrs)

        def visit(name):

known.add(name)

            return name

        # 3. Read the \_declared\_fields field in the parent class (the parent class is created before the subclass, and the serialization class supports inheritance )

base\_fields = [

(visit(name), f)

            for base in bases if hasattr(base, '\_declared\_fields' )

            for name, f in base.\_declared\_fields.items() if name not in known

]

        # 4. Package and return the fields in the parent class and child class, and assign them to the \_declared\_fields of the current class

        return OrderedDict(base\_fields + fields)

    def \_\_new\_\_(cls, name, bases, attrs):

attrs[ '\_declared\_fields' ] = cls .\_get\_declared\_fields(bases, attrs)

        return super().\_\_new\_\_( cls , name, bases, attrs)

```

```Python

class Serializer(BaseSerializer, metaclass=SerializerMetaclass):

...

class ModelSerializer(Serializer):

...

class RoleSerializer(serializers.ModelSerializer):

gender = serializers.CharField(source= "get\_gender\_display" )

    class Meta:

model = models.Role

fields = [ "id" , 'title' , "gender" ]

```

Therefore, when the class serialization class is loaded, the members in the class :

- Eliminate, field object .

```Python

RoleSerializer.gender does not exist

```

- New, \_declared\_fields , which is of type OrderedDict and contains all fields .

```Python

RoleSerializer.\_declared\_fields = {

      "gender" : CharField object

}

```

- Others, keep as they are .

```Python

RoleSerializer.Meta

```

**##### 5. Create a serialized class object**

In the view method, when using the serialization class to serialize the QuerySet or object obtained by orm , you need to initialize the class object first .

```Python

class SerializerMetaclass(type):

    def \_\_new\_\_(cls, name, bases, attrs):

attrs[ '\_declared\_fields' ] = cls .\_get\_declared\_fields(bases, attrs)

        return super().\_\_new\_\_( cls , name, bases, attrs)

```

```Python

class BaseSerializer(Field):

    def \_\_init\_\_(self, instance= None , data=empty, \*\*kwargs):

        self .instance = instance

        If data is not empty:

            self .initial\_data = data

        self .partial = kwargs.pop( 'partial' , False )

        self .\_context = kwargs.pop( 'context' , {})

kwargs.pop( 'many' , None )

super().\_\_init\_\_(\*\*kwargs)

    def \_\_new\_\_(cls, \*args, \*\*kwargs):

        if kwargs.pop( 'many' , False ):

            #Call many\_init method to get other objects and return

            return cls .many\_init(\*args, \*\*kwargs)

        #Create an empty object of the current class and return

        return super().\_\_new\_\_( cls , \*args, \*\*kwargs)

@classmethod

    def many\_init(cls, \*args, \*\*kwargs):

...

child\_serializer = cls (\*args, \*\*kwargs)

list\_kwargs = {

            'child' : child\_serializer,

}

meta = getattr( cls , 'Meta' , None )

list\_serializer\_class = getattr(meta, 'list\_serializer\_class' , ListSerializer)

        return list\_serializer\_class(\*args, \*\*list\_kwargs)

class Serializer(BaseSerializer, metaclass=SerializerMetaclass):

...

class ModelSerializer(Serializer):

...

class RoleSerializer(serializers.ModelSerializer):

gender = serializers.CharField(source= "get\_gender\_display" )

    class Meta:

model = models.Role

fields = [ "id" , 'title' , "gender" ]

```

```Python

instance = models.UserInfo.objects.all().first()

#Instantiate the object, internally: first execute \_\_new\_\_ , then execute \_\_init\_\_

#Step 1 : \_\_new\_\_​

#Default : many=True , returns ListSerializer object; many=False , returns the object of the current class InfoSerializer .

#Step 2 : \_\_init\_\_​

#Here , we need to execute the \_\_init\_\_ methods of different objects according to the different objects returned by \_\_new\_\_ .

# =====> Question: Why do you think he designed it this way? <======

ser = InfoSerializer(instance=instance, many= False )

#Get the serialized value

ser.data

```

**##### 6. Serialization - Current Class**

```Python

class Field:

    def get\_attribute(self, instance):

        # source\_attrs=[] or source\_attrs=["xx","xx","xxx"]

        return get\_attribute(instance, self .source\_attrs)

class CharField(Field):

    def to\_representation(self, value):

        return str(value)

```

```Python

class BaseSerializer(Field):

@property

    def data(self):

        #Step 2​​

        if not hasattr( self , '\_data' ):

            if self .instance is not None and not getattr( self , '\_errors' , None ):

                #Step 3 : Used to serialize and initialize the object .

                self .\_data = self .to\_representation( self .instance)

            elif hasattr( self , '\_validated\_data' ) and not getattr( self , '\_errors' , None ):

                #This is used to trigger execution when the request is verified .

                self .\_data = self .to\_representation( self .validated\_data)

            else :

                #This is used to pass the initial\_data parameter to the Serializer instead of the object .

                self .\_data = self .get\_initial()

        return self .\_data

class Serializer(BaseSerializer, metaclass=SerializerMetaclass):

@property

    def data(self):

        #Step 1​​

ret = super().data

        return ReturnDict(ret, serializer= self )

    def to\_representation(self, instance):

        #Step 4​​

ret = OrderedDict()

        #Step 5 : Get all non- write\_only fields in \_declared\_fields , that is, fields used for serialization .

        #If it is ModelSerializer , it will also look for the fields defined in its Meta + the bind method of the fields

fields = self .\_readable\_fields

        for field in fields:

            try :

                #Step 5 : Call the get\_attribute method in the field object

attribute = field.get\_attribute(instance)

            except SkipField:

                continue

check\_for\_none = attribute.pk if isinstance(attribute, PKOnlyObject) else attribute

            if check\_for\_none is None :

ret[field.field\_name] = None

            else :

                #Step 6 : Call the to\_representation method in the field object

ret[field.field\_name] = field.to\_representation(attribute)

        return ret

class ModelSerializer(Serializer):

...

class RoleSerializer(serializers.ModelSerializer):

gender = serializers.CharField(source= "get\_gender\_display" )

    class Meta:

model = models.Role

fields = [ "id" , 'title' , "gender" ]

```

```Python

instance = models.UserInfo.objects.all().first()

ser = InfoSerializer(instance=instance, many= False )

#Create an object of the InfoSerializer class and get the serialized value

ser.data

```

**##### 7. Serialization -ListSerializer**

```Python

class BaseSerializer(Field):

@property

    def data(self):

        if not hasattr( self , '\_data' ):

            if self .instance is not None and not getattr( self , '\_errors' , None ):

                #here​

                self .\_data = self .to\_representation( self .instance)

            elif hasattr( self , '\_validated\_data' ) and not getattr( self , '\_errors' , None ):

                self .\_data = self .to\_representation( self .validated\_data)

            else :

                self .\_data = self .get\_initial()

        return self .\_data

class ListSerializer(BaseSerializer):

@property

    def data(self):

ret = super().data

        return ReturnList(ret, serializer= self )

    def to\_representation(self, data):

iterable = data.all() if isinstance(data, models.Manager) else data

        return [

            #Loop , use the serialization class to process each object

            self .child.to\_representation(item) for item in iterable

]

```

**### 3.2 Data Verification**

Verify the request data sent by the user .

**#### 3.2.1 Built-in verification**

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

class InfoSerializer(serializers.Serializer):

title = serializers.CharField(required= True , max\_length= 20 , min\_length= 6 )

order = serializers.IntegerField(required= False , max\_value= 100 , min\_value= 10 )

level = serializers.ChoiceField(choices=[( "1" , " Advanced " ), ( 2 , " Intermediate " )])

class InfoView(APIView):

    def post(self, request):

ser = InfoSerializer(data=request.data)

        if ser.is\_valid():

            return Response(ser.validated\_data)

        else :

            return Response(ser.errors)

```

**#### 3.2.2 Regular Expression Verification**

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from django.core.validators import RegexValidator, EmailValidator

class InfoSerializer(serializers.Serializer):

title = serializers.CharField(required= True , max\_length= 20 , min\_length= 6 )

order = serializers.IntegerField(required= False , max\_value= 100 , min\_value= 10 )

level = serializers.ChoiceField(choices=[( "1" , " Advanced " ), ( 2 , " Intermediate " )])

    # email = serializers.EmailField()

email = serializers.CharField(validators=[EmailValidator(message= " Email format error " )])

more = serializers.CharField(validators=[RegexValidator( r "\d + " , message= " Format error " )])

code = serializers.CharField()

class InfoView(APIView):

    def post(self, request):

ser = InfoSerializer(data=request.data)

        if ser.is\_valid():

            return Response(ser.validated\_data)

        else :

            return Response(ser.errors)

```

**#### 3.2.3 Hook Verification**

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from rest\_framework import exceptions

class InfoSerializer(serializers.Serializer):

title = serializers.CharField(required= True , max\_length= 20 , min\_length= 6 )

order = serializers.IntegerField(required= False , max\_value= 100 , min\_value= 10 )

code = serializers.CharField()

    def validate\_code(self, value):

print(value)

        if len(value) > 6 :

            raise exceptions.ValidationError( " Field hook validation failed " )

        return value

    def validate(self, attrs):

print( "validate=" , attrs)

        # api\_settings.NON\_FIELD\_ERRORS\_KEY

        # raise exceptions.ValidationError(" Global hook validation failed ")

        return attrs

class InfoView(APIView):

    def post(self, request):

ser = InfoSerializer(data=request.data)

        if ser.is\_valid():

            return Response(ser.validated\_data)

        else :

            return Response(ser.errors)

```

**#### 3.2.4 Model Verification**

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from rest\_framework import exceptions

from api import models

from django.core.validators import RegexValidator

class RoleSerializer(serializers.ModelSerializer):

more = serializers.CharField(required= True )

    class Meta:

model = models.Role

fields = [ "title" , "order" , "more" ]

extra\_kwargs = {

            "title" : { "validators" : [RegexValidator( r "\d + " , message= " Format error " )]},

            "order" : { "min\_value" : 5 },

}

    def validate\_more(self, value):

        return value

    def validate(self, attrs):

        return attrs

class InfoView(APIView):

    def post(self, request):

ser = RoleSerializer(data=request.data)

        if ser.is\_valid():

            return Response(ser.validated\_data)

        else :

            return Response(ser.errors)

```

**#### 3.2.5 Verify + Save**

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from rest\_framework import exceptions

from api import models

from django.core.validators import RegexValidator

class RoleSerializer(serializers.ModelSerializer):

more = serializers.CharField(required= True )

    class Meta:

model = models.Role

fields = [ "title" , "order" , "more" ]

extra\_kwargs = {

            "title" : { "validators" : [RegexValidator( r "\d + " , message= " Format error " )]},

            "order" : { "min\_value" : 5 },

}

    def validate\_more(self, value):

        return value

    def validate(self, attrs):

        return attrs

class InfoView(APIView):

    def post(self, request):

ser = RoleSerializer(data=request.data)

        if ser.is\_valid():

ser.validated\_data.pop( "more" )

instance = ser.save() # ser.save(v1=123,v2=234)

print(instance)

            return Response(ser.validated\_data)

        else :

            return Response(ser.errors)

```

**#### 3.2.6 Verify + Save + FK + M2M**

![ image-20220917204013619 ]( assets/image-20220917204013619.png )

![ image-20220917204030432 ]( assets/image-20220917204030432.png )

```Python

from django.db import models

class Role(models.Model):

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

order = models.IntegerField(verbose\_name= " order " )

class Tag(models.Model):

caption = models.CharField(verbose\_name= " name " , max\_length= 32 )

class UserInfo(models.Model):

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

gender = models.SmallIntegerField(verbose\_name= " gender " , choices=(( 1 , " male " ), ( 2 , " female " )))

role = models.ForeignKey(verbose\_name= " role " , to= "Role" , on\_delete=models.CASCADE)

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

tags = models.ManyToManyField(verbose\_name= " Tag " , to= "Tag" )

```

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from rest\_framework import exceptions

from api import models

from django.core.validators import RegexValidator

import datetime

class UserInfoSerializer(serializers.ModelSerializer):

more = serializers.CharField(required= True )

    class Meta:

model = models.UserInfo

fields = [ "name" , "gender" , "role" , "tags" , "more" ]

extra\_kwargs = {

            "name" : { "validators" : [RegexValidator( r "n-\d + " , message= " Format error " )]},

}

    def validate\_more(self, value):

        return value

    def validate(self, attrs):

        return attrs

class InfoView(APIView):

    def post(self, request):

ser = UserInfoSerializer(data=request.data)

        if ser.is\_valid():

ser.validated\_data.pop( "more" )

instance = ser.save(ctime=datetime.datetime.now())

print(instance)

            # return Response(ser.validated\_data)

            return Response( " Success " )

        else :

            return Response(ser.errors)

```

**#### 3.2.7 Hook create**

save is executed , the create or update method will be called internally . If you want to customize the save rules, you can also process them here .

![ image-20220918083811336 ]( assets/image-20220918083811336.png )

**### 3.3 Verification + Serialization**

If a request is made, **\*\* Request verification \*\*** Need to do **Serialization​​** , how to do it? For example: add new data .

- Fields can be set by read\_only , write\_only , required

- is\_valid check

- Data call serialization

**#### 3.3.1 Two-in-one**

![ image-20210823210822789 ]( assets/image-20210823210822789.png )

![ image-20210823211016050 ]( assets/image-20210823211016050.png )

![ image-20210823211041662 ]( assets/image-20210823211041662.png )

```Python

# models.py

from django.db import models

class Role(models.Model):

    """ Character List """

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

class Department(models.Model):

    """ Department table """

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

class UserInfo(models.Model):

    """ User table """

level\_choices = (( 1 , " Ordinary Member " ), ( 2 , "VIP" ), ( 3 , "SVIP" ),)

level = models.IntegerField(verbose\_name= " level " , choices=level\_choices, default= 1 )

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

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

age = models.IntegerField(verbose\_name= " Age " , default= 0 )

email = models.CharField(verbose\_name= " Email " , max\_length= 64 , null= True , blank= True )

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

depart = models.ForeignKey(verbose\_name= " Department " , to= "Department" , on\_delete=models.CASCADE, null= True , blank= True )

roles = models.ManyToManyField(verbose\_name= " role " , to= "Role" )

```

```Python

# urls.py

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

from app01 import views

urlpatterns = [

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

]

```

```Python

# views.py

from django.core.validators import EmailValidator

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import serializers

from app01 import models

class DepartModelSerializer(serializers.ModelSerializer):

    class Meta:

model = models.Department

fields = [ 'id' , "title" ]

extra\_kwargs = {

            "id" : { "read\_only" : False }, #Data validation

            "title" : { " read\_only " : True } #Serialization

}

class RoleModelSerializer(serializers.ModelSerializer):

    class Meta:

model = models.Role

fields = [ 'id' , "title" ]

extra\_kwargs = {

            "id" : { "read\_only" : False }, #Data validation

            "title" : { " read\_only " : True } #Serialization

}

class UserModelSerializer(serializers.ModelSerializer):

level\_text = serializers.CharField(source= "get\_level\_display" , read\_only= True )

    # Serializer is nested and not read\_only . You must customize create and update , and customize the logic of adding and updating .

depart = DepartModelSerializer(many= False )

roles = RoleModelSerializer(many= True )

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

email2 = serializers.EmailField(write\_only= True )

    #Data verification: username , email , email2 , department, role information

    class Meta:

model = models.UserInfo

fields = [

            "username" , "age" , "email" , "level\_text" , "depart" , "roles" , "extra" , "email2"

]

extra\_kwargs = {

            "age" : { "read\_only" : True },

            "email" : { "validators" : [EmailValidator, ]},

}

    def get\_extra(self, obj):

        return 666

    def validate\_username(self, value):

        return value

    #When adding new data

    def create(self, validated\_data):

        """ If there is a nested Serializer , there are only two options when performing data validation :

1. Set nested serialization to read\_only

2. Customize the create and update methods, and customize the logic of new creation and update

            Note: The format of data submitted by the user .

"""

depart\_id = validated\_data.pop( 'depart' )[ 'id' ]

role\_id\_list = [ele[ 'id' ] for ele in validated\_data.pop( 'roles' )]

        #Add user table

validated\_data[ 'depart\_id' ] = depart\_id

user\_object = models.UserInfo.objects.create(\*\*validated\_data)

        #Add the corresponding relationship in the association table between the user table and the role table

user\_object.roles.add(\*role\_id\_list)

        return user\_object

class UserView(APIView):

    """ User Management """

    def get(self, request):

        """ Add user """

queryset = models.UserInfo.objects.all()

ser = UserModelSerializer(instance=queryset, many= True )

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

    def post(self, request):

        """ Add user """

ser = UserModelSerializer(data=request.data)

        if not ser.is\_valid():

            return Response({ 'code' : 1006 , 'data' : ser.errors})

ser.validated\_data.pop( 'email2' )

instance = ser.save(age= 18 , password= "123" , depart\_id= 1 )

        #Add an object after adding it (internal call to UserModelSerializer for serialization )

print(instance)

        # ser = UserModelSerializer(instance=instance, many=False)

        # ser.data

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

```

**#### 3.3.2 Separate**

When executing different functions, different serializers are used to process the business .

- GET request, return data (serialized A )

- POST request, submit data (serialization A ) + return data (serialization B ) .

**\*\* Underlying source code implementation: \*\***

The underlying source code implementation of serialization is different from the other components mentioned above. The definition and execution of serializer-related classes are called in the view, so the source code analysis process can be divided into: class definition, serialization, and data verification .

Source code 1 : Serialization process

![ image-20210823235237512 ]( assets/image-20210823235237512.png )

![ image-20210823235752483 ]( assets/image-20210823235752483.png )

Source code 2 : Data verification process

![ image-20210824001814091 ]( assets/image-20210824001814091.png )

![ image-20210824001844381 ]( assets/image-20210824001844381.png )

**## 4. 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 paging 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

CursorPagination (BasePagination) support Previous & Next Page Format pagination (uncommon )

```

**### 4.1 PageNumberPagination**

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

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

**### 4.2 LimitOffsetPagination**

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

**## 5. Get to know the view**

APIView` is in 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()),

    #Other paging, filtering and other conditions should be placed in ? page=1&size=9 Pass parameters .

]

```

```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' : "..." })

```