Developing WEB APIs by using 3rd Party Django REST Framework:

There are several 3rd party frameworks are available to build Django REST APIs like

1) Tastify

2) Django REST Framework(DRF)

etc

But DRF is the most commonly used and easy to use froamework t build REST APIs for Django Applications

Speciality of DRF:

Some reasons you might want to use REST framework:

1) The Web browsable API is a huge usability win for your developers.

2) Authentication policies including packages for OAuth1a and OAuth2.

3) Serialization that supports both ORM and non-ORM data sources.

4) Customizable all the way down

5) Extensive documentation and great community support.

6) Used and trusted by internationally recognised companies including Mozilla, Red Hat, Heroku, and Eventbrite.

How to install DRF:

django-rest-framework.org

Installation:

Step-1: Install DRF

pip install djangorestframework

pip install markdown # Markdown support for the browsable API.

pip install django-filter # Filtering support

Note: After installing all required softwares, it is highly recommended to save installed software information inside a file, so that, it is helpful for production environment to know version requirements.

pip freeze > requirements.txt

Step-2: Add 'rest\_framework' to our INSTALLED\_APPS settings.py file

INSTALLED\_APPS = [

....

'rest\_framework', # 3rd party application

]

Step-3: Adding required urls inside urls.py:

If we are using the browsable API, to make required urls available,add the following to our urls.py file

urlpatterns = [

...

url(r'^api-auth/', include('rest\_framework.urls'))

]

Serializers:

DRF Serializers are responsible for the following activities

1) Serialization

2) Deserialization

3) Validation

Note: DRF Serializers will work very similar to Django Forms and ModelForm classes

1) Serialization:

The process of converting complex objects like Model objects and QuerySets to Python native data types like dictionary etc,is called Serialization.

The main advantage of converting to python native data types is we can

convert(render) very easily to JSON,XML etc

Defining Serializer Class:

models.py

from django.db import models

class Employee(models.Model):

eno=models.IntegerField()

ename=models.CharField(max\_length=64)

esal=models.FloatField()

eaddr=models.CharField(max\_length=64)

serializers.py

from rest\_framework import serializers

class EmployeeSerializer(serializers.Serializer):

eno=serializers.IntegerField()

ename=serializers.CharField(max\_length=64)

esal=serializers.FloatField()

eaddr=serializers.CharField(max\_length=64)

Converting Employee Object to Python Native Data Type By using

EmployeeSerializer (Serialization Process):

>>> from testapp.models import Employee

>>> from testapp.serializers import EmployeeSerializer

>>> emp=Employee(eno=100,ename='Naresh',esal=1000,eaddr='Hyd')

>>> eserializer=EmployeeSerializer(emp)

>>> eserializer.data

{'eno': 100, 'ename': 'Naresh', 'esal': 1000.0, 'eaddr': 'Hyd'}

Just we converted Employee object to python native data type(dict)

Converting Python native data type to JSON:

>>> from rest\_framework.renderers import JSONRenderer

>>> json\_data=JSONRenderer().render(eserializer.data)

>>> json\_datab'{"eno":100,"ename":"Naresh","esal":1000.0,"eaddr":"Hyd"}'

How to perform serialization for QuerySet:

>>> qs=Employee.objects.all()

>>> qs<QuerySet [<Employee: Employee object>, <Employee: Employee object>]>

>>> eserializer=EmployeeSerializer(qs,many=True)

>>> eserializer.data

[OrderedDict([('eno', 100), ('ename', 'Naresh'), ('esal', 1000.0), ('eaddr', 'Hyderabad')]),

OrderedDict([('eno', 200), ('ename', 'Suresh'), ('esal', 2000.0), ('eaddr', 'Mumbai')])]

>>> json\_data=JSONRenderer().render(eserializer.data)

>>> json\_data

b'[{"eno":100,"ename":"Naresh","esal":1000.0,"eaddr":"Hyderabad"},{"eno":200,"ename

":"Suresh","esal":2000.0,"eaddr":"Mumbai"}]'

2) Deserialization:

The process of converting python native data types complex data types like Model objects is called deserialization.

First we have to convert json\_data to python native data type.

import io

from rest\_framework.parsers import JSONParser

stream=io.BytesIO(json\_data)

data=JSONParser().parse(stream)

Now, we have to convert python native data type to database supported complex type(deserialization)

serializer=EmployeeSerializer(data=data)

serializer.is\_valid()

serializer.validated\_data

Use Case of Serialization and Deserialization:

If the partner application sends get request,then we have to convert database specific model objects or querysets to json form and we have to send that json data to the partner application. For this serialization is required.

If the partner application sends either post or put request with some json data, then our django application has to convert that json data into database specific form. For this deserialization is required.

get() Method Implementation by using Serializers:

models.py

from django.db import models

# Create your models here.

class Employee(models.Model):

eno=models.IntegerField()

ename=models.CharField(max\_length=64)

esal=models.FloatField()

eaddr=models.CharField(max\_length=64)

admin.py

from django.contrib import admin

from testapp.models import Employee

# Register your models here.

admin.site.register(Employee)

serializers.py

from rest\_framework import serializers

class EmployeeSerializer(serializers.Serializer):

eno=serializers.IntegerField()

ename=serializers.CharField(max\_length=64)

esal=serializers.FloatField()

eaddr=serializers.CharField(max\_length=64)

views.py

from django.shortcuts import render

from django.views.generic import View

import io

from rest\_framework.parsers import JSONParser

from testapp.models import Employee

from testapp.serializers import EmployeeSerializer

from rest\_framework.renderers import JSONRenderer

from django.http import HttpResponse

# Create your views here.

class EmployeeCRUDCBV(View):

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

json\_data=request.body

stream=io.BytesIO(json\_data)

data=JSONParser().parse(stream)

id=data.get('id',None)

if id is not None:

emp=Employee.objects.get(id=id)

serializer=EmployeeSerializer(emp)

json\_data=JSONRenderer().render(serializer.data)

return HttpResponse(json\_data,content\_type='application/json')

qs=Employee.objects.all()

serializer=EmployeeSerializer(qs,many=True)

json\_data=JSONRenderer().render(serializer.data)

return HttpResponse(json\_data,content\_type='application/json')

test.py

import requests

import json

BASE\_URL='http://127.0.0.1:8000/'

ENDPOINT='api/'

def get\_resources(id=None):

data={}

if id is not None:

data={

'id':id

}

resp=requests.get(BASE\_URL+ENDPOINT,data=json.dumps(data))

print(resp.status\_code)

print(resp.json())

get\_resources()

Create Opeation/POST Request by using Serializers:

To perform create operation, we have to override create() method in the serializer class.

from rest\_framework import serializers

from testapp.models import Employee

class EmployeeSerializer(serializers.Serializer):

eno=serializers.IntegerField()

ename=serializers.CharField(max\_length=64)

esal=serializers.FloatField()

eaddr=serializers.CharField(max\_length=64)

def create(self,validated\_data):

return Employee.objects.create(\*\*validated\_data)

views.py(post method):

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

json\_data=request.body

stream=io.BytesIO(json\_data)

data=JSONParser().parse(stream)

serializer=EmployeeSerializer(data=data)

if serializer.is\_valid():

serializer.save()

msg={'msg':'Resource Created Succesfully'}

json\_data=JSONRenderer().render(msg)

return HttpResponse(json\_data,content\_type='application/json')

else:

json\_data=JSONRenderer().render(serializer.errors)

return HttpResponse(json\_data,content\_type='application/json')

Note:

1) To send post request csrf verification should be disabled

2) Before calling save() method,compulsory we should call is\_valid() method, otherwise we will get error.

AssertionError: You must call .is\_valid() before calling .save()

3) After validation we can print validated data by using

serializer.validated\_data variable

print(serializer.validated\_data)