26/1/22

→ Developing WEB API's by using 3rd Party Django Rest Framework.

Several 3rd party frameworks are available like :- ① Tastify
② DRF - Django Rest Framework
etc

- DRF is most commonly used & easy to use framework to build REST API's for Django Applications. (Django-rest-framework.org)

→ Serializers :- DRF Serializers are responsible for following activities :-
① Serialization ② Deserialization ③ Validation
Note :- DRF Serializers will work very similar to Django Forms & Model Form Classes.

① Serialization :- The process of Converting Complex objects like Model objects & Querysets to python native data types like dictonary 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

- Start project, Startapp
- Add 'rest_framework' in Installed apps & url in url-path
- Models.py

```
Class Employee(models.Model):
    eno, ename, esal, eaddr, ___, ___
```

- Create Serializers.py (In testapp Create)

```
from rest_framework import Serializers
Class EmployeeSerializer(serializers.Serializer):
    eno = Serializers.IntegerField()
    ename = Serializers.CharField(max_length=64)
    eaddr = ___, ___
    esal = ___, ___
```

- Admin.py, makemigrations, Migrate

pip freeze > requirements.txt → To know all things to send client

PAGE NO.
DATE / /

→ Converting Employee objects to python Native Data type by using EmployeeSerializer (serialization process) :-

```
>> shell
>> from testapp.models import Employee
>> from testapp.serializers import EmployeeSerializer
>> emp = Employee(eno=100, ename='sunas', esal=5000, eaddr='bang')
>> eserializer = EmployeeSerializer(emp)
>> eserializer.data
{'eno':100, 'ename':'sunas', esal:1000, eaddr:'bang'}
```

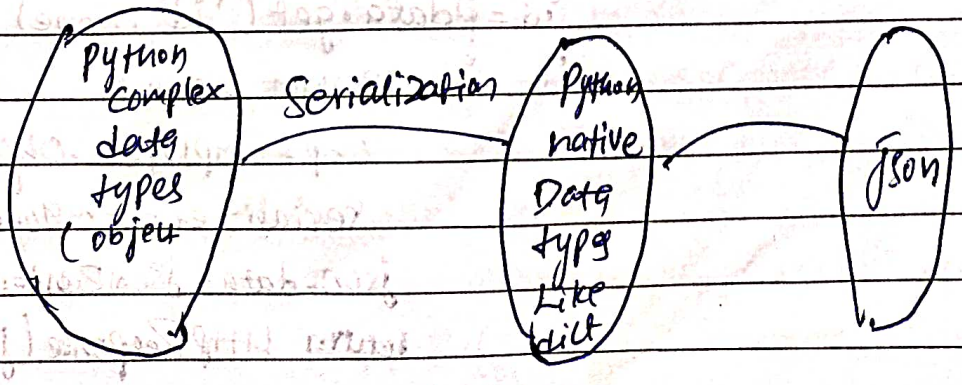
just we have converted Employee object to python native data type (dict)

→ 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":"sunas", esal=5000, eaddr=bang"}'
```

→ How to perform Serialization for QuerySet :-

```
>> qs = Employee.objects.all()
>> qs
>>> eserializer = EmployeeSerializer(qs, many=True)
>> eserializer.data
>> json_data = JSONRenderer().render(eserializer.data)
>> json_data
```



22/6/22

② Deserialization:

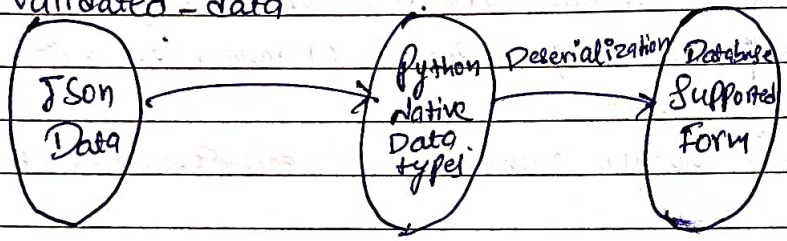
The process of Converting python native data types(dict) to Complex data types like Model objects is called deserialization.

- First we have to Convert json_data to python native data type.

```
import io
from rest_framework.parsers import JSONParser
stream = io.BytesIO(json_data)
pdata = JSONParser().parse(stream)
```

Now, we have to Convert python native data type to database supported Complex type (deserialization)

```
Serializer = EmployeeSerializer(data=pdata)
Serializer.is_valid()
Serializer.validated_data
```



To get & Create by using Serializers

→
```
@method_decorator(csrf_exempt, name='dispatch')
Class EmployeeCRUDCBV(View):
    def get(self, request, *args, **kwargs):
        json_data = request.body
        stream = io.BytesIO(json_data)
        pdata = JSONParser().parse(stream)
        id = pdata.get('id', None)
        if id is not None:
            emp = Employee.objects.get(id=id)
            Serializer = EmployeeSerializer(emp)
            json_data = JSONRenderer().render(Serializer.data)
            return HttpResponse(json_data, Content_type='application/jso
        as = Employee.objects.all()
        Serializer = EmployeeSerializer(as, many=True)
        json_data = JSONRenderer().render(serializer.data)
        return HttpResponse
```

In test.py

```
import requests, import json
BASE_URL = 'http://127.0.0.1:8000/'
ENDPOINT = 'api/'
def get_resource(id=None):
        data = {}
        if id is not None:
            data = {
                'id': id
            }
        resp = requests.get(BASE_URL + ENDPOINT, data = json.dumps(data))
        print(resp.status_code)
        print(resp.json())
```

To post: In test.py

```
def Create_resource():
        new_emp = {
            'eno': 500, ename - ,.- }
    resp = , —
Create_resource()

def Post(self, request, *args, **kwargs):
        json_data = request.body
        stream = —, —      , pdata = —, —
        Serializer = EmployeeSerializer(data=pdata)
        if Serializer.is_valid():
            Serializer.save()
            msg = {'msg': 'Resource Created Succesfull!!"}
            json_data = JSONRenderer().render(msg)
            return HttpResponse( —, —— )
        json_data = JSONRenderer().render(serializer.errors)
        return HttpResponse( —, —— )
```

Inside Serializers.py

```
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 = ——/——('ename', —/—ename)
    esal, eaddrs —/——
    instance.save()
    return instance

def put(self, request, *args, **kwargs):
    json_data = request.body
    stream = ——/——
    pdata = ——/——
    id = pdata.get('id')
    emp = Employee.objects.get(id=id)
    Serializer = EmployeeSerializer(emp, data=pdata, partial=True)
    if ——/——
    msg = {'msg': 'Resource updated successfull'}
    json_data = ——/——
```

In test.py

```
def update_resource(id):
    new_data = {
        'id': id,
        'esal': 5000,
        'eaddrs': 'G700',
    }
    resp = ——/——
update_resource(7)
```

In se

```
def delete(self, request, *args, **kwargs):
        json_data = ---
        Stream = ---
        pdata = ---
        id = pdata.get('id')
        emp = Employee.objects.get(id=id)
        emp.delete()
        msg = {'msg': 'Resourced deleted successfull!'}
        json_data = ---
        return HttpResponse(---)
```

In test.py:-

```
def delete_resource(id):
        data = {
            'id': id
        }
        resp = requests.delete(---)
```

→ Validation by Using Serializers:-

① Field level serializers ③ By using validators.

② Object level

① Field level :- eg:- To Check esal should be minimum 5000

In serilizers.py:-

```
Class EmployeeSerializer(Serializers.Serializer):
        def Validate_esal(self, value):
            if value < 5000:
                raise serializers.ValidationError("Employees
                Salary should be 5000")
            return value
```

② Object Level :- To perform validations for multiple fields simultaneously then we should go object level

eg: If 'ename' is Suhas Salary should be min 50,000

```
def validate(self, data):
    ename = data.get('ename')
    esal = data.get('esal')
    if ename.lower() == 'suhas':
        if esal < 50000:
            raise serializers.ValidationError("
Employee Salary should be Minimum 50000")
    return data
```

Use-Cases:-

① First entered pwd & re-entered pwd should be same.

② First entered Account No & re-entered Account No should be same.

23/2/22

③ Vadiation by using validator Field :-

In serializers.py: Outside the class.

```
def multiples_of_1000(value):
    if value % 1000 != 0
        raise serializers.ValidationError("Employee
Salary should be Multiples of 1000')

Class EmployeeSerializer(serializers.Serializer):
    esal = Serializers.IntegerField(validators=[multiples_of_1000,])
```

→ Model Serializers:-

- If our serializable objects are Django objects, then it is highly recommended to go for ModelSerializer.
- ModelSerializer Class is exactly same as regular Serializer Class except the following differences.

① The fields will be considered automatically based on the Model & we are not required to specify explicitly.

② It provides default implementation of create() & Methods() methods.

Note: ModelSerializer won't provide any shortcut & extra functionality.
We can define ModelSerializer class as follows:-
In Serializer.py

```
Class EmployeeSerializer(Serializer.ModelSerializer):
    Class Meta:
        model = Employee
        fields = "__all__"
```

→ Django Rest Framework Views :-
Two types of views : ① APIView [like FBV].
② Viewset [Like CBV]

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 standard Http methods as functions like get(), post(), put() etc.
- 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.

→ How to send Response in APIViews :- To send Response to the partner/client application, DRF provides Response class. It will Convert input data to json format automatically.

```
from rest_framework.views import APIView
from rest_framework.response import Response
from testapp.serializers import NameSerializer

Class TestApiView(APIViews):
    def get(self, request, *args, **kwargs):
        Color = ['red', 'blue', 'green']
        return Response({'msg': "Hello world", 'colors': Colors})

    def post(self, request, *args, **kwargs):
        serializer = NameSerializer(data = request.data)
        if serializer.is_valid():
            name = serializer.data.get('name')
            msg = 'Hello {}, Happy to meet you'.format(name)
            return Response({'msg': msg})
        else:
            return Response({msg': msg})
```

In serializers.py

```
from rest_framework import serializers
    class NameSerializer(serializer.Serializer):
        name = serializers.CharField(max_length=7)

    def put(self, request, pk=None)
        return Response({'msg': 'Response from put Method'})
```

Same for patch & delete.

In url's.py :- path('', Views.TestApiView.as_view())

→ Viewsets: By using viewsets, we can provide business logic for our API views.

- It is alternative to APIViews class.
- list() → To get all resources/records/objects.
- retrieve() → To get a specific resource.
- Create() → To create
- update(), partial_update(), destroy()

```
from rest_framework.viewsets import Viewset
Class TestViewSet(ViewSet):
        def list(  —,  —>)
        def Create(  —,  —>
        def ~~test, patch~~ retrieve()  (—,—)
            update(), Partial_update(), destroy()
```

In Url's.py

```
from testapp import views
from rest_framework.routers import DefaultRouter
router = DefaultRouter()
router.register('test-view-set', views.TestViewset, basename="test-view")
        Url patterns = [
                path('', include(router.urls))
                ]
```

24/9/22

→ Demo application on APIViews:

In models.py: Class Employee(models.Model):
eno, ename, esal, eaddrs

admin.py, makemigrations, migrate, superuser

In serializers.py

```
from rest_framework.serializers import ModelSerializer
Class EmployeeSerializer(ModelSerializer):
        Class Meta:
                model = Employee
                fields = '__all__'
```

In views.py: from rest_framework.views import APIView
from rest_framework.response import Response

```
Class EmployeeListAPIview(APIview):
    def get(self, request, *args, **kwargs):
        qs = Employee.objects.all()
        Serializer = EmployeeSerializer(qs, many=True)
        return Response(Serializer.data)
```

In urls.py:

```
path('api/', views.EmployeeListAPIview.as_view())
```

Note: In the above example, Serializer is responsible to Covert queryset —to→ Python native data type (dict), where Response object dict —to→ json.

→ To list all Employees by using ListAPIview Class (same methods above can be written as) If we want to get all list of all resources then ListAPIview Class is best suitable.

```
from rest_framework import generics
Class EmployeeAPIView(generics.ListAPIview):
    queryset = Employee.Objects.all()
    Serializer_class = EmployeeSerializer
```

(fixed names, can't be changed)

→ How to implement Search operation: If we want to implement search operation, we have overwride get_queryset() method, in our view class.

```
from rest_framework import generics
Class EmployeeAPIView(generics.ListAPIview):
    Serializer_Class = EmployeeSerializer
    def get_queryset(self):
        qs = Employee.objects.all()
        name = self.request.GET.get('ename')
```

```
if name is not None:
        qs = qs.filter(ename__icontains=name)
return qs
```

→ In Command prompt:- (or browser)

http://127.0.0.1:8000/api/?ename=sukas

→ To implement Create operation with ListCreateView Using APIView

```
from rest_framework import generics
Class EmployeeCreate(generics.ListCreateAPIView):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer.
```

→ To implement Retrive operation (detail operation)

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

In urls.py :- re_path('api/(?P<id>\d+)/$', views.EmployeeReteriveAPI...

→ For update :- Class EmployeeUpdateAPIView(generics.UpdateAPIView):

→ For Delete (Destroy) :- Class EmployDeleteApiview(generics.DestroyAPIView):

→ Implementing All Crud operation by using two end opints

```
Class EmployeeListCreateAPIView(generics.ListCreateAPIView):

Class EmployeeRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroy...
```

→ The following are predefined APIView Classes

① ListAPIView
② CreateAPIView
③ RetrieveAPIView
④ DestroyAPIView
⑤ ListCreateAPIView
⑥ RetrieveUpdateAPIView
⑦ RetrieveDestroy — / —
⑧ RetrieveUpdateDestroyAPIView

→ Mixins: Mixins are reusable Components. DRF Provides several mixins to provide basic view behaviour.
imported from rest_framework.mixins.

Direct child classes of object Contains only Methods.

Various Mixins available are:

① ListModelMixin
② CreateModelMixin
③ UpdateModelMixin
④ DestroyModelMixin.

(Internally APIViewSet is developed by using Mixins.)

→ Demo application using Viewset: → For standard operations go for viewset, for customised operations go for APIView

Create Models, admin, make migrations, migrate

In serializers.py:

```
from rest_framework.serializers import ModelSerializer
from testapp.models import Employee
class EmployeeSerializer(ModelSerializer):
    class Meta:
        model = Employee
        fields = "__all__"
```

In views.py:

```
from rest_framework.viewsets import ModelViewSet
class EmployeeCRUDCBV(ModelViewSet):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer
```

In urls.py:

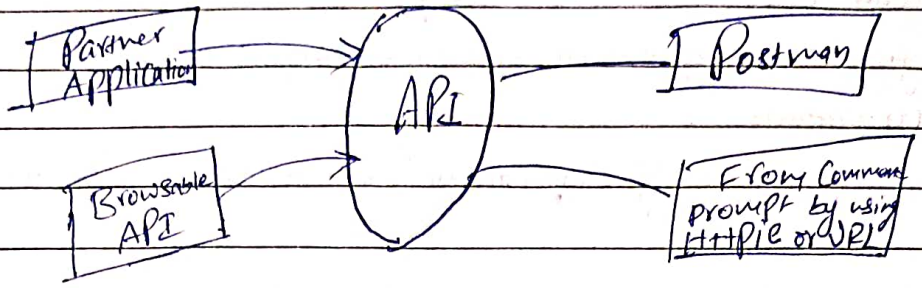
```
from rest_framework import routers
router = routers.DefaultRouter()
router.register('api', views.EmployeeCRUDCBV, basename='api')
```

```
path('', include(router.urls)),
```

Read operation :- Get, Options, Head → Safe Methods.
Write operation :- Post, Put, Patch, Delete

Adora
PAGE NO.
DATE / /

→ API Functionality Testing API Functionality :-



→ Authentication & Authorization :- The API's which are developed up this can accessed by anyone & everyone. So to provide security for API we should go for Authentication & Authorization.

→ Authentication :- The process of validating user is called authentication. By using username & password (or) tokens.
DRF Provides several inbuilt authentication Mechanisms :-
① Basic authentication ② Session ③ Token ④ JWT (Json Web Token)

→ Authorization :- The process of Validating Access permissions of User is Called Authorization. ① AllowAny ② IsAuthenticated ③ IsAdminUser ④ IsAuthenticatedOrReadOnly ⑤ DjangoModelPermissions

→ Token Based Authentication :- Every request will be authenticated by using token, which is nothing but authentication.
- To implement TokenAuthentication we have to use 3rd party application 'rest_framework.authtoken'

25/2/22
→ Steps to Implement Token-Authentication :-
① we have to include authtoken application in Installed_apps list inside settings.py.

```
INSTALLED_APPS = [ 'rest_framework'
                   'rest_framework.authtoken',
                   'testapp' ]
```

② Perform migrations so that required tables will be created in admin
③ 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. In urls.py

```
from rest_framework.authtoken import views
urlpatterns = [
    path('get-api-token', views.obtain_auth_token, name='get-api-token')
```

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="Rubas"
password="Rubas"
```

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

→ Enabling Authentication & Authorization (permissions) for View Class:

We can enable authentication & authorization for our view class either locally (or) Globally.

Enabling locally: Our application contain several view classes. If we want to enable authentication & authorization for a particular class then we have to go for local approach.

```
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated.
    Class EmployeeCRUDCBV (ModelViewset):
        ——— / ———
        ——— / ———
        authentication_classes = [TokenAuthentication,]
        permission_classes = [IsAuthenticated,]
```

From Command Prompt
how can you authenticate

```
http http://127.0.0.1:8000/api/
"authorization:Token ——"
```

Enabling Globally: If we want to enable authentication & authorization for all view classes, we have to use this approach.

We have to add the following lines in settings.py

From postman:: In headers part
key → Authorization
value → Token ——

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

→ Various Possible Permission Classes:

① AllowAny: Allows unrestricted access irrespective of authenticated (or) not.
② IsAuthenticated: Only Authenticated users has access
③ IsAuthenticatedOrReadOnly :- You can perform Read operations not Write operations. To access end point → Authentication must be required (Token)
④ DjangoModelPermissions :- Get → Authentication is enough model permissions not required. Post, Put, Patch, Delete → authentication + Model permissions
⑤ DjangoModelPermissionsOrAnonReadOnly
⑥ IsAdminUser :- user → staff status = True → admin

→ JWT (Json Web Token) Authentication :-

- pip install djangorestframework-jwt

→ (website : django-rest-framework-jwt)

- We can access JWT in 3 ways:

① access Token → 5 mins
② refresh Token → Non-expired token can be "refreshed" to obtain new token with renewed expiration time.
③ verify Token

In url's.py:

```
from rest_framework_jwt.views import obtain_jwt_token, refresh_jwt_token,
verify_jwt_token
urlpatterns = [
    path('auth-jwt/', obtain_jwt_token),
    path('auth-jwt-refresh/', refresh_jwt_token),
    path('auth_jwt-verify/', verify_jwt_token),
]
```

→ To access token :

In Command prompt :

http . http://127.0.0.1:8000/auth-jwt/ username="sudas" password="sudas"

Same in postman (In body username, password)

Step 1 : In postman : headers → Content-type - application/json Body → Authorization
In api → headers → Authorization JWT 'token'

→ To refresh token :

In setting.py

JWT_AUTH = {
'JWT_ALLOW_REFRESH': True, }

In CP :

http POST http://127.0.0.1:8000/auth-jwt-refresh/ token= " "

In postman :

In Headers → key :- Content-type value :- application/json

In postman In body :- token (→ To refresh) " username, password → To get token

In partner application: http://127.0.0.1:8000/api/ → Get → In headers Authorization JWT token

→ To verify Token :-

In cmd : http post ____ , ____ /auth-jwt-verify/ token= " "

→ Limitation in Token Authentication :-

① For every request DRF will communicate with the database to validate token & to identify corresponding User. This database interaction creates performance issues & Scalability will be down. To overcome this we should go for JWT Authentication.

→ How to enable JWT Token Authentication for View classes :-

In views.py: from rest_framework_jwt.authentication import JSONWebTokenAuthentication
from rest_framework.permissions import IsAuthenticated

Class EmployeeCRUDCBV (ModelViewSet) :

____ , ____

authentication_classes = [JSONWebTokenAuthentication,]
permission_classes = [IsAuthenticated,]

In settings.py : (optional)

'JWT_AUTH_HEADER_PREFIX': 'JWT',

In postman :

Create Token & Get requested URL : 127.0.0.1:8000/api/
In Header Authentication JWT token)
In Body :- username & password
In Headers :- Content-type application/json

→ To get JWT-token

→ Pagination : DRF Provides several pagination classes :
① PageNumberPagination ③ CursorPagination.
② LimitOffsetPagination

① Pagination :
Enabling Globally :

```
REST_FRAMEWORK = {
 'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
'PAGE_SIZE': 10
}
```

Enabling Locally :-
Create pagenation.py

```
Class MyPagination(PageNumberPagination):
    Page_size=5
    page_query_param = 'mypage'        (Optional)
    Page_size_query_param = 'num'
    max_page_size = 10        (optional)
```

default is 'page' ←

In views.py

```
from testapp.pagination import MyPagination.
Class EmployeeCRUDCBV (ModelViewset):
    pagination_class = MyPagination.
```

→ http://127.0.0.1:8000/api/?mypage=10&num=10

② LimitOffsetPagination :

```
from rest_framework.pagination import LimitOffsetPagination
Class MyPagination2(LimitOffsetPagination):
    default_limit=15
```

③ CursorPagination :

```
import CursorPagination
Class MyPagination3(CursorPagination):
    Ordering = 'esal'
    Page_size=5
```

→ DRF - Filtering :

For filtering :

In settings.py

REST_FRAMEWORK = {

'DEFAULT_FILTER_BACKENDS' : ('rest_framework.filters.SearchFilter',),

}

In views.py :

Class ___ / ___

Search_fields = ('ename',)

= means exact match ⇒ ('Aeno',)

^ means start with ⇒ ('=eno')

→ Ordering :

In settings.py

same as above

REST ___ / ___ = {

___ / ___ ('rest_framework.filters.OrderingFilter')

}

(In Search button)

Class EmployeeListView(generics.ListAPIView):

___ / ___

Ordering_fields = ('eno', 'esal')

28/2/22

→ DRF - Nested Serializers :

In Models.py

Class Author(models.Model):

first_name = models.CharField(max_length=64)

last_name, subject = / ___ / ___

def __str__(self):

return self.first_name.

```
Class Book(models.Model):
    title = models.CharField(max_length=256)
    author = models.ForeignKey(Author, on_delete=models.CASCADE, related_name='books_by_author')
    release_date = models.DateField()
    ratings = models.IntegerField()
    def __str__(self):
        return self.title
```

make migrations, migrate, admin.py, createsuperuser.

→ In serializers.py

```
from rest_framework import serializers
from testapp.models import Author, Book

Class BookSerializer(serializers.ModelSerializer):
    Class Meta:
        model = Book
        fields = '__all__'
Class AuthorSerializer(serializers.ModelSerializer):
    books_by_author = BookSerializer(read_only=True, many=True)
    Class Meta:
        model = Author
        fields = '__all__'
```

In views.py:

```
from rest_framework import generics
Class AuthorListView(generics.ListCreateAPIView):
    queryset = Author.objects.all()
    serializer_class = AuthorSerializer

Class AuthorView(generics.RetrieveUpdateDestroyAPIView):
```

Same view for books

(Not often used)

-> Basic Authentication (models, serializers, admin, Viewset views)

In views.py

```
from testapp.serializers import EmployeeSerializers
from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated.

Class EmployeeCRUDCBV(ModelViewset):
        queryset = Employee.objects.all()
        Serializer_class = EmployeeSerializer
        authentication_classes = [BasicAuthentication,]
        permission_classes = [IsAuthenticated,]
```

urls:

```
from rest_frame import routers
```

(Same as routers url's)

Problems with Basic Authentication

1. Username & pwd will send to the server in base-64 encoding, which is not secured.
2. We cannot customize look & feel of authentication form.

(Not often used)

-> Session Authentication: By using Django Auth application.

Using Django Auth

In urls.py

```
path('accounts/', include('django.contrib.auth.urls')),
```

Create In project folder -> templates -registration- login.html

In login.html

```
<form method="post">
    {% csrf_token %}
    {{form.as_p}}
    <button type='submit'>login</button>
```

In views.py: from restframework.authentication import SessionAuthentication

```
Class ...
    authentication_classes = [SessionAuthentication].
    Permission_classes = [IsAuthenticated]
```

→ Consuming 3rd Party Applications to get info:

Get signup with ipstack / Get url

In views.py:

```
import requests
def get_geographic_info(request):
    ip = request.META.get('HTTP_X_FORWARDED_FOR', '') or
         request.META.get('REMOTE_ADDR')
url = 'http://api.ipstack.com/' + str(ip) + '?access_key=___'
response = requests.get(url)
data = response.json()
return render(request, 'testapp/info.html', data)
```

In info.html

```
<h1> Your IP Address : {{ip}}
<h2> Your Continent name : {{Continent_name}}
```

30
(50) → Providing API For Django Applications :-

- step-1 : Add 'rest_framework' in settings.py
- step-2 : In your testapp Create a new folder api
- step-3 : In api folder Create views.py, urls.py, serializers.py

In Serializers.py

```
from rest_framework.serializers import ModelSerializer
from testapp.models import Employee
Class EmployeeSerializer(ModelSerializer):
    Class Meta:
        model = Employee
        fields = '__all__'
```

In views.py

```
from rest_framework.viewsets import ModelViewSet
from testapp.api.serializers import EmployeeSerializer
Class EmployeeCRUDCBV(ModelViewSet):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer
```

In urls.py:

```
from testapp.api.views import EmployeeCRUDCBV
from rest_framework import routers
router = routers.DefaultRouter()
router.register('info', EmployeeCRUDCBV)
urlpatterns = [ path('', include(router.urls)) ]
```

In Application level urls.py

```
path('api/', include('testapp.api.urls')),
```

→ POSTMAN → To test api functionality

Swagger → To test api functionality & To get documentation for api.

In settings.py

① Add 'rest_framework_swagger' in apps

② In setting.py add in Templates = [

```
'libraries': {
        'staticfiles': 'django.templatetags.static',
    }
```

In settings.py add

```
REST_FRAMEWORK = {'DEFAULT_SCHEMA_CLASS': 'rest_framework.schemas.
                  coreapi.AutoSchema'}
```

In urls.py

```
from rest_framework_swagger.views import get_swagger_view
schema_view = get_swagger_view(title="Employee API Using Swagger")
urlpatterns = [
        path('docs', schema_view),
    ]
```

Interview Questions:

① What is REST & RESTFUL API's ?

REST is basically an architecture where as RESTFUL API is an that Implements REST.

② Explain the architectural Style for Creating any web API ?

(1) We can use HTTP for Client server Communication.
(2) We can use xML/JSON to send & recieve messages. XML/JSON acts as formatting Language.
(3) Each Resource/service Can be accessed by unique URL
(4) Stateless Communication.

③ What is "Resource" in REST ?

Any Database Record which can be Image, Video File, html file can be Considered as Resource.

④ Which markup languages used in REST API ?

XML / JSON / YAML

⑤ What are various HTTP Methods support by REST ?

(1) GET (2) PUT (3) Delete
(2) POST (4) PATCH (6) Options :- It Represents various HTTP Methods supported to access a resource.
(7) HEAD :- It represents Header part of the GET Response. It provides META Info.

⑥ What is idempotent request ?

By Repeating the request Multiple times, If we are getting the same response such type of request is Called idempotent request. GET, PUT is Idempotent where as POST is not idempotent.

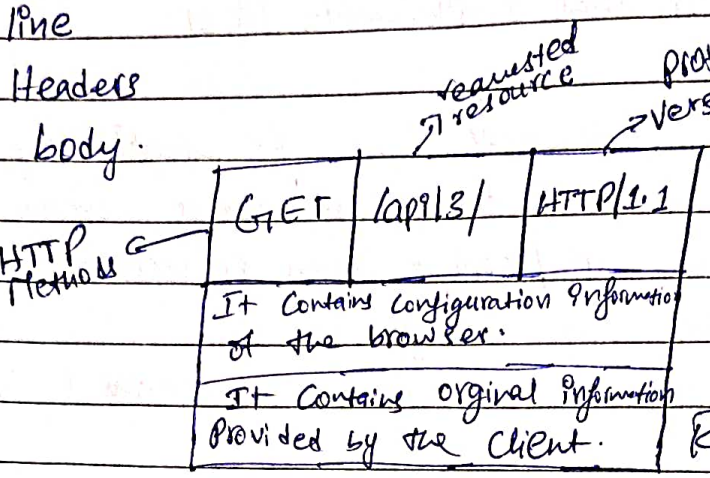
⑦ Difference b/w POST & PUT, PATCH ?

POST → Is to Create a new resource.
PUT → Is to Update an existing Resource Fully.
PATCH → Partial Updatation of Resource (only one or two field can be Updated)

⑧ Explain the format of HTTP Request ?

HTTP Request contains 3 parts :-

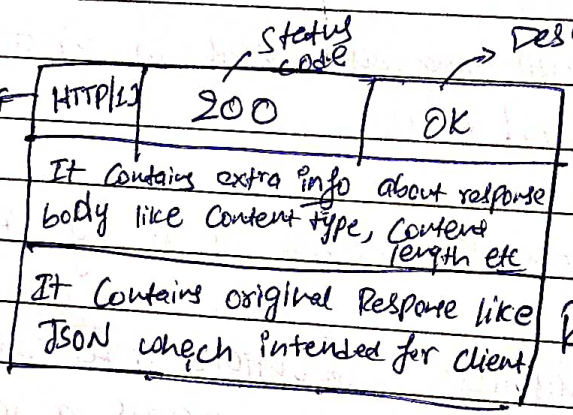
① Request line
② Request Headers
③ Request body.



⑨ Explain the format of HTTP Response ?

HTTP Response contains mainly 3 components :-

① Status Line
② Response Headers
③ Response Body.



⑩ What is Payload in RESTful web services ?

The payload is the data what we are transporting from client application to server application.

⑪ What is the term "Statelessness" with respect to RESTful Web Services ?

Statelessness means Complete isolation. Server won't maintain any information of the client. Every request to the server is treated as an independent new request.

(12) What is Caching Mechanism?

Caching is the process in which server response is stored so that a cached copy can be used when required & there is no need of generating the same response again.

(13) What is Status Code?

HTTP status code represents the status of the response like success (or) fail etc.

(14) List out some common status codes experienced in your previous project?

(1) 200 → This indicates success
(2) 201 → This indicates resource has been succesfully created.
(3) 204 → This indicates there is no content in the response body.
(4) 404 → This indicates there is no method available.
(5) 400 → BAD Request, states that invalid input is provided.
(6) 401 → Unauthorized Authentication information is not correct.
(7) 403 → Forbidden csrf verification fails
(8) 500 → Internal Sever Error.

DRF Interview Questions:

(15) From Command prompt, how to send HTTP request?

By Using CURL & HTTPie

(16) What are Mixins & explain Usage?

Mixins are special type of inheritence in python. Mixins meant for code reusability.

(17) What is serialization?

Converting object from One form to another is Called Serialization
eg: Converting python dict object to JSON

(16) In how many ways we can perform Serialization?

(i) By using python's Json. Module
(ii) By using django.core.serializers Module

(17) How to use dumpdata option, to display database data to the Console?

We can dump our database data either to the Console or to the file by using dumpdata option.

Commands:

```
py manage.py dumpdata testapp.Employee
```

(18) What is the role of Router in Viewset?

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.

(19) By Using which Tools we can Test functionality of our API?

1. Postman
2. Swagger
3. DRF Browsable API
4. We can write our own client (like python script) to test functionality
5. From the Command prompt by using HTTPie & CURL.

01/3/22