DJango Rest Framework:

1. Most of the people dont know this exciting technology
2. It is compulsory required technology. Django application with out REST API is like a human being without hands and legs.
3. It is very very easy technology
4. It is very very small technology.

1) API
2) Web API/Web Service
3) REST
4) REST API
5) DJango Rest Framework

API:

API → Application Programming Interface
The main objective of API is two applications can communicate with each other. API allows external agent to communicate (integrate and exchange information) with our application.

In Simple way: 'Methods of communication between software components'

Eg1: By using API a java application can communicate with python application. Bookmyshow application can communicate with Payment gateway application to complete our booking.

Eg2: Authentication with Facebook

Note: Interface of communication between the user and application is nothing but API. The user can be Human user, an android app or desktop application etc

Web API/Web Service:

The interface of communication between the user and application over the web by using HTTP is nothing but Web API.

REST: Representational State Transfer:

Representational State Transfer(REST) is an architectural style.
It defines several Rules/Guide Lines to develop Web APIs/Web Services

By using REST, we can develop web APIs very easily in concise way.
Hence REST is the most popular Architecture to develop Web Services.

RESTFul API:

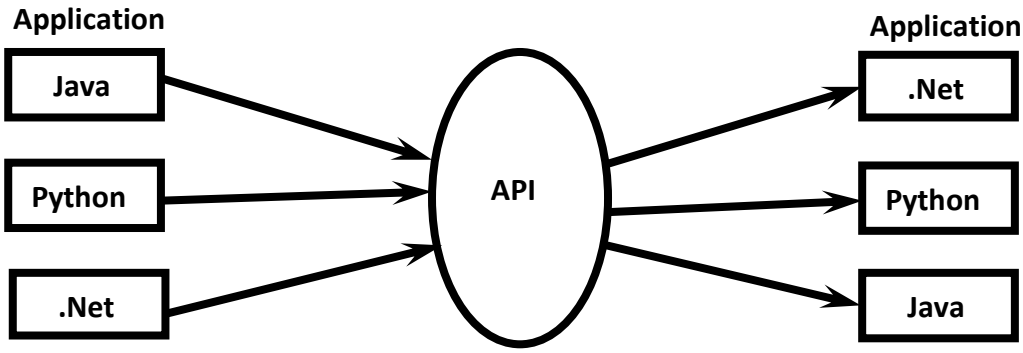
The API which is developed by using REST Architecture is nothing but RESTFul API. i.e interface between the user and application where API implements REST Architecture.

Note: REST is basically an architecture where as RESTFul API is an API that implements REST.

DJango Rest Framework:

Django REST framework is a powerful and flexible toolkit for building Web APIs.
It is the most commonly used framework in Python World to build WEB APIs.
This framework internally uses all Django facilities like models, views, templates, ORM etc
website: https://www.django-rest-framework.org/

Current Version of DJango Rest Framework: 3.9
It requires:
Python (2.7, 3.4, 3.5, 3.6, 3.7)
Django (1.11, 2.0, 2.1)



Java App → API → .NET App

By using Web Services, Any application in the world can communicates with any other application irrespective of language (like Java, Python, .Net etc) and platform (like windows,Linux,MAC etc).
The applications can communicate by using HTTP Protocol as the common language.
The Message Format is XML or JSON.

API: Can be used to access functionality of any application.
It may be stand alone application/web application/enterprise application

Web API/Web Service:

Can be used to access functionality of web application by using HTTP

REST: It is an architecture, which provides several guidelines to develop web APIs very easily and effectively.

RESTful API: **The Web API which implements REST principles is called RESTful API.**

Django REST Framework:

It provides toolkit to develop RESTful APIs for django based applications very easily.

The main objective of web APIs is interoperability. ie different applications can communicate with each other irrespective of develop language and platform.

The common communication language is: HTTP
The common message format is: XML/JSON

Note: **The most common data representation in web API is Java Script Object Notation (JSON). It is a collection of key-value pairs just like python dictionaries.**

{'eno':100,'ename':'durga','esal':1000,'eaddr':'hyd'}

The main advantage of JSON over XML is, it is Machine Friendly and Human Friendly Form.

HTTP Verbs:

HTTP Verbs represent the type of operation what we required.
Based on requirement we have to use the corresponding HTTP verb.

The following are various HTTP Verbs

1) **GET → To get one/more existing resources**
2) **POST → To create a new resource**
3) **PUT → To update an existing resource like update all fields of employee**
4) **PATCH → To perform partial updation of an existing resource like updating only salary of employee**
5) **DELETE → To delete an existing resource.**

Note: **These are only important HTTP Verbs. The following are not that much important verbs.**

- **OPTIONS**
- **HEAD**
- **CONNECT**
- **TRACE**
- **LOCK**
- **MOVE**
- **PROFIND**

etc

HTTP Verbs vs Database CRUD Operations:

C (CREATE) → POST
R (RETRIEVE/READ) → GET
U (UPDATE) → PUT/PATCH
D (DELETE) → DELETE

How to install Atom IDE:

Download installer from: atom.io

How to configure Atom for Python:

1) Python AutoCompletion:

File->Settings->Install->in the searchbox just type python-->autocomplete-python

2) django:

File → Settings → Install → In the searchbox just type djanngo → atom-django

How to install Django Rest Framework:

1) **pip install djangorestframework**
2) **pip install markdown # To provide support for browsable api**
3) **pip install django-filter # Filtering support**

Types of Web Services:

There are 2 types of web services

1) **SOAP Based WebServices**
2) **RESTful WebServices**

1) SOAP Based Web Services:

- **SOAP: Simple Object Access Protocol.**
- **SOAP is an XML based protocol for accessing web services.**
- **To describe SOAP based web services we have to use a special language: WSDL (Web Service Description Language).**
- **SOAP based web services are more secured. We can consume by using RPC Method calls. These web services can provide support for multiple protocols like HTTP, SMTP, FTP etc**

Limitations:

1) **SOAP Based web services will always provide data only in XML format. Parsing of this XML data is very slow, which creates performance problems.**
2) **Transfter of XML data over network requires more bandwidth.**
3) **Implementing SOAP Based Web Services is very difficult.**

Note: Because of heavy weight, less performance and more bandwidth requirements, SOAP based web services are not commonly used these days.

2) RESTful Web Services:

- REST stands for Representational State Transfer. It means that each unique URL is a represention of some object. We can get contents of this object by using HTTP GET, we can modify by using PUT/PATCH and we can delete by using DELETE.
- We can create by using POST.
- Most of the times RESTful web service will provide data in the form of JSON, parsing is not difficult. Hence this type of web services are faster when compared with SOAP based Web Services.
- Transfer of JSON Data over the network requires less bandwidth.

Limitations:

1) It is less secured.
2) It provide support only for the protocols which can provide URI, mostly HTTP.

Note: Because of ligth weight, high performance, less bandwidth requirements, easy development, human understandable message format, this type of web services are most commonly used type of web services.

Differences between SOAP and REST

SOAP	REST
1) XML Based Message Protocol.	1) An Architecture Syle but not protocol.
2) Uses WSDL for communication between consumer and provider.	2) Uses xml/json to send and receive data.
3) Invokes services by using RPC Method calls.	3) Invokes services by simply URL Path.
4) Does not return human readable result.	4) Returns readable results like plain xml or JSON.
5) These are heavy weight.	5) These are light weight.
6) These require more bandwidth.	6) These require less bandwidth.
7) Can provide support for multiple protocols like HTTP,SMTP,FTP etc.	7) Can provide support only for protocols that provide URI, mostly HTTP.
8) Performance is Low.	8) Perforamnce is High.
9) More Secured.	9) Less Secured.

Note: Most of the Google web services are SOAP Based.
Yahoo → RESTful
eBay and Amazon using both SOAP and Restful

Web Service Provider vs WebService Consumer:

- The application which is providing web services is called Web Service Provider.
- The application which is consuming web data through web services, is called Web service consumer.

Eg: Bookmyshow app <--> Payment Gateway app
To complete our booking, bookmyshow application will communicates with payment gateway application. Hence payment gateway applications acts as webservice provider and bookmyshow application acts as web service consumer.

Django View Function to send HTML Response:

```
def employee_data_view(request):
    employee_data={'eno':100,'ename':'Sunny Leone','esal':1000,'eaddr':'Hyderabad'}
    resp='<h1>Employee No:{}<br>Employee Name:{}<br>Employee Salary:{}<br>Employee
Address:{}</h1>'.format(employee_data['eno'],employee_data['ename'],employee_data[
'esal'],employee_data['eaddr'])
    return HttpResponse(resp)
```

Django View Function to send HTTPResponse with JSON Data:

To handle json data, python provides inbuilt module: json
This module contains dumps() function to convert python dictionary to json object.(serialization)

```
def employee_data_jsonview(request):
    employee_data={'eno':100,'ename':'Sunny Leone','esal':1000,'eaddr':'Hyderabad'}
    json_data=json.dumps(employee_data)
    return HttpResponse(json_data,content_type='application/json')
```

Note: This way of sending JSON response is very old. Newer versions of Django dprovided a special class JsonResponse.

Django View Function to send JsonResponse directly:

The main advantage of JsonResponse class is it will accept python dict object directly. It is the most convinient way to provide json response.

```
from djando.http import JsonResponse
def employee_data_jsondirectview(request):
    employee_data={'eno':100,'ename':'Sunny Leone','esal':1000,'eaddr':'Hyderabad'}
    return JsonResponse(employee_data)
```

Python Application to communicate with Django Application:

From python if we want to send http request we should go for requests module.

test.py:

```
1) import requests
2) BASE_URL='http://127.0.0.1:8000/'
3) ENDPOINT='api'
4) r=requests.get(BASE_URL+ENDPOINT)
5) data=r.json()
6) print('Employee Number:',data['eno'])
7) print('Employee Name:',data['ename'])
8) print('Employee Salary:',data['esal'])
9) print('Employee Address:',data['eaddr'])
```

Note: In the above case, python application communicates with django application to get employee data. For this common language used is: Http and common message format used is JSON.

HTTPie Module:

We can use this module to send http request from commond prompt.
We can install as follows

pip install httpie

We can http request as follows

```
1) C:\Users\LENOVO>http http://127.0.0.1:8000
2) HTTP/1.0 200 OK
3) Content-Length: 72
4) Content-Type: application/json
5) Date: Thu, 13 Dec 2018 10:17:54 GMT
6) Server: WSGIServer/0.2 CPython/3.6.5
7) X-Frame-Options: SAMEORIGIN
8)
9) {
10)    "eaddr": "Hyderabad",
11)    "ename": "Sunny Leone",
12)    "eno": 100,
13)    "esal": 1000
14) }
```

Class Based View(CBV) to send JSON Response:

Every class based view in django should extends View class.It present in the following package.

django.views.generic

Within the class we have to provide http methods like get(),post() etc
Whenever we are sending the request, the corresponding method will be executed.

```
1) from django.views.generic import View
2) class JsonCBV(View):
3)     def get(self,request,*args,**kwargs):
4)  employee_data={'eno':100,'ename':'Sunny Leone','esal':1000,'eaddr':'Hyderabad'}
5)         return JsonResponse(employee_data)
```

urls.py

```
url(r'^cbv1/', views.JsonCBV.as_view()),
```

*args vs **kwargs:

*args → Variable Length Arguments
f1(*args) → We can call this function by passing any number of arguments.
Internally this variable-length argument will be converted into tuple.

```
1) def sum(*args):
2)     total=0
3)     for x in args:
4)         total=total+x
5)     print('The Sum:',total)
6)
7) sum()
8) sum(10)
9) sum(10,20)
10) sum(10,20,30,40)
```

Output:

```
The Sum: 0
The Sum: 10
The Sum: 30
The Sum: 100
```

**kwargs → Variable Length Keyword Arguments.
f1(**kwargs) → We can call this function by passing any number of keyword arguments.
All these keyword arguments will be converted into dictionary.

```
1) def f1(**x):
2)     print(x)
3)
4) f1(name='durga',rollno=101,marks=80,gf1='sunny')
```

Output: {'name': 'durga', 'rollno': 101, 'marks': 80, 'gf1': 'sunny'}

Mixin(Mixed In):

- Mixins are special type of inheritance in Python.
- It is limited version of Multiple inheritance.
- In multiple inheritance, we can create object for parent class and parent class can extend other classes. But in Mixins, for the parent class we cannot create object and it should be direct child class of object.i.e parent class cannot extend any other classes.
- In Multiple inheritance, parent class can contain instance variables.But in Mixins, parent class cannot contain instance variable but can contain class level static variables.
- Hence the main purpose of parent class in Mixins is to provide functions to the child classes.

Differences between Mixins and Multiple Inheritance

Mixins	Multiple Inheritance
1) Parent class instantiation is not possible	1) Parent class instantitation is possible.
2) It contains only instance methods but not instance variables.	2) It contains both instance methods and variables.
3) The methods are useful only for child classes.	3) The methods are useful for both Parent and child classes.
4) Parent class should be direct child class of object.	4) Parent class can extend any class need not be object.

Note:

1) Mixins are reusable classes in django.
2) Mixins are available only in languages which provide support for multiple inheritance like Python,Ruby,Scala etc
3) Mixins are not applicable for Java and C#, because these languages won't support multiple inheritance.

Writing CBV by using Mixin class:

mixins.py:

```
1) from django.http import JsonResponse
2) class JsonResponseMixin(object):
3)     def render_to_json_response(self,context,**kwargs):
4)         return JsonResponse(context,**kwargs)
```

CBV:

```
1) from testapp.mixins import JsonResponseMixin
2) class JsonCBV2(JsonResponseMixin,View):
3)     def get(self,request,*args,**kwargs):
```

```
4)  employee_data={'eno':100,'ename':'Sunny Leone','esal':1000,'eaddr':'Hyderabad'}
5)          return self.render_to_json_response(employee_data)
```

Performing database CRUD operations by using web api without REST Framework:

models.py

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Employee(models.Model):
5)      eno=models.IntegerField()
6)      ename=models.CharField(max_length=64)
7)      esal=models.FloatField()
8)      eaddr=models.CharField(max_length=64)
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Employee
3)  # Register your models here.
4)  class EmployeeAdmin(admin.ModelAdmin):
5)      list_display=['id','eno','ename','esal','eaddr']
6)
7)  admin.site.register(Employee,EmployeeAdmin)
```

views.py

```
1)  from django.shortcuts import render
2)  from django.views.generic import View
3)  from testapp.models import Employee
4)  import json
5)  from django.http import HttpResponse
6)  # Create your views here.
7)  class EmployeeCRUDCBV(View):
8)      def get(self,request,*args,**kwargs):
9)          emp=Employee.objects.get(id=2)
10)         data={
11)         'eno':emp.eno,
12)         'ename':emp.ename,
13)         'esal':emp.esal,
14)         'eaddr':emp.eaddr,
15)         }
```

```
16)     json_data=json.dumps(data)
17)     return HttpResponse(json_data,content_type='application/json')
```

Without Hardcoding id Value

```
1)  def get(self,request,id,*args,**kwargs):
2)      emp=Employee.objects.get(id=id)
3)    ....
```

urls.py

url(r'^api/(?P<id>\d+)/$', views.EmployeeCRUDCBV.as_view()),

test.py

```
1)  import requests
2)  BASE_URL='http://127.0.0.1:8000/'
3)  ENDPOINT='api/'
4)  n=input('Enter required id:')
5)  r=requests.get(BASE_URL+ENDPOINT+n+'/')
6)  data=r.json()
7)  print(data)
```

Serialization:

The process of converting object from one form to another form is called serialization.

Eg: converting python dictionary object to json

```
json_data = json.dumps(data)
```

Serialization by using django.core.serializers Module:

djando provides inbuilt module serializers to perform serialization very easily.This module contains serialize() function for this activity.

```
1)  def get(self,request,id,*args,**kwargs):
2)      emp=Employee.objects.get(id=id)
3)      json_data=serialize('json',[emp,],fields=('eno','ename'))
4)      return HttpResponse(json_data,content_type='application/json')
```

If we are not specifying fields attribute, then all fields will be included in json data. For security reasons, if we don't want to provide access to some fields then this fields attribute is very helpful.

Note: Here exclude attribute is not allowed

To get all Records:

```
1)  class EmployeeListCBV(View):
2)      def get(self,request,*args,**kwargs):
3)          qs=Employee.objects.all()
4)          json_data=serialize('json',qs)
5)          return HttpResponse(json_data,content_type='application/json')
```

urls.py

```
url(r'^api/$', views.EmployeeListCBV.as_view()),
```

test.py

```
1)  import requests
2)  BASE_URL='http://127.0.0.1:8000/'
3)  ENDPOINT='api/'
4)  r=requests.get(BASE_URL+ENDPOINT)
5)  data=r.json()
6)  print(data)
```

Note: In the output we are getting some extra meta information also.

```
[{'model': 'testapp.employee', 'pk': 1, 'fields': {'eno': 100, 'ename': 'Sunny', 'esal': 1000.0,
'eaddr': 'Mumbai'}}, {'model': 'testapp.employee', 'pk': 2, 'fields': {'eno': 200, 'ename':
'Bunny', 'esal': 2000.0, 'eaddr
': 'Hyderabad'}}, {'model': 'testapp.employee', 'pk': 3, 'fields': {'eno': 300, 'ename':
'Chinny', 'esal': 3000.0, 'eaddr': 'Hyderabad'}}, {'model': 'testapp.employee', 'pk': 4,
'fields': {'eno': 400, 'ename': 'Vinny', '
esal': 4000.0, 'eaddr': 'Bangalore'}}]
```

How to get only Original Database Data:

```
1)  class EmployeeListCBV(View):
2)      def get(self,request,*args,**kwargs):
3)          qs=Employee.objects.all()
4)          json_data=serialize('json',qs)
5)          pdict=json.loads(json_data)
6)          final_list=[]
7)          for obj in pdict:
8)              final_list.append(obj['fields'])
9)          json_data=json.dumps(final_list)
10)         return HttpResponse(json_data,content_type='application/json')
```

Output:

[{'eno': 100, 'ename': 'Sunny', 'esal': 1000.0, 'eaddr': 'Mumbai'}, {'eno': 200, 'ename': 'Bunny', 'esal': 2000.0, 'eaddr': 'Hyderabad'}, {'eno': 300, 'ename': 'Chinny', 'esal': 3000.0, 'eaddr': 'Hyderabad'}, {'eno': 400, 'ename': 'Vinny', 'esal': 4000.0, 'eaddr': 'Bangalore'}]

Seperate serialization Code into SerializeMixin:

mixins.py

```
1)  from django.core.serializers import serialize
2)  import json
3)  class SerializeMixin(object):
4)      def serialize(self,qs):
5)          json_data=serialize('json',qs)
6)          pdict=json.loads(json_data)
7)          final_list=[]
8)          for obj in pdict:
9)              final_list.append(obj['fields'])
10) json_data=json.dumps(final_list)
11)         return json_data
```

views.py

```
1)  class EmployeeListCBV(SerializeMixin,View):
2)      def get(self,request,*args,**kwargs):
3)          qs=Employee.objects.all()
4)          json_data=self.serialize(qs)
5)          return HttpResponse(json_data,content_type='application/json')
```

We can also use mixin to get a particular record data as follows

```
1)  class EmployeeCRUDCBV(SerializeMixin,View):
2)      def get(self,request,id,*args,**kwargs):
3)          emp=Employee.objects.get(id=id)
4)          json_data=self.serialize([emp,])
5)          return HttpResponse(json_data,content_type='application/json')
```

Output: [{'eno': 200, 'ename': 'Bunny', 'esal': 2000.0, 'eaddr': 'Hyderabad'}]

Error Handling in the API:

It is not recommended to display our django error information directly to the partner applications.Hence it is highly recommened to perform error handling.

```
1)  class EmployeeCRUDCBV(SerializeMixin,View):
2)      def get(self,request,id,*args,**kwargs):
3)          try:
4)              emp=Employee.objects.get(id=id)
5)          except Employee.DoesNotExist:
6)              json_data=json.dumps({'msg':'Specified Record Not Found'})
7)          else:
8)              json_data=self.serialize([emp,])
9)          return HttpResponse(json_data,content_type='application/json')
```

Status Codes:

Status code represents the status of HttpResponse. The following are various possible status codes.

1XX → Informational
2XX → Successful
3XX → Redirection
4XX → Client Error
5XX → Server Error

Exception Handling in Partner Application (Python Script):

```
1)  import requests
2)  BASE_URL='http://127.0.0.1:8000/'
3)  ENDPOINT='api/'
4)  r=requests.get(BASE_URL+ENDPOINT+'1/')
5)  # if r.status_code in range(200,300):
6)  if r.status_code==requests.codes.ok:
7)      data=r.json()
8)      print(data)
9)  else:
10)     print('Something goes wrong')
11)     print('Status Code:',r.status_code)
```

How to add Status Code to HttpResponse explicitly:

By using status attribute

Eg: HttpResponse(json_data,content_type='application/json',status=403)

How to render HttpResponse By using Mixin:

```
1)  from django.http import HttpResponse
2)  class HttpResponseMixin(object):
3)      def render_to_http_response(self,data,status=200):
4)          return HttpResponse(data,content_type='application/json',status=status)
```

views.py:

```
1)  class EmployeeCRUDCBV(SerializeMixin,HttpResponseMixin,View):
2)      def get(self,request,id,*args,**kwargs):
3)          try:
4)              emp=Employee.objects.get(id=id)
5)          except Employee.DoesNotExist:
6)              json_data=json.dumps({'msg':'Specified Record Not Available'})
7)              return self.render_to_http_response(json_data,404)
8)          else:
9)              json_data=self.serialize([emp,])
10)             return self.render_to_http_response(json_data)
```

How to use dumpdata Option:

We can dump our database data either to the console or to the file by using dumpdata option. This option provides support for json and xml formats. The default format is json. We can write this data to files also.

Commands:

1) **py manage.py dumpdata testapp.Employee**
   Print data to the console in json format without identation
2) **py manage.py dumpdata testapp.Employee --indent 4**
   Print data to the console in json format with identation
3) **py manage.py dumpdata testapp.Employee >emp.json --indent 4**
   Write data to emp.json file instead of displaying to the console
4) **py manage.py dumpdata testapp.Employee --format json >emp.json --indent 4**
   We are specifying format as json explicitly
5) **py manage.py dumpdata testapp.Employee --format xml --indent 4**
   Print data to the console in xml format with identation
6) **py manage.py dumpdata testapp.Employee --format xml > emp.xml --indent 4**
   Write data to emp.xml file instead of displaying to the console

Note: dumpdata option provides support only for 3 formats

1) **json(default)**
2) **XML**
3) **YAML (YAML Ain't Markup Language)**

YAML is a human readable data serialization language, which is supported by multiple languages.

How to Create Resource from partner application by using API (POST Request):

test.py(Partner Application)

```
1)  import json
2)  import requests
3)  BASE_URL='http://127.0.0.1:8000/'
4)  ENDPOINT='api/'
5)  def create_resource():
6)      new_emp={
7)      'eno':600,
8)      'ename':'Shiva',
9)      'esal':6000,
10)     'eaddr':'Chennai',
11)     }
12)     r=requests.post(BASE_URL+ENDPOINT,data=json.dumps(new_emp))
13)     print(r.status_code)
14)     print(r.text)
15)     print(r.json())
16) create_resource()
```

Note: For POST Requests, compulsory CSRF verification should be done.If it fails our request will be aborted.

Error:

```
<h1>Forbidden <span>(403)</span></h1>
<p>CSRF verification failed. Request aborted.</p>
```

How to disable CSRF Verification:

- Just for our testing purposes we can disable CSRF verification, but not recommended in production environment.
- We can disable CSRF verification at Function level, class level or at project level

1) To disable at Function/Method Level:

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
  def my_view(request):
  body
```

This approach is helpful for Function Based Views(FBVs)

2) To disable at class level:

If we disable at class level then it is applicable for all methods present inside that class. This approach is helpful for class based views(CBVs).

Code:

```
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator

@method_decorator(csrf_exempt,name='dispatch')
class EmployeeListCBV(SerializeMixin,View):
```

3) To disable at Project globally:

Inside settings.py file comment the following middleware
'django.middleware.csrf.CsrfViewMiddleware'

post() Method Logic:

- Inside post method we can access data sent by partner application by using request.body.
- First we have to check whether this data is json or not.

How to Check Data is json OR not?

utils.py:

```
1)  import json
2)  def is_json(data):
3)      try:
4)          real_data=json.loads(data)
5)          valid=True
6)      except ValueError:
7)          valid=False
8)      return valid
```

views.py:

```
1)  from testapp.utils import is_json
2)  ...
3)  @method_decorator(csrf_exempt,name='dispatch')
4)  class EmployeeListCBV(HttpResponseMixin,SerializeMixin,View):
5)
6)      def post(self,request,*args,**kwargs):
```

```
7)        data=request.body
8)        if not is_json(data):
9)            return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
10)       json_data=json.dumps({'msg':'post method'})
11) return self.render_to_http_response(json_data)
```

Creating Model Based Form to hold Employee Data:

forms.py

```
1)  from django import forms
2)  from testapp.models import Employee
3)  class EmployeeForm(forms.ModelForm):
4)      class Meta:
5)          model=Employee
6)          fields='__all__'
```

To validate emp Salary:

```
1)  from django import forms
2)  from testapp.models import Employee
3)  class EmployeeForm(forms.ModelForm):
4)      #validations
5)      def clean_esal(self):
6)          inputsal=self.cleaned_data['esal']
7)          if inputsal < 5000:
8)              raise forms.ValidationError('The minimum salary should be 5000')
9)          return inputsal
10)
11)     class Meta:
12)         model=Employee
13)         fields='__all__'
```

views.py

```
1)  from django.views.decorators.csrf import csrf_exempt
2)  from django.utils.decorators import method_decorator
3)  from testapp.utils import is_json
4)  from testapp.forms import EmployeeForm
5)
6)  @method_decorator(csrf_exempt,name='dispatch')
7)  class EmployeeListCBV(HttpResponseMixin,SerializeMixin,View):
8)      def get(self,request,*args,**kwargs):
```

```
        qs=Employee.objects.all()
        json_data=self.serialize(qs)
        return HttpResponse(json_data,content_type='application/json')
    def post(self,request,*args,**kwargs):
        data=request.body
        if not is_json(data):
            return self.render_to_http_response(json.dumps({'msg':'plz send valid json data only'}),status=400)
        empdata=json.loads(request.body)
        form=EmployeeForm(empdata)
        if form.is_valid():
            obj = form.save(commit=True)
            return self.render_to_http_response(json.dumps({'msg':'resource created successfully'}))
        if form.errors:
            json_data=json.dumps(form.errors)
            return self.render_to_http_response(json_data,status=400)
```

Performing Update Operation (put() Method):

Partner Application(test.py)

```
import requests
import json
BASE_URL='http://127.0.0.1:8000/'
ENDPOINT='api/'
def update_resource():
    new_data={
    eaddr:'Ameerpet',
    }
    r=requests.put(BASE_URL+ENDPOINT+'8/',data=json.dumps(new_data))
    print(r.status_code)
    # print(r.text)
    print(r.json())
update_resource()
```

views.py

```
@method_decorator(csrf_exempt,name='dispatch')
class EmployeeCRUDCBV(SerializeMixin,HttpResponseMixin,View):
    def get_object_by_id(self,id):
        try:
            emp=Employee.objects.get(id=id)
        except Employee.DoesNotExist:
```

```
7)          emp=None
8)       return emp
9)    def put(self,request,id,*args,**kwargs):
10)      obj=self.get_object_by_id(id)
11)      if obj is None:
12)         json_data=json.dumps({'msg':'No matched record found, Not possible to per
    form updataion'})
13)         return self.render_to_http_response(json_data,status=404)
14)      data=request.body
15)      if not is_json(data):
16)         return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
17)      new_data=json.loads(data)
18)      old_data={
19)      'eno':obj.eno,
20)      'ename':obj.ename,
21)      'esal':obj.esal,
22)      'eaddr':obj.eaddr,
23)      }
24)      for k,v in new_data.items():
25)         old_data[k]=v
26)      form=EmployeeForm(old_data,instance=obj)
27)      if form.is_valid():
28)         form.save(commit=True)
29)         json_data=json.dumps({'msg':'Updated successfully'})
30)         return self.render_to_http_response(json_data,status=201)
31)      if form.errors:
32)         json_data=json.dumps(form.errors)
33)         return self.render_to_http_response(json_data,status=400)
```

Note:

1) form = EmployeeForm(old_data)
   form.save(commit=True)
   The above code will create a new record

2) form = EmployeeForm(old_data,instance=obj)
   form.save(commit=True)
   The above code will perform updations to existing object instead of creating new object.

Performing Delete Operation:

partner application(test.py)

```
1)  import requests
2)  import json
3)  BASE_URL='http://127.0.0.1:8000/'
4)  ENDPOINT='api/'
5)  def delete_resource():
6)      r=requests.delete(BASE_URL+ENDPOINT+'9/')
7)      print(r.status_code)
8)      # print(r.text)
9)      print(r.json())
10) delete_resource()
```

views.py

```
1)  def delete(self,request,id,*args,**kwargs):
2)          obj=self.get_object_by_id(id)
3)          if obj is None:
4)              json_data=json.dumps({'msg':'No matched record found, Not possible to per
    form deletion'})
5)              return self.render_to_http_response(json_data,status=404)
6)          status,deleted_item=obj.delete()
7)  if status==1:
8)              json_data=json.dumps({'msg':'Resource Deleted successfully'})
9)              return self.render_to_http_response(json_data,status=201)
10)         json_data=json.dumps({'msg':'unable to delete ...plz try again'})
11)         return self.render_to_http_response(json_data,status=500)
```

Note:

1) obj.delete() returns a tuple with 2 values.
2) The first value represents the status of the delete. If the deletion is success then its value will be 1.
3) The secord value represents the deleted object.

Problem with Our Own Web API Framework:

The following are endpoints for CRUD operations in the above application

1) To get a particular resource: api/id/
2) To get all resources : api/
3) To create(post) a resource: api/
4) To update a resource: api/id
5) To delete a resource: api/id

According industry standard, in all rest api frameworks, endpoint should be same for all CRUD operations. In our application we are violating this rule.

*****For all CRUD operations, ENDPOINT should be same**

Rewriting Total Application to satisfy Single ENDPOINT Rule:

Partner Application to send get() Request:

1) We may pass or may not pass id value.
2) If we are not passing id, then we have to get all records.
3) If we are passing id, then we have to get only that particular record.

```
1)  import requests
2)  import json
3)  BASE_URL='http://127.0.0.1:8000/'
4)  ENDPOINT='api/'
5)  def get_resources(id=None):
6)      data={}
7)      if id is not None:
8)          data={
9)          'id':id
10)         }
11)     resp=requests.get(BASE_URL+ENDPOINT,data=json.dumps(data))
12)     print(resp.status_code)
13)     print(resp.json())
```

views.py

```
1)  class EmployeeCRUDCBV(HttpResponseMixin,SerializeMixin,View):
2)      def get_object_by_id(self,id):
3)          try:
4)              emp=Employee.objects.get(id=id)
5)          except Employee.DoesNotExist:
6)              emp=None
7)          return emp
8)  def get(self,request,*args,**kwargs):
9)          data=request.body
10)         if not is_json(data):
11)             return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
12)         data=json.loads(request.body)
13)         id=data.get('id',None)
14)         if id is not None:
15)             obj=self.get_object_by_id(id)
```

```
16)         if obj is None:
17) return self.render_to_http_response(json.dumps({'msg':'No Matched Record
    Found with Specified Id'}),status=404)
18)         json_data=self.serialize([obj,])
19)         return self.render_to_http_response(json_data)
20)     qs=Employee.objects.all()
21)     json_data=self.serialize(qs)
22)     return self.render_to_http_response(json_data)
```

To Create a Resource (post() Method):

Same code of the previous post() method

test.py

```
1)  def create_resource():
2)      new_emp={
3)      'eno':2000,
4)      'ename':'Katrina',
5)      'esal':20000,
6)      'eaddr':'Mumbai',
7)      }
8)      r=requests.post(BASE_URL+ENDPOINT,data=json.dumps(new_emp))
9)      print(r.status_code)
10)     # print(r.text)
11)     print(r.json())
12) create_resource()
```

views.py

```
1)  def post(self,request,*args,**kwargs):
2)          data=request.body
3)          if not is_json(data):
4)              return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
5)          empdata=json.loads(request.body)
6)          form=EmployeeForm(empdata)
7)          if form.is_valid():
8)              obj = form.save(commit=True)
9)              return self.render_to_http_response(json.dumps({'msg':'resource created su
    ccessfully'}))
10)         if form.errors:
11)             json_data=json.dumps(form.errors)
12)             return self.render_to_http_response(json_data,status=400)
```

Update Resource (put Method):

test.py

```
1)  def update_resource(id):
2)      new_data={
3)      'id':id,
4)      'eno':7777,
5)      'ename':'Kareena',
6)      'eaddr':'Lanka',
7)      'esal':15000
8)      }
9)      r=requests.put(BASE_URL+ENDPOINT,data=json.dumps(new_data))
10)     print(r.status_code)
11)     # print(r.text)
12)     print(r.json())
13) update_resource(14)
```

views.py

```
1)  def put(self,request,*args,**kwargs):
2)          data=request.body
3)          if not is_json(data):
4)              return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
5)          data=json.loads(request.body)
6)          id=data.get('id',None)
7)          if id is None:
8)              return self.render_to_http_response(json.dumps({'msg':'To perform updatio
    n id is mandatory,you should provide'}),status=400)
9)          obj=self.get_object_by_id(id)
10)         if obj is None:
11)             json_data=json.dumps({'msg':'No matched record found, Not possible to per
    form updataion'})
12)             return self.render_to_http_response(json_data,status=404)
13)         new_data=data
14)         old_data={
15)         'eno':obj.eno,
16)         'ename':obj.ename,
17)         'esal':obj.esal,
18)         'eaddr':obj.eaddr,
19)         }
20)         # for k,v in new_data.items():
21)         #     old_data[k]=v
```

```
22)       old_data.update(new_data)
23)       form=EmployeeForm(old_data,instance=obj)
24)       if form.is_valid():
25)          form.save(commit=True)
26)          json_data=json.dumps({'msg':'Updated successfully'})
27)          return self.render_to_http_response(json_data,status=201)
28)       if form.errors:
29)          json_data=json.dumps(form.errors)
30)          return self.render_to_http_response(json_data,status=400)
```

To Delete a Resource (delete Method):

test.py

```
1)  def delete_resource(id):
2)     data={
3)     'id':id,
4)     }
5)     r=requests.delete(BASE_URL+ENDPOINT,data=json.dumps(data))
6)     print(r.status_code)
7)     # print(r.text)
8)     print(r.json())
9)  delete_resource(13)
```

views.py

```
1)  def delete(self,request,*args,**kwargs):
2)        data=request.body
3)        if not is_json(data):
4)           return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
5)        data=json.loads(request.body)
6)        id=data.get('id',None)
7)        if id is None:
8)           return self.render_to_http_response(json.dumps({'msg':'To perform delete,
    id is mandatory,you should provide'}),status=400)
9)        obj=self.get_object_by_id(id)
10)       if obj is None:
11)          json_data=json.dumps({'msg':'No matched record found, Not possible to per
    form delete operation'})
12)          return self.render_to_http_response(json_data,status=404)
13)       status,deleted_item=obj.delete()
14)       if status==1:
15)          json_data=json.dumps({'msg':'Resource Deleted successfully'})
```

```
16)         return self.render_to_http_response(json_data,status=201)
17)       json_data=json.dumps({'msg':'unable to delete ...plz try again'})
18)       return self.render_to_http_response(json_data,status=500)
```

bit ly link : https://bit.ly/2R8HDeV

Complete Application with Single ENDPOINT for all CRUD Operations:

models.py

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Employee(models.Model):
5)      eno=models.IntegerField()
6)      ename=models.CharField(max_length=64)
7)      esal=models.FloatField()
8)      eaddr=models.CharField(max_length=64)
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Employee
3)  # Register your models here.
4)  class EmployeeAdmin(admin.ModelAdmin):
5)      list_display=['id','eno','ename','esal','eaddr']
6)
7)  admin.site.register(Employee,EmployeeAdmin)
```

forms.py

```
1)  from django import forms
2)  from testapp.models import Employee
3)  class EmployeeForm(forms.ModelForm):
4)      #validations
5)      def clean_esal(self):
6)          inputsal=self.cleaned_data['esal']
7)          if inputsal < 5000:
8)              raise forms.ValidationError('The minimum salary should be 5000')
9)          return inputsal
10)
11)     class Meta:
12)         model=Employee
13)         fields='__all__'
```

mixins.py

```
1)  from django.core.serializers import serialize
2)  import json
3)  class SerializeMixin(object):
4)      def serialize(self,qs):
5)          json_data=serialize('json',qs)
6)          pdict=json.loads(json_data)
7)          final_list=[]
8)          for obj in pdict:
9)              final_list.append(obj['fields'])
10)         json_data=json.dumps(final_list)
11)         return json_data
12)
13) from django.http import HttpResponse
14) class HttpResponseMixin(object):
15)     def render_to_http_response(self,data,status=200):
16)         return HttpResponse(data,content_type='application/json',status=status)
```

utils.py

```
1)  import json
2)  def is_json(data):
3)      try:
4)          real_data=json.loads(data)
5)          valid=True
6)      except ValueError:
7)          valid=False
8)      return valid
```

urls.py

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from testapp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^api/$', views.EmployeeCRUDCBV.as_view()),
8)      # url(r'^api/$', views.EmployeeListCBV.as_view()),
9)  ]
```

views.py

```
1)  from django.shortcuts import render
```

```
2)  from django.views.generic import View
3)  from testapp.models import Employee
4)  import json
5)  from django.http import HttpResponse
6)  from django.core.serializers import serialize
7)  from testapp.mixins import SerializeMixin,HttpResponseMixin
8)  from django.views.decorators.csrf import csrf_exempt
9)  from django.utils.decorators import method_decorator
10) from testapp.utils import is_json
11) from testapp.forms import EmployeeForm
12)
13) @method_decorator(csrf_exempt,name='dispatch')
14) class EmployeeCRUDCBV(HttpResponseMixin,SerializeMixin,View):
15)    def get_object_by_id(self,id):
16)       try:
17)          emp=Employee.objects.get(id=id)
18)       except Employee.DoesNotExist:
19)          emp=None
20)       return emp
21)
22)    def get(self,request,*args,**kwargs):
23)       data=request.body
24)       if not is_json(data):
25)          return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
26)       data=json.loads(request.body)
27)       id=data.get('id',None)
28)       if id is not None:
29)          obj=self.get_object_by_id(id)
30)          if obj is None:
31)             return self.render_to_http_response(json.dumps({'msg':'No Matched Rec
    ord Found with Specified Id'}),status=404)
32)          json_data=self.serialize([obj,])
33)          return self.render_to_http_response(json_data)
34)       qs=Employee.objects.all()
35)       json_data=self.serialize(qs)
36)       return self.render_to_http_response(json_data)
37)
38)    def post(self,request,*args,**kwargs):
39)       data=request.body
40)       if not is_json(data):
41)          return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
42)       empdata=json.loads(request.body)
43)       form=EmployeeForm(empdata)
```

```
44)       if form.is_valid():
45)          obj = form.save(commit=True)
46)          return self.render_to_http_response(json.dumps({'msg':'resource created su
    ccessfully'}))
47)       if form.errors:
48)          json_data=json.dumps(form.errors)
49)          return self.render_to_http_response(json_data,status=400)
50)    def put(self,request,*args,**kwargs):
51)       data=request.body
52)       if not is_json(data):
53)          return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
54)       data=json.loads(request.body)
55)       id=data.get('id',None)
56)       if id is None:
57)          return self.render_to_http_response(json.dumps({'msg':'To perform updatio
    n id is mandatory,you should provide'}),status=400)
58)       obj=self.get_object_by_id(id)
59)       if obj is None:
60)          json_data=json.dumps({'msg':'No matched record found, Not possible to per
    form updataion'})
61)          return self.render_to_http_response(json_data,status=404)
62)
63)       new_data=data
64)       old_data={
65)       'eno':obj.eno,
66)       'ename':obj.ename,
67)       'esal':obj.esal,
68)       'eaddr':obj.eaddr,
69)       }
70)       # for k,v in new_data.items():
71)       #    old_data[k]=v
72)       old_data.update(new_data)
73)       form=EmployeeForm(old_data,instance=obj)
74)       if form.is_valid():
75)          form.save(commit=True)
76)          json_data=json.dumps({'msg':'Updated successfully'})
77)          return self.render_to_http_response(json_data,status=201)
78)       if form.errors:
79)          json_data=json.dumps(form.errors)
80)          return self.render_to_http_response(json_data,status=400)
81)    def delete(self,request,*args,**kwargs):
82)       data=request.body
83)       if not is_json(data):
```

```
84)         return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
85)       data=json.loads(request.body)
86)       id=data.get('id',None)
87)       if id is None:
88)         return self.render_to_http_response(json.dumps({'msg':'To perform delete,
    id is mandatory,you should provide'}),status=400)
89)       obj=self.get_object_by_id(id)
90)       if obj is None:
91)         json_data=json.dumps({'msg':'No matched record found, Not possible to per
    form delete operation'})
92)         return self.render_to_http_response(json_data,status=404)
93)       status,deleted_item=obj.delete()
94)       if status==1:
95)         json_data=json.dumps({'msg':'Resource Deleted successfully'})
96)         return self.render_to_http_response(json_data,status=201)
97)       json_data=json.dumps({'msg':'unable to delete ...plz try again'})
98)       return self.render_to_http_response(json_data,status=500)
```

test.py

```
1)  import requests
2)  import json
3)  BASE_URL='http://127.0.0.1:8000/'
4)  ENDPOINT='api/'
5)  def get_resources(id=None):
6)     data={}
7)     if id is not None:
8)        data={
9)        'id':id
10)       }
11)    resp=requests.get(BASE_URL+ENDPOINT,data=json.dumps(data))
12)    print(resp.status_code)
13)    print(resp.json())
14) def create_resource():
15)    new_emp={
16)    'eno':2000,
17)    'ename':'Katrina',
18)    'esal':20000,
19)    'eaddr':'Mumbai',
20)    }
21)    r=requests.post(BASE_URL+ENDPOINT,data=json.dumps(new_emp))
22)    print(r.status_code)
23)    # print(r.text)
24)    print(r.json())
```

```
25) create_resource()
26) def update_resource(id):
27)     new_data={
28)     'id':id,
29)     'eno':7777,
30)     'ename':'Kareena',
31)     'eaddr':'Lanka',
32)     'esal':15000
33)     }
34)     r=requests.put(BASE_URL+ENDPOINT,data=json.dumps(new_data))
35)     print(r.status_code)
36)     # print(r.text)
37)     print(r.json())
38) def delete_resource(id):
39)     data={
40)     'id':id,
41)     }
42)     r=requests.delete(BASE_URL+ENDPOINT,data=json.dumps(data))
43)     print(r.status_code)
44)     # print(r.text)
45)     print(r.json())
```

Complete Application-2 with Single ENDPOINT for all CRUD Operations:

models.py

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Student(models.Model):
5)      name=models.CharField(max_length=64)
6)      rollno=models.IntegerField()
7)      marks=models.IntegerField()
8)      gf=models.CharField(max_length=64)
9)      bf=models.CharField(max_length=64)
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Student
3)
4)  # Register your models here.
5)  class StudentAdmin(admin.ModelAdmin):
6)      list_display=['id','name','rollno','marks','gf','bf']
7)
```

```
8)  admin.site.register(Student,StudentAdmin)
```

mixins.py

```
1)  from django.http import HttpResponse
2)  class HttpResponseMixin(object):
3)      def render_to_http_response(self,data,status=200):
4)          return HttpResponse(data,content_type='application/json',status=status)
5)
6)  from django.core.serializers import serialize
7)  import json
8)  class SerializeMixin(object):
9)      def serialize(self,qs):
10)         json_data=serialize('json',qs)
11)         pdict=json.loads(json_data)
12)         final_list=[]
13)         for obj in pdict:
14)             final_list.append(obj['fields'])
15)         json_data=json.dumps(final_list)
16)         return json_data
```

utils.py

```
1)  import json
2)  def is_json(data):
3)      try:
4)          real_data=json.loads(data)
5)          valid=True
6)      except ValueError:
7)          valid=False
8)      return valid
```

forms.py

```
1)  from testapp.models import Student
2)  from django import forms
3)  class StudentForm(forms.ModelForm):
4)      def clean_marks(self):
5)          inputmarks=self.cleaned_data['marks']
6)          if inputmarks < 35:
7)              raise forms.ValidationError('Marks should be >= 35')
8)          return inputmarks
9)      class Meta:
10)         model=Student
11)         fields='__all__'
```

urls.py

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from testapp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^api/', views.StudentCRUDCBV.as_view()),
8)  ]
```

views.py

```
1)  from django.shortcuts import render
2)  from django.views.generic import View
3)  from testapp.utils import is_json
4)  from testapp.mixins import HttpResponseMixin,SerializeMixin
5)  import json
6)  from testapp.models import Student
7)  from testapp.forms import StudentForm
8)  from django.views.decorators.csrf import csrf_exempt
9)  from django.utils.decorators import method_decorator
10)
11) # Create your views here.
12) @method_decorator(csrf_exempt,name='dispatch')
13) class StudentCRUDCBV(SerializeMixin,HttpResponseMixin,View):
14)     def get_object_by_id(self,id):
15)         try:
16)             s=Student.objects.get(id=id)
17)         except Student.DoesNotExist:
18)             s=None
19)         return s
20)
21)     def get(self,request,*args,**kwargs):
22)         data=request.body
23)         valid_json=is_json(data)
24)         if not valid_json:
25)             return self.render_to_http_response(json.dumps({'msg':'please send valid js
    on only'}),status=400)
26)         pdata=json.loads(data)
27)         id=pdata.get('id',None)
28)         if id is not None:
29)             std=self.get_object_by_id(id)
30)             if std is None:
```

```
31)             return self.render_to_http_response(json.dumps({'msg':'No Matched Res
    ource Found for the given id'}),status=400)
32)         json_data=self.serialize([std,])
33)         return self.render_to_http_response(json_data)
34)     qs=Student.objects.all()
35)     json_data=self.serialize(qs)
36)     return self.render_to_http_response(json_data)
37)   def post(self,request,*args,**kwargs):
38)     data=request.body
39)     valid_json=is_json(data)
40)     if not valid_json:
41)         return self.render_to_http_response(json.dumps({'msg':'please send valid js
    on only'}),status=400)
42)     std_data=json.loads(data)
43)     form=StudentForm(std_data)
44)     if form.is_valid():
45)         form.save(commit=True)
46)         return self.render_to_http_response(json.dumps({'msg':'Resource Created S
    uccessfully'}))
47)     if form.errors:
48)         json_data=json.dumps(form.errors)
49)         return self.render_to_http_response(json_data,status=400)
50)   def put(self,request,*args,**kwargs):
51)     data=request.body
52)     valid_json=is_json(data)
53)     if not valid_json:
54)         return self.render_to_http_response(json.dumps({'msg':'please send valid js
    on only'}),status=400)
55)     provided_data=json.loads(data)
56)     id=provided_data.get('id',None)
57)     if id is None:
58)         return self.render_to_http_response(json.dumps({'msg':'To perform updatio
    n id is mandatory,plz provide id'}),status=400)
59)     std=self.get_object_by_id(id)
60)     original_data={
61)     'name':std.name,
62)     'rollno':std.rollno,
63)     'marks':std.marks,
64)     'gf':std.gf,
65)     'bf':std.bf
66)     }
67)     original_data.update(provided_data)
68)     form=StudentForm(original_data,instance=std)
69)     if form.is_valid():
70)         form.save(commit=True)
```

```
71)         return self.render_to_http_response(json.dumps({'msg':'Resource Updated
    Successfully'}))
72)       if form.errors:
73)         json_data=json.dumps(form.errors)
74)         return self.render_to_http_response(json_data,status=400)
75)
76)     def delete(self,request,*args,**kwargs):
77)       data=request.body
78)       if not is_json(data):
79)         return self.render_to_http_response(json.dumps({'msg':'plz send valid json
    data only'}),status=400)
80)       data=json.loads(request.body)
81)       id=data.get('id',None)
82)       if id is None:
83)         return self.render_to_http_response(json.dumps({'msg':'To perform delete,
    id is mandatory,you should provide'}),status=400)
84)       obj=self.get_object_by_id(id)
85)       if obj is None:
86)         json_data=json.dumps({'msg':'No matched record found, Not possible to per
    form delete operation'})
87)         return self.render_to_http_response(json_data,status=404)
88)       status,deleted_item=obj.delete()
89)       if status==1:
90)         json_data=json.dumps({'msg':'Resource Deleted successfully'})
91)         return self.render_to_http_response(json_data)
92)       json_data=json.dumps({'msg':'unable to delete ...plz try again'})
93)       return self.render_to_http_response(json_data,status=500)
```

test.py

```
1)  import requests
2)  import json
3)  BASE_URL='http://127.0.0.1:8000/'
4)  ENDPOINT='api/'
5)  def get_resources(id=None):
6)      data={}
7)      if id is not None:
8)          data={
9)          'id':id
10)         }
11)     resp=requests.get(BASE_URL+ENDPOINT,data=json.dumps(data))
12)     print(resp.status_code)
13)     print(resp.json())
14) #get_resources()
15) def create_resource():
```


```
16)    new_std={
17)    'name':'Dhoni',
18)    'rollno':105,
19)    'marks':32,
20)    'gf':'Deepika',
21)    'bf':'Yuvraj'
22)    }
23)    r=requests.post(BASE_URL+ENDPOINT,data=json.dumps(new_std))
24)    print(r.status_code)
25)    # print(r.text)
26)    print(r.json())
27) # create_resource()
28) def update_resource(id):
29)    new_data={
30)    'id':id,
31)    'gf':'Sakshi',
32)
33)    }
34)    r=requests.put(BASE_URL+ENDPOINT,data=json.dumps(new_data))
35)    print(r.status_code)
36)    # print(r.text)
37)    print(r.json())
38) # update_resource(5)
39) def delete_resource(id):
40)    data={
41)    'id':id,
42)    }
43)    r=requests.delete(BASE_URL+ENDPOINT,data=json.dumps(data))
44)    print(r.status_code)
45)    # print(r.text)
46)    print(r.json())
47) delete_resource(5)
```

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 framework ti 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

Requirements:

REST framework requires the following:
Python (2.7, 3.4, 3.5, 3.6, 3.7)
Django (1.11, 2.0, 2.1)

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

requirements.txt:

argon2-cffi==18.3.0
attrs==18.1.0
bcrypt==3.1.4
certifi==2018.11.29
.....

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

```
1) from django.db import models
2) class Employee(models.Model):
3)     eno=models.IntegerField()
4)     ename=models.CharField(max_length=64)
5)     esal=models.FloatField()
6)     eaddr=models.CharField(max_length=64)
```

serializers.py

```
1) from rest_framework import serializers
2) class EmployeeSerializer(serializers.Serializer):
3)     eno=serializers.IntegerField()
4)     ename=serializers.CharField(max_length=64)
5)     esal=serializers.FloatField()
6)     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='Durga',esal=1000,eaddr='Hyd')
>>> eserializer=EmployeeSerializer(emp)
>>> eserializer.data
{'eno': 100, 'ename': 'Durga', 'esal': 1000.0, 'eaddr': 'Hyd'}
```

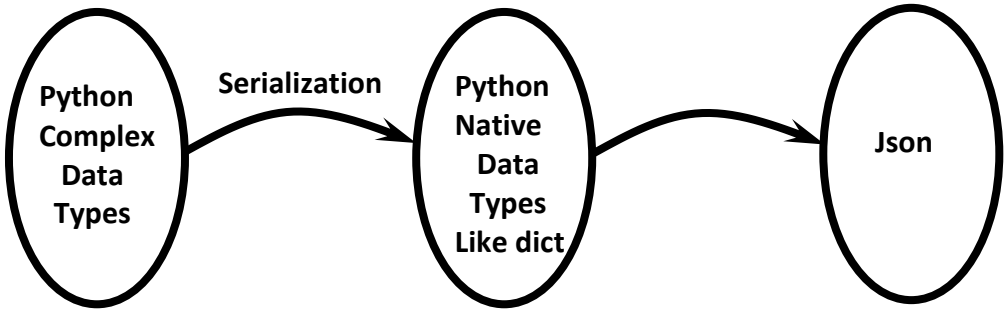
Just we converted Employee object to python native data type(dict)

Converting Python native data type to JSON:

```
>>> from rest_framework.renderers import JSONRenderer
>>> json_data=JSONRenderer().render(eserializer.data)
>>> json_data
b'{"eno":100,"ename":"Durga","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', 'Durga'), ('esal', 1000.0), ('eaddr', 'Hyderabad')]),
OrderedDict([('eno', 200), ('ename', 'Bunny'), ('esal', 2000.0), ('eaddr', 'Mumbai')])]
>>> json_data=JSONRenderer().render(eserializer.data)
>>> json_data
b'[{"eno":100,"ename":"Durga","esal":1000.0,"eaddr":"Hyderabad"},{"eno":200,"ename
":"Bunny","esal":2000.0,"ead
dr":"Mumbai"}]'
```



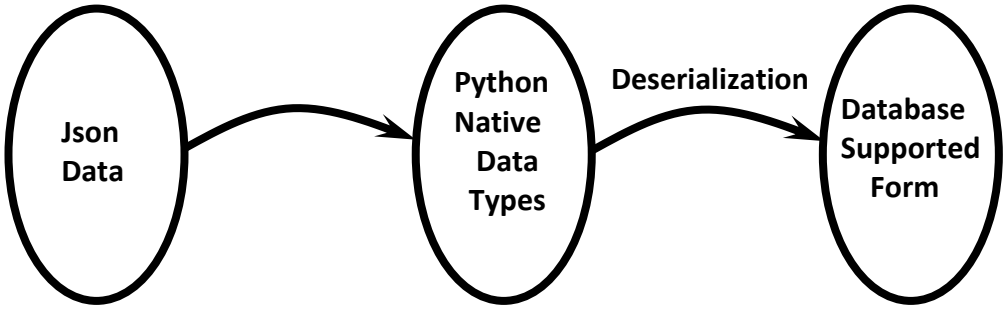
2) Deserialization:

- 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.

```
1) import io
2) from rest_framework.parsers import JSONParser
3) stream=io.BytesIO(json_data)
4) data=JSONParser().parse(stream)
```

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

```
1) serializer=EmployeeSerializer(data=data)
2) serializer.is_valid()
3) 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

```
1) from django.db import models
2)
3) # Create your models here.
4) class Employee(models.Model):
5)     eno=models.IntegerField()
6)     ename=models.CharField(max_length=64)
7)     esal=models.FloatField()
8)     eaddr=models.CharField(max_length=64)
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Employee
3)
4)  # Register your models here.
5)  admin.site.register(Employee)
```

serializers.py

```
1)  from rest_framework import serializers
2)  class EmployeeSerializer(serializers.Serializer):
3)      eno=serializers.IntegerField()
4)      ename=serializers.CharField(max_length=64)
5)      esal=serializers.FloatField()
6)      eaddr=serializers.CharField(max_length=64)
```

views.py

```
1)  from django.shortcuts import render
2)  from django.views.generic import View
3)  import io
4)  from rest_framework.parsers import JSONParser
5)  from testapp.models import Employee
6)  from testapp.serializers import EmployeeSerializer
7)  from rest_framework.renderers import JSONRenderer
8)  from django.http import HttpResponse
9)
10) # Create your views here.
11) class EmployeeCRUDCBV(View):
12)     def get(self,request,*args,**kwargs):
13)         json_data=request.body
14)         stream=io.BytesIO(json_data)
15)         data=JSONParser().parse(stream)
16)         id=data.get('id',None)
17)         if id is not None:
18)             emp=Employee.objects.get(id=id)
19)             serializer=EmployeeSerializer(emp)
20)             json_data=JSONRenderer().render(serializer.data)
21)             return HttpResponse(json_data,content_type='application/json')
22)         qs=Employee.objects.all()
23)         serializer=EmployeeSerializer(qs,many=True)
24)         json_data=JSONRenderer().render(serializer.data)
25)         return HttpResponse(json_data,content_type='application/json')
```

test.py

```
1) import requests
2) import json
3) BASE_URL='http://127.0.0.1:8000/'
4) ENDPOINT='api/'
5) def get_resources(id=None):
6)     data={}
7)     if id is not None:
8)         data={
9)         'id':id
10)        }
11)    resp=requests.get(BASE_URL+ENDPOINT,data=json.dumps(data))
12)    print(resp.status_code)
13)    print(resp.json())
14) get_resources()
```

Create Opeation/POST Request by using Serializers:

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

```
1) from rest_framework import serializers
2) from testapp.models import Employee
3) class EmployeeSerializer(serializers.Serializer):
4)     eno=serializers.IntegerField()
5)     ename=serializers.CharField(max_length=64)
6)     esal=serializers.FloatField()
7)     eaddr=serializers.CharField(max_length=64)
8)     def create(self,validated_data):
9)         return Employee.objects.create(**validated_data)
```

views.py(post method):

```
1) def post(self,request,*args,**kwargs):
2)         json_data=request.body
3)         stream=io.BytesIO(json_data)
4)         data=JSONParser().parse(stream)
5)         serializer=EmployeeSerializer(data=data)
6)         if serializer.is_valid():
7)             serializer.save()
8)             msg={'msg':'Resource Created Succesfully'}
9)             json_data=JSONRenderer().render(msg)
10)            return HttpResponse(json_data,content_type='application/json')
11) json_data=JSONRenderer().render(serializer.errors)
12)        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)

Update operation/put request by using Serializers:

To handle put requests, we have to override update() method in the serializer class.

```
1)  from rest_framework import serializers
2)  from testapp.models import Employee
3)  class EmployeeSerializer(serializers.Serializer):
4)      eno=serializers.IntegerField()
5)      ename=serializers.CharField(max_length=64)
6)      esal=serializers.FloatField()
7)      eaddr=serializers.CharField(max_length=64)
8)      def create(self,validated_data):
9)          return Employee.objects.create(**validated_data)
10)     def update(self,instance,validated_data):
11)         instance.eno=validated_data.get('eno',instance.eno)
12)         instance.ename=validated_data.get('ename',instance.ename)
13)         instance.esal=validated_data.get('esal',instance.esal)
14)         instance.eaddr=validated_data.get('eaddr',instance.eaddr)
15)         instance.save()
16)         return instance
```

views.py

```
1)  def put(self,request,*args,**kwargs):
2)          json_data=request.body
3)          stream=io.BytesIO(json_data)
4)          data=JSONParser().parse(stream)
5)          id=data.get('id')
6)          emp=Employee.objects.get(id=id)
7)          #serializer=EmployeeSerializer(emp,data=data)
8)          serializer=EmployeeSerializer(emp,data=data,partial=True)
9)          if serializer.is_valid():
10)             serializer.save()
11)             msg={'msg':'Resource Updated Succesfully'}
12)             json_data=JSONRenderer().render(msg)
13)             return HttpResponse(json_data,content_type='application/json')
```

```
14)     json_data=JSONRenderer().render(serializer.errors)
15)     return HttpResponse(json_data,content_type='application/json')
```

Note: By default for update operation, we have to provide all fields. If any field is missing, then we will get ValidationError.

If we don't want to provide all fields, then we have to use 'partial' attribute.

```
serializer = EmployeeSerializer(emp,data=data)
```
In this case we have to provide all fields for updation

```
serializer = EmployeeSerializer(emp,data=data,partial=True)
```
In this case we have to provide only required fields but not all.

Note: By using serializers, we can perform get(),post() and put() operations. There is role of serializers in delete operation.

3) Validations by using Serializers:

We can implement validations by using the following 3 ways

1) Field Level Validations
2) Object Level Validations
3) By using validators

1) Field Level Validations

Syntax: validate_fieldname(self,value):

Eg: To check esal should be minimum 5000

```
1)  class EmployeeSerializer(serializers.Serializer):
2)      ....
3)
4)      def validate_esal(self,value):
5)          if value<5000:
6)              raise serializers.ValidationError('Employee Salaray Should be Minimum 5000')
7)          return value
```

2) Object Level Validations:

If we want to perform validations for multiple fields simultaneously then we should go for object level validations.

Eg: If ename is 'Sunny' then salary should be minimum 60000

```
1)  def validate(self,data):
2)          ename=data.get('ename')
3)          esal=data.get('esal')
4)          if ename.lower()=='sunny':
5)             if esal<60000:
6)                raise serializers.ValidationError('Sunny Salary should be minimum 60K')
7)          return data
```

Use Cases:

1) First entered pwd and re-entered pwd must be same.
2) First entered account number and re-entered account number must be same

These validations we can implement at object level.

3) Validations by using Validator Field:

```
1)  def multiples_of_1000(value):
2)     if value % 1000 != 0:
3)        raise serializers.ValidationError('Salary should be multiples of 1000s')
4)
5)  class EmployeeSerializer(serializers.Serializer):
6)     ...
7)     esal=serializers.FloatField(validators=[multiples_of_1000,])
8)  ..
```

Note: If we implement all 3 types of validations then the order of priority is

1) validations by using validator
2) validations at field level
3) validations at object level

Complete Application for Serializers:

serializers.py

```
1)  from rest_framework import serializers
2)  from testapp.models import Employee
3)
4)  def multiples_of_1000(value):
5)     print('validations by using validator')
6)     if value % 1000 != 0:
7)        raise serializers.ValidationError('Salary should be multiples of 1000s')
8)
9)  class EmployeeSerializer(serializers.Serializer):
```

```
    eno=serializers.IntegerField()
    ename=serializers.CharField(max_length=64)
    esal=serializers.FloatField(validators=[multiples_of_1000,])
    eaddr=serializers.CharField(max_length=64)

    def validate_esal(self,value):
        print('validations at field level')
        if value<5000:
            raise serializers.ValidationError('Employee Salaray Should be Minimum 5000')
        return value
    def validate(self,data):
        print('validations at object level')
        ename=data.get('ename')
        esal=data.get('esal')
        if ename.lower()=='sunny':
            if esal<60000:
                raise serializers.ValidationError('Sunny Salary should be minimum 60K')
        return data

    def create(self,validated_data):
        return Employee.objects.create(**validated_data)
    def update(self,instance,validated_data):
        instance.eno=validated_data.get('eno',instance.eno)
        instance.ename=validated_data.get('ename',instance.ename)
        instance.esal=validated_data.get('esal',instance.esal)
        instance.eaddr=validated_data.get('eaddr',instance.eaddr)
        instance.save()
        return instance
```

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
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator

@method_decorator(csrf_exempt,name='dispatch')
class EmployeeCRUDCBV(View):
def get(self,request,*args,**kwargs):
```

```
15)     json_data=request.body
16)     stream=io.BytesIO(json_data)
17)     data=JSONParser().parse(stream)
18)     id=data.get('id',None)
19)     if id is not None:
20)         emp=Employee.objects.get(id=id)
21)         serializer=EmployeeSerializer(emp)
22)         json_data=JSONRenderer().render(serializer.data)
23)         return HttpResponse(json_data,content_type='application/json')
24)     qs=Employee.objects.all()
25)     serializer=EmployeeSerializer(qs,many=True)
26)     json_data=JSONRenderer().render(serializer.data)
27)     return HttpResponse(json_data,content_type='application/json')
28)   def post(self,request,*args,**kwargs):
29)     json_data=request.body
30)     stream=io.BytesIO(json_data)
31)     data=JSONParser().parse(stream)
32)     serializer=EmployeeSerializer(data=data)
33)     if serializer.is_valid():
34)         serializer.save()
35)         msg={'msg':'Resource Created Succesfully'}
36)         json_data=JSONRenderer().render(msg)
37)         return HttpResponse(json_data,content_type='application/json')
38)     json_data=JSONRenderer().render(serializer.errors)
39)     return HttpResponse(json_data,content_type='application/json')
40)   def put(self,request,*args,**kwargs):
41)     json_data=request.body
42)     stream=io.BytesIO(json_data)
43)     data=JSONParser().parse(stream)
44)     id=data.get('id')
45)     emp=Employee.objects.get(id=id)
46)     serializer=EmployeeSerializer(emp,data=data,partial=True)
47)     if serializer.is_valid():
48)         serializer.save()
49)         msg={'msg':'Resource Updated Succesfully'}
50)         json_data=JSONRenderer().render(msg)
51)         return HttpResponse(json_data,content_type='application/json')
52)     json_data=JSONRenderer().render(serializer.errors)
53)     return HttpResponse(json_data,content_type='application/json')
```

test.py

```
1) import requests
2) import json
3) BASE_URL='http://127.0.0.1:8000/'
```

```
4)  ENDPOINT='api/'
5)  # def get_resources(id=None):
6)  #     data={}
7)  #     if id is not None:
8)  #         data={
9)  #         'id':id
10) #         }
11) #     resp=requests.get(BASE_URL+ENDPOINT,data=json.dumps(data))
12) #     print(resp.status_code)
13) #     print(resp.json())
14) # get_resources()
15) # def create_resource():
16) #     new_emp={
17) #     'eno':300,
18) #     'ename':'Kareena',
19) #     'esal':3000,
20) #     'eaddr':'Hyderabad',
21) #     }
22) #     r=requests.post(BASE_URL+ENDPOINT,data=json.dumps(new_emp))
23) #     print(r.status_code)
24) #     # print(r.text)
25) #     print(r.json())
26) # create_resource()
27) def update_resource(id):
28)     new_data={
29)     'id':id,
30)     # 'eno':700,
31)     'ename':'Sunny123',
32)     'esal':15000,
33)     # 'eaddr':'Hyd'
34)
35)     }
36)     r=requests.put(BASE_URL+ENDPOINT,data=json.dumps(new_data))
37)     print(r.status_code)
38)     # print(r.text)
39)     print(r.json())
40) update_resource(3)
41) # def delete_resource(id):
42) #     data={
43) #     'id':id,
44) #     }
45) #     r=requests.delete(BASE_URL+ENDPOINT,data=json.dumps(data))
46) #     print(r.status_code)
47) #     # print(r.text)
48) #     print(r.json())
```

```
49) # delete_resource(5)
```

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:

```
1) class EmployeeSerializer(serializers.ModelSerializer):
2)     class Meta:
3)         model=Employee
4)         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.

```
1) def multiples_of_1000(value):
2)     print('validations by using validator')
3)     if value % 1000 != 0:
4)         raise serializers.ValidationError('Salary should be multiples of 1000s')
5) class EmployeeSerializer(serializers.ModelSerializer):
6)     esal=serializers.FloatField(validators=[multiples_of_1000,])
7)     class Meta:
8)         model=Employee
9)         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:

```
1) 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 {} Wish You Happy New Year !!!'.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":"Sunny"}
response:

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
   "msg": "Hello Sunny Wish You Happy New Year !!!"
}

input:  {"name":"Sunny Leone"}
response:

```

```
16) HTTP 400 Bad Request
17) Allow: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS
18) Content-Type: application/json
19) Vary: Accept
20)
21) {
22)    "name": [
23)       "Ensure this field has no more than 7 characters."
24)    ]
25) }
```

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

```
1)  class TestApiView(APIView):
2)     .....
3)     def put(self,request,pk=None):
4)        return Response({'msg':'Response from put method'})
5)     def patch(self,request,pk=None):
6)        return Response({'msg':'Response from patch method'})
7)     def delete(self,request,pk=None):
8)        return Response({'msg':'Response from delete method'})
```

Note: Test these methods by using browsable API(include screen shots)

Complete Application:

views.py

```
1)  from django.shortcuts import render
2)  from rest_framework.views import APIView
3)  from rest_framework.response import Response
4)  from testapp.serializers import NameSerializer
5)  # Create your views here.
6)  class TestApiView(APIView):
7)     def get(self,request,format=None):
8)        colors=['RED','BLUE','GREEN','YELLOW','INDIGO']
9)        return Response({'msg':'Welcome to Colorful Year','colors':colors})
10)    def post(self,request):
11)       serializer=NameSerializer(data=request.data)
12)       if serializer.is_valid():
13)          name=serializer.data.get('name')
14)          msg='Hello {} Wish You Happy New Year !!!'.format(name)
15)          return Response({'msg':msg})
16)       return Response(serializer.errors,status=400)
```

```
17)   def put(self,request,pk=None):
18)       return Response({'msg':'Response from put method'})
19)   def patch(self,request,pk=None):
20)       return Response({'msg':'Response from patch method'})
21)   def delete(self,request,pk=None):
22)       return Response({'msg':'Response from delete 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)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^api/', views.TestApiView.as_view()),
8)  ]
```

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()
post() → create()
put() → update()
patch() → partial_update()
delete() → destroy()

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

Sample Code for List Operation:

```
1) from rest_framework import viewsets
2) class TestViewSet(viewsets.ViewSet):
3)     def list(self,request):
4)         colors=['RED','GREEN','YELLOW','ORANGE']
5)         return Response({'msg':'Wish YOu Colorful Life in 2019','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':'Wish YOu Colorful Life in 2019','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 {} Your Life will be settled in 2019'.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) ]
```

Differences between APIView and ViewSet

APIView	ViewSet
1) Present in rest_framework.views Module.	1) Present in rest_framework.viewsets Modules.
2) Method Names reflect HTTP Methods like	2) Method Names reflect Database Model class actions/operations like

get(),post(),put(),patch(),delete()	list(),retrieve(),create(),update(),partial_update() and destroy()
3) We have to map views to urls explicitly.	3) We are not required to map views to urls explicitly. DefaultRouter will takes care url maapings automatically.
4) Most of the business logic we have to write explicitly.	4) Most of the business logic will be generated automatically.
5) Length of the code is more	5) Length of the code is less.
6) API Development time is more	6) API Development time is less
7) Developer has complete control over the logic	7) Developer won't have complete control over the logic.
8) Clear Execution Flow is possible	8) Clear Execution Flow is not possible
9) Best suitable for complex operations like using multiple data sources simultaneously, calling other APIs etc	9) Best suitable for developing simple APIs like developing CRUD interface for database models.

Demo applications on APIViews:

models.py

```
1)  from django.db import models
2)
3)  # Create your models here.
4)  class Employee(models.Model):
5)      eno=models.IntegerField()
6)      ename=models.CharField(max_length=64)
7)      esal=models.FloatField()
8)      eaddr=models.CharField(max_length=64)
9)      def __str__(self):
10)         return self.ename
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Employee
3)  # Register your models here.
4)  admin.site.register(Employee)
```

serializers.py

```
1)  from testapp.models import Employee
2)  from rest_framework import serializers
3)  class EmployeeSerializer(serializers.ModelSerializer):
```

```
4)     class Meta:
5)         model=Employee
6)         fields='__all__'
```

views.py

```
1)  from django.shortcuts import render
2)  from rest_framework.views import APIView
3)  from testapp.serializers import EmployeeSerializer
4)  from rest_framework.response import Response
5)  from testapp.models import Employee
6)  # Create your views here.
7)  class EmployeeListAPIView(APIView):
8)      def get(self,request,format=None):
9)          qs=Employee.objects.all()
10)         serializer=EmployeeSerializer(qs,many=True)
11)         return Response(serializer.data)
```

Note: In the above example, serializer is responsible to convert queryset to python native data type(dict) and Response object is responsible to convert that dict to json.

urls.py

```
1)  from django.conf.urls import url
2)  from django.contrib import admin
3)  from testapp import views
4)
5)  urlpatterns = [
6)      url(r'^admin/', admin.site.urls),
7)      url(r'^api/', views.EmployeeListAPIView.as_view()),
8)  ]
```

To List all Employees by using ListAPIView Class:

If we want to get list of all resources then ListAPIView class is best suitable. This class present in rest_framework.generics module.
Used for read-only endpoints to represent a collection of model instances.
Provides a get method handler.
Extends: GenericAPIView, ListModelMixin

```
1)  from rest_framework import generics
2)  class EmployeeAPIView(generics.ListAPIView):
3)      queryset=Employee.objects.all()
4)      serializer_class=EmployeeSerializer
```

How to implement Search Operation:

If we want to implement search operation, we have to override get_queryset() method in our view class.

```
1)  from rest_framework import generics
2)  class EmployeeAPIView(generics.ListAPIView):
3)      # queryset=Employee.objects.all()
4)      serializer_class=EmployeeSerializer
5)      def get_queryset(self):
6)          qs=Employee.objects.all()
7)          name=self.request.GET.get('ename')
8)          if name is not None:
9)              qs=qs.filter(ename__icontains=name)
10)         return qs
```

Note: If we override get_queryset() method then we are not required to specify queryset variable.

To list out all employee records the endpoint url is:
http://127.0.0.1:8000/api/

To list out all employee records where ename contains Sunny, the endpoint url is:
http://127.0.0.1:8000/api/?ename=Sunny

How to implement Create Operation with CreateAPIView:

CreateAPIView:
Used for create-only endpoints.
Provides a post method handler.
Extends: GenericAPIView, CreateModelMixin

```
class EmployeeCreateAPIView(generics.CreateAPIView):
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer

url(r'^api/', views.EmployeeCreateAPIView.as_view()),
```

How to implement Retrieve Operation by using RetrieveAPIView:

RetrieveAPIView:
Used for read-only endpoints to represent a single model instance.
Provides a get method handler.
Extends: GenericAPIView, RetrieveModelMixin

```
class EmployeeDetailAPIView(generics.RetrieveAPIView):
```

```
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer

url(r'^api/(?P<pk>\d+)/$', views.EmployeeDetailAPIView.as_view()),
```

In the url pattern compulsory we should use 'pk',otherwise we will get the following error.

AssertionError at /api/2/
Expected view EmployeeDetailAPIView to be called with a URL keyword argument named "pk". Fix your URL conf, or set the '.lookup_field' attribute on the view correctly.

If we want to use anyother name instead of 'pk' then we have to use lookup_field attribute in the view class.

```
class EmployeeDetailAPIView(generics.RetrieveAPIView):
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer
    lookup_field='id'

url(r'^api/(?P<id>\d+)/$', views.EmployeeDetailAPIView.as_view()),
```

How to implement Update Operation by using UpdateAPIView:

UpdateAPIView:
Used for update-only endpoints for a single model instance.
Provides put and patch method handlers.
Extends: GenericAPIView, UpdateModelMixin

```
class EmployeeUpdateAPIView(generics.UpdateAPIView):
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer
    lookup_field='id'

url(r'^api/(?P<id>\d+)/$', views.EmployeeUpdateAPIView.as_view()),
```

Note: In the browsable API, for PUT operation we have to provide values for all fields. But for PATCH operation we have to provide only required fields.

How to implement Delete Operation by using DestroyAPIView:

DestroyAPIView:
Used for delete-only endpoints for a single model instance.
Provides a delete method handler.
Extends: GenericAPIView, DestroyModelMixin

```
class EmployeeDeleteAPIView(generics.DestroyAPIView):
```


```
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer
    lookup_field='id'

url(r'^api/(?P<id>\d+)/$', views.EmployeeDeleteAPIView.as_view()),
```

Note:

ListAPIView → Specially designed class for List operation
CreateAPIView → Specially designed class for Create operation
RetrieveAPIView → Specially designed class for Detail operation
UpdateAPIView → Specially designed class for Update operation
DestroyAPIView → Specially designed class for delete operation

How to implement List and Create Operations by using ListCreateAPIView:

We can use ListCreateAPIView to develop read-write endpoints to represent a collection of model instances.
It provides get and post method handlers.

```
class EmployeeListCreateAPIView(generics.ListCreateAPIView):
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer

url(r'^api/', views.EmployeeListCreateAPIView.as_view()),
```

How to implement Read and Update Operations by using RetrieveUpdateAPIView:

We can use RetrieveUpdateAPIView to develop read and update endpoints to represent a single model instance.
It provides get, put and patch method handlers.

```
class EmployeeRetrieveUpdateAPIView(generics.RetrieveUpdateAPIView):
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer
lookup_field='id'

url(r'^api/(?P<id>\d+)/$', views.EmployeeRetrieveUpdateAPIView.as_view()),
```

How to implement Read and Delete Operations by using RetrieveDestroyAPIView:

We can use RetrieveDestroyAPIView to develop read and delete endpoints to represent a single model instance.
It provides get and delete method handlers.

```
class EmployeeRetrieveDestroyAPIView(generics.RetrieveDestroyAPIView):
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer
    lookup_field='id'

url(r'^api/(?P<id>\d+)/$', views.EmployeeRetrieveDestroyAPIView.as_view()),
```

How to implement Read, Update and Delete Operations by using RetrieveUpdateDestroyAPIView:

We can use RetrieveUpdateDestroyAPIView to develop read-update-delete endpoints to represent a single model instance.
It provides get, put, patch and delete method handlers.

```
class EmployeeRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAPIView):
    queryset=Employee.objects.all()
    serializer_class=EmployeeSerializer
    lookup_field='id'

url(r'^api/(?P<id>\d+)/$', views.EmployeeRetrieveUpdateDestroyAPIView.as_view()),
```

Implementing all CRUD Operations by using 2 End Points:

views.py

```
1) class EmployeeListCreateAPIView(generics.ListCreateAPIView):
2)     queryset=Employee.objects.all()
3)     serializer_class=EmployeeSerializer
4) class EmployeeRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAP
   IView):
5)     queryset=Employee.objects.all()
6)     serializer_class=EmployeeSerializer
7)     lookup_field='id'
```

urls.py

```
url(r'^api/$', views.EmployeeListCreateAPIView.as_view()),
url(r'^api/(?P<id>\d+)/$', views.EmployeeRetrieveUpdateDestroyAPIView.as_view()),
```

Note: **The following are various predefined APIView classes to build our API very easily**

- **ListAPIView**
- **CreateAPIView**
- **RetrieveAPIView**
- **UpdateAPIView**

- DestroyAPIView
- ListCreateAPIView
- RetrieveUpdateAPIView
- RetrieveDestroyAPIView
- RetrieveUpdateDestroyAPIView

Mixins:

Mixins are reusable components. DRF provides several mixins to provide basic view behaviour. Mixin classes provide several action methods like create(), list() etc which are useful while implementing handling methods like get(), post() etc
The mixin classes can be imported from rest_framework.mixins.

The following are various Mixin classes provided by DRF.

1) ListModelMixin
2) CreateModelMixin
3) RetrieveModelMixin
4) UpdateModelMixin
5) DestroyModelMixin

1) ListModelMixin:

- It can be used to implement list operation (get method handler).
- It provides list() method
  list(request,*args,**kwargs)

2) CreateModelMixin:

It can be used for implementing create operation. ie for creating and saving new model instance(post method handler).It provides create() method.
create(request,*args,**kwargs)

3) RetrieveModelMixin:

It can be used to implement retrieve/detail operation(get method handler). It provides retrieve() method
retrieve(request,*args,**kwargs)

4) UpdateModelMixin:

- It can be used to implement update operation(both put and patch)
- It provides update() method to implement put method handler.
  update(request,*args,**kwargs)
- It provides partial_update() method to implement patch method handler
  partial_update(request,*args,**kwargs)

5) DestroyModelMixin:

- It can be used to implement destroy operation(delete method handler)
- It provide destroy() method

  destroy(request,*args,**kwargs)

Demo Application:

```
1) from rest_framework import mixins
2) class EmployeeListModelMixin(mixins.CreateModelMixin,generics.ListAPIView):
3)     queryset=Employee.objects.all()
4)     serializer_class=EmployeeSerializer
5)     def post(self,request,*args,**kwargs):
6)         return self.create(request,*args,**kwargs)
7)
8) class EmployeeDetailAPIViewMixin(mixins.UpdateModelMixin,mixins.DestroyMod
   elMixin,generics.RetrieveAPIView):
9)     queryset=Employee.objects.all()
10)    serializer_class=EmployeeSerializer
11)    def put(self,request,*args,**kwargs):
12)        return self.update(request,*args,**kwargs)
13)    def patch(self,request,*args,**kwargs):
14)        return self.partial_update(request,*args,**kwargs)
15)    def delete(self,request,*args,**kwargs):
16)        return self.destroy(request,*args,**kwargs)
17)
18) url(r'^api/$', views.EmployeeListModelMixin.as_view()),
19) url(r'^api/(?P<pk>\d+)/$', views.EmployeeDetailAPIViewMixin.as_view()),
```

Demo Application by using ViewSet:

models.py

```
1) from django.db import models
2)
3) # Create your models here.
4) class Employee(models.Model):
5)     eno=models.IntegerField()
6)     ename=models.CharField(max_length=64)
7)     esal=models.FloatField()
8)     eaddr=models.CharField(max_length=64)
```

admin.py

```
1) from django.contrib import admin
```

```
2)  from testapp.models import Employee
3)  # Register your models here.
4)  class EmployeeAdmin(admin.ModelAdmin):
5)      list_display=['id','eno','ename','esal','eaddr']
6)
7)  admin.site.register(Employee,EmployeeAdmin)
```

serializers.py

```
1)  from testapp.models import Employee
2)  from rest_framework.serializers import ModelSerializer
3)  class EmployeeSerializer(ModelSerializer):
4)      class Meta:
5)          model=Employee
6)          fields='__all__'
```

views.py

```
1)  from django.shortcuts import render
2)  from testapp.models import Employee
3)  from testapp.serializers import EmployeeSerializer
4)  from rest_framework.viewsets import ModelViewSet
5)  class EmployeeCRUDCBV(ModelViewSet):
6)      serializer_class=EmployeeSerializer
7)      queryset=Employee.objects.all()
```

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('api',views.EmployeeCRUDCBV,base_name='api')
7)  router.register('api',views.EmployeeCRUDCBV)
8)  urlpatterns = [
9)      url(r'^admin/', admin.site.urls),
10)     url(r'', include(router.urls)),
11) ]
```

Note:

1) If we want to develop simple APIs very quickly then ViewSet is the best choice
2) In the case of normal ViewSet, at the time of registration with router, base_name attribute is mandatory. But if we are using ModelViewSet then this base_name attribute is optional.

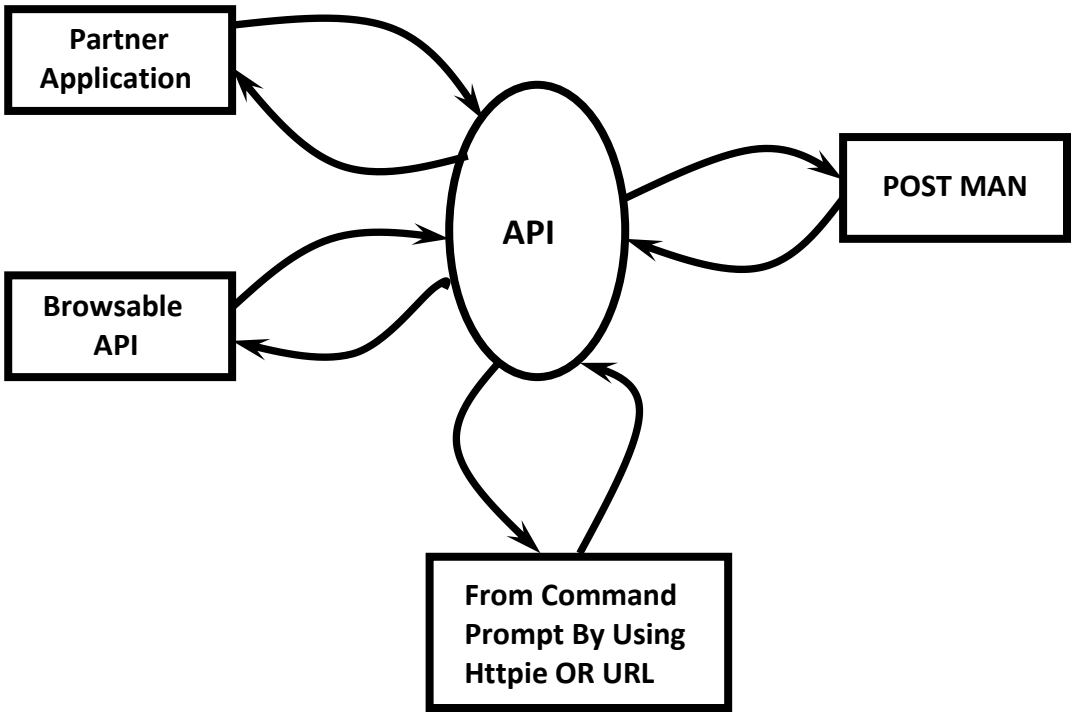
Eg:

```
router.register('api',views.EmployeeCRUDCBV,base_name='api')
router.register('api',views.EmployeeCRUDCBV)
```

API Functionality Testing by using POSTMAN:

We have to install postman software explicitly
website: getpostman.com

The 4 Ways of Testing API Functionality



1) Partner Application
2) Browsable API
3) POSTMAN

*****SCREEN SHOTS of the postman needs to be included**

Note:

1) In the case of POST, PUT and PATCH Requests,we have to provide input data as the part of Body in Postman.
2) Authentication information we have to provide in Headers part in Postman.

Authentication and Authorization:

The APIs, which are developed up to this can be accessed by every one.
By using ENDPOINT, any one can create a new resource,can modify and delete any existing resource. It causes security problems. To provide Security for our API, we should go for Authentication and Authorization.

Authentication:

The process of validating user is called authentication. Most of the times we can perform authentication by using username and password combination or by using tokens etc

DRF provides several inbuilt authentication mechanisms

1) **Basic Authentication**
2) **Session Authentication**
3) **Token Authentication**
4) **JWT(Json window Token) Authentication**
   etc

Note: By using DRF, we can implement our own custom authentication mechanism also

Authorization:

The process of validating access permissions of user is called authorization.
DRF provides several permission classes for authorization.

1) **AllowAny**
2) **IsAuthenticated**
3) **IsAdminUser**
4) **IsAuthenticatedOrReadOnly**
5) **DjangoModelPermissions**
6) **DjangoModelPermissionsOrAnonReadOnly**
   etc

Note: After authentication we have to perform authorization.

Token based Authentication:

- **Every request will be authenticated by using Token, which is nothing but Token Authentication.**
- **TokenAuthentication is best suitable for native desktop clients and mobile clients.**
- **To implement TokenAuthentication, We have to use 3rd party application 'rest_framework.authtoken',which is responsible to generate and validate required tokens. This application is the part of DRF.**

Steps to implement TokenAuthentication:

1) We have to include authtoken application in INSTALLED_APPS list inside settings.py file.

```
1) INSTALLED_APPS = [
2)     ....
3)     'rest_framework',
4)     'rest_framework.authtoken',
5)     'testapp'
6) ]
```

2) Perform migrations so that the required tables of authtoken application will be created in the database.
   The table name is Tokens.
3) We can generate Tokens in the backend from admin interface by selecting required user.
4) User also can send a request to authtoken application to generate token explicitly. For this auth application url-pattern we have to configure in urls.py file.

```
1) from rest_framework.authtoken import views
2) urlpatterns = [
3)     .....
4)     url(r'^get-api-token/', views.obtain_auth_token,name='get-api-token'),
5) ]
```

We can send request to this authtoken application to get token as follows

http POST http://127.0.0.1:8000/get-api-token/ username="sunny" password="durga123"

- authtoken application will validate this username and password.After valiation, it will check is any token already generated for this user or not. If it is already generated then return existing token from Tokens table.
- If it is not yet generated, then authtoken application will generate Token and save in Tokens table and then send that token to the client.

Note: From the postman also,we can send the request. But username and password we have to provide in Body section.

Enabling Authentication and Authorization (Permissions) for View Class:

- Upto this just we tested authtoken application to generate and store Tokens.

- We have to enable authentication and authorization for our view classes either locally OR globally.

Enabling Locally:

Our application may contain several view classes. If we want to enable authentication and authorization for a particular view class then we have to use this local approach.

```
1) from rest_framework.authentication import TokenAuthentication
2) from rest_framework.permissions import IsAuthenticated
3) class EmployeeCRUDCBV(ModelViewSet):
4)     ...
5)     authentication_classes=[TokenAuthentication,]
6)     permission_classes=[IsAuthenticated,]
```

Note: Now,if we want to access ENDPOINT compulsory we should send Token, otherwise we will get 401 Unauthorized error response.

```
1) D:\durgaclasses>http http://127.0.0.1:8000/api/
2) HTTP/1.0 401 Unauthorized
3) Allow: GET, POST, HEAD, OPTIONS
4) Content-Length: 58
5) Content-Type: application/json
6) Date: Mon, 21 Jan 2019 11:43:07 GMT
7) Server: WSGIServer/0.2 CPython/3.6.5
8) Vary: Accept
9) WWW-Authenticate: Token
10) X-Frame-Options: SAMEORIGIN
11)
12) {
13)     "detail": "Authentication credentials were not provided."
14) }
```

How to Send the Request with Token:

```
1) D:\durgaclasses>http http://localhost:8000/api/ "authorization:Token 3639020972
   202cc1d25114ab4a5f54e6078184a4"
2)
3) HTTP/1.0 200 OK
4) Allow: GET, POST, HEAD, OPTIONS
5) Content-Length: 136
6) Content-Type: application/json
7) Date: Mon, 21 Jan 2019 11:45:13 GMT
8) Server: WSGIServer/0.2 CPython/3.6.5
9) Vary: Accept
```

```
X-Frame-Options: SAMEORIGIN

[
    {
        "eaddr": "Mumbai",
        "ename": "Sunny",
        "eno": 100,
        "esal": 1000.0,
        "id": 1
    },
    {
        "eaddr": "Hyderabad",
        "ename": "Bunny",
        "eno": 200,
        "esal": 2000.0,
        "id": 2
    }
]
```

Enabling Globally:

- If we want to enable authentication and authorization for all view classes, we have to use this approach.
- We have to add the following lines inside settings.py file.

```
REST_FRAMEWORK={
'DEFAULT_AUTHENTICATION_CLASSES':('rest_framework.authentication.TokenAuthentication',),
'DEFAULT_PERMISSION_CLASSES':('rest_framework.permissions.IsAuthenticated',)
}
```

Q) What is use of 'AllowAny' Permission Class?

- AllowAny is the default value for the permission class.
- If we configure permission classes globally then applicable for all view classes. If we don't want authorization for a particular class then we should use this 'AllowAny' permission Class.

```
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import AllowAny
class EmployeeCRUDCBV(ModelViewSet):
    queryset=Employee.objects.all()
serializer_class=EmployeeSerializer
    authentication_classes=[TokenAuthentication,]
    permission_classes=[AllowAny,]
```

To access this end point now authentication and authorization is not required. Any one can access.

Various Possible Permission Classes:

DRF provides the following pre defined permission classes

1) **AllowAny**
2) **IsAuthenticated**
3) **IsAdminUser**
4) **IsAuthenticatedOrReadOnly**
5) **DjangoModelPermissions**
6) **DjangoModelPermissionsOrAnonReadOnly**

1) AllowAny:

- **The AllowAny permission class will allow unrestricted access irrespective of whether request is authenticated or not.**
- **This is default value for permission-class. It is very helpful to allow unrestricted access for a particular view class if global settings are enabled.**

2) IsAuthenticated:

- **The IsAuthenticated permission class will deny permissions to any unauthorized user. ie only authenticated users are allowed to access endpoint.**
- **This permission is suitable, if we want our API to be accessible by only registered users.**

Note:

We can send Token in postman inside Headers Section
Key: Authorization
Value: Token 3639020972202cc1d25114ab4a5f54e6078184a4

3) IsAdminUser:

- **If we use IsAdminUser permission class then only AdminUser is allowed to access.i.e the users where is_staff property is True.**
- **This type of permission is best suitable if we want our API to be accessible by only trusted administrators.**
- **If the user is not admin and if he is trying to access endpoint then we will get 403 status code error response saying:**

```
{
    "detail": "You do not have permission to perform this action."
}
```

4) IsAuthenticatedOrReadOnly:

- To perform read operation (safe methods:GET,HEAD,OPTIONS) authentication is not required. But for the remaining operations (POST,PUT,PATCH,DELETE) authentication must be required.
- If any person is allowed to perform read operation and only registered users are allowed to perform write operation then we should go for this permission class.
- Eg: In IRCTC application, to get trains information (read operation) registration is not required. But to book tickets (write operation) login must be required.

5) DjangoModelPermissions:

- This is the most powerful permission class. Authorization will be granted iff user is authenticated and has the relevant model permissions.
- DjangoModelPermissions = Authentication + Model Permissions.
- If the user is not authenticated(we are not providing token) then we will get 401 Unauthorized error message saying

```
{
    "detail": "Authentication credentials were not provided."
}
```

If we are providing Token (authenticated) but not having model permissions then we can perform only GET operation. But to perform POST,PUT,PATCH,DELETE compulsory model permissions must be required,otherwise we will get 403 Forbidden error message saying

```
{
    "detail": "You do not have permission to perform this action."
}
```

How to provide Model Permissions:

To perform POST operation the required model permission is 'add'
To perform PUT,PATCH operations the required model permission is 'change'
To pderform DELETE operation the required model permission is 'delete'

We have to provide these model permissions in admin interface under User permissions:

testapp | employee | Can change employee
testapp | employee | Can add employee
testapp | employee | Can delete employee

Note: DjangoModelPermissions class is more powerful and we have complete control on permissions.

6) DjangoModelPermissionsOrAnonReadOnly:

It is exactly same as DjangoModelPermissions class except that it allows unauthenticated users to have read-only access to the API.

```
1)  from rest_framework.viewsets import ModelViewSet
2)  from testapp.models import Employee
3)  from testapp.serializers import EmployeeSerializer
4)  from rest_framework.authentication import TokenAuthentication
5)  from rest_framework.permissions import IsAuthenticated,AllowAny,IsAdminUser,I
    sAuthenticatedOrReadOnly,DjangoModelPermissions,DjangoModelPermissionsOr
    AnonReadOnly
6)  class EmployeeCRUDCBV(ModelViewSet):
7)      queryset=Employee.objects.all()
8)      serializer_class=EmployeeSerializer
9)      authentication_classes=[TokenAuthentication,]
10)     permission_classes=[DjangoModelPermissionsOrAnonReadOnly,]
```

Custom Permissions: