Before starting with REST we will first take a overview on JSON

**JSON**

SON (JavaScript Object Notation) is a popular data format used for representing structured data. It's commonly used in transmit and receive data between a server and web application in JSON format

**NOTE:**

* JSON data looks like python dictionary data types, JSON data are always kept in double quote but dictionary can be in single or double quote.
* We can differentiate JSON data python dictionary using type()

**Json data types:**

In JSON, values must be one of the following data types

1. String
2. Number
3. Object (JSON object)
4. Array Boolean
5. null

**Json String**

Strings in JSON must be written in double quotes.

{ "name":"John" }

**Number**

Numbers in JSON must be an integer or a floating point.

{ "age":30 }

**JSON Object**

A JSON object contains data in the form of key/value pair. The keys are strings and the values are the JSON types. Keys and values are separated by colon. Each entry (key/value pair) is separated by comma.

*The { (curly brace) represents the JSON object*

*{*

*"employee": {*

*"name": "sonoo",*

*"salary": 56000,*

*"married": true*

*}*

*}*

**Json Array**

The data enclosed into '[',']' (square bracket) represents the JSON array.

*A JSON array can have values and objects.*

e.g—

1. Simple JSON array

["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]

2. Json array of json objects

[

{"name":"Ram", "email":"Ram@gmail.com"},

{"name":"Bob", "email":"bob32@gmail.com"}

]

**Boolean**

Boolean in JSON are either true or false

**Python data to JSON data types conversion tables.**



**Json data to python data conversion table**



**Parse JSON into Python**

The json module makes it easy to parse JSON strings and files containing JSON object

We have two methods to part it

1. json.loads(json\_data) ------ from JSON to python dictionary
2. json.load(file\_object) -------- from JSON file to python

**json.loads()**

using this method, we can convert JSON string into python data type or python data in a variable

We can parse a JSON string using json.loads() method.

*The method returns a dictionary if input data is validate json data else will get ValueError exception*.

Using load() method we can check if input data is a valid input data.

import json

person = '{"name": "Bob", "languages": ["English", "Fench"]}'

person\_dict = json.loads(person)

# Output: {'name': 'Bob', 'languages': ['English', 'Fench']}

print( person\_dict)

NOTE:

Here person is json string that is converted into python dictionary

**json.load(file\_object)**

We can use this method to convert JSON data stored in file to python variable

We can use json.load() method to read a file containing JSON object.

import json

with open('path\_to\_file’) as f:

  data = json.load(f)

print(data)

**Converting Python to JSON string**

**Python(dict,list,tuple etc) Convert to JSON string**

We can convert any of python data types into equivalent data types, for this we have below two methods

1. json.dumps(python\_var) ---------- from python variable to JSON string
2. json.dump(file\_path) ------------

**json.dumps(python\_var)**

using dumps(pyth\_variable) we can convert any python vraibale into it’s equivalent json data types

mport json

person\_dict = {'name': 'Bob',

'age': 12,

'children': None

}

person\_json = json.dumps(person\_dict)

# Output: {"name": "Bob", "age": 12, "children": null}

print(person\_json)

More about the dumps() can be check on next page

**json.dump(python\_data,file\_object)**

this method is used to write python data into JSON file or converting from python JSON variable to JSON file

import json

person\_dict = {"name": "Bob",

"languages": ["English", "Fench"],

"married": True,

"age": 32

}

with open('person.txt', 'w') as json\_file:

  json.dump(person\_dict, json\_file)

Above program will convert python data of variable person\_dict into person.txt file in which data will be in JSON format.

More j about json.dump() can be checked below

**json.dump(obj, skipkeys=False, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)**

1. If skipkeys is true (default: False), then dict keys that are not of a basic type (str, int, float, bool, None) will be skipped instead of raising a TypeError.
2. If indent is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or "" will only insert newlines.
3. Separators are used to specify how key and value will be separated for custom separation

separators=("separator\_for\_key", "separation\_for\_value ")

1. sort\_keys parameter to specify if the result should be sorted or not, if TRUE then the output of dictionaries will be sorted by key

**json.dumps(obj, skipkeys=False, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)**

Serialize obj to a JSON formatted str using this conversion table. *The arguments have the same meaning as in dump().*

**Encoders and Decoders**

**Encoders**

**class json.JSONEncoder(\*, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, sort\_keys=False, indent=None, separators=None, default=None)**

If we do conversion from Python to JSON data types they are converted to their equivalent data types as listed below



**Decoders**

Supports the following objects and types by default

**class json.JSONDecoder(\*, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, strict=True, object\_pairs\_hook=None)**



**requests**

Requests is a Python module that you can use to send all kinds of HTTP requests. It is an easy-to-use library with a lot of features ranging from passing parameters in URLs to sending custom headers and SSL Verification

It has below methods-

|  |  |
| --- | --- |
| [delete(url, args)](https://www.w3schools.com/python/ref_requests_delete.asp) | Sends a DELETE request to the specified url |
| [get(url, params, args)](https://www.w3schools.com/python/ref_requests_get.asp) | Sends a GET request to the specified url  It returns response object |
| [head(url, args)](https://www.w3schools.com/python/ref_requests_head.asp) | Sends a HEAD request to the specified url |
| patch(url, data, args) | Sends a PATCH request to the specified url |
| [post(url, data, json, args)](https://www.w3schools.com/python/ref_requests_post.asp)  *requests.post(*url*, data={*key*:* value*}, json={*key*:* value*},* args*) --- syntax*  arguments means zero or more named arguments | Sends a POST request to the specified url |
| put(url, data, args) | Sends a PUT request to the specified url |
| request(method, url, args) | Sends a request of the specified method to the specified url |

**Methods of attributes of response**

Response is obtained when ever we send request using get() to server. It has below methods/attributes

json() ----- retuns json object of result(if result is in JSON format else error)

status\_code ---- returns status that indicates status

url --- returns url of response

close() ---- close connection to server

**How to send HttpResponse using CLI**

We can send using command line http clients, we can send. There are many but few are:-

1. HttPie ------ we will work with HttPie
2. Curl
3. Wget

**Sending HttpRequest using HttPie**

http <url\_for\_request>

Note:

For sending request using cmd, as usual server need to be running

Views.py file

def home(request):

    print("tis is home view method")

    d={1:"hello",2:"Shyam",3:"Hare Hare"}

    json\_data=json.dumps(d)

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

urls.py

app\_name='no\_rest'

urlpatterns = [

    path('home/',views.home, name='no\_rest-home'),

]

sending httprequest

http http://127.0.0.1:8000/no\_rest/home/

output is below



**How to send dictionary object as Json type**

**Note:**

We can use **JsonResponse()** method from json module to send dict object as json object, method will itself convert from dictionary to Json

def home(request):

    print("tis is home view method")

    d={1:"hello",2:"Shyam",3:"Hare Hare"}

    json\_data=json.dumps(d)

    print('data type of d is: ', type(d))

    print('data type of json\_data  is: ', type(json\_data))

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

    return JsonResponse(d)

**How to send httprequest from python application**

Using 'requests' module we can send httprequest on a specified url.

In requests library we have **'get()'** using which we can call url which will fetch response

The returned response we can convert into dictionary object using json() as – response.json()

response =requests.get(url) #Getting response from url

dict\_data=response.json() # getting data back into json format (if returns json data)

test.py file

import requests

import json as js

base\_url='http://127.0.0.1:8000/no\_rest/'

end\_point='home'

response=requests.get(base\_url+end\_point)

print('this is test application')

print(type(response))

test.py file can be created/placed in any path(in this case I have created into rest\_api folder which is in project folder)



Common language b/t two app--- Http(java, Django etc)

Common message format --- JSON, XML(old technology)

**How to disable csrf\_token middleware**

In Settings.py file go to middleware section and comment it.

**What is Mixin**

Mixin in class in Django which acts as parent class to provide functionality to child class, it supports only to child class.

Mixin class always extends object class not any other class

It always provides support to child class

Let’s take an example of mixin, here we are going to deal with same above example.

Mixin.py file

import json

from django.http import JsonResponse

class JsonMininx:

    def json\_data(self,data):

        return JsonResponse(data)

views.py file

class JsonCBV(JsonMininx,View):

    def get(self,request):

        d={1:"hello",2:"Shyam",0:"Hare Hare"}

        return self.json\_data(d)

**NOTE**:

Now here we can notice that where ever is required to run json response we can call json\_data from JsonMininx class.(we can modify that for any number of argument using variable argument)

**Multiple inheritance vs Mixin**

|  |  |
| --- | --- |
| **Mixin** | **Multiple inheritance** |
| >Parent class instantiation not possible or of no use  >Parent class contain only instance method not instance variable.  >Methods are useful only for Child class.  >Parent class should be direct child class of object class. | >We can instantiate.  >It will contain both instance variable and method.  >Methods are useful for parent and child class.  >Parent class can inherit any other class also |

**What is difference between makemigrations and migrate command**

Makemigrations ---- create SQL query to create DB table

Migrate ------ creates table and inserts data into DB

**http Status code in Django**

1XX ----- this is for informational, it’s range is from 100 to 199

2XX ----- this is for success, it range is from 200 to 299

3XX ----- this is re-directional, it range from 300 to 399

4XX ----- Client Error, ranges from 400 to 499 (e.g --- page not found)

5XX ----- Server error

**Create a REST api to get data from database**

Here we will Employee model.

models.py file

class Employee(models.Model):

    ename=models.CharField(max\_length=100)

    esal=models.IntegerField()

    eadd=models.CharField(max\_length=100)

views.py file

class EmployeeCBV(View):

    def get(self,request,\*args):

        data=Employee.objects.get(id=1)

        print(data)

        emp\_data={'ename':data.ename,'esal':data.esal,'eadd':data.esal}

        json\_data=json.dumps(emp\_data)

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

urls.py file

urlpatterns = [

    #path('home/',views.home, name='no\_rest-home'),

    path('cbv/',views.JsonCBV.as\_view(),name="no\_rest\_cbv"),

    path('emp/',views.EmployeeCBV.as\_view(),name="no\_rest\_emp-cbv"),

]

Now from based on urls mapping and views we can get data from database using requests library

test.py file for getting json data

import requests

import json as js

#http://127.0.0.1:8000/no\_rest/cbv/

#http://127.0.0.1:8000/no\_rest/cbv/

base\_url='http://127.0.0.1:8000/no\_rest/'

end\_point='cbv'

response=requests.get(base\_url+end\_point)

**Customise above code in such way that it will accept employee id from requestor/url and then it will display detail of that employee.**

Hint- Modify urls.py file and class based view

urls.py file

urlpatterns = [

    #path('home/',views.home, name='no\_rest-home'),

    path('emp/<int:pk>',views.EmployeeCBV.as\_view(),name="no\_rest\_emp-cbv"),

]

Views.py file

class EmployeeCBV(View):

    pk=None

    def get(self,request,pk,\*args):

        data=Employee.objects.get(id=pk)

        print(data)

        emp\_data={'ename':data.ename,'esal':data.esal,'eadd':data.esal}

        json\_data=json.dumps(emp\_data)

        print(type(json\_data))

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

output

**What is Serialization**

Converting python dict object to json object is called serialization

(In general terminology converting from one object to another is called serialization)

Now suppose in case we used Entry.object.all() then there are chances that we may get thousands of data then converting them in json is big problem. To overcome this burden djanog have module called **serializer**

**Serializers**

Serializer is available in django.core import serializers

from django.core import serializers

Django’s serialization framework provides a mechanism for “translating” Django models into other formats. Usually these other formats will be text-based and used for sending Django data over a wire, but it’s possible for a serializer to handle any format (text-based or not).

**Serializing data**

We can use **serialize()** function to serialize data from queryset to json format

serialize(data\_format,[data])

this is function in django.core.serializers

This methods creates json data as json array**.**

data\_format----- data format in which you want to serialize, this is string. E.g—xml,json

data ---- data that want to serialize, it’s iterable data , e.g--- queryset

**Note:**

If data is iterable(e.g- queryset) then we can some use that if not iterable then give in list form

CASE1:

data=Employee.objects.all()

json\_data=serializers.serialize('json',data,fields=['ename','eadd'])

here data is queryset (an iterable data)

CASE 2:

data=Employee.objects.get(id=pk)

json\_data=serializers.serialize('json',[data])

here data is not iterable data so giving in list

e.g---

data = serializers.serialize('xml', Restaurant.objects.all())

serialize(data\_format,[data], fields=[files\_that\_want\_to\_show])

This method creates json data as **json array.**

fields--- files that you want to show or give to requestor, these are nothing but column of DB table

e.g.-

json\_data=serializers.serialize('json',[data],fields=['ename','eadd'])

Showing all files of model class

class EmployeeCBV(View):

    pk=None

    def get(self,request,pk,\*args):

        print('id is: ',pk)

        data=Employee.objects.get(id=pk)

        json\_data=serializers.serialize('json',[data,])

        print("data typed of serialized data is: ",type(json\_data))

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

Showing only required fields

class EmployeeCBV(View):

    pk=None

    def get(self,request,pk,\*args):

        print('id is: ',pk)

        data=Employee.objects.get(id=pk)

        json\_data=serializers.serialize('json',[data],fields=['ename','eadd'])

        print("data typed of serialized data is: ",type(json\_data))

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

NOTE:

When we used above two methods for fetching data then the fetched data will be list of dictonary.

Key of dictionary will be – model, fields, primary key of id

**fields**--- it denotes all data from model class , doesn’t matter it we have used fields attribute while converting to json data using serialize function

**WAP a view to get data only, don’t want keys**

final\_list=[]

for each in json\_data:

final\_list.append(each[‘fields’])

json\_data=json.dumps(final\_list)

class EmployeeListCBV(View):

    def get(self,request,\*args):

        print('this is EmployeeListCBV class ')

        qs=Employee.objects.all()

        json\_data=serializers.serialize('json',qs)#convert into json(list of dict will be)

        d=json.loads(json\_data) #get back in python data format

        fina\_list=[]

        for each in d:

            fina\_list.append(each['fields'])

        json\_data=json.dumps(fina\_list)#now convert into JSON data format

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

Let’s try to use here mixins to separate some code for getting required data only.

Views.py file

class EmployeeListCBV(View,EmployeeMixins):

    def get(self,request,\*args):

        print('this is EmployeeListCBV class ')

        qs=Employee.objects.all()

        data=self.json\_data(qs)

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

mixins.py file

class EmployeeMixins(object):

    def json\_data(self,qs):

        json\_data=srz.serialize('json',qs) #converting query set into json

        d=json.loads(json\_data) #converted back to python

        fina\_list=[]

        for each in d:

            fina\_list.append(each['fields'])

        json\_data=json.dumps(fina\_list)

        return json\_data #retunred json data

Now here we will not get some details like – model name, fields etc.

**Exception handling in API**

Let say we have below code in view.py file-

class EmployeeCBV(View,EmployeeMixins):

    pk=None

    def get(self,request,pk,\*args):

        print('i am working wih JSON data')

        print('id is: ',pk)

        qs=Employee.objects.get(id=pk)

        data=self.json\_data([qs,])

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

when user tries to access any data by using id then it’s going to give exception/error, that we need to handle that need to be handled as below-

*Method 1: More suggustable*

Modify the views.py file to handle exception as shown below-

view.py file

class EmployeeCBV(View,):

    pk=None

    def get(self,request,pk,\*args):

        print('i am working wih JSON data')

        print('id is: ',pk)

        try:

            qs=Employee.objects.get(id=pk)

        except Employee.DoesNotExist:

            data=json.dumps({'msg':"no such employee"})

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

        else:

            data=self.json\_data([qs,])

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

*Method 2:* handle in partner application

If Django application doesn’t handles exception then we can handle in our partner application.

Python application(test.py file)

data=requests.get('http://127.0.0.1:8000/no\_rest/emp/200/')

if data.status\_code in range(200,300):

    data=data.json() #get the json data obtained in response

    print(data)

else:

    print('something goes worng')

views.py file

class EmployeeCBV(View,EmployeeMixins):

    pk=None

    def get(self,request,pk,\*args):

        print('i am working wih JSON data')

        print('id is: ',pk)

        qs=Employee.objects.get(id=pk)

        data=self.json\_data([qs,])

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

mixin.py file

class EmployeeMixins(object):

    def json\_data(self,qs):

        json\_data=srz.serialize('json',qs) #converting query set into json

        d=json.loads(json\_data) #converted back to python

        fina\_list=[]

        for each in d:

            fina\_list.append(each['fields'])

        json\_data=json.dumps(fina\_list)

        return json\_data #retunred json data

**How to add status code manually in response**

We can add status code while returning response. The same status code will be retuned to partner application or in response

HttpResponse(data, content\_type='application/json', status=status\_code)

**Create operation**

For create operation we will be using POST() in our view(if class based view of not class based view then we have to check method name for which data is coming and then work accordingly) or from requests.

**How to dump data from database**

In Django we can dump data using manage.py file as below

python manage.py dumpdata <app\_name>.<model\_name> --format format\_value --indent int –output filename

Using above command, we can dump data to console for given app and model name.

--fomrat ----------> this is optional, default to JSON format

-- indent -----optional, specify the indent label you want to keep while displaying data

-- output ----- optional, specify the file where you want to write data, default is console

**How to disable csrf\_token verification**

We can disable csrf\_token verification in three ways

1. **Method level or function lelvel.**

To disable at function label import decorator '@csrf\_exempt' on the where for which we want to disable csrf token validation.

@csrf\_exempt

def myview(request,\*args):

#

1. **Class label**

For class label we need to use below two decorators with class name.

from django.views.decorators.csrf import csrf\_exempta

from django.utils.decorators import method\_decorator

@method\_decorator(csrf\_exempt, name='dispatcha)

Class myClassBasedview(View):

#

*name= ‘dispatch’* -------- then it disables csrf validation for all methods related to class.

1. Project label

Disable it from setting.py file

Now here we will try to send a create operation for our model Employee from python application (test.py file)

test.py file

def create\_resource():

    emp={'ename':"rakesh",'esal':2000,'eadd':"semarbari"}

    json\_data=json.dumps(emp)

    resp=requests.post('http://127.0.0.1:8000/no\_rest/emp/',data=json\_data)

    print('retundes statuse code is: ',resp.status\_code)

create\_resource()

views.py file

@method\_decorator(csrf\_exempt,name='dispatch')

class EmployeeListCBV(View,EmployeeMixins):

    def get(self,request,\*args):

        print('this is EmployeeListCBV class ')

        qs=Employee.objects.all()

        data=self.json\_data(qs)

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

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

        print('processing post request')

        data={'msg':'this is retunred message'}

        json\_resp=json.dumps(data)

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

Output will be :-

retundes statuse code is: 200 # 200 means we are able to send a post request,

**Adding data validation(data type) in view**

Now in my view we have not added validation to check if data sent by consumer is validation json data or not, we will check if and accordingly will sent a message to consumer.

Steps:

1. While receiving data try to convert into python data, is completed without any error then requestor have sent a valid input data.

Views.py file

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

        input\_data=request.body #retriving data from request

        try:

            json.loads(input\_data)

        except ValueError as e:

            print(e)

            data={'msg':'invalid data'}

            json\_resp=json.dumps(data)

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

        else:

            data={'msg':'we have got your data'}

#we will also add logic data validation with our model and will add in DB

            return HttpResponse(json.dumps(data),content\_type='application/json')

request.body ---- this is way to retrieve data from request. Check HttpRequest in djano

tesy.py file

def create\_resource():

    emp={'ename':"rakesh",'esal':2000,'eadd':"semarbari"}

    json\_data=json.dumps(emp)

    resp=requests.post('http://127.0.0.1:8000/no\_rest/emp/',data=json\_data)

    print('retundes statuse code is: ',resp.status\_code)

    print(resp.json())

create\_resource()

Output:

Now if we sent other than json data from client end then we will get ‘Invalid data’ as response at client end. (instead of data=json\_data try use data=emp in test.py file and see response/output)

**Now our requirement is to do data validation(data type, data related to model) and add data to DB**

@method\_decorator(csrf\_exempt,name='dispatch')

class EmployeeListCBV(View,EmployeeMixins):

    def get(self,request,\*args):

        print('this is EmployeeListCBV class ')

        qs=Employee.objects.all()

        data=self.json\_data(qs)

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

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

        input\_data=request.body

        try:

            py\_data=json.loads(input\_data)

            fm=forms.AddDataForm(py\_data)

            if fm.is\_valid():

                print('data was a valid data')

                fm.save()

                return HttpResponse(json.dumps({"msg":"data added to DB"}),content\_type='application/json')

        except ValueError as e:

            print(e)

            data={'msg':'invalid data'}

            json\_resp=json.dumps(data)

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

        else:

            data={'msg':'we have got your data'}

            return HttpResponse(json.dumps(data),content\_type='application/json')

**Create API to update details of a Employee**

For updating we need to use put() in views.py file and in python applications.

Consider a case where we can to update employee address and salary from our python applications for given eomployee, employee id is paswd in url

def update\_resource():

    url='http://127.0.0.1:8000/no\_rest/emp/1/' #here I at end is emp id to update

    emp={'ename':"mukesh",'esal':10,'eadd':"Delhi"}

    json\_emp=json.dumps(emp)

    resp=requests.put(url,data=json\_emp)

    print('retundes statuse code is: ',resp.status\_code)

    print(resp.json())

update\_resource()

Here from python application we are sending data in json form to update in database.

Utils.py file in app

import json

from .models import \*

from .forms import \*

#this method is for checking if data provided was in json format

def is\_valid\_json(data): #this is for checking it input data is valid json

    try:

        json\_data=json.loads(data)

        return True

    except ValueError as e:

        return False

    finally:

        pass

def get\_object\_by\_id(id): #this is just to check if employee exists with given id

    try:

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

        return True

    except Employee.DoesNotExist as e:

        return False

    finally:

        pass

forms.py file

from django import forms

from . import models

class AddDataForm(forms.ModelForm):

    class Meta:

        model=models.Employee

        fields='\_\_all\_\_'

views.py file

@method\_decorator(csrf\_exempt,name='dispatch')

class EmployeeCBV(View,EmployeeMixins):

    pk=None

    def put(self,request,pk,\*args,\*\*kwargs):

        print('employee id is: ',pk)

        print('input data is: ',request.body)

        if is\_valid\_json(request.body):

            if get\_object\_by\_id(pk):

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

                input\_data=request.body

                provided\_data=json.loads(input\_data)

                print(emp.id,emp.ename,emp.esal,emp.eadd)

                emp\_dict={'ename':emp.ename,'esal':emp.esal,"eadd":emp.esal}

                emp\_dict.update(provided\_data)

                print('updated employee  dictinary is: ',emp\_dict)

                form=AddDataForm(emp\_dict,instance=emp) #creating new emp obj to update

                if form.is\_valid():

                    msg={'msg':'form is valid'}

                    form.save()

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

                    print(emp.id)

                    print(emp.ename)

                    print(emp.esal)

                    return HttpResponse(json.dumps(msg),content\_type='application/json', status=200)

                if form.errors:

                    msg={'msg':'Error is form creations'}

                    return HttpResponse(json.dumps(msg),content\_type='application/json', status=200)

            else:

                msg={'msg':"object does not exists"}

                return HttpResponse(json.dumps(msg),content\_type='application/json', status=404)

        else:

            return HttpResponse(json.dumps({"msg":"invalid data format"}),content\_type='application/json', status=200)

**Note:**

1. emp\_dict.update(provided\_data)

I have used this to ceate new dicntionary which I will use while creating the employee object for updating the details, irrespective of any parameter that want to update.

1. form=AddDataForm(emp\_dict,instance=emp)

Here I havecreated new form object , I am using instance=emp so that while calling save() on this object it doesn’t create new rows instead that it update the old object(emp)

1. form.errors and form.is\_valid() these two methods are from forms class which are being used to get the error and validate the form(form=AddDataForm(emp\_dict,instance=emp)
2. Employee.DoesNotExist this coming from Employee model of model.py file to handle the exception when employee is not existing with given id.

**Create API to delete details or delete a Employee**

Try by you own

**Thumb Rule:**

1. *We should have only one endpoint for out api to all operations.*

Note:

1. In rest api there must be only one end point but here we have used two end points that should not be case.

Django Rest FrameWork

There are many frame work available in Django as—

DRF

Testify

Features of DRF

1. It provides inbuilt security features (TokenAuthentication, OAuth authentication, JWAuth etc)
2. It provides browsable api(by using browser we can check if API is working properly or not)
3. Serializers both ORM and non ORM
4. Extensive documentations and library
5. Trusted by big companies (Mozilla, amazon)

*pip freeze >file\_name.txt ------- this is will dump all installed library/software in file\_name.*

To use DRF we must install/add rest\_framework in out setting.py file.

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'rest\_framework',

]

If we are intending to use browsable api then we need to use path('api-auth/', include('rest\_framework.urls')) in out project’s url.py file.

urlpatterns = [

    path('admin/', admin.site.urls),

    path('api-auth/', include('rest\_framework.urls')),

]

**DRF serializers**

Below are three main roles of drf serializers.

1. Serialization
2. Deserialization
3. Validaitions

**Hint for future:**

Our DRF serializers concept simply access forms and model forms only. It will exacly work and forms and model forms.

Django forms

class Employee(forms.Form):

    ename=models.CharField(max\_length=100)

    esal=models.IntegerField()

    eadd=models.CharField(max\_length=100)

Using DRF serailizers

class Employee(serializers.Serialozers):

    ename=models.CharField(max\_length=100)

    esal=models.IntegerField()

    eadd=models.CharField(max\_length=100)

Django model forms

class AddDataForm(forms.ModelForm):

    class Meta:

        model=models.Employee

        fields='\_\_all\_\_'

Using DRF serailizers(Model Serializers)

class AddDataForm(serializers.ModelSerializers):

    class Meta:

        model=models.Employee

        fields='\_\_all\_\_'

**Serializations**

Process of converting from complex types (model instance or queryset OR JSON) to python native data types(dict) is called serializations

(Converting from JSON to python data types also)

Let’s take an example for discussing serializations. Here we have one Employee model and for which we want to define rest\_framework serializers class.

**Note:**

It’s always best practice to create serializers in a separate file.

Employee model in models.py file

class Employee(models.Model):

    eno=models.IntegerField()

    ename=models.CharField(max\_length=100)

esal=models.floatfiled()

Now above Employee model we will create on serializer. I have used serialisers.py file



Seliralizer.py file

from rest\_framework import serializers

class EmployeeSerializer(serializers.):

    enp=serializers.IntegerField()

    ename=serializers.CharField(max\_length=100)

    esal=serializers.FloatField()

Here what ever variable we are using eno, ename, seal only those variable will be serialized, if trying to serialized anyother variable then will get error. Check in serializer of APIView.

Here models is changed to serializers while defining the data types.

**Steps for converting into python data types from complex data types**

1. Get the model object instance
2. Get the instance of Serializer defined for that model on that model object(say—eser)
3. Instance\_of\_serializer.data --- this will have all data in python dict types(eser.data)

**Questions:**

**Get employee object and then convert that into python native(dict) datatypes using rest\_framework serializers.**

>>> from testapp.models import Employee

>>> from testapp.myser import EmployeeSer

>>>emp=Employee.objects.get(id=1)

>>> eser=EmployeeSer(emp)

>>> eser.data # data will in in python dict format, data is bultin attribute

**How to convert from python dict /native to json data types.**

In rest\_framework.renderer there is module by name JSONRenderer, using this the object of this module we can use render method of this module.

>>>from rest\_framework.renderers import JSONRenderer #import rendered

>>> json\_data=JSONRenderer().render(eser.data) #

**How to perform serialization of queryset**

1. Get the querset
2. Get the serialization instance using that query set, you must use many=True for queryset.
3. Rest all are same

>>> eser=EmployeeSer(emp,many=True)

>>> eser.data

**How to convert from python dict /native to json data types(when we got serialize data from qs)**

In rest\_framework.renderer there is module by name JSONRenderer, using this the object of this module we can use render method of this module.

This is same as previous.

>>>from rest\_framework.renderers import JSONRenderer #import rendered

>>> json\_data=JSONRenderer().render(eser.data) #

**Deserializations**

The process of converting python native data types into database supported complex types is called deserializations.

**Converting JSON into python native types**

In rest framework we have JSONParser() class in rest\_frameowrk.parser. Class JSONParser have method parse() which can convert to python data types.(data is parse() should be in io stream format)

from rest\_framework.parser import JSONParser

stream=io.ByteIO(json\_data)

py\_data=JSONParser().parse(stream)

**Converting python data to Database supported data (Deserialization)**

Create serializer object from defined serializer class

Validate the deserialized object

From deserialized object get data into db supported format

eser=EmployeeSer(data=python\_data) #creating serializer object

eser.is\_valid() #validate that it’s in db supported format

eser.validated\_data #data in db supported format

**Example 1:** Getting data from db using rest\_framwork

Views.py file

from django.shortcuts import render

from django.views.generic import View

import io

from rest\_framework.parsers import JSONParser

from .models import Employee

from .myser import EmployeeSer

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

# Create your views here.

@method\_decorator(csrf\_exempt,name='dispatch')

class EmployeeCRUDCBV(View):

    print('this is EmployeeCRUDCBV view')

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

        print('this is get view')

        json\_data=request.body

        stream=io.BytesIO(json\_data)

        py\_data=JSONParser().parse(stream) #data is in python format

        id=py\_data.get('id',None)

        if id is not None:

            emp=Employee.objects.get(id=id)#this is employee object

            print(emp)

            #converting emp object into JSON types

            eser=EmployeeSer(emp)

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

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

            #send id specific record

        #send all record

        qs=Employee.objects.all()

        eser=EmployeeSer(qs,many=True)

#py\_data=eser.data #now this data is in python native(list of dict) format

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

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

myser.py (serializer that I created)

from rest\_framework import serializers

class EmployeeSer(serializers.Serializer):

    eno=serializers.IntegerField()

    ename=serializers.CharField(max\_length=100)

    esal=serializers.FloatField()

tests.py file (python applications)

def get\_resource(id=None):

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

    end\_points='api/'

    data={}

    if id is not None:

        data={'id':id}

        json\_data=json.dumps(py\_data)

        resp=requests.get(BASE\_URL+end\_points,data=json\_data)

        print('fectched id specific data')

        print(resp.json())

        # print('status code is: ',resp.status\_code)

    else:

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

        print('fetched all data')

        print(resp.json())

        pass

get\_resource()

**Example 2: Post request using rest**

We need to define create() in our serializer class which will be invoked by save() for creating object in DB.

Views.pu file

@method\_decorator(csrf\_exempt,name='dispatch')

class EmployeeCRUDCBV(View):

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

        json\_data=request.body

        stream=io.BytesIO(json\_data)

        py\_data=JSONParser().parse(stream)

        eser=EmployeeSer(data=py\_data)

        if eser.is\_valid():

            eser.save()#internally it will call create method of EmployeeSer class

            msg={'msg':"resource created successfully"}

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

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

        else:

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

            return HttpResponse(json\_data,content\_type='application/json',status=400)

myser.py file (serializer created for employee class)

from rest\_framework import serializers

from .models import Employee

class EmployeeSer(serializers.Serializer):

    eno=serializers.IntegerField()

    ename=serializers.CharField(max\_length=100)

    esal=serializers.FloatField()

    def create(self,validate\_data):

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

tests.py file (python applications)

def create\_resource(id=None):

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

    end\_points='api/'

    data={'eno':510,'ename':'Vinod','esal':10000}

    resp=requests.post(BASE\_URL+end\_points,data=json.dumps(data))

    print(resp.status\_code)

    print(resp.json())

create\_resource()

**Update employee data using rest**

For doing any update we need to override update() in our serializer class and create a serializer object accordingly. Example is below

serializer\_class(model\_instance,data=python\_data,partial=True) ------ genral informations

* model\_instance --- model instance name obtained from DB for which we want to update data
* data ------- data in python dict form which need to be update in db for model\_intance obtained from DB
* partial=True--- to support for partial and full update

overriding update() in serializer class

def update(slef,model\_instance,validate\_data):

#code for change/update value of column

instance.ename=validated\_data.get('ename',instance.ename)

General hint/steps:

1. Override update method for details update and write save() on instance of model name that want to update.
2. Create a serializer instance from provided data , instance for which want to update and partial=True
3. Call save() from views that will call update() of serialzer class.

Example:

From a python application we want to update any of employee data, create a API

Serialiser class

class EmployeeSer(serializers.Serializer):

    eno=serializers.IntegerField()

    ename=serializers.CharField(max\_length=100)

    esal=serializers.FloatField()

    def create(self,validate\_data):

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

    def update(self,instance,validate\_data):

        instance.eno=validate\_data.get('eno',instance.eno)

        instance.ename=validate\_data.get('ename',instance.ename)

        instance.esal=validate\_data.get('esal',instance.esal)

        instance.save()

        return instance

instance.eno=validate\_data.get('eno',instance.eno) ------- these lines are just to update value with provide data if not provided then update with existing data(i.e no update in rfact)

@method\_decorator(csrf\_exempt,name='dispatch')

class EmployeeCRUDCBV(View):

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

        json\_data=request.body

        stream=io.BytesIO(json\_data)

        py\_data=JSONParser().parse(stream)#now data is in dict format

        print(py\_data)

        id=py\_data.get('id')

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

        eser=EmployeeSer(emp,data=py\_data,partial=True)

        if eser.is\_valid():

            eser.save() # this will internally call update() of serializer class

            msg={'msg':'data updated successfully'}

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

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

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

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

**Deleting record using serializer**

Serializer is not required here

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

Emp.delete()

**Validation using Serializer**

There are three types of validation if we do use serailizer validations

1. Field level validations ----- one filed, e.g --- validating salary of employee
2. Object level validations ----- multiple filed together, e.g --- name and salary
3. By using validators

**Field Level Validations**

We can write validation method for any filed which we want to validate in below fashion—

*The Below method will be invoked by python whenever we call is\_valid() on serialized object.*

def validate\_variable\_or\_filed\_to\_validate(self,value):

#validation logic

return value

**Example**

**Write a validation method in such a way that it will validate for salary to be greater than 5000.**

serializer.py file, remaining all file will be same

def validate\_esal(self,value):

        if value<5000:

            raise serializers.ValidationError('Salary must be more than 4999')

        return value

Now when ever we call is\_valid() on serialiser object that will call above validation method.

**Object level validations**

This is used for validating many filed together, e.g – validating ename, esal, eadd fileds

of employee data.

To do object level validate() method in our serializer class.

def validate(self,value):

#validation logic for fileds

**Example:**

**Update salary of employee whose id is 1 but for employee of name Sunny salary must not be less than 5555. Salary will be provided by python applications.**

serializer.py file, remaining all file will be same

class EmployeeSer(serializers.Serializer):

    eno=serializers.IntegerField()

    ename=serializers.CharField(max\_length=100)

    esal=serializers.FloatField()

    def validate(self,value):

        ename=value.get('ename')

        esal=value.get('esal')

        if ename.lower()=='sunny' and esal<55555:

            raise serializers.ValidationError('Sunny salary must be greater than 55554')

        return value

**Note:**

When we call is\_valid() on serialzer object then python does field level and object level validation both if it is defined.

**By Using validators**

This is done by default from python based on fields defined in serializers and field options or any other functions defined for validations. Validation will happen at is\_valid() call

If doing from function then declare that function at global level in you serializer file

Serialisers.filed\_name(validators=[function\_name\_for\_validatio])

Serializer.file\_name(field\_options)

**Example:**

**Before updating any salary make sure that salary is multiple of 1000 and max\_length of ename field is 5.**

serializer.py file, remaining all file will be same

def multiple\_1000(value):

    if value%1000!=0:

        raise serializers.ValidationError('salary must be multiple of 1000')

    return value

class EmployeeSer(serializers.Serializer):

    eno=serializers.IntegerField()

    ename=serializers.CharField(max\_length=5)

    esal=serializers.FloatField(validators=[multiple\_1000])

Let say we tried to update details of employee with data={'ename':'Shukla','esal':55555,'id':1} from our python application.

Output

{'ename': ['Ensure this field has no more than 5 characters.'], 'esal': ['salary must be multiple of 1000']}

Note:

1. Internally validation will happen for all type of validation(field , object, validators) if defined or asked to do on call of is\_valid().
2. Priority/Order of validations

Validators----->Field validations----->object validations

**Drawback of serializers**

1. Need to define fields explicitly
2. Need to define create(), update() methods.

################################################################################

**# MODEL SERIALIZERS #**

################################################################################

For creating model serializers we need to inherit ModelSerializer in our serializer class and use class Meta option for the fields that you want to include in this serializer class.

Once we have created model serializer class then we don’t need to define create() or update() methods.

class EmployeeSer(serializers.ModelSerializer):

    class Meta:

        model=Employee

        fields='\_\_all\_\_'

**Specifying Fields that need to be serialized**

fields='\_\_all\_\_' ---- serialize all fields of model class

fields= ['field1','filed2'....] --- serialize specified fields

exclude=['field1','field2',...] --- don't serialize specified fields of model/Meta class

Now we have model serializer then we can do all previous operation(get,update,delete,create) on previous class based view nothing else is required to change.

Model serializer provides default implementation for post and create so no need to override update() and create() in model serializer class.

**Validation using ModelSerializers.**

* If we want to do validation of any file then declare that field in serializer class how we do in serializer.
* Method creations for validation in modelserializers is same as for normal serializers for field validation, object validation or validators.

**Example 1:**

**Write a validator for salary to check salary is multiple of 1000.**

Serializers.py file, remaining all will be same

def multiple\_1000(value):

    if value%1000!=0:

        raise serializers.ValidationError('salary must be multiple of 1000')

    return value

class EmployeeSer(serializers.ModelSerializer):

    esal=serializers.FloatField(validators=[multiple\_1000])

    class Meta:

        model=Employee

        fields='\_\_all\_\_'

views.py file (this is same as for normal serializer nothing is changed)

@method\_decorator(csrf\_exempt,name='dispatch')

class EmployeeCRUDCBV(View):

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

        json\_data=request.body

        stream=io.BytesIO(json\_data)

        py\_data=JSONParser().parse(stream)#now data is in dict format

        print(py\_data)

        id=py\_data.get('id')

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

        eser=EmployeeSer(emp,data=py\_data,partial=True)

        if eser.is\_valid():

            eser.save()

            msg={'msg':'data updated successfully'}

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

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

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

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

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

        json\_data=request.body

        stream=io.BytesIO(json\_data)

        py\_data=JSONParser().parse(stream)

        eser=EmployeeSer(data=py\_data)

        if eser.is\_valid():

            eser.save()#internally it will call create method of EmployeeSer class

            msg={'msg':"resource created successfully"}

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

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

        else:

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

            return HttpResponse(json\_data,content\_type='application/json',status=400)

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

        print('this is get view')

        json\_data=request.body

        stream=io.BytesIO(json\_data)

        py\_data=JSONParser().parse(stream)

        id=py\_data.get('id',None)

        if id is not None:

            emp=Employee.objects.get(id=id)#this is employee object

            print(emp)

            #converting emp object into JSON types

            eser=EmployeeSer(emp)

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

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

        qs=Employee.objects.all()

        eser=EmployeeSer(qs,many=True)

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

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

**Field validations**

This is same as how we do in normal serializer validations

Serializer.py file

from rest\_framework import serializers

from .models import Employee

def multiple\_1000(value): #this is for validator validations not required for field val.

    print('validator validations')

    if value%1000!=0:

        raise serializers.ValidationError('salary must be multiple of 1000')

    return value

class EmployeeSer(serializers.ModelSerializer):

    esal=serializers.FloatField(validators=[multiple\_1000])

    def validate\_esal(self,value):

        print('file level valiations')

        if value<10000:

            raise serializers.ValidationError('Salary must be more than 4999')

        return value

    class Meta:

        model=Employee

        fields='\_\_all\_\_'

**Model validations**

This is exactly copy paste of normal validations.

Try to do it.

**Order/priority of validations**

Validator ------> Filed validation -----------> Object validations (same as normal serializer)

################################################################################

**# DRF VIEWS #**

################################################################################

DRF provides two class for views

1. APIView
2. ViewSet

APIView is smething like function based view of Django

ViewSet is something like class based view of django

**APIView vs ViewSet**

|  |  |
| --- | --- |
| APIView | ViewSet |
| >Present in rest\_framework.view module  >Method names reflect HTTP methods --- get(), put(), post(), update(), patch()  >We have to map views to urls  >Most of business logic we have to write  >Length of code is less  >API development time is more  >Developer has complete control over flow and execution  >Best for complex operation using multiple sources | >Present in rest\_framework.viewset module  >Methods names DB model class operations/names--- list(), retrive(), create(), update(), partial\_update(), destroy()  >We don’t need to map views to urls, done by routers  >Most business logic will be generated automatically  >Length of code is less  >API development time is less  >Developer won’t have complete control over flow and execution  >Best for standard operations |

**APIView**

REST framework provides an APIView class, which subclasses Django's View class.

APIView classes are different from regular View classes in the following ways:

1. Requests passed to the handler methods will be REST framework's Request instances, not Django's HttpRequest instances.
2. Handler methods may return REST framework's Response, instead of Django's HttpResponse. The view will manage content negotiation and setting the correct renderer on the response.
3. Any APIException exceptions will be caught and mediated into appropriate responses.
4. Incoming requests will be authenticated and appropriate permission and/or throttle checks will be run before dispatching the request to the handler method.

Most basic class to design API view

Child class of View.

We have to write code explicitly so we have complete control of execution.

It has clear execution flow.

Best suitable for complex operations (multiple DB operations, third party API use)

We have to map view to url manually.

Used when we are using third part APIs.

get(), put(), delete() , post().

**ViewSet**

A ViewSet class is simply a type of class-based View, that does not provide any method handlers such as .get() or .post(), and instead provides actions such as .list() and .create().

The method handlers for a ViewSet are only bound to the corresponding actions at the point of finalizing the view, using the .as\_view() method.

In ViewSet we use routers instead of url configurations.

list(), create(), details, update(), partial\_update(), destroy() --- methods we will be using

Most of execute will be already ready done so no complete control of execution , control

Good to developing simple APIs.

No need to map views to url (it will be done by Router concept).

**Testing of APIs**

Till now we were using python application/script to test our API but now we will be using now Browsable API given by DRF.

APIView

This is like function based view of Django. Let’s take an example to understand. We will be creating one API for get, post request from browsable API.

Serializer.py file

from rest\_framework import serializers

class NameSerializer(serializers.Serializer):

    name=serializers.CharField(max\_length=20)

Now for serializing or converting from JSON to python data types it will check for variable name to be name if no then serialization error will be raised.

Here we are not using any model class that’ why using normal serializer.

views.py file

from django.shortcuts import render

from rest\_framework.views import APIView

from rest\_framework.response import Response

from testapp.serializer import NameSerializer

# Create your views here.

class TestAPIView(APIView):

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

        colors=['Red','Green']

        msg={'msg':'Happy pongal'}

        return Response({'msg':'happy pongal','col':colors})

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

        print('data is: ',request.data)

        ser=NameSerializer(data=request.data)

        if ser.is\_valid():

            print('serialization completed')

            msg={'msg':'Operation completed'}

            return Response(msg)

        else:

            print('error in validations')

            return Response({'msg':ser.errors})

**Note:**

Here we don’t need to disable csrf token that being sent by browsable API.

**Response** is the class from rest\_framework.response module will will take care of deserialization(python to JSON)

Outputs:

<http://127.0.0.1:8000/api/>



Sending JSON data





Sending data with wrong variable name





It’s raising validation/serialization error and variable name “name” is expected b/c we have declared name variable in our serializer class.

Here we are written put(), delete(), patch() just for name shake

from django.shortcuts import render

from rest\_framework.views import APIView

from rest\_framework.response import Response

from testapp.serializer import NameSerializer

# Create your views here.

class TestAPIView(APIView):

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

        colors=['Red','Green']

        msg={'msg':'Happy pongal'}

        return Response({'msg':'happy pongal','col':colors})

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

        print('data is: ',request.data)

        ser=NameSerializer(data=request.data)

        if ser.is\_valid():

            print('serialization completed')

            msg={'msg':'Operation completed'}

            return Response(msg)

        else:

            print('error in validations')

            return Response({'msg':ser.errors})

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

        return Response({'msg':"this response is from PUT method"})

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

        return Response({'msg':"this response is from delete method"})

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

        return Response({'msg':"this response is from PATCH method"})

**ViewSet**

This is like class based view in Django. It has below methods. This requires less code to develop API as most of codes will be generated automatically.

When using ViewSet then url mapping is not required, it is done by routers.

**Best to use ViewSet**

If we want to develop a simple CRUD interface for our DB then go for VewiSet.

If we want to develop API then ViewSet is best choice

If not performing standard operations then ViewSet is best choice

list() ------- to get all resources or records

retrive() ---- to get specific esource

create() ------ to create a new resource

update() ----- tp perform updation

partial\_update() ---- to perform update

distroy() ------ to delete a resource

get() ---- list(), retrive()

post() ---- create()

put() ----- update()

patch() ----- partial\_update()

delete() ----- destroy()

**Using routers**

Rest\_frame work provide DesfaultRouters from rest\_framework.rputres using which we can register a router.

**Register router**

Creaate a intanc eof DefaultRouter() class

On this instance call register() with endpoint url(test-view as you can see below), views class name and a basename.

router=DefaultRouter()

router.register('test-view',views.TestViewSet,basename='test-view-set')

'test-view' this will be appended with base url while showing this url

**Example of Viewset**

urls.py file

from django.contrib import admin

from django.urls import path, include

from testapp import views

from rest\_framework.routers import DefaultRouter

router=DefaultRouter()

router.register('test-view',views.TestViewSet,basename='test-view-set')

urlpatterns = [

    path('admin/', admin.site.urls),

    path('api-auth/', include('rest\_framework.urls')),

    path('',include(router.urls)),

]

Views.py file

class TestViewSet(ViewSet):

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

        colors=['Red','Green']

        msg={'msg':'Happy pongal'}

        return Response({'msg':'happy pongal','col':colors})

    def retrive(self,request,pk=None,\*args,\*\*kwargs):

        colors=['Red','Green']

        msg={'msg':'Happy pongal'}

        return Response({'msg':'responce from retrive methods'})

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

        print('data is: ',request.data)

        ser=NameSerializer(data=request.data)

        if ser.is\_valid():

            print('serialization completed')

            msg={'msg':'Operation completed'}

            return Response(msg)

        else:

            print('error in validations')

            return Response({'msg':ser.errors})

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

        return Response({'msg':"this response is from PUT method"})



**################################################################################**

# Function based view in rest\_framework #

################################################################################

REST framework also allows you to work with regular function based views. It provides a set of simple decorators that wrap your function based views to ensure they receive an instance of Request (rather than the usual Django HttpRequest) and allows them to return a Response (instead of a Django HttpResponse), and allow you to configure how the request is processed.

1. **@api\_view**

The core of this functionality is the api\_view decorator, which takes a list of HTTP methods that your view should respond to. For example, this is how you would write a very simple view that just manually returns some data:

@api\_view(http\_method\_names=[http\_methods]) ---------signature

Example –

@api\_view(['GET', 'POST'])

def hello\_world(request):

if request.method == 'POST':

return Response({"message": "Got some data!", "data": request.data})

return Response({"message": "Hello, world!"})

Questions

WAP to return the response in JSON format using rest\_framework. Response will contain employee model data

Views.py file

@api\_view(http\_method\_names=['GET'])

def get\_fucntion(request):

    qs=Employee.objects.all()

    eser=EmployeeSer(qs,many=True)

    return Response(eser.data)

models.py class

class Employee(models.Model):

    eno=models.IntegerField()

    ename=models.CharField(max\_length=100)

    esal=models.FloatField(null=True)

    eadd=models.CharField(max\_length=100)

serializer file

class EmployeeSer(ModelSerializer):

    class Meta:

        model=Employee

        fields='\_\_all\_\_'

urls.py file

urlpatterns = [

    path('admin/', admin.site.urls),

    path('api-auth/', include('rest\_framework.urls')),

    #path('api/',EmployeeCreateAPIView.as\_view()),

    path('api-func/',get\_fucntion,name='getting-data-using-fincaton-based-view'),

]

Output

It will have all employee object data on hitting url <http://127.0.0.1:8000/api-func/>

**API policy decorators**

To override the default settings, REST framework provides a set of additional decorators which can be added to your views. These must come after (below) the @api\_view decorator

@renderer\_classes(...)

@parser\_classes(...)

@authentication\_classes(...)

@throttle\_classes(...)

@permission\_classes(...)

**View schema decorator**

To override the default schema generation for function based views you may use the @schema decorator. This must come after (below) the @api\_view decorator

################################################################################

# Request in rest\_framework #

################################################################################

REST framework's Request class extends the standard HttpRequest, adding support for REST framework's flexible request parsing and request authentication.

As REST framework's Request extends Django's HttpRequest, all the other standard attributes and methods are also available.

For example the request.META and request.session dictionaries are available as normal.

**Request parsing**

REST framework's Request objects provide flexible request parsing that allows you to treat requests with JSON data or other media types in the same way that you would normally deal with form data.

**.data**

request.data returns the parsed content of the request body. This is similar to the standard request.POST and request.FILES attributes

################################################################################ **# Response in rest\_framework #**

**################################################################################**

It takes python native data and convert that into JSON types as for HttpResponse.

REST framework supports HTTP content negotiation by providing a Response class which allows you to return content that can be rendered into multiple content types, depending on the client request.

Unlike regular HttpResponse objects, you do not instantiate Response objects with rendered content. Instead you pass in unrendered data, which may consist of any Python primitives.

The renderers used by the Response class cannot natively handle complex datatypes such as Django model instances, so you need to serialize the data into primitive datatypes before creating the Response object.

Response(data, status=None, template\_name=None, headers=None, content\_type=None)

--------- Signature

* **data:** The serialized data for the response.
* **status:** A status code for the response. Defaults to 200. See also status codes
* **template\_name:** A template name to use if HTMLRenderer is selected
* **headers:** A dictionary of HTTP headers to use in the response.
* **content\_type:** The content type of the response. Typically, this will be set automatically by the renderer as determined by content negotiation, but there may be some cases where you need to specify the content type explicitly.

**Attributes**

* **.data** ----------- The unrendered, serialized data of the response.
* **.status\_code** ---------- The numeric status code of the HTTP response.
* **.content** --------------- The rendered content of the response. The .render() method must have been called before .content can be accessed.
* **.template\_name** ------The template\_name, if supplied. Only required if HTMLRenderer or some other custom template renderer is the accepted renderer for the response.
* **.accepted\_renderer**-------The renderer instance that will be used to render the response.
* **.accepted\_media\_type** -----The media type that was selected by the content negotiation stage.
* **.renderer\_context** -------- A dictionary of additional context information that will be passed to the renderer's .render() method.

**Routers**

It provides functionality for automatically determining how the URLs for an application should be mapped to the logic that deals with handling incoming requests.

It automatically generates url based on how we are registerting Routers.

Rest framework provides below router classes-

1. SimpleRouter
2. DefaultRouter
3. Custom Routers

**Advantages for Routers**

* Routers generate automatic URL patterns and maps every URL to its respective method based on a type of the request.
* We can avoid configuring the URL's with views.
* Router generates standardized url patterns.
* We can expect consistent behaviour from viewsets and routers

**Registering Routers**

For registering any router we need to use register() on that router instance.

router\_instance.register(prefix,viewset\_class,basename=?)

prefix and viewset\_name are mandatory

* **prefix** - The URL prefix to use for this set of routes ----------- mandatory
* **viewset\_class** - The viewset class -------- mandatory
* **basename** - The base to use for the URL names that are created ---- optional

**SimpleRouter**

This router includes routes for the standard set of list, create, retrieve, update, partial\_update and destroy actions. The viewset can also mark additional methods to be routed, using the @action decorator.

By default the URLs created by SimpleRouter are appended with a trailing slash. This behavior can be modified by setting the trailing\_slash argument to False when instantiating the router.



**DefaultRouter**

This router is similar to SimpleRouter as above, but additionally includes a default API root view, that returns a response containing hyperlinks to all the list views. It also generates routes for optional .json style format suffixes.

As with SimpleRouter the trailing slashes on the URL routes can be removed by setting the trailing\_slash argument to False when instantiating the router



Generic Views

ListAPIView

CreateAPIView

function based views in rest\_framework

**APIView, ListAPIView, CreateAPIView**

ListAPIView and CreateAPIViews are views are the bulitin classes to list,create etc. , methods.

There are available in rest\_framework.genericviews module

**Using APIView**

Fetch all data from using APIViews

Views.py file

class EmployeeView(ListAPIView):

    #queryset=Employee.objects.all()

    serializer\_class=EmployeeSer

    def get\_queryset(self):

        name=self.request.GET.get('ename')

        qs=Employee.objects.all()

        if name is not None:

            qs=qs.filter(ename\_\_icontains=name)

        return qs

myser.py file

from rest\_framework.serializers import Serializer, ModelSerializer

from testapp.models import Employee

class EmployeeSer(ModelSerializer):

    class Meta:

        model=Employee

        fields='\_\_all\_\_'

views.py file

urlpatterns = [

    path('admin/', admin.site.urls),

    path('api-auth/', include('rest\_framework.urls')),

    #path('api/',EmployeeView.as\_view()),

    path('api/',EmployeeCreateAPIView.as\_view()),

]

Now we will get all data of Employee model class(as fetching data from Employee model class) in below format---



**Using LISTAPIView**

ListAPIView extends APIView class. This is bultin class in rest\_framwork for getting data.

If using LISTAPIView then don’t need to write any return statement or so any serializations.

View.py

class EmployeeView(ListAPIView):

    queryset=Employee.objects.all()

    serializer\_class=EmployeeSer

just by adding above line we will get even the same result as what we got from API view.

* queryset ----- this is builtin variable, which hold queryset type data
* serializer\_class ---- this is building variable which is name of serializer class to be used to serialization.

**Getting data by some name/id using LISTAPIView**

We can do it using the query string and overriding the get\_queryset().

query string ---- this is the string added in url to fetch any specific data. E.g ---

127.0.0.1:8000/api/?name=’rakesh’

name=’rakesh’ ----- this us query string. Here I am trying to fetch data for variable name.

views.py file

class EmployeeView(ListAPIView):

    queryset=Employee.objects.all()

    serializer\_class=EmployeeSer

    def get\_queryset(self):

        name=self.request.GET.get('ename')

        qs=Employee.objects.all()

        if name is not None:

            qs=qs.filter(ename\_\_icontains=name)

        return qs

* get\_queryset() ---- this method is used for getting queryset data/data from database. That could be all or some filtered data.

**CreateAPIView**

This is used for creating objects in database.

If we are not using CreateAPIView then we will have to provide implementation of post() to create object in database.

Views.py file

class EmployeeCreateAPIView(CreateAPIView):

    # add some validation login if you want

    queryset=Employee.objects.all()

    serializer\_class=EmployeeSer

Change urls.py file accordingly.

This much is enough for ceating any resource in Database. When we send data from browasable API, it will be itself created and saved in to databse.

If we can we can add some data validation logic.



Once we send data from above form(gererated by rest browasable API) it will be saved to DB

################################################################################# Generic View #

################################################################################

The generic views provided by REST framework allow you to quickly build API views that map closely to your database models.

If the generic views don't suit the needs of your API, you can drop down to using the regular APIView class, or reuse the mixins and base classes used by the generic views to compose your own set of reusable generic views.

The following classes are the concrete generic views. If you're using generic views this is normally the level you'll be working at unless you need heavily customized behavior.

**1.)CreateAPIView**

Used for create-only endpoints.

Provides a post method handler.

Extends: GenericAPIView, CreateModelMixin

**2.)ListAPIView**

Used for read-only endpoints to represent a collection of model instances.

Provides a get method handler.

Extends: GenericAPIView, ListModelMixin

**3.)RetrieveAPIView**

Used for read-only endpoints to represent a single model instance.

Provides a get method handler.

Extends: GenericAPIView, RetrieveModelMixin

**4.)DestroyAPIView**

Used for delete-only endpoints for a single model instance.

Provides a delete method handler.

Extends: GenericAPIView, DestroyModelMixin

**5.)UpdateAPIView**

Used for update-only endpoints for a single model instance.

Provides put and patch method handlers.

Extends: GenericAPIView, UpdateModelMixin

**6.)ListCreateAPIView**

Used for read-write endpoints to represent a collection of model instances.

Provides get and post method handlers.

Extends: GenericAPIView, ListModelMixin, CreateModelMixin

**7.)RetrieveUpdateAPIView**

Used for read or update endpoints to represent a single model instance.

Provides get, put and patch method handlers.

Extends: GenericAPIView, RetrieveModelMixin, UpdateModelMixin

**8.)RetrieveDestroyAPIView**

Used for read or delete endpoints to represent a single model instance.

Provides get and delete method handlers.

Extends: GenericAPIView, RetrieveModelMixin, DestroyModelMixin

**9.)RetrieveUpdateDestroyAPIView**

Used for read-write-delete endpoints to represent a single model instance.

Provides get, put, patch and delete method handlers.

Extends: GenericAPIView, RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin

**Methods in rest\_framework.generics module class**

1. **get\_queryset(self)**

* Returns the queryset that should be used for list views, and that should be used as the base for lookups in detail views. Defaults to returning the queryset specified by the queryset attribute.
* This method should always be used rather than accessing self.queryset directly, as self.queryset gets evaluated only once, and those results are cached for all subsequent requests.
* Can be overridden to provide dynamic behavior, such as returning a queryset, that is specific to the user making the request.

def get\_queryset(self):

qs=#get query on same filerting or something

return qs

1. **get\_serializer\_class(self)**

* Returns the class that should be used for the serializer. Defaults to returning the serializer\_class attribute.
* Can be overridden to provide dynamic behavior, such as using different serializers for read and write operations, or providing different serializers to different types of requrement.

def get\_serializer\_class(self):

if self.request.user.is\_staff:

return FullAccountSerializer

return BasicAccountSerializer

There are below some more methods which can be used while we are writing custom views using GenericAPIView.

1. **get\_serializer\_context(self)** - Returns a dictionary containing any extra context that should be supplied to the serializer. Defaults to including 'request', 'view' and 'format' keys.
2. **get\_serializer(self, instance=None, data=None, many=False, partial=False)** - Returns a serializer instance.
3. **get\_paginated\_response(self, data)** - Returns a paginated style Response object.
4. **paginate\_queryset(self, queryset)** - Paginate a queryset if required, either returning a page object, or None if pagination is not configured for this view.
5. **filter\_queryset(self, queryset)** - Given a queryset, filter it with whichever filter backends are in use, returning a new queryset.

**Some attributes**

rest\_framework.generics module provide below attributes-

1. **serializer\_class**--The serializer class that should be used for validating and deserializing input, and for serializing output. Typically, you must either set this attribute, or override the get\_serializer\_class() method.
2. **queryset**--The queryset that should be used for returning objects from this view. Typically, you must either set this attribute, or override the get\_queryset() method

**.user**

request.user typically returns an instance of django.contrib.auth.models.User, although the behavior depends on the authentication policy being used.

If user not authenticated then retuns Anonymous user

**.content\_type**

request.content\_type, returns a string object representing the media type of the HTTP request's body, or an empty string if no media type was provided.

**.stream**

request.stream returns a stream representing the content of the request body.