**# day13 drf- Part 1**

**## 1. Separation of front-end and back-end**

<img src = "assets/image-20220903102113338.png" alt = "image-20220903102113338" style = "zoom:50%;" />

![ image-20220903102150169 ]( assets/image-20220903102150169.png )

**2.FBV and CBV**

- FBV , function base views , is actually writing functions to handle business requests .

```Python

  from django.contrib import admin

  from django.urls import path

  from app01 import views

urlpatterns = [

path( 'users/' , views.users),

]

```

```Python

  from django.http import JsonResponse

  def users(request,\*args, \*\*kwargs):

      if request.method == "GET" :

          return JsonResponse({ "code" : 1000 , "data" : "xxx" })

      elif request.method == 'POST' :

          return JsonResponse({ "code" : 1000 , "data" : "xxx" })

...

```

- CBV , class base views , is actually writing classes to handle business requests .

```Python

  from django.contrib import admin

  from django.urls import path

  from app01 import views

urlpatterns = [

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

]

```

```Python

  from django.views import View

  class UserView(View):

      def get(self, request, \*args, \*\*kwargs):

          return JsonResponse({ "code" : 1000 , "data" : "xxx" })

      def post(self, request, \*args, \*\*kwargs):

          return JsonResponse({ "code" : 1000 , "data" : "xxx" })

```

In fact, the underlying implementation of CBV and FBV is essentially the same .

**## 3.drf**

Django RestFramework It provides us with many convenient functions based on Django , allowing us to develop restful API based on Django more conveniently . Let's take a simple example to quickly understand it :

<img src = "../../../../../Source Code Academy /09 Courses /Django Project Class ( VIP ) /Django Project Class - Courseware Materials /drf/assets/image-20210819132209726.png" alt = "image-20210819132209726" style = "zoom:25%;" />

Note: drf supports cbv and fbv , but when developing APIs based on drf , cbv is generally used .

**### 3.1 drf project (pure version )**

For sample code, see: `drf\_demo-1.zip`

```

pip install django==3.2

pip install djangorestframework

```

**#### 3.1.1 Core Configuration**

```Python

INSTALLED\_APPS = [

    # 'django.contrib.admin',

    # 'django.contrib.auth',

    # 'django.contrib.contenttypes',

    # 'django.contrib.sessions',

    # 'django.contrib.messages',

    'django.contrib.staticfiles' ,

    'api.apps.ApiConfig' ,

    'rest\_framework'

]

MIDDLEWARE = [

    'django.middleware.security.SecurityMiddleware' ,

    # 'django.contrib.sessions.middleware.SessionMiddleware',

    'django.middleware.common.CommonMiddleware' ,

    # 'django.middleware.csrf.CsrfViewMiddleware',

    # 'django.contrib.auth.middleware.AuthenticationMiddleware',

    # 'django.contrib.messages.middleware.MessageMiddleware',

    'django.middleware.clickjacking.XFrameOptionsMiddleware' ,

]

TEMPLATES = [

{

        'BACKEND' : 'django.template.backends.django.DjangoTemplates' ,

        'DIRS' : [],

        'APP\_DIRS' : True ,

        'OPTIONS' : {

            'context\_processors' : [

                'django.template.context\_processors.debug' ,

                'django.template.context\_processors.request' ,

                # 'django.contrib.auth.context\_processors.auth',

                # 'django.contrib.messages.context\_processors.messages',

],

},

},

]

REST\_FRAMEWORK = {

    "UNAUTHENTICATED\_USER" : None

}

```

**#### 3.1.2 Routing**

```Python

from django.contrib import admin

from django.urls import path

from api import views

urlpatterns = [

    # path('admin/', admin.site.urls),

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

]

```

**#### 3.1.3 Views**

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

class UserView(APIView):

    def get(self, request):

        return Response( "..." )

```

**### 3.2 request and parameters**

**#### 3.2.1 Parameters**

For sample code, see: `drf\_demo-2.zip`

![ image-20220903122455440 ]( assets/image-20220903122455440.png )

![ image-20220903122947215 ]( assets/image-20220903122947215.png )

![ image-20220903123042906 ]( assets/image-20220903123042906.png )

**#### 3.2.2 request object**

**##### 1. Attributes**

Let’s first learn a little bit about object-oriented knowledge .

```Python

class Request(object):

    def \_\_init\_\_(self, req, xx):

        self .\_request = req

        self .xx = xx

obj = Request( 1 , 2 )

print(obj.xx)

print(obj.\_request)

```

When you get a member of an object, you essentially call `\_\_getattribute\_\_` method, by default we don't define it and use the one in the parent class .

```Python

class Request(object):

    def \_\_init\_\_(self, req, xx):

        self .\_request = req

        self .xx = xx

    def \_\_getattribute\_\_(self, item):

print( " Execute \_\_getattribute\_\_" , item)

        return super().\_\_getattribute\_\_(item)

obj = Request( 1 , 2 )

print(obj.xx)

print(obj.\_request)

# int(obj.v1) # error

#Note : If it is not a member of the object, an error will be reported .

```

However, if you want to access a member that does not exist in the object, you can define `\_\_getattr\_\_` implementation .

- Execute your own first `\_\_getattribute\_\_`

- Then execute the parent class's `\_\_getattribute\_\_`

  - It is its own object, directly obtain and return

  - Not its own object, calling `\_\_getattr\_\_`

```Python

class Request(object):

    def \_\_init\_\_(self, req, xx):

        self .\_request = req

        self .xx = xx

    def \_\_getattribute\_\_(self, item):

print( " Execute \_\_getattribute\_\_" , item)

        return super().\_\_getattribute\_\_(item)

    def \_\_getattr\_\_(self, item):

print( "\_\_getattr\_\_" , item)

        return 999

obj = Request( 1 , 2 )

print(obj.xx)

print(obj.\_request)

print(obj.v1)

```

**##### 2. Object encapsulation**

```Python

class HttpRequest(object):

    def \_\_init\_\_(self):

        pass

    def v1(self):

print( "v1" )

    def v2(self):

print( "v1" )

class Request(object):

    def \_\_init\_\_(self,req,xx):

        self .\_request = req

        self .xx = xx

request = HttpRequest()

request.v1()

request.v2()

request = Request(request, 111 )

request.\_request.v1()

request.\_request.v2()

```

```Python

class HttpRequest(object):

    def \_\_init\_\_(self):

        pass

    def v1(self):

print( "v1" )

    def v2(self):

print( "v1" )

class Request(object):

    def \_\_init\_\_(self,req,xx):

        self .\_request = req

        self .xx = xx

    def \_\_getattr\_\_(self, attr):

        try :

            return getattr( self .\_request, attr)

        except AttributeError:

            return self .\_\_getattribute\_\_(attr)

request = HttpRequest()

request.v1()

request.v2()

request = Request(request, 111 )

request.v1()

request.v2()

```

**##### 3. Source code analysis**

![ image-20210819150601089 ]( assets/image-20210819150601089.png )

**##### 4. request object**

The request in drf is actually a re-encapsulation of the request. Its purpose is to encapsulate some values needed in drf based on the original request object .

For sample code, see: `drf\_demo-3.zip`

![ image-20220904073610480 ]( assets/image-20220904073610480.png )

**### 3.3 Authentication**

In the process of developing API , some functions need to be logged in to access, while some do not. The authentication component in drf is mainly used to implement this function .

Regarding the authentication component, we will first use cases to learn common usage scenarios and then analyze the source code .

**#### 3.3.1 Case 1**

> The project needs to develop three interfaces, one of which does not require login and two interfaces that must be logged in to access .

>

> For sample code, see: `drf\_demo-4.zip`

![ image-20220904082534101 ]( assets/image-20220904082534101.png )

![ image-20220904085937250 ]( assets/image-20220904085937250.png )

Visit in the browser: `/order/token=xxxdsfsdfdf`

The two values returned by the authentication component are assigned to: `request.user` and `request.auth` .

**#### 3.3.2 Case 2**

> The project needs to develop 100 interfaces, including 1 interface that does not require login and 99 interfaces that must be logged in to access .

>

> At this point, you need to use the global configuration of drf (the class of the authentication component cannot be placed in the view view.py , which will cause a circular reference due to the import of APIView ) .

>

> For sample code, see: `drf\_demo-5.zip`

![ image-20220904084906568 ]( assets/image-20220904084906568.png )

**#### 3.3.3 Case 3**

> The project will develop 100 interfaces, including 1 interface that does not require login, 98 interfaces that must be logged in to access, and 1 public interface (displays public when not logged in / displays personal information when logged in) .

>

> For sample code, see: `drf\_demo-6.zip`

![ image-20220904090855727 ]( assets/image-20220904090855727.png )

**#### 3.3.4 Case 4**

> The project will develop 100 interfaces, including 1 interface that does not require login, 98 interfaces that must be logged in to access, and 1 public interface (displays public when not logged in / displays personal information when logged in) .

>

> The original authentication information can only be passed in the URL . If the program supports it, it can be placed in many places, such as: in the URL , in the request header , etc.

>

> In the authentication component, if multiple authentication classes are used, the `authenticate` methods will be executed one by one in order.

>

> - Return None or no return value, indicating that the subsequent authentication class will continue to execute

> - return  (user, auth) tuple, then no further progress is made and the value is assigned to request.user and request.auth

> - throw an exception `AuthenticationFailed(...)` , authentication failed, no further steps will be taken .

>

> For sample code, see: `drf\_demo-7.zip`

![ image-20220904093128480 ]( assets/image-20220904093128480.png )

**#### 3.3.5 Source code analysis**

![ image-20210822092707803 ]( assets/image-20210822092707803.png )

**### 3.4 Permissions**

In drf development, if some interfaces must meet the following conditions at the same time: A condition, B condition, and C condition.  Some interfaces only need to meet conditions B and C. In this case, you can use the permission component to write these conditions .

- And relationship, default support: A condition And condition B And the C condition is satisfied at the same time .

```Python

  classPermissionA (BasePermission):

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

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

          if request.user.role == 2 :

              return True

          return False

      # Write this for now

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

          return True

```

- or relationship, custom (easy to expand )

```Python

  class APIView(View):

    def check\_permissions(self, request):

          """

    Check if the request should be permitted.

    Raises an appropriate exception if the request is not permitted.

    """

          for permission in self .get\_permissions():

              if not permission.has\_permission(request, self ):

                  self .permission\_denied(

request,

message=getattr(permission, 'message' , None ),

code=getattr(permission, 'code' , None )

)

```

**#### Thinking question: Customizing the request object**

request object during development ?

**### 3.5 Current Limitation**

Limit the access frequency of users, for example: users can access at most 100 times in 1 minute or SMS verification codes can be sent 50 times a day . Prevent fraudulent use .

- For anonymous users, the user IP is used as the unique identifier .

- For logged in users, use the user ID or name as the unique identifier .

```Python

cache ={

    User ID: [ 12 : 33 , 12 : 32 , 12 : 31 , 12 : 30 , 12 ,] 1 hour / 5 times   12 : 34   11 : 34

{

```

```

pip3 install django-redis

```

```Python

# settings.py

CACHES = {

    "default" : {

        "BACKEND" : "django\_redis.cache.RedisCache" ,

        "LOCATION" : "redis://127.0.0.1:6379" ,

        "OPTIONS" : {

            "CLIENT\_CLASS" : "django\_redis.client.DefaultClient" ,

            "PASSWORD" : "qwe123" ,

}

}

}

```

![ image-20210822115201724 ]( assets/image-20210822115201724.png )

```Python

CACHES = {

    "default" : {

        "BACKEND" : "django\_redis.cache.RedisCache" ,

        "LOCATION" : "redis://127.0.0.1:6379" ,

        "OPTIONS" : {

            "CLIENT\_CLASS" : "django\_redis.client.DefaultClient" ,

            "PASSWORD" : "qwe123" ,

}

}

}

```

```Python

from django.urls import path, re\_path

from app01 import views

urlpatterns = [

path( 'api/order/' , views.OrderView.as\_view()),

]

```

```Python

# views.py

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework import exceptions

from rest\_framework import status

from rest\_framework.throttling import SimpleRateThrottle

from django.core.cache import cache as default\_cache

class ThrottledException(exceptions.APIException):

status\_code = status.HTTP\_429\_TOO\_MANY\_REQUESTS

default\_code = 'throttled'

class MyRateThrottle(SimpleRateThrottle):

cache = default\_cache #Access records are stored in Django 's cache (caching needs to be set )

scope = "user"  #Construct the key in the cache

cache\_format = 'throttle\_%(scope)s\_%(ident)s'

    #Set the access frequency, for example: 10 accesses are allowed per minute

    #Others : 's', 'sec', 'm', 'min', 'h', 'hour', 'd', 'day'

THROTTLE\_RATES = { "user" : "10/m" }

    def get\_cache\_key(self, request, view):

        if request.user:

ident = request.user.pk #User ID

        else :

ident = self .get\_ident(request) #Get the requesting user IP ( find the request header in request )

        # throttle\_u # throttle\_user\_11.11.11.11ser\_2

        return self .cache\_format % { 'scope' : self .scope, 'ident' : ident}

    def throttle\_failure(self):

wait = self .wait()

detail = {

            "code" : 1005 ,

            "data" : " Access frequency limit " ,

            'detail' : " Wait {}s before access " .format(int(wait))

}

        raise ThrottledException(detail)

class OrderView(APIView):

throttle\_classes = [MyRateThrottle, ]

    def get(self, request):

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

```

**\*\* Multiple current limiting categories \*\***

In essence, each current limiting class has a `allow\_request` Method, there can be three situations inside this method :

- Return True , indicating that the current current limiting class allows access and the subsequent current limiting classes continue to execute .

- Return False , indicating that the current limit class is not allowed to access, and continue to execute the subsequent limit classes. After all limit classes are executed, read all the limit classes that are not allowed, and calculate the waiting time .

- Throwing an exception indicates that the current current limiting class is not allowed to access, and subsequent current limiting classes will no longer be executed .

**\*\* Global Configuration \*\***

```Python

REST\_FRAMEWORK = {

    "DEFAULT\_THROTTLE\_CLASSES" :[ "xxx.xxx.xx. current limiting class " , ],

    "DEFAULT\_THROTTLE\_RATES" : {

        "user" : "10/m" ,

        "xx" : "100/h"

}

}

```

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

![ image-20210822121259284 ]( assets/image-20210822121259284.png )

![ image-20210822120127336 ]( assets/image-20210822120127336.png )

**### Version 3.6**

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

**#### 3.1 URL 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 )

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

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

**#### 3.3 Request header transmission**

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

**\*\* Global Configuration \*\***

In the above example, if you want to apply a Version In the form of, you need to define class variables in each view class :

```Python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from rest\_framework.versioning import QueryParameterVersioning

class UserView(APIView):

versioning\_class = QueryParameterVersioning

...

```

If your project is large and requires many view classes, it would be troublesome to write them in each class. All drf also supports global configuration .

```Python

# settings.py

REST\_FRAMEWORK = {

    "DEFAULT\_VERSIONING\_CLASS" : "rest\_framework.versioning.QueryParameterVersioning" , #Path to the class that handles the version

    "VERSION\_PARAM" : "version" , # key when passing URL parameters , for example: xxxx?version=v1

    "ALLOWED\_VERSIONS" : [ "v1" , "v2" , "v3" ], #Limit supported versions, None means no limit

    "DEFAULT\_VERSION" : "v1" , #Default version

}

```

![ image-20210820113538002 ]( assets/image-20210820113538002.png )

Access URL :

````

http://127.0.0.1:8000/api/users/?version=v1

http://127.0.0.1:8000/api/users/?version=v2

http://127.0.0.1:8000/api/users/?version=v3

http://127.0.0.1:8000/api/admin/?version=v1

http://127.0.0.1:8000/api/admin/?version=v2

http://127.0.0.1:8000/api/admin/?version=v3

http://127.0.0.1:8000/api/v1/order/

http://127.0.0.1:8000/api/v2/order/

http://127.0.0.1:8000/api/v3/order/

````

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

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

**\*\* Reverse generate URL\*\***

`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.png )

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

**### 3.7 Parser**

Previously used `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 .

**#### 3.7.1 JSONParser ( \* )**

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

**#### 3.7.2 FormParser**

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

**#### 3.7.3 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>

```

**#### 3.7.4 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( "..." )

```

**### 3.8 Serializer ( \* )**

DRF provides us with Serializer , which has two main functions :

- Verify the request data (the underlying call Django 's Form and ModelForm )

- Serialize objects queried from the database

**#### 3.8.1 Data Verification**

Example 1 : Based on Serializer :

![ image-20210823084033952 ]( assets/image-20210823084033952.png )

Example 2 : Based on ModelSerializer :

```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 )

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

    Foreign Keys​

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

    # Many-to-Many

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

```

![ image-20210823085008103 ]( assets/image-20210823085008103.png )

*\* Tip: The save method will return the newly generated data object. \**

Example 3 : Based on ModelSerializer (including FK+M2M ) :

![ image-20210823085945420 ]( assets/image-20210823085945420.png )

*\* Tip: The save method will return the newly generated data object. \**

**#### 3.8.2 Serialization**

A QuerySet obtained from the database through the ORM or Object Can be serialized into json format data .

```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" )

```

Example 1 : Serializing basic fields

![ image-20210823160227040 ]( assets/image-20210823160227040.png )

```Python

#Remember , If what is obtained from the database is not a QuerySet object, but a single object, for example :

data\_object = modes.UserInfo.objects.filter(id= 2 ).first()

ser = UserModelSerializer(instance=data\_object,many= False )

print(ser.data)

```

Example 2 : Custom fields

![ image-20210823161608120 ]( assets/image-20210823161608120.png )

Example 3 : Nesting of serialized classes

![ image-20210823162145013 ]( assets/image-20210823162145013.png )

**#### 3.8.3 Data Verification & Serialization**

The above examples all belong to a single function (either verification or serialization). In fact, when we write a serialization class, we can do both data verification and serialization, for example :

![ 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})

```

**\*\* 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 )

**###**