**IMDB Clone API**

<https://www.django-rest-framework.org/>

Model Serializer is most important.

1. Create folder
2. Create virtual env inside the project
   1. Python -m venv env
3. Go inside the environment
   1. Env\Scripts\activate
4. Pip install Django
5. Pip install djangorestframework
6. Django-admin startproject watchmate
7. Cd watchmate
8. Django-admin startapp watchlist\_app
9. Go to the settings.py and register app and rest\_framework

INSTALLED\_APPS = [

    'watchlist\_app',

    'rest\_framework',

]

1. Create API folder inside the watchlist\_app.
2. Create 3 files.
   1. Urls.py
   2. Views.py
   3. Serializers.py

(File name should be same as here. We create new api folder bcz we want all files togher.)

1. Create a model – models.py

from django.db import models

class WatchList(models.Model):

    title = models.CharField(max\_length=50)

    storyline = models.CharField(max\_length=200)

    active = models.BooleanField(default=False)

    created\_date = models.DateTimeField(auto\_now\_add=True)

    def \_\_str\_\_(self):

        return self.title

1. Register the model in admin.py

from watchlist\_app.models import WatchList

admin.site.register(WatchList)

1. Make the migrations for WatchList app

Python manage.py makemigrations

Python manage.py migrate

1. Create the super user

Vishal

vishal

**Serializers**

Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

There are many types of Serializers we are first using Serializer then we use ModelSerializer.

1. Make the serializer for WatchList model

from rest\_framework import serializers

class WatchListSerializer(serializers.Serializer):

    id = serializers.IntegerField(read\_only=True)

    title = serializers.CharField()

    storyline = serializers.CharField()

    active = serializers.BooleanField(default=False)

    created\_date = serializers.DateTimeField()

Here I did not use Meta class bcz I am using Function Based View.

1. create the view for the WatchList model

from watchlist\_app.models import WatchList

from watchlist\_app.api.serializers import WatchListSerializer

from rest\_framework.response import Response

from rest\_framework.decorators import api\_view

@api\_view()

def WatchListView(request):

    wathlist = WatchList.objects.all()

    serializer = WatchListSerializer(wathlist, many=True)

    return Response(serializer.data)

Here why we use many=True

If we have multiple entries or we can say multiple movies so we have to define

* Here We are using @api\_view() basically this is a decorator and default it contain get types of method.
* Suppose if we want to add new movie then I have to use post method so we can defined inside the @api\_view()

@api\_view([‘GET’,’POST’])

1. create the main url and app url

main.url

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('watch/', include('watchlist\_app.api.urls')),

]

Watchlist\_app.url

from django.urls import path

from watchlist\_app.api.views import WatchListView

urlpatterns = [

    path('', WatchListView, name='watchlist'),

]

1. Enter the data in watchlist Model through admin panel.
2. Create the Detail View

@api\_view()

def WatchListDetail(request, pk):

    watchlist = WatchList.objects.get(pk=pk)

    serializer = WatchListSerializer(watchlist)

    return Response(serializer.data)

1. Create the url.
2. Import the WatchListDetail View.

    path('<int:pk>/', WatchListDetail, name='watchlistDetail'),

Now I want to add the new movie through Browsable API only (thru Browser Only).

First we have to tell decorator that we want to post the request from Browser

Create the post method in WatchListView app

Create the create method in serializer.

\*\*validated\_data -> bcz it contains all the fields name.

Views.py

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

def WatchListView(request):

    if request.method == 'GET':

        wathlist = WatchList.objects.all()

        serializer = WatchListSerializer(wathlist, many=True)

        return Response(serializer.data)

    if request.method == 'POST':

        serializer = WatchListSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data)

        else:

            return Response(serializer.errors)

Serializers.py

class WatchListSerializer(serializers.Serializer):

    id = serializers.IntegerField(read\_only=True)

    title = serializers.CharField()

    storyline = serializers.CharField()

    active = serializers.BooleanField(default=False)

    created\_date = serializers.DateTimeField()

    def create(self, validated\_data):

        return WatchList.objects.create(\*\*validated\_data)

**PUT and DELETE methods**

* I want to delete and update (put) the existing record.
* I have to create put and delete method in views (detail view) and update method in serializer.
* I don’t need to write delete method in serializer.

Look in this snap we use Get, Put and Delete inside the decorator. It means it is telling that we can perform this operations on this view.

**WatchListDetail.py**

@api\_view(['GET','PUT','DELETE'])

def WatchListDetail(request, pk):

    if request.method == 'GET':

        watchlist = WatchList.objects.get(pk=pk)

        serializer = WatchListSerializer(watchlist)

        return Response(serializer.data)

    if request.method == 'PUT':

        movie = WatchList.objects.get(pk=pk)

        serializer = WatchListSerializer(movie, data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data)

        else:

            return Response(serializer.errors)

    if request.method == 'DELETE':

        movie = WatchList.objects.get(pk=pk)

        movie.delete()

        data = {

            'detail':"Deleted successfully",

        }

        return Response(data)

If we get any error in the deletion of any record so we use learn about the Status code.

If we want to update anything then we have tell serializer that we are performing this operations.

**Serializer.py**

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

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

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

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

  instance.created\_date = validate\_data.get('created\_date',instance.created\_date)

   instance.save()

   return instance

**Status Code**

Status code basically tell you that what is status of you command.

Suppose You <http://127.0.0.1:8000/watch/4/> to find id number 4’s Movie

But it is not present in your database so it through error. So we can use status code here. Using try or catch or and also in normal way.

* [**Status Codes**](https://www.django-rest-framework.org/api-guide/status-codes/#status-codes)
* [Informational - 1xx](https://www.django-rest-framework.org/api-guide/status-codes/#informational-1xx)
* [Successful - 2xx](https://www.django-rest-framework.org/api-guide/status-codes/#successful-2xx)
* [Redirection - 3xx](https://www.django-rest-framework.org/api-guide/status-codes/#redirection-3xx)
* [Client Error - 4xx](https://www.django-rest-framework.org/api-guide/status-codes/#client-error-4xx)
* [Server Error - 5xx](https://www.django-rest-framework.org/api-guide/status-codes/#server-error-5xx)
* [Helper functions](https://www.django-rest-framework.org/api-guide/status-codes/#helper-functions)

Example :

HTTP\_200\_OK

HTTP\_201\_CREATED

HTTP\_202\_ACCEPTED

HTTP\_203\_NON\_AUTHORITATIVE\_INFORMATION

HTTP\_204\_NO\_CONTENT

HTTP\_404\_NOT\_FOUND

Import status -

from rest\_framework import status

When we delete the movie so it should so 204 status code.

Check in the return statement. In detail View

if request.method == 'DELETE':

        movie = WatchList.objects.get(pk=pk)

        movie.delete()

        data = {

            'detail':"Deleted successfully",

        }

        return Response(data, status=status.HTTP\_204\_NO\_CONTENT)

Or if want to get that movie which is not present in our data so it give us error.

So we can use. Try and except

if request.method == 'GET':

        try:

            watchlist = WatchList.objects.get(pk=pk)

        except:

            return Response(status=status.HTTP\_404\_NOT\_FOUND)

        serializer = WatchListSerializer(watchlist)

        return Response(serializer.data)

**Validators**

**It introduces a proper separation of concerns, making your code behavior more obvious**.

If I want to check condition like movie name should be greater than 3 then use it.

There are 3 types of validators.

1. Field level
2. mulpleFieldLevle
3. Field level but at the time of field declarations.

And define it’s function outside the class

Serializers.py

# Field level Validation

    def validate\_title(self, value):

        if len(value) < 3:

            raise serializers.ValidationError("Title length is too short.")

        return value

    # Validation on multiple fields

    def validate(self, data):

        if data['title'].lower() == data['storyline'].lower():

            raise serializers.ValidationError("Title and its Description should be Different.")

        return data

Here validated\_title so tile is a field name

Suppose I want to validate active field then use validated\_active()

**Class Based View**

Comment the Function Based View and import APIView model

Don’t comment the create and update method in serializer.

from rest\_framework.views import APIView

class WatchListView(APIView):

    def get(self, request):

        wathlist = WatchList.objects.all()

        serializer = WatchListSerializer(wathlist, many=True)

        return Response(serializer.data)

    def post(self, request):

        serializer = WatchListSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data)

        else:

            return Response(serializer.errors)

class WatchListDetail(APIView):

    def get(self, request, pk):

        try:

            watchlist = WatchList.objects.get(pk=pk)

        except:

            return Response(status=status.HTTP\_404\_NOT\_FOUND)

        serializer = WatchListSerializer(watchlist)

        return Response(serializer.data)

    def put(self, request, pk):

        movie = WatchList.objects.get(pk=pk)

        serializer = WatchListSerializer(movie, data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data)

        else:

            return Response(serializer.errors)

    def delete(self, request, pk):

        try:

            movie = WatchList.objects.get(pk=pk)

        except:

            return Response(status=status.HTTP\_404\_NOT\_FOUND)

        movie.delete()

        data = {

            'detail':"Deleted successfully",

        }

        return Response(data, status=status.HTTP\_204\_NO\_CONTENT)

**Urls.py** ( We have to use as\_view() for class based view)

urlpatterns = [

    # Class based View Urls

    path('', WatchListView.as\_view(), name="watchlist"),

    path('<int:pk>/', WatchListDetail.as\_view(), name="watchListDetail"),

]

**Model Serializer**

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().

Declaring a ModelSerializer looks like this:

class AccountSerializer(serializers.ModelSerializer):

class Meta:

model = Account

fields = ['id', 'account\_name', 'users', 'created']

Serializer.py

In serializer.py comment all the classes and use only ModelSerializer

class WatchListSerializer(serializers.ModelSerializer):

    class Meta:

        model = WatchList

        fields = "\_\_all\_\_"

“\_\_all\_\_” can take all the fields.

**That’s all now you don’t need to define any field, any create update method and all.**

If I have 100 fields but I don’t want to show 3 fields the I can exclude them

class WatchListSerializer(serializers.ModelSerializer):

    class Meta:

        model = WatchList

        # fields = "\_\_all\_\_"

        # or

        # fields = ['title','storyline','active','created\_date']

        # or if I want to exclude some fields then

        exclude = ('created\_date',)

Creating New Model - StreamPlateform ( Which movie hosted on which plateform)

**Models.py**

class StreamPlateform(models.Model):

    name = models.CharField(max\_length=50)

    about = models.CharField(max\_length=200)

    website = models.URLField(max\_length=100)

    def \_\_str\_\_(self):

        return self.name

**serializers.py**

Import the model first

from watchlist\_app.models import WatchList, StreamPlateform

class StreamPlateformSerializer(serializers.ModelSerializer):

    class Meta:

        model = StreamPlateform

        fields = "\_\_all\_\_"

create the views List and Detail View Same as Watchlist

**Views.py**

class StreamPlateformList(APIView):

    def get(self, request):

        plateforms = StreamPlateform.objects.all()

        serializer = StreamPlateformSerializer(plateforms, many=True)

        return Response(serializer.data)

    def post(self, request):

        serializer = StreamPlateformSerializer(data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data)

        else:

            return Response(serializer.errors)

class StreamPlateformDetail(APIView):

    def get(self, request, pk):

        try:

            plateform = StreamPlateform.objects.get(pk=pk)

        except:

            return Response(status=status.HTTP\_404\_NOT\_FOUND)

        serializer = StreamPlateformSerializer(plateform)

        return Response(serializer.data)

    def put(self, request, pk):

        plateform = StreamPlateform.objects.get(pk=pk)

        serializer = StreamPlateformSerializer(plateform, data=request.data)

        if serializer.is\_valid():

            serializer.save()

            return Response(serializer.data)

        else:

            return Response(serializer.errors)

    def delete(self, request, pk):

        try:

            plateform = StreamPlateform.objects.get(pk=pk)

        except:

            return Response(status=status.HTTP\_404\_NOT\_FOUND)

        plateform.delete()

        data = {

            'detail':"Deleted successfully",

        }

        return Response(data, status=status.HTTP\_204\_NO\_CONTENT)

urls.py

First import the views.

    # Stream Plateform Urls

path('stream/', StreamPlateformList.as\_view(), name='streamPlateformList'),

path('stream/<int:pk>/', StreamPlateformDetail.as\_view(), name='streamPlateformDetail'),

**Django Permissions.**

<https://www.webforefront.com/django/setuprelationshipsdjangomodels.html>

There are 3 types of relationship

1. One to one
2. One to many – many to one
3. Many to may

One movie can have only one platform but one platform can have many movies.

Delete all the entries so we

Can’t mess with default values.

Models.py

Here I connect StreamPlateform to Watchlist. It means One Movie have One platform.

class WatchList(models.Model):

    title = models.CharField(max\_length=50)

    storyline = models.CharField(max\_length=200)

    platform = models.ForeignKey(StreamPlateform, on\_delete=models.CASCADE, related\_name='watchlist')

    active = models.BooleanField(default=False)

    created\_date = models.DateTimeField(auto\_now\_add=True)

Now you can add the streaming platform. And when you will add new Movie then you can see that there is a platform list.

But we don’t have relationship in serializer so we have to nested serializer so we can see on Browser.

Now I want to show all the movies That Netflix (stream Platform) have.

I there is nor connection between WatchListSerializer and StreamPlatformSerializer.

**Serializers.py**

class WatchListSerializer(serializers.ModelSerializer):

    class Meta:

        model = WatchList

        fields = "\_\_all\_\_"

class StreamPlateformSerializer(serializers.ModelSerializer):

    watchlist = WatchListSerializer(many=True, read\_only=True)

    class Meta:

        model = StreamPlateform

        fields = "\_\_all\_\_"

Here I connect platform to watchlist

**This watchlist name comes from related name which we defined in models.py**

Now you can see that movies of particular platform.

<http://127.0.0.1:8000/watch/stream/>

**Review Model**

from django.core.validators import MaxValueValidator, MinValueValidator

class Review(models.Model):

    rating = models.PositiveIntegerField(validators=[MinValueValidator(1), MaxValueValidator(5)])

    description = models.CharField(max\_length=200, null=True)

    watchlist = models.ForeignKey(WatchList, on\_delete=models.CASCADE, related\_name='reviews')

    active = models.BooleanField(default=True)

    created\_date = models.DateTimeField(auto\_now\_add=True)

    updated\_date = models.DateTimeField(auto\_now=True)

    def \_\_str\_\_(self):

        return str(self.rating) + " | " + self.watchlist.title

Here we first import Min and MaxValueValidator bcz we want to give rating between 1 – 5.

One movie can have multiple reviews.

def \_\_str\_\_(self):

        return str(self.rating) + " | " + self.watchlist.title

Here we use watchlist.tilte watchlist have all the fields.

**Serializers.py**

Import Review model

class ReviewSerializer(serializers.ModelSerializer):

    class Meta:

        model = Review

        fields = "\_\_all\_\_"

Now I want get all the review of a movie.

Now we will use generic view or mixing for the review

**Generic View**

<https://www.django-rest-framework.org/tutorial/3-class-based-views/#using-mixins>

Using generic view along with mixings

1. Import the Review model and serializer

class ReviewList(mixins.ListModelMixin,

mixins.CreateModelMixin,

generics.GenericAPIView):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

    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 ReviewDetail(mixins.RetrieveModelMixin, generics.GenericAPIView):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

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

        return self.retrieve(request, \*args, \*\*kwargs)

    # Reviews URLs

    path('reviews/',ReviewList.as\_view(), name="reviews"),

    path('reviews/<int:pk>/', ReviewDetail.as\_view(), name="reviewDetail"),

Here you can see I use the list in get method bcz I want List as a output but If I want only 1 record then I have to use **retrieve method** in get method as I used in Detail view.

<http://127.0.0.1:8000/watch/>

Now run this url you will get all the reviews for a particular Movie.

But here One problem One user can give multiple Reviews which is not good.