Django REST Framework

Serializers-:

In Django REST Framework, serializers are responsible for converting complex data such as querysets and model instances to native Python datatypes (called serialization) that can then be easily rendered into JSON, XML or other content types which are understandable by Front End.

Complex data type — — serialization — → **python native data type** — -json →**json data**

Serializers are also responsible for deserialization which means it allows parsed data to be converted back into complex types, after first validating the incoming data.

- Serialization
- Deserialization

How to Create Serializer Class-:

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
```

ModelSerializer Class-:

The ModelSerializer class provides a shortcut that lets you automatically create a Serializer class with fields that correspond to the Model fields.

The ModelSerializer class is the same as a regular Serializer class, except that:

It will automatically generate a set of fields for you, based on the model.

It will automatically generate validators for the serializer, such as unique_together validators.

It includes simple default implementations of .create() and .update().

How to Create Model Serializer Class-:

```
from rest_framework import serializers
class StudentSerializer(serializers.ModelSerializer):
    name = serializers.CharField(read_only=True)
    class Meta:
        model = Student
        fields = ['id', 'name', 'roll', 'city']
```

HyperlinkedModelSerializer-:

The HyperlinkedModelSerializer class is similar to the ModelSerializer class except that it uses hyperlinks to represent relationships, rather than primary keys. By

default the serializer will include a url field instead of a primary key field. The url field will be represented using a HyperlinkedIdentityField serializer field, and any relationships on the model will be represented using a HyperlinkedRelatedField serializer field.

How to Create HyperlinkedModelSerializer Classs-:

```
class SerializerName(serializers.HyperlinkedModelSerializer):
        class Meta:
                model = ModelName
                fields = List of Fields
```

De-serialization-:

Serializers are also responsible for deserialization which means it allows parsed data to be converted back into complex types, after first validating the incoming data.

JSON Data — **Parse data** →Python native data — **De-serialization** →Complex Data

How to Create De-serialization Class-:

```
@csrf_exempt
def student_create(request):
   if request.method=="POST":
      json_data=request.body
      stream= io.BytesIO(json_data)
      pythondata= JSONParser().parse(stream)
```

 # in below line convert python data to complex data type :- de-serialization

```
serializer=StudentSerializer(data=pythondata)
if serializer.is_valid():
   serializer.save()
   res= {'msg':'Data Created'}
   json_data = JSONRenderer().render(res)
   return HttpResponse(
     json_data, content_type='application/json'
     )
```

Serializer Fields-:

BooleanField-:A boolean field used to wrap True or False values.

CharField-: CharField is used to store text representation.

EmailField-: EmailField is also a text representation and it validates the text to be a valid e-mail address.

IPAddressField-: IPAddressField is a field that ensures the input is a valid IPv4 or IPv6 string.

IntegerField-:IntegerField is basically a integer field that validates the input against Python’s int instance.

FloatField-:FloatField is basically a float field that validates the input against Python’s float instance.

DecimalField-:DecimalField is basically a decimal field that validates the input against Python's decimal instance.

DateTimeField-:DateTimeField is a serializer field used for date and time representation.

DateField-:DateField is a serializer field used for date representation.

Core arguments in serializer fields-:

read_only-: Set this to True to ensure that the field is used when serializing a representation, but is not used when creating or updating an instance during deserialization

write_only-: Set this to True to ensure that the field may be used when updating or creating an instance, but is not included when serializing the representation.

Required-: Setting this to False also allows the object attribute or dictionary key to be omitted from output when serializing the instance.

Validators-: A list of validator functions which should be applied to the incoming field input, and which either raise a validation error or simply return.

error_messages-: A dictionary of error codes to error messages.

Note-:

Render the Data into Json-:

json_data = JSONRenderer().render(serializer.data)

Validation-:

Field Level Validation-:

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
        name = serializers.CharField(max_length=100)
        roll = serializers.IntegerField()
        city = serializers.CharField(max_length=100)
        def  validate_roll(self, value):
                if value > = 200 :
                        raise serializers.ValidationError('Seat Full')
                return value
```

Object Level Validation-:

```
from rest_framework import serializers
class StudentSerializer(serializers.Serializer):
        name = serializers.CharField(max_length=100)
        roll = serializers.IntegerField()
        city = serializers.CharField(max_length=100)
        def  validate(self, data):
                nm = data.get('name')
                ct = data.get('city')
```

```
        if nm.lower() == 'rohit' and ct.lower() != 'ranchi' :
            raise serializers.ValidationError('City must be Ranchi')
        return data
```

Validators-:

```
from rest_framework import serializers
def  starts_with_r(value):
    if value['0'].lower() != 'r' :
        raise serializers.ValidationError('Name should start with R')
class StudentSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100, validators=[starts_with_r])
    roll = serializers.IntegerField()
    city = serializers.CharField(max_length=100)
```

Function Based api_view-:

Django views facilitate processing the HTTP requests and providing HTTP responses. On receiving an HTTP request, Django creates an HttpRequest instance, and it is passed as the first argument to the view function. This instance contains HTTP verbs such as GET, POST, PUT, PATCH, or DELETE. The view function checks the value and executes the code based on the HTTP verb. Here the code uses @csrf_exempt decorator to set a CSRF (Cross-Site Request Forgery) cookie. This makes it possible to POST to this view from clients that won't have a CSRF token. Let's get into the implementation process. You can add the below code in the views.py file.

Code-:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
@api_view(['POST'])
def student_create(request):
  if request.method == 'POST':
    serializer = StudentSerializer(data = request.data)
    if serializer.is_valid():
      serializer.save()
      res = {'msg': 'Data Created'}
      return Response(res, status=status.HTTP_201_CREATED)
    return Response(serializer.error, status=status.HTTP_400_BAD_REQUEST)
```

APIView Class-:

APIView class provides commonly required behavior for standard list and detail views. With APIView class, we can rewrite the root view as a class-based view. They provide action methods such as get(), post(), put(), patch(), and delete() rather than defining the handler methods.

How to API View Class-:

```
from rest_framework.views import APIView
class StudentAPI(APIView):
  def get(self, request, format=None):
      stu = Student.objects.all()
      serializer = StudentSerializer(stu, many=True)
      return Response(serializer.data)
   def post(self, request, format=None):
      serializer = StudentSerializer(data=request.data)
      if serializer.is_valid():
         serializer.save()
         return Response({'msg': 'Data Created' },
status=status.HTTP_201_CREATED)
      return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Generic class-based views-:

To make use of generic class-based views, the view classes should import from *rest_framework.generics.*

- **CreateAPIView**: It provides a post method handler and it is used for create-only endpoints. CreateAPIView extends GenericAPIView and CreateModelMixin
- **ListAPIView**: It provides a get method handler and is used for read-only endpoints to represent a collection of model instances. ListAPIView extends GenericAPIView and ListModelMixin.

- **RetrieveAPIView**: It provides a get method handler and is used for read-only endpoints to represent a single model instance. RetrieveAPIView extends GenericAPIView and RetrieveModelMixin.
- **DestroyAPIView**: It provides a delete method handler and is used for delete-only endpoints for a single model instance. DestroyAPIView extends GenericAPIView and DestroyModelMixin.
- **UpdateAPIView**: It provides put and patch method handlers and is used for update-only endpoints for a single model instance. UpdateAPIView extends GenericAPIView and UpdateModelMixin.
- **ListCreateAPIView**: It provides get and post method handlers and is used for read-write endpoints to represent a collection of model instances. ListCreateAPIView extends GenericAPIView, ListModelMixin, and CreateModelMixin..
- **RetrieveUpdateAPIView**: It provides get, put, and patch method handlers. It is used to read or update endpoints to represent a single model instance. RetrieveUpdateAPIView extends GenericAPIView, RetrieveModelMixin, and UpdateModelMixin.
- **RetrieveDestroyAPIView**: It provides get and delete method handlers and it is used for read or delete endpoints to represent a single model instance. RetrieveDestroyAPIView extends GenericAPIView, RetrieveModelMixin, and DestroyModelMixin.
- **RetrieveUpdateDestroyAPIView**: It provides get, put, patch, and delete method handlers. It is used for read-write-delete endpoints to represent a single model instance. It extends GenericAPIView, RetrieveModelMixin, UpdateModelMixin, and DestroyModelMixin.

Creating views using generic class-based views-:

```
from rest_framework import generics
from transformers.models import Transformer
from transformers.serializers import TransformerSerializer
class TransformerList(generics.ListCreateAPIView):
    queryset = Transformer.objects.all()
    serializer_class = TransformerSerializer
class TransformerDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Transformer.objects.all()
    serializer_class = TransformerSerializer
```

Mixins-:

Mixin classes allow us to compose reusable bits of behavior. They can be imported from rest_framework.mixins. Let's discuss the different types of mixin classes

- ListModelMixin : It provides a *.list(request, *args, **kwargs)* method for listing a queryset. If the queryset is populated, the response body has a *200 OK* response with a serialized representation of the queryset.
- CreateModelMixin: It provides a *.create(request, *args, **kwargs)* method for creating and saving a new model instance. If the object is created, the response body has a *201 Created* response, with a serialized representation of the object. If invalid, it returns a *400 Bad Request* response with the error details.

- RetrieveModelMixin: It provides a *.retrieve(request, *args, **kwargs)* method for returning an existing model instance in a response. If an object can be retrieved, the response body has a *200 OK* response, with a serialized representation of the object. Otherwise, it will return a *404 Not Found*.
- UpdateModelMixin: It provides a *.update(request, *args, **kwargs)* method for updating and saving an existing model instance. It also provides a *.partial_update(request, *args, **kwargs)* method for partially updating and an existing model instance. . If the object is updated, the response body has a *200 OK* response, with a serialized representation of the object. Otherwise, 400 Bad Request response will be returned with the error details.
- DestroyModelMixin: It provides a *.destroy(request, *args, **kwargs)* method for deleting an existing model instance. If an object is deleted, the response body has a *204 No Content* response, otherwise, it will return a *404 Not Found*.

Code-:

```
from rest_framework import mixins
from rest_framework import generics
from transformers.models import Transformer
from transformers.serializers import TransformerSerializer

class TransformerList(mixins.ListModelMixin,
                      mixins.CreateModelMixin,
                      generics.GenericAPIView):
    queryset = Transformer.objects.all()
```

```
    serializer_class = TransformerSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

class TransformerDetail(mixins.RetrieveModelMixin,
                        mixins.UpdateModelMixin,
                        mixins.DestroyModelMixin,
                        generics.GenericAPIView):
    queryset = Transformer.objects.all()
    serializer_class = TransformerSerializer

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def patch(self, request, *args, **kwargs):
        return self.partial_update(request, *args, **kwargs)
```

```
    def delete(self, request, *args, **kwargs):
    return self.destroy(request, *args, **kwargs)
```

ViewSet and ModelViewSet-:

Django Rest Framework (DRF) allows you to combine the logic for a set of related views in a single class, called a ViewSet.

Routers are used to wiring up the URL configurations so, we do not need to write URL configurations externally.

Code-:

```
from rest_framework import viewsets
class StudentViewSet(viewsets.ViewSet):
    def list(self, request): ……
    def create(self, request):  …….
    def retrieve(self, request, pk=None): ……
    def update(self, request, pk=None): …….
    def partial_update(self, request, pk=None): ……
    def destroy(self, request, pk=None): ……..
```

Code-:

```
class StudentViewSet(viewsets.ViewSet):
    def list(self,request):
        stu=Student.objects.all()
        serializer=StudentSerializer(stu,many=True)
        return Response(serializer.data)

    def retreive(self, request, pk=None):
        id=pk
        if id is not None:
            stu=Student.objects.get(id=id)
            serializer=StudentSerializer(stu)
            return  Response(serializer.data)

    def create(self,request):
        serializer=StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response({'msg':'Data  created'},
status=status.HTTP_201_CREATED)
        return Response(serializer.errors , status=status.HTTP_400_BAD_REQUEST)

    def destroy(self, request,pk):
        id=pk
        stu=Student.objects.get(pk=id)
```

```
    stu.delete()
    return Response({'msg':'Data Deleted'})
```

ModelViewSet Class-:

The ModelViewSet class inherits from GenericAPIView and includes implementations for various actions, by mixing in the behavior of the various mixin classes.

The actions provided by the ModelViewSet class are list(), retrieve(), create(), update(), partial_update(), and destroy(). You can use any of the standard attributes or method overrides provided by GenericAPIView

Code-:

```
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

ReadOnlyModelViewSet Class-:

The ReadOnlyModelViewSet class also inherits from GenericAPIView. As with ModelViewSet it also includes implementations for various actions, but unlike ModelViewSet only provides the 'read-only' actions, list() and retrieve().

You can use any of the standard attributes and method overrides available to GenericAPIView

Code-:

```
class StudentReadOnlyModelViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

Authentication-:

Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with. The permission and throttling policies can then use those credentials to determine if the request should be permitted.

Authentication is always run at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.

- BasicAuthentication
- SessionAuthentication
- TokenAuthentication
- RemoteUserAuthentication
- Custom authentication

Permission Classes-:

- AllowAny
- IsAuthenticated
- IsAdminUser
- IsAuthenticatedOrReadOnly
- DjangoModelPermissions
- DjangoModelPermissionsOrAnonReadOnly
- DjangoObjectPermissions
- Custom Permissions

BasicAuthentication-:

Setting the authentication scheme globally-:

```
REST_FRAMEWORK = {
        'DEFAULT_AUTHENTICATION_CLASSES': [
                'rest_framework.authentication.BasicAuthentication',
        ]
}
```

Function-based views -:

```
@api_view(['GET'])
@authentication_classes([BasicAuthentication])
@permission_classes([IsAuthenticated])
```

Class-based views-:

```
class ClassBasedView(APIView):
    authentication_classes = [BasicAuthentication]
    permission_classes = [IsAuthenticated]
```

SessionAuthentication-:

SessionAuthentication uses Django's default session backend for authentication. Session authentication is appropriate for AJAX clients that are running in the same session context as your website. If successfully authenticated, SessionAuthentication provides the following credentials.

Code-:

```
class StudentModelViewSet(viewsets.ModelViewSet):
    queryset= Student.objects.all()
    serializer_class = StudentSerializer
    authentication_classes=[SessionAuthentication]
    permission_classes=[IsAuthenticated]
```

How Session Authentication is different from Basic Authentication?

In the **basic authentication** we need to send the username and password for every request.
In the **session authentication** we will send username and password at initial request. Then from the server response we get the session id which stores in the browser and gonna use that for requests.

Token Authentication-:

Token authentication refers to exchanging username and password for a token that will be used in all subsequent requests to identify the user on the server side.This article revolves about implementing token authentication using Django REST Framework to make an API. The token authentication works by providing token in exchange for exchanging usernames and passwords.

rest_framework settings-:

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

}
```

Import router and rest_framework.authtoken for token authentication

path('api/', include(router.urls)),
path('api-token-auth/', views.obtain_auth_token,) name='api-token-auth'),

http POST http://localhost:8081/api-token-auth/ username='your_username' password="your_password"

http http://localhost:8081/api/user/ "Authorization: Token API_KEY_HERE"

JWT Authentication-:

JSON Web Token is an open standard for securely transferring data within parties using a JSON object. JWT is used for stateless authentication mechanisms for users and providers, this means maintaining session is on the client-side instead of storing sessions on the server.

```
REST_FRAMEWORK = {
        'DEFAULT_AUTHENTICATION_CLASSES': [
                'rest_framework_simplejwt.authentication.JWTAuthentication',
        ],
}
```

```
from django.urls import path, include
from rest_framework_simplejwt import views as jwt_views


urlpatterns = [
    path('api/token/',
        jwt_views.TokenObtainPairView.as_view(),
        name ='token_obtain_pair'),
    path('api/token/refresh/',
        jwt_views.TokenRefreshView.as_view(),
        name ='token_refresh'),
    path('', include('app.urls')),
]
```

```
$ http post http://127.0.0.1:4000/api/token/ username=spider password=Vinayak
$  http http://127.0.0.1:4000/hello/ "Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjo
xNTg3Mjc5NDIxLCJqdGkiOiIzYWMwNDgzOTY3NjE0ZDgxYmFjMjBiMTBjMDlkMmY
wOCIsInVzZXJfaWQiOjF9.qtNrUpyPQI8W2K2T22NhcgVZGFTyLN1UL7uqJ0KnF0Y"
```

Throttling-:

Throttling is similar to permissions, in that it determines if a request should be authorized. Throttles indicate a temporary state, and are used to control the rate of requests that clients can make to an API.

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',
        'user': '1000/day'
    }
}
```

Filter data in Django Rest Framework-:

Django REST Framework's generic list view, by default, returns the entire query sets for a model manager. For real-world applications, it is necessary to filter the queryset to retrieve the relevant results based on the need. So, let's discuss how to create a RESTful Web Service that provides filtering capabilities.

- DjangoFilterBackend
- SearchFilter
- OrderingFilter

DjangoFilterBackend-:

The DjangoFilterBackend class is used to filter the queryset based on a specified set of fields. This backend class automatically creates a FilterSet (django_filters.rest_framework.FilterSet) class for the given fields. We can also create our own FilterSet class with customized settings.

```
REST_FRAMEWORK = {
        'DEFAULT_FILTER_BACKENDS'
                'django_filters.rest_framework.DjangoFilterBackend',
        ),
}
```

SearchFilter-:

The SearchFilter class supports a single query parameter-based searching feature, and it is based on the Django admin's search function.

By default, SearchFilter class uses case-insensitive partial matches, and it may contain multiple search terms (should be whitespace and/or comma-separated). We can also restrict the search behavior by prepending various characters to the search_fields.

'^' Starts-with search.

'=' Exact matches.

'@' Full-text search. (for Django's PostgreSQL backend)

'$' Regex search

```
class RobotList(generics.ListCreateAPIView):
        queryset = Robot.objects.all()
        serializer_class = RobotSerializer
        name = 'robot-list'
        search_fields = (
                '^name',
        )
```

OrderingFilter-:

The OrderingFilter class allows you to order the result based on the specified fields. By default, the query parameter is named ordering, and it can be

overridden with the ORDERING_PARAM setting. The ordering_field attribute specifies a tuple of strings, which indicates the field names to sort the results.

Code-:

```
class RobotList(generics.ListCreateAPIView):
        queryset = Robot.objects.all()
        serializer_class = RobotSerializer
        name = 'robot-list'
        ordering_fields = (
                'price',
        )
```

Pagination in APIs -:

Imagine you have huge amount of details in your database. Do you think that it is wise to retrieve all at once while making an HTTP GET request? Here comes the importance of the Django REST framework pagination feature. It facilitates splitting the large result set into individual pages of data for each HTTP request.

```
REST_FRAMEWORK = {
        'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
        'PAGE_SIZE': 2,
}
```

LimitOffsetPagination-:

In LimitOffsetPagination style, client includes both a “limit” and an “offset” query parameter. The limit indicates the maximum number of items to return, same as that of the page_size. The offset indicates the starting position of the query w.r.t unpaginated items. To enable the LimitOffsetPagination style globally, you can set *rest_framework.pagination.LimitOffsetPagination* class to *DEFAULT_PAGINATION_CLASS*. The configuration as follows:

```
REST_FRAMEWORK = {
        'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.LimitOffsetPagination',
        'PAGE_SIZE': 2,
}
```

CursorPagination-:

The CursorPagination provides a cursor indicator to page through the result set. It provides only forward or reverse controls and doesn’t permit the client to navigate to arbitrary positions. The CursorPagination style assumes that there must be a created timestamp field on the model instance and it orders the results by ‘-created’. To enable the CursorPagination style you can mention *rest_framework.pagination.CursorPagination* class in *DEFAULT_PAGINATION_CLASS*

```
REST_FRAMEWORK = {
        'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.CursorPagination',
        'PAGE_SIZE': 2,
}
```