API

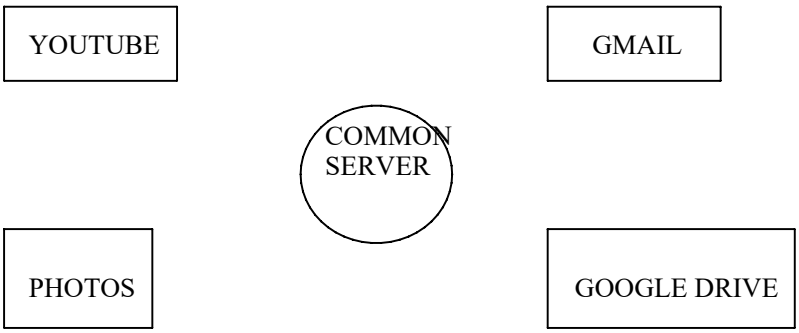
23 March 2023 10:49

WHAT is API:
---

1. API stands for Application Programming Interface
2. API is a server that you can use to retrieve and send data to using code.
3. APIs are most commonly used to retrieve data.
4. API is a software intermediary that allows two applications to talk to each other.

WHY/What are the Advanatages of Using API's:
---

Scenario 1:
---
Different Soft-wares with same server




Drawbacks:
---
1. If u r using Same server for different applications then if there is a problem in one app it will affect other apps as well

Scenario-2:
---
All Applications are having different servers:
---


Drawbacks:
---

1. In above Scenario we have store and validate the information of Ashu in each and every application seperately
2. So for example 1 lakh people are willing to make an account
3. Now for 1 lakh people we need to make 4 lakh accounts in all these 4 Applications
4. In this case The Data Redundancy will be there MORE

5. **Data redundancy** occurs when the same piece of **data** exists in multiple places

Web-Services:
---
1. A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.
   (Or)
   Web-service is Services Provided B/W one server to another server through internet or web or online

Note:
---
Inter-operable means able to exchange and make use of information

### APIs vs Web Services

1. Every web service is an API
   since it exposes an application's data and/or functionality
2. But not every API is a web service.
3. Web-service requires Internet but API's can work both offline or online

Different Types of Web-Service API's:
---
1. SOAP(Simple Object Access Protocol)
2. Graph-QL
3. REST(Representational State Transfer)
   The Web-service which is Written by using REST is known as Restful-API's

Differences B/W SOAP and REST

| **SOAP** | **REST** |
|---|---|
| It has strict rules and advanced security to follow. | There are loose guidelines to follow allowing developers to make recommendations easily |
| It is driven by Function | It is driven by Data |
| It requires more Bandwidth | It requires minimum Bandwidth |

Note:
---
1. When we interact with the server directly then it will give u the entire HTML page as Response
2. But now we are interacting with the API, it will give only Data.

API uses:
---
1. API is mainly Used for data Transactions(Sending, receiving, Modifying the data)
2. Whenever we are doing data transactions we cannot send the data in our own format

Standard formats of data is
1. XML
2. JSON

1. XML:
---
In earlier days we are doing Data transaction in the XML format Bcoz it was very easy to Seggregate and identify the Data sent by the sender

Normal format:
---
[1234,'harshad',25,'Bangalore',Karnataka']  # this format cannot be understood to the reciever

XML format:  it is an Organized form of Data

```
<xml>
    <emp1>
            <eid>1234</eid>
            <ename>harshad</ename>
            <eage>25</eage>
            <native>Bangalore</native>
    </emp1>
</xml>
```

Drawbacks of XML format:

1. Difficult to Write the commands in the form of the tags
2. So in-order to avoid that difficulties only JSON format has come into Picture

JSON: JAVA SCRIPT OBJECT NOTATION

1. Many of the people are agreed to store and send the data in form of JSON
2. In JSON we will be storing the data in the form of the key and value pairs
3. JSON is Similar to python Dictionary only

Example:

```
{
    Key1 : value1 ;
    Key2 : value2 ;
    Key3 : value3 ;
    Key4 : value4 ;
    |
    |
    |
    Keyn : valuen ;

}
```

The above data if we specify in Json format will look like Below

```
{
    'Emp1':
            {
                'eid' : 1234,
                'ename' : 'harshad',
                'age' : 25,
                'native' : 'Bangalore'


            }
}
```

## Views Of Django-Restframework

1. Whenever we get a request from user or from external environment we need to showcase him something or we need to Project him something that is known as views

   In DRF we will create Views in 2 ways
   1. APIView
   2. ViewSet

APIView:

---

1. APIView is responsible for Describing the logic to make api endpoint(apiview)
2. What u project to the user is known as endpoint
3. APIView contains the standard HTTP methods(get, post, put,patch and delete) as functions, and this r used for providing Various views for the requests
4. APIView will ive the most control over the logic to Developer
5. APIView is the Most used technique while implementing of Complex Logic
6. APIView is used when we r dealing with local files
7. APIView is most used when ever we process the files and render the Synchronous response

Note:

---

A synchronous response returns to the client in the same HTTP connection as the request.

Procedure to create views in DRF:

---

1. We will write the views inside the views.py file only
2. All the views that we write for the API's are Class Based Views only
3. Whenever we wanted to write a APIView class then our CBV must be inherited from APIView class of rest_framework
   from rest_framework.views import APIView

   Syntax for creating the View of APIView:

   ---

   ```
   Class SampleApiView(APIView):
       pass
   ```

4. In-order give the response from the CBV we have to use Response function so import it by below syntax

   From rest_framework.response import Response

5. Syntax of Response

   ---

   Response(data, status=None, template_name=None, headers=None, content_type=None)

   - 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.

6. Response Should be in the form of Dictionary

## Serialization

---

1. Serialization is a process of Converting the data of Any format into the Json Objects
2. Serialization can be Achieved by Serializers
3. Serializers are used for Receiving or accepting the data from the user in Form formats

Procedure to create the Serializers:

---

First create serializers.py file inside ur Application

Each and every user defined serializer classes must be inherited from ModelSerializer class of serializers module

From rest_framework import serializers

Creation of serializers is same as our Built in forms of django

Sample syntax of user defined Serializer

---

```
Class SerializerClassname(serializer.ModelSerializer):
        Class Meta:
           model=ModelClassName
           fields='__all__'
```

Procedure to Access The Submitted Value From the Serializer form In Backend:

---

1. In above we have already created  Serializer class now we r trying to Access the Submitted Data
2. In View methods(get, post, put, patch or delete) we need to create the instance of Serializer class with below Syntax

   Syntax:

   ---

   ```
   Serializer_class_instance==self.serializer_class(data=request.data)
   ```

3. After Collecting the data we need to validate the data by using is_valid() method
4. If the submitted data is valid then the data will be bound in validated_data dictionary
5. Syntax for Accessing the Submitted data after validation

   Syntax:

   ---

   ```
   Serializer_class_instance.validated_data.get('element_name')
   ```

Sample Function Based View Which is dealing with the ModelSerializer

---

Models.py content

---

```
from django.db import models
# Create your models here.
class Emp(models.Model):
    eid=models.IntegerField(primary_key=True)
    ename=models.CharField(max_length=100)
    sal=models.IntegerField()
    job=models.CharField(max_length=100)
    def __str__(self) -> str:
        return self.ename
```

Serializers.py

---

```
from app.models import *
from rest_framework import serializers
class EmpForm(serializers.ModelSerializer):
    class Meta:
        model=Emp
        fields='__all__'
```

Note:
---
As I am willing to display all data I have given fields value as '__all__' .We can change it Based on your requirement

Views.Py file content:
---

```
from django.shortcuts import render
# Create your views here.
from app.models import *
from rest_framework.response import Response
from rest_framework.decorators import api_view
from app.serializers import *
@api_view(['GET','POST'])
def listemps(request):
    queryset=Emp.objects.all()
    esc=EmpForm(queryset,many=True)
    data=esc.data
    return Response(data)
```

Note:
---
In above scenario data will be displayed in both scenarios of admin user login and logout If u want to display the details only in case of admin login we have to decorate FBV with permission_classes decorated

Example of views.py:
---

```
from django.shortcuts import render
```

```python
# Create your views here.
from app.models import *
from rest_framework.response import Response
from rest_framework.decorators import api_view,permission_classes
from app.serializers import *
from rest_framework.permissions import IsAuthenticated
@api_view(['GET','POST'])
@permission_classes([IsAuthenticated])
def listemps(request):
    queryset=Emp.objects.all()
    esc=EmpForm(queryset,many=True)
    data=esc.data
    return Response(data)
```

Git Link:

---

https://github.com/harshadvali1111/api_FBV

APIView Class HTTP methods:

---

1. When we r working with APIViews we need to Concentrate on HTTP request methods
2. HTTP request methods are used for Processing the requests

We have 5 Different HTTP methods:

---

1. Get
2. Post
3. Put
4. Patch
5. Delete

1. Get method:

   ---

   1. Get method is used when-ever we wanted to get or retrieve the data
   2. Syntax of get method is

```
def get(self,request,format=None):
    |
    |
    Return Response(Dict data)
```

2. post method:

   ---

a. post method is used when-ever we wanted to send the data
b. Post requests are handled by the post method in APIView
c. Syntax of post method is

```
Def post(self,request,format=None):
    |
    |
    Return Response(Dict data)
```

3. put method:

---

a. put method is used when-ever we wanted to update the data
b. Put requests are handled by the put method in APIView
c. Syntax of put method is

```
Def put(self,request,format=None):
|
|
Return Response(Dict data)
```

4. patch method:

---

a. patch method is used when-ever we wanted to partially update the data
b. Patch requests are handled by the patch method in APIView
c. Syntax of patch method is

```
Def patch(self,request,format=None):
|
|
Return Response(Dict data)
```

5. delete method:

---

a. delete method is used when-ever we wanted to delete the data
b. delete requests are handled by the delete method in APIView
c. Syntax of delete method is

```
Def delete(self,request,format=None):
|
|
Return Response(Dict data)
```

Sample example of APIVIew class:

---

Models.py file Content:
---

```
from django.db import models
# Create your models here.

class Product_Category(models.Model):
    category_name=models.CharField(max_length=100)
    category_id=models.PositiveIntegerField()
    def __str__(self):
        return self.category_name
class Product(models.Model):
    category_name=models.ForeignKey(Product_Category, on_delete=models.CASCADE)
    Pname=models.CharField(max_length=100)
    Pid=models.PositiveIntegerField()
    price=models.DecimalField(max_digits=8,decimal_places=2)
    date=models.DateField()
    def __str__(self):
        return self.Pname
```

Serializers.py file Content:
---

```
from rest_framework import serializers

from app.models import *
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model=Product
        fields='__all__'
```

Views.py File Content
---

```
from django.shortcuts import render
# Create your views here.
from rest_framework.views import APIView
from app.models import *
from app.serializers import *
```

```
from rest_framework.response import Response
class ProductCrud(APIView):
    def get(self,request,id):
        PQS=Product.objects.all()
        PJD=ProductSerializer(PQS,many=True)
        return Response(PJD.data)

    def post(self,request,id):
        PMSD=ProductSerializer(data=request.data)
        if PMSD.is_valid():
            PMSD.save()
            return Response({'message':'Product is created'})
        return Response({'Failed':'Product is not created'})
    def put(self,request,id):
        id=request.data['id']
        PO=Product.objects.get(id=id)
        UPO=ProductSerializer(PO,data=request.data)
        if UPO.is_valid():
            UPO.save()
            return Response({'message':'Product is Updated'})
        return Response({'Failed':'Product is not Updated'})

    def patch(self,request,id):
        id=request.data['id']
        PO=Product.objects.get(id=id)
        UPO=ProductSerializer(PO,data=request.data,partial=True)
        if UPO.is_valid():
            UPO.save()
            return Response({'message':'Product is Updated'})
        return Response({'Failed':'Product is not Updated'})


    def delete(self,request,id):
        Product.objects.get(id=id).delete()
        return Response({'Success':'Product is Deleted'})
```

Urls.py file Content:

---

```
from django.contrib import admin
from django.urls import path
from app.views import *
urlpatterns = [
    path('admin/', admin.site.urls),
```

```
    path('ProductCrud/<id>',ProductCrud.as_view(),name='ProductCrud'),

]
```

Viewsets
---

4. Viewsets is a type of view in django framework which will take care of most Typical logic part
5. Viewsets are used whenever we r dealing with the DB related operations and as well as Data Structures
6. It is the fastest way to create the Interface which responsible for interacting with DB
7. With Little amount of Logic(code) we can perform the CRUD operations of DataBase

Viewsets Supports the view methods such as

8. List
9. Create
10. Update
11. Retrieve
12. Partial_update
13. Destroy

Note:
---
After creating the viewsets class we will be performing the mapping
Of Http Methods To Actions by using the routing

Routing
---

1. Routing is process of receiving the control that comes from the browser when the user request
   Through url and connects to respective Viewset

Or

Routing routes an incoming HTTP request to a particular action method on a Web API controller.

1. To do routing we will use routers of rest_framework

```
from rest_framework import routers
```

2. After importing routers create a DefaultRouter class Instance of routers

router=routers.DefaultRouter()

3. After creating router instance,we allow router to do further Navigation by registering the Viewsets

router.register('suffix/',viewclassname,basename='name_of_mapping')

1. After registering include them in urlpatterns by using include function

Urlpatterns[
Path('suffix/',include(router.urls)),
]

Classification of Viewsets:
---
We are Having Two Types Of Viewsets
1. ModelViewSet
2. ViewSet

ModelViewSet Example:
---

Views.Py file Content:
---

```
from django.shortcuts import render
# Create your views here.
from rest_framework import viewsets
from app.models import *
from app.serializers import *
class ProductCrudActions(viewsets.ModelViewSet):
    queryset=Product.objects.all()
    serializer_class=ProductSerializer
```

Urls.py file Content:
---

```
from django.contrib import admin
from django.urls import path,include
from app.views import *
from rest_framework.routers import DefaultRouter
DRO=DefaultRouter()
DRO.register('products',ProductCrudActions,basename='products')
urlpatterns = [
    path('admin/', admin.site.urls),
    path('app/',include(DRO.urls)),
]
```

Working with Viewsets:

---

14. Inside views.py file first import viewsets with below syntax
    From rest_framework import viewsets

15. After importing, create a class in views.py file which must be inherited from viewsets.Viewset
16. And after creating the class create list method
17. List method is responsible for listing out the features and Functionalities of api

Views.py file content:
---

```
from django.shortcuts import render
# Create your views here.
from rest_framework.views import APIView
from rest_framework.viewsets import ViewSet
from app.models import *
from app.serializers import *
from rest_framework.response import Response

class ProductCrud(ViewSet):
    def list(self,request):
        PQS=Product.objects.all()
        PJD=ProductSerializer(PQS,many=True)
        return Response(PJD.data)
    def retrieve(self,request,pk):
        PQS=Product.objects.get(pk=pk)
        PJD=ProductSerializer(PQS)
        return Response(PJD.data)

    def create(self,request):
        PMSD=ProductSerializer(data=request.data)
        if PMSD.is_valid():
            PMSD.save()
            return Response({'message':'Product is created'})
        return Response({'Failed':'Product is not created'})
    def update(self,request,pk):
        PO=Product.objects.get(pk=pk)
        UPO=ProductSerializer(PO,data=request.data)
        if UPO.is_valid():
            UPO.save()
            return Response({'message':'Product is Updated'})
        return Response({'Failed':'Product is not Updated'})
    def partial_update(self,request,pk):
        PO=Product.objects.get(pk=pk)
        UPO=ProductSerializer(PO,data=request.data,partial=True)
        if UPO.is_valid():
            UPO.save()
            return Response({'message':'Product is partially Updated'})
        return Response({'Failed':'Product is not Updated'})
    def destroy(self,request,pk):
        PO=Product.objects.get(pk=pk)
        PO.delete()
        return Response({'message':'Product Deleted Successfully'})
```