ModelSerializers:

If our serializable objects are Django model objects, then it is highly recommended to go for ModelSerializer.

- ModelSerializer class is exactly same as regular serializer classe except the following differences

1) The fields will be considered automatically based on the model and we are not required to specify explicitly.

2) It provides default implementation for create() and update() methods.

Note: ModelSerializer won't provide any extra functionality and it is just for typing shortcut.

We can define ModelSerializer class as follows:

class EmployeeSerializer(serializers.ModelSerializer):

class Meta:

model=Employee

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

Here we are not required to specify fields and these will be considered automatically based on Model class. We are not required to implement create() and update() methods, because ModelSerializer class will provide these methods.

\*\*\*Note: If we want to define validations for any field then that particular field we have to declare explicitly.

def multiples\_of\_1000(value):

print('validations by using validator')

if value % 1000 != 0:

raise serializers.ValidationError('Salary should be multiples of 1000s')

class EmployeeSerializer(serializers.ModelSerializer):

esal=serializers.FloatField(validators=[multiples\_of\_1000,])

class Meta:

model=Employee

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

Q) In how many Ways we can specify Fields in ModelSerializer Class?

3 Ways

1) To include all fields

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

2) To include only some fields

fields = ('eno','ename','eaddr')

This approach is helpful if we want to include very less number of fields.

3) To exclude some fields

exclude = ('esal')

Except esal, all remaining fields will be considered.

If we want to consider majarity of the fileds then this approach is helpful.

Django REST Framework Views:

DRF provides 2 classes to define business logic for our API Views.

1) APIView

2) ViewSet

1) APIView:

--> It is the most basic class to build REST APIs. It is similar to Django traditional View class.

--> It is the child class of Django's View class.

--> It allows us to use standard HTTP methods as functions like get(),post(),put() etc

--> Here, we have to write complete code for business logic and hence programmer having complete control on the logic. We can understand flow of execution very clearly.

--> Best suitable for complex operations like working with multiple datasources, calling other APIs etc

--> We have to define url mappings manually.

Where APIViews are best suitable?

1) If we want complete control over the logic

2) If we want clear execution flow

3) If we are calling other APIs in the same request

4) If we want to work with multiple data sources simultaneously

5) If we want to perform any complex operations

etc

How to send Response in APIViews/ViewSets:

To send response to the partner/client application, DRF provides Response class. It will convert input data to json format automatically.

For get Request:

from rest\_framework.views import APIView

from rest\_framework.response import Response

class TestApiView(APIView):

def get(self,request,format=None):

colors=['RED','BLUE','GREEN','YELLOW','INDIGO']

return Response({'msg':'Welcome to Colorful Year','colors':colors})

For Post Request:

In post request, partner/client application will send resource data in the form of json. To convert this json data to python native types, serializer is required.

serializers.py

from rest\_framework import serializers

class NameSerializer(serializers.Serializer):

name=serializers.CharField(max\_length=7)

views.py

def post(self,request):

serializer=NameSerializer(data=request.data)

if serializer.is\_valid():

name=serializer.data.get('name')

msg='Hello {} !!!'.format(name)

return Response({'msg':msg})

return Response(serializer.errors,status=400)

How to test POST Method:

We should provide json as input and we should use double quotes

input: {"name":"naresh"}

response:

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

{

"msg": "Hello naresh !!!"

}

input: {"name":"naresh"}

response:

HTTP 400 Bad Request

Allow: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

{

"name": [ "Ensure this field has no more than 7 characters."

]

}

How to implement put(), patch() and delete() Methods:

class TestApiView(APIView):

.....

def put(self,request,pk=None):

return Response({'msg':'Response from put method'})

def patch(self,request,pk=None):

return Response({'msg':'Response from patch method'})

def delete(self,request,pk=None):

return Response({'msg':'Response from delete method'})

Complete Application:

views.py

from django.shortcuts import render

from rest\_framework.views import APIView

from rest\_framework.response import Response

from testapp.serializers import NameSerializer

# Create your views here.

class TestApiView(APIView):

def get(self,request,format=None):

colors=['RED','BLUE','GREEN','YELLOW','INDIGO']

return Response({'msg':'Welcome ','colors':colors})

def post(self,request):

serializer=NameSerializer(data=request.data)

if serializer.is\_valid():

name=serializer.data.get('name')

msg='Hello {} Wish You Happy New Year !!!'.format(name)

return Response({'msg':msg})

return Response(serializer.errors,status=400)

def put(self,request,pk=None):

return Response({'msg':'Response from put method'})

def patch(self,request,pk=None):

return Response({'msg':'Response from patch method'})

def delete(self,request,pk=None):

return Response({'msg':'Response from delete method'})

serializers.py

from rest\_framework import serializers

class NameSerializer(serializers.Serializer):

name = serializers.CharField(max\_length=7)

urls.py

from django.conf.urls import url,include

from django.contrib import admin

from testapp import views

urlpatterns = [

url('admin/', admin.site.urls),

url('api/', views.TestApiView.as\_view()),

]

2) ViewSets:

--> By using ViewSets, we can provide business logic for our API views.

--> It is alternative to APIView class.

--> In the case of APIView, we can use HTTP Methods as functions like get(), post() etc. But in ViewSet, We have to use Model class actions/operations for function names.

-->list() To get all resources/records/objects

-->retrieve() To get a specific resource

--> create() To create a new resource

--> update() To update a resource

--> partial\_update() To perform partial updation of resource.

--> destroy() To delete a resource.

Mapping:

get() list() and retrieve()In APIViews, we have to write total logic. But in ViewSets most of the logic will be provided automatically. Hence we can provide more functionality with less code and we can develop API very quickly in less time.

When ViewSets are Best Choice:

1) If we want to develop a simple CRUD interface to our database.

2) If we want to develop a simple and quick API to manage predefined objects

3) If we are performing only standard operations with very less or no customization.

4) If we are not performing any complex operations like calling other APIs,using multiple data sources etc

post() create()

put() update()

patch() partial\_update()

delete() destroy()

Sample Code for List Operation:

from rest\_framework import viewsets

class TestViewSet(viewsets.ViewSet):

def list(self,request):

colors=['RED','GREEN','YELLOW','ORANGE']

return Response({'msg':'Hello ','colors':colors})

Defining Router for TestViewSet:

In APIViews, we have to map views to urls manually. But in ViewSet, we are not required to do explicitly. DRF provides a 'DefaultRouter' class to map ViewSet to the urls, which are used by partner application. Routers provide an easy way of automatically determining the URL configurations.

Routers are required only for views developed by ViewSet.

 We have to add the following lines to urls.py:

1) from rest\_framework import routers

2) router=routers.DefaultRouter()

3) router.register('test-viewset',views.TestViewSet,base\_name='test-viewset')

4)

5) urlpatterns = [

6) ...

7) url(r'',include(router.urls))

8) ]

urls.py

1) from django.conf.urls import url,include

2) from django.contrib import admin

3) from testapp import views

4) from rest\_framework import routers

5) router=routers.DefaultRouter()

6) router.register('test-viewset',views.TestViewSet,base\_name='test-viewset')

7)

8) urlpatterns = [

9) url(r'^admin/', admin.site.urls),

10) # url(r'^api/', views.TestApiView.as\_view()),

11) url(r'',include(router.urls))

12) ]

create(),retrieve(),update(),partial update() and destroy() Methods:

1) from rest\_framework import viewsets

2) class TestViewSet(viewsets.ViewSet):

3) ...

4) def create(self,request):

5) serializer=NameSerializer(data=request.data)

6) if serializer.is\_valid():

7) name=serializer.data.get('name')

8) msg='Hello {} Your Life will be settled in 2019'.format(name)

9) return Response({'msg':msg})

10) return Response(serializer.errors,status=400)

11) def retrieve(self,request,pk=None):

12) return Response({'msg':'Response from retrieve method'})

13) def update(self,request,pk=None):

14) return Response({'msg':'Response from update method'})

15) def partial\_update(self,request,pk=None):

16) return Response({'msg':'Response from partial\_update method'})

17) def destroy(self,request,pk=None):

18) return Response({'msg':'Response from destroy method'})

Complete Application:

views.py

1) from rest\_framework.response import Response

2) from testapp.serializers import NameSerializer

3) from rest\_framework import viewsets

4) class TestViewSet(viewsets.ViewSet):

5) def list(self,request):

6) colors=['RED','GREEN','YELLOW','ORANGE']

7) return Response({'msg':'Hello','colors':colors})

8) def create(self,request):

9) serializer=NameSerializer(data=request.data)

10) if serializer.is\_valid():

11) name=serializer.data.get('name')

12) msg='Hello {} '.format(name)

13) return Response({'msg':msg})

14) return Response(serializer.errors,status=400)

15) def retrieve(self,request,pk=None):

16) return Response({'msg':'Response from retrieve method'})

17) def update(self,request,pk=None):

18) return Response({'msg':'Response from update method'})

19) def partial\_update(self,request,pk=None):

20) return Response({'msg':'Response from partial\_update method'})

21) def destroy(self,request,pk=None):

22) return Response({'msg':'Response from destroy method'})

serializers.py

1) from rest\_framework import serializers

2) class NameSerializer(serializers.Serializer):

3) name = serializers.CharField(max\_length=7)

urls.py

1) from django.conf.urls import url,include

2) from django.contrib import admin

3) from testapp import views

4) from rest\_framework import routers

5) router=routers.DefaultRouter()

6) router.register('test-viewset',views.TestViewSet,base\_name='test-viewset')

7)

8) urlpatterns = [

9) url(r'^admin/', admin.site.urls),

10) # url(r'^api/', views.TestApiView.as\_view()),

11) url(r'',include(router.urls))

12) ]